GPU TECHNIQUES APPLIED TO EULER FLOW SIMULATIONS AND COMPARISON TO CPU PERFORMANCE

BY BLAKE KOOP

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Mechanical and Aerospace Engineering

Written under the direction of

Doyle D. Knight

and approved by

_____

_____

_____

New Brunswick, New Jersey

May, 2014

ABSTRACT OF THE THESIS

GPU Techniques Applied to Euler Flow Simulations and Comparison to CPU Performance

by BLAKE KOOP

Thesis Director: Doyle D. Knight

With the decrease in cost of computing, and the increasingly friendly programming environments, the demand for computer generated models of real world problems has surged. Each generation of computer hardware becomes marginally faster than its predecessor, allowing for decreases in required computation time. However, the progression is slowing and will soon reach a barrier as lithography reaches its natural limits. General Purpose Graphics Processing Unit (GPGPU) programming, rather than traditional programming written for Central Processing Unit (CPU) architectures may be a viable way for computational scientists to continue to realize wall clock time reductions at a Moore's Law pace. If a code can be modified to take advantage of the Single-Input-Multiple-Data (SIMD) architecture of Graphics Processing Units (GPUs), it may be possible to gain the functionality of hundreds or thousands of cores available on a GPU card. This paper details the investigation of a specific compressible flow simulation and its functionality in both CPU and GPU programming schemes. The flow is governed by the unsteady Euler flow equations and it is checked for validity against the known solution in all three directions. It is then run over varying grid sizes using both the CPU and GPU programming schemes to evaluate wall clock time reductions.

Acknowledgements

I would first like to thank the University and specifically the School of Engineering. Throughout this process, I have been enabled with resources without which none of this would have been possible. I would also like to specially thank my advisor, Dr. Doyle Knight. I have worked with Dr. Knight in several capacities over the past few years, and have relied on his guidance, advice, and patience to complete this work. The mathematical development of the strategies employed in this simulation are taken from his previous work. To everyone else who has spared their valuable time on my behalf, I say thank you as well. I have learned a tremendous amount, and am grateful to all of you who have enabled me. Additional thanks are extended to the New Jersey Space Grant Consortium (NJSGC) and Everson Tesla, Inc. for providing funding for the work which led to this document.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 History of Processing Capabilities

The development of the microprocessor marked a turning point in the world of digital computing. The reduction in production cost and increase in computation speed allowed a large number of applications to make use of computing power, including applications in the scientific computing community.

One of the first commercially available microprocessors was the Intel 8088, which was released in 1979 for $124.80, (which according the Bureau of Labor Statistics, is equivalent to $402.04 in 2013). According to Intel, the 8088 contained a total of 29,000 transistors, and boasted a maximum clock rate of 5 MHz to 10 MHz, and could store up to 1 MB in address space [4]. Given that the logic units required to perform basic arithmetic operations required multiple clock cycles, a 5 MHz processor had an absolute performance of approximately 10^6 (1 MHz) operations per second. To put this number in perspective, in the world of scientific computing, it is not hard to imagine a simulation involving 10^4 finite volume elements, each requiring 10^2 operations per time step over 10^3 time steps. Even such a simple simulation would require 10^9 operations and require 1000 seconds to run.

The popularity of the Intel 8088 was driven by the popularity of the IBM PC, which was one of several personal computing options available at the time. The market for personal computing was racing to more functionality and lower costs, which requried smaller, denser processors. The more efficient processors increased user productivity and further fueled the

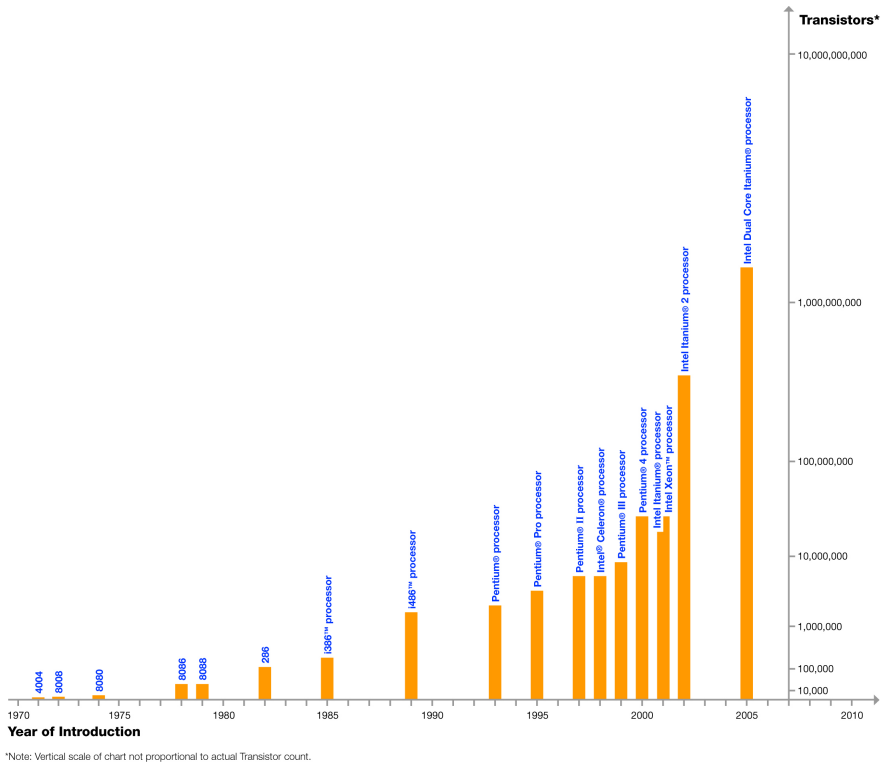


Figure 1.1: Moore's Law Poster Microprocessor Chart [1]

demand on the personal computing market. The result was an exponential increase in the operations per second capabilities of the processors, and more specifically an exponential increase in the number of transistors per unit area on the semiconductor surfaces being used in the multiprocessors. This trend was famously predicted by Gordon Moore, co-founder of Intel, in 1965 and was coined "Moore's Law".

The scientific computing community, although not a primary driver of the progress in commercially available technology, stood to benefit from the improvements. Faster processors meant less wall clock time required by simulations which increased the number of applications where computer simulations and models were appropriate. The transistor count on a processor is a generally accepted way to approximate processor capability. The trend in figure 1.1 shows the transistor count in Intel products through 2004.

The driving force behind the exponential increase in transistor count per processor was almost completely attributable to the exponential decrease in the size of each transistor.

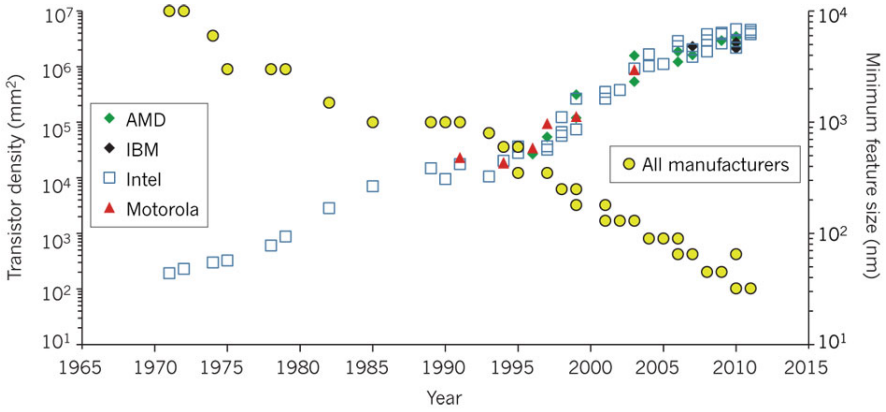


Figure 1.2: Various Manufacturers Comparison of Transistor Density versus Minimum Feature Size. Originally printed by Pierre, Collinge, and Collinge in "Multigate transistors as the future of classical metaloxidesemiconductor field-effect transistors" 17 November 2011.

Figure 1.2 shows the transition from the technology of the early 1970's to the technology of today. Transistors have reduced in size from tens of *micrometers* to tens of *nanometers*. Behind the size reduction was similarly scaled improvement in the manufacturing process, particularly in the most common process: photo-lithography.

The photo-lithographic process used in the semiconductor industry to make microchips applies the same basic techniques as lithography, which has been used in various forms of printing for hundreds of years. In the modern version, wafers are coated in a light-sensitive material, the photo-resist, then exposed to a particular wavelength of light which has been passed through a designed mask and usually optically reduced. The result is an image in the photo-resist on the wafer. Although additional steps are required to complete the process, one can now begin to quantify the size of the features being printed. Resolution can be expressed as: [5]

$$R = k_1 \frac{\lambda}{NA} \tag{1.1}$$

where R is feature resolution, also referred to as the critical dimension, λ is the wavelength

of light used for exposure, k_1 is a constant reflective of the manufacturing process with a range of $.3 < k_1 < 1$, and NA is the numerical aperture of the optical system involved in the image reduction. The combination of improvements in these three factors along with improving materials has pushed the minimum feature size on integrated circuits from the micrometers of the Intel 8088 to the nanometers of today's processors.

Wavelength. The early 1980s lithography techniques made use of Mercury lamps which produced spectral lines at several UV wavelengths, the most commonly used being the 436 nm line. Today's cutting edge manufacturing tools make use of the 193 nm ArF line and the 158 nm F_2 line. Extreme Ultraviolet lithography is looked to by some as the next big step improvement, with capabilities postulated to sub 10 nm patterning.

Optics. The improvements in the optical systems have taken the numerical aperture values from approximately 0.30 to approximately 0.85. Systems with even higher NA values have been researched, but the loss of image focus depth that comes with increasing NA values has proved to be a difficult hurdle.

Process. The k_1 scaling factor tends to be a good measure of the complexity involved in printing a particular feature. There is somewhat of a natural lower limit to this factor at $k_1 = .25$ due the diffraction process limiting the full pitch to $.5\frac{\lambda}{NA}$ for line gratings [6]. Attempts have been made to break through this barrier, but for the near future, given the natural boundaries in both this scaling factor and the numerical aperture factor, it will be reasonable to approximate the minimum printable feature size as $\frac{1}{4}$ of the exposure wavelength.

Lithography has survived predictions of its slowed growth or even demise before, and perhaps it will do so again, but even the biggest advocate will admit that the process has a fundamental limit in minimum feature size as the printing capabilities approach the atomic level. Keeping up with Moore's Law has been an impressive feat, but the industry will most likely need to move away from transistor density increases as a primary source of performance increases. There is evidence of this switch in production already. In 2002, Intel produced a version of the Intel Pentium 4 processor which contained 2 cores. In 2010, the 6 core version of the Core i7 processor was released. It is now difficult to find a desktop

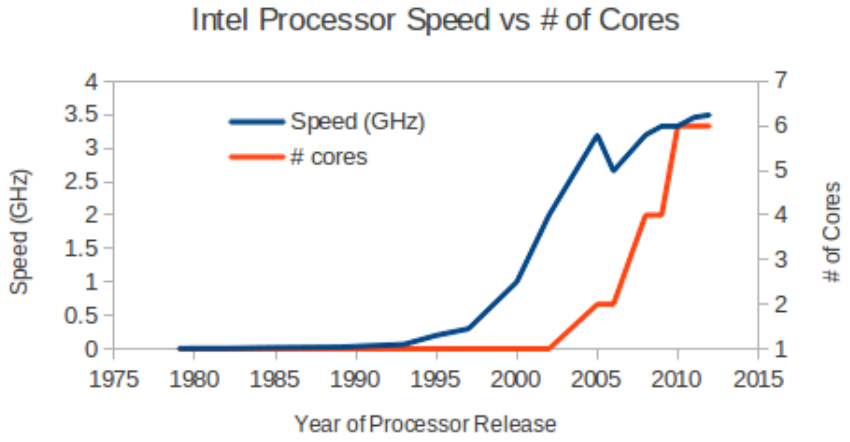


Figure 1.3: Intel Processor Speed Versus Number of Cores

computer using a processor with only a single core. Figure 1.3 is simply a compilation of processor data made available by Intel. The simultaneous increase in core count and leveling of the speed per core is apparent.

This was perhaps not an unforeseeable transition. For years, the performance improvements in super computers paralleled that of the commercial computer, not by increasing the functionality of a single core, but by using more cores simultaneously. As the commercial computing hardware approaches the natural performance boundaries imposed by the fabrication process, and the demand for performance improvements remains constant, the industry has been forced to providing multiple cores rather than higher performance single cores [7].

1.2 GPU versus CPU

In the 1980s and 1990s, the industry pushed towards a more user friendly graphics driven computer experience. Complex graphics, which were moving into 2D and eventually into 3D, required large amounts of computing power. Early graphics based system used the existing CPUs for all functions. As the graphics increased in complexity, performance was

threatened and the market for a processing system capable of executing graphical based applications without compromising performance for picture was born.

The graphics hardware, driven by demanding markets in video media based entertainment and visual based operating systems, evolved quickly. In 1995, NVIDIA produced the NV1 graphics processor. The NV1 was capable of 12 million operations per second (MOPS) and communicated with the CPU across a 0.6 GB/s memory bandwidth. In 1999, NVIDIA released the GeForce 256, which was capable of 480 MOPS and communicated with the CPU with a 2.7 GB/s memory bandwidth. The improvement on an operations per second basis was a factor of 40 in 5 years. This was quite an improvement. Operating systems also adapted to the improvements. The increasingly popular Microsoft Windows OS combined with DirectX, a programming environment for game and visual media developers, allowed for widespread programming of advanced 3D graphics. Personal computers running graphical based software continued to permeate both the professional and non-professional computer markets. With each generation, the improved graphical user interfaces made personal computers more accessible to users of all abilities and needs, expanding the potential market size. Increasingly attractive graphics also pushed the gaming industry revenues to new heights. The huge demand allowed more investment in the hardware, and consequently the technology grew rapidly. In the late 1990s, leading gaming companies SEGA, Playstation and Nintendo all released games with 3D visual effects. For the first time, games were able to offer a first person experience for the user. Each generation of a game attempted to achieve a more realistic, or at least detailed and interactive graphical environment. With personal computing and gaming markets driving demand for increasingly advanced graphics, the graphics hardware advancements scaled with the advancements in the general purpose microprocessing hardware described in the previous section. Figure 1.4 details the growth, measured in floating point operations per second (FLOPS), which continues through the present. The net result is commercially available hardware with impressive computational capabilities, albeit not the same architecture as traditionally used hardware.

Early graphics hardware, eventually coined Graphics Processing Units (GPUs) by NVIDIA

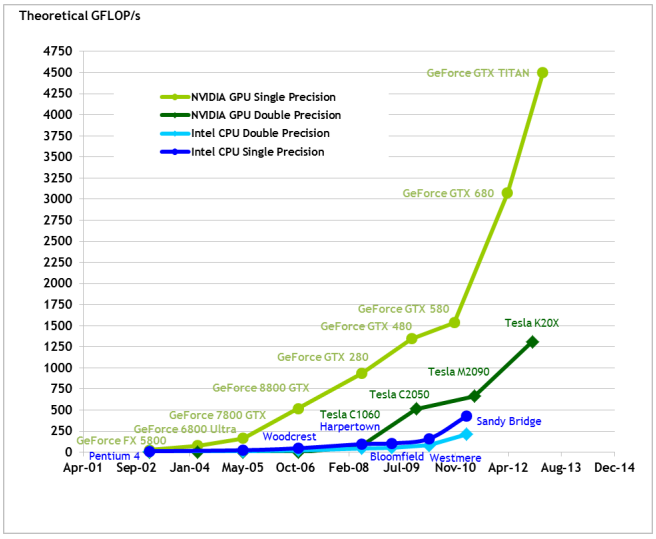


Figure 1.4: Floating Point Operations per Second for Various NVIDIA Hardware [2]

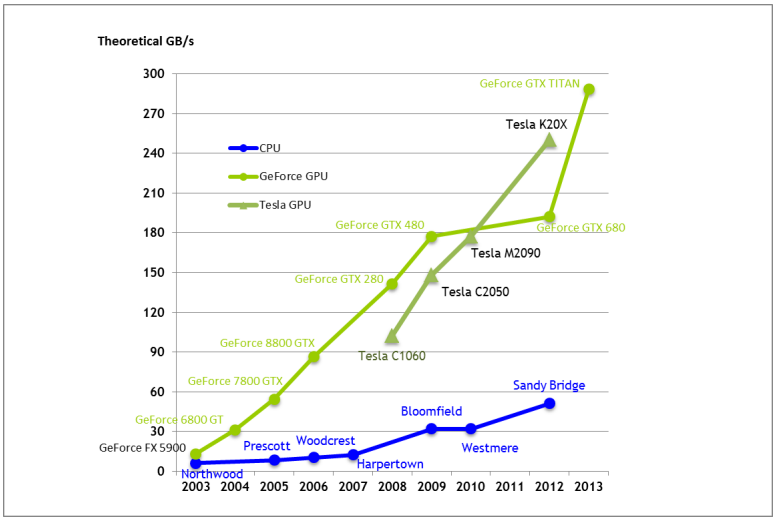


Figure 1.5: Bandwidth Capabilities for Various NVIDIA Hardware [2]

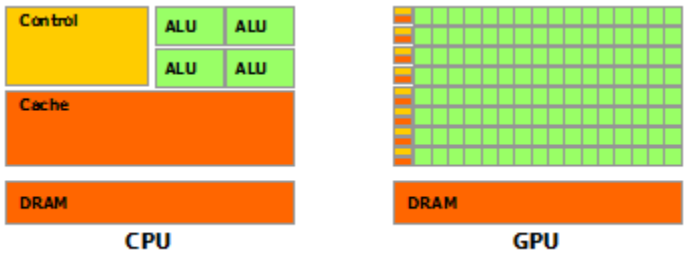


Figure 1.6: Schematic Comparing CPU to GPU Data Processing [2]

in 1999 with the release of the GeForce 256, were not programmable enough to be taken advantage of in the world of scientific computing. The single input multiple data (SIMD) architecture of the GPUs required scientists to recreate problems as graphical analogies, and provided very limited debugging capabilities. Commercial processors packaged in commercial and professional computers were generally of the multiple input multiple data (MIMD) variety, where each processor fetches its own instructions and operates on its own data. The alternative architecture on GPUs, SIMD, makes use of multiple cores operating on different data streams, but is limited in that each core executes the same set of instructions. The programming limitations notwithstanding, it is not difficult to imagine problems in general purpose computing that require repeating the same instruction set over multiple data streams. Rather than repeating instruction set 'A' in serial over every element in an array of length 'N' using a single computational device, a problem could be approached by separating the data and computing instruction set 'A' on 'N' unique computational devices simultaneously working on separate data sets. If the advantages of *simultaneously* computing the instructions outweigh the cost of splitting up the data to each computational device and recombining the result, then a net advantage has been achieved. Figure 1.6 below illustrates the difference in architecture. However, without a reasonable programming interface, the power of the SIMD Architecture GPUs remained largely inaccessible.

NVIDIA was the first to provide a means of easy access to the GPU for general programming purposes. This access was provided in 2006 with the release of the CUDA Architecture. The modifications to the hardware involved converting the microprocessor features on the

GPU from strictly graphical functionality to features that could be programmed to execute a variety of operations useful in general purpose computing. The CUDA Architecture was first released with the NVIDIA GeForce 8800 GTX. CUDA provided a language for accessing this architecture, CUDA C, which was essentially industry standard C with some additions for accessing the GPU hardware [7]. With CUDA C, anyone familiar with the C programming language could receive a small amount of training and be competent enough to develop programs capable of utilizing the hundreds or thousands of cores available on GPU hardware. Although the scientific computing community had little to do with the development of the technology in its early development, GPU based High Powered Computing (HPC) systems are now available and affordable to many computational science budgets.

Specifically in computational fluid dynamics, GPU applications are being tested by a number of sources. A group out of the Naval Research Laboratory (NRL), Patnaik, Corrigan, Obenschain, Schwer, and Fyfe released a paper investigating efficient CPU-GPU cluster structures using a jet engine noise reduction (JENRE) code [8]. Another group out of the NRL is using GPU hardware in an attempt to speed up detection schemes for aircraft with ever decreasing radar cross sections [9]. Yet another group working out of the CFD Center at George Mason University investigate the applications of GPU clusters in blast simulations [10]. All of these groups are testing legacy codes simulating specific conditions against a GPU compatible counterpart. The goal is to find a more economical, and relatively user friendly way to achieve higher compute performance. This research attempts to do a similar code restructure and comparison.

In a performance report released by NVIDIA detailing specific hardware, the NVIDIA Tesla K20/K20X is listed to be capable of 1.31 TFLOPS (Tera-floating point operations per second) of double precision performance. This card is for sale on Amazon.com for less than $3000. For comparison, the report chooses Intel's Xeon E5-2687W 8 core processor which lists a peak performance of 0.17 TFLOPS DP and can be purchased on Amazon for just under $2000. It is clear that the GPU technology has the potential to achieve

more on a performance per dollar basis, given the right application. This paper details the investigation of a specific compressible flow simulation and its functionality in the NVIDIA CUDA Architecture.

Chapter 2

Physical Problem

This chapter introduces the equations governing the physical system being approximated and presents them in various forms which will be useful in subsequent sections. The initial and boundary conditions are also presented.

2.1 Governing Equations

A one dimensional, unsteady, inviscid, compressible flow was simulated. Such a flow (1-D Euler flow) can be described using conservation laws in the differential form, as shown below [3].

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} = 0 \tag{2.1}$$

$$\frac{\partial \rho u}{\partial t} + \frac{\partial \rho u^2}{\partial x} = -\frac{\partial p}{\partial x} \tag{2.2}$$

$$\frac{\partial \rho e}{\partial t} + \frac{\partial (\rho e + p)u}{\partial x} = 0 \tag{2.3}$$

where ρ is the density, u is the velocity, p is the pressure, e is the total energy defined by $e = e_i + \frac{1}{2}u^2$, and x and t are the spatial and temporal coordinates, respectively. The flow is assumed to be both calorically and thermally perfect, i.e., the internal energy is defined as

$$e_i = c_v T \tag{2.4}$$

and

$$p = \rho RT \tag{2.5}$$

where c_p is the specific heat at constant pressure. T is the static temperature. Given the static enthalpy

$$h = e_i + \frac{p}{\rho} \tag{2.6}$$

combined with the ideal gas assumption (2.5) and Mayer's Relation of specific heats:

$$R = c_p - c_v \tag{2.7}$$

The static enthalpy volume can be shown to be

$$h = c_p T \tag{2.8}$$

The speed of sound is defined by

$$a = \sqrt{\frac{\partial p}{\partial \rho}\Big|_s} \tag{2.9}$$

Again using ideal gas the speed of sound can be shown as

$$a = \sqrt{\gamma RT} \tag{2.10}$$

The conservation of energy shown in (2.3) can also be expressed in terms of entropy,

$$\frac{\partial s}{\partial t} + u\frac{\partial s}{\partial x} = 0 \tag{2.11}$$

2.2 Vector Notation

The conservation equations can also be expressed in compact vector notation.

$$\frac{\partial \mathcal{Q}}{\partial t} + \frac{\partial \mathcal{F}}{\partial x} = 0 \tag{2.12}$$

where

$$\mathcal{Q} = \begin{Bmatrix} \rho \\ \rho u \\ \rho e \end{Bmatrix} \quad \text{and} \quad \mathcal{F} = \begin{Bmatrix} \rho u \\ \rho u^2 + p \\ \rho e u + p u \end{Bmatrix}$$

The pressure, p, can be evaluated as

$$p = (\gamma - 1)\left(\rho e - \frac{1}{2}\rho u^2\right) \tag{2.13}$$

The flux, $\mathcal{F}$, can be expressed in terms of the dependent variable vector, $\mathcal{Q}$, with some algebraic substitutions.

$$\mathcal{F} = \begin{Bmatrix} \mathcal{Q}_2 \\ \frac{\mathcal{Q}_2^2}{\mathcal{Q}_1} + (\gamma - 1)\left(\mathcal{Q}_3 - \frac{1}{2}\frac{\mathcal{Q}_2^2}{\mathcal{Q}_1}\right) \\ \frac{\mathcal{Q}_2\mathcal{Q}_3}{\mathcal{Q}_1} + (\gamma - 1)\frac{\mathcal{Q}_2}{\mathcal{Q}_1}\left(\mathcal{Q}_3 - \frac{1}{2}\frac{\mathcal{Q}_2^2}{\mathcal{Q}_1}\right) \end{Bmatrix} \tag{2.14}$$

Taking advantage of this form of the flux vector, the conservative vector notation can be written as

$$\frac{\partial \mathcal{Q}}{\partial t} + \mathcal{A}\frac{\partial \mathcal{Q}}{\partial x} = 0 \tag{2.15}$$

where $\mathcal{A}$ is the Jacobian matrix

$$\mathcal{A} = \frac{\partial \mathcal{F}}{\partial \mathcal{Q}} \tag{2.16}$$

which can be determined by differentiating term by term to show

$$\mathcal{A} = \begin{pmatrix} 0 & 1 & 0 \\ \frac{\gamma-3}{2}\left(\frac{\mathcal{Q}_2}{\mathcal{Q}_1}\right)^2 & (3-\gamma)\frac{\mathcal{Q}_2}{\mathcal{Q}_1} & \gamma - 1 \\ -\gamma\frac{\mathcal{Q}_2\mathcal{Q}_3}{\mathcal{Q}_1^2} + (\gamma-1)\left(\frac{\mathcal{Q}_2}{\mathcal{Q}_1}\right)^3 & \gamma\frac{\mathcal{Q}_3}{\mathcal{Q}_1} - \frac{3}{2}(\gamma-1)\left(\frac{\mathcal{Q}_2}{\mathcal{Q}_1}\right)^2 & \gamma\frac{\mathcal{Q}_2}{\mathcal{Q}_1} \end{pmatrix} \tag{2.17}$$

The definition of the vector $\mathcal{Q}$ can be used to show $\mathcal{A}$ in terms of the flow variables,

$$\mathcal{A} = \begin{pmatrix} 0 & 1 & 0 \\ \frac{\gamma-3}{2}u^2 & (3-\gamma)u & \gamma-1 \\ -\gamma eu + (\gamma-1)u^3 & \gamma e - \frac{3}{2}(\gamma-1)u^2 & \gamma u \end{pmatrix} \tag{2.18}$$

This form of the Euler equations, with the Jacobian matrix $\mathcal{A}$ mapping the vector $\mathcal{Q}$ to the vector $\mathcal{F}$ will be further developed in chapter 4 for use in the determination of the flux vector.

2.3 Isentropic Relations

The isentropic relations will also be helpful. Starting with the change in entropy per unit mass,

$$Tds = de_i - \frac{p}{\rho^2}d\rho \tag{2.19}$$

and using (2.4) and (2.5) for a calorically and thermally perfect gas,

$$ds = \frac{c_v}{T}dT - \frac{R}{\rho}d\rho \tag{2.20}$$

which can be integrated to show

$$s - s_0 = c_v \ln\left(\frac{T}{T_0}\right) - R\ln\left(\frac{\rho}{\rho_0}\right) \tag{2.21}$$

from which two alternate forms can be found by substituting (2.5) for the temperature term and the density term in turn.

$$s - s_0 = c_p \ln\left(\frac{T}{T_0}\right) - R\ln\left(\frac{p}{p_0}\right) \tag{2.22}$$

$$s - s_0 = c_v \ln\left(\frac{p}{p_0}\right) - c_p\ln\left(\frac{\rho}{\rho_1}\right) \tag{2.23}$$

So for any isentropic flow beginning in state "0", two pressure relations can be shown:

$$c_p \ln\left(\frac{T}{T_0}\right) = R \ln\left(\frac{p}{p_0}\right) \tag{2.24}$$

$$\ln\left(\frac{T}{T_0}\right)^{\frac{c_p}{R}} = \ln\left(\frac{p}{p_0}\right) \tag{2.25}$$

If we define the ratio of specific heats, $\frac{c_p}{c_v} = \gamma$, we find the first isentropic pressure relation,

$$\frac{p}{p_0} = \left(\frac{T}{T_0}\right)^{\frac{\gamma}{\gamma-1}} \tag{2.26}$$

And similar algebra yields the second,

$$\frac{p}{p_0} = \left(\frac{\rho}{\rho_0}\right)^{\gamma} \tag{2.27}$$

2.4 Convective Derivative Expression

The conservation equations can also be expressed in terms of convective derivatives. The conservation of mass in convective form is the same as shown in (2.1) and is used below to transform the momentum and energy equations into their convective forms. Expanding (2.2), and eliminating the conservation of mass terms,

$$\cancel{u\frac{\partial \rho}{\partial t}} + \rho\frac{\partial u}{\partial t} + \rho u\frac{\partial u}{\partial x} + \cancel{u\frac{\partial \rho u}{\partial x}} = -\frac{\partial p}{\partial x} \tag{2.28}$$

Which yields the conservation of momentum in terms of the convective derivative of u,

$$\rho\Big(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x}\Big) = -\frac{\partial p}{\partial x} \tag{2.29}$$

Now expanding the energy conservation equation (2.3),

$$\rho\frac{\partial e}{\partial t} + e\frac{\partial \rho}{\partial t} + (\rho e + p)\frac{\partial u}{\partial x} + u\Big(\frac{\partial(\rho e + p)}{\partial x}\Big) = 0 \tag{2.30}$$

and taking advantage of the linearity of differentiation with one additional product rule expansion,

$$\rho\frac{\partial e}{\partial t} + e\frac{\partial \rho}{\partial t} + (\rho e + p)\frac{\partial u}{\partial x} + u\Big(\frac{\partial \rho e}{\partial x} + \frac{\partial p}{\partial x}\Big) = 0 \tag{2.31}$$

$$\rho\frac{\partial e}{\partial t} + e\frac{\partial \rho}{\partial t} + (\rho e + p)\frac{\partial u}{\partial x} + u\Big(e\frac{\partial \rho}{\partial x} + \rho\frac{\partial e}{\partial x} + \frac{\partial p}{\partial x}\Big) = 0 \tag{2.32}$$

which can be condensed to

$$\rho\Big(\frac{\partial e}{\partial t} + u\frac{\partial e}{\partial x}\Big) + e\Big(\cancel{\frac{\partial \rho}{\partial t} + u\frac{\partial \rho}{\partial x} + \rho\frac{\partial u}{\partial x}}\Big)^{0} = -\frac{\partial pu}{\partial x} \tag{2.33}$$

After eliminating the conservation of mass, we are left the conservation of energy in terms of the convective derivative of e,

$$\rho\Big(\frac{\partial e}{\partial t} + u\frac{\partial e}{\partial x}\Big) = -\frac{\partial pu}{\partial x} \tag{2.34}$$

Equations (2.28) and (2.33) are the conservation of momentum and energy expressed using the convective derivatives. These expressions will be used later in the development of the analytical solution.

2.5 Characteristic Form

The governing equations can also be cast in characteristic form. This form is useful in describing wavelike behavior, and is apt to evaluate the system from the perspective of a particle who is traveling with the wave using a Lagrangian coordinate system.

For a particle exhibiting some scalar quantity $f(x,t)$, as it travels along some path, $x(t)$, for a period of time, the time rate of change of f along that particular path can be described by the convective derivative of that quantity.

$$\frac{df}{dt} = \frac{d}{dt}f(x,t) = \frac{\partial f}{\partial x}\frac{dx(t)}{dt} + \frac{\partial f}{\partial t} \tag{2.35}$$

Previously shown forms of the conservation equations can be manipulated to more closely represent equation (2.35). Starting with the conservation equations in their convective forms,

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u)}{\partial x} = 0 \tag{2.36}$$

$$\rho\Big(\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x}\Big) = -\frac{\partial p}{\partial x} \tag{2.37}$$

$$\rho\Big(\frac{\partial e}{\partial t} + u\frac{\partial e}{\partial x}\Big) = -\frac{\partial pu}{\partial x} \tag{2.38}$$

but replacing the energy equation with entropy,

$$\rho\Big(\frac{\partial s}{\partial t} + u\frac{\partial s}{\partial x}\Big) = 0 \tag{2.39}$$

It is clear that the entropy equation is already an analog of equation (2.35). Using the notation of $\frac{d}{dt}$ to represent the convective derivative of a quantity, and assuming nonzero densities, it can be said by inspection,

$$\frac{ds}{dt} = 0 \qquad on \qquad \frac{dx}{dt} = u \tag{2.40}$$

which is a mathematical way of saying that entropy is conserved when following a fluid particle along some path, $x(t)$. This is a transformed conservation equation which applies along the *characteristic* $\frac{dx}{dt} = u(x(t), t)$. With some additional manipulation and the incorporation of the mass and energy equations, additional characteristics can be found. Starting with equation (2.22),

$$s - s_0 = c_v \ln\Big(\frac{p}{p_0}\Big) - c_p \ln\Big(\frac{\rho}{\rho_0}\Big) \tag{2.41}$$

And if we consolidate constant terms,

$$s - s_0 = c_v \ln p - c_p \ln \rho + constant \tag{2.42}$$

Then taking the convective derivative of both sides, and knowing that entropy is conserved along the characteristics,

$$\frac{c_v}{p}\left[\frac{\partial p}{\partial t} + u\frac{\partial p}{\partial x}\right] - \frac{c_p}{\rho}\left[\frac{\partial \rho}{\partial t} + u\frac{\partial \rho}{\partial x}\right] = 0 \tag{2.43}$$

and one final manipulation,

$$\frac{\partial p}{\partial t} + u\frac{\partial p}{\partial x} - \frac{p\gamma}{\rho}\left[\frac{\partial \rho}{\partial t} + u\frac{\partial \rho}{\partial x}\right] = 0 \tag{2.44}$$

Now, The conservation of mass can be expanded as

$$\frac{\partial \rho}{\partial t} + \rho\frac{\partial u}{\partial x} + u\frac{\partial \rho}{\partial x} = 0 \tag{2.45}$$

and substituted into the third term of (2.44) to yield

$$\frac{\partial p}{\partial t} + u\frac{\partial p}{\partial x} + p\gamma\frac{\partial u}{\partial x} = 0 \tag{2.46}$$

and given that $a^2 = \frac{p\gamma}{\rho}$,

$$\frac{\partial p}{\partial t} + u\frac{\partial p}{\partial x} + a^2\rho\frac{\partial u}{\partial x} = 0 \tag{2.47}$$

This is a general statement of conservation following a fluid particle. If we then take the conservation of momentum in convective form, (2.37), multiply all terms by a and add (2.47),

$$\left[\frac{\partial p}{\partial t} + (u+a)\frac{\partial p}{\partial x}\right] + \rho a\left[\frac{\partial u}{\partial t} + (u+a)\frac{\partial u}{\partial x}\right] = 0 \tag{2.48}$$

And by comparison to the definition of the convective derivative this is the same as

$$\frac{1}{\rho a}\frac{dp}{dt} + \frac{du}{dt} = 0 \quad on \quad \frac{dx}{dt} = u + a \tag{2.49}$$

We can use the isentropic relations, (2.19) and (2.20), along with (2.10) to show that both ρ and a are functions of p alone. Multiplying by a differential increment in time,

$$\left[\frac{dp}{\rho a} + du\right] = 0 \quad on \quad \frac{dx}{dt} = u + a \tag{2.50}$$

and then taking the indefinite integral

$$\int \frac{dp}{\rho a} + \int du = constant \quad on \quad \frac{dx}{dt} = u + a \tag{2.51}$$

and differentiating in time

$$\frac{d}{dt}\left[\int \frac{dp}{\rho a} + u\right] = 0 \quad on \quad \frac{dx}{dt} = u + a \tag{2.52}$$

which can be further simplified using the isentropic pressure relations to

$$\frac{d}{dt}\left[\frac{2}{\gamma - 1}a + u\right] = 0 \quad on \quad \frac{dx}{dt} = u + a \tag{2.53}$$

Using the same process, but multiplying the conservation of momentum by $(-a)$ rather than a, it is possible to show

$$\frac{d}{dt}\left[\frac{2}{\gamma - 1}a - u\right] = 0 \quad on \quad \frac{dx}{dt} = u - a \tag{2.54}$$

Observing (2.40), (2.53), and (2.54), the conservation equations have been recast in a form showing a constant convective derivative along three characteristics. Here they are repeated:

$$\frac{d}{dt}\left[\frac{2}{\gamma - 1}a + u\right] = 0 \quad on \quad \frac{dx}{dt} = u + a \tag{2.55}$$

$$\frac{d}{dt}\left[\frac{2}{\gamma - 1}a - u\right] = 0 \quad on \quad \frac{dx}{dt} = u - a \tag{2.56}$$

$$\frac{ds}{dt} = 0 \qquad on \qquad \frac{dx}{dt} = u \tag{2.57}$$

Where u and a are functions of x and t. The above equations indicate that there are certain reference frames from which it will appear that the velocity profile will not vary in time. The reference frames, or characteristic curves, are described by the $\frac{dx}{dt}$ requirements for each invariant.

2.6 Boundary and Initial Conditions

For this specific investigation, the flow was given a sinusoidal initial velocity profile described by:

$$u(x,0) = \epsilon a_0 sin(\kappa x) \tag{2.58}$$

where ϵ is a dimensionless constant, a_0 is the initial speed of sound at $x = 0$ and κ is a dimensionless constant chosen to be

$$\kappa = \frac{2\pi}{\lambda} \tag{2.59}$$

where λ is the wavelength of the initial condition. The boundary conditions in the flow direction were oscillating, and when the three-dimensional case was considered, the boundaries conditions normal to the flow direction were symmetrical.

The flow is simulated from $t = 0$ until the time of shock formation for comparison to the analytical solution developed in the next chapter.

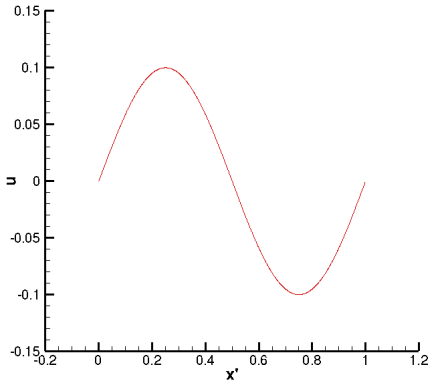


Figure 2.1: Initial velocity as a function of nondimensional length

Chapter 3

Analytical Solution

3.1 Solution Attempt Using the Method of Characteristics

Using the initial conditions described in the previous section, and the method of characteristics described in Section 2.5, it appears that a solution can be found directly. Repeating the unsteady, one-dimensional, Euler equations in characteristic form:

$$\frac{d}{dt}\left[\frac{2}{\gamma-1}a+u\right]=0 \quad on \quad \frac{dx}{dt}=u+a \tag{3.1}$$

$$\frac{d}{dt}\left[\frac{2}{\gamma-1}a-u\right]=0 \quad on \quad \frac{dx}{dt}=u-a \tag{3.2}$$

$$\frac{ds}{dt}=0 \qquad on \qquad \frac{dx}{dt}=u \tag{3.3}$$

and generalizing the initial conditions to

$$u(x,0)=u_0(x) \tag{3.4}$$

$$a(x,0)=a_0(x) \tag{3.5}$$

a point p can be imagined which is an infinitesimal time increment, dt, after the initial conditions, at which the characteristics $u+a$, $u-a$, and u intersect. Given that it is not possible for the characteristics to be parallel, [1] this intersection is assured to exist. Evaluating equations (3.1) and (3.2),

[1] For parallel characteristics, the speed of sound would be required to be $a=0$. This would require a temperature equal to absolute zero.

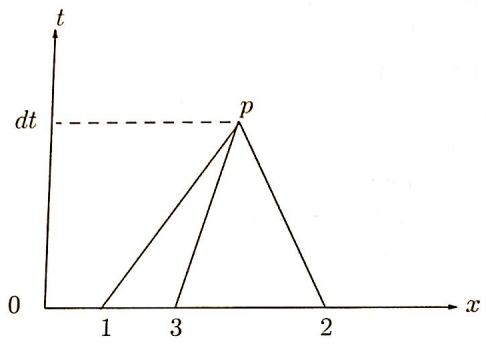


Figure 3.1: Intersecting characteristics drawn for $a > u > 0$ [3].

$$\frac{2}{\gamma-1}a_q + u_q = \frac{2}{\gamma-1}a_1 + u_1 \tag{3.6}$$

$$\frac{2}{\gamma-1}a_q - u_q = \frac{2}{\gamma-1}a_2 - u_2 \tag{3.7}$$

which can be rearranged for u_p and a_p as functions of the given initial conditions at points 1, 2, and 3.

$$a_q = \frac{\gamma-1}{2}(u_1 - u_q) + a_1 \tag{3.8}$$

$$u_q = \frac{2}{\gamma-1}(a_q - a_2) + u_2 \tag{3.9}$$

Substituting (3.9) into (3.8),

$$a_q = \frac{1}{2}\left[(a_1 + a_2) + \frac{\gamma-1}{2}(u_1 - u_2)\right] \tag{3.10}$$

and then solving (3.8),

$$u_q = \frac{1}{\gamma-1}(a_1 - a_2) + \frac{1}{2}(u_1 + u_2) \tag{3.11}$$

and given that our flow is isentropic along the characteristics, we have the isentropic pressure relations to determine the remaining state variables pressure, density, and temperature.

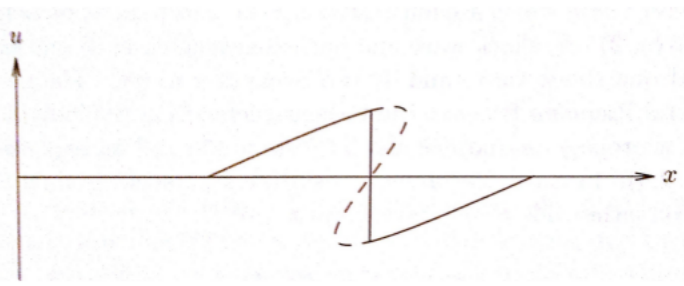


Figure 3.2: Non-unique solution from the method of characteristics shown along with a shock formation [3].

This solution technique does seem to present a repeatable method to solve for the current flow variables from conditions at a previous time. It would not be difficult to write an algorithm which repeated this process for some finite number of points approximating some physical domain. The algorithm would yield a complete solution for $u(x,t)$ and $a(x,t)$. However, there are conditions where this method breaks down. The shortcoming can be described in reference to figure 3.1 Although the nature of the method of characteristics guarantees an intersection of the 3 characteristics, it says nothing of the potential for *more* than three points on the initial condition to follow slopes which will converge to point q. In other words, there are conditions for which equations (3.1), (3.2), (3.3) do not provide a *unique* solution.

The result of method of characterisic solution attempt when such conditions are present is a set of equations that can exhibit multi-valued solutions, i.e., the solution would permit more than one velocity at a single point in space. This is physically unfeasible, and the conditions which lead to such a solution in the method of characteristics are the conditions which lead, in reality, to the formation of a shock wave. Figure 3.2 demonstrates the non-unique solution, which results given the sinusoidal initial conditions, superimposed on the shock wave solution.

3.2 Shock Formation

As previously discussed, it is possible, given certain initial conditions, for the method of characteristics to yield non-unique solutions. A non-unique solution from the method of characteristics indicates a discontinuity in the solution of the general Euler equations, which indicates the solution containing a shock wave. It is possible to determine the conditions required for a shock wave to form. We will follow the development of Knight (2006)[3].

Starting with the governing equations in differential form,

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u)}{\partial x} = 0 \tag{3.12}$$

$$\frac{\partial (\rho u)}{\partial t} + \frac{\partial (\rho u^2)}{\partial x} = -\frac{\partial p}{\partial x} \tag{3.13}$$

$$\frac{\partial (\rho e)}{\partial t} + \frac{\partial (\rho^2 e + \rho p)}{\partial x} = 0 \tag{3.14}$$

but writing the energy equation in terms of entropy:

$$\frac{\partial s}{\partial t} + u\frac{\partial (s)}{\partial x} = 0 \tag{3.15}$$

If we assume that the velocity u is a function of density ρ, we con modify the conservation of mass

$$\frac{\partial \rho}{\partial t} + \frac{d(\rho u)}{dx}\frac{\partial \rho}{\partial x} = 0 \tag{3.16}$$

which can be rearranged to show

$$\frac{\partial \rho}{\partial t}\Big(\frac{\partial \rho}{\partial x}\Big)^{-1} = -\frac{d\rho u}{du} \tag{3.17}$$

In order to evaluate the left hand side, which contains the partial differentiation of ρ with respect to both time and space, let us consider the gradient of density in time and space.

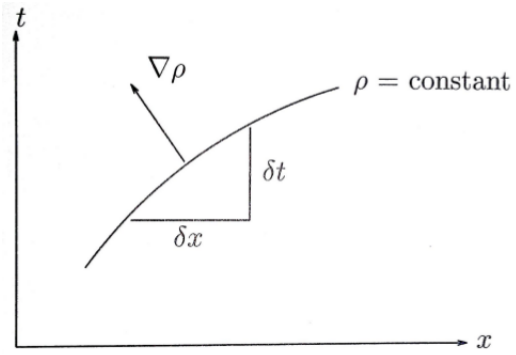


Figure 3.3: Isocontour of density [3]

$$\nabla\rho = \frac{\partial\rho}{\partial t}\vec{e}_x + \frac{\partial\rho}{\partial x}\vec{e}_t \tag{3.18}$$

If a vector is built in the $x-t$ plane, $\delta x\vec{e}_x + \delta t\vec{e}_t$,

which represents the slope of an isocontour of density at any point along the isocontour in the x-t plane, we can then use the orthogonality of the slope and the gradient vector

$$\left(\frac{\partial\rho}{\partial x}\vec{e}_x + \frac{\partial\rho}{\partial t}\vec{e}_t\right)\cdot(\delta x\vec{e_x} + \delta t\vec{e_t}) = 0 \tag{3.19}$$

$$\frac{\partial\rho}{\partial x}\delta x + \frac{\partial\rho}{\partial t}\delta t = 0 \tag{3.20}$$

$$\frac{\partial\rho}{\partial x}\delta x = -\frac{\partial\rho}{\partial t}\delta t \tag{3.21}$$

therefore,

$$-\frac{\delta x}{\delta t} = \frac{\partial\rho}{\partial t}\left(\frac{\partial\rho}{\partial x}\right)^{-1} \tag{3.22}$$

And still considering an isocontour of density in the $x-t$ plane, we can use the partial derivative

$$-\frac{\partial x}{\partial t}\Big|_{\rho} = \frac{\partial \rho}{\partial t}\Big(\frac{\partial \rho}{\partial x}\Big)^{-1} \tag{3.23}$$

Then comparing to (3.17), we can write

$$\frac{\partial x}{\partial t}\Big|_{\rho} = \frac{dpu}{d\rho} \tag{3.24}$$

Looking at the conservation of momentum written in terms of the convective derivative of u, we can write

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + \frac{1}{\rho}\frac{\partial p}{\partial x} = 0 \tag{3.25}$$

And if we expand the pressure gradient term,

$$\frac{\partial p}{\partial x} = \frac{\partial p}{\partial \rho}\Big|_{s}\frac{\partial \rho}{\partial x} + \frac{\partial p}{\partial s}\Big|_{\rho}\frac{\partial s}{\partial x} \tag{3.26}$$

Given that the flow is assumed adiabatic and frictionless, the second term on the right is zero. We can then use (2.9) to rewrite the above as

$$\frac{\partial p}{\partial x} = a^2\frac{\partial \rho}{\partial x} \tag{3.27}$$

Because we are assuming that u is a function of ρ, we can also write the density gradient as

$$\frac{\partial \rho}{\partial x} = \frac{d\rho}{du}\frac{\partial u}{\partial x} \tag{3.28}$$

or,

$$\frac{\partial \rho}{\partial x} = \Big(\frac{du}{d\rho}\Big)^{-1}\frac{\partial u}{\partial x} \tag{3.29}$$

So the pressure gradient term in (3.27) can be expressed as

$$\frac{\partial p}{\partial x} = a^2\Big(\frac{du}{d\rho}\Big)^{-1}\frac{\partial u}{\partial x} \tag{3.30}$$

which can be used in the conservation of momentum (3.15)

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + \frac{1}{\rho}a^2\Big(\frac{du}{d\rho}\Big)^{-1}\frac{\partial u}{\partial x} = 0 \tag{3.31}$$

to yield

$$\frac{\partial u}{\partial t}\Big(\frac{\partial u}{\partial x}\Big)^{-1} = -\Big[u + \frac{a^2}{\rho}\Big(\frac{du}{d\rho}\Big)^{-1}\Big] \tag{3.32}$$

The left hand side of (3.32) is analogous to the left hand side of (3.17). If a similar argument is constructed, but using isocontours of velocity rather than density, it can be shown that

$$\frac{\partial u}{\partial t}\Big(\frac{\partial u}{\partial x}\Big)^{-1} = -\frac{\partial x}{\partial t}\Big|_u \tag{3.33}$$

And because we assume that u is a function of ρ,

$$\frac{\partial x}{\partial t}\Big|_u = \frac{\partial x}{\partial t}\Big|_\rho \tag{3.34}$$

Then using (3.23) and (3.32),

$$\frac{d\rho u}{d\rho} = u + \frac{a^2}{\rho}\Big(\frac{du}{d\rho}\Big)^{-1} \tag{3.35}$$

and the left side can be expanded to show

$$\cancel{u\frac{d\rho}{d\rho}} + \rho\frac{du}{d\rho} = \cancel{u} \quad + \frac{a^2}{\rho}\Big(\frac{du}{d\rho}\Big)^{-1} \tag{3.36}$$

and then

$$\Big(\frac{du}{d\rho}\Big)^2 = \frac{a^2}{\rho^2} \tag{3.37}$$

which leads to the derivative of velocity with respect to density,

$$\frac{du}{d\rho} = \pm\frac{a}{\rho} \tag{3.38}$$

which can be integrated as

$$u = \pm \int \frac{a}{\rho} d\rho \tag{3.39}$$

If we then use the isentropic pressure equations (2.26) and (2.27) along with (2.10) to find a relation between ρ and a, the integral can be further simplified. Equating the pressure relations we find

$$\left(\frac{\rho}{\rho_1}\right)^{\gamma} = \left(\frac{T}{T_1}\right)^{\frac{\gamma}{\gamma-1}} \tag{3.40}$$

and then using (2.10),

$$\left(\frac{\rho}{\rho_1}\right)^{\gamma-1} = \left(\frac{a}{a_1}\right)^{2} \tag{3.41}$$

which can be shown as

$$\frac{\gamma-1}{2} \ln\left(\frac{\rho}{\rho_1}\right) = \ln\left(\frac{a}{a_1}\right) \tag{3.42}$$

Therefore,

$$\int \frac{1}{\rho} d\rho = \frac{2}{\gamma-1} \int \frac{1}{a} da \tag{3.43}$$

and

$$\frac{1}{\rho} d\rho = \frac{2}{\gamma-1} \frac{1}{a} da \tag{3.44}$$

which leads us to

$$d\rho = \frac{2}{\gamma-1} \frac{\rho}{a} da \tag{3.45}$$

and if (3.45) is substituted into (3.39),

$$u = \pm \frac{2}{\gamma - 1} \int da \tag{3.46}$$

and

$$u - u_0 = \pm \frac{2}{\gamma - 1}(a - a_0) \tag{3.47}$$

where a_0 is the speed of sound at the location where $u = u_0$. If we assume that at some time $u_0 = 0$, we can then say

$$a = a_0 \pm \frac{\gamma - 1}{2} u \tag{3.48}$$

and if we divide by a_0 and use (3.41) it can be shown that

$$\rho = \rho_0 \Big[1 \pm \frac{\gamma - 1}{2} \frac{u}{a_0} \Big]^{\frac{2}{\gamma - 1}} \tag{3.49}$$

Equation (2.26) then yields

$$p = p_0 \Big[1 \pm \frac{\gamma - 1}{2} \frac{u}{a_0} \Big]^{\frac{2\gamma}{\gamma - 1}} \tag{3.50}$$

We can then use (3.32), (3.33), (3.38) to show

$$\frac{\partial x}{\partial t}\Big|_u = u \pm a \tag{3.51}$$

and then using (3.47) where $u_0 = 0$,

$$\frac{\partial x}{\partial t}\Big|_u = \pm a_0 + \frac{\gamma + 1}{2} u \tag{3.52}$$

from which we can express position, x, as a function of time, t

$$x = \Big[\pm a_0 + \frac{\gamma + 1}{2} u \Big] t + f(u) \tag{3.53}$$

where $f(u)$ is the position as a function of velocity at time $t = 0$. To evaluate the solution in regards to the formation of a shock wave, consider the characteristic of a right traveling wave which will have the slope:

$$\frac{\partial x}{\partial t}\Big|_u = a_0 + \frac{\gamma + 1}{2}u \tag{3.54}$$

Thus, for a given velocity profile, $u(x, 0)$, the profile will begin to travel to the right with each point moving at a rate proportional to the velocity at that point. Considering this, if a region in space is moving at a velocity less than that of a second region directly to its left, the left region would "catch" and surpass the original region. The condition, when considering a right moving wave in the 1-D solution, for one region to "catch" another is satisfied by velocity values decreasing with increasing x values. Any initial condition which contains a region of $\frac{\partial u}{\partial x} < 0$ will at some time, t_s, will imply the existence of single points in space with multiple values for velocity. This is a physical impossibility, so a shock wave must form at t_s.

At the formation of the shock, the slope of the velocity profile goes to infinity, but there can be no multi-valued solutions in space. These two conditions imply

$$\frac{\partial x}{\partial u}\Big|_t = 0 \quad and \quad \frac{\partial^2 x}{\partial u^2} = 0 \tag{3.55}$$

We can then consider these restraints on (3.53),

$$\frac{\partial x}{\partial u} = \frac{\gamma + 1}{2}t_s + \frac{\partial f}{\partial u} \tag{3.56}$$

which leads to

$$t_s = -\frac{2}{\gamma - 1}\frac{df}{du} \tag{3.57}$$

and

$$\frac{d^2 f}{du^2} = 0 \tag{3.58}$$

These two equations allow the determination of the time at which the shock will form, given proper initial conditions. The time, t_s, can be used, for a known f, in (3.53) to find the location of the shock.

For this investigation, the initial condition was chosen to be periodic,

$$u(x,0) = \epsilon a_0 sin(\kappa x) \tag{3.59}$$

Where ϵ is a dimensionless constant, a_0 is the speed of sound where $u = 0$, and $\kappa = \frac{2\pi}{\lambda}$. This can be expressed with position being the independent variable as

$$x = f(u) = \frac{1}{\kappa} \sin^{-1}\left(\frac{u}{\epsilon a_0}\right) \tag{3.60}$$

We then need to evaluate the first and second derivatives to determine the time of occurrence and position of the shock. Over a full period, there are multiple positions with the same velocity. Because we are using velocity as the dependent variable, we will need to evaluate $\frac{df}{du}$ in segments in such a way that each segment is a smooth differentiable function. From inspection of figure 2.1, let us break $f(u)$ into the following segments which span an entire period.

$$\frac{\pi}{2} \leq \kappa x \leq \frac{3\pi}{2} \qquad and \qquad -\frac{\pi}{2} \leq \kappa x \leq \frac{\pi}{2} \tag{3.61}$$

The first derivative can then be evaluated as

$$\frac{df}{du} = \begin{cases} -\kappa^{-1}\left[(\epsilon a_0)^2 - u^2\right]^{-\frac{1}{2}} & \frac{\pi}{2} \leq \kappa x \leq \frac{3\pi}{2} \\ \kappa^{-1}\left[(\epsilon a_0)^2 - u^2\right]^{-\frac{1}{2}} & -\frac{\pi}{2} \leq \kappa x \leq \frac{\pi}{2} \end{cases} \tag{3.62}$$

and

$$\frac{d^2 f}{du^2} = \begin{cases} -2u\frac{\kappa^{-1}}{2}\left[(\epsilon a_0)^2 - u^2\right]^{-\frac{3}{2}} & \frac{\pi}{2} \leq \kappa x \leq \frac{3\pi}{2} \\ 2u\frac{\kappa^{-1}}{2}\left[(\epsilon a_0)^2 - u^2\right]^{-\frac{3}{2}} & -\frac{\pi}{2} \leq \kappa x \leq \frac{\pi}{2} \end{cases} \tag{3.63}$$

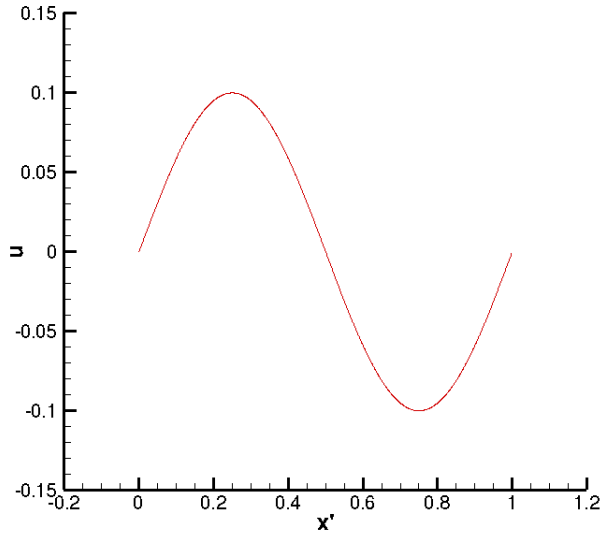
Combinging (3.58) and (3.63), we see that there is an inflection point at $u = 0$. We can then solve for $\frac{df}{du}$ at $u = 0$ to find

$$t_s = \frac{2}{(\lambda + 1)\epsilon\kappa a_0} \tag{3.64}$$

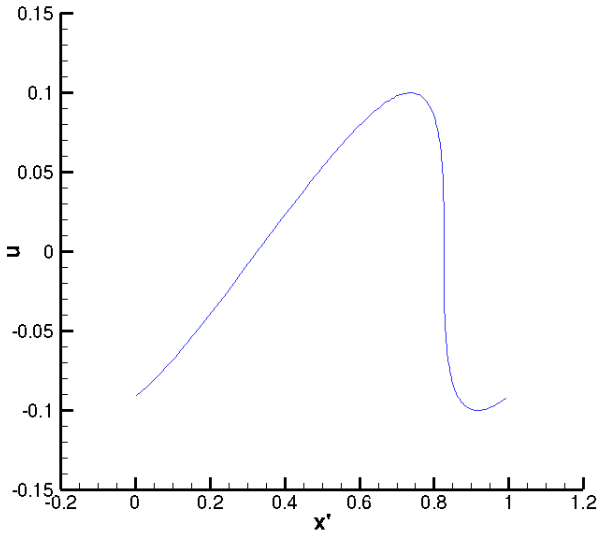
Where the negative value of $\frac{df}{du}$ is used in the evaluation. It is then clear from (3.53) that the position of the shock can be found by

$$x = a_o t_s + f(0) \tag{3.65}$$

Figure 3.1(b) plots the solution just before the formation of the shock wave, given the initial condition shown in figure 3.1(a). The position is shown normalized to the wavelength of the initial periodic velocity profile. The non-dimensional constant was chosen to be $\epsilon = 0.1$, and the speed of sound at $u = 0$ was chosen to be $a_0 = 1$. This solution will be used for comparison to solutions generated by the code developed for both serial CPU execution and parallel execution on the GPU.



(a) Initial Conditions



(b) Just Prior to Shock

Figure 3.4: Velocity profiles as a function of non-dimensional position

Chapter 4

Approximation Techniques and Serial Solution

4.1 Problem Reduction

In order to reduce the problem one which can be solved with a repeatable algorithm, we will begin with the conservative form of the Euler equations.

$$\frac{\partial \mathcal{Q}}{\partial t} + \frac{\partial \mathcal{F}}{\partial x} = 0 \tag{4.1}$$

where

$$\mathcal{Q} = \left\{ \begin{array}{c} Q_1 \\ Q_2 \\ Q_3 \end{array} \right\} = \left\{ \begin{array}{c} \rho \\ \rho u \\ \rho e \end{array} \right\} \tag{4.2}$$

and

$$\mathcal{F} = \left\{ \begin{array}{c} F_1 \\ F_2 \\ F_3 \end{array} \right\} = \left\{ \begin{array}{c} \rho u \\ \rho u^2 + p \\ \rho e u + p u \end{array} \right\} \tag{4.3}$$

Following the development shown in (Knight 2006) [3], we seek a solution for the velocity field in a system governed by these equations. The approximation will make use of the integral form of (4.2) and (4.3),

$$\frac{d}{dt} \int_V \mathcal{Q} dx dy + \int_{\partial V} \mathcal{F} dy = 0 \tag{4.4}$$

First we consider a finite volume element within the flow. In one dimension the element can be shown schematically as shown in figure 4.1.

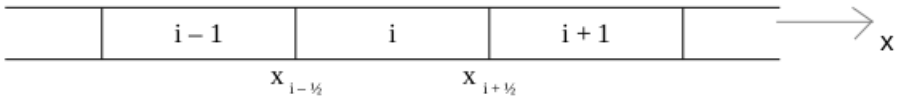


Figure 4.1: Schematic of control volumes used in 1-D flow analysis

Space is approximated as a one dimensional grid along the x-axis consisting of N finite volume elements, each having volume (per unit depth) of $\Delta x \Delta y$. Time can be approximated by a series of discrete points shown as $t^n, n = 1...$ where

$$t^{n+1} = t^n + \Delta t^n \tag{4.5}$$

For each volume element i, we can express a volume-averaged vector of dependent variables,

$$Q_i(t) = \frac{1}{V_i} \int_{V\,i} \mathcal{Q} dx dy \tag{4.6}$$

At the cell faces, we can average the flux over the area of the face. For instance, at the face mating volume i and volume $i + 1$, we have

$$F_{i+\frac{1}{2}} = \frac{1}{A_{i+\frac{1}{2}}} \int_{x+\frac{1}{2}} \mathcal{F} dy \tag{4.7}$$

where $A_{i+\frac{1}{2}}$ is the surface area (per unit depth) of the mating faces and is equal to $A_{i+\frac{1}{2}} = \Delta y$. At this point we have not yet specified a method for determining the specific elements of the flux vector, $\mathcal{F}$. Using (4.6) and (4.7) in the Euler equations presented in (4.1), we get

$$\frac{d}{dt}\Big[\frac{1}{V_i} \int_V \mathcal{Q} dx dy\Big] + \frac{d}{dx}\Big[\frac{1}{A_{i+\frac{1}{2}}} \int_{i+\frac{1}{2}} \mathcal{F} dy\Big] = 0 \tag{4.8}$$

because we assume that V_i does not vary in time, (4.8) reduces to

$$\frac{dQ_i}{dt} + \frac{\left(F_{i+\frac{1}{2}} - F_{i-\frac{1}{2}}\right)}{\Delta x} = 0 \tag{4.9}$$

Where the flux is approximated as linear within a cell. The system of partial differential equations describing the one dimensional Euler flow has been transformed into a system of ordinary differential equations. The system can then be integrated in time to yield an approximate solution for the dependent variables contained in the vector $\mathcal{Q}_i$, given an initial condition at $t = t^n$,

$$Q_i^{n+1} = Q_i^n - \frac{1}{\Delta x}\int_{t^n}^{t^{n+1}} \left(F_{i+\frac{1}{2}} - F_{i-\frac{1}{2}}\right) dt \tag{4.10}$$

where the superscript notes time and subscript space. We have reduced the problem to generating an algorithm which can evaluate the flux term (4.10) at a specific time at each cell interface. The flux can then be used to step from known conditions at $t = t^n$ to $t = t^{n+1}$.

4.2 Reconstruction

In this discrete approximation, the flow variables are evaluated at some number, say N, of cells along the x-axis at the cell centroids. It is clear from (4.10) that in order to describe the time evolution of the flow we will not only need flow variables at the centroids, but the values at the cell interfaces as well.

Within each cell, a function $Q_i(x)$ can be defined. This function will serve as a local reconstruction of the exact function. In this investigation, the simplest reconstruction is used,

$$Q_i(x) = Q_i \tag{4.11}$$

where Q_i is the vector of cell averaged flow variables. This leads to step discontinuities in Q at the interfaces between adjacent cells, due to the fact that, in general, $Q_i \neq Q_{i+1}$. Figure 4.2 illustrates the discontinuities. Each interface will have a left-hand and right-hand value. For instance, from Figure 4.2, the face separating cell $i-1$ and cell i, has on its left side, $Q^l_{i-\frac{1}{2}} = Q_{i-1}$, and on its right side, $Q^r_{i-\frac{1}{2}} = Q_i$. These two values will be used for the

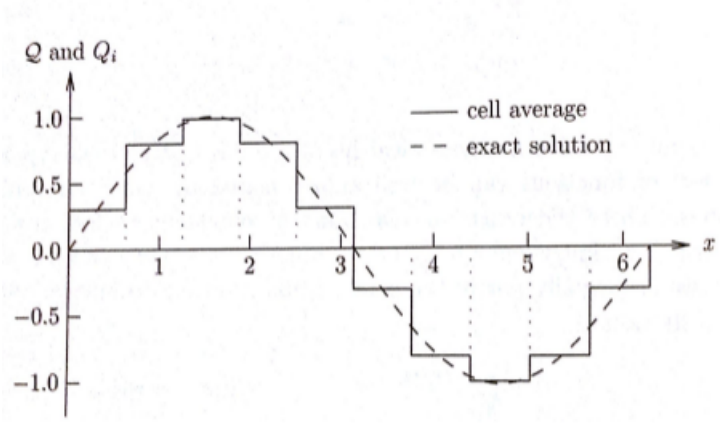


Figure 4.2: Exact function $\mathcal{Q}(x) = sin(x)$ against the reconstructed function using a cell averaged Q_i [3]

.

calculation of the flux between cells, i.e., $F_{i+\frac{1}{2}}$ and $F_{i-\frac{1}{2}}$.

4.3 Determining the Flux Vector

When attempting to find the flux at each cell face, the fundamental problem is the step discontinuity in the reconstructed function at each cell interface. The discontinuities are analogs to the General Riemann Problem. If algorithms can be constructed to solve the Riemann problem at each face to determine the flux vector, then applying equation (4.10) leads directly to a solution. The strategy of modeling the discontinuities as Riemann problems was originally developed by Godunov (1959). This investigation uses a method developed by Roe (1981) which seeks an exact solution to an approximation of the Riemann problem. In order to detail the method developed by Roe, we will first summarize the solutions to the General Riemann Problem.

4.3.1 The General Riemann Problem

The General Riemann Problem consists of two flow states, at time $t = 0$, which are separated by a contact surface. The left state is given as velocity u_1, pressure p_1, and temperature

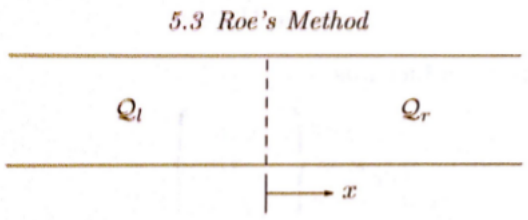


Figure 4.3: Riemann problem at the interface between 2 cells [3]

T_1. The right state is defined by velocity u_4, pressure p_4, and temperature T_4.

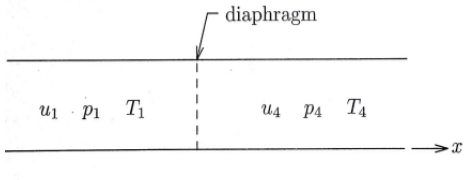


Figure 4.4: General Riemann Probelm at $t = 0$ [3]

If the two states are released for $t > 0$, there are four possible solutions. The solutions depend on the values of the state variables at $t = 0$, and more specifically the value of the pressure at the contact surface, p^*. The contact surface pressure, p^*, is defined by the trancendental equation [3].

$$a_1 f(p^*, p_1) + a_4 f(p^*, p_4) - u_1 - u_4 = 0 \tag{4.12}$$

where

$$f(p^*, p) = \begin{cases} \frac{1}{\gamma}\left(\frac{p^*}{p} - 1\right)\left[\frac{\gamma+1}{2\gamma}\frac{p^*}{p} + \frac{\gamma-1}{2\gamma}\right]^{-\frac{1}{2}} & for \quad p^* \geq p \\ \frac{2}{\gamma-1}\left[\left(\frac{p^*}{p}\right)^{\frac{(\gamma-1)}{2\gamma}} - 1\right] & for \quad p^* \leq p \end{cases} \tag{4.13}$$

The above equation can be solved iteratively for p^*. There are four possible solutions, depending on initial conditions, which are shown in Table 4.1.

Each solution is separated into four regions separated by three surfaces, namely, the contact surface and both propagating waves. The four solutions are presented graphically in Figure

Case			Result
1: $p* > p_1$	*and*	$p* > p_4$	Two Shock Waves
2: $p* > p_1$	*and*	$p* < p_4$	Shock and Expansion Wave
3: $p* < p_1$	*and*	$p* > p_4$	Expansion and Shock Wave
4: $p* < p_1$	*and*	$p* < p_4$	Two Expansion Waves

Table 4.1: Four Solutions to the General Riemann Problem

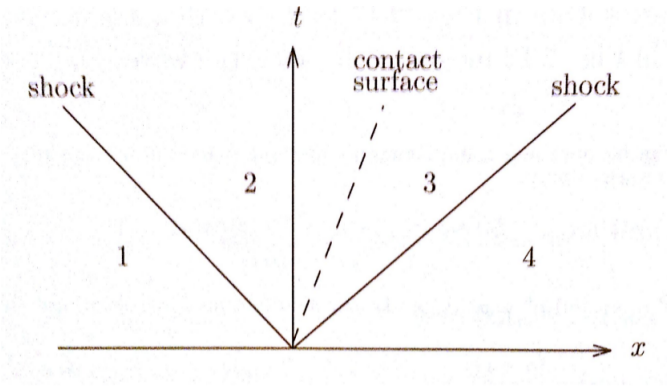


Figure 4.5: Two Shock Waves

4.5. [3].

4.3.2 Roe's Method

Roe's method for determining the flux at each interface makes use of the nonconservative vector form of the Euler equations.

$$\frac{\partial \mathcal{Q}}{\partial t} + \mathbf{A}\frac{\partial \mathcal{Q}}{\partial x} = 0 \tag{4.14}$$

The matrix $\mathbf{A}$ is the Jacobian matrix coming from the chain rule,

$$\mathbf{A} = \frac{\partial \mathcal{F}}{\partial \mathcal{Q}} \tag{4.15}$$

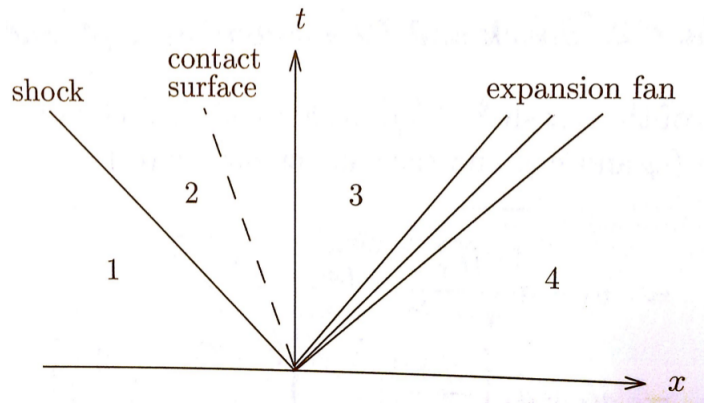


Figure 4.6: Shock and Expansion

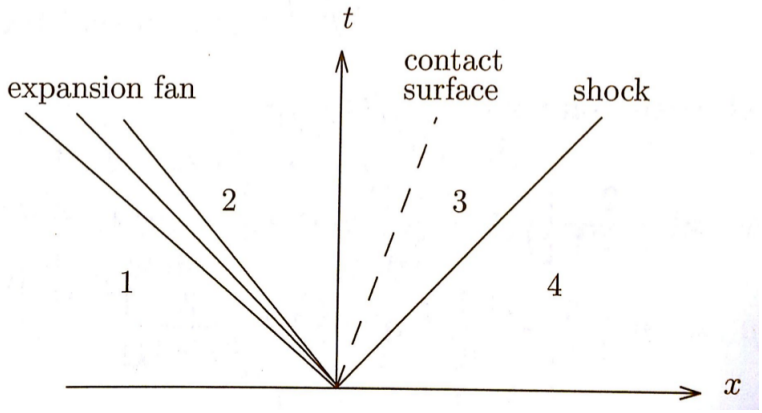


Figure 4.7: Expansion and Shock

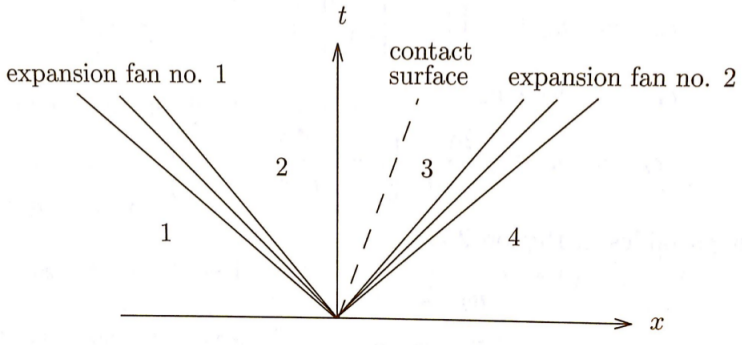


Figure 4.8: Two Expansion Waves

In Chapter 2, the Jacobian matrix was developed to

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 \\ \frac{\gamma-3}{2}u^2 & (3-\gamma)u & \gamma-1 \\ -\gamma eu + (\gamma-1)u^3 & \gamma e - \frac{3}{2}(\gamma-1)u^2 & \gamma u \end{pmatrix} \tag{4.16}$$

The total energy per unit mass, e, can be eliminated with some algebraic manipulation. Beginning with the total energy per unit mass being equal to the sum of internal energy and kinetic energy, $e = e_i + \frac{1}{2}u^2$, and using the calorically perfect assumption, i.e., $e_i = c_v T$, along with the definition of the speed of sound, $a = \sqrt{\gamma RT}$, it can be shown

$$e = \frac{a^2}{\gamma(\gamma-1)} + \frac{1}{2}u^2 \tag{4.17}$$

which can be used to write $\mathbf{A}$ in terms of u and a,

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 0 \\ \frac{\gamma-3}{2}u^2 & (3-\gamma)u & \gamma-1 \\ \frac{-ua^2}{\gamma-1} + (\gamma-1)u^3 & \frac{a^2}{\gamma-1} + \frac{3-2\gamma}{2}u^2 & \gamma u \end{pmatrix} \tag{4.18}$$

And then incorporating the definition of total enthalpy per unit mass, $H = c_p T + \frac{1}{2}u^2$, the matrix can be manipulated further. Using the speed of sound, $a = \sqrt{\gamma RT}$,

$$H = \frac{c_p a^2}{\gamma R} + \frac{1}{2}u^2 \tag{4.19}$$

and knowing the ratio of specific heats, $\gamma = c_p/c_v$ and the species gas constant, $R = c_p - c_v$, the enthalpy can be shown as

$$H = \frac{a^2}{\gamma-1} + \frac{1}{2}u^2 \tag{4.20}$$

which can be substituted into the matrix $\mathbf{A}$ to remove a. For instance, the first element of the third row:

$$A_{31}(u,a) = \frac{-ua^2}{\gamma - 1} + (\gamma - 1)u^3 \tag{4.21}$$

but substituting in $a^2 = (\gamma - 1)(H - u^2/2)$ from above,

$$A_{31}(u,H) = -u(H - \frac{u^2}{2}) + (\frac{\gamma - 2}{2})u^3 \tag{4.22}$$

$$A_{31}(u,H) = -Hu + u^3\frac{\gamma - 1}{2} \tag{4.23}$$

A_{32} can be found in a similar way. The result is

$$\mathbf{A}(u,H) = \begin{pmatrix} 0 & 1 & 0 \\ \frac{\gamma-3}{2}u^2 & (3-\gamma)u & \gamma - 1 \\ \frac{\gamma-1}{2}u^3 - Hu & H - (\gamma-1)U^2 & \gamma u \end{pmatrix} \tag{4.24}$$

Restating the governing equation,

$$\frac{\partial \mathcal{Q}}{\partial t} + \mathbf{A}\frac{\partial \mathcal{Q}}{\partial x} = 0 \tag{4.25}$$

where $\mathbf{A}$ can be expressed in the various forms shown previously. This is just another form of the one-dimensional, unsteady, nonlinear Euler equations which describe flow continuously in space and time. Roe modified this equation to solve the Riemann problem at each interface with an approximate form of the equation above

$$\frac{\partial \mathcal{Q}}{\partial t} + \tilde{\mathbf{A}}(\mathcal{Q}_l, \mathcal{Q}_r)\frac{\partial \mathcal{Q}}{\partial x} = 0 \tag{4.26}$$

where the matrix $\tilde{\mathbf{A}}$ is a function of the left and right states of the dependent variable vector at each cell interface but constant for a given interface. The approximation matrix $\tilde{\mathbf{A}}$ is constructed to satisfy the following properties [11]

1. $\tilde{\mathbf{A}}$ provides a linear mapping of $\mathcal{Q}$ to $\mathcal{F}$
2. $\tilde{\mathbf{A}}(\mathcal{Q}_l, \mathcal{Q}_r) \rightarrow \mathbf{A}(\mathcal{Q})$ as $\mathcal{Q}_l \rightarrow \mathcal{Q}_r \rightarrow \mathcal{Q}$
3. For any $\mathcal{Q}_l$ and $\mathcal{Q}_r$, $\tilde{\mathcal{A}}(\mathcal{Q}_l, \mathcal{Q}_r) \times (\mathcal{Q}_l - \mathcal{Q}_r) \equiv \mathcal{F}_l - \mathcal{F}_r$

4. The eigenvectors of $\tilde{\mathbf{A}}$ are linearly independent

To find $\tilde{\mathbf{A}}$, Roe introduced a parametrization vector ν,

$$\nu = \begin{Bmatrix} \sqrt{\rho} \\ \sqrt{\rho}u \\ \sqrt{\rho}H \end{Bmatrix} \tag{4.27}$$

This can be used to express $\mathcal{Q}$ and $\mathcal{F}$ as

$$\mathcal{Q} = \begin{Bmatrix} \nu_1^2 \\ \nu_1\nu_2 \\ \nu_1\nu_3/\gamma + (\gamma-1)\nu_2^2/2\gamma \end{Bmatrix} \tag{4.28}$$

and

$$\mathcal{F} = \begin{Bmatrix} \nu_1\nu_2 \\ (\gamma-1)\nu_1\nu_3/\gamma + (\gamma-1)\nu_2^2/2\gamma \\ \nu_2\nu_3 \end{Bmatrix} \tag{4.29}$$

The Δ operator will be used to represent the difference in the left and right states of a variable, i.e., $\Delta\nu = \nu_l - \nu_r$. Applying this notation to $\mathcal{Q}$,

$$\Delta\mathcal{Q} = \begin{Bmatrix} \Delta Q_1 \\ \Delta Q_2 \\ \Delta Q_3 \end{Bmatrix} = \begin{Bmatrix} \nu_{1l}^2 - \nu_{1r}^2 \\ \nu_{1l}\nu_{2l} - \nu_{1r}\nu_{2r} \\ \left(\nu_{1l}\nu_{3l}/\gamma + (\gamma-1)\nu_{2l}^2/2\gamma\right) - \left(\nu_{1r}\nu_{3r}/\gamma + (\gamma-1)\nu_{2r}^2/2\gamma\right) \end{Bmatrix} \tag{4.30}$$

Noticing that $\Delta\mathcal{Q}_1$ is a difference of squares, and making use of the identity $\bar{f} \equiv \frac{1}{2}(f_l + f_r)$, it can be shown that

$$\Delta\mathcal{Q}_1 = 2\bar{\nu_1}\Delta\nu_1 \tag{4.31}$$

Then using the identity $\Delta(fg) = \bar{f}\Delta g + \Delta f\bar{g}$ it can be shown that

$$\Delta\mathcal{Q}_2 = \bar{\nu_2}\Delta\nu_1 + \bar{\nu_1}\Delta\nu_2 \tag{4.32}$$

And both identities can be used to show that

$$\Delta\mathcal{Q}_3 = \bar{\nu_3}\Delta\nu_1/\gamma + \bar{\nu_1}\Delta\nu_3/\gamma + (\gamma-1)\bar{\nu_2}\Delta\nu_2/\gamma \tag{4.33}$$

The same identities can be used to write the $\Delta\mathcal{F}$ vector. Summarizing, we now have a vector of flow variables built from left and right flow states, and a vector of flux terms also built from left and right states. They are printed here:

$$\Delta\mathcal{Q} = \left\{ \begin{array}{c} 2\bar{\nu_1}\Delta\nu_1 \\ \bar{\nu_2}\Delta\nu_1 + \bar{\nu_1}\Delta\nu_2 \\ \bar{\nu_3}\Delta\nu_1/ + (\gamma-1)\bar{\nu_2}\Delta\nu_2/\gamma + \gamma + \bar{\nu_1}\Delta\nu_3/\gamma \end{array} \right\} \tag{4.34}$$

$$\Delta\mathcal{F} = \left\{ \begin{array}{c} \bar{\nu_2}\Delta\nu_1 + \bar{\nu_1}\Delta\nu_2 \\ (\gamma-1)\bar{\nu_3}\Delta\nu_1/\gamma + (\gamma-1)\bar{\nu_2}\Delta\nu_2/\gamma + (\gamma-1)\bar{\nu_1}\Delta\nu_3/\gamma \\ \bar{\nu_3}\Delta\nu_2 + \bar{\nu_2}\Delta\nu_3 \end{array} \right\} \tag{4.35}$$

Upon inspection, it is apparent that two matrices can be constructed which map the parametrization vector $\Delta\nu$ to the $\mathcal{Q}$ and $\mathcal{F}$ vectors.

$$\Delta\mathcal{Q} = \mathbf{B}\Delta\nu \tag{4.36}$$

Where, by inspection,

$$\mathbf{B} = \begin{pmatrix} 2\bar{\nu_1} & 0 & 0 \\ \bar{\nu_2} & \bar{\nu_1} & 0 \\ \bar{\nu_3}/\gamma & (\gamma-1)\bar{\nu_2}/\gamma & \bar{\nu_1}/\gamma \end{pmatrix} \tag{4.37}$$

and

$$\Delta\mathcal{F} = \mathbf{C}\Delta\nu \tag{4.38}$$

where

$$\mathbf{C} = \begin{pmatrix} \bar{\nu_2} & \bar{\nu_1} & 0 \\ (\gamma-1)\bar{\nu_3}/\gamma & (\gamma+1)\bar{\nu_2}/\gamma & (\gamma-1)\bar{\nu_1}/\gamma \\ 0 & \bar{\nu_3} & \bar{\nu_2} \end{pmatrix} \tag{4.39}$$

Considering the Roe matrix, $\tilde{\mathbf{A}}$, and stating the first property mathematically,

$$\mathcal{F} = \tilde{\mathbf{A}}\mathcal{Q} \tag{4.40}$$

A comparison of left and right flow states can then be written as,

$$\Delta\mathcal{F} = \tilde{\mathbf{A}}\Delta\mathcal{Q} \tag{4.41}$$

So,

$$C\Delta\nu = \tilde{\mathbf{A}}\Delta\mathcal{Q} \tag{4.42}$$

$$C\Delta\nu = \tilde{\mathbf{A}}\mathbf{B}\Delta\nu \tag{4.43}$$

and therefore

$$\tilde{\mathbf{A}} = \mathbf{C}\mathbf{B}^{-1} \tag{4.44}$$

So finding the Roe matrix $\tilde{\mathbf{A}}$ reduces to finding $\mathbf{B}$ and $\mathbf{C}$, which were detailed above. Taking the inverse of the matrix $\mathbf{B}$

$$\mathbf{B}^{-1} = \frac{\gamma}{2\bar{\nu_1}^2} \begin{pmatrix} \frac{\bar{\nu_1}^2}{\gamma} & 0 & 0 \\ \frac{-\bar{\nu_1}\bar{\nu_2}}{\gamma} & \frac{2\bar{\nu_1}^2}{\gamma} & 0 \\ \frac{(\gamma-1)\bar{\nu_2}^2-\bar{\nu_1}\bar{\nu_3}}{\gamma} & \frac{-2(\gamma-1)\bar{\nu_1}\bar{\nu_2}}{\gamma} & 2\bar{\nu_1}^2 \end{pmatrix} \tag{4.45}$$

The Roe matrix can then be written out as

$$\tilde{\mathbf{A}} = \frac{\gamma}{2\bar{\nu_1}^2} \begin{pmatrix} \bar{\nu_2} & \bar{\nu_1} & 0 \\ (\gamma-1)\bar{\nu_3}/\gamma & (\gamma+1)\bar{\nu_2}/\gamma & (\gamma-1)\bar{\nu_1}/\gamma \\ 0 & \bar{\nu_3} & \bar{\nu_2} \end{pmatrix} \begin{pmatrix} \frac{\bar{\nu_1}^2}{\gamma} & 0 & 0 \\ \frac{-\bar{\nu_1}\bar{\nu_2}}{\gamma} & \frac{2\bar{\nu_1}^2}{\gamma} & 0 \\ \frac{(\gamma-1)\bar{\nu_2}^2-\bar{\nu_1}\bar{\nu_3}}{\gamma} & \frac{-2(\gamma-1)\bar{\nu_1}\bar{\nu_2}}{\gamma} & 2\bar{\nu_1}^2 \end{pmatrix} \tag{4.46}$$

The elements in the first row are straightforward to evaluate. The results are $\tilde{A}_{11} = 0$, $\tilde{A}_{12} = 1$, and $\tilde{A}_{13} = 0$. The remaining elements will be simplified with the introduction of two new parametrization vectors

$$\tilde{u} \equiv \frac{\bar{\nu_2}}{\bar{\nu_1}} = \frac{\sqrt{\rho_l}u_l + \sqrt{\rho_r}u_r}{\sqrt{\rho_l} + \sqrt{\rho_r}} \tag{4.47}$$

and

$$\tilde{H} \equiv \frac{\bar{\nu_3}}{\bar{\nu_1}} = \frac{\sqrt{\rho_l}H_l + \sqrt{\rho_r}H_r}{\sqrt{\rho_l} + \sqrt{\rho_r}} \tag{4.48}$$

Where $\tilde{u}$ is the Roe-averaged velocity, and $\tilde{H}$ is the Roe-averaged total enthalpy. For instance the element $\tilde{A}_{21}$ can be simplified as follows:

$$\tilde{A}_{21} = \frac{1}{2\nu_1^2\gamma}\left[-\nu_2^2(\gamma+1) + (\gamma-1)^2\nu_2^2\right] \tag{4.49}$$

$$\tilde{A}_{21} = \frac{1}{2}\left[-\frac{\nu_2^2}{\nu_1^2}\frac{(\gamma+1)-(\gamma-1)^2}{\gamma}\right] \tag{4.50}$$

$$\tilde{A}_{21} = \frac{1}{2}\left[\tilde{u}^2\frac{-(\gamma+1)+(\gamma-1)^2}{\gamma}\right] \tag{4.51}$$

$$\tilde{A}_{21} = \frac{1}{2}\tilde{u}^2(\gamma-3) \tag{4.52}$$

The remaining elements are simplified in a similar fashion to yield

$$\tilde{\mathbf{A}} = \begin{pmatrix} 0 & 1 & 0 \\ \frac{\gamma-3}{2}\tilde{u}^2 & (3-\gamma)\tilde{u} & (\gamma-1) \\ -\tilde{H}\tilde{u} + \frac{(\gamma-1)\tilde{u}^3}{2} & \tilde{H} - (\gamma-1)\tilde{u}^2 & \gamma u \end{pmatrix} \tag{4.53}$$

With the Roe-averaged parameters, the Roe matrix, $\tilde{\mathbf{A}}$ is nicely analagous to the exact Jacobian matrix $\mathbf{A}$ shown in equation (4.24). The four properties which the Roe matrix were required to satisfy can now be evaluated. The first and third are satisfied by construction. The second can be shown to be satisfied by inspecting $\tilde{u}$ and $\tilde{H}$. Both Roe-averaged quantities approach their analogous exact quantities u and H as the left state and right state of the Riemann problem approach equality. The fourth property requires solving for the eigenvalues and eigenvectors of $\tilde{\mathbf{A}}$.

To find the eigenvalues, we seek the characteristic polynomial containing the roots of $\tilde{\lambda}$ where

$$det(\tilde{\mathbf{A}} - \tilde{\lambda}\mathbf{I}) = 0 \tag{4.54}$$

Expanding the determinate,

$$\begin{aligned} det(\tilde{\mathbf{A}} - \tilde{\lambda}\mathbf{I}) = &-\gamma \left\{ \left((3-\gamma)\tilde{u} - \lambda\right)(\gamma\tilde{u} - \lambda) - (\gamma-1)(\tilde{H} - (\gamma-1)\tilde{u}^2) \right\} \\ &- \left\{ (\gamma-3)(\frac{\tilde{u}^2}{2}(\gamma u - \lambda) - (\gamma-1)\left((\gamma-1)\frac{\tilde{u}^3}{2} - \tilde{H}\tilde{u}\right) \right\} \end{aligned} \tag{4.55}$$

And if the Roe-averaged total enthalpy is substituted with its analog to equation (4.20),

$$\tilde{H} = \frac{\tilde{a}^2}{\gamma-1} + \frac{1}{2}\tilde{u}^2 \tag{4.56}$$

Then the equation for the determinate can be simplified to

$$(\tilde{\lambda} - \tilde{u})(\tilde{\lambda} - (\tilde{u} + \tilde{a}))(\tilde{\lambda} - (\tilde{u} - \tilde{a})) = 0 \tag{4.57}$$

So the three eigenvalues of $\tilde{\mathbf{A}}$ have been found,

$$\tilde{\lambda}_1 = \tilde{u}, \qquad \tilde{\lambda}_2 = \tilde{u} + \tilde{a}, \qquad \tilde{\lambda}_3 = \tilde{u} - \tilde{a} \tag{4.58}$$

Denoting the eigenvector corresponding to $\tilde{\lambda}_1$ as $\vec{e}_1$, the eigenvector can be found directly using

$$\vec{e}_1 = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \quad and \quad \tilde{\mathbf{A}}\vec{e}_1 = \tilde{\lambda}_1 \tilde{e}_1 \tag{4.59}$$

which can be broken into three equations

$$v_2 = \tilde{u} v_1 \tag{4.60}$$

$$v_1 \frac{(\gamma-3)\tilde{u}^2}{2} + v_2(3-\gamma)\tilde{u} + v_3(\gamma-1) = \tilde{u} v_2 \tag{4.61}$$

$$v_1 \left(-\tilde{H}\tilde{u} + \frac{\gamma-1}{2}\tilde{u}^3 \right) + v_2 \left(\tilde{H} - (\gamma-1)\tilde{u}^2 \right) + v_3 \gamma u = \tilde{u} v_3 \tag{4.62}$$

Substituting the first into the second,

$$v_1 \frac{(\gamma-3)\tilde{u}^2}{2} + v_2(3-\gamma)\tilde{u} + v_3(\gamma-1) = \tilde{u}^2 v_1 \tag{4.63}$$

and rearranging,

$$v_1 \tilde{u}^2 \left(\frac{\gamma-3}{2} + (3-\gamma) - 1 \right) + v_3(\gamma-1) = 0 \tag{4.64}$$

which simplifies to

$$v_3 = \frac{1}{2}\tilde{u}^2 v_1 \tag{4.65}$$

As is customary, it will be said that $v_1 = 1$, so the eigenvector for $\tilde{\lambda}_1 = \tilde{u}$ is

$$\vec{e}_1 = \begin{pmatrix} 1 \\ \tilde{u} \\ \frac{1}{2}\tilde{u}^2 \end{pmatrix} \tag{4.66}$$

The same linear algebra can be performed on the remaining eigenvalues to determine the eigenvectors $\vec{e}_2$ and $\vec{e}_3$

$$\vec{e}_2 = \begin{pmatrix} 1 \\ \tilde{u} + \tilde{a} \\ \tilde{H} + \tilde{u}\tilde{a} \end{pmatrix}, \quad and \quad \begin{pmatrix} 1 \\ \tilde{u} - \tilde{a} \\ \tilde{H} - \tilde{u}\tilde{a} \end{pmatrix} \tag{4.67}$$

Where equation (4.56) has again been used. By inspection, the eigenvectors are linearly independent, so the four properties of the approximation matrix $\tilde{\mathbf{A}}$ are proven satisfied. Looking again at the approximate Riemann problem sought by Roe,

$$\frac{\partial \mathcal{Q}}{\partial t} + \tilde{\mathbf{A}}(\mathcal{Q}_l, \mathcal{Q}_r)\frac{\partial \mathcal{Q}}{\partial x} = 0 \tag{4.68}$$

the matrix $\tilde{\mathbf{A}}$ is known and constant for a given left state and right state of cell interface. Consequently, because the flux vector $\mathcal{F}$ has been approximated by $\mathcal{F} = \tilde{\mathbf{A}}\mathcal{Q}$, the matrix $\tilde{\mathbf{A}}$ also allows for the approximation of the flux at the left and right states. Looking at equation (4.9), if an accurate approximation of the flux vector is known, then the problem has been effectively reduced to a system of ordinary differential equations. An explicit determination of the flux vector is now shown [3].

The Roe matrix $\tilde{\mathbf{A}}$ can be diagonalized as

$$\tilde{\mathbf{A}} = \mathbf{S}\Lambda\mathbf{S}^{-1} \tag{4.69}$$

where $\mathbf{S}$ is the matrix of right eigenvectors of $\tilde{\mathbf{A}}(\mathcal{Q}_l, \mathcal{Q}_r)$,

$$\tilde{\mathbf{S}} = \begin{pmatrix} 1 & 1 & 1 \\ \tilde{u} & \tilde{u} + \tilde{a} & \tilde{u} - \tilde{a} \\ \frac{1}{2}\tilde{u}^2 & \tilde{H} + \tilde{u}\tilde{a} & \tilde{H} + \tilde{u}\tilde{a} \end{pmatrix} \tag{4.70}$$

and S^{-1} can be shown as

$$\tilde{\mathbf{S}^{-1}} = \begin{pmatrix} 1 - \frac{(\gamma-1)\tilde{u}^2}{2\tilde{a}^2} & \frac{(\gamma-1)\tilde{u}}{\tilde{a}^2} & -\frac{\gamma-1}{\tilde{a}^2} \\ \frac{(\gamma-1)\tilde{u}^2}{4\tilde{a}^2} - \frac{\tilde{u}}{2\tilde{a}} & -\frac{(\gamma-1)\tilde{u}}{2\tilde{a}^2} - \frac{1}{2\tilde{a}} & \frac{\gamma-1}{2\tilde{a}^2} \\ \frac{(\gamma-1)\tilde{u}^2}{4\tilde{a}^2} + \frac{\tilde{u}}{2\tilde{a}} & -\frac{(\gamma-1)\tilde{u}}{2\tilde{a}^2} + \frac{1}{2\tilde{a}} & \frac{\gamma-1}{2\tilde{a}^2} \end{pmatrix} \tag{4.71}$$

The approximate Riemann problem can then be written as

$$\frac{\partial \mathcal{Q}}{\partial t} + \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}\frac{\partial \mathcal{Q}}{\partial x} = 0 \tag{4.72}$$

then multiplying all terms by $\mathbf{S}^{-1}$ and defining the matrix $\mathbf{R}$ as $\mathbf{R} \equiv \mathbf{S}^{-1}\mathbf{Q}$,

$$\frac{\partial \mathbf{R}}{\partial t} + \mathbf{\Lambda}\frac{\partial \mathbf{R}}{\partial x} = 0 \tag{4.73}$$

where $\mathbf{R}$ is a function of x and t and $\mathbf{\Lambda}$ is the matrix

$$\mathbf{\Lambda} = \begin{pmatrix} u & 0 & 0 \\ 0 & \tilde{u} + \tilde{a} & 0 \\ 0 & 0 & \tilde{u} - \tilde{a} \end{pmatrix} \tag{4.74}$$

From the previous development of the method of characteristics, the solution to (4.73) is $\mathbf{R}$ = constant on the curves defined by $\frac{dx}{dt} = \tilde{\lambda}$, or

$$R_1 = \quad constant \quad on \quad \frac{dx}{dt} = \tilde{\lambda}_1 = \tilde{u} \tag{4.75}$$

$$R_2 = \quad constant \quad on \quad \frac{dx}{dt} = \tilde{\lambda}_2 = \tilde{u} + \tilde{a} \tag{4.76}$$

$$R_3 = \quad constant \quad on \quad \frac{dx}{dt} = \tilde{\lambda}_3 = \tilde{u} - \tilde{a} \tag{4.77}$$

Each solution is shown graphically in Figures 4.9-4.11.

Along the characteristic curve, R_1 is constant, and the left and right states, R_{1l} and R_{1r} exist on either side. The same can be done for the remaining characteristics.

The solutions can be compiled showing the different solution regions for the Riemann problem.

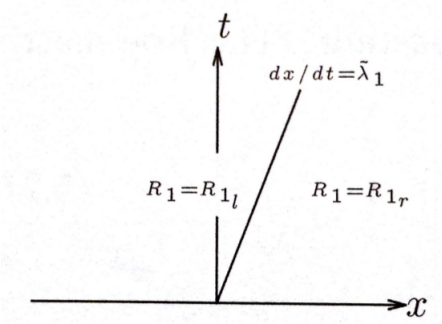


Figure 4.9: First characteristic of Roe's solution to the Riemann Problem [3]

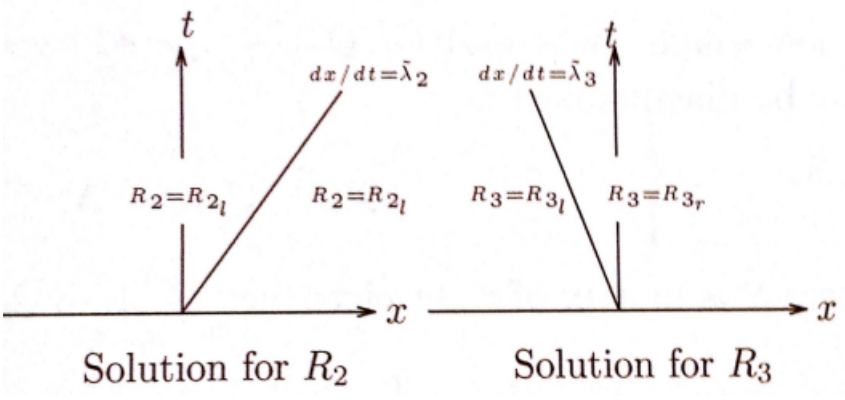


Figure 4.10: Second and third characteristic of Roe's solution to the General Riemann Problem [3]

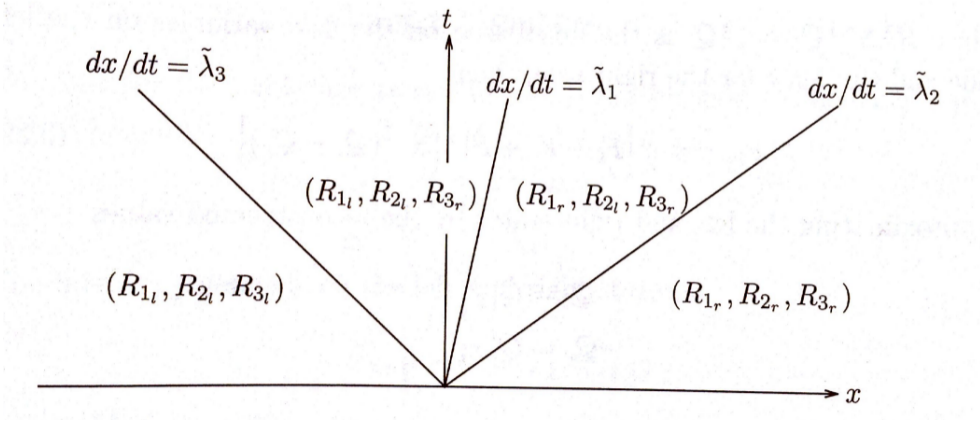


Figure 4.11: All characteristic curves and solution regions of Roe's solution to the General Riemann Problem [3]

For some point in time after $t = 0$ where conditions are known, the elements R_1, R_2, and R_3 can now be expressed explicitly. First R_1 will be investigated. The initial time where condition are know will be expressed as t^n and t^{n+1} will represent one time step forward. The face between two cells, which is the discontinuity in the Riemann problem, will be at the location defined by $x_{i+\frac{1}{2}}$. For the figure shown, which is drawn for $\tilde{a} > \tilde{u} > 0$, the components of $\vec{R}$ are

$$R_1|_{i+\frac{1}{2}}^{n+1} = \frac{1}{2}(R_{1l} + R_{1r}) + \frac{1}{2}(R_{1l} - R_{1r}) \tag{4.78}$$

$$R_2|_{i+\frac{1}{2}}^{n+1} = \frac{1}{2}(R_{2l} + R_{2r}) + \frac{1}{2}(R_{2l} - R_{2r}) \tag{4.79}$$

$$R_3|_{i+\frac{1}{2}}^{n+1} = \frac{1}{2}(R_{3l} + R_{3r}) - \frac{1}{2}(R_{3l} - R_{3r}) \tag{4.80}$$

And more generally it can be written,

$$R_k|_{i+\frac{1}{2}}^{n+1} = \frac{1}{2}(R_{kl} + R_{kr}) + \frac{1}{2}sign(\tilde{\lambda})(R_{kl} - R_{kr}) \tag{4.81}$$

where

$$sign(\tilde{\lambda}) = \begin{cases} +1 & for \quad \tilde{\lambda}_k > 0 \\ -1 & for \quad \tilde{\lambda}_k < 0 \\ 0 & for \quad \tilde{\lambda}_k = 0 \end{cases} \tag{4.82}$$

Restating equation (4.10),

$$Q_i^{n+1} = Q_i^n - \frac{1}{\Delta x}\int_{t^n}^{t^{n+1}} \left(F_{i+\frac{1}{2}} - F_{i+\frac{1}{2}}\right) dt \tag{4.83}$$

where the flux has been shown as

$$F_{i+\frac{1}{2}} = \mathcal{F}_{i+\frac{1}{2}} = \tilde{\mathbf{A}}\mathcal{Q} = \left(\tilde{\mathbf{S}}\tilde{\mathbf{\Lambda}}\tilde{\mathbf{S}}^{-1}\right) \tag{4.84}$$

and the flow variable vector, $\mathcal{Q}$, is defined as

$$\mathcal{Q} = (\tilde{\mathbf{S}}\vec{R}) \tag{4.85}$$

so the flux can be written as

$$F_{i+\frac{1}{2}} = \left(\tilde{\mathbf{S}}\tilde{\mathbf{\Lambda}}\tilde{\mathbf{S}}^{-1}\right)(\tilde{\mathbf{S}}\vec{R})_{i+\frac{1}{2}} = (\tilde{\mathbf{S}}\tilde{\mathbf{\Lambda}}\vec{R})_{i+\frac{1}{2}} \tag{4.86}$$

Using (4.81) and (4.86),

$$F_{i+\frac{1}{2}} = \frac{1}{2}\tilde{\mathbf{S}}\tilde{\mathbf{\Lambda}}(R_l + R_r) + \frac{1}{2}\tilde{\mathbf{S}}|\tilde{\mathbf{\Lambda}}|(R_l - R_r) \tag{4.87}$$

Using the definition $\vec{R} = \tilde{\mathbf{S}}^{-1}\mathcal{Q}$,

$$F_{i+\frac{1}{2}} = \frac{1}{2}\left[\tilde{\mathbf{S}}\tilde{\mathbf{\Lambda}}\tilde{\mathbf{S}}^{-1}(\mathcal{Q}_l + \mathcal{Q}_r) + \tilde{\mathbf{S}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{S}}^{-1}(\mathcal{Q}_l - \mathcal{Q}_r)\right] \tag{4.88}$$

then using (4.69),

$$F_{i+\frac{1}{2}} = \frac{1}{2}\left[\tilde{\mathbf{A}}(\mathcal{Q}_l + \mathcal{Q}_r) + \tilde{\mathbf{S}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{S}}^{-1}(\mathcal{Q}_l - \mathcal{Q}_r)\right] \tag{4.89}$$

and since $\mathcal{F} = \tilde{\mathbf{A}}\mathcal{Q}$, we can use (4.27) and (4.29) to write

$$F_{i+\frac{1}{2}} = \frac{1}{2}\left[F_l + F_r + \tilde{\mathbf{S}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{S}}^{-1}(\mathcal{Q}_l - \mathcal{Q}_r)\right] \tag{4.90}$$

where F_l and F_r are defined by the left or right state only. Approximating the flow variable vector, $\mathcal{Q}$, with the reconstruction scheme,

$$\mathcal{Q}_l = Q^l_{i+\frac{1}{2}} \tag{4.91}$$

$$\mathcal{Q}_r = Q^r_{i+\frac{1}{2}} \tag{4.92}$$

We can write,

$$F_{i+\frac{1}{2}} = \frac{1}{2}\left[F_l + F_r + \tilde{\mathbf{S}}|\tilde{\mathbf{\Lambda}}|\tilde{\mathbf{S}}^{-1}(Q^l_{i+\frac{1}{2}} - Q^r_{i+\frac{1}{2}})\right] \tag{4.93}$$

The same techniques can be applied to the interface at $x = x_{i-\frac{1}{2}}$. Thus we have developed and employed Roe's method for determining the flux, which depends on the left and right state of a cell interface, and the approximation matrix $\tilde{\mathbf{A}}$. The semi discrete Euler equations shown in (4.9) can now be solved.

The flux can be expressed in a form more conducive to writing an algorithm as [3]

$$F_{i+\frac{1}{2}} = \frac{1}{2}\left[F_l + F_r + \sum_{j=1}^{j=3} \alpha_j |\tilde{\lambda}| \tilde{e}_j\right] \tag{4.94}$$

where

$$\alpha_1 = \left[1 - \frac{(\gamma-1)\tilde{u}^2}{2\tilde{a}^2}\right]\Delta\rho + \left[(\gamma-1)\frac{\tilde{u}}{\tilde{a}^2}\right]\Delta\rho u + \left[\frac{(\gamma-1)}{\tilde{a}^2}\right]\Delta\rho e \tag{4.95}$$

$$\alpha_1 = \left[\frac{(\gamma-1)\tilde{u}^2}{4\tilde{a}^2} - \frac{\tilde{u}}{2\tilde{a}}\right]\Delta\rho + \left[\frac{1}{2\tilde{a}} - \frac{(\gamma-1)\tilde{u}}{2\tilde{a}^2}\right]\Delta\rho u + \left[\frac{(\gamma-1)}{2\tilde{a}^2}\right]\Delta\rho e \tag{4.96}$$

$$\alpha_1 = \left[\frac{(\gamma-1)\tilde{u}^2}{4\tilde{a}^2} + \frac{\tilde{u}}{2\tilde{a}}\right]\Delta\rho - \left[\frac{1}{2\tilde{a}} + \frac{(\gamma-1)\tilde{u}}{2\tilde{a}^2}\right]\Delta\rho u + \left[\frac{(\gamma-1)}{2\tilde{a}^2}\right]\Delta\rho e \tag{4.97}$$

In the development of a code which simulates Euler flow, the portion of the code in which the flux array is determined is the most computationally intensive. Roe's method described above must be applied at every cell face for each stage of the time integration technique at each time step, i.e, it will be performed twice per face per time step for a second order Runge-Kutta technique. This technique is described in the next chapter.

4.4 Solution in Time

This investigation makes use a two-stage Runge-Kutta technique. The technique makes use of an intermediate phase between time $t = t^n$ and $t = t^{n+1}$. The technique can be described mathematically as [3],

$$\begin{aligned} Q^0 &= Q^n \\ Q^1 &= Q^0 + \frac{\Delta t}{2} F^0 \\ Q^2 &= Q^0 + \frac{\Delta t}{2} F^1 \\ Q^{n+1} &= Q^2 \end{aligned} \tag{4.98}$$

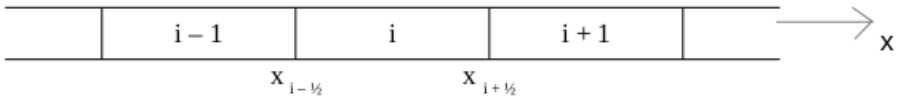


Figure 4.12: One dimensional grid of cells

In this description, F^0 represents some value (or set of values) which have been determined from Q^0. F^1 represents some value (or set of values) which have been determined from Q^1. So the value (or values) contained in Q^1 are not part of the solution to the Euler equations at any time step. They are just intermediate values used in stepping from a time where the solution is known at $t = t^n$ to the next time step at $t = t^{n+1}$. In this specific investigation, the intermediate values are used to determine the flux values at what can be thought of as the midpoint between time steps. The technique is second order accurate.

Some fundamental observations of the physical problem will be used to determine a reasonable time step. Given a grid with cells of length Δx, for the simulation to have any physical meaning, the time integration scheme must take steps forward in time which do not allow any portion of flow to pass completely through the cell without being accounted for, i.e, $\Delta x < u\Delta t$. This concept is easiest to visualize in a one dimensional grid, such as the one shown in figure 4.1, which is shown here.

The time step should be sized such that the flow state at $x_{i-\frac{1}{2}}$ at time t^n, which will be moving at the reconstructed velocity, u_i, does not reach $x_{i+\frac{1}{2}}$ before evaluating the next time step at $t = t^{n+1}$. If we define $\Delta t = t^{n+1} - t^n$, then for a wave speed of c, the wave will pass through a given cell in

$$\Delta t = \frac{\Delta x}{c} \tag{4.99}$$

In this simulation, we can generalize this condition over the entire grid as

$$\Delta t_{CFL} = min_i \left(min_j \frac{\Delta x}{|\tilde{\lambda}_j|} \right) \tag{4.100}$$

and

$$\Delta t = \mathcal{C} \Delta t_{CFL} \tag{4.101}$$

where $\tilde{\lambda}_j$ are the eigenvalues of the Roe matrix $\tilde{\mathbf{A}}$, and $0 < \mathcal{C} < 1$.. This is known as the Courant-Friedrichs-Lewis condition and is a necessary but not sufficient condition to ensure a convergent solution.

Chapter 5

Computational Techniques

The basic strategy used in solving for the flow can be summarized as:

1. *Build a spatial grid to approximate the physical problem.*
2. *Allocate space for and construct the arrays of dependent flow variables.*
3. *Runge-Kutta integration (repeated for every time step):*
    - *i) Apply the appropriate boundary conditions.*
    - *ii) Calculate the flux array for $t = t^n$.*
    - *iii) Use flux array to determine intermediate flow values.*
    - *iv) Calculate the flux array for $t = t^{n+1}$ from intermediate flow values.*
    - *v) Calculate the flow values for t^{n+1} and store in the dependent variable arrays.*
    - *vi) Repeat for desired number of time steps*
4. *Store and process results.*

An application written with traditional serial techniques would make use of a single core reading and writing to a large memory space where the flow variables are stored. Each phase of the code would have access to the entire memory space, and each subsequent phase would be aware of changes made by the previous phase. In a parallel structure however, a vast number of cores can be employed to read and write from/to the dependent variable memory space. The GPU hardware is limited in that there is a relatively small amount of memory shared by all cores employed in a given block. The hardware used in this investigation is limited to 48 KB of shared memory. This type of memory can be used virtually the same way as memory in serial applications, but for grids of any meaningful size, a larger memory space is required. Therefore, rather than store the flow variables in shared memory available for read/write by any core, the flow data is copied for each core, and each core then

operates in its own local memory space. This leads to the fundamental limitation of GPU hardware. Each core is unaware of any changes made by any other core. For this reason, data must be copied back to the CPU host and consolidated before taking any steps which require communication between cells. The costs of the memory transfers must be overcome by the benefits of the multi core execution.

For obvious physical reasons, the flow must be computed serially in time. For similarly obvious reasons, the flow at each stage in a Runge-Kutta scheme must be solved in a serial manner. The application is then optimized by reducing the computation time required to solve for the flow variables within each R-K phase within a time step. The most computationally intensive operation is the determination of the flux values at the cell interfaces, so parallelizing the flux algorithm was the focus of this investigation. The one dimensional flow described previously was applied to a three dimensional grid in space. This allows for the analytical verification of the code, while also providing data for practical grid sizes thus requiring computation times more representative of practical problems. The algorithms are written to accommodate flow in three directions, i.e., five dependent variables are tracked rather than just three, but the initial conditions are set such that the problem is reduced to flow in a single direction.

Observing figure 5.1, we can make some observations about the computations required. More specifically, observations can be made about the computational requirements within a R-K stage. For each cell interface, of which there are three per cell, an algorithm to determine the flux using Roe's method will have to be executed for five dependent variables. If we define the number of cells in each direction i, j, and k as N_i, N_j, and N_k, then a serial application will have to perform Roe's method to determine the flux a number of times equal to $3 \times N_i \times N_j \times N_K$ for each R-K stage of each time step. Roe's method required many operations for a single computation. For a grid of typical size, say 10^4 cells, Roe's method would now have to be performed 30,000 times per R-K stage. In this series of simulations, the number of cells in the i and j directions will be held constant, and the required computation times will be recorded as the number of cells in the k is increased the

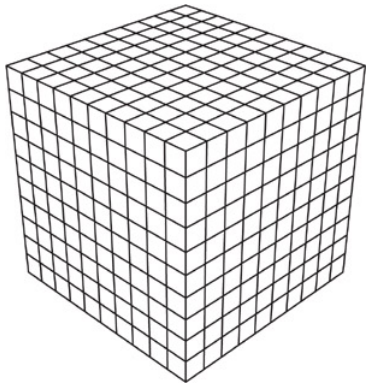


Figure 5.1: Hypothetical three dimensional grid

limits of the GPU hardware.

We can now look at a basic GPU architecture for potential advantages. The NVIDIA Tesla C2070 offers a grid-block-thread system. A user can simultaneously launch multiple "blocks" in a three dimensional "grid". From within each block an array of threads can then be launched. See figure 5.2.

For our purposes, it will be logical to launch one block per cell. Within each block, three threads can be launched to execute Roe's method in each direction. In total, we will have $(3 \times N_i \times N_j \times N_K)$ threads simultaneously executing the operations required by Roe's method to calculate flux at each face. The price we pay is the time required to transfer the memory from the host (CPU) to the device (GPU) and back again for both stages of the R-K scheme. Once the data is back on the host we perform the boundary condition operations. This simulation makes use of oscillating boundary conditions in the flow direction and symmetric boundary conditions in both of the other directions. The oscillation condition is achieved by assigning the values stored in the ghost cells on one edge of the grid to the real cells on the far side of the grid. In this simulation, N_i real cells were used and two ghost cells were used per side in all three directions. Figure 5.3 is a small one-dimensional analog. The data from the left most ghost cell on the right end is assigned the the left most real cell, and vice versa.

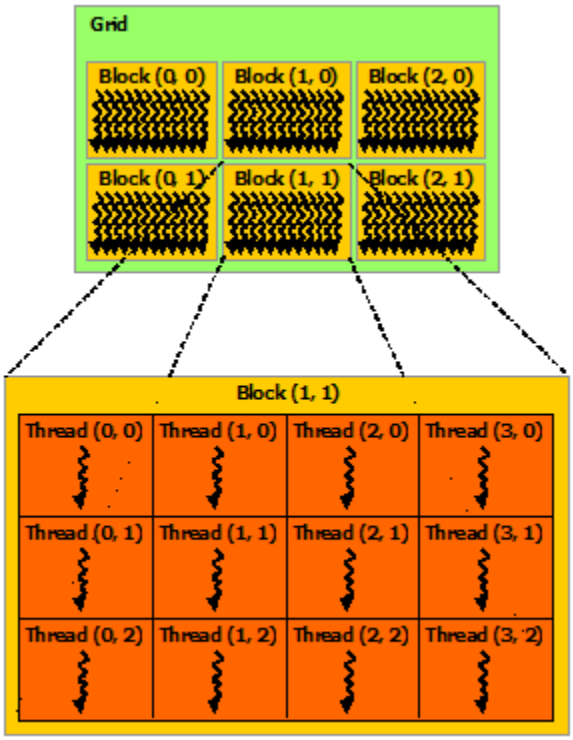


Figure 5.2: Hypothetical three dimensional grid [2]

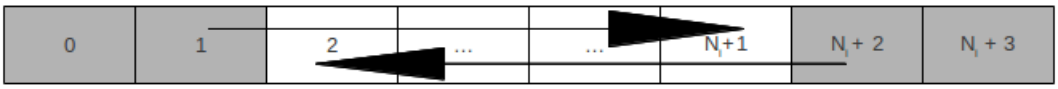


Figure 5.3: One-dimensional oscillating boundary condition

Chapter 6

Results Comparison

Two codes were generated. One was written in CUDA-C and made use of an NVIDIA Tesla C2070. The specifics of the hardware are detailed in Chapter 7.1. The other was simply modified to eliminate any GPU hardware functionality, and executed in a completely serial fashion. The front end CPU was an 800 MHz AMD Opteron 6176. This processor completed the entire serial computation as well as the serial portion of the GPU code. Figure 6.1 shows the crucial result. For our application and hardware, it appears that the GPU calculation does in fact reduce the required wall clock time as the grid size increases. Using the slopes of the lines in figure 6.1, we can say that the GPU code required .0024 seconds/cell, while the serial code required nearly double the amount of time per cell at .0044 seconds/cell. To see more of an improvement, more operations would have to be shifted to the GPU per memory transfer without increasing the size of the memory transfer.

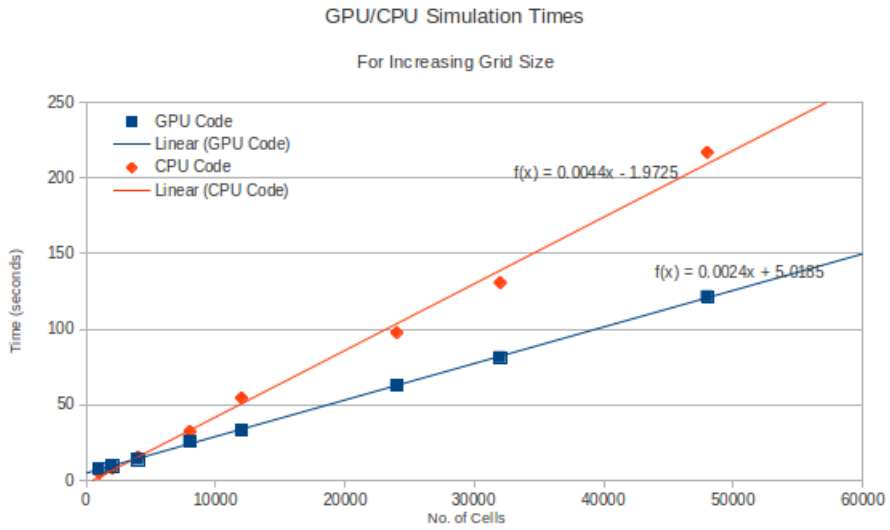


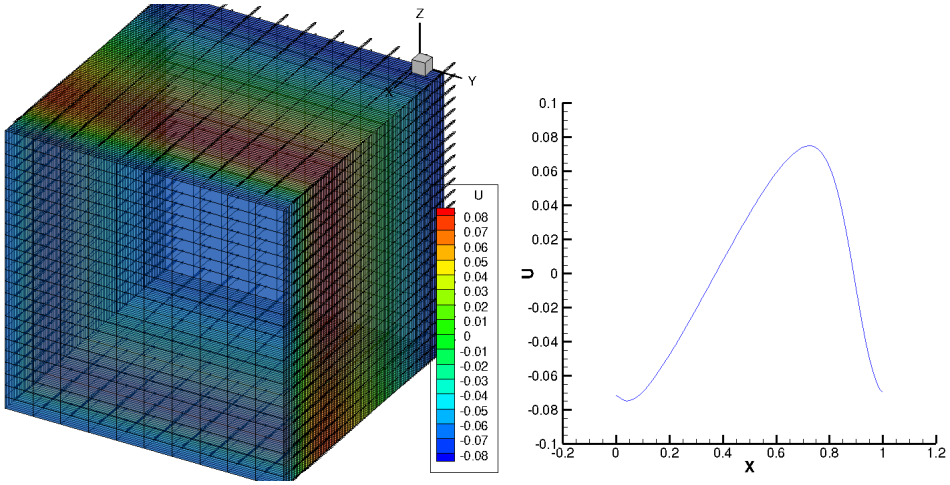
Figure 6.1: Simulation time comparison

					Wall Clock Time	
I cells	J cells	K cells	# Cells	Timesteps	GPU Code	CPU Code
100	10	1	1000	304	7.9	4.9
100	10	2	2000	304	9.6	8.1
100	10	4	4000	304	13.8	15.6
100	10	8	8000	304	26	32.5
100	10	12	12000	304	33.4	54.6
100	10	24	24000	304	62.8	97.7
100	10	32	32000	304	81.3	130.8
100	10	48	48000	304	121.5	217

Figure 6.2: Simulation time comparison data for varying grid sizes. Wall clock time is measured in seconds and each simulation is carried out to the same physical time just before shock formation.

The simulation was run for a constant number of cells in both the i and j directions. The maximum number of cells in any direction was held constant at 100 for all simulations, thus allowing the same number of time steps to be taken to reflect identical physical times.

Post-processed results for one of the grids are shown below.



(a) 3D post processed result

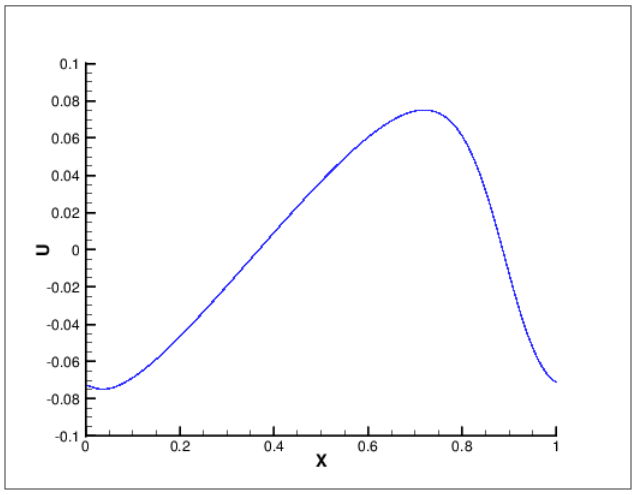
(b) u vs x

Figure 6.3: Velocity Profiles in x-direction as a function of non-dimensional position

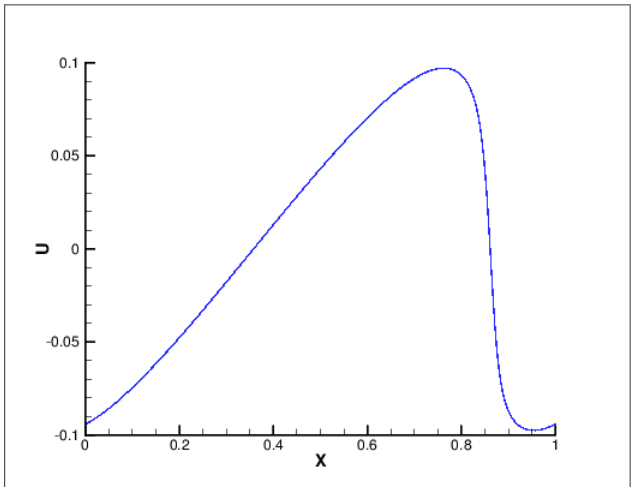
The accuracy of the simulation is first order, due to the reconstruction being first order. Figure 6.4 shows how the approximation improves as the grid becomes finer.

The code was also verified for accuracy for flow in the other two directions. The result for sinusoidal initial conditions and periodic boundary conditions are shown in Figure 6.5 and Figure 6.6.

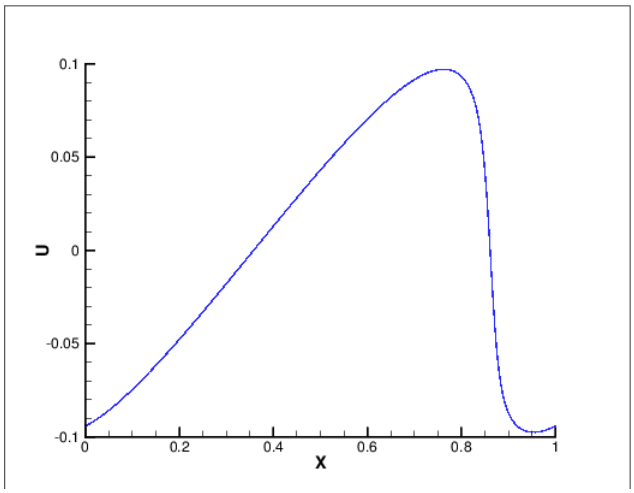
The results are encouraging for computational scientists. Even for an application with a relatively small portion of the code shifted to a GPU device, reductions in computation times can be achieved. As the improvements in individual CPU units begin to slow, and the focus switches to multi-processor hardware, users can tap into the massively parallel resources available in GPU hardware. The cost, scalability, and user-friendly features of compute-capable GPU units will only improve, which will expand the number of applications capable of realizing benefits from massively parallel strategies and GPU hardware.



(a) 100 cell result

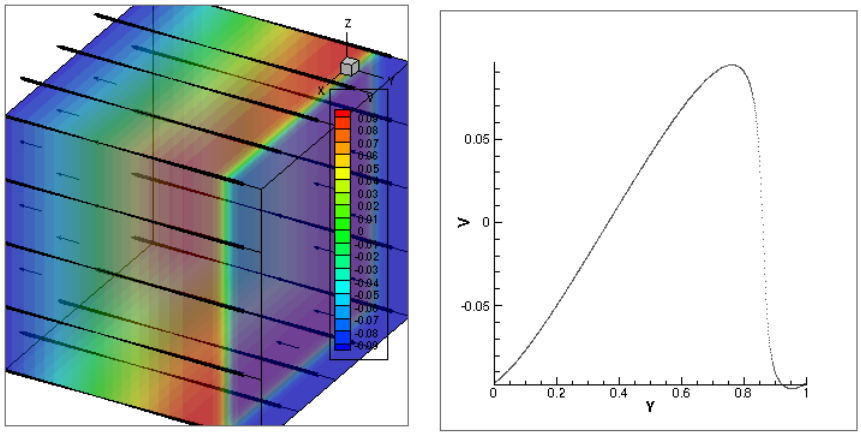


(b) 1000 cell result



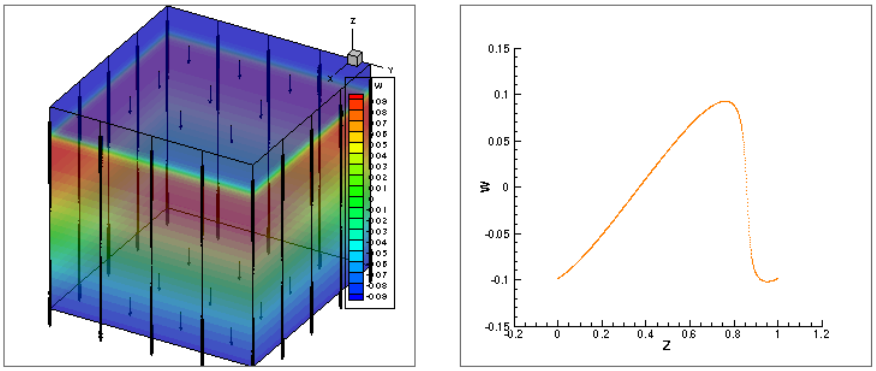
(c) Theoretical Result

Figure 6.4: Velocity profiles just before shock formation shown for an increasing number of cells in the flow direction



(a) 3D contour plot of y-direction flow (b) v vs y 1000 cell result

Figure 6.5: Velocity profiles in y-direction just before shock formation



(a) 3D contour plot of z-direction flow (b) w vs z 1000 cell result

Figure 6.6: Velocity profiles in z-direction just before shock formation

Chapter 7

Conclusions

All of this was undertaken to discover the potential of GPU hardware in a computational fluids application. Specifically, a supersonic, inviscid flow simulation using Roe's method for computing the flux was to be developed and executed NVIDIA Tesla 2070 graphics card. NVIDIA provides a modified C programming language called CUDA-C, which allows a user to transfer data to and from NVIDIA graphics hardware for parallel computation. The code was to simulate an initial sinusoidal velocity perturbation in space experiencing periodic boundary conditions. It was then to be validated against a known analytical solution. The parallel code could then be modified to execute in a serial fashion, and the wall clock time compared against the parallel version.

A code was successfully developed to simulate the supersonic, inviscid flow conditions described. The portion of the code computing Roe's method was off loaded to the GPU, and the code simultaneously computed Roe's method at every cell face. For the largest grid, the code simultaneously computed Roe's method on roughly 150,000 faces. The results indicate that the GPU code is capable of outperforming the serial code by up to 45%.

The code now is an excellent template for future investigations. It has been shown that Roe's method is a good candidate to be ported to the GPU, but there are other techniques which may prove worthy as well. It may be worthwhile to investigate higher order reconstruction, as the techniques can be computationally intensive. Also, if the bandwidth of the memory transfers to the GPU improves, the costs of transferring memory to a graphics device will decrease, making more applications good candidates for GPU execution. Even beyond fluid dynamics, the structure developed in this code can be used as a template for

transforming traditional serial code in a code which can tap into hundreds or thousands of processing cores available with graphics hardware.

Chapter 8

Appendices

8.1 Hardware Specifications

8.1.1 CPU Specifications

```
processor : 0
vendor_id : AuthenticAMD
cpu family  : 16
model   : 9
model name  : AMD Opteron(tm) Processor 6176
stepping  : 1
cpu MHz   : 800.000
cache size  : 512 KB
physical id : 0
siblings  : 12
core id   : 0
cpu cores : 12
apicid    : 16
initial apicid  : 0
fpu   : yes
fpu_exception : yes
cpuid level : 5
wp    : yes
flags   : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht
     syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm 3dnowext 3dnow constant_tsc rep_good nonstop_tsc extd_apicid
     amd_dcm pni monitor cx16 popcnt lahf_lm cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3
     dnowprefetch osvw ibs skinit wdt nodeid_msr arat
bogomips  : 4600.83
TLB size  : 1024 4K pages
clflush size  : 64
cache_alignment : 64
address sizes : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate
```

8.1.2 GPU Specifications

```
  --- General Information for device 0 ---
Name: Tesla C2070
Compute Capability: 2.0
Clock rate: 1147000
```

```
Device copy overlap: Enabled
Kernel execution timout: Disabled
  --- Memory Information for Device 0 ---
Total global memory: 1341587456
Total constant memory: 65536
Max mem pitch: 2147483647
Texture Alignment: 512
  ---Multi-Processor Information for device 0 ---
Multiprocessor count: 14
Shared mem per mp: 49152
Registers per mp: 32768
Threads in warp: 32
Max Threads per block: 1024
Max Thread Dimensions: (1024, 1024, 64)
Max grid dimensions: (65535, 65535, 65535)

  --- General Information for device 1 ---
Name: Tesla C2075
Compute Capability: 2.0
Clock rate: 1147000
Device copy overlap: Enabled
Kernel execution timout: Disabled
  --- Memory Information for Device 1 ---
Total global memory: 1341587456
Total constant memory: 65536
Max mem pitch: 2147483647
Texture Alignment: 512
  ---Multi-Processor Information for device 1 ---
Multiprocessor count: 14
Shared mem per mp: 49152
Registers per mp: 32768
Threads in warp: 32
Max Threads per block: 1024
Max Thread Dimensions: (1024, 1024, 64)
Max grid dimensions: (65535, 65535, 65535)
```

8.2 Serial Code

The serial code is broken into the following files:

```
allocate.cu     bc.cu           cidx.cu     dtensor1.cu
filopn.cu       flux.cu         fluxh.cu    free_dtensor1.cu
gidx.cu         input.cu        main.cu     output.cu
ran.cu          reconstruct.cu  rk2.cu      tecprep.cu
timestep.cu     unitv.cu
```

Which call upon the headers

```
defs.h
extern.h
function.h
global.h
```

The code is sequenced by the following makefile:

```
#
SHELL        = /bin/sh
CC      = nvcc
CLINKER      = $(CC)
OPTFLAGS     = -O0 -G -g
OPTFLAGSL    = -lm
#
a.out:  allocate.o \
  bc.o \
  cidx.o \
  dtensor1.o \
  filopn.o \
  flux.o \
  fluxh.o \
  free_dtensor1.o \
  gidx.o \
  input.o \
  main.o \
  output.o \
  ran.o \
  reconstruct.o \
  rk2.o \
  tecprep.o \
  timestep.o \
  unitv.o
  $(CLINKER) -o a.out \
  allocate.o \
  bc.o \
  cidx.o \
  dtensor1.o \
  filopn.o \
  flux.o \
  fluxh.o \
  free_dtensor1.o \
  gidx.o \
  input.o \
  main.o \
  output.o \
  ran.o \
  reconstruct.o \
  rk2.o \
  tecprep.o \
  timestep.o \
  unitv.o $(OPTFLAGSL)
allocate.o: defs.h extern.h global.h allocate.cu
  $(CC) $(OPTFLAGS) -c allocate.cu
bc.o: defs.h extern.h global.h bc.cu
  $(CC) $(OPTFLAGS) -c bc.cu
cidx.o: defs.h extern.h global.h cidx.cu
  $(CC) $(OPTFLAGS) -c cidx.cu
dtensor1.o: defs.h extern.h global.h dtensor1.cu
  $(CC) $(OPTFLAGS) -c dtensor1.cu
filopn.o: defs.h extern.h global.h filopn.cu
  $(CC) $(OPTFLAGS) -c filopn.cu
flux.o: defs.h extern.h global.h flux.cu
  $(CC) $(OPTFLAGS) -c flux.cu
fluxh.o: defs.h extern.h global.h fluxh.cu
```

```
58    $(CC) $(OPTFLAGS) -c fluxh.cu
59  free_dtensor1.o: defs.h extern.h global.h free_dtensor1.cu
60    $(CC) $(OPTFLAGS) -c free_dtensor1.cu
61  gidx.o: defs.h extern.h global.h gidx.cu
62    $(CC) $(OPTFLAGS) -c gidx.cu
63  input.o: defs.h extern.h global.h input.cu
64    $(CC) $(OPTFLAGS) -c input.cu
65  main.o: defs.h extern.h global.h main.cu
66    $(CC) $(OPTFLAGS) -c main.cu
67  output.o: defs.h extern.h global.h output.cu
68    $(CC) $(OPTFLAGS) -c output.cu
69  ran.o: defs.h extern.h global.h ran.cu
70    $(CC) $(OPTFLAGS) -c ran.cu
71  reconstruct.o: defs.h extern.h global.h reconstruct.cu
72    $(CC) $(OPTFLAGS) -c reconstruct.cu
73  rk2.o: defs.h extern.h global.h rk2.cu
74    $(CC) $(OPTFLAGS) -c rk2.cu
75  tecprep.o: defs.h extern.h global.h tecprep.cu
76    $(CC) $(OPTFLAGS) -c tecprep.cu
77  timestep.o: defs.h extern.h global.h timestep.cu
78    $(CC) $(OPTFLAGS) -c timestep.cu
79  unitv.o: defs.h extern.h global.h unitv.cu
80    $(CC) $(OPTFLAGS) -c unitv.cu
81  #
82  #   Notes:
83  #
84  #   1.  If the following message is received:
85  #
86  #       make: file `Makefile' line 9: Must be a separator (: or ::)
87  #             for rules (bu39)
88  #
89  # then the corresponding line did NOT begin with a tab.
90  #
```

The file *rstart* contains the grid and initial conditions which is generated by a separate code and not detailed here. The remaining files, *datain*, *dataou*, *rstore*, and *tecprep* are data files for initializing, storing and processing results.

8.2.1 allocate.cu

```
1   /*
2     function allocate
3
4     allocate storage for data on CPU
5
6   */
7   /*
8     includes
9   */
10  #include "defs.h"
11  #include "extern.h"
12  /*
13    externals
14  */
```

```
15
16 void allocate()
17 {
18   c = (struct cell*) malloc( (size_t)(d.nc*sizeof(struct cell))) ;
19   if ( c==NULL )
20   {
21     fputs("Cannot allocate memory for c[].  Stop.\n",stderr) ;
22     exit(1) ;
23   }
24   g = (struct grid*) malloc( (size_t)(d.nn*sizeof(struct grid))) ;
25   if ( g==NULL )
26   {
27     fputs("Cannot allocate memory for g[].  Stop.\n",stderr) ;
28     exit(1) ;
29   }
30
31 }
```

8.2.2 bc.cu

```
1  /*
2
3    function bc
4
5    updates ghost cells
6
7    21 jun 13  written
8
9    description
10
11     updates ghost cells using periodic boundary conditions
12
13     does not update the gh x gh x gh cube at each of the eight corners
14
15 */
16 /*
17   includes
18 */
19 #include "defs.h"
20 #include "extern.h"
21 /*
22   externals
23 */
24 extern int cidx(int, int, int, int, int, int) ;
25
26 void bc(int m)
27 {
28   int  gh, gh2, i, i1, i2, il, j, jl, k, kl ;
29
30   gh  = d.gh ;
31   gh2 = 2*d.gh ;
32   il  = d.il ;
33   jl  = d.jl ;
34   kl  = d.kl ;
35
36   /*  xi = 0  */
```

```
37
38  for ( k = gh ; k < gh + kl ; k++ )
39  {
40    for ( j = gh ; j < gh + jl ; j++ )
41    {
42      for ( i = 0 ; i < gh ; i++ )
43      {
44        i1 = cidx(i,j,k,il,jl,gh) ;
45        i2 = cidx(i+il,j,k,il,jl,gh) ;
46        c[i1].rho[m]  = c[i2].rho[m] ;
47        c[i1].rhou[m] = c[i2].rhou[m] ;
48        c[i1].rhov[m] = c[i2].rhov[m] ;
49        c[i1].rhow[m] = c[i2].rhow[m] ;
50        c[i1].rhoe[m] = c[i2].rhoe[m] ;
51      }
52    }
53  }
54
55  /*  xi = 1  */
56
57  for ( k = gh ; k < gh + kl ; k++ )
58  {
59    for ( j = gh ; j < gh + jl ; j++ )
60    {
61      for ( i = il + gh ; i < il + gh2 ; i++ )
62      {
63        i1 = cidx(i,j,k,il,jl,gh) ;
64        i2 = cidx(i-il,j,k,il,jl,gh) ;
65        c[i1].rho[m]  = c[i2].rho[m] ;
66        c[i1].rhou[m] = c[i2].rhou[m] ;
67        c[i1].rhov[m] = c[i2].rhov[m] ;
68        c[i1].rhow[m] = c[i2].rhow[m] ;
69        c[i1].rhoe[m] = c[i2].rhoe[m] ;
70      }
71    }
72  }
73
74  /*  eta = 0  */
75
76  for ( k = gh ; k < gh + kl ; k++ )
77  {
78    for ( i = gh ; i < gh + il ; i++ )
79    {
80      for ( j = 0 ; j < gh ; j++ )
81      {
82        i1 = cidx(i,j,k,il,jl,gh) ;
83        i2 = cidx(i,j+jl,k,il,jl,gh) ;
84        c[i1].rho[m]  = c[i2].rho[m] ;
85        c[i1].rhou[m] = c[i2].rhou[m] ;
86        c[i1].rhov[m] = c[i2].rhov[m] ;
87        c[i1].rhow[m] = c[i2].rhow[m] ;
88        c[i1].rhoe[m] = c[i2].rhoe[m] ;
89      }
90    }
91  }
92
93  /*  eta = 1  */
94
```

```
95    for ( k = gh ; k < gh + kl ; k++ )
96    {
97      for ( i = gh ; i < gh + il ; i++ )
98      {
99        for ( j = jl + gh ; j < jl + gh2 ; j++ )
100       {
101         i1 = cidx(i,j,k,il,jl,gh) ;
102         i2 = cidx(i,j-jl,k,il,jl,gh) ;
103         c[i1].rho[m]  = c[i2].rho[m] ;
104         c[i1].rhou[m] = c[i2].rhou[m] ;
105         c[i1].rhov[m] = c[i2].rhov[m] ;
106         c[i1].rhow[m] = c[i2].rhow[m] ;
107         c[i1].rhoe[m] = c[i2].rhoe[m] ;
108       }
109     }
110   }
111
112   /*  zeta = 0  */
113
114   for ( j = gh ; j < gh + jl ; j++ )
115   {
116     for ( i = gh ; i < gh + il ; i++ )
117     {
118       for ( k = 0 ; k < gh ; k++ )
119       {
120         i1 = cidx(i,j,k,il,jl,gh) ;
121         i2 = cidx(i,j,k+kl,il,jl,gh) ;
122         c[i1].rho[m]  = c[i2].rho[m] ;
123         c[i1].rhou[m] = c[i2].rhou[m] ;
124         c[i1].rhov[m] = c[i2].rhov[m] ;
125         c[i1].rhow[m] = c[i2].rhow[m] ;
126         c[i1].rhoe[m] = c[i2].rhoe[m] ;
127       }
128     }
129   }
130
131   /*  zeta = 1  */
132
133   for ( j = gh ; j < gh + jl ; j++ )
134   {
135     for ( i = gh ; i < gh + il ; i++ )
136     {
137       for ( k = kl + gh ; k < kl + gh2 ; k++ )
138       {
139         i1 = cidx(i,j,k,il,jl,gh) ;
140         i2 = cidx(i,j,k-kl,il,jl,gh) ;
141         c[i1].rho[m]  = c[i2].rho[m] ;
142         c[i1].rhou[m] = c[i2].rhou[m] ;
143         c[i1].rhov[m] = c[i2].rhov[m] ;
144         c[i1].rhow[m] = c[i2].rhow[m] ;
145         c[i1].rhoe[m] = c[i2].rhoe[m] ;
146       }
147     }
148   }
149
150 }
```

8.2.3 cidx.cu

```
/*
  computes index in array cell
*/
/*
  includes
*/
#include "defs.h"
#include "extern.h"
/*
  externals
*/

int cidx(int i, int j, int k, int il, int jl, int gh)
{
  int gh2 ;
  gh2 = 2*gh ;
  return (gh2+il)*(k*(gh2+jl)+j)+i ;
}
```

8.2.4 defs.h

```
/*
  definition of structure

  cell

    rho[2]          conservative variables at old [0] and new [1] level
    rhou[2]
    rhov[2]
    rhow[2]
    rhoe[2]

    flux[i][j]      flux on the ith face for the jth equation
                    where

                    i = 0: xi-face
                    i = 1: eta-face
                    i = 2: zeta-face

                    j = 0: mass
                    j = 1: x-momentum
                    j = 2: y-momentum
                    j = 3: z-momentum
                    j = 4: energy

    n[i][j]         unit normal on ith face in jth direction

                    i = 0: xi-face
                    i = 1: eta-face
                    i = 2: zeta-face

                    j = 0: x-direction
                    j = 1: y-direction
                    j = 2: z-direction
```

```
34
35      s[i][j]         unit vector in ith face in jth direction
36
37                      i = 0: xi-face
38                      i = 1: eta-face
39                      i = 2: zeta-face
40
41                      j = 0: x-direction
42                      j = 1: y-direction
43                      j = 2: z-direction
44
45      t[i][j]         unit vector in ith face in jth direction
46
47                      i = 0: xi-face
48                      i = 1: eta-face
49                      i = 2: zeta-face
50
51                      j = 0: x-direction
52                      j = 1: y-direction
53                      j = 2: z-direction
54
55      da[i]           area of ith face
56
57                      i = 0: xi-face
58                      i = 1: eta-face
59                      i = 2: zeta-face
60  */
61
62  /*
63     include statements for C
64  */
65  #include <stdio.h>
66  #include <stdlib.h>
67  #include <string.h>
68  #include <math.h>
69  #include <limits.h>
70  #include <stddef.h>
71  #include <sys/time.h>
72  #include <sys/resource.h>
73  #include <unistd.h>
74  #include <ctype.h>
75
76  /*
77     include statements for CUDA
78     The nvcc compiler recognizes these headers, but disregards for serial code
79  */
80  #include <cuda.h>
81  #include <driver_types.h>
82
83  /*
84     define statements
85  */
86  #define ARGS            1    /*  flag for argc  */
87  #define CONVERGE        1.0E-6
88  #define EPSILON         0.1  /*  harten correction  */
89  #define FILES           10
90  #define ITERMAX         100
91  #define OUTPUTPERLINE   5
```

```
92
93  #define ABS(X)        ((X) < 0 ? -(X) : (X))
94  #define MAX(X,Y)      (((X)>(Y))? (X):(Y))
95  #define MIN(X,Y)      (((X)<(Y))? (X):(Y))
96
97  /*
98     structures
99  */
100
101 struct cell {
102   double rho[2]  ;
103   double rhou[2] ;
104   double rhov[2] ;
105   double rhow[2] ;
106   double rhoe[2] ;
107   double flux[3][5] ;                 /*  fluxes                      */
108   double n[3][3] ;                    /*  unit normal to surface      */
109   double s[3][3] ;                    /*  unit vector in surface      */
110   double t[3][3] ;                    /*  unit vector in surface      */
111   double da[3] ;                      /*  area of surface             */
112   double inv ;                        /*  inverse of volume           */
113 } ;
114
115 struct data {
116   int check ;           /*  modulus for writing rstore file             */
117   int gh ;              /*  no. of ghost cells at each face             */
118   int il ;              /*  no. of cells in xi excluding ghost cells    */
119   int iter ;            /*  number of time steps                        */
120   int jl ;              /*  no. of cells in eta excluding ghost cells   */
121   int kl ;              /*  no. of cells in zeta excluding ghost cells  */
122   int nc ;              /*  total number of cells including ghost cells */
123   int nn ;              /*  total number of nodes including ghost cells */
124   int tint ;            /*  time integration (see above)                */
125
126   double cfl ;          /*  courant number                              */
127   double dt ;           /*  time step                                   */
128   double gamma ;        /*  ratio of specific heats                     */
129   double machinf ;      /*  mach number                                 */
130   double time ;         /*  accumulated physical time                   */
131 } ;
132
133 struct grid {
134   double x ;
135   double y ;
136   double z ;
137 } ;
138
139 struct point {
140   FILE *fp[FILES] ;     /* pointer to files */
141 } ;
```

8.2.5 dtensor1.cu

```
1  /*
2     function dtensor1
3
```

```
   allocates memory for first order tensor of type double

   17 Jan 98  Written

*/
/*
  includes
*/
#include "defs.h"
#include "extern.h"
/*
  externals
*/

double *dtensor1(int nrow)
{
   double *t ;

   /*  allocate pointers to rows  */

   t = (double *) malloc((size_t)(nrow*sizeof(double))) ;

   if ( t==NULL )
   {
     fprintf(stderr,"Error in dtensor1.\n") ;
      exit(0) ;
   }

   return t ;
}
```

8.2.6 extern.h

```
/*
  variables
*/
extern struct cell *c ;                 /* c denotes cell  */
extern struct data d ;                  /* d denotes data  */
extern struct grid *g ;                 /* g denotes grid  */
extern FILE *fp[] ;                     /* file pointers   */
```

8.2.7 filopn.cu

```
/*
  function filopn

  opens all files as indicated

*/
/*
  includes
*/
#include "defs.h"
#include "extern.h"
```

```
/*
  externals
*/

void filopn()
{
  char  error[160], filename[80] ;

  if ( FILES < 6 )
  {
    fputs("Value of FILES is too small.  Stop.\n",stderr) ;
    exit(0) ;
  }
  /* ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ */
  sprintf(filename,"datain") ;
  if  ((fp[0] = fopen(filename,"r")) == NULL)
  {
     sprintf(error,"Cannot open %s.  Stop.\n",filename) ;
     fputs(error,stderr) ;
     exit(0) ;
  }
  /* ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ */
  sprintf(filename,"dataou") ;
  if  ((fp[1] = fopen(filename,"w")) == NULL)
  {
     sprintf(error,"Cannot open %s.  Stop.\n",filename) ;
     fputs(error,stderr) ;
     exit(0) ;
  }
  /* ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ */
  sprintf(filename,"rstart") ;
  if  ((fp[2] = fopen(filename,"r")) == NULL)
  {
     sprintf(error,"Cannot open %s.  Stop.\n",filename) ;
     fputs(error,stderr) ;
     exit(0) ;
  }
  /* ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ */
  sprintf(filename,"rstore") ;
  if  ((fp[3] = fopen(filename,"w")) == NULL)
  {
     sprintf(error,"Cannot open %s.  Stop.\n",filename) ;
     fputs(error,stderr) ;
     exit(0) ;
  }
  /* ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ */
  sprintf(filename,"tecprep") ;
  if  ((fp[4] = fopen(filename,"w")) == NULL)
  {
     sprintf(error,"Cannot open %s.  Stop.\n",filename) ;
     fputs(error,stderr) ;
     exit(0) ;
  }
  /* ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ */

}
```

8.2.8 flux.cu

```
/*

  function flux

  computes inviscid flux CPU

  definitions

  nf  pointer to number of faces
  nfm   largest number of faces in xi-, eta- or zeta-directions

  da      area of face
  frho    flux for conservation of mass
  frhou   flux for conservation of x-momentum
  frhov   flux for conservation of y-momentum
  frhow   flux for conservation of z-momentum
  frhoe   flux for conservation of energy
  rhol    reconstructed density to left face
  rhor    reconstructed density to right face
  rhoul   reconstructed density*x-velocity to left face
  rhour   reconstructed density*x-velocity to right face
  rhovl   reconstructed density*y-velocity to left face
  rhovr   reconstructed density*y-velocity to right face
  rhowl   reconstructed density*z-velocity to left face
  rhowr   reconstructed density*z-velocity to right face
  rhoel   reconstructed density*energy to left face
  rhoer   reconstructed density*energy to right face
  nx      x-component of normal to the face
  ny      y-component of normal to the face
  nz      z-component of normal to the face
  sx      x-component of unit vector in face
  sy      y-component of unit vector in face
  sz      z-component of unit vector in face
  tx      x-component of unit vector in face
  ty      y-component of unit vector in face
  tz      z-component of unit vector in face



*/


/*
  includes
*/
#include "defs.h"
#include "extern.h"
/*
  externals
*/
extern int cidx(int, int, int, int, int, int) ;
extern double *dtensor1(int) ;
extern void fluxh(double *, double *, double *,
    double *, double *,double *, double *,
    double *, double *, double *,double *,
    double *, double *, double *, double *,double *,
```

```
    double *, double *, double *, double *,double *,
    double *, double *, double *, double *,
    int *, int, int, int, int, int) ;
extern void free_dtensor1(double *) ;
extern void reconstruct(double *, double *, double *, double *, double *,
                        double *, double *, double *, double *, double *,
                        double *, double *, double *, double *, double *,
                        double *, double *, double *, double *, double *,
                        int, int) ;
extern void free_dtensor1(double *);

void flux(int m, int step, int *nf, double *da, double *rhol, double *rhor, double *rhoul,
  double *rhour, double *rhovl, double *rhovr, double *rhowl, double *rhowr, double *rhoel,
  double *rhoer, double *nx, double *ny, double *nz, double *sx, double *sy,
  double *sz, double *tx, double *ty, double *tz, double *frho, double *frhou,
  double *frhov, double *frhow, double *frhoe)
{
  /*  host  */
  int     f, gh, i, ii, il, j, jl, jj, k, kl,
    mm, nfa, nfi, nfj, nfk, nfm, nft, nn ;

  mm  = m ;
  gh  = d.gh ;
  il  = d.il ;
  jl  = d.jl ;
  kl  = d.kl ;

  nfi = (il+1)*jl*kl ;
  nfj = il*(jl+1)*kl ;
  nfk = il*jl*(kl+1) ;
  nft = nfi + nfj +nfk ;
  nfm = MAX(nfi,nfj) ;
  nfm = MAX(nfm,nfk) ;
  nfa = 3*nfm ;




  for ( nn = 0 ; nn < 3 ; nn++ )
  {
    switch (nn)
    {
      case 0:
        nf[nn] = (il+1)*jl*kl ;
        break ;
      case 1:
        nf[nn] = il*(jl+1)*kl ;
        break ;
      case 2:
        nf[nn] = il*jl*(kl+1) ;
        break ;
      default:
        printf("Invalid value of nn in flux().\n") ;
        exit(1) ;
    }
    /*
       reconstruction
    */
```

```
115
116     reconstruct(da,rhol,rhor,rhoul,rhour,rhovl,
117                 rhovr,rhowl,rhowr,rhoel,rhoer,
118                 nx,ny,nz,sx,sy,sz,tx,ty,tz,mm,nn) ;
119   }
120
121
122
123
124     /*
125        compute fluxes on negative faces
126     */
127
128
129     fluxh(da, rhol,  rhor,  rhoul, rhour, rhovl,
130                     rhovr, rhowl, rhowr, rhoel, rhoer, nx,
131                     ny,    nz,    sx,    sy,    sz,    tx,
132                     ty,    tz,    frho,  frhou, frhov, frhow,
133                     frhoe, nf, mm, gh, il, jl, kl) ;
134
135
136     /*
137        save flux
138     */
139
140   for ( nn = 0 ; nn < 3 ; nn++)
141   {
142     switch (nn)
143     {
144       case 0:
145       {
146         jj = 0 ;
147         for ( k = gh ; k < gh + kl ; k++ )
148         {
149           for ( j = gh ; j < gh + jl ; j++ )
150           {
151             for ( i = gh ; i <= gh + il ; i++ )
152             {
153               ii               = cidx(i,j,k,il,jl,gh) ;
154               c[ii].flux[0][0] = frho[nn*nfm+jj] ;
155               c[ii].flux[0][1] = frhou[nn*nfm+jj] ;
156               c[ii].flux[0][2] = frhov[nn*nfm+jj] ;
157               c[ii].flux[0][3] = frhow[nn*nfm+jj] ;
158               c[ii].flux[0][4] = frhoe[nn*nfm+jj] ;
159               jj++ ;
160             }
161           }
162         }
163         break ;
164       }
165       case 1:
166       {
167         jj = 0 ;
168         for ( k = gh ; k < kl + gh ; k++ )
169         {
170           for ( i = gh ; i < il + gh ; i++ )
171           {
172             for ( j = gh ; j <= jl + gh ; j++ )
```

```
173             {
174               ii                = cidx(i,j,k,il,jl,gh) ;
175               c[ii].flux[1][0] = frho[nn*nfm+jj] ;
176               c[ii].flux[1][1] = frhou[nn*nfm+jj] ;
177               c[ii].flux[1][2] = frhov[nn*nfm+jj];
178               c[ii].flux[1][3] = frhow[nn*nfm+jj] ;
179               c[ii].flux[1][4] = frhoe[nn*nfm+jj] ;
180               jj++ ;
181             }
182           }
183         }
184         break ;
185       }
186       case 2:
187       {
188         jj = 0 ;
189         for ( j = gh ; j < jl + gh ; j++ )
190         {
191           for ( i = gh ; i < il + gh ; i++ )
192           {
193             for ( k = gh ; k <= kl + gh ; k++ )
194             {
195               ii                = cidx(i,j,k,il,jl,gh) ;
196               c[ii].flux[2][0] = frho[nn*nfm+jj] ;
197               c[ii].flux[2][1] = frhou[nn*nfm+jj] ;
198               c[ii].flux[2][2] = frhov[nn*nfm+jj] ;
199               c[ii].flux[2][3] = frhow[nn*nfm+jj] ;
200               c[ii].flux[2][4] = frhoe[nn*nfm+jj] ;
201               jj++ ;
202             }
203           }
204         }
205         break ;
206       }
207       default:
208       {
209         printf("Invalid value of nn in flux().\n") ;
210         exit(1) ;
211       }
212     }
213   }
214 }
```

8.2.9 fluxh.cu

```
1  /*
2
3    function fluxh
4
5    computes flux on CPU
6
7    definitions
8
9     hl, hr    total enthalpy at left and right faces
10    ul, ur    ubar at left and right faces
11    vl, vr    vbar at left and right faces
```

```
12      wl, wr    wbar at left and right faces
13      pl, pr    pressure at left and right faces
14      qql, qqr  kinetic energy at left and right faces
15      rl, rr    density at left and right faces
16      rli, rri  inverse of density at left and right faces
17      rel, rer  rhoe at left and right faces
18
19      at        atilda
20      at2       atilda*atilda
21      ht        htilda
22      lt[]      abs(lambdatilda[])
23      qt        0.5*(ut*ut+vt*vt+wt*wt)
24      rm[][]    Roe matrix
25      sqrl      sqrt(rl)
26      sqrr      sqrt(rr)
27      ut,vt,wt  ubartilda, vbartilda, wbartilda
28
29      dr[]      delta R[]
30
31      left face    the face whose outwards normal points in the negative
32                   direction of the transformed coordinate (e.g., negative xi)
33      right face   the face whose outwards normal points in the positive
34                   direction of the transformed coordinate (e.g., positive xi)
35
36
37  */
38  /*
39    includes
40  */
41  #include "defs.h"
42  #include "extern.h"
43
44
45   int findex(int nn, int i, int nfm)
46  {
47    return nn*nfm + i ;
48  }
49
50  int cidx(int, int, int, int, int, int) ;
51
52
53  void fluxh(double *da, double *rhol, double *rhor,
54                         double *rhoul, double *rhour, double *rhovl,
55                         double *rhovr, double *rhowl, double *rhowr,
56                         double *rhoel, double *rhoer, double *nx,
57                         double *ny, double *nz, double *sx,
58                         double *sy, double *sz, double *tx,
59                         double *ty, double *tz, double *frho,
60                         double *frhou, double *frhov, double *frhow,
61                         double *frhoe, int *nf, int mm, int gh, int il,
62            int jl, int kl)
63  {
64    int    i, ii, i1, i2, j, jj, k, nn, nfi, nfj, nfk, nft, nfm, nfa, nfr ;
65
66    double at, at2, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, dr[5],
67           gamma, h[5], hl, hr, ht, lt[5], pl, pr, qql, qqr, qt,
68           rel, rer, rl, rli, rm[5][5], rr, rri, sqrl, sqrr, ul,
69           ur, ut, vl, vr, vt, wl, wr, wt ;
```

```
70
71    nfi = (il+1)*jl*kl ;
72    nfj = il*(jl+1)*kl ;
73    nfk = il*jl*(kl+1) ;
74    nft = nfi + nfj +nfk ;
75    nfm = MAX(nfi,nfj) ;
76    nfm = MAX(nfm,nfk) ;
77    nfa = 3*nfm ;
78
79
80
81
82  for ( nn = 0 ; nn < 3 ; nn++){
83
84  switch (nn)
85   {
86    case 0:
87    {
88
89  jj = 0 ;
90   for ( k = gh ; k < kl + gh ; k++ ){
91    for ( j = gh ; j < jl + gh ; j++ ){
92     for ( i = gh ; i <= il + gh ; i++ ){
93
94
95
96    gamma = 1.4 ;
97
98      ii  = findex(nn, jj, nfm) ;
99      rl  = rhol[ii] ;
100     rli = 1./rl ;
101     ul  = rli*(rhoul[ii]*nx[ii] + rhovl[ii]*ny[ii] + rhowl[ii]*nz[ii]) ;
102     vl  = rli*(rhoul[ii]*sx[ii] + rhovl[ii]*sy[ii] + rhowl[ii]*sz[ii]) ;
103     wl  = rli*(rhoul[ii]*tx[ii] + rhovl[ii]*ty[ii] + rhowl[ii]*tz[ii]) ;
104     rel = rhoel[ii] ;
105
106     qql = 0.5*rli*rli*(rhoul[ii]*rhoul[ii]+rhovl[ii]*rhovl[ii]+
107                        rhowl[ii]*rhowl[ii]) ;
108     pl  = (gamma-1.0)*(rel-rl*qql) ;
109     hl  = (gamma/(gamma-1.0))*pl*rli + qql ;
110
111     rr  = rhor[ii] ;
112     rri = 1./rr ;
113     ur  = rri*(rhour[ii]*nx[ii] + rhovr[ii]*ny[ii] + rhowr[ii]*nz[ii]) ;
114     vr  = rri*(rhour[ii]*sx[ii] + rhovr[ii]*sy[ii] + rhowr[ii]*sz[ii]) ;
115     wr  = rri*(rhour[ii]*tx[ii] + rhovr[ii]*ty[ii] + rhowr[ii]*tz[ii]) ;
116     rer = rhoer[ii] ;
117
118     qqr = 0.5*rri*rri*(rhour[ii]*rhour[ii]+rhovr[ii]*rhovr[ii]+
119                        rhowr[ii]*rhowr[ii]) ;
120     pr  = (gamma-1.0)*(rer-rr*qqr) ;
121     hr  = (gamma/(gamma-1.0))*pr*rri + qqr ;
122
123     sqrl = sqrt(fabs(rl)) ;
124     sqrr = sqrt(fabs(rr)) ;
125
126     c1  = 1./(sqrl+sqrr) ;
127     ut  = c1*(sqrl*ul+sqrr*ur) ;
```

```
128     vt  = c1*(sqrl*vl+sqrr*vr) ;
129     wt  = c1*(sqrl*wl+sqrr*wr) ;
130     qt  = 0.5*(ut*ut+vt*vt+wt*wt) ;
131     ht  = c1*(sqrl*hl+sqrr*hr) ;
132     at2 = (gamma-1.)*(ht-qt) ;
133     at  = sqrt(fabs(at2)) ;
134
135     lt[0] = fabs(ut) ;
136     lt[1] = lt[0] ;
137     lt[2] = lt[0] ;
138     lt[3] = fabs(ut+at) ;
139     lt[4] = fabs(ut-at) ;
140
141     /*  harten correction  */
142
143     for ( i1 = 0 ; i1 < 5 ; i1++ )
144     {
145       if ( lt[i1] < 2.0*EPSILON*at )
146       {
147         lt[i1] = lt[i1]*lt[i1]/(4.0*EPSILON*at) + EPSILON*at ;
148       }
149     }
150
151     /*  roe matrix  */
152
153     /*  note that the indices for lt and rm are one less than in the notes  */
154
155     c1       = (gamma-1.)/(2.*at2) ;
156     c2       = lt[3] + lt[4] - 2.*lt[2] ;
157     c3       = ut/(2.*at) ;
158     c4       = 1./(2.*at) ;
159     rm[0][0] = lt[2] + c1*qt*c2 + c3*(lt[4]-lt[3]) ;
160     rm[0][1] = -c1*ut*c2 - c4*(lt[4]-lt[3]) ;
161     rm[0][2] = -c1*vt*c2 ;
162     rm[0][3] = -c1*wt*c2 ;
163     rm[0][4] = c1*c2 ;
164
165     c5       = (ut+at)*lt[3] + (ut-at)*lt[4] - 2.*ut*lt[2] ;
166     c6       = (ut-at)*lt[4] - (ut+at)*lt[3] ;
167     rm[1][0] = ut*lt[2] + c1*qt*c5 + c3*c6 ;
168     rm[1][1] = -(c1*ut*c5 + c4*c6) ;
169     rm[1][2] = -c1*vt*c5 ;
170     rm[1][3] = -c1*wt*c5 ;
171     rm[1][4] =  c1*c5 ;
172
173     rm[2][0] = vt*(lt[2]-lt[0] + c1*qt*c2 + c3*(lt[4]-lt[3])) ;
174     rm[2][1] = vt*(-c1*ut*c2 + c4*(lt[3]-lt[4])) ;
175     rm[2][2] = lt[0] - c1*vt*vt*c2 ;
176     rm[2][3] = -c1*vt*wt*c2 ;
177     rm[2][4] = c1*vt*c2 ;
178
179     rm[3][0] = wt*(lt[2]-lt[1] + c1*qt*c2 + c3*(lt[4]-lt[3])) ;
180     rm[3][1] = wt*(-c1*ut*c2 + c4*(lt[3]-lt[4])) ;
181     rm[3][2] = -c1*vt*wt*c2 ;
182     rm[3][3] = lt[1] - c1*wt*wt*c2 ;
183     rm[3][4] = c1*wt*c2 ;
184
185     c7       = ht + ut*at ;
```

```
186     c8       = ht - ut*at ;
187     c9       = c7*lt[3] + c8*lt[4] - 2.*qt*lt[2] ;
188     c10      = c7*lt[3] - c8*lt[4] ;
189     rm[4][0] = c1*qt*c9 - c3*c10 + qt*lt[2] - vt*vt*lt[0] - wt*wt*lt[1] ;
190     rm[4][1] = -c1*ut*c9 + c4*c10 ;
191     rm[4][2] = vt*(-c1*c9 + lt[0]) ;
192     rm[4][3] = wt*(-c1*c9 + lt[1]) ;
193     rm[4][4] = c1*c9 ;
194
195     dr[0] = rl - rr ;
196     dr[1] = rl*ul - rr*ur ;
197     dr[2] = rl*vl - rr*vr ;
198     dr[3] = rl*wl - rr*wr ;
199     dr[4] = rel - rer ;
200
201     for (i1 = 0 ; i1 < 5 ; i1++ )
202     {
203       h[i1] = 0. ;
204       for ( i2 = 0 ; i2 < 5 ; i2++ )  h[i1] += rm[i1][i2]*dr[i2] ;
205     }
206
207     h[0] = 0.5*(rl*ul + rr*ur + h[0]) ;
208     h[1] = 0.5*(rl*ul*ul + pl + rr*ur*ur + pr + h[1]) ;
209     h[2] = 0.5*(rl*vl*ul + rr*vr*ur + h[2]) ;
210     h[3] = 0.5*(rl*wl*ul + rr*wr*ur + h[3]) ;
211     h[4] = 0.5*((rel+pl)*ul + (rer+pr)*ur + h[4]) ;
212
213
214     frho[ii]  = h[0]*da[ii] ;
215     frhou[ii] = (h[1]*nx[ii] + h[2]*sx[ii] + h[3]*tx[ii])*da[ii] ;
216     frhov[ii] = (h[1]*ny[ii] + h[2]*sy[ii] + h[3]*ty[ii])*da[ii] ;
217     frhow[ii] = (h[1]*nz[ii] + h[2]*sz[ii] + h[3]*tz[ii])*da[ii] ;
218     frhoe[ii] = h[4]*da[ii] ;
219    jj++;
220    }
221   }
222  }
223 }
224
225   case 1:
226   {
227 jj = 0 ;
228  for ( k = gh ; k < kl + gh ; k++ ){
229   for ( i = gh ; i < il + gh ; i++ ){
230    for ( j = gh ; j <= jl + gh ; j++ ){
231
232
233
234   gamma = 1.4 ;
235
236     ii  = findex(nn, jj, nfm) ;
237     rl  = rhol[ii] ;
238     rli = 1./rl ;
239     ul  = rli*(rhoul[ii]*nx[ii] + rhovl[ii]*ny[ii] + rhowl[ii]*nz[ii]) ;
240     vl  = rli*(rhoul[ii]*sx[ii] + rhovl[ii]*sy[ii] + rhowl[ii]*sz[ii]) ;
241     wl  = rli*(rhoul[ii]*tx[ii] + rhovl[ii]*ty[ii] + rhowl[ii]*tz[ii]) ;
242     rel = rhoel[ii] ;
243
```

```
244     qql = 0.5*rli*rli*(rhoul[ii]*rhoul[ii]+rhovl[ii]*rhovl[ii]+
245                        rhowl[ii]*rhowl[ii]) ;
246     pl  = (gamma-1.0)*(rel-rl*qql) ;
247     hl  = (gamma/(gamma-1.0))*pl*rli + qql ;
248
249     rr  = rhor[ii] ;
250     rri = 1./rr ;
251     ur  = rri*(rhour[ii]*nx[ii] + rhovr[ii]*ny[ii] + rhowr[ii]*nz[ii]) ;
252     vr  = rri*(rhour[ii]*sx[ii] + rhovr[ii]*sy[ii] + rhowr[ii]*sz[ii]) ;
253     wr  = rri*(rhour[ii]*tx[ii] + rhovr[ii]*ty[ii] + rhowr[ii]*tz[ii]) ;
254     rer = rhoer[ii] ;
255
256     qqr = 0.5*rri*rri*(rhour[ii]*rhour[ii]+rhovr[ii]*rhovr[ii]+
257                        rhowr[ii]*rhowr[ii]) ;
258     pr  = (gamma-1.0)*(rer-rr*qqr) ;
259     hr  = (gamma/(gamma-1.0))*pr*rri + qqr ;
260
261     sqrl = sqrt(fabs(rl)) ;
262     sqrr = sqrt(fabs(rr)) ;
263
264     c1  = 1./(sqrl+sqrr) ;
265     ut  = c1*(sqrl*ul+sqrr*ur) ;
266     vt  = c1*(sqrl*vl+sqrr*vr) ;
267     wt  = c1*(sqrl*wl+sqrr*wr) ;
268     qt  = 0.5*(ut*ut+vt*vt+wt*wt) ;
269     ht  = c1*(sqrl*hl+sqrr*hr) ;
270     at2 = (gamma-1.)*(ht-qt) ;
271     at  = sqrt(fabs(at2)) ;
272
273     lt[0] = fabs(ut) ;
274     lt[1] = lt[0] ;
275     lt[2] = lt[0] ;
276     lt[3] = fabs(ut+at) ;
277     lt[4] = fabs(ut-at) ;
278
279     /*  harten correction  */
280
281     for ( i1 = 0 ; i1 < 5 ; i1++ )
282     {
283       if ( lt[i1] < 2.0*EPSILON*at )
284       {
285         lt[i1] = lt[i1]*lt[i1]/(4.0*EPSILON*at) + EPSILON*at ;
286       }
287     }
288
289     /*  roe matrix  */
290
291     /*  note that the indices for lt and rm are one less than in the notes  */
292
293     c1       = (gamma-1.)/(2.*at2) ;
294     c2       = lt[3] + lt[4] - 2.*lt[2] ;
295     c3       = ut/(2.*at) ;
296     c4       = 1./(2.*at) ;
297     rm[0][0] = lt[2] + c1*qt*c2 + c3*(lt[4]-lt[3]) ;
298     rm[0][1] = -c1*ut*c2 - c4*(lt[4]-lt[3]) ;
299     rm[0][2] = -c1*vt*c2 ;
300     rm[0][3] = -c1*wt*c2 ;
301     rm[0][4] = c1*c2 ;
```

```
302
303     c5       = (ut+at)*lt[3] + (ut-at)*lt[4] - 2.*ut*lt[2] ;
304     c6       = (ut-at)*lt[4] - (ut+at)*lt[3] ;
305     rm[1][0] = ut*lt[2] + c1*qt*c5 + c3*c6 ;
306     rm[1][1] = -(c1*ut*c5 + c4*c6) ;
307     rm[1][2] = -c1*vt*c5 ;
308     rm[1][3] = -c1*wt*c5 ;
309     rm[1][4] =  c1*c5 ;
310
311     rm[2][0] = vt*(lt[2]-lt[0] + c1*qt*c2 + c3*(lt[4]-lt[3])) ;
312     rm[2][1] = vt*(-c1*ut*c2 + c4*(lt[3]-lt[4])) ;
313     rm[2][2] = lt[0] - c1*vt*vt*c2 ;
314     rm[2][3] = -c1*vt*wt*c2 ;
315     rm[2][4] = c1*vt*c2 ;
316
317     rm[3][0] = wt*(lt[2]-lt[1] + c1*qt*c2 + c3*(lt[4]-lt[3])) ;
318     rm[3][1] = wt*(-c1*ut*c2 + c4*(lt[3]-lt[4])) ;
319     rm[3][2] = -c1*vt*wt*c2 ;
320     rm[3][3] = lt[1] - c1*wt*wt*c2 ;
321     rm[3][4] = c1*wt*c2 ;
322
323     c7       = ht + ut*at ;
324     c8       = ht - ut*at ;
325     c9       = c7*lt[3] + c8*lt[4] - 2.*qt*lt[2] ;
326     c10      = c7*lt[3] - c8*lt[4] ;
327     rm[4][0] = c1*qt*c9 - c3*c10 + qt*lt[2] - vt*vt*lt[0] - wt*wt*lt[1] ;
328     rm[4][1] = -c1*ut*c9 + c4*c10 ;
329     rm[4][2] = vt*(-c1*c9 + lt[0]) ;
330     rm[4][3] = wt*(-c1*c9 + lt[1]) ;
331     rm[4][4] = c1*c9 ;
332
333     dr[0] = rl - rr ;
334     dr[1] = rl*ul - rr*ur ;
335     dr[2] = rl*vl - rr*vr ;
336     dr[3] = rl*wl - rr*wr ;
337     dr[4] = rel - rer ;
338
339     for (i1 = 0 ; i1 < 5 ; i1++ )
340     {
341       h[i1] = 0. ;
342       for ( i2 = 0 ; i2 < 5 ; i2++ )  h[i1] += rm[i1][i2]*dr[i2] ;
343     }
344
345     h[0] = 0.5*(rl*ul + rr*ur + h[0]) ;
346     h[1] = 0.5*(rl*ul*ul + pl + rr*ur*ur + pr + h[1]) ;
347     h[2] = 0.5*(rl*vl*ul + rr*vr*ur + h[2]) ;
348     h[3] = 0.5*(rl*wl*ul + rr*wr*ur + h[3]) ;
349     h[4] = 0.5*((rel+pl)*ul + (rer+pr)*ur + h[4]) ;
350
351
352     frho[ii]  = h[0]*da[ii] ;
353     frhou[ii] = (h[1]*nx[ii] + h[2]*sx[ii] + h[3]*tx[ii])*da[ii] ;
354     frhov[ii] = (h[1]*ny[ii] + h[2]*sy[ii] + h[3]*ty[ii])*da[ii] ;
355     frhow[ii] = (h[1]*nz[ii] + h[2]*sz[ii] + h[3]*tz[ii])*da[ii] ;
356     frhoe[ii] = h[4]*da[ii] ;
357    jj++;
358    }
359   }
```

```
 }
}

  case 2:
  {
jj = 0 ;
 for ( j = gh ; j < jl + gh ; j++ ){
  for ( i = gh ; i < il + gh ; i++ ){
    for ( k = gh ; k <= kl + gh ; k++ ){


  gamma = 1.4 ;

    ii  = findex(nn, jj, nfm) ;
    rl  = rhol[ii] ;
    rli = 1./rl ;
    ul  = rli*(rhoul[ii]*nx[ii] + rhovl[ii]*ny[ii] + rhowl[ii]*nz[ii]) ;
    vl  = rli*(rhoul[ii]*sx[ii] + rhovl[ii]*sy[ii] + rhowl[ii]*sz[ii]) ;
    wl  = rli*(rhoul[ii]*tx[ii] + rhovl[ii]*ty[ii] + rhowl[ii]*tz[ii]) ;
    rel = rhoel[ii] ;

    qql = 0.5*rli*rli*(rhoul[ii]*rhoul[ii]+rhovl[ii]*rhovl[ii]+
                       rhowl[ii]*rhowl[ii]) ;
    pl  = (gamma-1.0)*(rel-rl*qql) ;
    hl  = (gamma/(gamma-1.0))*pl*rli + qql ;

    rr  = rhor[ii] ;
    rri = 1./rr ;
    ur  = rri*(rhour[ii]*nx[ii] + rhovr[ii]*ny[ii] + rhowr[ii]*nz[ii]) ;
    vr  = rri*(rhour[ii]*sx[ii] + rhovr[ii]*sy[ii] + rhowr[ii]*sz[ii]) ;
    wr  = rri*(rhour[ii]*tx[ii] + rhovr[ii]*ty[ii] + rhowr[ii]*tz[ii]) ;
    rer = rhoer[ii] ;

    qqr = 0.5*rri*rri*(rhour[ii]*rhour[ii]+rhovr[ii]*rhovr[ii]+
                       rhowr[ii]*rhowr[ii]) ;
    pr  = (gamma-1.0)*(rer-rr*qqr) ;
    hr  = (gamma/(gamma-1.0))*pr*rri + qqr ;

    sqrl = sqrt(fabs(rl)) ;
    sqrr = sqrt(fabs(rr)) ;

    c1  = 1./(sqrl+sqrr) ;
    ut  = c1*(sqrl*ul+sqrr*ur) ;
    vt  = c1*(sqrl*vl+sqrr*vr) ;
    wt  = c1*(sqrl*wl+sqrr*wr) ;
    qt  = 0.5*(ut*ut+vt*vt+wt*wt) ;
    ht  = c1*(sqrl*hl+sqrr*hr) ;
    at2 = (gamma-1.)*(ht-qt) ;
    at  = sqrt(fabs(at2)) ;

    lt[0] = fabs(ut) ;
    lt[1] = lt[0] ;
    lt[2] = lt[0] ;
    lt[3] = fabs(ut+at) ;
    lt[4] = fabs(ut-at) ;

    /*  harten correction  */

```

```
418     for ( i1 = 0 ; i1 < 5 ; i1++ )
419     {
420       if ( lt[i1] < 2.0*EPSILON*at )
421       {
422         lt[i1] = lt[i1]*lt[i1]/(4.0*EPSILON*at) + EPSILON*at ;
423       }
424     }
425
426     /*  roe matrix  */
427
428     /*  note that the indices for lt and rm are one less than in the notes  */
429
430     c1       = (gamma-1.)/(2.*at2) ;
431     c2       = lt[3] + lt[4] - 2.*lt[2] ;
432     c3       = ut/(2.*at) ;
433     c4       = 1./(2.*at) ;
434     rm[0][0] = lt[2] + c1*qt*c2 + c3*(lt[4]-lt[3]) ;
435     rm[0][1] = -c1*ut*c2 - c4*(lt[4]-lt[3]) ;
436     rm[0][2] = -c1*vt*c2 ;
437     rm[0][3] = -c1*wt*c2 ;
438     rm[0][4] = c1*c2 ;
439
440     c5       = (ut+at)*lt[3] + (ut-at)*lt[4] - 2.*ut*lt[2] ;
441     c6       = (ut-at)*lt[4] - (ut+at)*lt[3] ;
442     rm[1][0] = ut*lt[2] + c1*qt*c5 + c3*c6 ;
443     rm[1][1] = -(c1*ut*c5 + c4*c6) ;
444     rm[1][2] = -c1*vt*c5 ;
445     rm[1][3] = -c1*wt*c5 ;
446     rm[1][4] =  c1*c5 ;
447
448     rm[2][0] = vt*(lt[2]-lt[0] + c1*qt*c2 + c3*(lt[4]-lt[3])) ;
449     rm[2][1] = vt*(-c1*ut*c2 + c4*(lt[3]-lt[4])) ;
450     rm[2][2] = lt[0] - c1*vt*vt*c2 ;
451     rm[2][3] = -c1*vt*wt*c2 ;
452     rm[2][4] = c1*vt*c2 ;
453
454     rm[3][0] = wt*(lt[2]-lt[1] + c1*qt*c2 + c3*(lt[4]-lt[3])) ;
455     rm[3][1] = wt*(-c1*ut*c2 + c4*(lt[3]-lt[4])) ;
456     rm[3][2] = -c1*vt*wt*c2 ;
457     rm[3][3] = lt[1] - c1*wt*wt*c2 ;
458     rm[3][4] = c1*wt*c2 ;
459
460     c7       = ht + ut*at ;
461     c8       = ht - ut*at ;
462     c9       = c7*lt[3] + c8*lt[4] - 2.*qt*lt[2] ;
463     c10      = c7*lt[3] - c8*lt[4] ;
464     rm[4][0] = c1*qt*c9 - c3*c10 + qt*lt[2] - vt*vt*lt[0] - wt*wt*lt[1] ;
465     rm[4][1] = -c1*ut*c9 + c4*c10 ;
466     rm[4][2] = vt*(-c1*c9 + lt[0]) ;
467     rm[4][3] = wt*(-c1*c9 + lt[1]) ;
468     rm[4][4] = c1*c9 ;
469
470     dr[0] = rl - rr ;
471     dr[1] = rl*ul - rr*ur ;
472     dr[2] = rl*vl - rr*vr ;
473     dr[3] = rl*wl - rr*wr ;
474     dr[4] = rel - rer ;
475
```

```
476     for (i1 = 0 ; i1 < 5 ; i1++ )
477     {
478       h[i1] = 0. ;
479       for ( i2 = 0 ; i2 < 5 ; i2++ )  h[i1] += rm[i1][i2]*dr[i2] ;
480     }
481
482     h[0] = 0.5*(rl*ul + rr*ur + h[0]) ;
483     h[1] = 0.5*(rl*ul*ul + pl + rr*ur*ur + pr + h[1]) ;
484     h[2] = 0.5*(rl*vl*ul + rr*vr*ur + h[2]) ;
485     h[3] = 0.5*(rl*wl*ul + rr*wr*ur + h[3]) ;
486     h[4] = 0.5*((rel+pl)*ul + (rer+pr)*ur + h[4]) ;
487
488
489     frho[ii]  = h[0]*da[ii] ;
490     frhou[ii] = (h[1]*nx[ii] + h[2]*sx[ii] + h[3]*tx[ii])*da[ii] ;
491     frhov[ii] = (h[1]*ny[ii] + h[2]*sy[ii] + h[3]*ty[ii])*da[ii] ;
492     frhow[ii] = (h[1]*nz[ii] + h[2]*sz[ii] + h[3]*tz[ii])*da[ii] ;
493     frhoe[ii] = h[4]*da[ii] ;
494    jj++;
495    }
496   }
497  }
498 } /* close case */
499 } /* close switch */
500 } /* close for loop */
501
502
503 return ;
504 }
```

8.2.10 free_ dtensor1.cu

```
1  /*
2     ----------------------
3     function free_dtensor1
4     ----------------------
5
6     frees memory for first order tensor of type float
7
8     17 Jan 98  Written
9
10 */
11 /*
12   includes
13 */
14 #include "defs.h"
15 #include "extern.h"
16 /*
17   externals
18 */
19
20 void free_dtensor1(double *t)
21 {
22    free(t) ;
23 }
```

8.2.11 gidx.cu

```
/*
  computes index in array grid
*/
/*
  includes
*/
#include "defs.h"
#include "extern.h"
/*
  externals
*/

int gidx(int i, int j, int k, int il, int jl, int gh)
{
  int gh2 ;
  gh2 = 2*gh ;
  return (gh2+il+1)*(k*(gh2+jl+1)+j)+i ;
}
```

8.2.12 global.h

```
/*
  variables
*/

struct cell *c ;                /* c denotes cell  */

struct data d ;                 /* d denotes data  */

struct grid *g ;                /* g denotes grid  */

FILE *fp[FILES] ;               /* file pointers   */
```

8.2.13 input.cu

```
/*
  function input

*/
/*
  includes
*/
#include "defs.h"
#include "extern.h"
/*
  externals
*/
extern void allocate() ;

void input()
{
  char title[20] ;
```

```
18  int  i ;
19
20  rewind(fp[0]) ;
21
22  fprintf(fp[1],"--------------\n") ;
23  fprintf(fp[1],"3-D Euler Code\n") ;
24  fprintf(fp[1],"--------------\n") ;
25
26  /*  rstart  */
27
28  rewind(fp[2]) ;
29
30  fscanf(fp[2],"%d%d%d%d",&d.il,&d.jl,&d.kl,&d.gh) ;
31  fscanf(fp[2],"%lf",&d.time) ;
32
33  d.nc = (2*d.gh+d.il)*(2*d.gh+d.jl)*(2*d.gh+d.kl) ;
34  d.nn = (2*d.gh+d.il+1)*(2*d.gh+d.jl+1)*(2*d.gh+d.kl+1) ;
35
36  allocate() ;
37
38  for ( i = 0 ; i < d.nc ; i++ )
39  fscanf(fp[2],"%lf%lf%lf%lf%lf",
40                  &c[i].rho[0],&c[i].rhou[0],&c[i].rhov[0],
41                  &c[i].rhow[0],&c[i].rhoe[0]) ;
42
43  for ( i = 0 ; i < d.nn ; i++ )
44    fscanf(fp[2],"%lf%lf%lf",&g[i].x,&g[i].y,&g[i].z) ;
45
46  fprintf(fp[1],"Read rstart file:\n") ;
47  fprintf(fp[1],"  il          %d\n",d.il) ;
48  fprintf(fp[1],"  jl          %d\n",d.jl) ;
49  fprintf(fp[1],"  kl          %d\n",d.kl) ;
50  fprintf(fp[1],"  gh          %d\n",d.gh) ;
51  fprintf(fp[1],"  nc          %d\n",d.nc) ;
52  fprintf(fp[1],"  nn          %d\n",d.nn) ;
53
54  /*  datain  */
55
56  fprintf(fp[1],"Read datain file:\n") ;
57  fscanf(fp[0],"%s%lf",&title[0],&d.cfl) ;
58  fprintf(fp[1],"  %s         %lf\n",title,d.cfl) ;
59  fscanf(fp[0],"%s%lf",&title[0],&d.dt) ;
60  fprintf(fp[1],"  %s          %lf\n",title,d.dt) ;
61  fscanf(fp[0],"%s%d",&title[0],&d.tint) ;
62  fprintf(fp[1],"  %s        %d\n",title,d.tint) ;
63
64  fscanf(fp[0],"%s%lf",&title[0],&d.gamma) ;
65  fprintf(fp[1],"  %s       %lf\n",title,d.gamma) ;
66
67  d.machinf = 1.0 ;
68
69  fscanf(fp[0],"%s%d",&title[0],&d.check) ;
70  fprintf(fp[1],"  %s       %d\n",title,d.check) ;
71  fscanf(fp[0],"%s%d",&title[0],&d.iter) ;
72  fprintf(fp[1],"  %s        %d\n",title,d.iter) ;
73
74 }
```

8.2.14 bc.cu

```
/*

  function bc

  updates ghost cells

  21 jun 13  written

  description

    updates ghost cells using periodic boundary conditions

    does not update the gh x gh x gh cube at each of the eight corners

*/
/*
  includes
*/
#include "defs.h"
#include "extern.h"
/*
  externals
*/
extern int cidx(int, int, int, int, int, int) ;

void bc(int m)
{
  int  gh, gh2, i, i1, i2, il, j, jl, k, kl ;

  gh  = d.gh ;
  gh2 = 2*d.gh ;
  il  = d.il ;
  jl  = d.jl ;
  kl  = d.kl ;

  /*  xi = 0  */

  for ( k = gh ; k < gh + kl ; k++ )
  {
    for ( j = gh ; j < gh + jl ; j++ )
    {
      for ( i = 0 ; i < gh ; i++ )
      {
        i1 = cidx(i,j,k,il,jl,gh) ;
        i2 = cidx(i+il,j,k,il,jl,gh) ;
        c[i1].rho[m]  = c[i2].rho[m] ;
        c[i1].rhou[m] = c[i2].rhou[m] ;
        c[i1].rhov[m] = c[i2].rhov[m] ;
        c[i1].rhow[m] = c[i2].rhow[m] ;
        c[i1].rhoe[m] = c[i2].rhoe[m] ;
      }
    }
  }

  /*  xi = 1  */

```

```
57  for ( k = gh ; k < gh + kl ; k++ )
58  {
59    for ( j = gh ; j < gh + jl ; j++ )
60    {
61      for ( i = il + gh ; i < il + gh2 ; i++ )
62      {
63        i1 = cidx(i,j,k,il,jl,gh) ;
64        i2 = cidx(i-il,j,k,il,jl,gh) ;
65        c[i1].rho[m]  = c[i2].rho[m] ;
66        c[i1].rhou[m] = c[i2].rhou[m] ;
67        c[i1].rhov[m] = c[i2].rhov[m] ;
68        c[i1].rhow[m] = c[i2].rhow[m] ;
69        c[i1].rhoe[m] = c[i2].rhoe[m] ;
70      }
71    }
72  }
73
74  /*  eta = 0  */
75
76  for ( k = gh ; k < gh + kl ; k++ )
77  {
78    for ( i = gh ; i < gh + il ; i++ )
79    {
80      for ( j = 0 ; j < gh ; j++ )
81      {
82        i1 = cidx(i,j,k,il,jl,gh) ;
83        i2 = cidx(i,j+jl,k,il,jl,gh) ;
84        c[i1].rho[m]  = c[i2].rho[m] ;
85        c[i1].rhou[m] = c[i2].rhou[m] ;
86        c[i1].rhov[m] = c[i2].rhov[m] ;
87        c[i1].rhow[m] = c[i2].rhow[m] ;
88        c[i1].rhoe[m] = c[i2].rhoe[m] ;
89      }
90    }
91  }
92
93  /*  eta = 1  */
94
95  for ( k = gh ; k < gh + kl ; k++ )
96  {
97    for ( i = gh ; i < gh + il ; i++ )
98    {
99      for ( j = jl + gh ; j < jl + gh2 ; j++ )
100     {
101       i1 = cidx(i,j,k,il,jl,gh) ;
102       i2 = cidx(i,j-jl,k,il,jl,gh) ;
103       c[i1].rho[m]  = c[i2].rho[m] ;
104       c[i1].rhou[m] = c[i2].rhou[m] ;
105       c[i1].rhov[m] = c[i2].rhov[m] ;
106       c[i1].rhow[m] = c[i2].rhow[m] ;
107       c[i1].rhoe[m] = c[i2].rhoe[m] ;
108     }
109   }
110 }
111
112 /*  zeta = 0  */
113
114 for ( j = gh ; j < gh + jl ; j++ )
```

```
115   {
116     for ( i = gh ; i < gh + il ; i++ )
117     {
118       for ( k = 0 ; k < gh ; k++ )
119       {
120         i1 = cidx(i,j,k,il,jl,gh) ;
121         i2 = cidx(i,j,k+kl,il,jl,gh) ;
122         c[i1].rho[m]  = c[i2].rho[m] ;
123         c[i1].rhou[m] = c[i2].rhou[m] ;
124         c[i1].rhov[m] = c[i2].rhov[m] ;
125         c[i1].rhow[m] = c[i2].rhow[m] ;
126         c[i1].rhoe[m] = c[i2].rhoe[m] ;
127       }
128     }
129   }
130
131   /*  zeta = 1  */
132
133   for ( j = gh ; j < gh + jl ; j++ )
134   {
135     for ( i = gh ; i < gh + il ; i++ )
136     {
137       for ( k = kl + gh ; k < kl + gh2 ; k++ )
138       {
139         i1 = cidx(i,j,k,il,jl,gh) ;
140         i2 = cidx(i,j,k-kl,il,jl,gh) ;
141         c[i1].rho[m]  = c[i2].rho[m] ;
142         c[i1].rhou[m] = c[i2].rhou[m] ;
143         c[i1].rhov[m] = c[i2].rhov[m] ;
144         c[i1].rhow[m] = c[i2].rhow[m] ;
145         c[i1].rhoe[m] = c[i2].rhoe[m] ;
146       }
147     }
148   }
149
150 }
```

8.2.15 main.cu

```
1  /*
2
3    ----
4    main
5    ----
6
7    description
8
9    the Euler code is developed for a CPU system. It is intended to be compated to
10   a hybrid CPU/GPU counterpart
11
12   the basic structure of the code is:
13
14
15   o  the spatial reconstruction is first order accurate
16   o  the temporal integration is second order Runge-Kutta
17   o  the flux calculation is performed on a single core
```

```
  o  the boundary conditions are periodic in all three directions

  the flow variables are non-dimensionalized by

  o  rho_infinity
  o  T_infinity
  o  U_infinity
  o  L

  this requires the value of the Mach number

     U_infinity/sqrt(gamma*RGAS*T_infinity)

  to be read by input() as d.machinf

  the velocity scale U_infinity can be chosen to be anything desired.
  a common value is the reference speed of sound which therefore
  implies that d.machinf equals unity

  usage

  a.out

*/

#include "defs.h"
#include "global.h"

/*
  function prototyping
*/

double *dtensor1(int) ;
void free_dtensor1(double *) ;
void filopn() ;
void input() ;
void output() ;
void rk2(double, int, int *, double *, double *, double *, double *,
  double *, double *, double *, double *, double *, double *,
  double *, double *, double *, double *, double *, double *,
  double *, double *, double *, double *, double *, double *,
  double *, double *, double *, double *, double *, double *,
  double *, double *, double *) ;
void tecprep() ;
double timestep(int) ;
void unitv() ;

/*
  program
*/

main(int argc, char **argv)
{
  int i, f, gh, il, jl, kl, m, mm, nfa, nfi, nfj, nfk, nfm, *nf ;
  double dt ;
  double *da, *rhol, *rhor, *rhoul, *rhour,
   *rhovl, *rhovr, *rhowl, *rhowr, *rhoel,
   *rhoer, *nx, *ny, *nz, *sx, *sy, *sz,
   *tx, *ty, *tz, *frho, *frhou, *frhov,
```

```
   *frhow, *frhoe;


  filopn() ;

  input() ;

  unitv() ;


  il  = d.il ;
  jl  = d.jl ;
  kl  = d.kl ;
  nfi = (il+1)*jl*kl ;
  nfj = il*(jl+1)*kl ;
  nfk = il*jl*(kl+1) ;
  nfm = MAX(nfi,nfj) ;
  nfm = MAX(nfm,nfk) ;
  nfa = 3*nfm ;

/* Allocate memory */
  nf    = (int *) malloc((size_t) 3*sizeof(int)) ;
  if ( nf==NULL )
  {
    fprintf(stderr,"Cannot allocate memory for nf in flux()\n") ;
    exit(1) ;
  }
    da    = dtensor1(nfa);
    rhol  = dtensor1(nfa);
    rhor  = dtensor1(nfa);
    rhoul  = dtensor1(nfa);
    rhour  = dtensor1(nfa);
    rhovl  = dtensor1(nfa);
    rhovr  = dtensor1(nfa);
    rhowl  = dtensor1(nfa);
    rhowr  = dtensor1(nfa);
    rhoel  = dtensor1(nfa);
    rhoer  = dtensor1(nfa);
    nx    = dtensor1(nfa);
    ny    = dtensor1(nfa);
    nz    = dtensor1(nfa);
    sx    = dtensor1(nfa);
    sy    = dtensor1(nfa);
    sz    = dtensor1(nfa);
    tx    = dtensor1(nfa);
    ty    = dtensor1(nfa);
    tz    = dtensor1(nfa);
    frho  = dtensor1(nfa);
    frhou  = dtensor1(nfa);
    frhov  = dtensor1(nfa);
    frhow  = dtensor1(nfa);
    frhoe  = dtensor1(nfa);
```

```
  for ( i = 0 ; i < d.iter ; i++ )
  {
    dt = timestep(i) ;
    rk2(dt, i ,nf, da, rhol, rhor, rhoul, rhour,
   rhovl, rhovr, rhowl, rhowr, rhoel,
   rhoer, nx, ny, nz, sx, sy, sz,
   tx, ty, tz, frho, frhou, frhov, frhow, frhoe) ;
    d.time += dt ;
    if ( (i > 0)&(i%d.check==0) )  output() ;
  }









  free_dtensor1(da) ;
  free_dtensor1(rhol) ;
  free_dtensor1(rhor) ;
  free_dtensor1(rhoul) ;
  free_dtensor1(rhour) ;
  free_dtensor1(rhovl) ;
  free_dtensor1(rhovr) ;
  free_dtensor1(rhowl) ;
  free_dtensor1(rhowr) ;
  free_dtensor1(rhoel) ;
  free_dtensor1(rhoer) ;
  free_dtensor1(nx) ;
  free_dtensor1(ny) ;
  free_dtensor1(nz) ;
  free_dtensor1(sx) ;
  free_dtensor1(sy) ;
  free_dtensor1(sz) ;
  free_dtensor1(tx) ;
  free_dtensor1(ty) ;
  free_dtensor1(tz) ;
  free_dtensor1(frho) ;
  free_dtensor1(frhou) ;
  free_dtensor1(frhov) ;
  free_dtensor1(frhow) ;
  free_dtensor1(frhoe) ;


  tecprep() ;

  output() ;

  exit(0) ;
}
```

8.2.16 Makefile

```
#
SHELL       = /bin/sh
CC      = nvcc
CLINKER     = $(CC)
```

```
OPTFLAGS    = -O0 -G -g
OPTFLAGSL   = -lm
#
a.out:  allocate.o \
  bc.o \
  cidx.o \
  dtensor1.o \
  filopn.o \
  flux.o \
  fluxh.o \
  free_dtensor1.o \
  gidx.o \
  input.o \
  main.o \
  output.o \
  ran.o \
  reconstruct.o \
  rk2.o \
  tecprep.o \
  timestep.o \
  unitv.o
  $(CLINKER) -o a.out \
  allocate.o \
  bc.o \
  cidx.o \
  dtensor1.o \
  filopn.o \
  flux.o \
  fluxh.o \
  free_dtensor1.o \
  gidx.o \
  input.o \
  main.o \
  output.o \
  ran.o \
  reconstruct.o \
  rk2.o \
  tecprep.o \
  timestep.o \
  unitv.o $(OPTFLAGSL)
allocate.o: defs.h extern.h global.h allocate.cu
  $(CC) $(OPTFLAGS) -c allocate.cu
bc.o: defs.h extern.h global.h bc.cu
  $(CC) $(OPTFLAGS) -c bc.cu
cidx.o: defs.h extern.h global.h cidx.cu
  $(CC) $(OPTFLAGS) -c cidx.cu
dtensor1.o: defs.h extern.h global.h dtensor1.cu
  $(CC) $(OPTFLAGS) -c dtensor1.cu
filopn.o: defs.h extern.h global.h filopn.cu
  $(CC) $(OPTFLAGS) -c filopn.cu
flux.o: defs.h extern.h global.h flux.cu
  $(CC) $(OPTFLAGS) -c flux.cu
fluxh.o: defs.h extern.h global.h fluxh.cu
  $(CC) $(OPTFLAGS) -c fluxh.cu
free_dtensor1.o: defs.h extern.h global.h free_dtensor1.cu
  $(CC) $(OPTFLAGS) -c free_dtensor1.cu
gidx.o: defs.h extern.h global.h gidx.cu
  $(CC) $(OPTFLAGS) -c gidx.cu
```

```
input.o: defs.h extern.h global.h input.cu
  $(CC) $(OPTFLAGS) -c input.cu
main.o: defs.h extern.h global.h main.cu
  $(CC) $(OPTFLAGS) -c main.cu
output.o: defs.h extern.h global.h output.cu
  $(CC) $(OPTFLAGS) -c output.cu
ran.o: defs.h extern.h global.h ran.cu
  $(CC) $(OPTFLAGS) -c ran.cu
reconstruct.o: defs.h extern.h global.h reconstruct.cu
  $(CC) $(OPTFLAGS) -c reconstruct.cu
rk2.o: defs.h extern.h global.h rk2.cu
  $(CC) $(OPTFLAGS) -c rk2.cu
tecprep.o: defs.h extern.h global.h tecprep.cu
  $(CC) $(OPTFLAGS) -c tecprep.cu
timestep.o: defs.h extern.h global.h timestep.cu
  $(CC) $(OPTFLAGS) -c timestep.cu
unitv.o: defs.h extern.h global.h unitv.cu
  $(CC) $(OPTFLAGS) -c unitv.cu
#
#   Notes:
#
#   1.  If the following message is received:
#
#       make: file `Makefile' line 9: Must be a separator (: or ::)
#             for rules (bu39)
#
# then the corresponding line did NOT begin with a tab.
#
```

8.2.17 output.cu

```
/*

  function output

*/
/*
  includes
*/
#include "defs.h"
#include "extern.h"
/*
  externals
*/

void output()
{
  int i ;

  rewind(fp[3]) ;

  fprintf(fp[3],"%d %d %d %d\n",d.il,d.jl,d.kl,d.gh) ;
  fprintf(fp[3],"%lf\n",d.time) ;

  for ( i = 0 ; i < d.nc ; i++ )
    fprintf(fp[3],"%20.12e %20.12e %20.12e %20.12e %20.12e\n",
```

```
26        c[i].rho[0],c[i].rhou[0],c[i].rhov[0],c[i].rhow[0],c[i].rhoe[0]) ;
27
28    for ( i = 0 ; i < d.nn ; i++ )
29      fprintf(fp[3],"%20.12e %20.12e %20.12e\n",g[i].x,g[i].y,g[i].z) ;
30
31    rewind(fp[3]) ;
32
33  }
```

8.2.18 ran.cu

```
1  /*
2
3     function ran
4
5     generates a random number of type FLOAT between 0. and 1.
6
7  */
8  /*
9    includes
10 */
11 #include "defs.h"
12 #include "extern.h"
13 /*
14   externals
15 */
16
17 double ran()
18 {
19    return ( (double) random() ) / 2147483647. ;
20 }
```

8.2.19 reconstruct.cu

```
1  /*
2
3    function reconstruct
4
5    reconstructs flow variables to the negative face and defines unit vectors
6
7    the reconstructed variables are (rho, rhou, rhov, rhow, rhoe) where
8    rhou is the product of the density and the x-direction velocity, etc
9
10   m is the solution level for the conservative variables
11
12   example for cells in xi-direction:
13
14
15       negative face of cell ___     ___ cell (i,j,k)
16                                |   |
17
18               |       |       |       |       |       |
19               |  i-2  |  i-1  |   i   |  i+1  |  i+2  |
20               |       |       |       |       |       |
```

```
21                                ^ ^
22                                | |
23            left side of face --- --- right side of face
24            reconstruct rhol[]        reconstruct rhor[]
25
26  */
27  /*
28    includes
29  */
30  #include "defs.h"
31  #include "extern.h"
32  /*
33    externals
34  */
35  extern int cidx(int, int, int, int, int, int) ;
36
37
38  void reconstruct(double *da, double *rhol, double *rhor, double *rhoul,
39                   double *rhour, double *rhovl, double *rhovr, double *rhowl,
40                   double *rhowr, double *rhoel, double *rhoer, double *nx,
41                   double *ny, double *nz, double *sx, double *sy,
42                   double *sz, double *tx, double *ty, double *tz,
43                   int m, int n)
44  {
45    char    error[160] ;
46    int     gh, i, ii, iil, iir, il, j, jl, k, kl,
47            nfi, nfj, nfk, nfm ;
48
49    gh  = d.gh ;
50    il  = d.il ;
51    jl  = d.jl ;
52    kl  = d.kl ;
53
54    nfi = (il+1)*jl*kl ;
55    nfj = il*(jl+1)*kl ;
56    nfk = il*jl*(kl+1) ;
57    nfm = MAX(nfi,nfj) ;
58    nfm = MAX(nfm,nfk) ;
59
60    switch (n)
61    {
62      case 0:
63      {
64        ii = n*nfm ;
65        for ( k = gh ; k < kl + gh ; k++ )
66        {
67          for ( j = gh ; j < jl + gh ; j++ )
68          {
69            for ( i = gh ; i <= il + gh ; i++ )
70            {
71              iil       = cidx(i-1,j,k,il,jl,gh) ;
72              iir       = cidx(i,j,k,il,jl,gh) ;
73              da[ii]    = c[iir].da[n] ;
74              rhol[ii]  = c[iil].rho[m] ;
75              rhoul[ii] = c[iil].rhou[m] ;
76              rhovl[ii] = c[iil].rhov[m] ;
77              rhowl[ii] = c[iil].rhow[m] ;
78              rhoel[ii] = c[iil].rhoe[m] ;
```

```
79            rhor[ii]  = c[iir].rho[m] ;
80            rhour[ii] = c[iir].rhou[m] ;
81            rhovr[ii] = c[iir].rhov[m] ;
82            rhowr[ii] = c[iir].rhow[m] ;
83            rhoer[ii] = c[iir].rhoe[m] ;
84            nx[ii]    = c[iir].n[0][0] ;
85            ny[ii]    = c[iir].n[0][1] ;
86            nz[ii]    = c[iir].n[0][2] ;
87            sx[ii]    = c[iir].s[0][0] ;
88            sy[ii]    = c[iir].s[0][1] ;
89            sz[ii]    = c[iir].s[0][2] ;
90            tx[ii]    = c[iir].t[0][0] ;
91            ty[ii]    = c[iir].t[0][1] ;
92            tz[ii]    = c[iir].t[0][2] ;
93            ii++ ;
94          }
95        }
96      }
97      break ;
98    }
99    case 1:
100   {
101     ii = n*nfm ;
102     for ( k = gh ; k < kl + gh ; k++ )
103     {
104       for ( i = gh ; i < il + gh ; i++ )
105       {
106         for ( j = gh ; j <= jl + gh ; j++ )
107         {
108           iil       = cidx(i,j-1,k,il,jl,gh) ;
109           iir       = cidx(i,j,k,il,jl,gh) ;
110           da[ii]    = c[iir].da[n] ;
111           rhol[ii]  = c[iil].rho[m] ;
112           rhoul[ii] = c[iil].rhou[m] ;
113           rhovl[ii] = c[iil].rhov[m] ;
114           rhowl[ii] = c[iil].rhow[m] ;
115           rhoel[ii] = c[iil].rhoe[m] ;
116           rhor[ii]  = c[iir].rho[m] ;
117           rhour[ii] = c[iir].rhou[m] ;
118           rhovr[ii] = c[iir].rhov[m] ;
119           rhowr[ii] = c[iir].rhow[m] ;
120           rhoer[ii] = c[iir].rhoe[m] ;
121           nx[ii]    = c[iir].n[1][0] ;
122           ny[ii]    = c[iir].n[1][1] ;
123           nz[ii]    = c[iir].n[1][2] ;
124           sx[ii]    = c[iir].s[1][0] ;
125           sy[ii]    = c[iir].s[1][1] ;
126           sz[ii]    = c[iir].s[1][2] ;
127           tx[ii]    = c[iir].t[1][0] ;
128           ty[ii]    = c[iir].t[1][1] ;
129           tz[ii]    = c[iir].t[1][2] ;
130           ii++ ;
131         }
132       }
133     }
134     break ;
135   }
136   case 2:
```

```
    {
      ii = n*nfm ;
      for ( j = gh ; j < jl + gh ; j++ )
      {
        for ( i = gh ; i < il + gh ; i++ )
        {
          for ( k = gh ; k <= kl + gh ; k++ )
          {
            iil       = cidx(i,j,k-1,il,jl,gh) ;
            iir       = cidx(i,j,k,il,jl,gh) ;
            da[ii]    = c[iir].da[n] ;
            rhol[ii]  = c[iil].rho[m] ;
            rhoul[ii] = c[iil].rhou[m] ;
            rhovl[ii] = c[iil].rhov[m] ;
            rhowl[ii] = c[iil].rhow[m] ;
            rhoel[ii] = c[iil].rhoe[m] ;
            rhor[ii]  = c[iir].rho[m] ;
            rhour[ii] = c[iir].rhou[m] ;
            rhovr[ii] = c[iir].rhov[m] ;
            rhowr[ii] = c[iir].rhow[m] ;
            rhoer[ii] = c[iir].rhoe[m] ;
            nx[ii]    = c[iir].n[2][0] ;
            ny[ii]    = c[iir].n[2][1] ;
            nz[ii]    = c[iir].n[2][2] ;
            sx[ii]    = c[iir].s[2][0] ;
            sy[ii]    = c[iir].s[2][1] ;
            sz[ii]    = c[iir].s[2][2] ;
            tx[ii]    = c[iir].t[2][0] ;
            ty[ii]    = c[iir].t[2][1] ;
            tz[ii]    = c[iir].t[2][2] ;
            ii++ ;
          }
        }
      }
      break ;
    }
    default:
    {
      sprintf(error,"Invalid value for n in reconstruct.  Stop.\n") ;
      fputs(error,stderr) ;
      exit(1) ;
    }
  }
  return ;
}
```

8.2.20 rk2.cu

```
/*

  runge-kutta integration (second order)

*/
/*
  includes
*/
```

```
#include "defs.h"
#include "extern.h"
/*
  externals
*/
extern void bc(int) ;
extern int cidx(int, int, int, int, int, int) ;
extern void flux(int, int, int *, double *, double *, double *, double *,
  double *, double *, double *, double *, double *, double *,
  double *, double *, double *, double *, double *, double *,
  double *, double *, double *, double *, double *, double *,
  double *, double *, double *) ;

void rk2(double dt, int step, int *nf, double *da, double *rhol,
   double *rhor, double *rhoul, double *rhour,
   double *rhovl, double *rhovr, double *rhowl,
   double *rhowr, double *rhoel, double *rhoer,
   double *nx, double *ny, double *nz, double *sx,
   double *sy, double *sz, double *tx, double *ty,
   double *tz, double *frho, double *frhou,
   double *frhov, double *frhow, double *frhoe)
{
  int    i, ii, il, i1, i2, i3, i4, j, jl, k, kl, m, ne, gh, gh2 ;
  double dt2, rhs[5] ;

  gh  = d.gh ;
  gh2 = 2*gh ;
  il  = d.il ;
  jl  = d.jl ;
  kl  = d.kl ;
  dt2 = 0.5*dt ;

  /*  step no. 1  */

  m = 0 ;
  bc(m) ;
  flux( m, step, nf, da, rhol, rhor, rhoul, rhour, rhovl, rhovr,
     rhowl, rhowr, rhoel, rhoer, nx, ny, nz, sx, sy,
     sz, tx, ty, tz, frho, frhou, frhov, frhow,
     frhoe) ;

  for ( k = gh ; k < gh + kl ; k++ )
  {
    for ( j = gh ; j < gh + jl ; j++ )
    {
      for ( i = gh ; i < gh + il ; i++ )
      {
        i1 = cidx(i,j,k,il,jl,gh) ;
        i2 = cidx(i+1,j,k,il,jl,gh) ;
        i3 = cidx(i,j+1,k,il,jl,gh) ;
        i4 = cidx(i,j,k+1,il,jl,gh) ;
        for ( ne = 0 ; ne < 5 ; ne++ )
        {
          /*  note that the minus sign has been incorporated  */
          rhs[ne] = c[i1].inv* (c[i1].flux[0][ne] - c[i2].flux[0][ne] +
                                c[i1].flux[1][ne] - c[i3].flux[1][ne] +
                                c[i1].flux[2][ne] - c[i4].flux[2][ne] ) ;
        }
```

```
/*  if( rhs[2] != 0. )
  {
   printf ( "rhs[2] = %f at   step,i,j,k,m = %d,%d,%d,%d,%d\n", rhs[2],step,i,j,k,m );
   exit(1);
  }
  */    c[i1].rho[1]  = c[i1].rho[0]  + dt2*rhs[0] ;
        c[i1].rhou[1] = c[i1].rhou[0] + dt2*rhs[1] ;
        c[i1].rhov[1] = c[i1].rhov[0] + dt2*rhs[2] ;
        c[i1].rhow[1] = c[i1].rhow[0] + dt2*rhs[3] ;
        c[i1].rhoe[1] = c[i1].rhoe[0] + dt2*rhs[4] ;
      }
    }
  }

  /*  step no. 2  */

  m = 1 ;
  bc(m) ;
  flux( m, step, nf, da, rhol, rhor, rhoul, rhour, rhovl, rhovr,
     rhowl, rhowr, rhoel, rhoer, nx, ny, nz, sx, sy,
     sz, tx, ty, tz, frho, frhou, frhov, frhow,
     frhoe) ;

  for ( k = gh ; k < gh + kl ; k++ )
  {
    for ( j = gh ; j < gh + jl ; j++ )
    {
      for ( i = gh ; i < gh + il ; i++ )
      {
        i1 = cidx(i,j,k,il,jl,gh) ;
        i2 = cidx(i+1,j,k,il,jl,gh) ;
        i3 = cidx(i,j+1,k,il,jl,gh) ;
        i4 = cidx(i,j,k+1,il,jl,gh) ;
        for ( ne = 0 ; ne < 5 ; ne++ )
        {
          /*  note that the minus sign has been incorporated  */
          rhs[ne] = c[i1].inv* (c[i1].flux[0][ne] - c[i2].flux[0][ne] +
                                c[i1].flux[1][ne] - c[i3].flux[1][ne] +
                                c[i1].flux[2][ne] - c[i4].flux[2][ne] ) ;
        }
        c[i1].rho[1]  = c[i1].rho[0]  + dt*rhs[0] ;
        c[i1].rhou[1] = c[i1].rhou[0] + dt*rhs[1] ;
        c[i1].rhov[1] = c[i1].rhov[0] + dt*rhs[2] ;
        c[i1].rhow[1] = c[i1].rhow[0] + dt*rhs[3] ;
        c[i1].rhoe[1] = c[i1].rhoe[0] + dt*rhs[4] ;
      }
    }
  }

  /* store  */

  for ( k = 0 ; k < gh2 + kl ; k++ )
  {
    for ( j = 0 ; j < gh2 + jl ; j++ )
    {
      for ( i = 0 ; i < gh2 + il ; i++ )
      {
        ii = cidx(i,j,k,il,jl,gh) ;
```

```
        c[ii].rho[0]  = c[ii].rho[1] ;
        c[ii].rhou[0] = c[ii].rhou[1] ;
        c[ii].rhov[0] = c[ii].rhov[1] ;
        c[ii].rhow[0] = c[ii].rhow[1] ;
        c[ii].rhoe[0] = c[ii].rhoe[1] ;
      }
    }
  }

}
```

8.2.21 tecprep.cu

```
/*

  function tecprep

  24 aug 04  begin writing

  description of variables

    nil    number of nodes in xi-direction
    njl    number of nodes in xi-direction
    nkl    number of nodes in xi-direction

*/
/*
  includes
*/
#include "defs.h"
#include "extern.h"
/*
  externals
*/
extern int cidx(int, int, int, int, int, int) ;
extern int gidx(int, int, int, int, int, int) ;

void tecprep()
{
  char   error[160] ;
  int    gh, i, i1, il, ip, j, jl, k, kl, nil, njl, nkl ;
  double mm, pp, tt, vv ;

  gh  = d.gh ;
  il  = d.il ;
  jl  = d.jl ;
  kl  = d.kl ;
  nil = il + 1 ;
  njl = jl + 1 ;
  nkl = kl + 1 ;

  fprintf(fp[4],"TITLE = %cTime = %lf%c\n",0x22,d.time,0x22) ;
  fprintf(fp[4],"VARIABLES = X,Y,Z,U,V,W,P,T,M\n") ;
  fprintf(fp[4],"ZONE T=%cTime = %lf%c, I=%d, J=%d, K=%d, ",
          0x22,d.time,0x22,nil,njl,nkl) ;
  fprintf(fp[4],"DATAPACKING=BLOCK,VARLOCATION=([1,2,3]=NODAL),") ;
```

```
44  fprintf(fp[4],"VARLOCATION=([4,5,6,7,8,9,10]=CELLCENTERED)\n") ;
45
46  /*  x  */
47
48  ip = 0 ;
49  for ( k = gh ; k <= gh + kl ; k++ )
50  {
51    for ( j = gh ; j <= gh + jl ; j++ )
52    {
53      for ( i = gh ; i <= gh + il ; i++ )
54      {
55        i1 = gidx(i,j,k,il,jl,gh) ;
56        fprintf(fp[4],"%lf ",g[i1].x) ;
57        ip++ ;
58        if ( ip==OUTPUTPERLINE )
59        {
60          fprintf(fp[4],"\n") ;
61          ip = 0 ;
62        }
63      }
64    }
65  }
66  if ( ip != 0 )  fprintf(fp[4],"\n") ;
67
68  /*  y  */
69
70  ip = 0 ;
71  for ( k = gh ; k <= gh + kl ; k++ )
72  {
73    for ( j = gh ; j <= gh + jl ; j++ )
74    {
75      for ( i = gh ; i <= gh + il ; i++ )
76      {
77        i1 = gidx(i,j,k,il,jl,gh) ;
78        fprintf(fp[4],"%lf ",g[i1].y) ;
79        ip++ ;
80        if ( ip==OUTPUTPERLINE )
81        {
82          fprintf(fp[4],"\n") ;
83          ip = 0 ;
84        }
85      }
86    }
87  }
88  if ( ip != 0 )  fprintf(fp[4],"\n") ;
89
90  /*  z  */
91
92  ip = 0 ;
93  for ( k = gh ; k <= gh + kl ; k++ )
94  {
95    for ( j = gh ; j <= gh + jl ; j++ )
96    {
97      for ( i = gh ; i <= gh + il ; i++ )
98      {
99        i1 = gidx(i,j,k,il,jl,gh) ;
100       fprintf(fp[4],"%lf ",g[i1].z) ;
101       ip++ ;
```

```
        if ( ip==OUTPUTPERLINE )
        {
          fprintf(fp[4],"\n") ;
          ip = 0 ;
        }
      }
    }
  }
  if ( ip != 0 )  fprintf(fp[4],"\n") ;

  /*  u  */

  ip = 0 ;
  for ( k = gh ; k < gh + kl ; k++ )
  {
    for ( j = gh ; j < gh + jl ; j++ )
    {
      for ( i = gh ; i < gh + il ; i++ )
      {
        i1 = cidx(i,j,k,il,jl,gh) ;
        if ( c[i1].rho[0] > 0. )
  {
          fprintf(fp[4],"%lf ",c[i1].rhou[0]/c[i1].rho[0]) ;
        }
        else
        {
          sprintf(error,"Density = %lf in tecprep. Stop\n",c[i1].rho[0]) ;
          fputs(error,stderr) ;
          exit(0) ;
        }
        ip++ ;
        if ( ip==OUTPUTPERLINE )
        {
          fprintf(fp[4],"\n") ;
          ip = 0 ;
        }
      }
    }
  }
  if ( ip != 0 )  fprintf(fp[4],"\n") ;

  /*  v  */

  ip = 0 ;
  for ( k = gh ; k < gh + kl ; k++ )
  {
    for ( j = gh ; j < gh + jl ; j++ )
    {
      for ( i = gh ; i < gh + il ; i++ )
      {
        i1 = cidx(i,j,k,il,jl,gh) ;
        fprintf(fp[4],"%lf ",c[i1].rhov[0]/c[i1].rho[0]) ;
        ip++ ;
        if ( ip==OUTPUTPERLINE )
        {
          fprintf(fp[4],"\n") ;
          ip = 0 ;
        }
```

```
    }
  }
}
if ( ip != 0 )  fprintf(fp[4],"\n") ;

/*  w  */

ip = 0 ;
for ( k = gh ; k < gh + kl ; k++ )
{
  for ( j = gh ; j < gh + jl ; j++ )
  {
    for ( i = gh ; i < gh + il ; i++ )
    {
      i1 = cidx(i,j,k,il,jl,gh) ;
      fprintf(fp[4],"%lf ",c[i1].rhow[0]/c[i1].rho[0]) ;
      ip++ ;
      if ( ip==OUTPUTPERLINE )
      {
        fprintf(fp[4],"\n") ;
        ip = 0 ;
      }
    }
  }
}
if ( ip != 0 )  fprintf(fp[4],"\n") ;

/*  static p  */

ip = 0 ;
for ( k = gh ; k < gh + kl ; k++ )
{
  for ( j = gh ; j < gh + jl ; j++ )
  {
    for ( i = gh ; i < gh + il ; i++ )
    {
      i1 = cidx(i,j,k,il,jl,gh) ;
      pp = (d.gamma-1.0)*
           (c[i1].rhoe[0] - 0.5*(c[i1].rhou[0]*c[i1].rhou[0] +
            c[i1].rhov[0]*c[i1].rhov[0] + c[i1].rhow[0]*c[i1].rhow[0])/
            c[i1].rho[0]) ;
      fprintf(fp[4],"%lf ",pp) ;
      ip++ ;
      if ( ip==OUTPUTPERLINE )
      {
        fprintf(fp[4],"\n") ;
        ip = 0 ;
      }
    }
  }
}
if ( ip != 0 )  fprintf(fp[4],"\n") ;

/*  static temperature  */

ip = 0 ;
for ( k = gh ; k < gh + kl ; k++ )
{
```

```
    for ( j = gh ; j < gh + jl ; j++ )
    {
      for ( i = gh ; i < gh + il ; i++ )
      {
        i1 = cidx(i,j,k,il,jl,gh) ;
        tt = d.gamma*(d.gamma-1.0)*d.machinf*d.machinf*
             (c[i1].rhoe[0] - 0.5*(c[i1].rhou[0]*c[i1].rhou[0] +
              c[i1].rhov[0]*c[i1].rhov[0] + c[i1].rhow[0]*c[i1].rhow[0])/
              c[i1].rho[0])/c[i1].rho[0] ;
        fprintf(fp[4],"%lf ",tt) ;
        ip++ ;
        if ( ip==OUTPUTPERLINE )
        {
          fprintf(fp[4],"\n") ;
          ip = 0 ;
        }
      }
    }
  }
  if ( ip != 0 )  fprintf(fp[4],"\n") ;

  /*  Mach number  */

  ip = 0 ;
  for ( k = gh ; k < gh + kl ; k++ )
  {
    for ( j = gh ; j < gh + jl ; j++ )
    {
      for ( i = gh ; i < gh + il ; i++ )
      {
        i1 = cidx(i,j,k,il,jl,gh) ;
        tt = d.gamma*(d.gamma-1.0)*d.machinf*d.machinf*
             (c[i1].rhoe[0] - 0.5*(c[i1].rhou[0]*c[i1].rhou[0] +
              c[i1].rhov[0]*c[i1].rhov[0] + c[i1].rhow[0]*c[i1].rhow[0])/
              c[i1].rho[0])/c[i1].rho[0] ;
        vv = sqrt(c[i1].rhou[0]*c[i1].rhou[0] + c[i1].rhov[0]*c[i1].rhov[0] +
                  c[i1].rhow[0]*c[i1].rhow[0])/c[i1].rho[0] ;
        mm = d.machinf*vv/sqrt(tt) ;
        fprintf(fp[4],"%lf ",mm) ;
        ip++ ;
        if ( ip==OUTPUTPERLINE )
        {
          fprintf(fp[4],"\n") ;
          ip = 0 ;
        }
      }
    }
  }
  if ( ip != 0 )  fprintf(fp[4],"\n") ;


}
```

8.2.22 timestep.cu

```
/*
```

```
2
3   function timestep
4
5   definitions
6
7   aa      speed of sound
8   pp      static pressure
9   tt      static temperature
10  uu      x-component of velocity
11  vv      y-component of velocity
12  ww      z-component of velocity
13
14 */
15 /*
16   includes
17 */
18 #include "defs.h"
19 #include "extern.h"
20 /*
21   externals
22 */
23 extern int cidx(int, int, int, int, int, int) ;
24
25 double timestep(int n)
26 {
27   int    gh, i, il, ii, i0, i1, j, jl, k, kl, m ;
28   double aa, da[2], dt, eigmax, inv, l1, l2, l3, lmax, nda[3], pp, rinv, rr, tt,
29          uu, vv, ww ;
30
31   if ( d.tint == 1 )  return d.dt ;
32
33   gh  = d.gh ;
34   il  = d.il ;
35   jl  = d.jl ;
36   kl  = d.kl ;
37
38   for ( k = gh ; k < gh + kl ; k++ )
39   {
40     for ( j = gh ; j < gh + jl ; j++ )
41     {
42       for ( i = gh ; i < gh + il ; i++ )
43       {
44         ii    = cidx(i,j,k,il,jl,gh) ;
45         rr    = c[ii].rho[0] ;
46         rinv  = 1./rr ;
47         uu    = c[ii].rhou[0]*rinv ;
48         vv    = c[ii].rhov[0]*rinv ;
49         ww    = c[ii].rhow[0]*rinv ;
50         pp    = (d.gamma-1.0)*(c[ii].rhoe[0]-0.5*rr*(uu*uu+vv*vv+ww*ww)) ;
51         tt    = d.gamma*d.machinf*d.machinf*pp*rinv ;
52         if ( tt <= 0. )
53         {
54           fprintf(fp[1],"temperature = %13.5E in timestep() at (%d,%d,%d)\n",
55                   tt,i,j,k) ;
56           exit(1) ;
57         }
58         else
59         {
```

```
60            aa    = sqrt(tt)/d.machinf ;
61          }
62          inv   = c[ii].inv ;
63
64          i0    = ii ;
65          da[0] = c[i0].da[0] ;
66          for ( m = 0 ; m < 3 ; m++ )  nda[m]  = c[i0].n[0][m] * da[0] ;
67          i1    = cidx(i+1,j,k,il,jl,gh) ;
68          da[1] = c[i1].da[0] ;
69          for ( m = 0 ; m < 3 ; m++ )  nda[m] += c[i1].n[0][m] * da[1] ;
70          l1 = 0.5*(fabs(uu*nda[0]+vv*nda[1]+ww*nda[2]) + aa*(da[0]+da[1]))*inv ;
71
72          da[0] = c[i0].da[1] ;
73          for ( m = 0 ; m < 3 ; m++ )  nda[m]  = c[i0].n[1][m] * da[0] ;
74          i1    = cidx(i,j+1,k,il,jl,gh) ;
75          da[1] = c[i1].da[1] ;
76          for ( m = 0 ; m < 3 ; m++ )  nda[m] += c[i1].n[1][m] * da[1] ;
77          l2 = 0.5*(fabs(uu*nda[0]+vv*nda[1]+ww*nda[2]) + aa*(da[0]+da[1]))*inv ;
78
79          da[0] = c[i0].da[2] ;
80          for ( m = 0 ; m < 3 ; m++ )  nda[m]  = c[i0].n[2][m] * da[0] ;
81          i1    = cidx(i,j,k+1,il,jl,gh) ;
82          da[1] = c[i1].da[2] ;
83          for ( m = 0 ; m < 3 ; m++ )  nda[m] += c[i1].n[2][m] * da[1] ;
84          l3 = 0.5*(fabs(uu*nda[0]+vv*nda[1]+ww*nda[2]) + aa*(da[0]+da[1]))*inv ;
85
86          lmax = MAX(l1,l2) ;
87          lmax = MAX(lmax,l3) ;
88
89          if ( (i==gh)&&(j==gh)&&(k==gh) )   eigmax = lmax ;
90          else                               eigmax = MAX(eigmax,lmax) ;
91        }
92      }
93    }
94    if ( eigmax > 0. )
95    {
96      dt = d.cfl/eigmax  ;
97      fprintf(fp[1],"Step no. %d: dt = %13.5E\n",n,dt) ;
98      fflush(fp[1]) ;
99      return dt ;
100   }
101   else
102   {
103     fprintf(fp[1],"eigmax is zero in timestep() at step %d.  Stop\n",n) ;
104     fflush(fp[1]) ;
105     exit(1) ;
106   }
107 }
```

8.2.23 unitv.cu

```
1  /*
2    function unitv
3
4    computes unit vectors and area on
5    three faces of each cell and cell volume
```

```
6
7     10 aug 04  written
8
9   */
10  /*
11    includes
12  */
13  #include "defs.h"
14  #include "extern.h"
15  /*
16    externals
17  */
18  extern int cidx(int, int, int, int, int, int) ;
19  extern int gidx(int, int, int, int, int, int) ;
20  extern double ran() ;
21
22  void unitv()
23  {
24    char   error[80] ;
25    int    i, ii, i1, i2, i3, i4, il, j, jl, k, kl, gh, gh2 ;
26    double cx, cy, cz, da, dai, dx, dx1, dx2, dy, dy1, dy2, dz, dz1, dz2,
27           nrm, nx, ny, nz, rx, ry, rz, sx, sy, sz, tx, ty, tz, vol ;
28
29    gh  = d.gh ;
30    gh2 = 2*gh ;
31    il  = d.il ;
32    jl  = d.jl ;
33    kl  = d.kl ;
34    rx  = ran() ;
35    ry  = ran() ;
36    rz  = ran() ;
37
38    /*  xi-face  */
39
40    for ( k = 0 ; k < gh2 + kl ; k++ )
41    {
42      for ( j = 0 ; j < gh2 + jl ; j++ )
43      {
44        for ( i = 0 ; i < gh2 + il ; i++ )
45        {
46          ii  = cidx(i,j,k,il,jl,gh) ;
47          i1  = gidx(i,j,k,il,jl,gh) ;
48          i2  = gidx(i,j+1,k+1,il,jl,gh) ;
49          i3  = gidx(i,j+1,k,il,jl,gh) ;
50          i4  = gidx(i,j,k+1,il,jl,gh) ;
51          dx1 = g[i2].x - g[i1].x ;
52          dy1 = g[i2].y - g[i1].y ;
53          dz1 = g[i2].z - g[i1].z ;
54          dx2 = g[i4].x - g[i3].x ;
55          dy2 = g[i4].y - g[i3].y ;
56          dz2 = g[i4].z - g[i3].z ;
57          nx  = 0.5*(dy1*dz2-dz1*dy2) ;
58          ny  = 0.5*(dz1*dx2-dx1*dz2) ;
59          nz  = 0.5*(dx1*dy2-dy1*dx2) ;
60          da  = sqrt(nx*nx+ny*ny+nz*nz) ;
61          if ( da > 0. )
62          {
63            dai = 1./da ;
```

```
64          }
65          else
66          {
67             sprintf(error,"Area of xi-face is zero at (%d,%d,%d).  Stop\n",
68                     i,j,k) ;
69             fputs(error,stderr) ;
70             exit(0) ;
71          }
72          nx *= dai ;
73          ny *= dai ;
74          nz *= dai ;
75
76          sx  = ny*rz - nz*ry ;
77          sy  = nz*rx - nx*rz ;
78          sz  = nx*ry - ny*rx ;
79          nrm = sqrt(sx*sx+sy*sy+sz*sz) ;
80          if ( nrm > 0. )
81          {
82            nrm = 1./nrm ;
83          }
84          else
85          {
86             sprintf(error,"Vector s on xi-face is zero at (%d,%d,%d).  Stop\n",
87                     i,j,k) ;
88             fputs(error,stderr) ;
89             exit(0) ;
90          }
91          sx *= nrm ;
92          sy *= nrm ;
93          sz *= nrm ;
94
95          tx  = ny*sz - nz*sy ;
96          ty  = nz*sx - nx*sz ;
97          tz  = nx*sy - ny*sx ;
98
99          /*  this should not be necessary but do it anyway  */
100
101         nrm = sqrt(tx*tx+ty*ty+tz*tz) ;
102         if ( nrm > 0. )
103         {
104           nrm = 1./nrm ;
105         }
106         else
107         {
108            sprintf(error,"Vector t on xi-face is zero at (%d,%d,%d).  Stop\n",
109                    i,j,k) ;
110            fputs(error,stderr) ;
111            exit(0) ;
112         }
113         tx *= nrm ;
114         ty *= nrm ;
115         tz *= nrm ;
116
117         c[ii].n[0][0] = nx ;
118         c[ii].n[0][1] = ny ;
119         c[ii].n[0][2] = nz ;
120         c[ii].s[0][0] = sx ;
121         c[ii].s[0][1] = sy ;
```

```
122         c[ii].s[0][2] = sz ;
123         c[ii].t[0][0] = tx ;
124         c[ii].t[0][1] = ty ;
125         c[ii].t[0][2] = tz ;
126         c[ii].da[0]   = da ;
127       }
128     }
129   }
130
131   /*  eta-face  */
132
133   for ( k = 0 ; k < gh2 + kl ; k++ )
134   {
135     for ( j = 0 ; j < gh2 + jl ; j++ )
136     {
137       for ( i = 0 ; i < gh2 + il ; i++ )
138       {
139         ii  = cidx(i,j,k,il,jl,gh) ;
140         i1  = gidx(i,j,k,il,jl,gh) ;
141         i2  = gidx(i+1,j,k+1,il,jl,gh) ;
142         i3  = gidx(i,j,k+1,il,jl,gh) ;
143         i4  = gidx(i+1,j,k,il,jl,gh) ;
144         dx1 = g[i2].x - g[i1].x ;
145         dy1 = g[i2].y - g[i1].y ;
146         dz1 = g[i2].z - g[i1].z ;
147         dx2 = g[i4].x - g[i3].x ;
148         dy2 = g[i4].y - g[i3].y ;
149         dz2 = g[i4].z - g[i3].z ;
150         nx  = 0.5*(dy1*dz2-dz1*dy2) ;
151         ny  = 0.5*(dz1*dx2-dx1*dz2) ;
152         nz  = 0.5*(dx1*dy2-dy1*dx2) ;
153         da  = sqrt(nx*nx+ny*ny+nz*nz) ;
154         if ( da > 0. )
155         {
156           dai = 1./da ;
157         }
158         else
159         {
160            sprintf(error,"Area of eta-face is zero at (%d,%d,%d).  Stop\n",
161                    i,j,k) ;
162            fputs(error,stderr) ;
163            exit(0) ;
164         }
165         nx *= dai ;
166         ny *= dai ;
167         nz *= dai ;
168
169         sx  = ny*rz - nz*ry ;
170         sy  = nz*rx - nx*rz ;
171         sz  = nx*ry - ny*rx ;
172         nrm = sqrt(sx*sx+sy*sy+sz*sz) ;
173         if ( nrm > 0. )
174         {
175           nrm = 1./nrm ;
176         }
177         else
178         {
179            sprintf(error,"Vector s on eta-face is zero at (%d,%d,%d).  Stop\n",
```

```
                i,j,k) ;
         fputs(error,stderr) ;
         exit(0) ;
       }
       sx *= nrm ;
       sy *= nrm ;
       sz *= nrm ;

       tx  = ny*sz - nz*sy ;
       ty  = nz*sx - nx*sz ;
       tz  = nx*sy - ny*sx ;

       /*  this should not be necessary but do it anyway  */

       nrm = sqrt(tx*tx+ty*ty+tz*tz) ;
       if ( nrm > 0. )
       {
         nrm = 1./nrm ;
       }
       else
       {
         sprintf(error,"Vector t on eta-face is zero at (%d,%d,%d).  Stop\n",
                i,j,k) ;
         fputs(error,stderr) ;
         exit(0) ;
       }
       tx *= nrm ;
       ty *= nrm ;
       tz *= nrm ;

       c[ii].n[1][0] = nx ;
       c[ii].n[1][1] = ny ;
       c[ii].n[1][2] = nz ;
       c[ii].s[1][0] = sx ;
       c[ii].s[1][1] = sy ;
       c[ii].s[1][2] = sz ;
       c[ii].t[1][0] = tx ;
       c[ii].t[1][1] = ty ;
       c[ii].t[1][2] = tz ;
       c[ii].da[1]   = da ;
     }
   }
 }


 /*  zeta-face  */

 for ( k = 0 ; k < gh2 + kl ; k++ )
 {
   for ( j = 0 ; j < gh2 + jl ; j++ )
   {
     for ( i = 0 ; i < gh2 + il ; i++ )
     {
       ii  = cidx(i,j,k,il,jl,gh) ;
       i1  = gidx(i,j,k,il,jl,gh) ;
       i2  = gidx(i+1,j+1,k,il,jl,gh) ;
       i3  = gidx(i+1,j,k,il,jl,gh) ;
       i4  = gidx(i,j+1,k,il,jl,gh) ;
```

```
238         dx1 = g[i2].x - g[i1].x ;
239         dy1 = g[i2].y - g[i1].y ;
240         dz1 = g[i2].z - g[i1].z ;
241         dx2 = g[i4].x - g[i3].x ;
242         dy2 = g[i4].y - g[i3].y ;
243         dz2 = g[i4].z - g[i3].z ;
244         nx  = 0.5*(dy1*dz2-dz1*dy2) ;
245         ny  = 0.5*(dz1*dx2-dx1*dz2) ;
246         nz  = 0.5*(dx1*dy2-dy1*dx2) ;
247         da  = sqrt(nx*nx+ny*ny+nz*nz) ;
248         if ( da > 0. )
249         {
250           dai = 1./da ;
251         }
252         else
253         {
254            sprintf(error,"Area of zeta-face is zero at (%d,%d,%d).  Stop\n",
255                    i,j,k) ;
256            fputs(error,stderr) ;
257            exit(0) ;
258         }
259         nx *= dai ;
260         ny *= dai ;
261         nz *= dai ;
262
263         sx  = ny*rz - nz*ry ;
264         sy  = nz*rx - nx*rz ;
265         sz  = nx*ry - ny*rx ;
266         nrm = sqrt(sx*sx+sy*sy+sz*sz) ;
267         if ( nrm > 0. )
268         {
269           nrm = 1./nrm ;
270         }
271         else
272         {
273            sprintf(error,"Vector s on zeta-face is zero at (%d,%d,%d).  Stop\n",
274                    i,j,k) ;
275            fputs(error,stderr) ;
276            exit(0) ;
277         }
278         sx *= nrm ;
279         sy *= nrm ;
280         sz *= nrm ;
281
282         tx  = ny*sz - nz*sy ;
283         ty  = nz*sx - nx*sz ;
284         tz  = nx*sy - ny*sx ;
285
286         /*  this should not be necessary but do it anyway  */
287
288         nrm = sqrt(tx*tx+ty*ty+tz*tz) ;
289         if ( nrm > 0. )
290         {
291           nrm = 1./nrm ;
292         }
293         else
294         {
295            sprintf(error,"Vector t on zeta-face is zero at (%d,%d,%d).  Stop\n",
```

```
                  i,j,k) ;
          fputs(error,stderr) ;
          exit(0) ;
        }
        tx *= nrm ;
        ty *= nrm ;
        tz *= nrm ;

        c[ii].n[2][0] = nx ;
        c[ii].n[2][1] = ny ;
        c[ii].n[2][2] = nz ;
        c[ii].s[2][0] = sx ;
        c[ii].s[2][1] = sy ;
        c[ii].s[2][2] = sz ;
        c[ii].t[2][0] = tx ;
        c[ii].t[2][1] = ty ;
        c[ii].t[2][2] = tz ;
        c[ii].da[2]   = da ;
      }
    }
  }

  /*  volume  */

  for ( k = 0 ; k < gh2 + kl ; k++ )
  {
    for ( j = 0 ; j < gh2 + jl ; j++ )
    {
      for ( i = 0 ; i < gh2 + il ; i++ )
      {
        ii  = cidx(i,j,k,il,jl,gh) ;
        i1  = gidx(i,j,k,il,jl,gh) ;
        i2  = gidx(i+1,j+1,k+1,il,jl,gh) ;
        dx  = g[i2].x - g[i1].x ;
        dy  = g[i2].y - g[i1].y ;
        dz  = g[i2].z - g[i1].z ;
        cx  = c[ii].n[0][0]*c[ii].da[0] + c[ii].n[1][0]*c[ii].da[1] +
              c[ii].n[2][0]*c[ii].da[2] ;
        cy  = c[ii].n[0][1]*c[ii].da[0] + c[ii].n[1][1]*c[ii].da[1] +
              c[ii].n[2][1]*c[ii].da[2] ;
        cz  = c[ii].n[0][2]*c[ii].da[0] + c[ii].n[1][2]*c[ii].da[1] +
              c[ii].n[2][2]*c[ii].da[2] ;
        vol = (cx*dx+cy*dy+cz*dz)/3. ;
        if ( vol > 0. )
        {
          c[ii].inv = 1./vol ;
        }
        else
        {
           sprintf(error,"Volume is %lf at (%d,%d,%d)\n",vol,i,j,k) ;
           fputs(error,stderr) ;
           sprintf(error,"(cx,cy,cz) = (%lf,%lf,%lf)\n",cx,cy,cz) ;
           fputs(error,stderr) ;
           sprintf(error,"(dx,dy,dz) = (%lf,%lf,%lf)\n",dx,dy,dz) ;
           fputs(error,stderr) ;
           exit(0) ;
        }
      }
```

```
354     }
355   }
356 }
```

8.3 GPU Code

The GPU code contains several files which are identical to the serial code, namely:

```
allocate.cu     bc.cu              cidx.cu         dtensor1.cu
filopn.cu       free_dtensor1.cu   gidx.cu         input.cu
output.cu       ran.cu             reconstruct.cu  tecprep.cu
timestep.cu     unitv.cu
```

The following files differ from their serial analogs, and the file fluxd.cu is new entirely.

```
flux.cu     fluxd.cu     main.cu     rk2.cu
```

The GPU code is sequenced by the makefile:

```
#
SHELL       = /bin/sh
CC      = /usr/local/cuda/bin/nvcc
CLINKER     = $(CC)
OPTFLAGS    = -O0 -G -g -arch=sm_20
OPTFLAGSL   = -lm
#
a.out:  allocate.o \
  bc.o \
  cidx.o \
  dtensor1.o \
  filopn.o \
  flux.o \
  fluxd.o \
  free_dtensor1.o \
  gidx.o \
  input.o \
  main.o \
  map_recon.o \
  output.o \
  ran.o \
  reconstruct.o \
  rk2.o \
  tecprep.o \
  timestep.o \
  unitv.o
  $(CLINKER) -o a.out \
  allocate.o \
  bc.o \
  cidx.o \
  dtensor1.o \
  filopn.o \
  flux.o \
  fluxd.o \
```

```
35    free_dtensor1.o \
36    gidx.o \
37    input.o \
38    main.o \
39    map_recon.o \
40    output.o \
41    ran.o \
42    reconstruct.o \
43    rk2.o \
44    tecprep.o \
45    timestep.o \
46    unitv.o $(OPTFLAGSL)
47  allocate.o: defs.h extern.h global.h allocate.cu
48    $(CC) $(OPTFLAGS) -c allocate.cu
49  bc.o: defs.h extern.h global.h bc.cu
50    $(CC) $(OPTFLAGS) -c bc.cu
51  cidx.o: defs.h extern.h global.h cidx.cu
52    $(CC) $(OPTFLAGS) -c cidx.cu
53  dtensor1.o: defs.h extern.h global.h dtensor1.cu
54    $(CC) $(OPTFLAGS) -c dtensor1.cu
55  filopn.o: defs.h extern.h global.h filopn.cu
56    $(CC) $(OPTFLAGS) -c filopn.cu
57  flux.o: defs.h extern.h global.h flux.cu
58    $(CC) $(OPTFLAGS) -c flux.cu
59  fluxd.o: defs.h extern.h global.h fluxd.cu
60    $(CC) $(OPTFLAGS) -c fluxd.cu
61  free_dtensor1.o: defs.h extern.h global.h free_dtensor1.cu
62    $(CC) $(OPTFLAGS) -c free_dtensor1.cu
63  gidx.o: defs.h extern.h global.h gidx.cu
64    $(CC) $(OPTFLAGS) -c gidx.cu
65  input.o: defs.h extern.h global.h input.cu
66    $(CC) $(OPTFLAGS) -c input.cu
67  main.o: defs.h extern.h global.h main.cu
68    $(CC) $(OPTFLAGS) -c main.cu
69  map_recon.o: defs.h extern.h map_recon.cu
70    $(CC) $(OPTFLAGS) -c map_recon.cu
71  output.o: defs.h extern.h global.h output.cu
72    $(CC) $(OPTFLAGS) -c output.cu
73  ran.o: defs.h extern.h global.h ran.cu
74    $(CC) $(OPTFLAGS) -c ran.cu
75  reconstruct.o: defs.h extern.h global.h reconstruct.cu
76    $(CC) $(OPTFLAGS) -c reconstruct.cu
77  rk2.o: defs.h extern.h global.h rk2.cu
78    $(CC) $(OPTFLAGS) -c rk2.cu
79  tecprep.o: defs.h extern.h global.h tecprep.cu
80    $(CC) $(OPTFLAGS) -c tecprep.cu
81  timestep.o: defs.h extern.h global.h timestep.cu
82    $(CC) $(OPTFLAGS) -c timestep.cu
83  unitv.o: defs.h extern.h global.h unitv.cu
84    $(CC) $(OPTFLAGS) -c unitv.cu
85  #
86  #   Notes:
87  #
88  #   1.  If the following message is received:
89  #
90  #       make: file ‘Makefile’ line 9: Must be a separator (: or ::)
91  #             for rules (bu39)
92  #
```

```
# then the corresponding line did NOT begin with a tab.
#
```

8.3.1 flux.cu

```
/*

  function flux

  computes inviscid flux on GPU and returns to CPU

  definitions

  nf  pointer to number of faces
  nfm   largest number of faces in xi-, eta- or zeta-directions

  da      area of face
  frho    flux for conservation of mass
  frhou   flux for conservation of x-momentum
  frhov   flux for conservation of y-momentum
  frhow   flux for conservation of z-momentum
  frhoe   flux for conservation of energy
  rhol    reconstructed density to left face
  rhor    reconstructed density to right face
  rhoul   reconstructed density*x-velocity to left face
  rhour   reconstructed density*x-velocity to right face
  rhovl   reconstructed density*y-velocity to left face
  rhovr   reconstructed density*y-velocity to right face
  rhowl   reconstructed density*z-velocity to left face
  rhowr   reconstructed density*z-velocity to right face
  rhoel   reconstructed density*energy to left face
  rhoer   reconstructed density*energy to right face
  nx      x-component of normal to the face
  ny      y-component of normal to the face
  nz      z-component of normal to the face
  sx      x-component of unit vector in face
  sy      y-component of unit vector in face
  sz      z-component of unit vector in face
  tx      x-component of unit vector in face
  ty      y-component of unit vector in face
  tz      z-component of unit vector in face

  the above variable with a "d" suffix indicate storage on the device


*/


/*
  includes
*/
#include "defs.h"
#include "extern.h"
/*
  externals
*/
```

```
52 extern int cidx(int, int, int, int, int, int) ;
53 extern double *dtensor1(int) ;
54 extern __global__ void fluxd(double *, double *, double *,
55     double *, double *,double *, double *,
56     double *, double *, double *,double *,
57     double *, double *, double *, double *,double *,
58     double *, double *, double *, double *,double *,
59     double *, double *, double *, double *,
60     int *, int, int, int, int, int) ;
61 extern void free_dtensor1(double *) ;
62 extern void reconstruct(double *, double *, double *, double *, double *,
63                         double *, double *, double *, double *, double *,
64                         double *, double *, double *, double *, double *,
65                         double *, double *, double *, double *, double *,
66                         int, int) ;
67
68 void flux(int m, int step, int *nfd, double *dad, double *rhold,
69   double *rhord, double *rhould, double *rhourd,
70   double *rhovld, double *rhovrd, double *rhowld,
71    double *rhowrd, double *rhoeld, double *rhoerd,
72    double *nxd, double *nyd, double *nzd, double *sxd,
73   double *syd, double *szd, double *txd, double *tyd,
74   double *tzd, double *frhod, double *frhoud,
75   double *frhovd, double *frhowd, double *frhoed)
76 {
77   /*  host  */
78   int     f, gh, i, ii, il, j, jl, jj, k, kl,
79     mm, *nf, nfa, nfi, nfj, nfk, nfm, nft, nn ;
80   double  *da, *frho, *frhou,
81     *frhov, *frhow, *frhoe,
82           *rhol, *rhor, *rhoul, *rhour,
83     *rhovl, *rhovr, *rhowl, *rhowr,
84           *rhoel, *rhoer, *nx, *ny, *nz,
85     *sx, *sy, *sz, *tx, *ty, *tz ;
86   cudaError_t cudaGetLastError(void) ;
87   cudaError_t error ;
88   const char* cudaGetErrorString(cudaError_t) ;
89
90   mm  = m ;
91   gh  = d.gh ;
92   il  = d.il ;
93   jl  = d.jl ;
94   kl  = d.kl ;
95
96   nfi = (il+1)*jl*kl ;
97   nfj = il*(jl+1)*kl ;
98   nfk = il*jl*(kl+1) ;
99   nft = nfi + nfj +nfk ;
100  nfm = MAX(nfi,nfj) ;
101  nfm = MAX(nfm,nfk) ;
102  nfa = 3*nfm ;
103
104
105
106  /*  allocate memory on host  */
107
108
109
```

```
110   nf    = (int *) malloc((size_t) 3*sizeof(int)) ;
111   if ( nf==NULL )
112   {
113     fprintf(stderr,"Cannot allocate memory for nf in flux()\n") ;
114     exit(1) ;
115   }
116
117
118
119
120     da    = dtensor1(nfa);
121     rhol  = dtensor1(nfa);
122     rhor  = dtensor1(nfa);
123     rhoul  = dtensor1(nfa);
124     rhour  = dtensor1(nfa);
125     rhovl  = dtensor1(nfa);
126     rhovr  = dtensor1(nfa);
127     rhowl  = dtensor1(nfa);
128     rhowr  = dtensor1(nfa);
129     rhoel  = dtensor1(nfa);
130     rhoer  = dtensor1(nfa);
131     nx    = dtensor1(nfa);
132     ny    = dtensor1(nfa);
133     nz    = dtensor1(nfa);
134     sx    = dtensor1(nfa);
135     sy    = dtensor1(nfa);
136     sz    = dtensor1(nfa);
137     tx    = dtensor1(nfa);
138     ty    = dtensor1(nfa);
139     tz    = dtensor1(nfa);
140     frho  = dtensor1(nfa);
141     frhou  = dtensor1(nfa);
142     frhov  = dtensor1(nfa);
143     frhow  = dtensor1(nfa);
144     frhoe  = dtensor1(nfa);
145
146   for ( nn = 0 ; nn < 3 ; nn++ )
147   {
148     switch (nn)
149     {
150       case 0:
151         nf[nn] = (il+1)*jl*kl ;
152         break ;
153       case 1:
154         nf[nn] = il*(jl+1)*kl ;
155         break ;
156       case 2:
157         nf[nn] = il*jl*(kl+1) ;
158         break ;
159       default:
160         printf("Invalid value of nn in flux().\n") ;
161         exit(1) ;
162     }
163     /*
164        reconstruction
165     */
166
167     reconstruct(da,rhol,rhor,rhoul,rhour,rhovl,
```

```
168                 rhovr,rhowl,rhowr,rhoel,rhoer,
169                 nx,ny,nz,sx,sy,sz,tx,ty,tz,mm,nn) ;
170   }
171
172
173     if ( cudaMemcpy( nfd, nf, 3*sizeof(int), cudaMemcpyHostToDevice)
174          != cudaSuccess )
175     {
176       printf("Error copy to device for nf \n") ;
177       exit(1) ;
178     }
179
180     if ( cudaMemcpy( dad, da, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
181          != cudaSuccess )
182     {
183       error = cudaGetLastError() ;
184       printf("Error copy to device for da[]\n%s\nerror # = %d\n",cudaGetErrorString(error),error) ;
185       exit(1) ;
186     }
187
188     if ( cudaMemcpy( rhold, rhol, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
189          != cudaSuccess )
190     {
191       printf("Error copy to device for rhol\n") ;
192       exit(1) ;
193     }
194     if ( cudaMemcpy( rhord, rhor, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
195          != cudaSuccess )
196     {
197       printf("Error copy to device for rhor\n") ;
198       exit(1) ;
199     }
200     if ( cudaMemcpy( rhould, rhoul, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
201          != cudaSuccess )
202     {
203       printf("Error copy to device for rhoul\n") ;
204       exit(1) ;
205     }
206     if ( cudaMemcpy( rhourd, rhour, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
207          != cudaSuccess )
208     {
209       printf("Error copy to device for rhour\n") ;
210       exit(1) ;
211     }
212     if ( cudaMemcpy( rhovld, rhovl, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
213          != cudaSuccess )
214     {
215       printf("Error copy to device for rhovl\n") ;
216       exit(1) ;
217     }
218     if ( cudaMemcpy( rhovrd, rhovr, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
219          != cudaSuccess )
220     {
221       printf("Error copy to device for rhovr\n") ;
222       exit(1) ;
223     }
224     if ( cudaMemcpy( rhowld, rhowl, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
225          != cudaSuccess )
```

```
226     {
227       printf("Error copy to device for rhowl\n") ;
228       exit(1) ;
229     }
230     if ( cudaMemcpy( rhowrd, rhowr, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
231          != cudaSuccess )
232     {
233       printf("Error copy to device for rhowr\n") ;
234       exit(1) ;
235     }
236     if ( cudaMemcpy( rhoeld, rhoel, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
237          != cudaSuccess )
238     {
239       printf("Error copy to device for rhoel\n") ;
240       exit(1) ;
241     }
242     if ( cudaMemcpy( rhoerd, rhoer, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
243          != cudaSuccess )
244     {
245       printf("Error copy to device for rhoer\n") ;
246       exit(1) ;
247     }
248
249     if ( cudaMemcpy( nxd, nx, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
250          != cudaSuccess )
251     {
252       printf("Error copy to device for nx\n") ;
253       exit(1) ;
254     }
255     if ( cudaMemcpy( nyd, ny, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
256          != cudaSuccess )
257     {
258       printf("Error copy to device for ny\n") ;
259       exit(1) ;
260     }
261     if ( cudaMemcpy( nzd, nz, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
262          != cudaSuccess )
263     {
264       printf("Error copy to device for nz @ step %d m %d\n", step, m) ;
265       exit(1) ;
266     }
267     if ( cudaMemcpy( sxd, sx, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
268          != cudaSuccess )
269     {
270       printf("Error copy to device for sx\n") ;
271       exit(1) ;
272     }
273     if ( cudaMemcpy( syd, sy, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
274          != cudaSuccess )
275     {
276       printf("Error copy to device for sy\n") ;
277       exit(1) ;
278     }
279     if ( cudaMemcpy( szd, sz, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
280          != cudaSuccess )
281     {
282       printf("Error copy to device for sz\n") ;
283       exit(1) ;
```

```
284     }
285     if ( cudaMemcpy( txd, tx, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
286          != cudaSuccess )
287     {
288       printf("Error copy to device for tx\n") ;
289       exit(1) ;
290     }
291     if ( cudaMemcpy( tyd, ty, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
292          != cudaSuccess )
293     {
294       printf("Error copy to device for ty\n") ;
295       exit(1) ;
296     }
297     if ( cudaMemcpy( tzd, tz, (nfa)*sizeof(double), cudaMemcpyHostToDevice)
298          != cudaSuccess )
299     {
300       printf("Error copy to device for tz\n") ;
301       exit(1) ;
302     }
303
304     /*
305        compute fluxes on negative faces
306     */
307
308     fluxd<<<nfm,3>>>(dad, rhold,  rhord,  rhould, rhourd, rhovld,
309                      rhovrd, rhowld, rhowrd, rhoeld, rhoerd, nxd,
310                      nyd,    nzd,    sxd,    syd,    szd,    txd,
311                      tyd,    tzd,    frhod,  frhoud, frhovd, frhowd,
312                      frhoed, nfd, mm, gh, il, jl, kl) ;
313     /*
314        copy arrays back to host
315     */
316
317     if ( cudaMemcpy( frho, frhod, (nfa)*sizeof(double), cudaMemcpyDeviceToHost)
318          != cudaSuccess )
319     {
320       printf("Error copy to host for frho...m = \n") ;
321       exit(1) ;
322     }
323     if ( cudaMemcpy( frhou, frhoud, (nfa)*sizeof(double), cudaMemcpyDeviceToHost)
324          != cudaSuccess )
325     {
326       printf("Error copy to host for frhou\n") ;
327       exit(1) ;
328     }
329     if ( cudaMemcpy( frhov, frhovd, (nfa)*sizeof(double), cudaMemcpyDeviceToHost)
330          != cudaSuccess )
331     {
332       printf("Error copy to host for frhov\n") ;
333       exit(1) ;
334     }
335     if ( cudaMemcpy( frhow, frhowd, (nfa)*sizeof(double), cudaMemcpyDeviceToHost)
336          != cudaSuccess )
337     {
338       printf("Error copy to host for frhow\n") ;
339       exit(1) ;
340     }
341     if ( cudaMemcpy( frhoe, frhoed, (nfa)*sizeof(double), cudaMemcpyDeviceToHost)
```

```
342         != cudaSuccess )
343     {
344       printf("Error copy to host for frhoe\n") ;
345       exit(1) ;
346     }
347
348     /*
349        save flux
350     */
351
352   for ( nn = 0 ; nn < 3 ; nn++)
353   {
354     switch (nn)
355     {
356       case 0:
357       {
358         jj = 0 ;
359         for ( k = gh ; k < gh + kl ; k++ )
360         {
361           for ( j = gh ; j < gh + jl ; j++ )
362           {
363             for ( i = gh ; i <= gh + il ; i++ )
364             {
365               ii               = cidx(i,j,k,il,jl,gh) ;
366               c[ii].flux[0][0] = frho[nn*nfm+jj] ;
367               c[ii].flux[0][1] = frhou[nn*nfm+jj] ;
368               c[ii].flux[0][2] = frhov[nn*nfm+jj] ;
369               c[ii].flux[0][3] = frhow[nn*nfm+jj] ;
370               c[ii].flux[0][4] = frhoe[nn*nfm+jj] ;
371               jj++ ;
372             }
373           }
374         }
375         break ;
376       }
377       case 1:
378       {
379         jj = 0 ;
380         for ( k = gh ; k < kl + gh ; k++ )
381         {
382           for ( i = gh ; i < il + gh ; i++ )
383           {
384             for ( j = gh ; j <= jl + gh ; j++ )
385             {
386               ii               = cidx(i,j,k,il,jl,gh) ;
387               c[ii].flux[1][0] = frho[nn*nfm+jj] ;
388               c[ii].flux[1][1] = frhou[nn*nfm+jj] ;
389               c[ii].flux[1][2] = frhov[nn*nfm+jj];
390               c[ii].flux[1][3] = frhow[nn*nfm+jj] ;
391               c[ii].flux[1][4] = frhoe[nn*nfm+jj] ;
392               jj++ ;
393             }
394           }
395         }
396         break ;
397       }
398       case 2:
399       {
```

```
400         jj = 0 ;
401         for ( j = gh ; j < jl + gh ; j++ )
402         {
403           for ( i = gh ; i < il + gh ; i++ )
404           {
405             for ( k = gh ; k <= kl + gh ; k++ )
406             {
407               ii                = cidx(i,j,k,il,jl,gh) ;
408               c[ii].flux[2][0] = frho[nn*nfm+jj] ;
409               c[ii].flux[2][1] = frhou[nn*nfm+jj] ;
410               c[ii].flux[2][2] = frhov[nn*nfm+jj] ;
411               c[ii].flux[2][3] = frhow[nn*nfm+jj] ;
412               c[ii].flux[2][4] = frhoe[nn*nfm+jj] ;
413               jj++ ;
414             }
415           }
416         }
417         break ;
418       }
419       default:
420       {
421         printf("Invalid value of nn in flux().\n") ;
422         exit(1) ;
423       }
424     }
425   }
426
427
428 /*
429 FREE MEMORY -- causing a segfault as of 20 August 2013 on zhukovsky
430 28 august 2012 --no segfault on GPUNode01
431 */
432
433
434   free(da) ;
435   free(rhol) ;
436   free(rhor) ;
437   free(rhoul) ;
438   free(rhour) ;
439   free(rhovl) ;
440   free(rhovr) ;
441   free(rhowl) ;
442   free(rhowr) ;
443   free(rhoel) ;
444   free(rhoer) ;
445   free(nx) ;
446   free(ny) ;
447   free(nz) ;
448   free(sx) ;
449   free(sy) ;
450   free(sz) ;
451   free(tx) ;
452   free(ty) ;
453   free(tz) ;
454   free(frho) ;
455   free(frhou) ;
456   free(frhov) ;
457   free(frhow) ;
```

```
458   free(frhoe) ;
459
460
461 }
```

8.3.2 fluxh.cu

```
1  /*
2
3    function fluxd
4
5    computes flux on GPU
6
7    definitions
8
9     hl, hr    total enthalpy at left and right faces
10    ul, ur    ubar at left and right faces
11    vl, vr    vbar at left and right faces
12    wl, wr    wbar at left and right faces
13    pl, pr    pressure at left and right faces
14    qql, qqr  kinetic energy at left and right faces
15    rl, rr    density at left and right faces
16    rli, rri  inverse of density at left and right faces
17    rel, rer  rhoe at left and right faces
18
19    at        atilda
20    at2       atilda*atilda
21    ht        htilda
22    lt[]      abs(lambdatilda[])
23    qt        0.5*(ut*ut+vt*vt+wt*wt)
24    rm[][]    Roe matrix
25    sqrl      sqrt(rl)
26    sqrr      sqrt(rr)
27    ut,vt,wt  ubartilda, vbartilda, wbartilda
28
29    dr[]      delta R[]
30
31    left face    the face whose outwards normal points in the negative
32                 direction of the transformed coordinate (e.g., negative xi)
33    right face   the face whose outwards normal points in the positive
34                 direction of the transformed coordinate (e.g., positive xi)
35
36
37 */
38 /*
39   includes
40 */
41 #include "defs.h"
42 #include "extern.h"
43
44
45 __device__ int findex(int nn, int i, int nfm)
46 {
47   return nn*nfm + i ;
48 }
49
```

```
50
51 __global__ void fluxd(double *dad, double *rhold, double *rhord,
52                       double *rhould, double *rhourd, double *rhovld,
53                       double *rhovrd, double *rhowld, double *rhowrd,
54                       double *rhoeld, double *rhoerd, double *nxd,
55                       double *nyd, double *nzd, double *sxd,
56                       double *syd, double *szd, double *txd,
57                       double *tyd, double *tzd, double *frhod,
58                       double *frhoud, double *frhovd, double *frhowd,
59                       double *frhoed, int *nfd, int mm, int gh, int il,
60           int jl, int kl)
61 {
62   int    i, ii, i1, i2, nn, nfi, nfj, nfk, nft, nfm, nfa, nfr ;
63
64   double at, at2, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, dr[5],
65          gamma, h[5], hl, hr, ht, lt[5], pl, pr, qql, qqr, qt,
66          rel, rer, rl, rli, rm[5][5], rr, rri, sqrl, sqrr, ul,
67          ur, ut, vl, vr, vt, wl, wr, wt ;
68
69   nfi = (il+1)*jl*kl ;
70   nfj = il*(jl+1)*kl ;
71   nfk = il*jl*(kl+1) ;
72   nft = nfi + nfj +nfk ;
73   nfm = MAX(nfi,nfj) ;
74   nfm = MAX(nfm,nfk) ;
75   nfa = 3*nfm ;
76
77
78  nn = threadIdx.x ;
79  i  = blockIdx.x  ;
80  switch (nn)
81   {
82     case 0:
83       nfd[nn] = (il+1)*jl*kl ;
84       break ;
85     case 1:
86       nfd[nn] = il*(jl+1)*kl ;
87       break ;
88     case 2:
89       nfd[nn] = il*jl*(kl+1) ;
90       break ;
91   }
92
93   gamma = 1.4 ;
94
95   if ( i < nfd[nn] )
96   {
97     ii  = findex(nn, i, nfm) ;
98     rl  = rhold[ii] ;
99     rli = 1./rl ;
100    ul  = rli*(rhould[ii]*nxd[ii] + rhovld[ii]*nyd[ii] + rhowld[ii]*nzd[ii]) ;
101    vl  = rli*(rhould[ii]*sxd[ii] + rhovld[ii]*syd[ii] + rhowld[ii]*szd[ii]) ;
102    wl  = rli*(rhould[ii]*txd[ii] + rhovld[ii]*tyd[ii] + rhowld[ii]*tzd[ii]) ;
103    rel = rhoeld[ii] ;
104
105    qql = 0.5*rli*rli*(rhould[ii]*rhould[ii]+rhovld[ii]*rhovld[ii]+
106                       rhowld[ii]*rhowld[ii]) ;
107    pl  = (gamma-1.0)*(rel-rl*qql) ;
```

```
108     hl  = (gamma/(gamma-1.0))*pl*rli + qql ;
109
110     rr  = rhord[ii] ;
111     rri = 1./rr ;
112     ur  = rri*(rhourd[ii]*nxd[ii] + rhovrd[ii]*nyd[ii] + rhowrd[ii]*nzd[ii]) ;
113     vr  = rri*(rhourd[ii]*sxd[ii] + rhovrd[ii]*syd[ii] + rhowrd[ii]*szd[ii]) ;
114     wr  = rri*(rhourd[ii]*txd[ii] + rhovrd[ii]*tyd[ii] + rhowrd[ii]*tzd[ii]) ;
115     rer = rhoerd[ii] ;
116
117     qqr = 0.5*rri*rri*(rhourd[ii]*rhourd[ii]+rhovrd[ii]*rhovrd[ii]+
118                        rhowrd[ii]*rhowrd[ii]) ;
119     pr  = (gamma-1.0)*(rer-rr*qqr) ;
120     hr  = (gamma/(gamma-1.0))*pr*rri + qqr ;
121
122     sqrl = sqrt(fabs(rl)) ;
123     sqrr = sqrt(fabs(rr)) ;
124
125     c1  = 1./(sqrl+sqrr) ;
126     ut  = c1*(sqrl*ul+sqrr*ur) ;
127     vt  = c1*(sqrl*vl+sqrr*vr) ;
128     wt  = c1*(sqrl*wl+sqrr*wr) ;
129     qt  = 0.5*(ut*ut+vt*vt+wt*wt) ;
130     ht  = c1*(sqrl*hl+sqrr*hr) ;
131     at2 = (gamma-1.)*(ht-qt) ;
132     at  = sqrt(fabs(at2)) ;
133
134     lt[0] = fabs(ut) ;
135     lt[1] = lt[0] ;
136     lt[2] = lt[0] ;
137     lt[3] = fabs(ut+at) ;
138     lt[4] = fabs(ut-at) ;
139
140     /*  harten correction  */
141
142     for ( i1 = 0 ; i1 < 5 ; i1++ )
143     {
144       if ( lt[i1] < 2.0*EPSILON*at )
145       {
146         lt[i1] = lt[i1]*lt[i1]/(4.0*EPSILON*at) + EPSILON*at ;
147       }
148     }
149
150     /*  roe matrix  */
151
152     /*  note that the indices for lt and rm are one less than in the notes  */
153
154     c1       = (gamma-1.)/(2.*at2) ;
155     c2       = lt[3] + lt[4] - 2.*lt[2] ;
156     c3       = ut/(2.*at) ;
157     c4       = 1./(2.*at) ;
158     rm[0][0] = lt[2] + c1*qt*c2 + c3*(lt[4]-lt[3]) ;
159     rm[0][1] = -c1*ut*c2 - c4*(lt[4]-lt[3]) ;
160     rm[0][2] = -c1*vt*c2 ;
161     rm[0][3] = -c1*wt*c2 ;
162     rm[0][4] = c1*c2 ;
163
164     c5       = (ut+at)*lt[3] + (ut-at)*lt[4] - 2.*ut*lt[2] ;
165     c6       = (ut-at)*lt[4] - (ut+at)*lt[3] ;
```

```
166     rm[1][0] = ut*lt[2] + c1*qt*c5 + c3*c6 ;
167     rm[1][1] = -(c1*ut*c5 + c4*c6) ;
168     rm[1][2] = -c1*vt*c5 ;
169     rm[1][3] = -c1*wt*c5 ;
170     rm[1][4] =  c1*c5 ;
171
172     rm[2][0] = vt*(lt[2]-lt[0] + c1*qt*c2 + c3*(lt[4]-lt[3])) ;
173     rm[2][1] = vt*(-c1*ut*c2 + c4*(lt[3]-lt[4])) ;
174     rm[2][2] = lt[0] - c1*vt*vt*c2 ;
175     rm[2][3] = -c1*vt*wt*c2 ;
176     rm[2][4] = c1*vt*c2 ;
177
178     rm[3][0] = wt*(lt[2]-lt[1] + c1*qt*c2 + c3*(lt[4]-lt[3])) ;
179     rm[3][1] = wt*(-c1*ut*c2 + c4*(lt[3]-lt[4])) ;
180     rm[3][2] = -c1*vt*wt*c2 ;
181     rm[3][3] = lt[1] - c1*wt*wt*c2 ;
182     rm[3][4] = c1*wt*c2 ;
183
184     c7       = ht + ut*at ;
185     c8       = ht - ut*at ;
186     c9       = c7*lt[3] + c8*lt[4] - 2.*qt*lt[2] ;
187     c10      = c7*lt[3] - c8*lt[4] ;
188     rm[4][0] = c1*qt*c9 - c3*c10 + qt*lt[2] - vt*vt*lt[0] - wt*wt*lt[1] ;
189     rm[4][1] = -c1*ut*c9 + c4*c10 ;
190     rm[4][2] = vt*(-c1*c9 + lt[0]) ;
191     rm[4][3] = wt*(-c1*c9 + lt[1]) ;
192     rm[4][4] = c1*c9 ;
193
194     dr[0] = rl - rr ;
195     dr[1] = rl*ul - rr*ur ;
196     dr[2] = rl*vl - rr*vr ;
197     dr[3] = rl*wl - rr*wr ;
198     dr[4] = rel - rer ;
199
200     for (i1 = 0 ; i1 < 5 ; i1++ )
201     {
202       h[i1] = 0. ;
203       for ( i2 = 0 ; i2 < 5 ; i2++ )  h[i1] += rm[i1][i2]*dr[i2] ;
204     }
205
206     h[0] = 0.5*(rl*ul + rr*ur + h[0]) ;
207     h[1] = 0.5*(rl*ul*ul + pl + rr*ur*ur + pr + h[1]) ;
208     h[2] = 0.5*(rl*vl*ul + rr*vr*ur + h[2]) ;
209     h[3] = 0.5*(rl*wl*ul + rr*wr*ur + h[3]) ;
210     h[4] = 0.5*((rel+pl)*ul + (rer+pr)*ur + h[4]) ;
211
212
213     frhod[ii]  = h[0]*dad[ii] ;
214     frhoud[ii] = (h[1]*nxd[ii] + h[2]*sxd[ii] + h[3]*txd[ii])*dad[ii] ;
215     frhovd[ii] = (h[1]*nyd[ii] + h[2]*syd[ii] + h[3]*tyd[ii])*dad[ii] ;
216     frhowd[ii] = (h[1]*nzd[ii] + h[2]*szd[ii] + h[3]*tzd[ii])*dad[ii] ;
217     frhoed[ii] = h[4]*dad[ii] ;
218   }
219   __syncthreads() ;
220 return ;
221 }
```

8.3.3 main.cu

```
/*

  ----
  main
  ----

  21 jun 13   begin writing code

  description

  the Euler code is developed for the hybrid CPU/GPU system

  the basic structure of the code is:

  o  the Euler equations are solved on a single block structured grid
  o  the spatial reconstruction is first order accurate
  o  the temporal integration is second order Runge-Kutta
  o  the flux calculation is performed on the GPU
  o  the Runge-Kutta predictor and corrector steps are performed on the CPU
  o  the boundary conditions are periodic in all three directions

  the flow variables are non-dimensionalized by

  o  rho_infinity
  o  T_infinity
  o  U_infinity
  o  L

  this requires the value of the Mach number

     U_infinity/sqrt(gamma*RGAS*T_infinity)

  to be read by input() as d.machinf

  the velocity scale U_infinity can be chosen to be anything desired.
  a common value is the reference speed of sound which therefore
  implies that d.machinf equals unity

  usage

  a.out

*/

#include "defs.h"
#include "global.h"

/*
  function prototyping
*/

void filopn() ;
void input() ;
void output() ;
void rk2(double, int, int *, double *, double *, double *, double *,
  double *, double *, double *, double *, double *, double *,
```

```
  double *, double *, double *, double *, double *, double *,
  double *, double *, double *, double *, double *, double *,
  double *, double *, double *) ;
void tecprep() ;
double timestep(int) ;
void unitv() ;

/*
  program
*/

main(int argc, char **argv)
{
  int i, f, gh, il, jl, kl, m, mm, nfm, *nfd ;
  double dt ;
  double *dad, *rhold, *rhord, *rhould, *rhourd,
   *rhovld, *rhovrd, *rhowld, *rhowrd, *rhoeld,
   *rhoerd, *nxd, *nyd, *nzd, *sxd, *syd, *szd,
   *txd, *tyd, *tzd, *frhod, *frhoud, *frhovd,
   *frhowd, *frhoed;
  cudaError_t stat ;
  cudaError_t cudaGetLastError(void) ;


  filopn() ;

  input() ;

  unitv() ;


  mm  = m ;
  gh  = d.gh ;
  il  = d.il ;
  jl  = d.jl ;
  kl  = d.kl ;

  nfm = MAX((il+1)*jl*kl,il*(jl+1)*kl) ;
  nfm = MAX(nfm,il*jl*(kl+1)) ;
  size_t length = 3*nfm*sizeof(double) ;

  /*  nfd  */
  if (cudaMalloc( (void**) &nfd, 3*sizeof(int) ) != cudaSuccess)
  {
    printf("Device memory allocation failure for nfd\n") ;
    exit(1) ;
  }


    /*  dad  */
    stat =  cudaMalloc( (void**) &dad, length ) ;
    if (stat != cudaSuccess)
    {
      printf("Device memory allocation failure for dad\n") ;
      exit(1) ;
    }
    /*  rhod  */
    if ( cudaMalloc( (void**) &rhold, length) != cudaSuccess )
```

```
115     {
116       printf("Device memory allocation failure for rhold\n") ;
117       exit(1) ;
118     }
119     if ( cudaMalloc( (void**) &rhord, length ) != cudaSuccess )
120     {
121       printf("Device memory allocation failure for rhord\n") ;
122       exit(1) ;
123     }
124     /*  rhoud  */
125     if ( cudaMalloc( (void**) &rhould, length ) != cudaSuccess )
126     {
127       printf("Device memory allocation failure for rhould\n") ;
128       exit(1) ;
129     }
130     if ( cudaMalloc( (void**) &rhourd, length ) != cudaSuccess )
131     {
132       printf("Device memory allocation failure for rhourd\n") ;
133       exit(1) ;
134     }
135     /*  rhovd  */
136     if ( cudaMalloc( (void**) &rhovld, length ) != cudaSuccess )
137     {
138       printf("Device memory allocation failure for rhovld\n") ;
139       exit(1) ;
140     }
141     if ( cudaMalloc( (void**) &rhovrd, length ) != cudaSuccess )
142     {
143      printf("Device memory allocation failure for rhovrd\n") ;
144       exit(1) ;
145     }
146     /*  rhowd  */
147     if ( cudaMalloc( (void**) &rhowld, length ) != cudaSuccess )
148     {
149       printf("Device memory allocation failure for rhowld\n") ;
150       exit(1) ;
151     }
152     if ( cudaMalloc( (void**) &rhowrd, length ) != cudaSuccess )
153     {
154       printf("Device memory allocation failure for rhowrd\n") ;
155       exit(1) ;
156     }
157     /*  rhoed  */
158     if ( cudaMalloc( (void**) &rhoeld, length ) != cudaSuccess )
159     {
160       printf("Device memory allocation failure for rhoeld\n") ;
161       exit(1) ;
162     }
163     if ( cudaMalloc( (void**) &rhoerd, length ) != cudaSuccess )
164     {
165       printf("Device memory allocation failure for rhoerd\n") ;
166       exit(1) ;
167     }
168
169     /*  nxd  */
170    if ( cudaMalloc( (void**) &nxd, length ) != cudaSuccess )
171     {
172       printf("Device memory allocation failure for nxd\n") ;
```

```
173       exit(1) ;
174     }
175     /*  nyd  */
176     if ( cudaMalloc( (void**) &nyd, length ) != cudaSuccess )
177     {
178       printf("Device memory allocation failure for nyd\n") ;
179       exit(1) ;
180     }
181     /*  nzd  */
182     if ( cudaMalloc( (void**) &nzd, length ) != cudaSuccess )
183     {
184       printf("Device memory allocation failure for nzd \n") ;
185       exit(1) ;
186     }
187     /*  sxd  */
188     if ( cudaMalloc( (void**) &sxd, length ) != cudaSuccess )
189     {
190       printf("Device memory allocation failure for sxd\n") ;
191       exit(1) ;
192     }
193       /*  syd  */
194     if ( cudaMalloc( (void**) &syd, length ) != cudaSuccess )
195     {
196       printf("Device memory allocation failure for syd\n") ;
197       exit(1) ;
198     }
199     /*  szd  */
200     if ( cudaMalloc( (void**) &szd, length ) != cudaSuccess )
201     {
202       printf("Device memory allocation failure for szd\n") ;
203       exit(1) ;
204     }
205     /*  txd  */
206     if ( cudaMalloc( (void**) &txd, length ) != cudaSuccess )
207     {
208       printf("Device memory allocation failure for txd\n") ;
209       exit(1) ;
210     }
211     /*  tyd  */
212     if ( cudaMalloc( (void**) &tyd, length ) != cudaSuccess )
213     {
214       printf("Device memory allocation failure for tyd\n") ;
215       exit(1) ;
216     }
217     /*  tzd  */
218     if ( cudaMalloc( (void**) &tzd, length ) != cudaSuccess )
219     {
220       printf("Device memory allocation failure for tzd\n") ;
221       exit(1) ;
222     }
223     /*  frhod  */
224     if ( cudaMalloc( (void**) &frhod, length ) != cudaSuccess )
225     {
226       printf("Device memory allocation failure for frhod\n") ;
227       exit(1) ;
228     }
229     /*  frhoud  */
230     if ( cudaMalloc( (void**) &frhoud, length ) != cudaSuccess )
```

```
231     {
232       printf("Device memory allocation failure for frhoud\n") ;
233       exit(1) ;
234     }
235     /*  frhovd  */
236     if ( cudaMalloc( (void**) &frhovd, length ) != cudaSuccess )
237     {
238       printf("Device memory allocation failure for frhovd\n") ;
239       exit(1) ;
240     }
241     /*  frhowd  */
242     if ( cudaMalloc( (void**) &frhowd, length ) != cudaSuccess )
243     {
244       printf("Device memory allocation failure for frhowd\n") ;
245       exit(1) ;
246     }
247     /*  frhoed  */
248     if ( cudaMalloc( (void**) &frhoed, length ) != cudaSuccess )
249     {
250       printf("Device memory allocation failure for frhoed\n") ;
251       exit(1);
252     }
253
254
255
256
257
258
259   for ( i = 0 ; i < d.iter ; i++ )
260   {
261     dt = timestep(i) ;
262     rk2(dt, i, nfd, dad, rhold, rhord, rhould, rhourd,
263    rhovld, rhovrd, rhowld, rhowrd, rhoeld,
264    rhoerd, nxd, nyd, nzd, sxd, syd, szd,
265    txd, tyd, tzd, frhod, frhoud, frhovd, frhowd, frhoed) ;
266     d.time += dt ;
267     if ( (i > 0)&(i%d.check==0) )  output() ;
268   }
269
270
271
272
273
274
275
276
277 /*   BK 10 July 2013 (2)
278        Need to free allocated memory before program exit
279 */
280   cudaFree(nfd) ;
281   cudaFree(dad) ;
282   cudaFree(rhold) ;
283   cudaFree(rhord) ;
284   cudaFree(rhould) ;
285   cudaFree(rhourd) ;
286   cudaFree(rhovld) ;
287   cudaFree(rhovrd) ;
288   cudaFree(rhowld) ;
```

```
289   cudaFree(rhowrd) ;
290   cudaFree(rhoeld) ;
291   cudaFree(rhoerd) ;
292   cudaFree(nxd) ;
293   cudaFree(nyd) ;
294   cudaFree(nzd) ;
295   cudaFree(sxd) ;
296   cudaFree(syd) ;
297   cudaFree(szd) ;
298   cudaFree(txd) ;
299   cudaFree(tyd) ;
300   cudaFree(tzd) ;
301   cudaFree(frhod) ;
302   cudaFree(frhoud) ;
303   cudaFree(frhovd) ;
304   cudaFree(frhowd) ;
305   cudaFree(frhoed) ;
306
307
308
309   tecprep() ;
310
311   output() ;
312
313   exit(0) ;
314 }
```

8.3.4 rk2.cu

```
1  /*
2
3    runge-kutta integration (second order)
4
5  */
6  /*
7    includes
8  */
9  #include "defs.h"
10 #include "extern.h"
11 /*
12   externals
13 */
14 extern void bc(int) ;
15 extern int cidx(int, int, int, int, int, int) ;
16 extern void flux(int, int, int *, double *, double *, double *, double *,
17   double *, double *, double *, double *, double *, double *,
18   double *, double *, double *, double *, double *, double *,
19   double *, double *, double *, double *, double *, double *,
20   double *, double *, double *) ;
21
22 void rk2(double dt, int step, int *nfd, double *dad, double *rhold,
23     double *rhord, double *rhould, double *rhourd,
24     double *rhovld, double *rhovrd, double *rhowld,
25     double *rhowrd, double *rhoeld, double *rhoerd,
26     double *nxd, double *nyd, double *nzd, double *sxd,
27     double *syd, double *szd, double *txd, double *tyd,
```

```
28     double *tzd, double *frhod, double *frhoud,
29     double *frhovd, double *frhowd, double *frhoed)
30 {
31   int    i, ii, il, i1, i2, i3, i4, j, jl, k, kl, m, ne, gh, gh2 ;
32   double dt2, rhs[5] ;
33
34   gh  = d.gh ;
35   gh2 = 2*gh ;
36   il  = d.il ;
37   jl  = d.jl ;
38   kl  = d.kl ;
39   dt2 = 0.5*dt ;
40
41   /*  step no. 1  */
42
43   m = 0 ;
44   bc(m) ;
45   flux( m, step, nfd, dad, rhold, rhord, rhould, rhourd, rhovld, rhovrd,
46      rhowld, rhowrd, rhoeld, rhoerd, nxd, nyd, nzd, sxd, syd,
47      szd, txd, tyd, tzd, frhod, frhoud, frhovd, frhowd,
48      frhoed) ;
49
50   for ( k = gh ; k < gh + kl ; k++ )
51   {
52     for ( j = gh ; j < gh + jl ; j++ )
53     {
54       for ( i = gh ; i < gh + il ; i++ )
55       {
56         i1 = cidx(i,j,k,il,jl,gh) ;
57         i2 = cidx(i+1,j,k,il,jl,gh) ;
58         i3 = cidx(i,j+1,k,il,jl,gh) ;
59         i4 = cidx(i,j,k+1,il,jl,gh) ;
60         for ( ne = 0 ; ne < 5 ; ne++ )
61         {
62           /*  note that the minus sign has been incorporated  */
63           rhs[ne] = c[i1].inv* (c[i1].flux[0][ne] - c[i2].flux[0][ne] +
64                                 c[i1].flux[1][ne] - c[i3].flux[1][ne] +
65                                 c[i1].flux[2][ne] - c[i4].flux[2][ne] ) ;
66         }
67         c[i1].rho[1]  = c[i1].rho[0]  + dt2*rhs[0] ;
68         c[i1].rhou[1] = c[i1].rhou[0] + dt2*rhs[1] ;
69         c[i1].rhov[1] = c[i1].rhov[0] + dt2*rhs[2] ;
70         c[i1].rhow[1] = c[i1].rhow[0] + dt2*rhs[3] ;
71         c[i1].rhoe[1] = c[i1].rhoe[0] + dt2*rhs[4] ;
72       }
73     }
74   }
75
76   /*  step no. 2  */
77
78   m = 1 ;
79   bc(m) ;
80   flux( m, step, nfd, dad, rhold, rhord, rhould, rhourd, rhovld, rhovrd,
81      rhowld, rhowrd, rhoeld, rhoerd, nxd, nyd, nzd, sxd, syd,
82      szd, txd, tyd, tzd, frhod, frhoud, frhovd, frhowd,
83      frhoed) ;
84
85   for ( k = gh ; k < gh + kl ; k++ )
```

```
86  {
87    for ( j = gh ; j < gh + jl ; j++ )
88    {
89      for ( i = gh ; i < gh + il ; i++ )
90      {
91        i1 = cidx(i,j,k,il,jl,gh) ;
92        i2 = cidx(i+1,j,k,il,jl,gh) ;
93        i3 = cidx(i,j+1,k,il,jl,gh) ;
94        i4 = cidx(i,j,k+1,il,jl,gh) ;
95        for ( ne = 0 ; ne < 5 ; ne++ )
96        {
97          /*  note that the minus sign has been incorporated  */
98          rhs[ne] = c[i1].inv* (c[i1].flux[0][ne] - c[i2].flux[0][ne] +
99                                c[i1].flux[1][ne] - c[i3].flux[1][ne] +
100                               c[i1].flux[2][ne] - c[i4].flux[2][ne] ) ;
101       }
102       c[i1].rho[1]  = c[i1].rho[0]  + dt*rhs[0] ;
103       c[i1].rhou[1] = c[i1].rhou[0] + dt*rhs[1] ;
104       c[i1].rhov[1] = c[i1].rhov[0] + dt*rhs[2] ;
105       c[i1].rhow[1] = c[i1].rhow[0] + dt*rhs[3] ;
106       c[i1].rhoe[1] = c[i1].rhoe[0] + dt*rhs[4] ;
107     }
108   }
109 }
110
111 /* store  */
112
113 for ( k = 0 ; k < gh2 + kl ; k++ )
114 {
115   for ( j = 0 ; j < gh2 + jl ; j++ )
116   {
117     for ( i = 0 ; i < gh2 + il ; i++ )
118     {
119       ii = cidx(i,j,k,il,jl,gh) ;
120       c[ii].rho[0]  = c[ii].rho[1] ;
121       c[ii].rhou[0] = c[ii].rhou[1] ;
122       c[ii].rhov[0] = c[ii].rhov[1] ;
123       c[ii].rhow[0] = c[ii].rhow[1] ;
124       c[ii].rhoe[0] = c[ii].rhoe[1] ;
125     }
126   }
127 }
128
129 }
```

References

[1] Intel, "Moore's Law 40th Anniversary," Press Release, April 2005.

[2] NVIDIA, *CUDA C Programming Guide v5.5*, July 2013.

[3] Knight, D., *Elements of Numerical Methods for Compressible Flows*, Cambridge University Press, 2006.

[4] Intel, *Intel 8088 User Manual*, August 1990.

[5] Harriot, L. R., "Limits of Lithography," *Proceeding of the IEEE*, Vol. 89, 2001, pp. 366–374.

[6] Brunner, T. A., "Why optical lithography will live forever," *Journal of Vacuum Science Technology year =*.

[7] Sanders, J. and Kandrot, E., *CUDA By Example*, Addison Wesley, 2011.

[8] Patnaik, Corrigan, O. S. and Fyfe, "Efficient Utilization of a CPU-GPU Cluster," *American Institute of Aeronautics and Astronautics*, Vol. 50, 2012, pp. 563.

[9] Smith, W. and Cox, K., "Performance Tradeoff Considerations in a Graphics Processing Unit (GPU) Implementation of a Low-Detectable Aircraft Sensor System," *American Institute of Aeronautics and Astronautics*, Vol. 51, 2013, pp. 374.

[10] Löhner, C. and Baum, "Large Scale Blast Calculations on GPU Clusters," *American Institute of Aeronautics and Astronautics*, Vol. 50, 2012, pp. 565.

[11] Roe, P., "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," *Journal of Computational Physics*, Vol. 43, 1981, pp. 357–372.