EXPERIMENTAL AND NUMERICAL ANALYSIS OF TRANSIENT LIQUID SLUG MOTION IN A VOIDED LINE

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

OF

THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÖZGÜR UĞRAŞ BARAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE

IN

THE DEPARTMENT OF CIVIL ENGINEERING

DECEMBER 1999

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Tayfur ÖZTÜRK
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Mustafa TOKYAY
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Zafer BOZKUŞ
Co-Supervisor

Prof. Dr. Metin GER
Supervisor

Examining Committee Members

Prof. Dr. Metin GER

Assist. Prof. Dr. Zafer BOZKUŞ

Assoc. Prof. Dr. İsmail AYDIN

Prof. Dr. Mustafa GÖĞÜŞ

Prof. Dr. Turgut TOKDEMİR

ABSTRACT

EXPERIMENTAL AND NUMERICAL ANALYSIS OF TRANSIENT LIQUID SLUG MOTION IN A VOIDED LINE

Baran, Özgür Uğraş

M.Sc., Department of Civil Engineering

Supervisor: Prof. Dr. Metin GER

Co-Supervisor: Assist. Prof. Dr. Zafer BOZKUŞ

December 1999, 190 pages

In this thesis, the hydrodynamics of a solitary transient slug in a voided line was investigated numerically and experimentally. In the experiment, the liquid slugs of various lengths were propelled into an inclined smooth steel pipe under several different air driving pressures. The pipe segment terminates in a bend; the pressure time histories are recorded at this elbow. The recorded peak pressures are correlated to the initial tank pressures and pipe and slug geometry. The flow is investigated numerically using several Godunov type schemes, namely, basic Godunov Scheme, and TVD based Weighted Average Flux (WAF) method and two MUSCL methods. These schemes employ Godunov's approach facilitating exact and HLLC type solution methods solving the Riemann problem of gas dynamics equations. Furthermore the recorded peak pressures at the elbow are compared with the output of the numerical analysis.

Keywords: Liquid Slug, Transient Flow, Godunov's Approach, WAF method, MUSCL method, TVD, Gas dynamics, Riemann problem, HLLC approximate state Riemann solver

ÖZ

BOŞ BİR BORU SİSTEMİNDE DEĞİŞKEN AKIMLI BİR SU KÜTLESİNİN HAREKETİNİN DENEYSEL VE NÜMERİK OLARAK İNCELENMESİ

Baran, Özgür Uğraş
Yüksek Lisans, İnşaat Mühendisliği Bölümü
Tez Danışmanı: Prof.Dr. Metin GER
Yardımcı Danışman: Yar. Doç. Dr. Zafer BOZKUŞ

Aralık 1999, 190 sayfa

Bu tez çalışmasında, tekil bir su paketinin boş bir boru içerisindeki hareketinin hidrodinamiği, hem numerik hem de deneysel olarak incelenmiştir. Deneylerde, değişik uzunluklardaki su paketleri, farklı hava basınçları yardımıyla boş bir boruda hızlandırılmıştır. Boru, bir dirsekle sonlandırılmış ve bu dirsekteki basıncın zaman içerisindeki değişimi kaydedilmiştir. Kaydedilen en yüksek basınçlar, tank basınçları ve boru ve su paketi geometrisi ile ilişkilendirilmiştir. Akım, Godunov yaklaşımı kullanan yöntemlerden, temel Godunov yaklaşımı, ile TVD esaslı olan Ağırlıklı Ortalama Akı (WAF) ve iki farklı MUSCL tipi yöntem kullanılarak bir boyutlu olarak çözülmüştür. Bu metodlarda gaz dinamiği denklemlerine uyarlanmış Riemann probleminin kesin ve HLLC yöntemi ile yaklaşık çözümlerinden yaralanılmaktadır. Ölçülen en yüksek basınçlar, numerik çözümlerle karşılaştırılmıştır.

Anahtar Kelimeler: Su paketi, Değişken akım Godunov yaklaşımı, WAF yöntemi, MUSCL yöntemi, TVD, Gaz dinamiği, Riemann problemi, HLLC yaklaşık Riemann çözücüsü

To my father…

ACKNOWLEDGEMENTS

I express my sincere appreciation to my supervisors Prof. Dr. Metin GER and Assist. Prof. Dr. Zafer Bozkuş for their guidance, suggestions and comments throughout my research.

I would like to thank to the technical staff of the Hydraulics Laboratory, particularly to Cengiz Tufaner, Hüseyin Değirmen for their technical assistance, close collaboration and moral support.

Special thanks goes to Dilek Yıldız, for the long nights we spent together studying, and also the way she believed in me even when I didn't.

A very special thanks extends to Kerem Pekkan for his invaluable advises which will always be appreciated.

I am also grateful to for all of the laboratory personnel who worried about me. Furthermore I would also thank to nightguards of METU who were worried about me, and - of course- for the tea constantly provided me with in the midnight!

Finally, I would like to thank to my family for their support, understanding, patience, and unshakable faith in me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

The liquid slug motion in a voided line is observed especially at piping systems that carry high pressure steam such as power plant piping. The damage caused by liquid slug trapped in such systems has been a concern of power industry. In such power plants, the energy is produced using steam turbines. When the system is shut down for maintenance purpose or any other reason, the steam condenses and accumulates in lower sections of the pipeline. As soon as the system is reactivated, the high-pressure steam accelerates these trapped water "slugs", and they hit the bends, elbows, partially open valves etc. This creates very high impact forces on these discontinuities, causing sometimes serious damages in the pipeline.

Damage in the system means interruption of the power production during a long reconstruction period, causing serious economic losses. Therefore, the estimation of the impact forces on critical sections of the pipeline such as bends becomes very important in order to prevent damages in the system. Prior to any

attempt of scrutinisation of this phenomenon, it must be noted that, the phenomenon is of very complex nature. Therefore, any mathematical analysis should also be checked with experimental studies. The experimental study need not be a full simulation of a power plant piping system, but it must be suitable to observe the behaviour of liquid slug through a line and at a regular bend. Furthermore, it is worth mentioning that the most needed liquid slug data that is those taken for large diameter pipes commonly used in power-plant piping.

There are some "handy" methods to estimate the loads at elbows in literature. However, most of the times "handy" means "rough". That is, the need for an accurate, convergent and reliable numerical method is obvious for the design stage of piping systems. The rapid development of the computer technology in last decades allows the application of very sophisticated methods. The incredible progress in computer hardware speed brings the development of new methods in computational fluid dynamics (CFD). Indeed, the applications using these advanced methods in CFD is more accurate and applicable than it was a decade ago.

There is not a wide range of studies on the behaviour the liquid slug motion in piping systems. The most recent attempts are by, Fenton [6], Fenton and Griffith [7], Bozkuş [1, 4], Bozkuş and Wiggert [2, 3, 5], Yang and Wiggert [15] among others. Two of these works, namely Bozkuş's, and Fenton's must be emphasized due to their original experimental and theoretical frameworks.

Fenton [6], in his master's thesis, analysed the behaviour of liquid slug motion in a 1" pipe. The pipe used is inclined 8% and can be filled with water into its lower section. This pipe is connected to a tank, which can be filled both with steam and air, after an orifice. There was a ball valve between orifice and the tank. When this valve

opened suddenly, the trapped water moves very rapidly and hits to the elbow at the end of the pipe, which is open to the atmosphere. This elbow is not rigidly connected to the pipe, but is connected to a support behind it, on which a strain gage force transducer installed. Fenton [6] measured and stored the force at the elbow in time electronically using this transducer directly. The experimental apparatus of Fenton [6] is given at the Figure I-1.

In the analytical study, Fenton [6, 7] used a simplified control volume of liquid slug for both while it is moving through the pipe and it is at the elbow. A simple conservation of momentum analysis gives the acceleration of the slug and force at the elbow. In the pipe, the driving force is the upstream pressure. Fenton assumed a varied pressure behind the slug, which is calculated as the choked pressure after the orifice. The varied pressure at the tank is calculated by the equation of state. This is a rough assumption but still useable since the pipe diameter is very small.

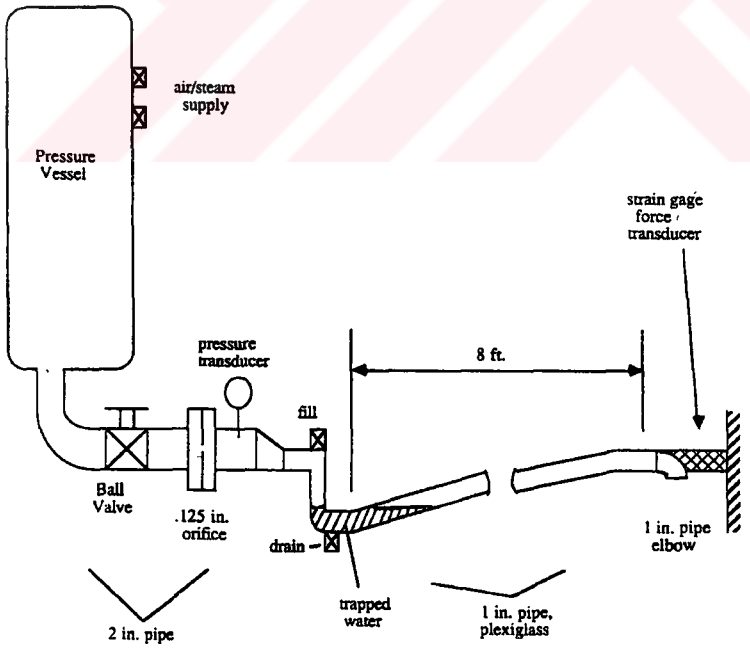

Figure I-1: The experimental apparatus used by Fenton [6]

Another significant attempt is made by Bozkuş [1]. In this study, 31ft long and 2inch diameter clear straight PVC pipe laid horizontally with a 90 degree elbow is connected to a ball valve. Behind the valve there was a pipe section with variable length, which is connected to an air tank. The experimental procedure is similar to that of Fenton's, when the ball valve opened suddenly, water trapped into the slug section accelerates by the pressurised air behind, hits the elbow and releases to the atmosphere. Despite Fenton [6], Bozkuş measured the pressure at the elbow. This pressure is then assumed to apply all over the elbow section.

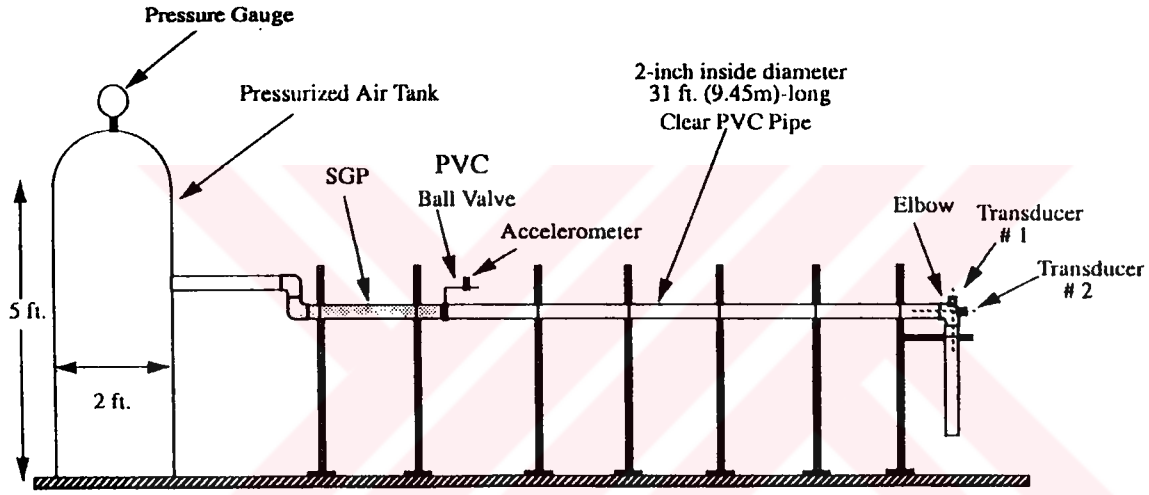

Figure I-2: The experimental apparatus used by Bozkuş [1]

In his analytical work, Bozkuş [1, 2, 3, 4, 5] used a control volume approach both for slug into pipe and at the elbow. For the motion of the gas, he used a method of characteristics based solution technique, which is a great leap over the method of Fenton [6,7] that uses constant pressure.

However, besides its advantages into the numerical work, the experimental study of Bozkuş, carries some disadvantages. He pointed out that for longer slugs,

slug separated into two pieces, acting as two slugs into the pipe. It must be noted, however that, even though this study is a well defined tool for slugs moving into horizontal lines, there are some critics in literature, that points out that this experiment is dependent on the valve opening speed.

A recent attempt, using numerical analysis, taking air entrainment into account is made by Yang and Wiggert [15] is worth mentioning. This study is tested against the data of Bozkuş [1]. In this interesting quasi 2-D study the water slug is modelled as number of concentric cylinders sliding through each other. As the inner cylinders moves faster than the outer ones, the air enters to the slug and the tail of the outer cylinders are considered as a holdup. At the elbow, the concentric cylinders keep on sliding each other, until the void between the cylinders reaches to the bend. Their solution generally underestimates the experimental study.

Observing the findings of former works on liquid slug problem and also the complex nature of it, it is concluded that, the solution procedure needs a combined wave-like solution technique. One of the objectives of this study is to develop such a numerical method.

The other objective of this study is to obtain a new set of data. A new experimental set-up, that is a combination of Fenton's [6] and Bozkuş's [1] set-up, avoiding the disadvantages of both was built. The set-up includes an inclined pipe like Fenton's, that prevents separation of slug, a pressure transducer at the end of the pipe like Bozkuş's, and a valve behind the slug to minimise the dependence on the valve opening speed.

The most significant development that this new set-up offers is the 4" pipe diameter. In former experiments, small diameter pipes (1 inch at Fenton's 2 inches at

Bozkuş's set-up) were used. However, in a power plant piping system larger diameter pipes are generally in use. Initial studies shows that significance of the frictional forces reduces with the increasing pipe diameter. Therefore, 10cm (4 inch) diameter pipe is used at the present study, in order to supply reliable data for power-plant piping design and risk analysis.

In what follows, the development of the theoretical background is given in Chapter 2. This chapter also gives the basic methodology of the numerical simulation of the liquid slug motion. A very detailed description of the numerical methods, including Godunov scheme, Weighted Average Flux (WAF) scheme and various MUSCL schemes, and the respective source codes are given in appendices A, B, E and F. In Chapter 3, the experimental methodology is explained in detail. The description of set-up briefed in this chapter, too. For detailed description of the set-up reader referred to the appendices C, D and F. In this chapter a dimensional analysis performed in order to establish a relationship among the peak pressure at the elbow to the upstream pressure and both to the pipe and slug geometry is presented. Also included in this chapter are the comparison of the present experimental findings and the former studies together with the comparison of the experimental observation and the numerical studies. The last and the fourth chapter discusses the present study and summarises recommendations for the future studies.

The present study, furthermore, aims at updating the hydraulics laboratory's data acquisition facilities. This study brings a new kind of experimental set-up; installation of the piezoelectric transducers and updating the data acquisition system, all of which explained in detail in Appendices C, and G.

CHAPTER II

THEORY AND THE MATHEMATICAL MODEL

II.1 INTRODUCTION

One of the goals of this study is to improve the level of sophistication and accuracy of the solution methods of the previous attempts. To achieve this goal, the dynamics of the slug body and the compressible gas upstream will be considered separately. The dynamics of the gas downstream of the slug is not considered since it does not interfere with the dynamics of the flow, at large. The dynamics of the slug is handled for two consecutive phases. The first phase is the period while the slug is moving through the pipe and the second one is when it just hits the elbow.

II.2 DYNAMICS OF THE SLUG

II.2.1 PRESSURE WAVES IN THE MOVING SLUG

The pressure waves travelling through the slug body is a very complex mechanism. When an upstream or a downstream pressure changes or an impact pressure is produced, the pressure waves propagate through the slug with the speed of sound, a. The propagation of pressure waves may be, in general, solved with a waterhammer solver as described by Streeter [18]. However, for the present study, since i) the slugs are very short; ii) the wave propagation speed of the water is much faster than the average wave speed of the air; and iii) the slug does not reach the terminal velocity in the pipe, there is no need for such a treatment. In other words, since the flow field is analogous to the establishment of steady flow in a pipe after sudden opening of a valve (Daugherty et. al. [16] section 13.4.) the wave propagation analysis within the moving slug is not considered within the present study.

II.2.2 SLUG MOVING THROUGH THE PIPE

Based on the above discussion, the slug is considered as a moving (rigid) body driven by the pressure difference between the upstream and downstream. Unlike the Bozkuş's study, slug is considered as constant mass, in other words, no holdup through the motion takes place. Following assumptions are made:

- Flow is one dimensional
- Fluid is incompressible, hence the density of the slug is constant. This assumption actually states that, the compressibility of the fluid is so small

that, the density changes are very small, but finite speed sound waves are observed.

- Friction factor for the pipe is taken as constant that is taken from the fully rough zone. This assumption is valid, since the slug reaches very high speeds immediately.
- Flow is one phase, in other words, fluid does not mix with air as the slug moves through the pipe.
- Pipe is rigid and fixed in space.

Using the above assumptions, the moving control volume given at the figure II.1 is used to determine the forces over the slug body. In this figure, P_1 and P_2 stands for upstream and downstream pressures, respectively, α is the pipe inclination, D is the pipe diameter, L is the slug length, m is the slug mass and τ is the wall shear stress.

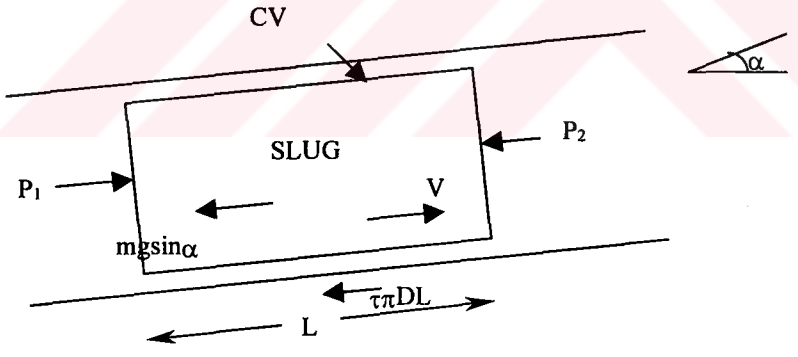

Figure II-1: Control Volume for the slug in the pipe

The force balance for the CV reveals that

$$\sum F = (P_1 - P_2)A - \tau\pi DL - mg\sin\alpha \qquad (1)$$

where $\Sigma F = m\frac{dV}{dt}$ such that V is the speed of the slug, m is the mass of the slug.

One must note that, the most dominant term in the total force balance equation is the first one, representing pressure force. Although the other terms have relatively smaller magnitudes considering the fairly short pipe used in the experiments these terms are not omitted in the calculations.

II.2.3 SLUG AT THE ELBOW

At the very instant a contact is established between the slug and the elbow's vertical face the pressure at the elbow can be determined using energy equation assuming that there is a stagnation on the elbow face (in other words at just the point where the transducers takes data as it is described at section III-1 and figures III-1 and III-2). Then dynamic pressure at the elbow can be written as,

$$P_e = P_0 + \tfrac{1}{2}\rho V^2 \qquad (2)$$

Where, P_e is the pressure at the elbow, P_0 is the slug pressure just upstream the elbow, V is the velocity of the slug and ρ is the density of the slug. It is worth noting that the density of the slug is taken as constant after the assumption that there is no air entry into the slug.

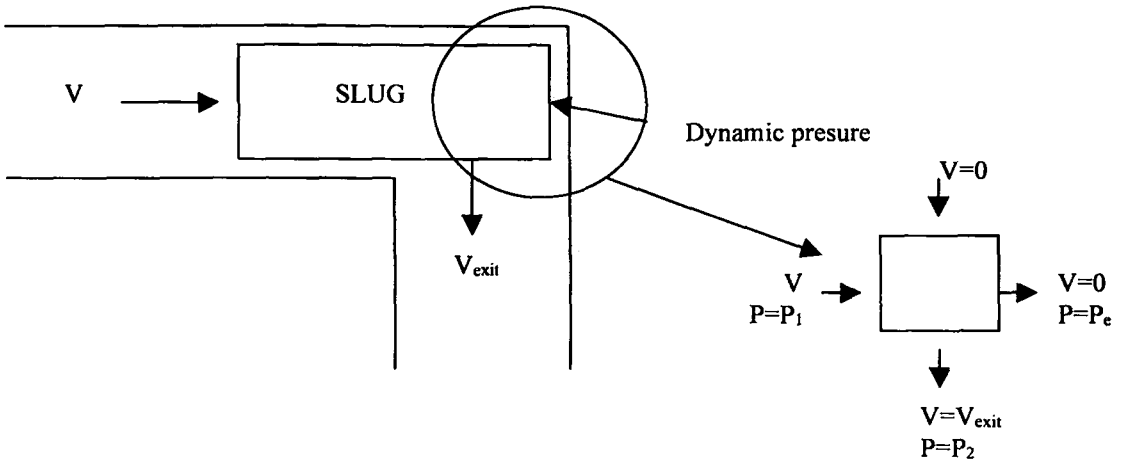

Figure II-2: Control Volume for the moment that the slug hits to the elbow

The flow field at this very instant that the slug hits the elbow resembles the situation when the pipe exit is closed suddenly by a valve (the elbow face) while the water is moving with the velocity V in the pipe. This is an instantaneous waterhammer problem the pressure rise due to the compressibility of the water can be found in many textbooks dealing with fluid transients. For example Streeter [18] gives the pressure rise as

$$\Delta P = \rho a \Delta V \qquad (3)$$

Where a is the speed of sound of the incompressible fluid, ΔP is the pressure rise due to the compressibility of the fluid, and ΔV is the local change in velocity.

Therefore, the dynamic pressure at the elbow face at the instant of first contact given in equation (2) must be corrected by the addition of ΔP such that

$$P_e = P_0 + \rho a \Delta V \qquad (4)$$

That is, the dynamic pressure may assume any value within the range defined by equation (3) and (4).

The last term in equation (4) (i.e. $\rho a \Delta V$) may assume some unrealistically large magnitudes. Indeed, some preliminary numerical runs show that this instantaneous pressure rise may assume values in the order of several hundred or even thousand bars. However, experimental observations do not support these large pressure increases. In other words, the assumption of sudden closure assumption for the waterhammer analogy is not justified. This assumption overestimates the force at bend.

In order to get a better estimate of the pressure rise, taking into account the exit way at the elbow, one may use the analogy of waterhammer with a surge tank. Considering the exit pipe as a surge tank of the same diameter with the pipe located just upstream of the valve (i.e. at the elbow) the head rise at the surge tank is given as by Daugherty et. al. [16]

$$V^2 = \frac{2gAD^2}{LA_s f^2}\left(1 - \frac{fA_s}{AD}z\right) - Ce^{-(fA_s/AD)z} \qquad (5)$$

where, L is the pipe length set equal to that of the slug, f is the friction factor, V is the velocity, g is the gravitational acceleration, D is the diameter of the pipe, z is the head rise in the surge tank interpreted as the pressure rise for our case. Furthermore, A_s is the cross sectional area of the surge tank and A is the area of the pipe, which are the same in this case. Then, the equation (5) reduces to:

$$V^2 = \frac{2gD^2}{Lf^2}\left(1 - \frac{f}{D}z\right) - Ce^{-(f/D)z} \qquad (6)$$

The only undefined parameter is C, is the integration constant. To determine this constant, the initial conditions V is set equal to slug velocity and z is set equal to zero. The value for the pressure head, z, obtained from equation (6) is considered as the pressure rise at the elbow.

This kind of an approach explains the pressure absorption mechanism at the elbow. As the pressure rise is generated by the impact, it is absorbed by the release of water immediately, as if there were a surge tank.

II.3 THE DYNAMICS OF THE GAS UPSTREAM THE SLUG

II.3.1 GOVERNING EQUATIONS FOR COMPRESSIBLE GAS SOLVER

The governing equations used for the compressible gas part are 1-D Euler equation:

$$U_t + F(U)_x = S(U) \qquad (7)$$

where:

$$U = \begin{bmatrix} \rho \\ \rho u \\ E \end{bmatrix}, \qquad F = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{bmatrix}, \qquad (8)$$

and S(U) is the source term and is equal to zero assuming an inviscid flow and subscripts refer to derivatives in respective variables.

II.3.2 GENERAL SOLUTION METHODOLOGIES

Due to their robustness and shock capturing abilities that are required for inviscid compressible gases characteristic based schemes are preferred. For this reason Flux Difference Splitting type methods which uses Godunov Approach, are considered. The original Godunov Approach is the basic one. Even tough there are some disadvantages of this method, such as difficulties in programming and slowness of the code, since it represents the physical processes realistically, it seems to be the most appropriate method. Therefore, this method constitutes the backbone of the Euler solver developed.

The numerical solution methods are, including higher order schemes, discussed in detail in Appendices A and B. Furthermore, all the codes developed are given in Appendices F and G.

Since solution methodology used is independent of the Euler solver in what follows the coupling of different flow regimes and interfaces is described

II.3.3 COMPLETE NUMERICAL SOLUTION OF THE PROBLEM

The physical domain was divided into three sub-domains. The first sub-domain is the air tank, the second one is the pipe section upstream the slug, and the third one is the slug itself. The solution strategy for each domain is explained below:

II.3.3.1 THE AIR TANK

For the air tank, the Euler equations can be solved via any of the methods presented above. The boundary conditions for the downstream of the tank is reflective since the tank wall is rigid. The handling of the reflective boundary conditions are

given at section is discussed in Appendix A. Since a large area change exists between the tank and the pipe, Handling the downstream boundary condition needs some attention. This area change leads to velocity difference between the tank and the pipe inlet such that:

$$U_{tank} = U_{pipe} \frac{A_{pipe}}{A_{tank}} \qquad (9)$$

where, U_{tank} is the velocity at the last node of the tank, U_{pipe} is the velocity at the first node of the pipe, A stands for the cross sectional area. This equation is nothing but the conservation of mass equation. Then, transmissive boundary conditions are used for the upstream boundary condition using U_{pipe}.

II.3.3.2 THE PIPE SECTION BEHIND THE SLUG

The second computational sub-domain is the pipe section upstream the slug. The downstream boundary condition for this sub-domain is a transmissive BC, while the upstream boundary is a moving boundary. The velocity of this boundary is equal to the velocity of slug. In other words, the slug is treated such that it is moving as a rigid wall in front of the expanding gas. In brief, the computational sub-domain is expanding to the upstream direction. Instead of using a moving grid technique, a fixed grid method is chosen for the sake of simplicity. The very downstream node is moved with the boundary such that the size of the cell is adjacent to the boundary change. When the size of the cell exceeds 1.5 times of he original size, its size is equated to the original and a new cell is added to account for the extension. Under any circumstances, cell size is kept large to prevent a very small time step, which is

dependent on the cell size. The flow data of the new node is adopted from the unsplitted cell.

II.3.3.3 THE SLUG

The acceleration of the slug for each time step is calculated by the conservation of momentum equation derived as discussed above. Using this acceleration, the velocity of the slug, that is, the velocity and the location of the upstream and downstream fronts of the slug are determined.

If the downstream front reaches to the elbow, then the dynamic pressure at the elbow is calculated using equation (2) as discussed in detail above.

II.3.4 SUMMARY OF THE NUMERICAL SIMULATION

The following computational procedure is adopted for the numerical simulation of the liquid slug motion:

1. Enter initial conditions for the domains.
2. Solve Euler equations for the sub-domain upstream the slug the selected solver subject to the appropriate boundary conditions.
3. Solve Euler equations for the tank section applying the boundary condition obtained at step 2.
4. Calculate the acceleration of the slug and update the velocity. Also update the velocity of the moving boundary to be used at step 2 during the next time level.

5. Calculate the position of the slug front. If slug reached to the elbow then calculate the impact pressure at the elbow.
6. Repeat steps 1 to 5 until the slug passes by.
7. Print the elbow pressure in time.

CHAPTER III

THE EXPERIMENTAL SET-UP

In order to simulate the liquid slug phenomenon, the set-up shown in Figure III.1 is constructed. The set-up consists of an air tank, a pipe with an elbow and a ball valve connecting them. The air tank is filled with pressurized air to simulate the pressurized steam at the power plant piping. The pipe system connected to the tank can be partially filled with water. A ball valve is connected to the system is used in order to simulate the rapid opening of a diaphragm. The impact pressure at the elbow is measured by a transducer located at the elbow. The detailed description of the components of the set-up is given below.

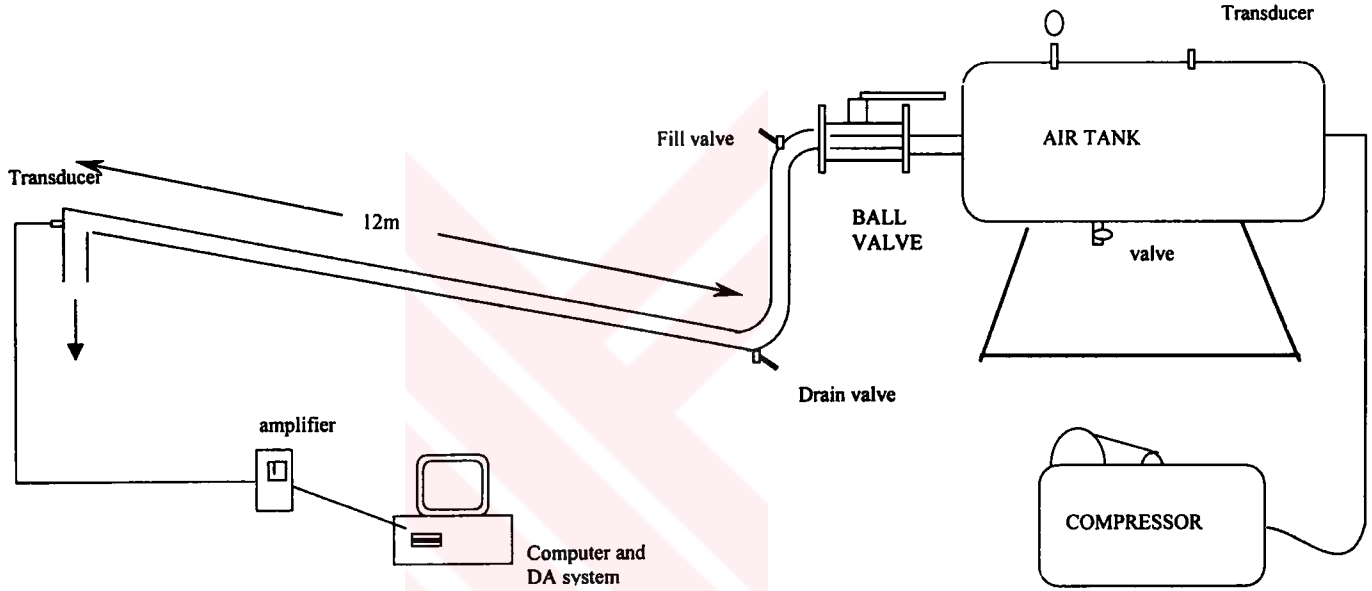

Figure III-1: The experimental setup

III.1 COMPRESSOR

A commercial compressor is used to supply pressurised air to the system. The compressor is run up to a pressure of 12bars, which exceeds the maximum pressure requirement of the experiments. It has a 250-lt air tank with the 250lt/min air supply capacity to the system and it has a 250-lt air tank.

III.2 AIR TANK

The compressor is connected to a pressure tank with a 14 bars capacity is of 500 liters. The tank has several outlets; a valve for small amount of air release is connected to one of the outlets, a glycerin manometer is installed into another one and yet there is also a pressure transducer hole. There is also a 10cm diameter outlet on the tank with the same diameter ball valve mounted on it. Of course, there is a connection to the compressor, too. Tank is located horizontally 1.0m above the ground level from bottom.

III.3 THE PIPE

The pipe drops down vertically 120cm just after it exits the tank. There are two ball valves on the upper and lower ends of the drop: one for filling and the other for drainage of water out of the system. Then the pipe extents 12m long with an upward slope of 8% with a 12m length. At the very end of the inclination there is a vertical downward elbow with a transducer hole on it.

III.4 CASING FOR TRANSDUCER

During the initial test runs, it was observed that the stain particles damaged the surface of the transducer. To overcome this problem, a casing details of which is shown in figure III-2 is manufactured and attached to the elbow.

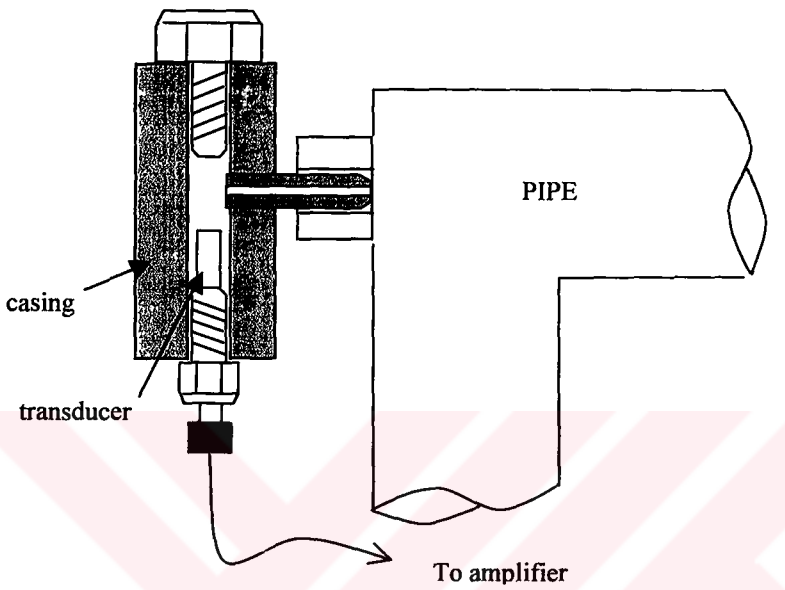

Figure III-2: Casing for the transducer

III.5 THE DATA ACQUISITION

The electronic system consists of an Intel 486 PC, a piezoelectric transducer mounted at the center of the elbow with the casing described above, an amplifier connected to the transducer and a data acquisition system which is connected to the computer. The description of the electronic system, is given at the appendix C.

III.6 TESTS

All of the theoretical, the numerical and the dimensional analyses revealed the tank pressure, the slug mass and the distance travelled by the slug as the most significant parameters affecting the peak pressure at the elbow. Because of the physical limitations, the travel length could only be monitored by the slug length. Furthermore, since the pipe geometry is fixed, change in slug mass is congruent with the change in the slug length.

That is, during the tests only slug mass and tank pressure is controlled. Dictated by the capacity of the compressor, tank pressures of 2, 3, 4 and 5 bars are selected. Also concerning the pipe properties, different slug masses, (simulating different slug lengths and thus travel distance) of 24, 32, 40 and 48 kg are selected. Tests are conducted for all combinations of these values. In other words there were sixteen runs.

III.6.1 REPEATABILITY OF THE EXPERIMENTS

The data, the variation of the pressure at the elbow with time, collected suffered some fluctuations. To be able to separate the oscillations associated with the electronical noise and the flexibility of the pipeline, a smoothing operation is carried out. Smoothing is performed through a forward smoothing, taking the mean of five consecutive data, such that,

$$P_i = \frac{1}{5} \sum_{n=i}^{n=i+4} P_n \qquad (1)$$

In order to investigate the repeatability of the experiments a series of five tests were run using the same tank pressure-slug mass values. As reported by Bozkuş, the valve opening speed may have significant effects on the output of the experiments. Furthermore, it is also suggested that this significance increases with increasing tank pressure. Therefore, the most severe case, P=5 bar and m=24 kg, is selected in order to examine the repeatability of the tests. In figure III.3, the smoothed output of these tests together with their mean normalized with respect to the maximum average output is shown.

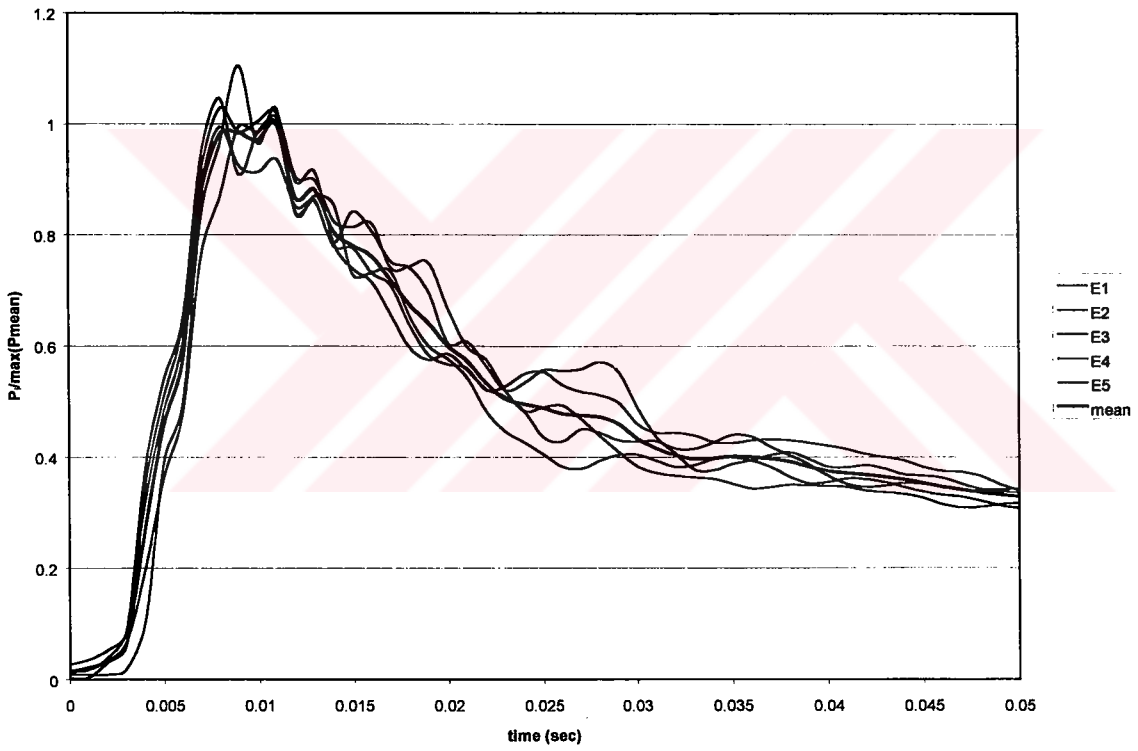

Figure III-3: Normalized readings for the experiments and the mean of them.

The agreement is reasonably good. To further this observation, the variation of the standard deviation of each output with time from the overall average is shown in figure III.4. The maximum value of the standard deviation is 0.114. However, it

must be noted that this value is at the rising limb of the data, where, the most abrupt change of pressure observed. Indeed around the peak pressure, the standard deviation drops below 8%. Based on this finding it is concluded that the acceptance of the repeatability of the tests is justified.

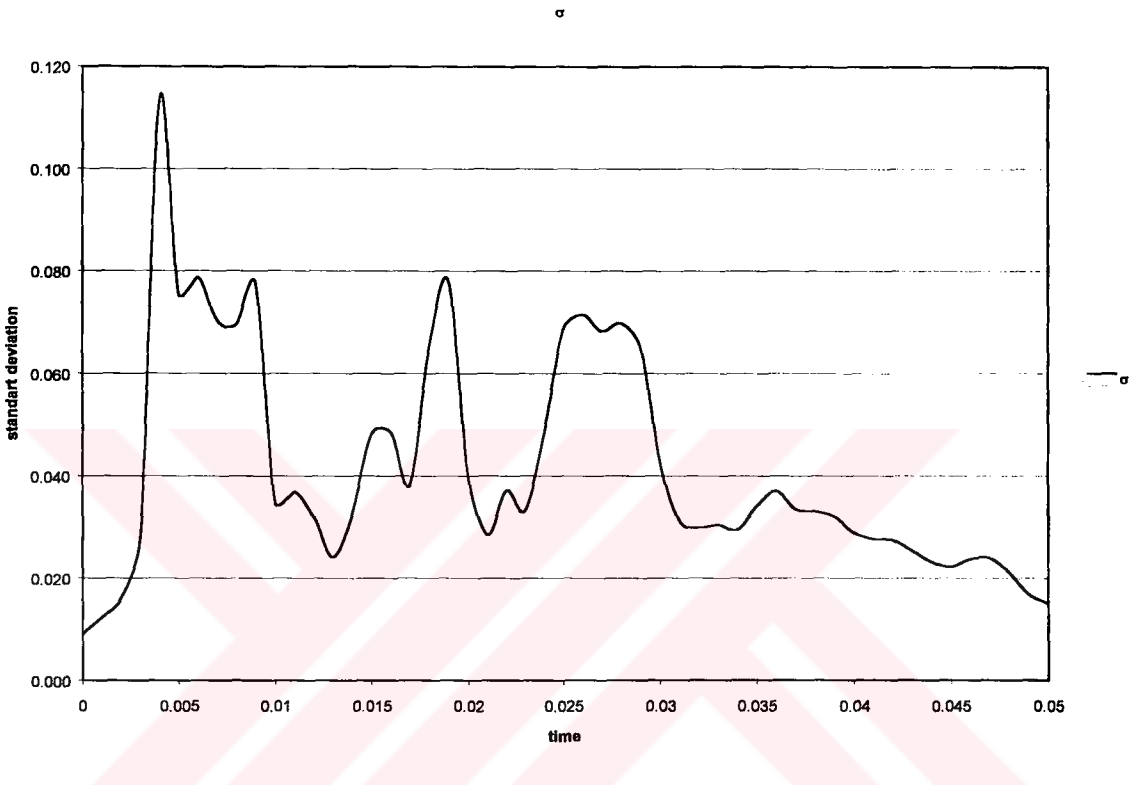

Figure III-4: The standard deviation of each data point

III.7 EXPERIMENTAL FINDINGS

The experimental results obtained are presented in figures III-5 through III-12. The figures III-5 through III-8 demonstrates the effect of change in tank pressure on the peak pressure for four different slug lengths (i. e. slug masses). Similarly, the effect of the slug length (i.e. slug mass) on the peak pressure for four different tank pressures is depicted in figure III-9 through III-12.

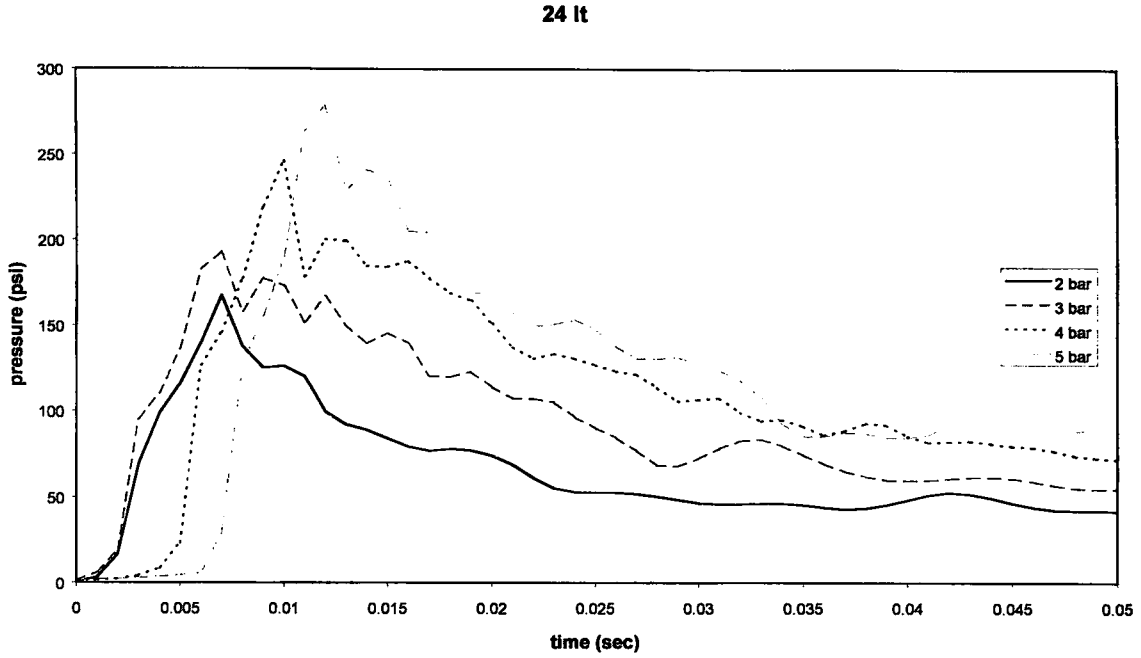

Figure III-5: Pressure history at the elbow for 24 liters

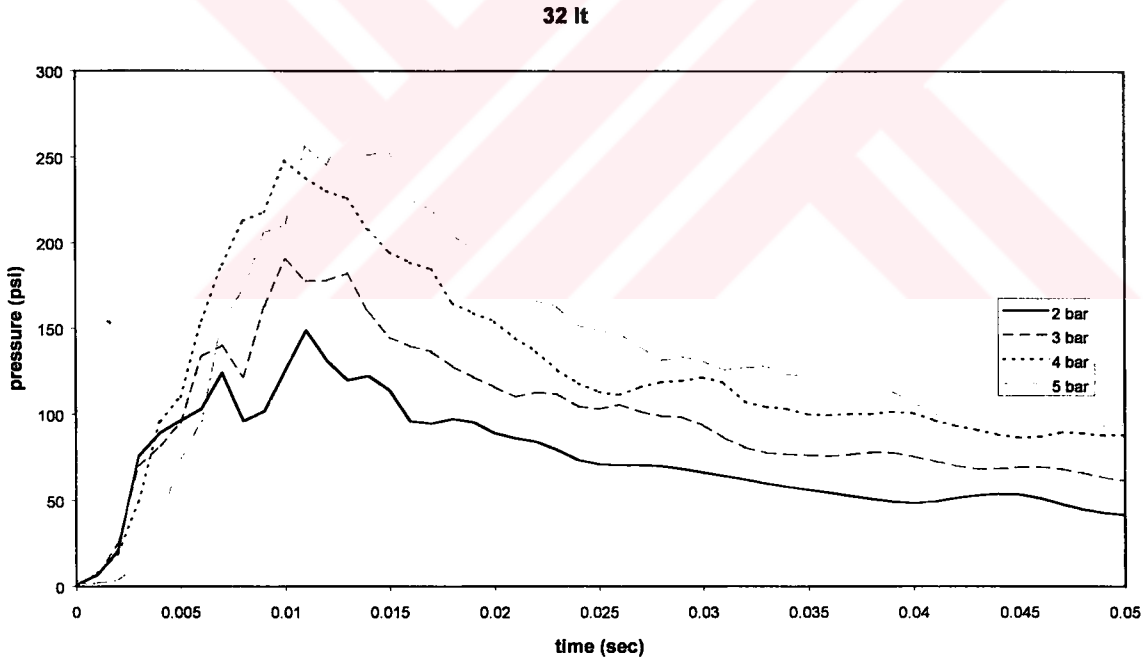

Figure III-6: Pressure history at the elbow for 32 liters

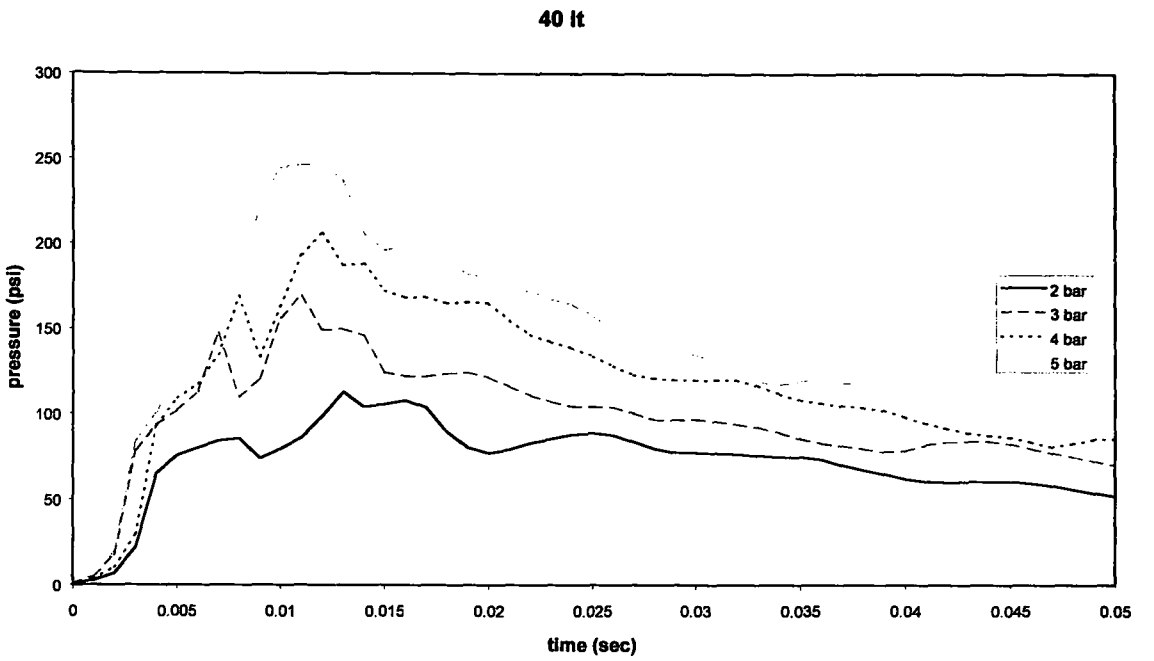

Figure III-7: Pressure history at the elbow for 40 liters

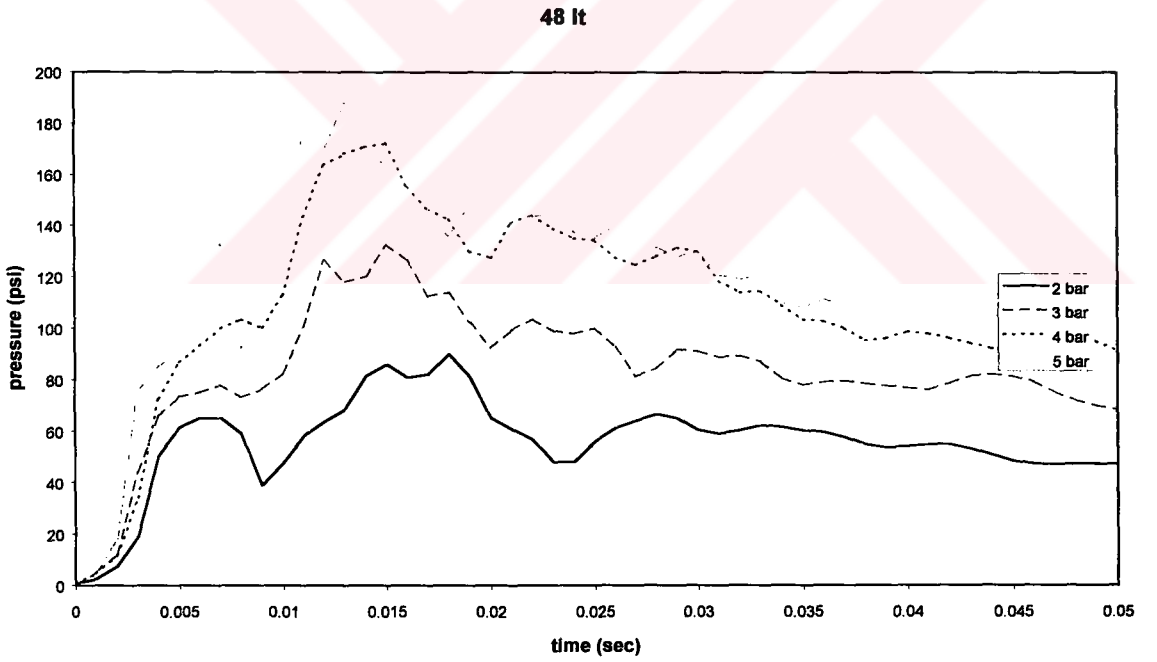

Figure III-8: Pressure history at the elbow for 48 liters

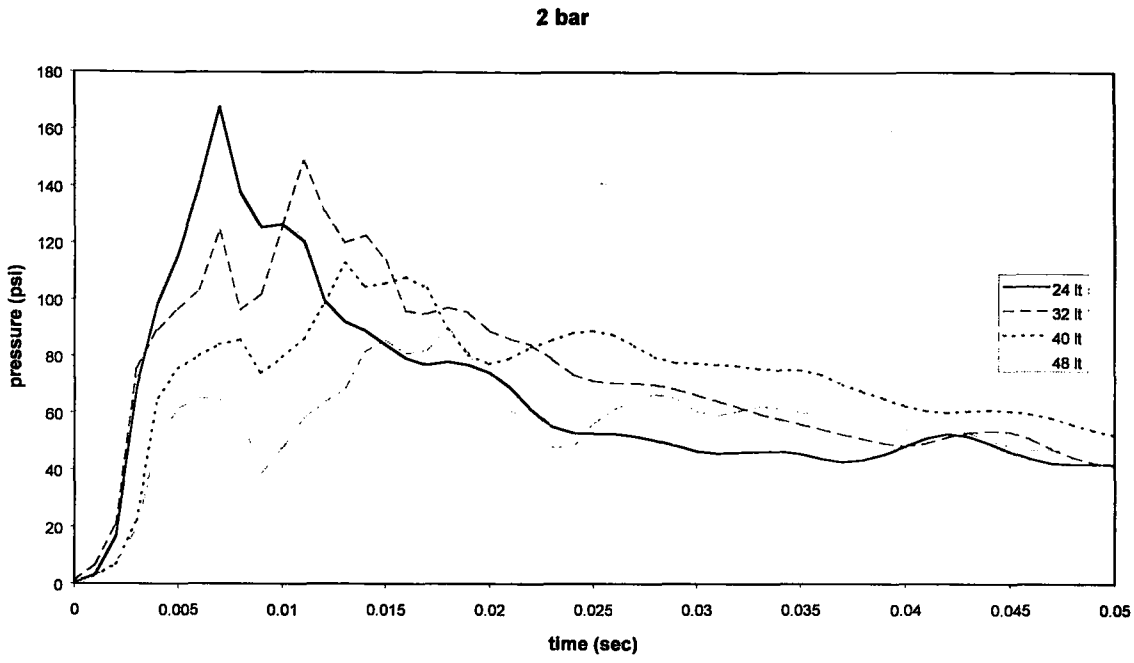

Figure III-9: Pressure history at the elbow for 2 bars

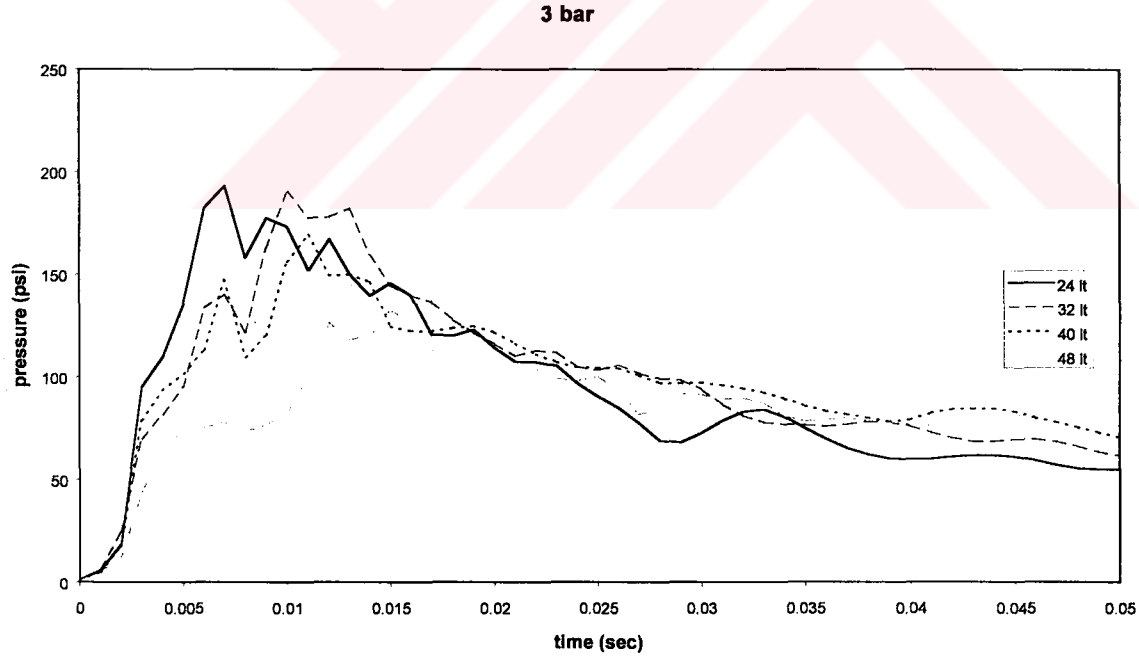

Figure III-10: Pressure history at the elbow for 3 bars

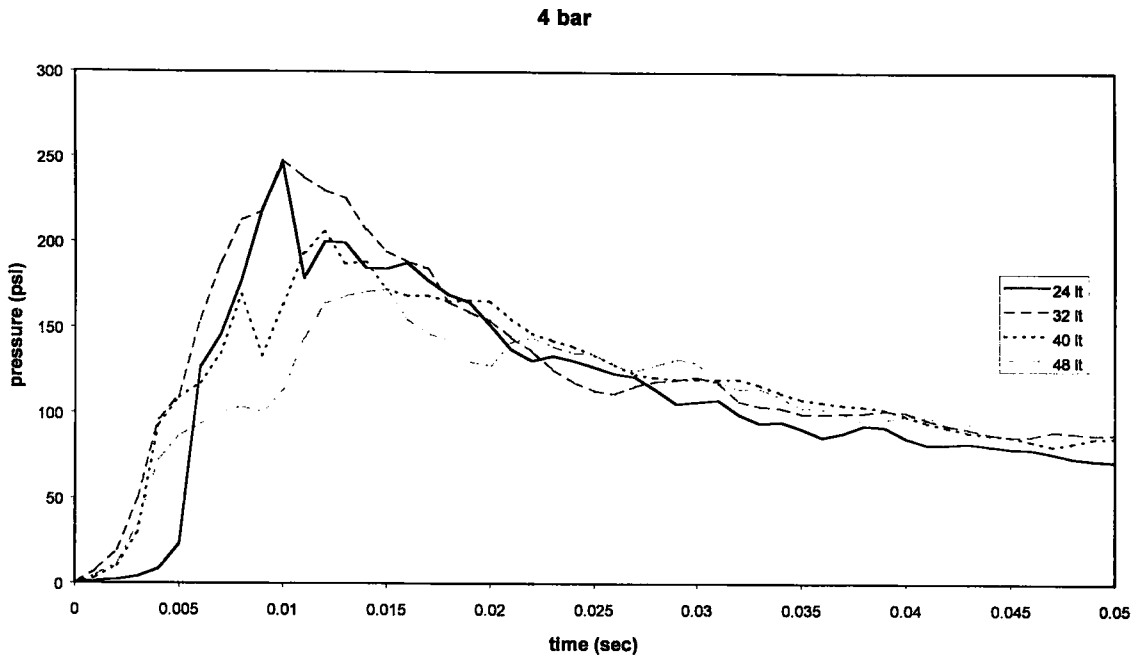

Figure III-11: Pressure history at the elbow for 4 bars

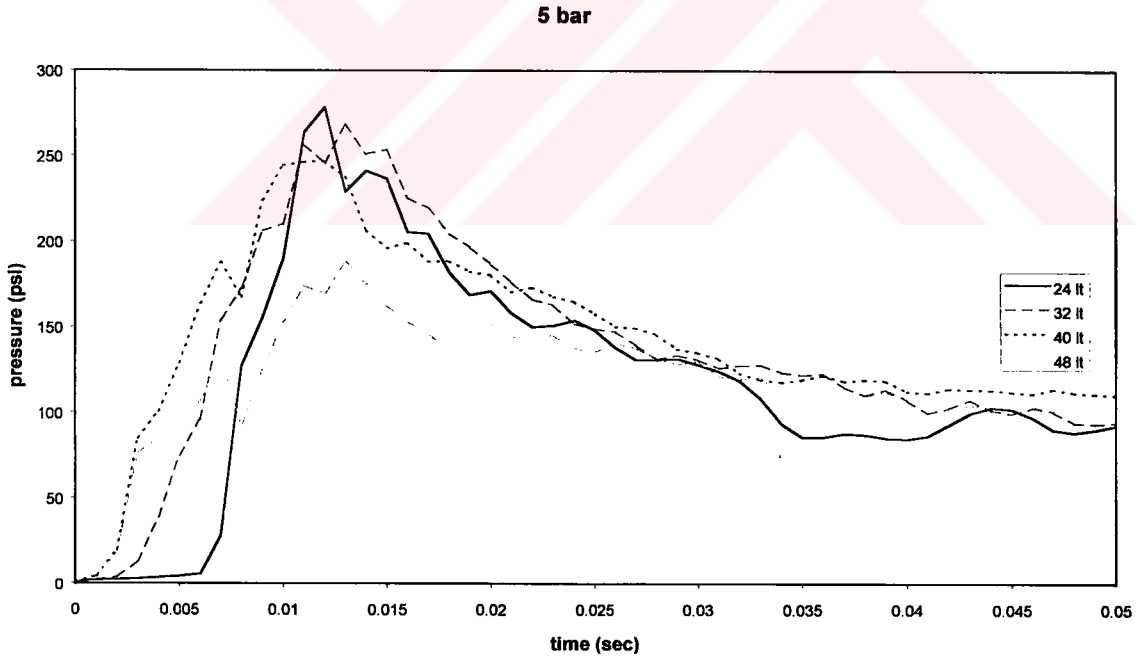

Figure III-12: Pressure history at the elbow for 5 bars

III.8 DIMENSIONAL ANALYSIS

After completing the theoretical investigation a dimensional analysis is carried out to discern important variables and obtain some dimensionless groups of these variables. This will enhance the utility of the findings of this study such that the findings may be scaled up from the laboratory-sized apparatus to actual power plant piping system. An analysis indicates that the pressure at the elbow, P_p, depends on the pipe properties, such as cross sectional area, A_p, of the pipe and pipe roughness, f, slug mass, m_s, upstream pressure, P_t, travel distance of the slug, L_t, length of the slug L_s and also several fluid properties, such as viscosity, ν, and compressibility, K. That is,

$$P_p = f_1(P_t, A_p, f, m_s, L_t, L_s, g, \rho, \nu, K) \qquad (2)$$

However, initial numerical studies as well as experimental evidence available suggests that effects of viscosity and compressibility are very small. Then the pressure at the elbow reduces to be dependent on the following parameters.

$$P_p = f_2(P_t, A_p, m_s, L_t, L_s, g) \qquad (3)$$

There are seven variables and three fundamental dimensions. Therefore there are four dimensionless groups. Selecting P_t, g and A_p as the repeating variables one obtains the following dimensionless numbers.

$$\Pi_1 = \frac{P_p}{P_t}, \qquad \Pi_2 = \frac{m_s g}{P_t A_p}$$
$$\Pi_3 = \frac{L_t}{L_S}, \qquad \Pi_4 = \frac{L_t^2}{A_P} \qquad (4)$$

Yet, for the experimental set-up, L_t+L_s is constant. That is to say, one of the parameters is redundant. After scrutinization of the data, Π_4 is dropped out. Thus,

$$\Pi_1 = f_3(\Pi_2, \Pi_3) \qquad (5)$$

III.9 PROCESSING THE DATA

In figure III-13, the data collected is plotted in terms of dimensionless numbers as Π_1 vs Π_2. As depicted in this figure the scatter in the data may be due to Π_3 not being taken into account. Once, as shown in figure III-14, Π_3 is also included in by combining Π_1 and Π_3 as $\Pi_1^{3/2}/\Pi_3$, while $\Pi_2^{1/2}$ is considered, the scatter is eliminated. The data then can be represented by:

$$\Pi_1^{3/2} = 0.1 * \Pi_3 \Pi_2^{1/2} \qquad (7)$$

or,

$$\left(\frac{P_p}{P_t}\right)^{3/2} = 10 * \left(\frac{m_s g}{P_t A_p}\right)^{1/2} \left(\frac{l_t}{l_s}\right) \qquad (8)$$

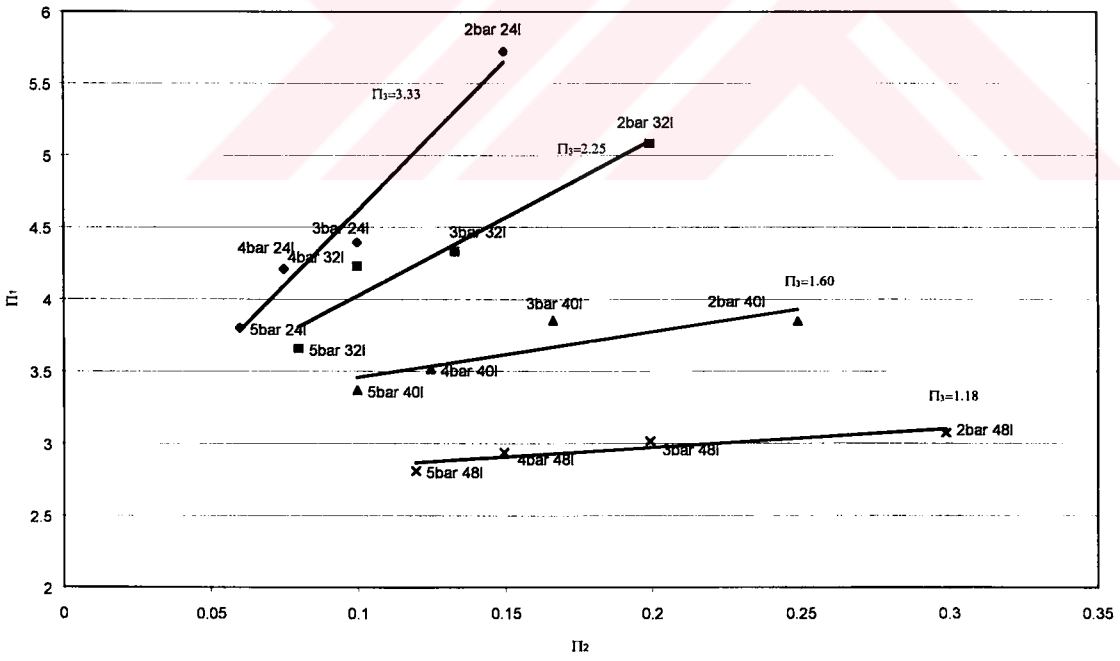

Figure III-13: $\prod_1$ vs $\prod_2$

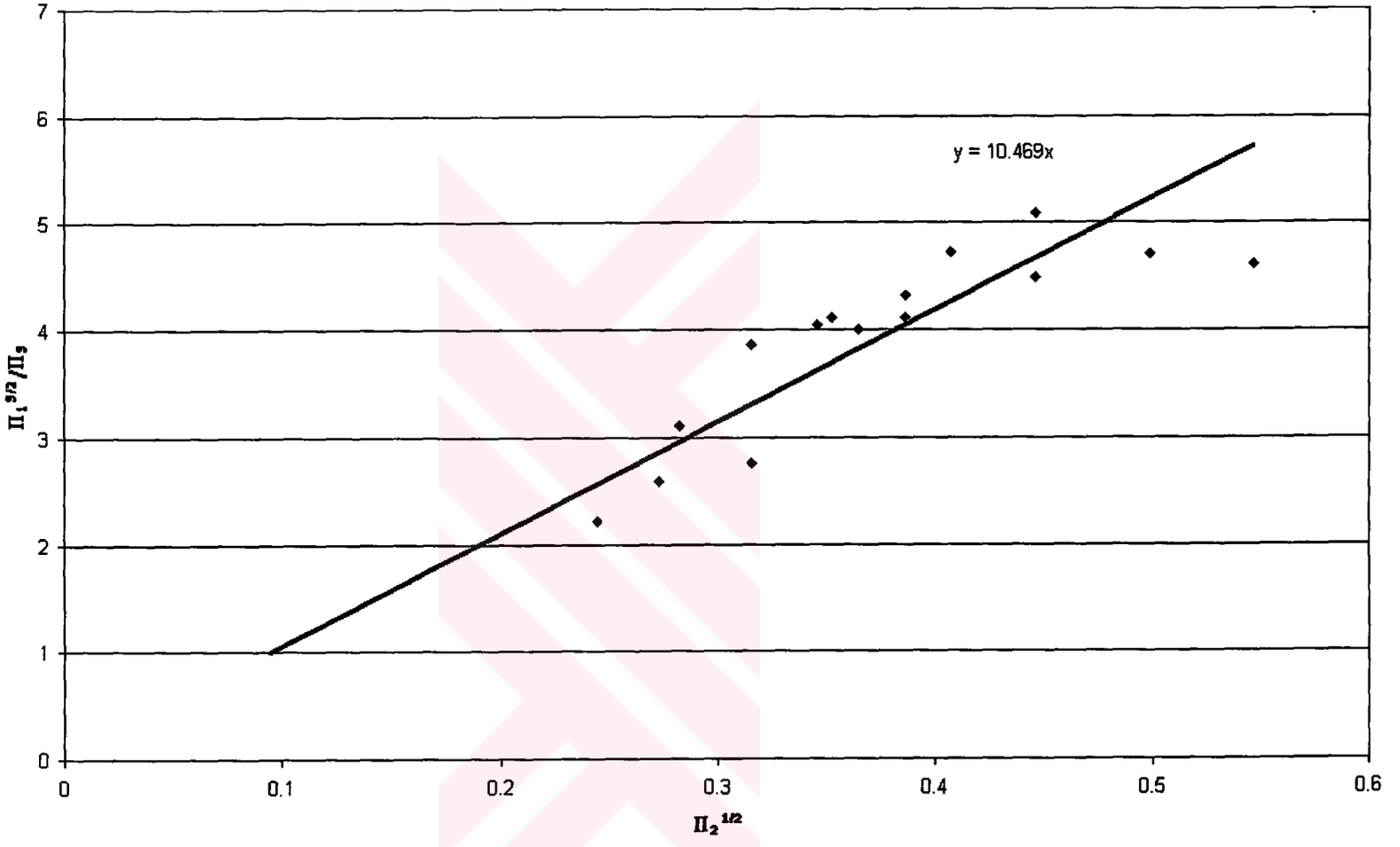

Figure III-14: Dimensionless relation found for liquid slug problem

III.10 COMPARISON OF DATA WITH FORMER STUDIES

The correlation between the non-dimensional parameters obtained through the dimensional analysis using the data of this study was tested against the available data in the literature.

There are two sets of data made available, namely by Bozkuş [1] and by Fenton [7]. The latter one is not compatible with this study and that of Bozkuş's. It does not provide information about the primitive variables forming the dimensionless parameters.

The comparison of the experimental results of this study and that of Bozkuş's is presented in figure III.15. the agreement is very good despite the fact that the diameters are of different sized and the pipe inclinations are different.

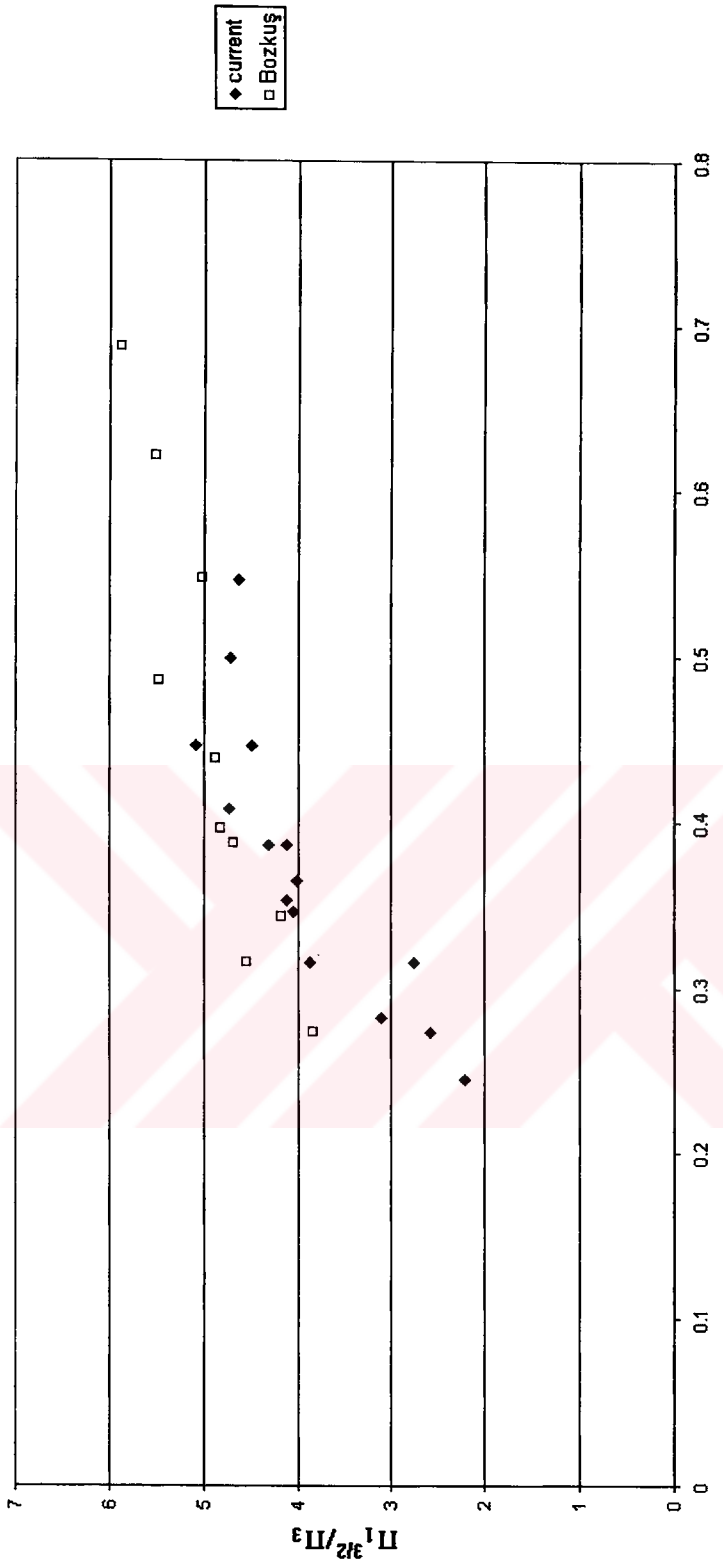

Figure III-15: Comparison the data of the Bozkuş [1] and the current study

III.11 COMPARISON OF THE NUMERICAL RESULTS WITH EXPERIMENTS

In order to verify the mathematical model developed and discussed in chapter II several runs were made using the input data compatible with that of experiments. In figures III-16 and III-17, the numerical output using the least accurate numerical scheme, Godunov solver with HLLC Riemann sampler is compared with the experimental findings. The numerical predictions are consistently lower than the experimental findings.

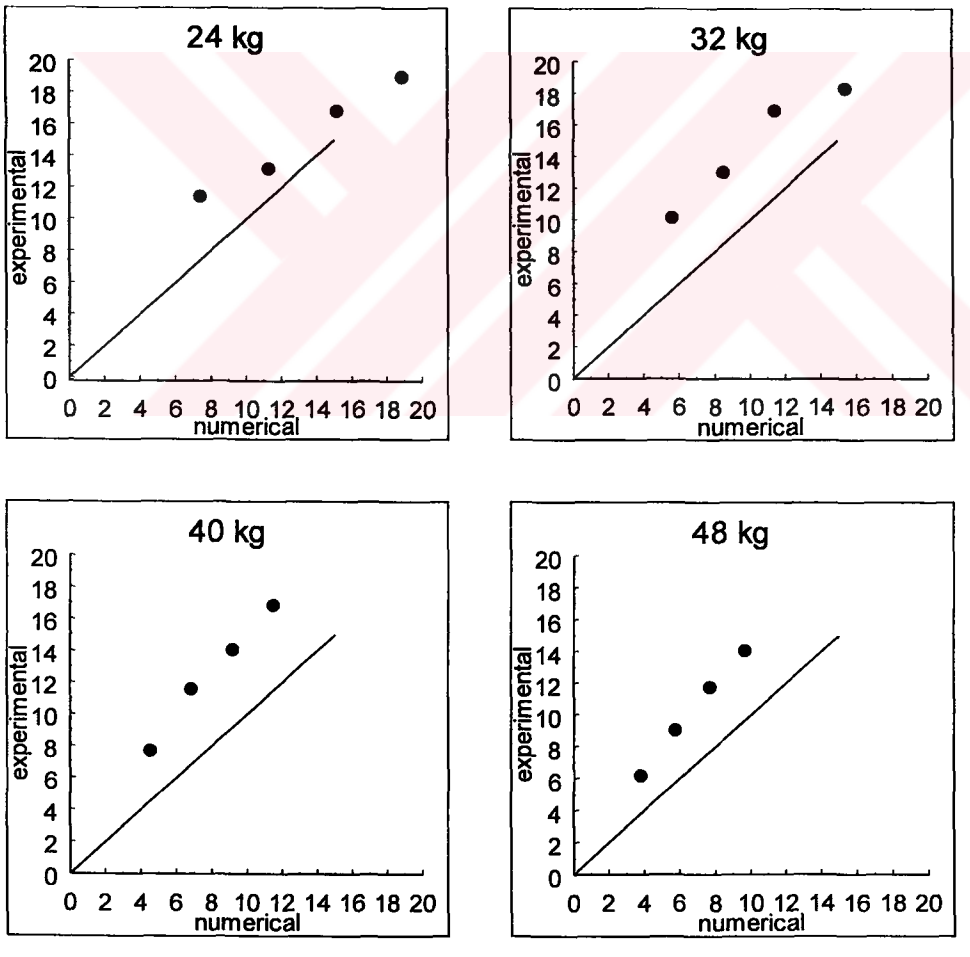

Figure III-16: Comparison of experimental data and the numerical output based on slug mass

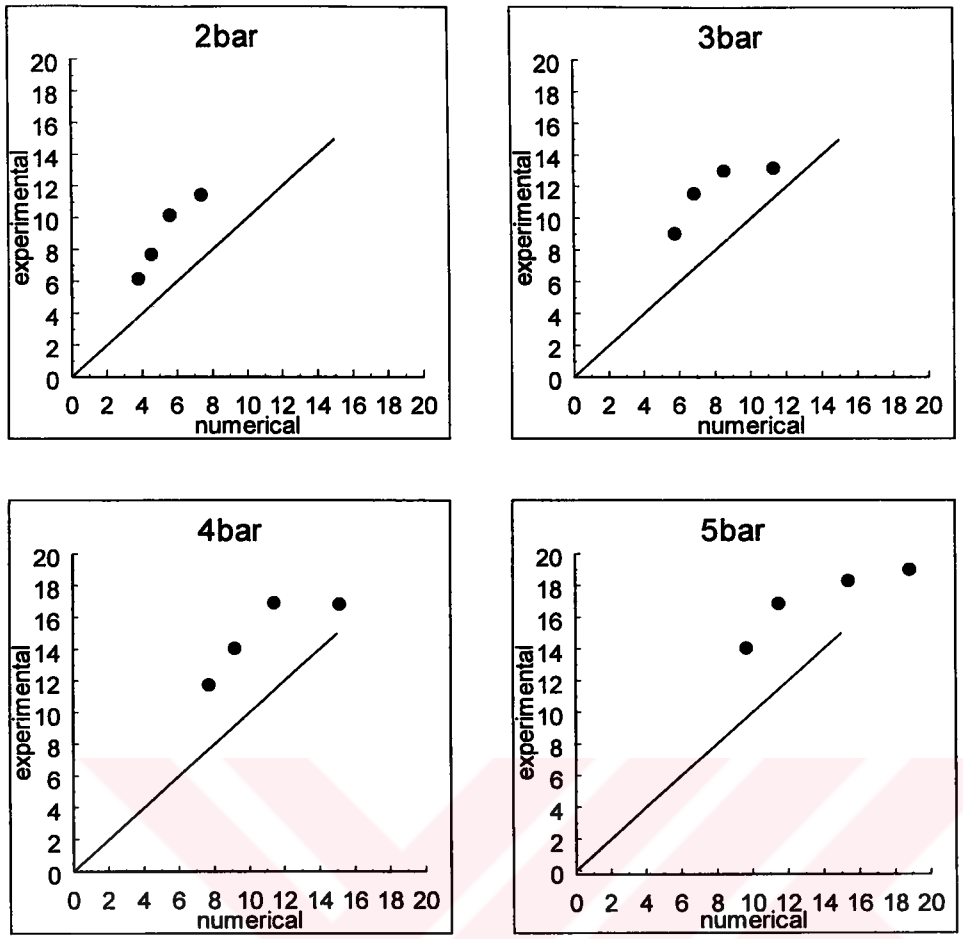

Figure III-17: Comparison of experimental data and the numerical output based on tank pressure

This situation does not change with the improved accuracy of the numerical scheme used. Indeed, as shown in Table III-1, the prediction of the several schemes previously described, are compared to substantiate this argument.

Table III-1 : Comparison of the numerical methods

Slug mass	Tank pressure	Experiment	Godunov	WAF	MUSCLsl	MUSCL
24	2	11.45	7.59	8.07	7.29	8.15
24	3	13.18	11.31	12.02	13.11	11.12
24	4	16.84	15.17	15.63	17.28	15.25
24	5	19.00	19.10	21.66	18.97	21.02
32	2	10.16	5.73	5.41	6.12	6.32
32	3	12.99	8.66	8.72	10.17	7.58
32	4	16.91	11.44	13.21	13.23	11.54
32	5	18.29	15.41	13.55	17.37	13.63
40	2	7.70	4.61	5.49	4.18	4.66
40	3	11.55	6.97	7.35	8.16	6.79
40	4	14.06	9.21	8.91	9.51	10.49
40	5	16.84	11.50	13.25	10.62	13.37
48	2	6.15	3.88	3.68	3.50	4.25
48	3	9.05	5.83	5.58	5.18	6.33
48	4	11.75	7.72	8.80	8.09	8.04
48	5	14.06	9.69	11.42	10.41	11.03

The examination of these figures further reveals that there is an almost constant difference between the predicted and observed values. This difference can be interpreted as the reflection of the deficiency of the model, which treats the elbow with an open end as a surge tank at the location just upstream the valve with a sudden closure. Yet, as previously spectaculated, the sudden closure assumption without a surge tank greatly overestimates the elbow pressure. In other words, elbow must be simulated such that an intermediate response between these extremes are employed.

CHAPTER IV

CONCLUSION AND THE FUTURE RECOMMENDATIONS

In this thesis, the behaviour of an individual liquid slug in a voided line is investigated. To achieve this goal, an experimental set-up is built. The experimental findings analysed through a dimensional analysis revealed that there exists a correlation between the peak pressure at the elbow and the other primitive variables described previously. The relationship is in the form:

$$\left(\frac{P_p}{P_t}\right)^{3/2} = 0.1 * \left(\frac{m_s g}{P_t A_p}\right)^{1/2} \left(\frac{l_t}{l_s}\right) \qquad (1)$$

and represents the data of both this study and that of Bozkuş's [1].

In brief, the data obtained complies with the need for data of the liquid slug motion in large diameter pipes. Indeed, the dimensionless relationship obtained may be used to extend the utility of the experimental findings.

The mathematical model underestimates the peak pressures. It is mainly due to the lack of proper representation of fluid-solid interaction between the slug and the elbow. It is therefore strongly recommended that this issue must be improved to the mathematical model.

Furthermore, the study of the numerical schemes employed may be envisaged the possibility of the utilisation of this schemes with 2-D and 3-D mathematical models that may override the limitations of the 1-D model used in this study.

REFERENCES

[1] BOZKUŞ, Z. 1991 The hydrodynamics of an individual transient slug in a voided line. Ph.D. dissertation. Department of Civil and Environmental Engineering. Michigan State University, East Lansing, MI, USA.

[2] BOZKUŞ, Z. & WIGGERT, D.C. 1991 Slug motion and impact in a voided line. In Fluid Transients and Fluid Structure Interaction (eds D.C. WIGGERT & F. J. MOODY). PVP-Vol. 224/FED-Vol.126, pp. 25-27. New York. ASME.

[3] BOZKUŞ, Z. & WIGGERT, D.C. 1992 Hydromechanics of slug motion in a voided line. In Unsteady Flow and Fluid Transients (eds R. BETTESS & J. WATTS), pp. 77-86. Rotterdam: AA. Balkema

[4] KIM, J. H. 1987 Water hammer prevention, mitigation and accommodations: a perspective. Transactions of the American Nuclear Society 55. 733-734.

[5] BOZKUŞ, Z. & WIGGERT, D.C. 1997 Liquid slug motion in a voided line. Journal of Fluids and Structures, 11, pp. 947-963.

[6] FENTON, R. M., 1989, the forces at a pipe bend due to the clearing of water trapped upstream, Ph.D. thesis. Department of Mechanical Engineering, Massachusetts Institute of Technology

[7] FENTON, R. M. & GRIFFITH, P. 1990 The force at a pipe bend due to the clearing of water trapped upstream. In Transient Thermal Hydraulics and Resulting Loads on Vessel and Piping Systems (eds F. J. MOODY, Y. W. SHIN & J. COLTON) PVP-Vol. 190, pp. 59-67. New York: ASME.

[8] MARTIN, C. S. & PADMANABHAN, M. 1979 Pressure pulse propagation in two-component slug flow. ASME Journal of Fluids Engineering, Vol. 101, pp. 44-52.

[9] MERILO, M., VAN DUYNE, D. A., SAFWAT, H. H. & ARASTU, A. H. 1990 Reducing the frequency of water hammer in nuclear power plants. In Transient Thermal Hydraulics and Resulting Loads on Vessel and Piping Systems (eds F. J. MOODY, Y. W. SHIN & J. COLTON) PVP-Vol. 190, pp. 1-7. New York: ASME.

[10] MOODY, F. J. 1990 Introduction to Unsteady Thermofluid Mechanics. New York: Wiley.

[11] PAPADAKIS, C. N. & HOLLINGSHEAD, D. 1985 Transients in empty pipes subject to rapid filling. In Proceedings Hydraulics Division Specialty Conference on Hydraulics and Hydrology In the Small Computer Age. pp. 1376-1381. New York: ASCE.

[12] VARDY, A. E. & HWANG, K. L. 1991 A characteristics model of transient friction in pipes. Journal of Hydraulic Research, Vol. 29, pp. 669-684.

[13] WHEELER, A. J. & SIEGEL, E. A. 1982 Measurements of forces in a safety valve discharge line. ASME Paper 82-WA/NE-8. New York: ASME.

[14] YANG, JIANDONG & WIGGERT, D. C. 1998 Analysis of fluid slug motion in voided line. ASME Journal of Pressure Vessel Technology 120 (in press)

[15] DAUGHERTY R. L., FRANZINI J. B., FINNEMORE E. J.,1985, Fluid mechanics with engineering applications, McGraw-Hill.

[16] CHAUDHRY M. H., 1979, Applied hydraulic transients, Van Nostrand Reinhold Company.

[17] WYLIE E. B., STREETER V. L., 1978, Fluid transients, McGraw-Hill.

[18] TORO, Eleuterio F., 1997, Riemann solvers and numerical methods for fluid dynamics: an introduction, Springer-Verlag, Heidelberg-Berlin

[19] TORO, Eleuterio F., The Weighted Average Flux method Applied to the time dependent Euler equations of gas dynamics, Phil. Trans. Roy. Soc. London, A341, pp. 449-530,1992

[20] TORO, Eleuterio F., Riemann Problems and the WAF method in solving two dimensional shallow water equations, Phil. Trans. Roy. Soc. London, A338, pp. 43-68,1992

[21] Godunov, S. K., A finite difference method for the computations of discontinuous solutions of the equations of fluid dynamics. Mat Sb. Vol 47, pp. 357-393, 1959

[22] Versteeg, H. K., Malalasekera, W. (1995) *An Introduction to Computational Fluid Dynamics The Finite Volume Method*, Longman Press, 1995, England

[23] Hirsch, C. (1990). *Numerical Computation of Internal and External Flows*, Vol 1&2, John Wiley & Sons, Chichester, England

[24] Shyy, W. (1996) *Computational Fluid Dynamics With Moving Boundaries*, Taylor & Francis, London

[25] Hirt, C. W., Amdsen, A. A. and Cook J. L. (1974). An Arbitrary Lagrangian-Eulerian Computing Method for All Flow Speeds, *J. Comp. Phys,* vol 14. pp. 227-253

[26] Hirt, C. W. and Nichols, B. D. (1981), Volume of Fluid (VOF) Method for the Dynamics of Free Boundaries, *J. Comp. Phys,* vol 39. pp. 201-225

[27] Ashgriz, N. and Poo, J. Y. (1990), FLAIR Flux Line-Segment Model for Advection and Interface Reconstruction, *J. Comp. Phys,* vol 93. pp. 449-468

[28] Sabanca, M. (1997), *Development of a 2-D Euler solver Using Finite Volume Method for External Flows*, Master Thesis, Middle East Technical University, Ankara, Turkey

[29] Roe, P. L., 1981, Approximate Riemann solvers, parameter vectors, and Difference Schemes. J. Comp Phys, vol43 pp: 357-372

[30] Osher, S. and Chakravarthy, 1984, S., Upwind Schemes and boundary conditions with applications to Euler equations in general geometries. SIAM J Numer. Anal., valo:21(5) pp: 955-984

[31] Harten, A., Lax, P. D. and van Leer, B., On upstream differencing and godunov type schemes for hyperbolic conservation laws. SIAM review, vol: 25(1), pp:35-61

[32] Colella P. and Woodward P. R., the piecewise parabolic method (PPM) for gas dynamics simulation. J. Comp. Phys. Vol: 54, pp: 174-201

APPENDIX A

NUMERICAL METHODS

A.2 INTRODUCTION

In this section, a general review of Euler solvers in terms of the solution strategies for compressible gas flow is introduced. An in depth examination of the schemes was carried out on the sudden breakdown of a diaphragm in a long 1-D tube, shock tube, separating two initial gas states at different pressures and densities.

A.3 THE BASIC GODUNOV SCHEME

The original Godunov method includes following three steps:

Step 1: Domain is discretised in a finite volume type representation and a piecewise constant approximation of the solution at $t = n\Delta t$ is defined as it is depicted

in the figure A-1. The finite volume type representation has been adopted in the previous studies. Based on these, integrating the conservation equation $\partial U / \partial t + \partial F / \partial x = 0$ over x in the domain (a,b) gives

$$\frac{d}{dt}\int_a^b U(x,t)dx = F(a,t) - F(b,t) \qquad (1)$$

and if it is then integrated in time:

$$\int_a^b U^{n+1}(x)dx - \int_a^b U^n(x)dx = -\Delta t\left[\overline{F}(U(b)) - \overline{F}(U(a))\right] \qquad (2)$$

where $\overline{F}$ is the time average of the physical flux between time level n and n+1. Defining the average state variable over the cell (i - ½ i + ½) as

$$\overline{U}_i = \frac{1}{\Delta x}\int_{i-1/2}^{i+1/2} U(x,t)dx \qquad (3)$$

the integral conservation equation becomes:

$$\overline{U}_i^{n+1} - \overline{U}_i^n = -\frac{\Delta t}{\Delta x}\left[\overline{F}(U_{i+1/2}) - \overline{F}(U_{i-1/2})\right] \qquad (4)$$

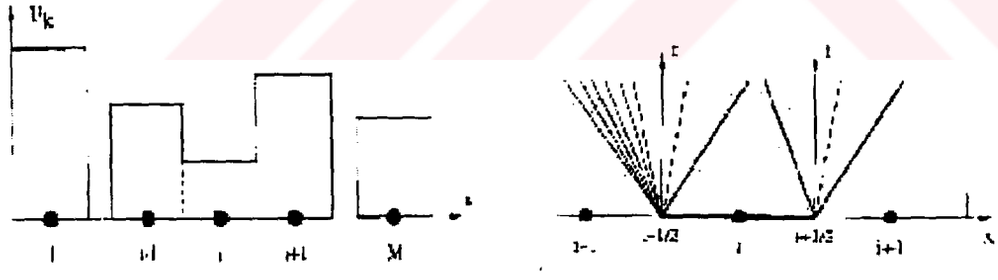

Figure A-1: Data reconstruction for Godunov's method

Step 2:

At this step the solution for the local Riemann problem at the cell interfaces is obtained. This step is the sole step of the whole procedure, which reflects the physics. The conservation equations are solved for each cell exactly. This is done by solving

the Riemann problem for each cell as it is described in the appendix B. In brief, $\overline{F}_{i+1/2}$ and $\overline{F}_{i-1/2}$ of equation 4 are calculated at this step.

Since a time discretisation is used, a certain CFL number and condition must be indicated. This selection is done by:

$$\Delta t = CFL \frac{\frac{1}{2}\Delta x}{S^n_{max}} \tag{5}$$

where CFL < 1.0

where S^n_{max} indicates the maximum wave speed within cells. This condition indicates that the information propagated by waves remain within cells such that they do not affect other cell. CFL<0.9 range is proposed by Toro [19] as a good selection of.

A.3.1 TEST CASES

Some cases to test the scheme suggested by Toro[19] were adopted. The initial conditions and results are given in Table 1 and Figures A-2 to A-5 respectively.

Table A-1: Test cases to be used for the numerical methods

Test	ρ_L	p_L	u_L	ρ_R	p_R	u_R
1	1.0	0.75	1.0	0.125	0.0	0.1
2	1.0	-2.0	0.4	1.0	2.0	0.4
3	1.0	0.0	1000.0	1.0	0.0	0.01
4	5.99924	19.5975	460.894	5.99242	-6.19633	46.0950

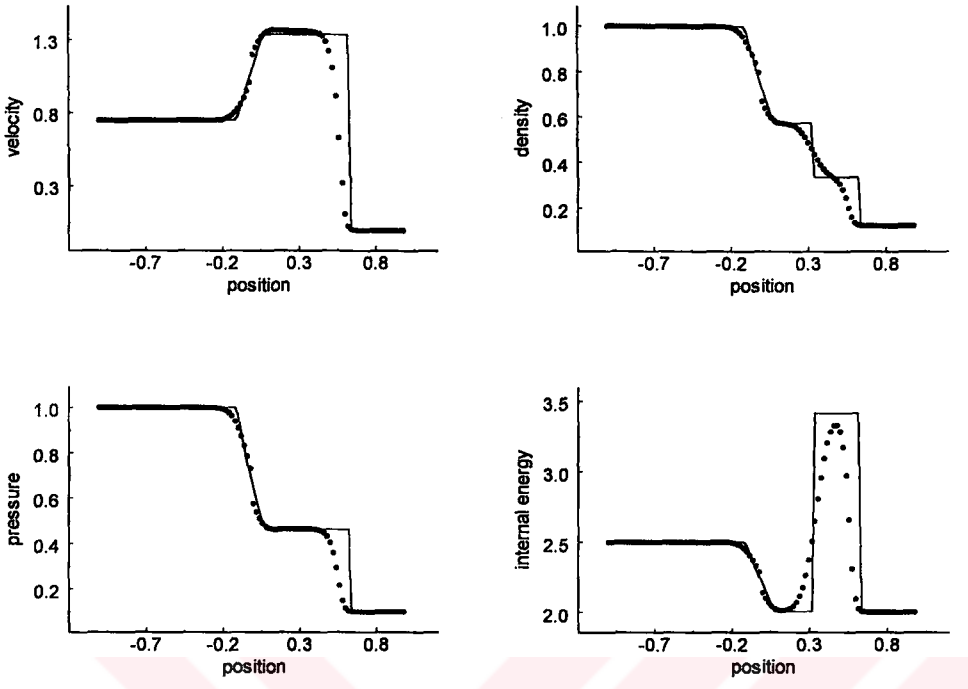

Figure A-2: Test I on Godunov Solver using exact Riemann solver

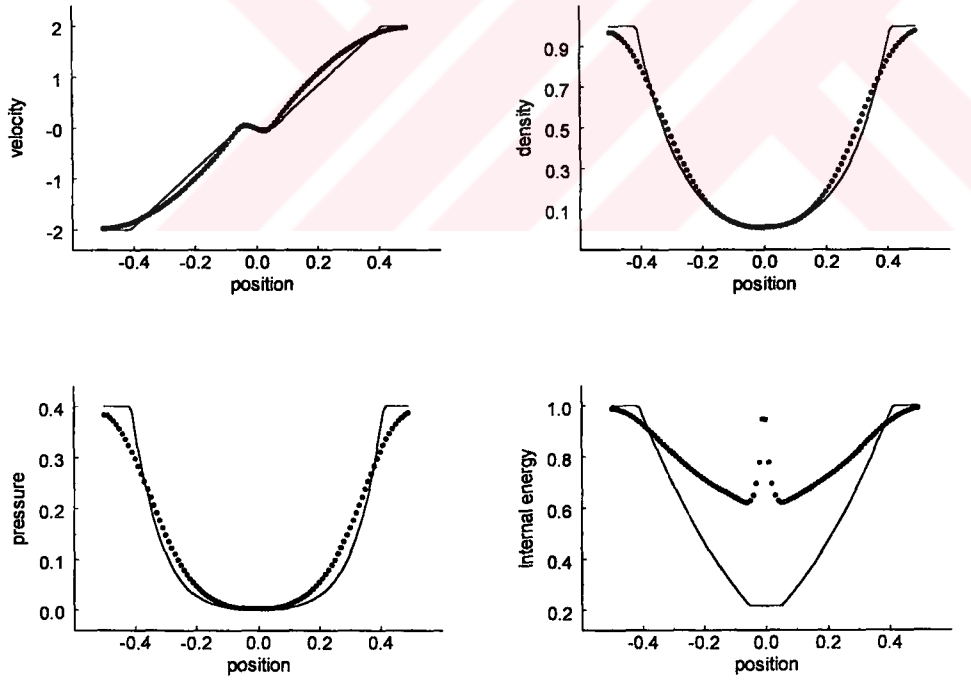

Figure A-3: Test 2 on Godunov Solver using exact Riemann solver

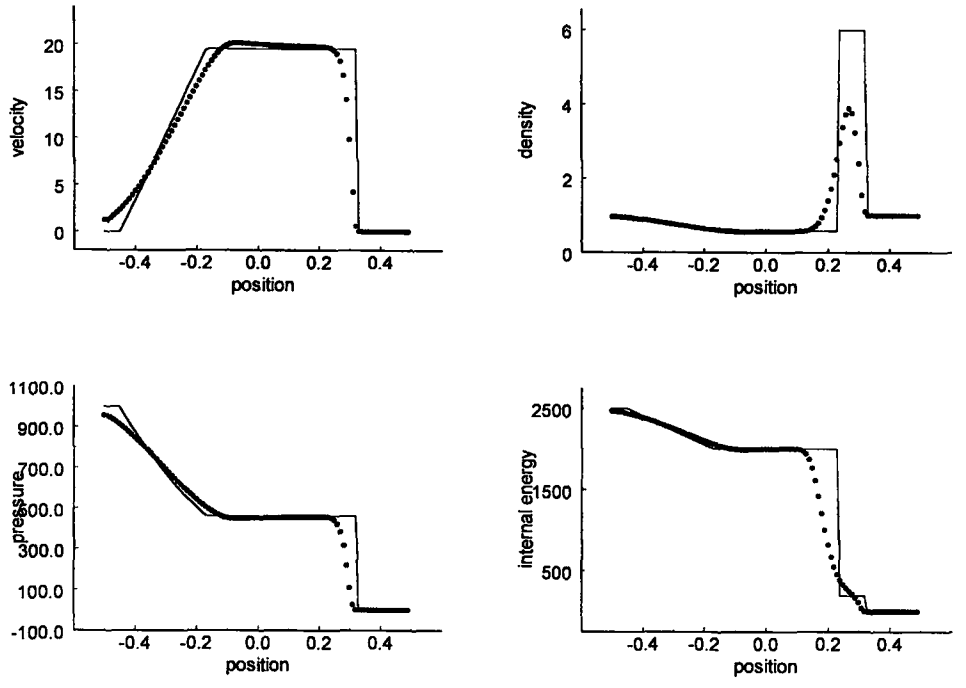

Figure A-4: Test 3 on Godunov Solver using exact Riemann solver

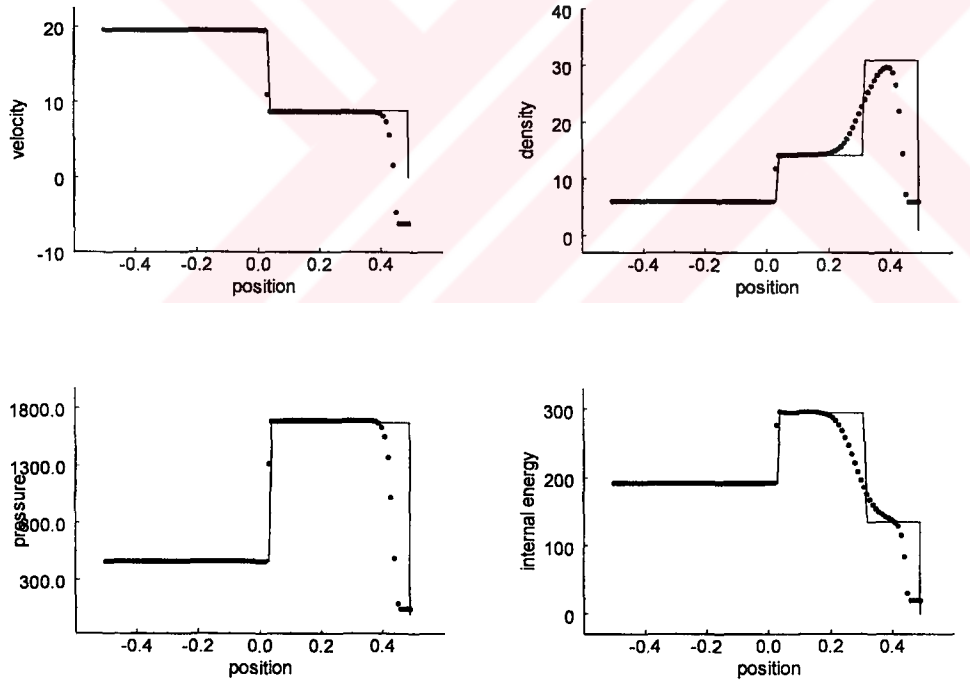

Figure A-5: Test 4 on Godunov Solver using exact Riemann solver

A.3.2 BOUNDARY CONDITIONS

The boundary conditions (BC) are the most important part of the solution. The type of boundary conditions used in this section are primitive boundary conditions; three kinds of primitive boundary conditions are considered:

Transmissive BC's: This kind of BC is used when the ends of the shock tube are free. Then, BC's are given as:

$$p_0 = p_1, \qquad u_0 = u_1, \qquad \rho_0 = \rho_1, \tag{6.a}$$

$$p_{M+1} = p_M, \qquad u_{M+1} = u_M, \qquad \rho_{M+1} = \rho_M, \tag{6.b}$$

Reflective BC's: This kind of BC is used when the ends of shock the tube are solid walls. BC's are given as:

$$p_0 = p_1, \qquad u_0 = -u_1, \qquad \rho_0 = \rho_1, \tag{7.a}$$

$$p_{M+1} = p_M, \qquad u_{M+1} = -u_M, \qquad \rho_{M+1} = \rho_M, \tag{7.b}$$

Moving Boundary: This kind of BC is used when the ends of the shock tube are moving solid walls. Then, BC's are given as:

$$p_0 = p_1, \qquad u_0 = -u_1+2u_w, \qquad \rho_0 = \rho_1, \tag{8.a}$$

$$p_{M+1} = p_M, \qquad u_{M+1} = -u_M+2u_w, \qquad \rho_{M+1} = \rho_M, \tag{8.b}$$

where M is the number of nodes, and u_w is the speed of the moving wall.

A.4 APPROXIMATE-STATE RIEMANN SOLVERS

In the previous section, the application of the Godunov method with an exact Riemann solver is presented in detail. However, solution of exact Riemann problem, consists of Newton- Raphson iteration in the calculation of p_*, leads to an extra computational cost. For large-scale problems, an enormous time is needed for the solution of the fluxes due to this iterative process. Also, for higher order methods, disscussed below, faster schemes are required.

To overcome this problem, some attempts to develop approximate-state Riemann solvers are made by Harten, Lax and van Leer [32], Roe [30], Osher [31], and many others. In the next section, a modified version of the former method will be discussed.

A.4.1 HLLC RIEMANN SOLVER

One of the most efficient and easy- to-use approximations evaluating the numerical flux is the Harten, Lax and van Leer's approach, abbraviated as HLL solver. A modified version of this approach is called as HLLC solver, where C stands for the contact discontinuity wave. The most important advantages of HLLC solver are as follows: First of all, it is non-iterative so that very cheap and secondly it gives fluxes directly that is without evaluating primitive values. This leads an extra speed-up. The details of the method is given below.

A.4.1.1 RIEMANN PROBLEM AND GODUNOV FLUX

In the previous section, discretisation of equation (1) is given as:

$$U_i^{n+1} = U_i^n + \frac{\Delta t}{\Delta x}\left(F_{i-1/2}^n - F_{i+1/2}^n\right) \tag{9}$$

with Godunov intercell numerical flux is given as:

$$F_{i-1/2} = F(U_{i-1/2}(0)) \tag{10}$$

in which $U_{i+1/2}$ is the exact solution of the Riemann problem evaluated at x/t=0 and such that:

$$U(x,0) = \left.\begin{cases} U_t + F(U)_x \\ \begin{cases} U_L & if \quad x < 0 \\ U_R & if \quad x > 0 \end{cases} \end{cases}\right\} \tag{11}$$

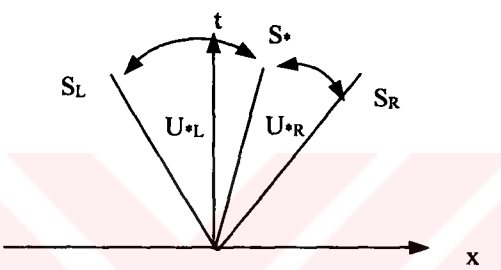

Figure A-6: Structure for HLLC solver; the middle wave separates two constant states in star region.

There are three waves associated with the problem, as mentioned previously. These are S_L and S_R which are the left and fastest signal velocities perturbing the data, and S_*, which is the speed of the contact discontinuity. Consider the wave structure of the complete Riemann problem with average star region states U_{*L}, and U_{*R} as shown in the figure A-6. The integral form of equation (1) for this structure is:

$$\int_{x_L}^{x_R} U(x,t)dx = \int_{x_L}^{x_R} U(x,0)dx + \int_0^T F(U(x_L,t))dt - \int_0^T F(U(x_R,t))dt \tag{12}$$

Evaluation of the right hand-side gives

$$\int_{x_L}^{x_R} U(x,t)dx = x_R U_R - x_L U_L + T(F_L - F_R) \tag{13}$$

Then, re-writing the left hand side

$$\int_{x_L}^{x_R} U(x,t)dx = \int_{x_L}^{TS_L} U(x,t)dx + \int_{TS_L}^{TS_R} U(x,t)dx + \int_{TS_R}^{x_R} U(x,t)dx \tag{14}$$

$$\int_{x_L}^{x_R} U(x,t)dx = \int_{TS_L}^{TS_R} U(x,t)dx + (TS_L - x_L)U_L + (x_R - TS_R)U_R \tag{15}$$

and substitution of (9) into (7) gives:

$$\frac{1}{T(S_R - S_L)} \int_{TS_L}^{TS_R} U(x,t)dx = \frac{S_R U_R - S_L U_L + F_L - F_R}{S_R - S_L} \tag{16}$$

revealing that, the integral average of the exact Riemann problem between fastest and the slowest signals, in other words in the star region, is constant.

Now, re-writing the left hand side of equation (16) considering the wave S*, sepearting the *L and *R regions:

$$\frac{1}{T(S_R - S_L)} \int_{TS_L}^{TS_R} U(x,t)dx = \frac{1}{T(S_R - S_L)} \int_{TS_L}^{TS_*} U(x,t)dx + \frac{1}{T(S_R - S_L)} \int_{TS_*}^{TS_R} U(x,t)dx \tag{17}$$

Let's define

$$\left.\begin{aligned} U_{*L} &= \frac{1}{T(S_R - S_L)} \int_{TS_L}^{TS_*} U(x,t)dx \\ U_{*R} &= \frac{1}{T(S_R - S_L)} \int_{TS_*}^{TS_R} U(x,t)dx \end{aligned}\right\} \tag{18}$$

It can be easily proven that both U_{*L} and U_{*R} also give constant integral averages.

Then, since these averages are constants, one can use these average values for the star regions, such that:

$$U(x,t)=\begin{cases} U_L & if \quad x/t < S_L \\ U_{*L} & if \quad S_L < x/t < S_* \\ U_{*R} & if \quad S_* < x/t < S_R \\ U_R & if \quad x/t > S_R \end{cases} \tag{19}$$

Applying the Rankine-Hugoniot conditions across each of the waves one obtains:

$$\begin{aligned} F_{*L} &= F_L + S_L(U_{*L} - U_L) \\ F_{*R} &= F_R + S_R(U_{*R} - U_R) \end{aligned} \tag{20}$$

The details of the solution of U_{*L} is ommited here, since it is given elsewhere (Harten, Lax and van Leer's original paper [32]). The solution for the U_{*L} and U_{*R} are as follows:

$$U_{*K} = \rho\left(\frac{S_* - U_K}{S_K - S_*}\right)\begin{bmatrix} 1 \\ S_* \\ \dfrac{E_K}{\rho_k} + (S_* - U_K)\left[S_* + \dfrac{p_K}{\rho_K(S_* - U_K)}\right] \end{bmatrix} \tag{21}$$

for K=L and K=R.

Then, HLLC flux for the Godunov method becomes:

$$F_{i+1/2}^{hllc} = \begin{cases} F_L & if \quad 0 < S_L \\ F_{*L} = F_L + S_L(U_{*L} - U_L) & if \quad S_L \le 0 \le S_* \\ F_{*R} = F_R + S_R(U_{*R} - U_R) & if \quad S_* \le 0 \le S_R \\ F_R & if \quad 0 > S_R \end{cases} \tag{22}$$

As seen from equation (22), the wave speeds must be known to determine the HLLC flux.

A.4.1.2 WAVE SPEED ESTIMATES:

There are quite a few approaches to estimate wave speeds S_L, S_*, S_R. The simplest, one which sometimes do not give satisfactory results ise S=u±a. Toro [19] proposed a time consuming yet an effective method: Suppose that the characteristics of the contact discontinuity, p_*, u_* is known. Then, the wave the speeds are given as:

$$S_L=u_L- a_L\, q_L, \qquad S_*= u_*, \qquad S_R=u_R- a_R\, q_R \tag{23}$$

Where, the correction for the rarefaction wave q_k (K=L&R) is defined as.

$$q_K = \begin{cases} 1 & \text{if} \quad p_* \le p_K \\ \left[1+\dfrac{\gamma+1}{2\gamma}(p_*/p_k-1)\right]^{1/2} & \text{if} \quad p_* \le p_K \end{cases} \tag{24}$$

A refinement is also proposed for the middle wave speed S_*, such that:

$$S_* = \frac{p_R - p_L + \rho_L u_L(S_L-u_L) - \rho_L u_R(S_R-u_R)}{\rho_L(S_L-u_L) - \rho_L(S_R-u_R)} \tag{25}$$

In the solver, it is observed that the second approximation for the middle wave gives more reliable results.

A.4.1.3 Estimation for p_* and u_*

In this formulation, the only remaining unknowns left are u_* and p_*. There are some approximate solutions based on the assumption that both states are known are available to determine their values. Three such solvers introduced below, all of which are derived from the exact solution.

A.4.1.3.1 Primitive Value Riemann Solver (PVRS)

One of the most simple estimates for the middle wave characteristics of the shock tube problem which is the set of algebraic equation of primitive values given below:

$$p_{*_{pv}} = \frac{1}{2}(p_L + p_R) - \frac{1}{2}(u_R - u_L)\overline{\rho a}$$
$$u_{*_{pv}} = \frac{1}{2}(u_R + u_L) - \frac{1}{2}\frac{(p_R - p_L)}{\overline{\rho a}} \tag{26}$$

where,

$$\overline{\rho} = \frac{1}{2}(\rho_L + \rho_R),$$
$$\overline{a} = \frac{1}{2}(a_L + a_R) \tag{27}$$

Further below, for the construction of TVD limiters densities at the left and the right states will be needed. These values are given as:

$$\rho_{*K} = \rho_K + (p_* - p_K)/a_L^2 \tag{28}$$

K is either L or R for left and right values respectively, of density at the star region.

A.4.1.3.2 Two-Rarefaction Riemann Solver (TRRS)

Assuming both the left and right going waves are rarefaction waves, following solutions are given for the p* and u*:

$$p_{*TR} = \left[\frac{a_L + a_R - \frac{(\gamma-1)}{2}(u_R - u_L)}{a_L / p_L^z - a_R / p_R^z}\right]^{1/2}$$
$$u_{*TR} = \left[\frac{p_{LR}u_L / a_L + u_R / a_R \frac{2(p_{LR}-1)}{(\gamma-1)}}{p_{LR} / a_L + 1/a_R}\right]^{1/2} \tag{29}$$

where,

$$p_{LR} = (p_L / p_R)^z, \qquad z = \frac{\gamma - 1}{2\gamma} \tag{30}$$

The right and left densities at the star region are given as:

$$\rho_{*K} = \rho_K \left(\frac{p_*}{p_K} \right)^{1/\gamma} \tag{31}$$

A.4.1.3.3 Two-Shock Riemann Solver (TSRS)

Assuming both the left and right going waves are shock waves, following solutions are given for the p_* and u_*:

$$\begin{aligned} p_{*TS} &= \frac{g_L(p_0)p_L + g_R(p_0)p_R - (u_R - u_L)}{g_L(p_0) + g_R(p_0)} \\ u_{*TR} &= \frac{1}{2}(u_R + u_L) + \frac{1}{2}(p_{*TS} - p_R) * g_R(p_0) \end{aligned} \tag{32}$$

where,

$$g_K(p) = \left[\frac{A_K}{p + B_K} \right]^{\frac{1}{2}}, \qquad p_0 = \max(0, p_{*PV}) \tag{33}$$

Note that, A_K and B_K are defined with the exact Riemann solver. Also, p_{*PV} is the solution obtained by PVRS approximation was also defined previously. The densities are given as:

$$\rho_{*K} = \rho_K \left[\frac{\frac{p_*}{p_K} + \frac{(\gamma-1)}{(\gamma+1)}}{\frac{(\gamma-1)}{(\gamma+1)} \frac{p_*}{p_K} + 1} \right] \tag{34}$$

where, K is either L or R for left and right values respectively, of density at the star region.

A.4.1.3.4 An Hybrid Scheme

Each of the three solvers introduced above are accurate and advantageous for different conditions. Therefore, a combined scheme as depicted in the flowchart below is proposed in order to enhance the range of accuracy. It is worth noting that, Q_{user} in this scheme is suggested to be taken as 2.0, by many researchers based on empirical evidence.

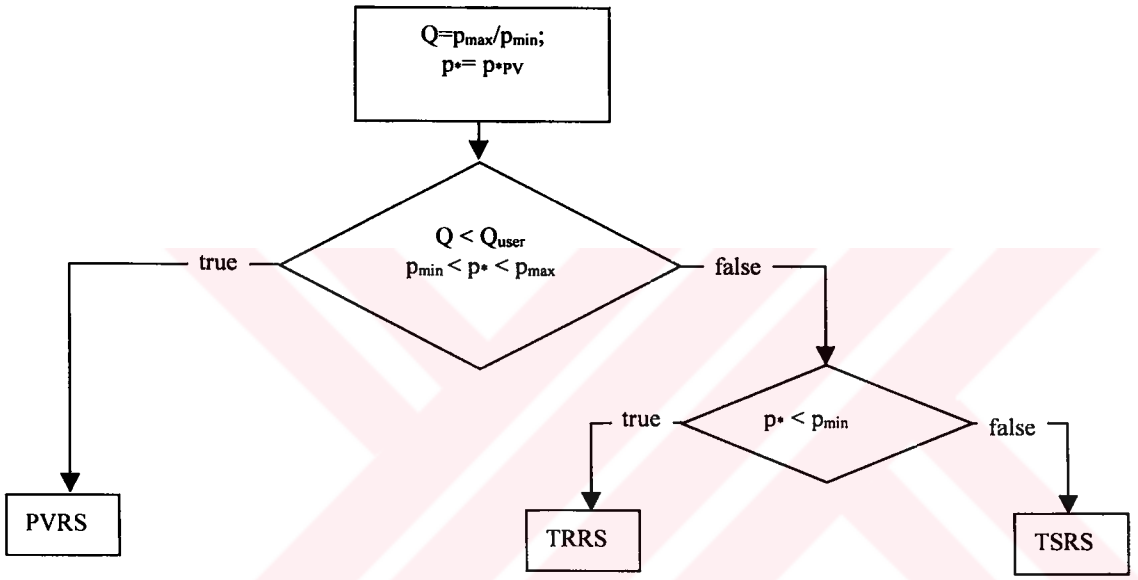

Figure A-7: Flowchart for the hybrid scheme to find p* and u* in HLLC Riemann solver

A.4.1.4 SUMMARY OF THE METHOD

The program developed in the present study, named GodunovH.cpp provided in Appendix E, is the main program of the Godunov scheme using HLLC approximate Riemann solver. This program uses HLLC.h header file that contains a C++ class that is calculating HLLC flux. The only difference between the exact Riemann version is the change of the class of flux variable, a Riemann class for the

former. Solvexy() component of the class calculates the HLLC flux. The solution methodology is as follows:

1. Give the initial data as $U_L=U_i$, $U_R=U_{i+1}$ for a cell.
2. Calculate the characteristics of the contact discontinuity, p_*, u_* using the flowchart given at … and formulas given … through … .
3. Calculate the Wave Speeds S_L, S_*, S_R as it is described at section …
4. Calculate the HLLC flux for x/t=0.0 with the … and …
5. Repeat the steps 1 to 5 for each cell
6. Calculate the U^{n+1} using the fluxes calculated at step 4.
7. Go to step 1 to calculate the new time step.

A.4.1.5 NUMERICAL RESULTS

The same tests made for Godunov method with exact Riemann solver are adopted for the HLLC solver. The results seems to be as good as the exact Riemann solver. That is such a non-iterative scheme seems to be very effective and much cheaper.

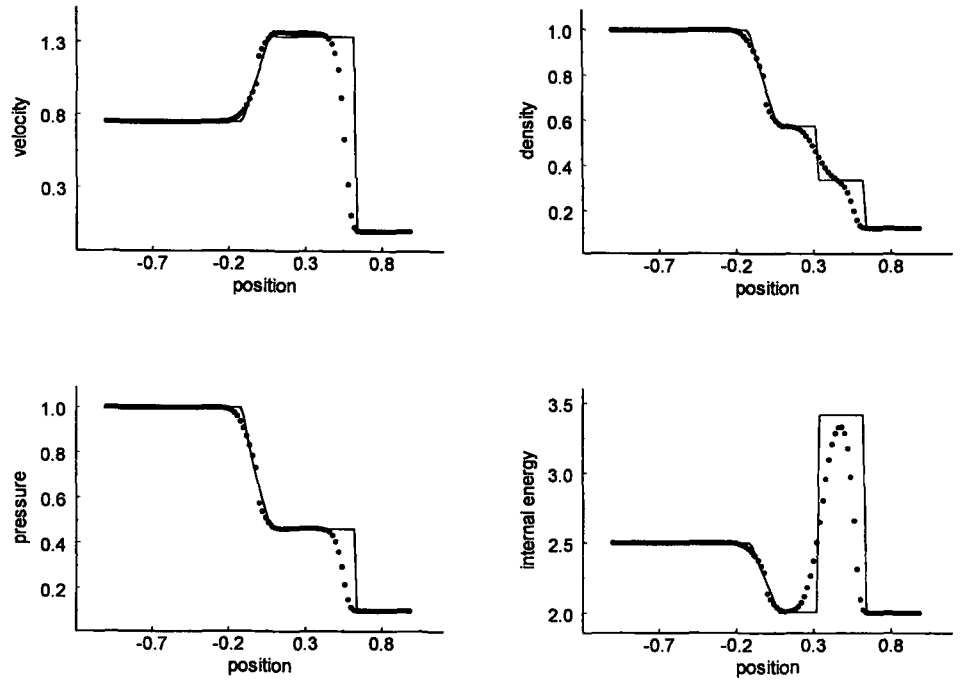

Figure A-8: Test 1 on Godunov Solver using HLLC Riemann solver

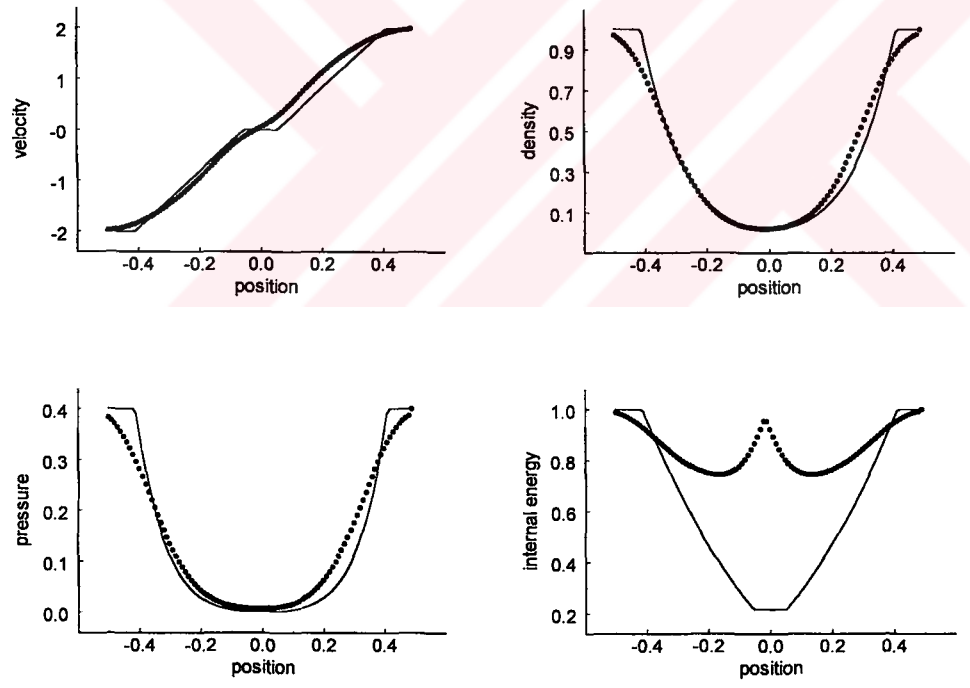

Figure A-9: Test 2 on Godunov Solver using HLLC Riemann solver

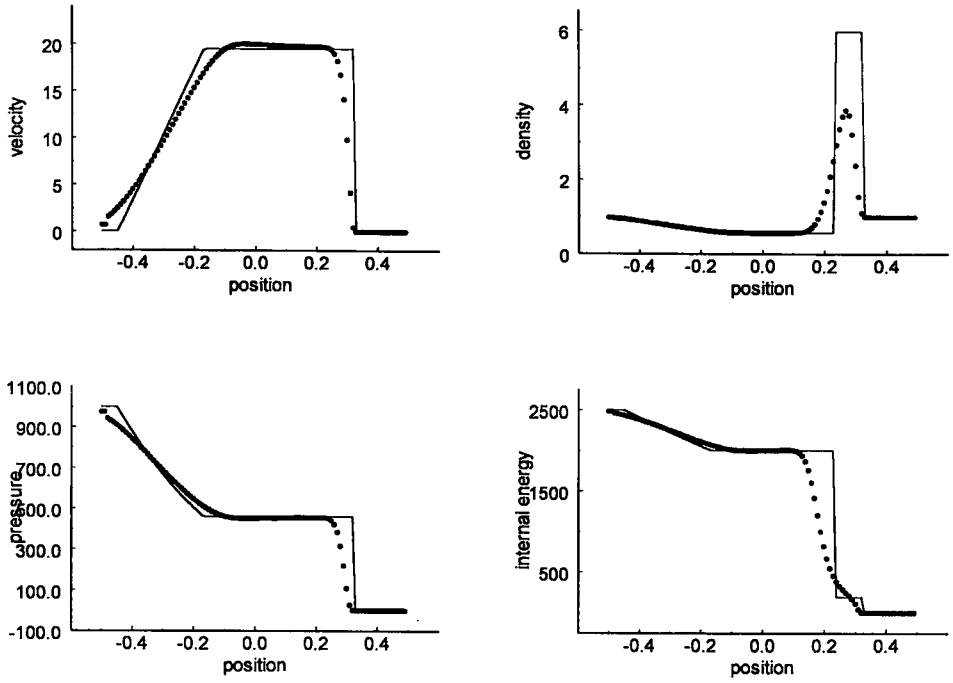

Figure A-10: Test 3 on Godunov Solver using HLLC Riemann solver

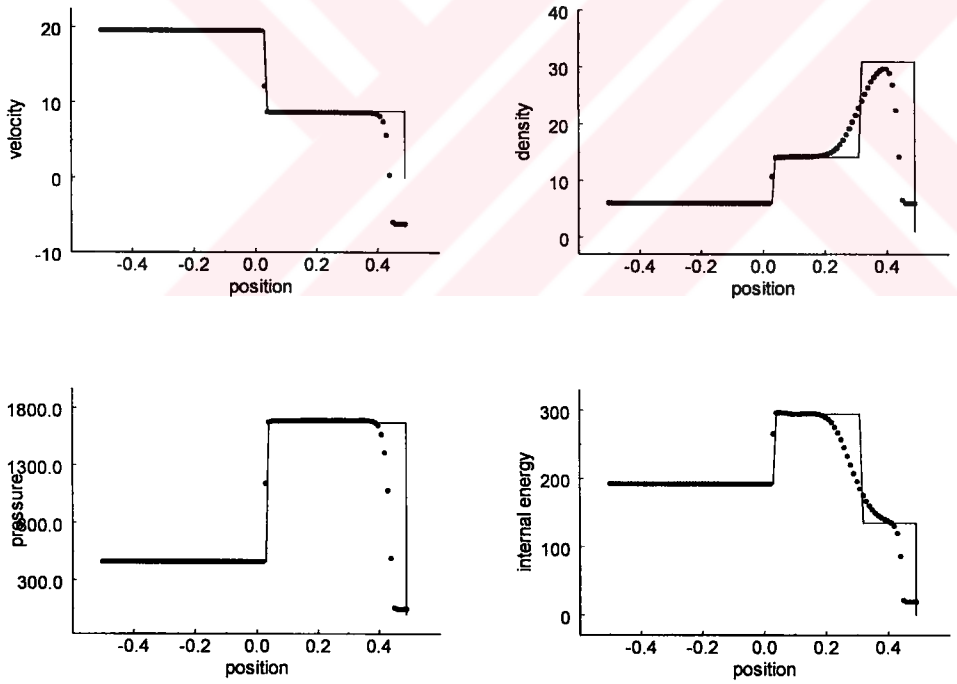

Figure A-11: Test 4 on Godunov Solver using HLLC Riemann solver

A.5 HIGHER ORDER METHODS

Up to this point, only first order methods are presented. From this point forward, some methods, which are higher order accurate, based on Godunov's approach will be presented. Some problems on stability and accuracy as well as some refinements on these methods will be discussed.

A.5.1 WEIGHTED AVERAGE FLUX (WAF) METHOD:

In this method, fluxes are calculated at the mid-cells, too. However, unlike the Godunov's method, rather than assuming a constant distribution of the u_i, one considers u_i as the integral average of the $u(x,t^n)$ through $x_{i-1/2}$ to $x_{i+1/2}$ such that:

$$u_i^n = \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} u(x,t^n)dx \qquad (35)$$

Therefore, all required is the definition of the numerical flux at the mid-cells. In the original presentation of the WAF method, the intercell flux is defined as:

$$f_{i+1/2}^{waf} = \frac{1}{\Delta x}\int_{-\Delta x/2}^{\Delta x/2} f\left[u_{i+1/2}(x,\frac{\Delta t}{2})\right]dx \qquad (36)$$

Note that, $u_{i+1/2}(x,\Delta t)$ is the solution of the shock tube problem in the cell from Euler equation.

Then the rest of the solution is the same as the Godunov solver so that.

$$U_i^{n+1} = U_i^n + \frac{\Delta t}{\Delta x}(F_{i-1/2} - F_{i+1/2}) \qquad (37)$$

A.5.1.1 (WAF) SCHEMES FOR COMPRESSIBLE GAS SOLVER

In the previous section, a numerical weighted average flux was defined such that;

$$F_{i+1/2} = \frac{1}{\Delta x}\int_{-\Delta x/2}^{\Delta x/2} F\left[u_{i+1/2}(x, \frac{\Delta t}{2})\right]dx \qquad (38)$$

Considering that solution of an Euler equation with shock tube data includes three waves of speeds S_1, S_2, S_3 (which are S_L, S_* and S_R for our case, where S_L and S_R are shock or rarefaction waves and S_* is contact discontinuity and is always in the middle) as in the figure A-12 and A-13 below. So that there are four states, which is the left data state $U^{(1)}=U_i^n$, the left side of the star region, $U^{(2)}=U_{*L}$, the right side of the star region, $U^{(3)}=U_{*R}$ and which is the right data state, $U^{(2)}=U_{i+L}{}^n$,. Then, since constant states are assumed, the integral term in eqn (38) reduces to the summation:

$$F_{i+1/2} = \sum_{k=1}^{N+1} \beta_k F_{i+1/2}^k \qquad (39)$$

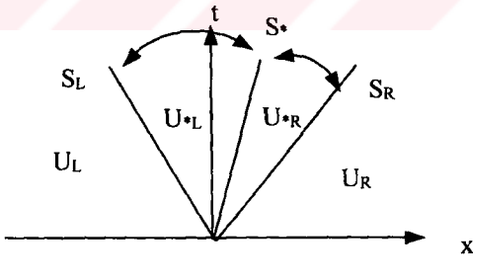

Figure A-12: States of Riemann problem of Euler equations

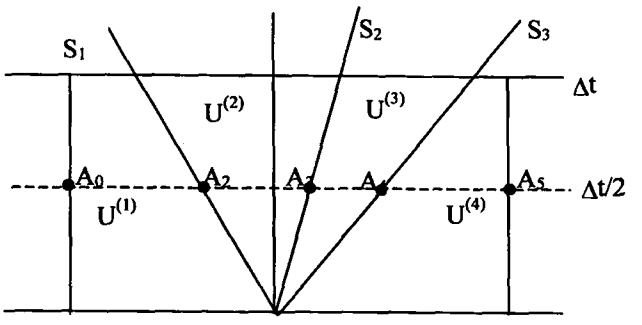

Figure A-13: WAF discretisation of a cell

where $F^{(k)}_{i+1/2}=F(u^{(k)})$, N is the number of waves in the solution of Riemann problem (so that in the case defined above N=4, k=1,.., 4) and β_k is the "weight" of the wave defined

$$\beta_k = \frac{|A_{k-1}A_k|}{\Delta x} \tag{40}$$

It can be seen from the figure A-13 that these "weights" can be determined using the wave speeds S_k such that

$$\beta_k = \frac{1}{2}(c_k - c_{k-1}) \tag{41}$$

$$c_k = \frac{\Delta t S_k}{\Delta x}, \qquad c_0=-1, \qquad c_{N+1}=1 \tag{42}$$

Note that, c_k is the normalised position of A_k, where $c_0/2$ is at the left data state (-0.5), $c_{N+1/2}$ is at the right data state (0.5), $c_k/2$ is the distance travelled by the wave S_k after $\Delta t/2$ seconds.

A.5.1.2 THE RIEMANN SOLVER:

It seems appropriate to use three waves and four constant states, yet it is not a must. One can use any number of segments to have weighted average. For example,

is the Godunov solver the number of waves taken is 0 and hence, there is only one data state which is $U^{(1)}=U(x/t=0)$ with the weight $\beta_1=1$. That is, many schemes are subset of WAF type schemes and any number of points can be used.

Actually, the type of the Riemann solver has an important role in the selection of the number of waves. For example, if one uses the exact Riemann solver, since the rarefaction wave is a non-constant state, one must use a special treatment. It would be better to have more than one point inside the rarefaction fan. If HLL type of solver is used, since contact discontinuity is not considered in this solver, two waves, S_L and S_R are separating the constant states U^L, U^R and U_*^{hll} will be enough. For HLLC solver, addition of contact wave S_* will bring the third wave and fourth state.

Concerning about the various characteristics of the Riemann solvers, HLLC solver is chosen due to following reasons:

- They are non-iterative, so that much faster than the exact solver.
- Since F_{*R}^{hllc} and F_{*L}^{hllc} through the respective star regions are assumed to be constant, it is enough to calculate four states only. For the exact solver, one needs four to six additional points to estimate a correct flux coming from the rarefaction wave.
- Their accuracy is as good as the exact solver. They are slower but more accurate than the HLL type solvers. Furthermore, they are simpler to use as compared to the other possibilities such as Roe's and Osher's solver.

A.5.1.3 PROCEDURE FOR THE WAF METHOD

The code for the WAF method is given Appendix E in WAF.cpp file. This file uses HLLCWAF.h header file, which is a modified version of HLLC.h. The modifications are as follows:

- HLLCWAF.h computes four flux vectors since WAF uses all four states whereas, HLLC.h calculates only the flux at x/t=0.
- HLLCWAF.h calculates the densities at the right and the left star regions. These values will be used in section (A.4.4)

Then the following procedure is applied within the main program:

1. For a cell equate the left and right data states as $U_i=U_L$, $U_{i+1}=U_R$,
2. Calculate the S_L, S_*, S_R as it is described at the previous section,
3. Calculate the $F^{(1)}=F_L$, $F^{(2)}=F_{*L}^{hllc}$, $F^{(3)}=F_{*R}^{hllc}$, $F^{(4)}=F_R$,
4. Calculate the weights c_k, k=1,…,5 using $S^{(1)}=S_L$, $S^{(2)}=S_*$, $S^{(3)}=S_R$
5. Calculate the flux $F_{i+1/2}$
6. Repeat steps 1 to 5 for each cell
7. Calculate new values of U_i,U_I^{n+1} for each cell
8. Repeat steps 1 to 7 to the next time level

A.5.1.4 NUMERICAL RESULTS

The same set of tests used with the Godunov-Exact and Godunov- HLLC solver is used for the WAF method, too. The results are shown in figures below with the same order. One easily observes high oscillations especially around sharp edges.

The reasons and corrections for these oscillations will be discussed at the section A.4.3 The results seem to of no utility.

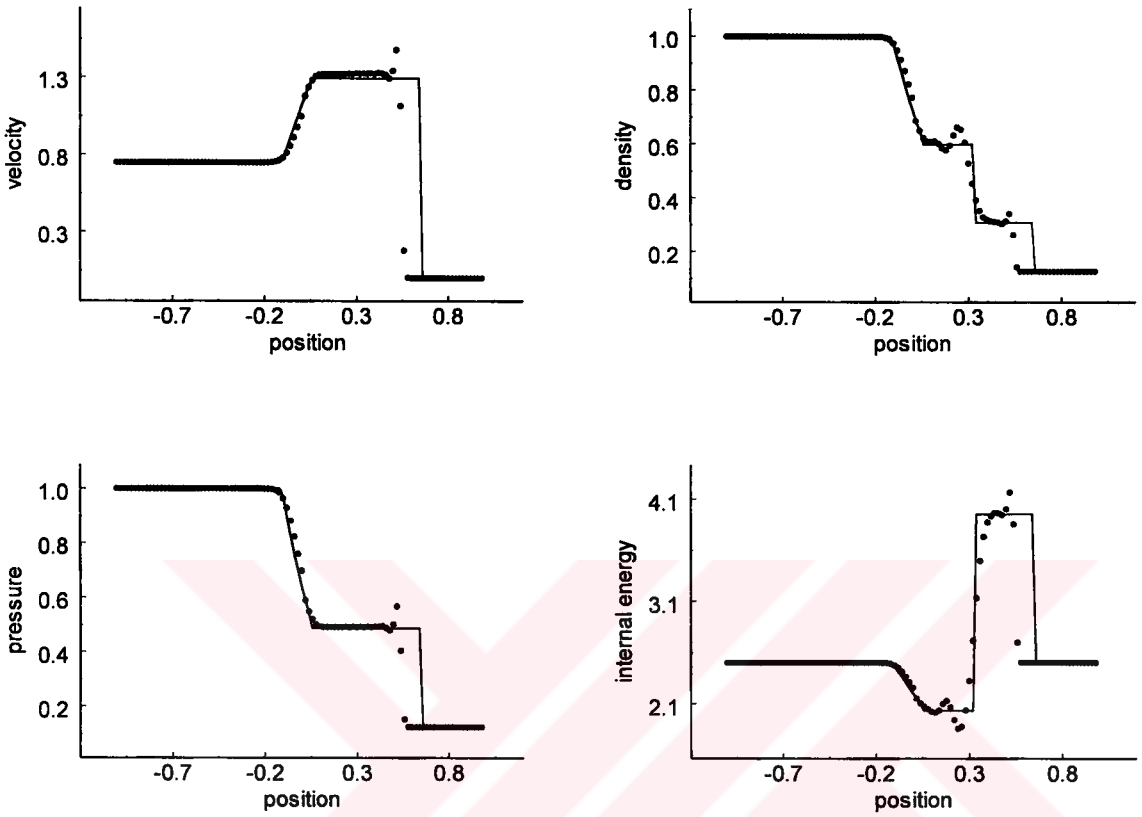

Figure A-14: Test 1 applied on WAF method

A.5.2 MUSCL METHOD

The second higher order method that will be discussed is the famous MUSCL method of van Leer. MUSCL stands for Monotone Upstream Scheme for Conservation Laws. MUSCL is a very popular scheme and has many variants. In this study, some basic versions are examined.

In the Godunov's method one assumes a constant distribution of U_i within the cell. In the MUSCL type schemes, a piece-wise linear distribution of U_i in a cell is assumed. The difference between these methods can be seen from at the figure A-15.

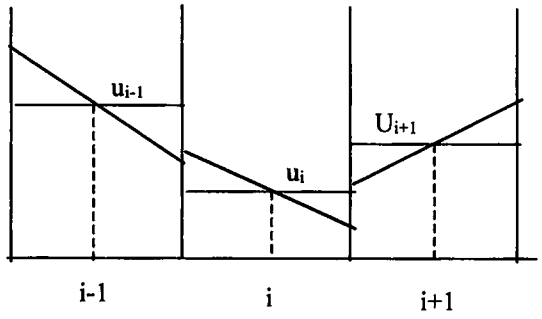

Figure A-15: Reconstruction of the data for MUSCL method

So that the local reconstruction of u_i^n obtained based on Figure A-15 is given as;

$$u_i(x) = u_i^n + \frac{(x - x_i)}{\Delta x}\Delta_i \quad x \in [0, \Delta x] \tag{43}$$

Δ_i is a vector representing the slope of U_i. So one can define extrapolated values to left and right such that;

$$u_i^L = u_i^n - \frac{1}{2}\Delta_i \tag{44}$$

$$u_i^R = u_i^n + \frac{1}{2}\Delta_i \tag{45}$$

Consider the generalized Riemann problem

$$u_t + f(u)_x = 0 \tag{46.a}$$

$$u(x,0) = \begin{cases} u_i(x), & x < 0 \\ u_{i+1}(x), & x > 0 \end{cases} \tag{46.b}$$

Then, using the MUSCL scheme, the problem reduces to:

$u_t+f(u)_x=0$

$$u(x,0)=\begin{cases}u_i^R, & x<0\\ u_{i+1}^L, & x>0\end{cases} \qquad (47)$$

The choice of the Δ_i is not clear yet. The classical choice of the slope is given as

$$\Delta_i=\frac{1}{2}(1+\omega)\Delta u_{i-1/2}+\frac{1}{2}(1-\omega)\Delta u_{i+1/2} \qquad (48)$$

where

$$\Delta u_{i-1/2}\equiv u_i^n-u_{i-1}^n \quad \text{and} \quad \Delta u_{i+1/2}\equiv u_{i+1}^n-u_i^n \qquad (49)$$

and ω is a free parameter in the interval [-1,1].

The higher order extensions of the method can be found in the literature. For example Collela and Woodward [33] presented a parabolic distribution of the data through the cell. The method is called as The Piece-wise Parabolic Method (PPM).

Many schemes are subset of the MUSCL method. For example, for $\Delta_i=0$ the method reduces to Godunov's first order method. For $\omega=0$ the scheme is nothing but a central differencing; for $\omega=1/2$ the method is called Fromm's method.

A.5.2.1 A VARIANT OF THE SCHEME

One of the most popular variants of the scheme is the one proposed by Van Leer. This method includes an additional step, (can be named a corrector step), to estimate left and right data states.

Step (I): The same procedure as the basic procedure; data reconstruction step with boundary extrapolated values such that;

$$u_i^L = u_i^n - \frac{1}{2}\Delta_i \qquad u_i^R = u_i^n + \frac{1}{2}\Delta_i \tag{50}$$

Step (II): Evolution of u_i^L, u_i^R by a time $1/2\Delta t$ as;

$$\bar{u}_i^L = u_i^L + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[f(u_i^L) - f(u_i^R)\right] \tag{51}$$

$$\bar{u}_i^R = u_i^R + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[f(u_i^L) - f(u_i^R)\right] \tag{52}$$

Step (III): Solution of piece-wise constant data Riemann problem:

$$u_t + f(u)_x = 0$$

$$u(x,0) = \begin{cases} \bar{u}_i^R, & x < 0 \\ \bar{u}_{i+1}^L, & x > 0 \end{cases} \tag{53}$$

to find the similarity solution $u_{i+1/2}$ (x/t)

The intercell numerical flux $f_{i+1/2}$ is obtained with exactly the same way as in the Godunov's method.

A.5.2.2 PROCEDURE FOR THE MUSCL METHOD

The main program for the MUSCL method is given in appendix E in MUSCL.cpp file. This file uses HLLC.h header file, containing a HLLC flux class. The procedure is as follows;

1. For the first step, the slope vector Δ_i must be calculated for a cell. For this purpose one can use different choices of variables, such as a primitive value, W=(ρ,u,p) or a physical variable, U. To achieve a faster and simpler code, extrapolation of physical variables are chosen. For this case the slope vector Δ_i is calculated as:

$$\Delta_i = \frac{1}{2}(1-\omega)\Delta_{i-1/2} + \frac{1}{2}(1+\omega)\Delta_{i+1/2} \tag{54}$$

where

$$\Delta_{i-1/2} \equiv U_i^n - U_{i-1}^n \; ; \qquad \Delta_{i+1/2} \equiv U_{i+1}^n - U_i^n \tag{55}$$

and $\omega \in [-1,1]$

2. Estimate the left and right extrapolated values as

$$U_i^L = U_i - \frac{1}{2}\Delta_i \; ; \qquad U_i^R = U_i + \frac{1}{2}\Delta_i \tag{56}$$

3. Improve values of U_i^L and U_i^R as;

$$\overline{U}_i^L = U_i^L + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[F(U_i^L) - F(U_i^R)\right] \tag{57.a}$$

$$\overline{U}_i^R = U_i^R + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[F(U_i^L) - F(U_i^R)\right] \tag{57.b}$$

4. Compute the intercell flux $F_{i+1/2}$ the same way as in the Godunov method by solving the conventional Riemann problem with initial data

$$u_L \equiv \overline{u}_i^R \; ; \qquad u_R \equiv \overline{u}_{i+1}^L \tag{58}$$

5. Repeat steps 1-4 for each cell
6. Compute u_I^{n+1} for each cell as it is in Godunov method.
7. Repeat steps 1-6 for the next time step.

A.5.2.3 NUMERICAL RESULTS

Only one test is run for MUSCL method. The result is shown in the figure below. It can easily be observed that there exist big oscillations near high gradients, same as but more severe than the case in WAF method presented previously. The reasons and the solutions for this unacceptable phenomenon will be presented in the proceeding section.

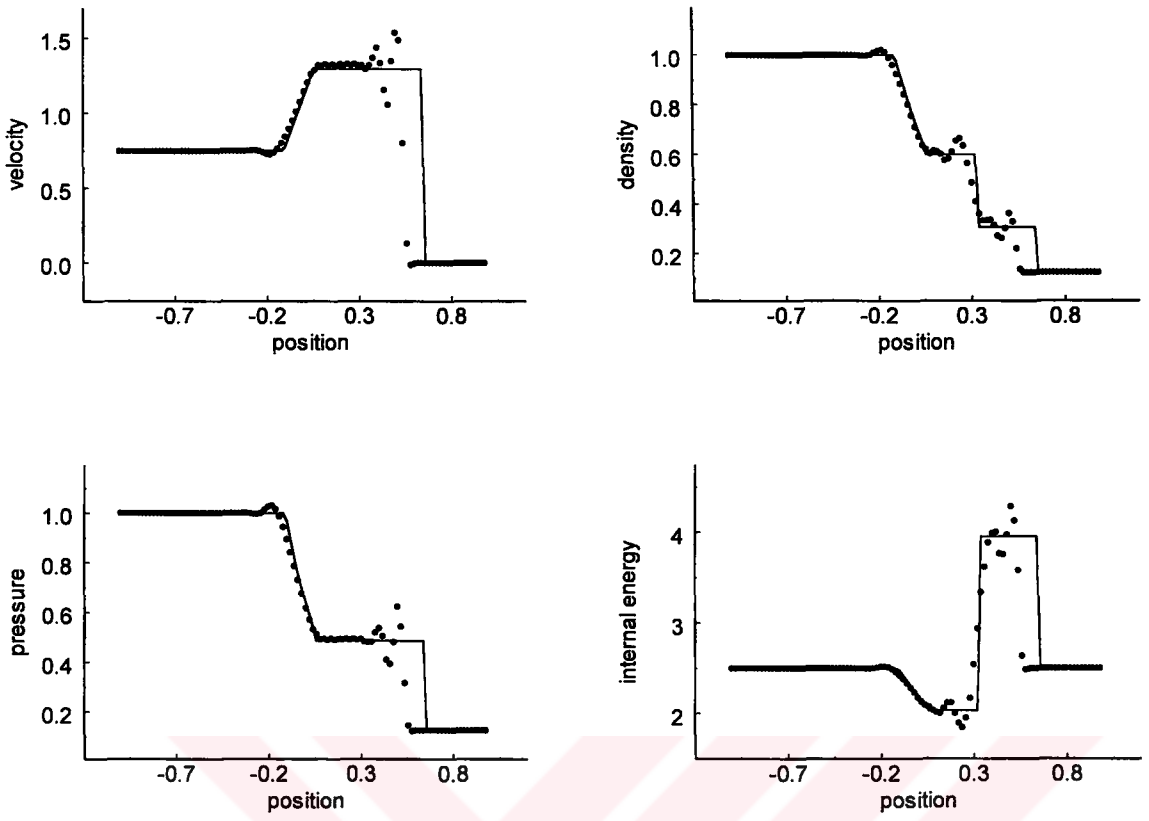

Figure A-16: Test 1 applied on WAF method

A.5.3 PROPERTIES OF HIGHER ORDER SCHEMES

Even though higher order schemes are much better simulating the smooth variations, the solutions made available in the two previous sections obtained using in second (or higher) order schemes, demonstrate large oscillations near high gradients or discontinuities. This type of behaviour is not inherent in the Euler and Navier-Stokes equations. Indeed, 1-D solutions do not have such anomalies. To improve the

response of higher order solutions there have been proposed several fixing schemes, some of which are introduced below.

A.5.3.1 MONOTICITY

A scheme is said to be monotone if it does not lead to an oscillation in the numerical solution. Let us write the discretised form of the scalar conservation law;

$u_t + f_x = 0$ as;

$$u_i^{n+1} = H(u_{i-k}^n, u_{i-k-1}^n, ..., u_{i+k}^n) \qquad (59)$$

where, k is a positive integer.

The scheme is said to be monotone if H is a monotone increasing function of each of its arguments; that is:

$$\frac{\partial H}{\partial u_j^n} = 0 \qquad \forall j \qquad (60)$$

It is shown by Harten that, converged solutions of a monotone scheme always correspond to physically acceptable states. In other words, they do not produce physically non-realizable solutions.

A.5.3.2 GODUNOV'S THEOREM

Godunov's theorem states that "There are no monotone linear schemes for equation $[\partial U / \partial t + \partial F / \partial x = 0]$ of second or higher order of accuracy". Another way to express Godunov's theorem is that the monotone schemes are first-order accurate.

A.5.3.3 DATA COMPATIBILITY

A good approach to fix a high order scheme may be to enforce the solution to adjust itself to the nature of the solution. This kind of a approach leads to a variable coefficient scheme.

The definition of the data compatibility is given below:

Definition: A scheme is compatible with a given data set $\{u_i^n\}$if the solution u_i^{n+1} at each point i as given by the algorithm, is bounded by the upwind pair (u_{i-s}^n, u_i^n), where,

$$s \equiv sign(a) = \frac{a}{|a|} \qquad (61)$$

Now, one can rephrase the same definition such that:

$$\min\{u_{i-s}^n, u_i^n\} \le u_i^{n+1} \le \max\{u_{i-s}^n, u_i^n\} \qquad (62)$$

The most important feature of a data compatible scheme is its upwind characteristics. The value node u_i^{n+1} being calculated is bounded only in the upwind direction, which means that, its bound changes only in the direction of propagation of information.

Proposition: The data compatibility condition is equivalent to requiring

$$0 \le \frac{u_i^{n+1} - u_i^n}{u_{i-s}^n - u_i^n} = r \le 1 \qquad (62)$$

A.5.3.4 TOTAL VARIATION DIMINISHING (TVD) METHODS

Given a function u= u(x). The total variation is defined as:

$$TV = \int \left| \frac{\partial u}{\partial x} \right| dx \qquad (63)$$

and in a discretized form

$$TV = \int \left| u_{i+1}^{n} - u_{i}^{n} \right| dx \qquad (64)$$

A numerical scheme is said to be total variation diminishing, if

$$TV(u^{n+1}) \leq TV(u^{n}) \qquad (65)$$

Definition: If the following monotonicity properties are maintained as a function at t:

1. No new local extrema in x can be created
2. The value of local minimum is non-decreasing, the value of local minimum is increasing.

Then the scheme is said to be *monotonicity preserving*. This means that, if u_i^n is monotone, then u_{i+1}^n is also monotone.

Then the following hierarchy exists:

1. All monotone schemes are TVD
2. All TVD schemes are monotonicity preserving.

A.5.3.4.1 TVD Limiters

Based on the above discussion concludes that a scheme can be forced to be monotone by limiting (diminishing) the total (and/or local) variations. Use of

specially designed limiters can maintain monotonicity and restrict the oscillations. That is the algorithm of an TVD scheme is.

1. Select a first-order numerical flux
2. Extend it to the second order accuracy
3. Restrict the amplitude of the gradient appearing in the additional terms by limiters to ensure TVD conditions

Two types of limiters will be presented in proceeding sections; Flux limiters and slope limiters.

A.5.3.4.2 TVD Regions

Considering the equation (62) in section A.4.3.3, it is seen that if the scheme is convergent, r is bounded by the constants of the scheme. Now, let's make a new definition of r

$$r_{i+1/2} = \begin{cases} \dfrac{u_i^n - u_{i-1}^n}{u_{i+1}^n - u_i^n} & ,a > 0 \\ \dfrac{u_i^n - u_{i-1}^n}{u_{i+1}^n - u_i^n} & ,a < 0 \end{cases} \qquad (66)$$

or $r_{i+1/2}$ is defined as the ratio of upwind changes to the local changes shown by,

$$r_{i+1/2} = \frac{\Delta_{upw}}{\Delta_{Loc}} \qquad (67)$$

It can be shown that any first order scheme can be described as a combination of Lax- Wendroff and Warming and Beam scheme as discussed in Hirsch [24]. Now,

let's define a limiter function of $\phi(r)$. Based on the definitions given above and these two schemes, one can define a region that the ϕ is limiting the total variation such as;

$\psi > 0$

$\psi < 2$

$\psi > 2r$

The graphical representation of this region is the shaded area in figure A-16

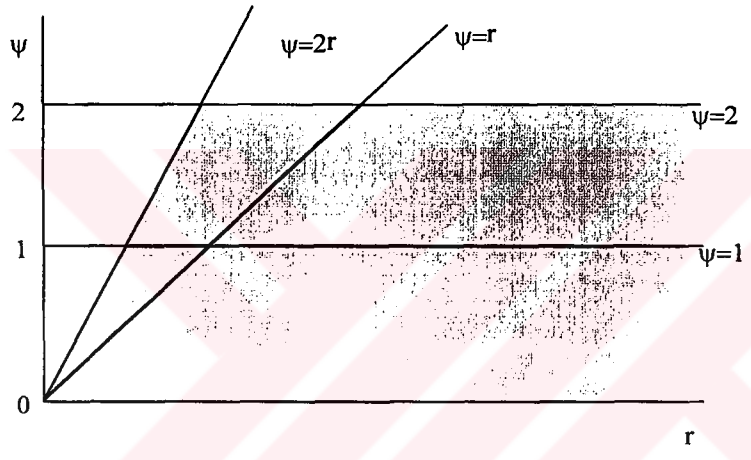

Figure A-17: TVD Region for second order schemes

Then, ϕ is reported to be between the lines $\phi=1$ and $\phi=r$. The limiter function which is outside of this area is reported to be over-compressive by Sweby. So selection of a limiter function in this area seems to be useful. The limiter region is shown by the shaded area at the figure A-17 below.

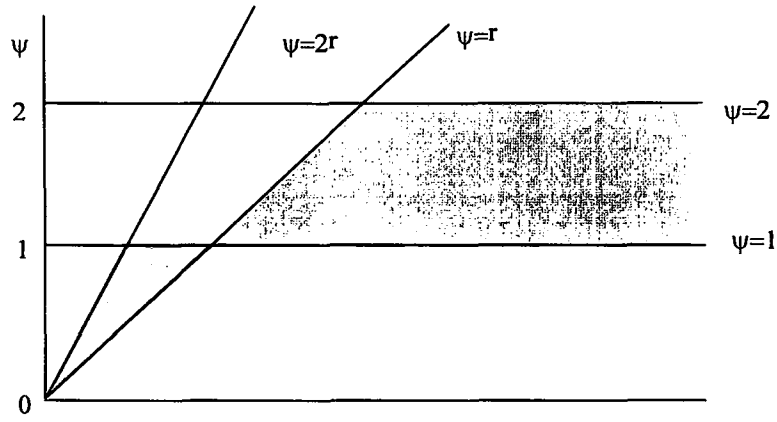

Figure A-18: Limiter region for second order TVD schemes

There are number of limiter functions using this shaded area. These limiter functions are shown below. The two extremes are superbee and minbee limiters. Superbee follows the uppermost line, so that it allows higher variations, in other words higher gradients. This results in better simulation of shocks, but sometimes under-limiting of the smooth solutions.

On the other hand, minbee follows the lowermost line, so it over-limits at the discontinuities, but very successful in smooth data. Van-Albada and Van-Leer limiters are two approximations between these two extremes.

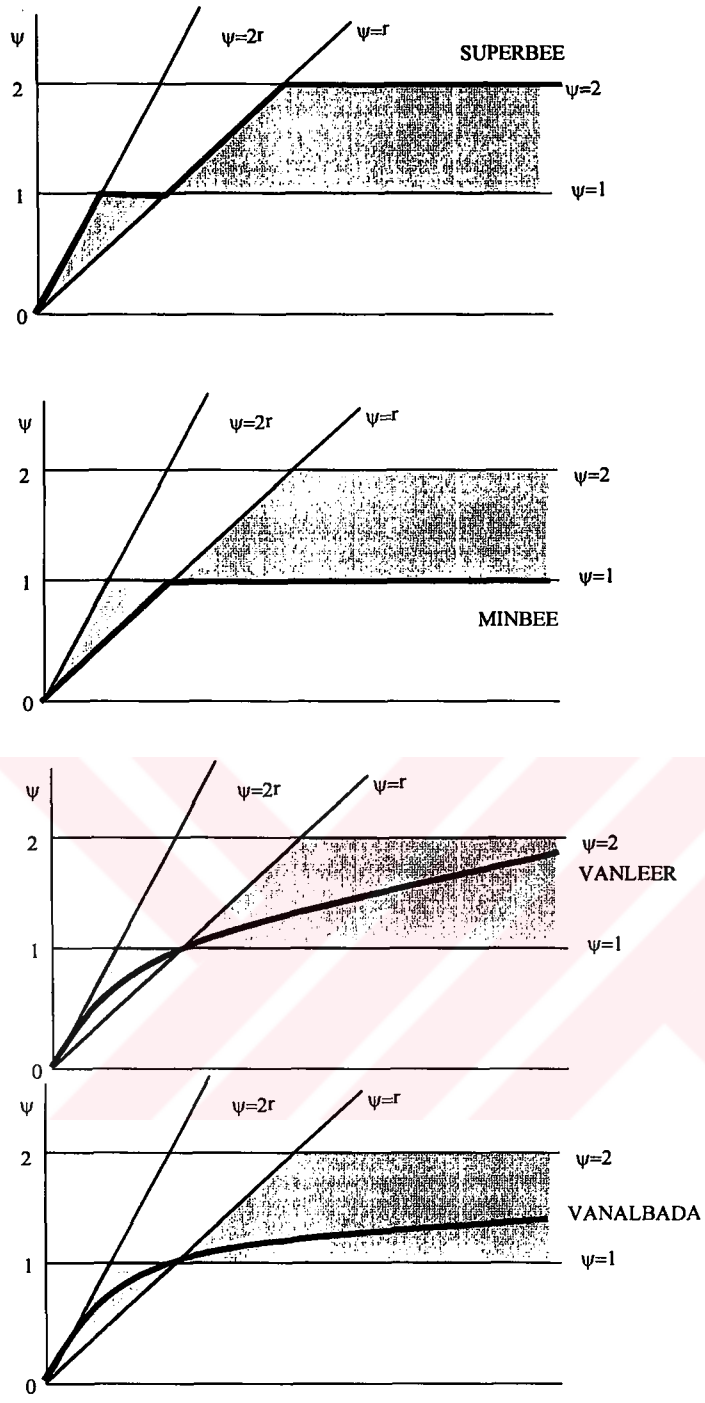

Figure A-19: The paths of different TVD limiters

The limiter function ψ will not be used in the text. Instead, two different types of TVD limiters, Flux limiter function, ϕ, and Slope limiter function, ξ, will be presented. These functions are based on ψ; such a detailed description of the

relationship among these will not be iterated. However, it must be noted that the limiter function ψ is the basis of the TVD method, regardless the type of the conservation equation i. All other types of limiters are adaptation to schemes and the equations.

A.5.4 TVD VERSION OF WAF METHOD

In the section A.4.1, it is seen that WAF method creates spurious oscillations. To solve this problem, the TVD version of the method is given below.

The most important feature of the WAF method is that it separates region into three waves. Then, recalling the limiter variable r being the ratio of upwind changes to the local changes, one must determine the upwind direction for each cell. However, there are three waves in a cell, which can exhibit different upwind properties. So that the limiting process for a cell must be conducted for three phases separately. Then weighted average of fluxes are considered. The limiting process made on fluxes seems to be cheaper than that made on variables. Hence, flux limiters are used for the TVD version.

The TVD modification of WAF method using the flux limiters is given as:

$$F_{i+1/2} = \frac{1}{2}(F_i + F_{i+1}) - \frac{1}{2}\sum_{k=1}^{N} \text{sign}(c_k)\phi_{i+1/2}^{(k)} \Delta F_{i+1/2}^{(k)} \qquad (68)$$

where ϕ is the flux limiter and function of r.

$$\phi_{i+1/2}^{(k)} = \phi_{i+1/2}(r^{(k)}) \qquad (69)$$

the parameter $r^{(k)}$ refers to the upwind change of the wave k. $r^{(k)}$ is given as

$$r^{(k)} = \begin{cases} \dfrac{\Delta q^{(k)}_{i-1/2}}{\Delta^{\bullet} q^{(k)}_{i+1/2}} & ,c_k > 0 \\ \dfrac{\Delta q^{(k)}_{i+3/2}}{\Delta q^{(k)}_{i+1/2}} & ,c_k < 0 \end{cases} \qquad (70)$$

q is a flow parameter. The best choices for q are given as density, ρ, specific internal energy, e. For the sake of simplicity, density is, ρ chosen.

A.5.4.1 FLUX LIMITERS

Flux limiters for WAF flux are equivalent to ψ, such as:

$$\phi(r) = 1 - (1 - |c|)\,\psi(r) \qquad (80)$$

Omitting to subscripts and superscripts for the sake of convenience, the following limiter functions are given;

Superbee:

$$\phi_{sb}(r,|c|) = \begin{cases} 1 & \text{if } r \le 0 \\ 1 - 2(1-|c|)r & \text{if } 0 \le r \le 1/2 \\ |c| & \text{if } 1/2 \le r \le 1 \\ 1 - (1-|c|)r & \text{if } 1 \le r \le 2 \\ 2|c| - 1 & \text{if } r \ge 2 \end{cases} \qquad (81)$$

Van Leer:

$$\phi_{vl}(r,|c|) = \begin{cases} 1 & \text{if } r \le 0 \\ 1 - \dfrac{(1-|c|)2r}{(1+r)} & \text{if } r \ge 0 \end{cases} \qquad (82)$$

Van Albada:

$$\phi_{va}(r,|c|) = \begin{cases} 1 & \text{if } r \le 0 \\ 1 - \dfrac{(1-|c|)r(1+r)}{(1+r^2)} & \text{if } r \ge 0 \end{cases} \tag{83}$$

Minbee:

$$\phi_{mb}(r,|c|) = \begin{cases} 1 & \text{if } r \le 0 \\ r & \text{if } 0 \le r \le 1 \\ |c| & \text{if } r \ge 1 \end{cases} \tag{84}$$

A.5.4.2 PROCEDURE FOR WAF METHOD WITH TVD LIMITERS

The code for the TVD version of WAF method is given in Appendix E as WAFTVD.cpp file. This file uses HLLCWAF.h header file which is a modified version of HLLC.h. The modifications have been explained previously in section A.4.1.3. The densities for each stage is calculated to be used in the calculation of $r^{(k)}$. Densities in the cell (i+1/2) is stored as $\rho^{(k)}$, k=0,...,N, such that

$$r_{i+1/2}^{(k)} = \begin{cases} \dfrac{\rho_{i-1/2}^{(k)} - \rho_{i-1/2}^{(k-1)}}{\rho_{i+1/2}^{(k)} - \rho_{i+1/2}^{(k-1)}} & \text{,if } c > 0 \\ \dfrac{\rho_{i+3/2}^{(k)} - \rho_{i+3/2}^{(k-1)}}{\rho_{i+1/2}^{(k)} - \rho_{i+1/2}^{(k-1)}} & \text{,if } c \le 0 \end{cases} \tag{85}$$

Note that $\rho^{(k)}$ and $\rho^{(k-1)}$ are densities at the left and fight of the wave k.

Also the header file LIMITER.h is included in the main file. LIMITER.h includes flux limiter functions given by the equation (81) through (84). The function CHECK is the limiter function that does not limit the function, therefore gives the

same results with the unlimited version. This "Non-limiting" limiter function is defined as:

$\phi_{check}(r,c)=|c|$ for all values of r.

Then the following procedure is applied at the main program.

1. For a cell equate the left and right data states as $U_i=U_L$, $U_{i+1}=U_R$,
2. Calculate the S_L, S_*, S_R as it is described at the previous section,
3. Calculate the $F^{(1)}=F_L$, $F^{(2)}=F_{*L}{}^{hllc}$, $F^{(3)}=F_{*R}{}^{hllc}$, $F^{(4)}=F_R$,
4. Calculate the weights c_R, k=1,…,5 using $S^{(1)}=S_c$, $S^{(2)}=S_*$, $S^{(3)}=S_R$
5. Calculate r using the formula …
6. Calculate the limiter function ϕ
7. Calculate the flux $F_{i+1/2}$
8. Repeat steps 1-7 for each cells
9. Calculate u_i^{n+1} for each cell
10. Repeat steps 1-9 for the next time level

A.5.4.3 NUMERICAL RESULTS

The same tests applied to previous methods were also applied for TVD version of WAF method. The results improved significantly such that the oscillations are fixed. All the solutions presented are " limited" with superbee limiter, which is the most successful. In any event, the superbee limiter seems to be a better alternative for the simulation of shock tube data.

It should be reported that, computational costs of the TVD version is very high. The code runs four to five times slower than the first order Godunov, and at least twice as slower as the original WAF method. The code can run faster with a

good optimisation, and such an optimisation is suggested for higher order schemes. Memory is optimised good enough, and memory usage is nearly the same with the Godunov's first order method.

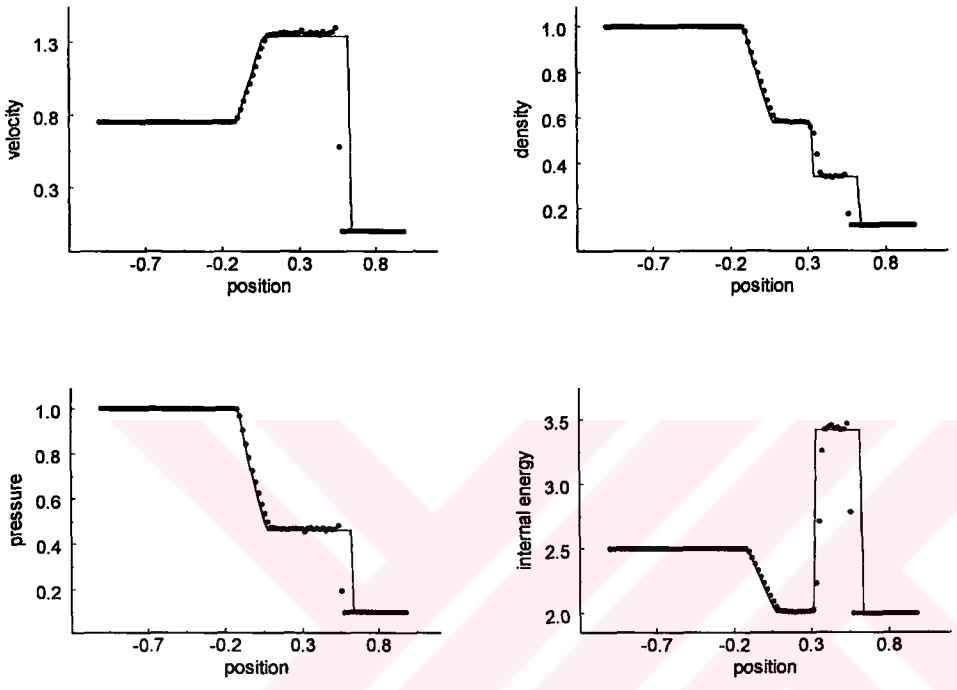

Figure A-20: Test 1 on TVD version of WAF Scheme

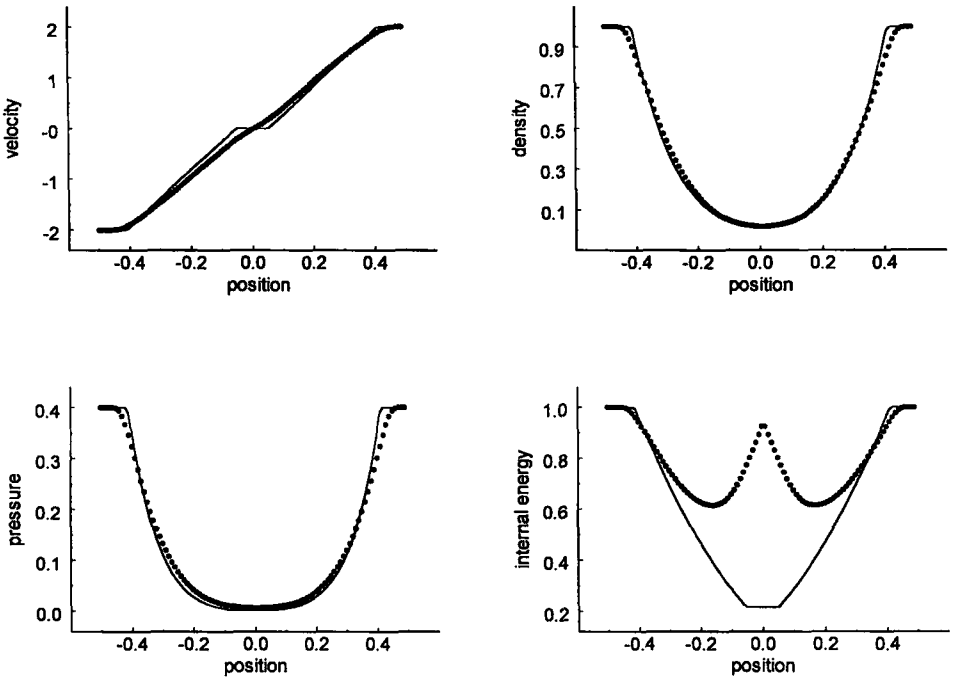

Figure A-21: Test 2 on TVD version of WAF Scheme

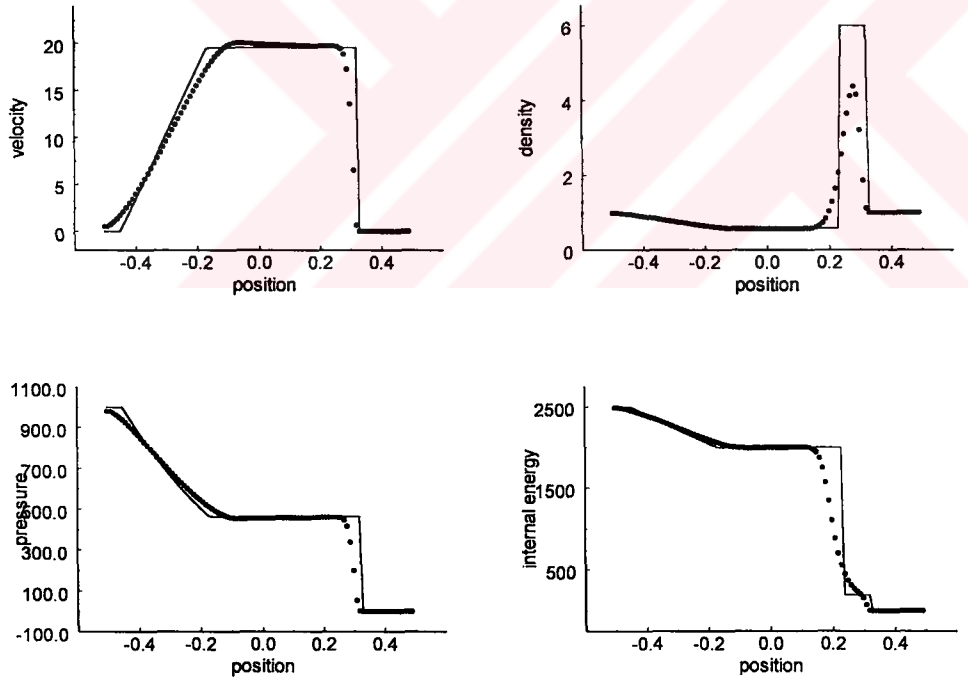

Figure A-22: Test 3 on TVD version of WAF Scheme

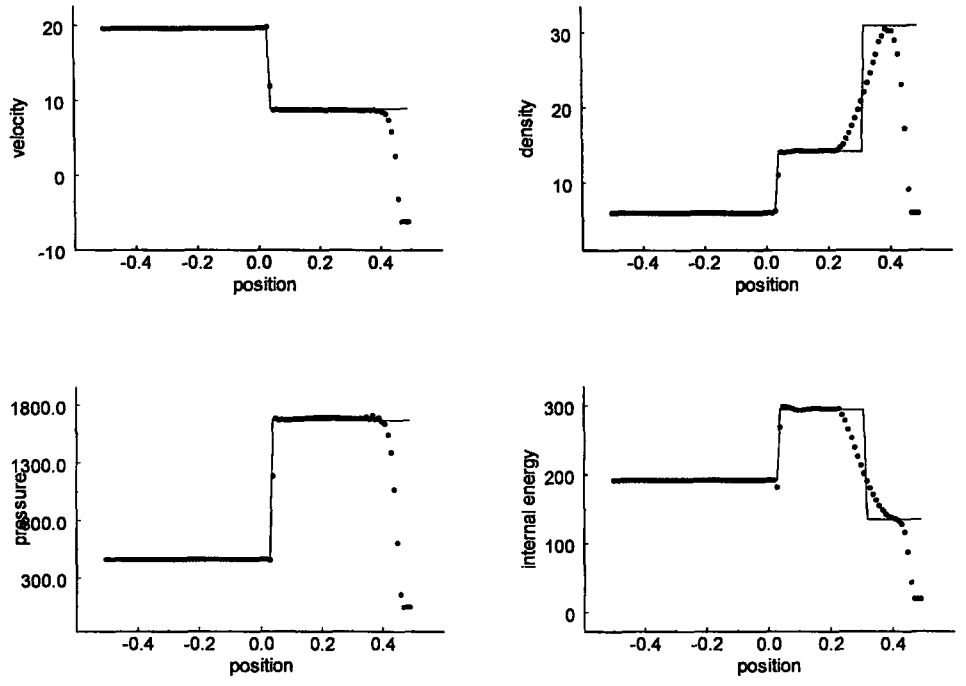

Figure A-23: Test 4 on TVD version of WAF Scheme

A.5.5 TVD VERSION OF MUSCL METHOD

Above, the MUSCL method show spurious oscillations around sharp discontinuities. This is most probably caused by over-reconstructing of the data among these waves. As mentioned before, in order to have a monotone scheme, the reconstruction of the data must be limited. Since the "reconstructing" parameter is the slope vector, Δ_i for the MUSCL method, limitation of this parameter is required to achieve monotonicity.

There are a number of ways of limiting this slope in literature. There are only two basic limiting processes; two different MUSCL schemes are considered in this study. The first approach is the limited slope approach and the second one is the slope limiters approach which is analogous to the limiters discussed previously. These

methods introduce a new limited slope $\overline{\Delta}_i$ in different ways. For both methods, remember the Δ_I is a vector entity and limiting process must be applied to the each and every component.

A.5.5.1 LIMITED SLOPES

The first approach, limiter slopes approach is simpler than the other one. The limited slopes chosen are upwind slopes $\Delta_{i-1/2}$, and $\Delta_{i+1/2}$, such that:

$$\overline{\Delta}_i = \begin{cases} \max\left(0, \min\left(\beta\Delta_{i-1/2}, \Delta_{i+1/2}\right), \min\left(\Delta_{i-1/2}, \beta\Delta_{i+1/2}\right)\right), & \Delta_{i+1/2} > 0 \\ \min\left(0, \max\left(\beta\Delta_{i-1/2}, \Delta_{i+1/2}\right), \max\left(\Delta_{i-1/2}, \beta\Delta_{i+1/2}\right)\right), & \Delta_{i+1/2} < 0 \end{cases}$$

(86)

Then this value is used instead of the Δ_i. The complete formulation is given at the section A.4.5.3. The parameter β is a limiting variable and $\beta=1$ reproduces the MINBEE limiter, while $\beta=2$ produces the SUPERBEE and $\beta=0$ reproduces the Godunov method.

A.5.5.2 SLOPE LIMITERS

The second approach aims at finding a slope limiter such that:

$$\overline{\Delta}_i = \xi_i \Delta_i \qquad (87)$$

with Δ_i defined in equation (54). ξ_i is called a slope limiter, which is analogous to the conventional flux limiters. For convenience omitting to subscripts and superscripts, the following limiter functions are given;

Superbee:

$$\xi_{sb}(r)=\begin{cases}0 & \text{if } r\le 0\\ 2r & \text{if } 0\le r\le 1/2\\ 1 & \text{if } 1/2\le r\le 1\\ \min(r,\xi(r),2) & \text{if } r>1\end{cases} \tag{88}$$

Van Leer:

$$\xi_{vl}(r,\xi(r))=\begin{cases}1 & \text{if } r\le 0\\ \min\left(\dfrac{2r}{(1+r)},\xi(r)\right) & \text{if } r\ge 0\end{cases} \tag{89}$$

Van Albada:

$$\xi_{va}(r,\xi(r))=\begin{cases}1 & \text{if } r\le 0\\ \min\left(\dfrac{r(1+r)}{(1+r^2)},\xi(r)\right) & \text{if } r\ge 0\end{cases} \tag{90}$$

Minbee:

$$\xi_{mb}(r,\xi(r))=\begin{cases}1 & \text{if } r\le 0\\ r & \text{if } 0\le r\le 1\\ \min(1,\xi(r)) & \text{if } r\ge 1\end{cases} \tag{91}$$

where,

$$\xi_R=\frac{2}{1-w+(1+w)r} \tag{92}$$

A.5.5.3 PROCEDURE FOR MUSCL WITH LIMITED SLOPES

The main program for the MUSCL method with limited slopes is given in theappendix E in MUSCLSl.cpp file. This file uses HLLC.h header file, containing a HLLC flux class. The procedure is as follows;

1. For the first step, the slope vector Δ_i must be calculated for a cell. For this purpose one can use different choices of variables, such as the primitive values, W=(ρ,u,p) or physical variables U. To achieve a faster and simpler code, extrapolation of physical variables are chosen. For this case the limited slope vector is calculated as:

$$\overline{\Delta}_i = \begin{cases} \max\left(0, \min\left(\beta\Delta_{i-1/2}, \Delta_{i+1/2}\right), \min\left(\Delta_{i-1/2}, \beta\Delta_{i+1/2}\right)\right), & \Delta_{i+1/2} > 0 \\ \min\left(0, \max\left(\beta\Delta_{i-1/2}, \Delta_{i+1/2}\right), \max\left(\Delta_{i-1/2}, \beta\Delta_{i+1/2}\right)\right), & \Delta_{i+1/2} < 0 \end{cases} \quad (93)$$

where

$$\Delta_{i-1/2} \equiv U_i^n - U_{i-1}^n \; ; \qquad \Delta_{i+1/2} \equiv U_{i+1}^n - U_i^n \quad (94)$$

with a proper choice of β.

2. Estimate the left and right extrapolated values as

$$U_i^L = U_i - \frac{1}{2}\overline{\Delta}_i \; ; \qquad U_i^R = U_i + \frac{1}{2}\overline{\Delta}_i \quad (95)$$

3. This is the 2nd step describe the evolved values of U_I^L and U_I^L as;

$$\overline{U}_i^L = U_i^L + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[F(U_i^L) - F(U_i^R)\right] \quad (96)$$

$$\overline{U}_i^R = U_i^R + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[F(U_i^L) - F(U_i^R)\right] \quad (97)$$

4. Compute the intercell flux $F_{i+1/2}$ with the same way as in the Godunov method such that solving the conventional Riemann problem with initial data

$$u_L \equiv \bar{u}_i^{R} \; ; \qquad u_R \equiv \bar{u}_{i+1}^{L} \tag{98}$$

5. Repeat steps 1-4 for each cell
6. Compute u_I^{n+1} for each cell as it is in Godunov method.
7. Repeat steps 1-6 for the next time step.

A.5.5.4 NUMERICAL RESULTS

The same tests applied to previous methods applied to Limited Slope version of MUSCL method, too. The results are very successful, and one may observe that the oscillations are fixed. Computational costs of this version is not as high as WAF, but its resolution is a bit lower. One must note that, this method is one of the most convergent methods among all.

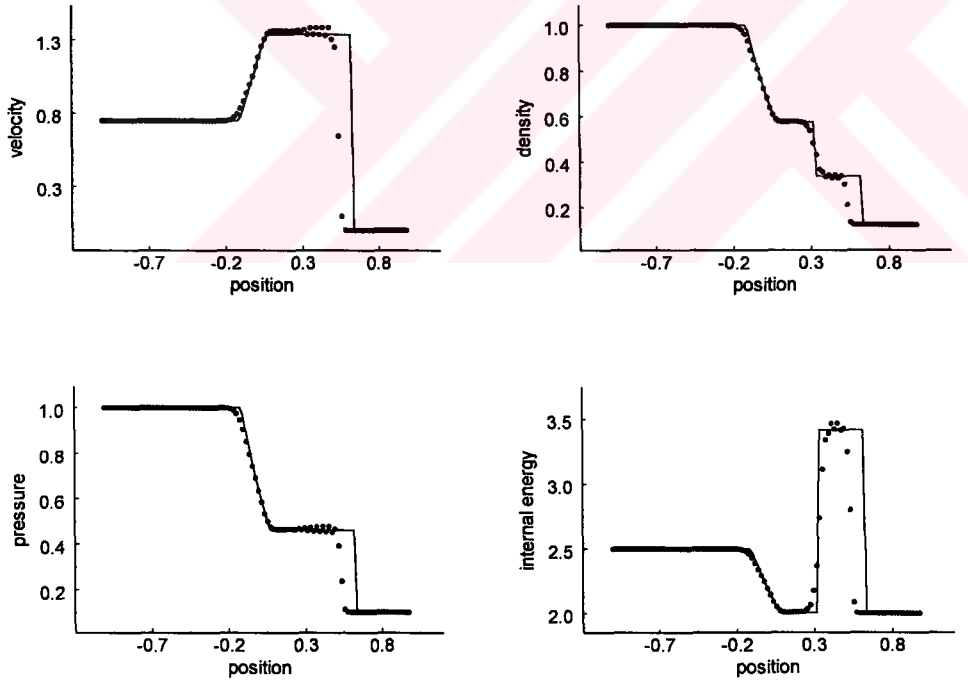

Figure A-24: Test 1 on the TVD version of the MUSCL method using Limited Slopes

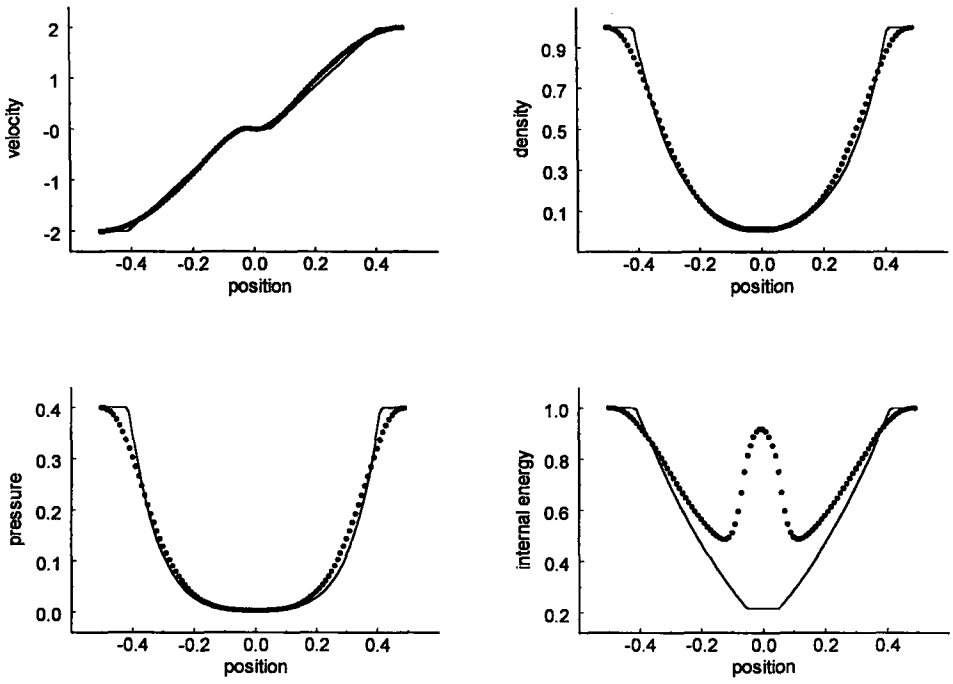

Figure A-25: Test 2 on the TVD version of the MUSCL method using Limited Slopes

test 2 musclsl

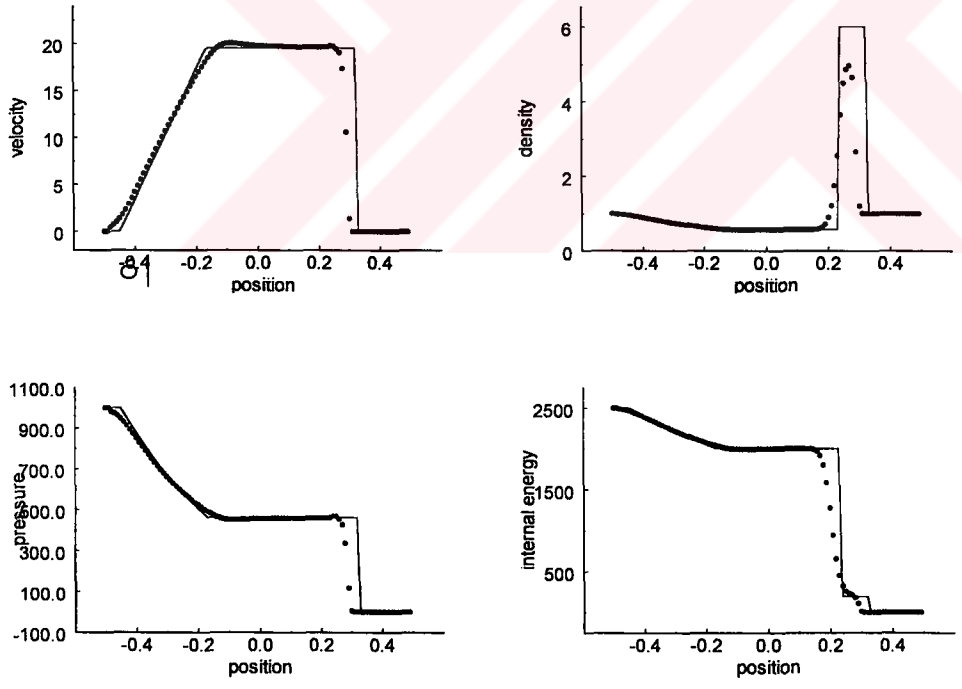

Figure A-26: Test 3 on the TVD version of the MUSCL method using Limited Slopes

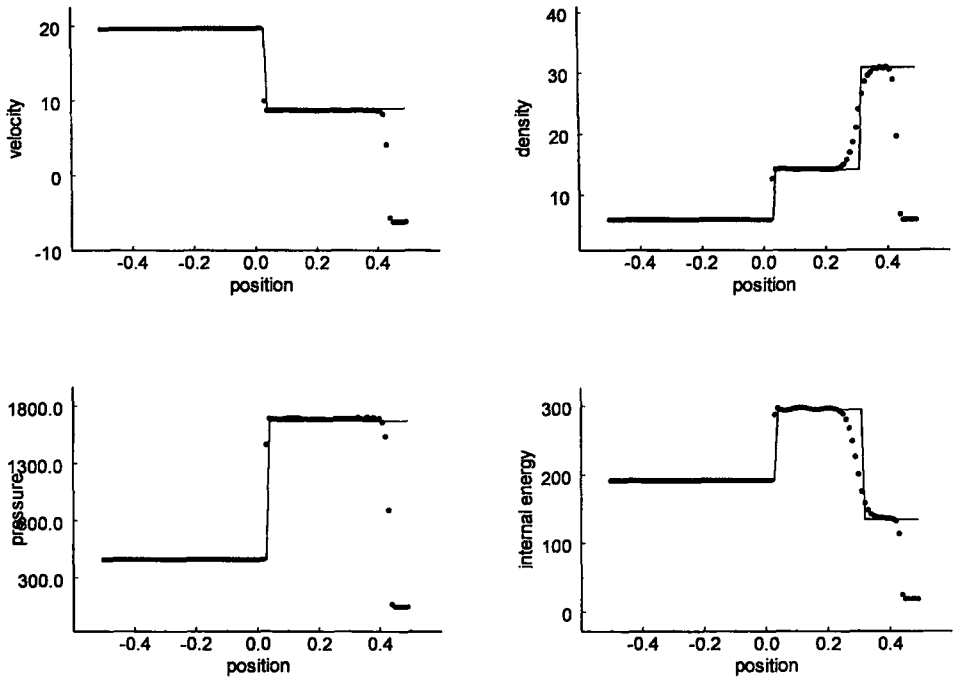

Figure A-27: Test 4 on the TVD version of the MUSCL method using Limited Slopes

A.5.5.5 PROCEDURE FOR MUSCL WITH SLOPE LIMITERS

The main program for the MUSCL method with limited slopes is given in the appendix E in MUSCLSl.cpp file. This file uses HLLC.h header file, containing a HLLC flux class. The procedure is as follows;

1. For the first step, the slope vector Δ_i must be calculated for a cell. For this purpose one can use different choices of variables, such as the primitive values, W=(ρ,u,p) or physical variables U. To achieve a faster and simpler code, extrapolation of physical variables are chosen. For this case the slope vector Δ_i is calculated as:

$$\Delta_i = \frac{1}{2}(1-\omega)\Delta_{i-1/2} + \frac{1}{2}(1+\omega)\Delta_{i+1/2} \qquad (99)$$

where

$$\Delta_{i-1/2} \equiv U_i^n - U_{i-1}^n \; ; \qquad \Delta_{i+1/2} \equiv U_{i+1}^n - U_i^n \qquad (100)$$

and $\omega \in [-1,1]$

2. Limit the slope using the limiters given in the LIMITERS header file which contains the formulas … through ….
3. Estimate the left and right extrapolated values as

$$U_i^L = U_i - \frac{1}{2}\Delta_i \; ; \qquad U_i^R = U_i + \frac{1}{2}\Delta_i \tag{101}$$

4. This is the 2nd step describe the evolved values of U_I^L and U_I^L as;

$$\overline{U}_i^L = U_i^L + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[F(U_i^L) - F(U_i^R)\right] \tag{102.a}$$

$$\overline{U}_i^R = U_i^R + \frac{1}{2}\frac{\Delta t}{\Delta x}\left[F(U_i^L) - F(U_i^R)\right] \tag{103.b}$$

5. Compute the intercell flux $F_{i+1/2}$ with the same way as in the Godunov method such that solving the conventional Riemann problem with initial data

$$u_L \equiv \overline{u}_i^R \; ; \qquad u_R \equiv \overline{u}_{i+1}^L \tag{104}$$

6. Repeat steps 1-4 for each cell
7. Compute u_I^{n+1} for each cell as it is in Godunov method.
8. Repeat steps 1-6 for the next time step.

A.5.5.6 NUMERICAL RESULTS

The same tests applied to previous methods applied to Slope limiter version of MUSCL method. The results are very successful, and one may observe that the oscillations are fixed. Computational costs of this version is similar that of WAF, but its resolution is a bit lower than it. This method gives better resolution than limited slope version, but its convergence is not as good as it.

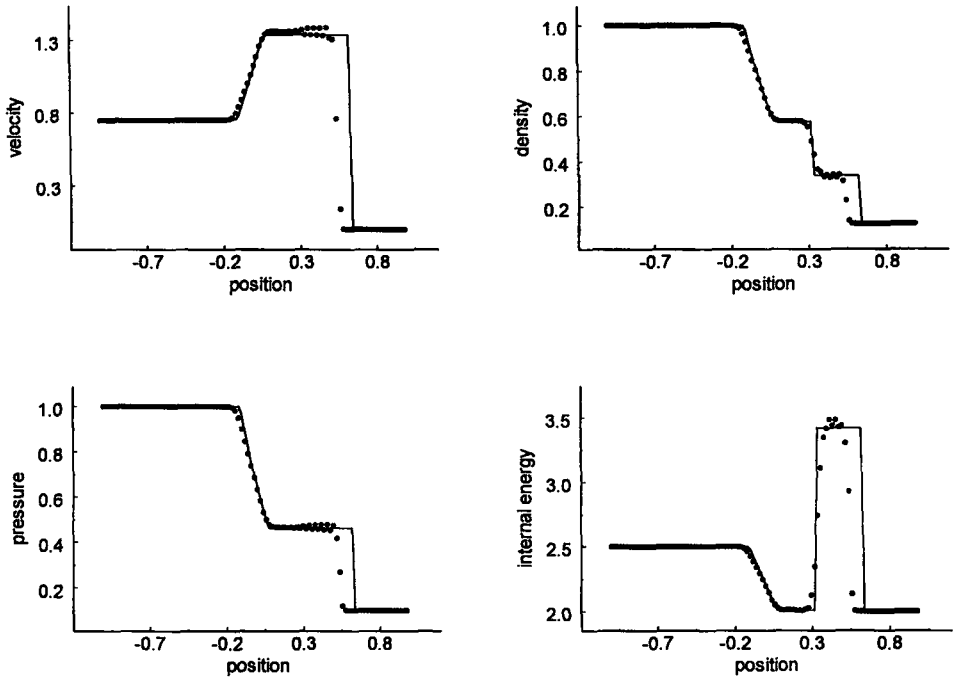

Figure A-28: Test 1 on the TVD version of the MUSCL method using Slope Limiters

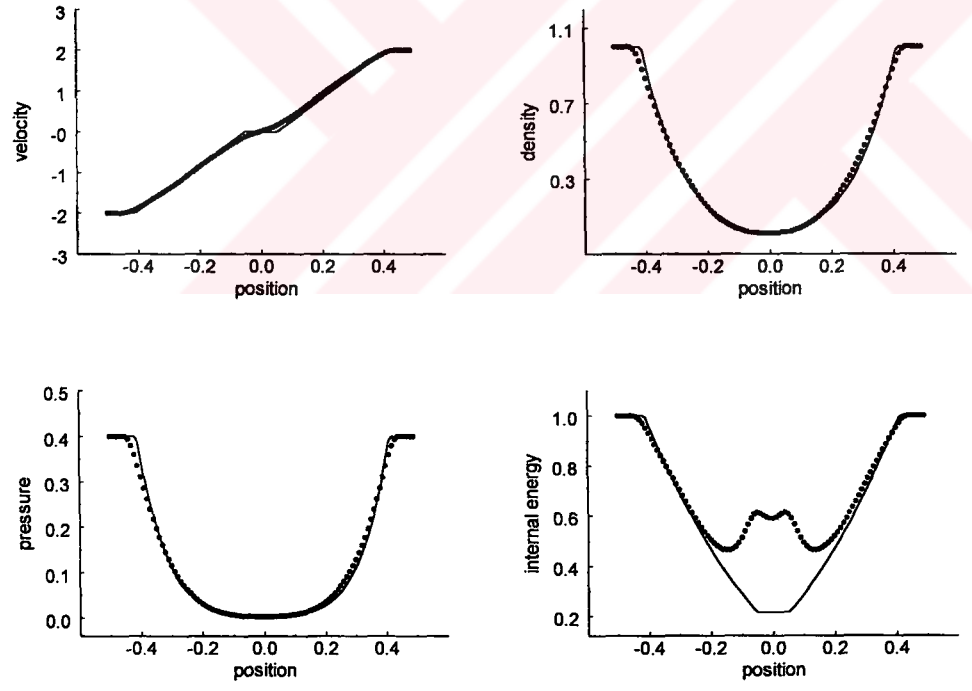

Figure A-29: Test 2 on the TVD version of the MUSCL method using Slope Limiters

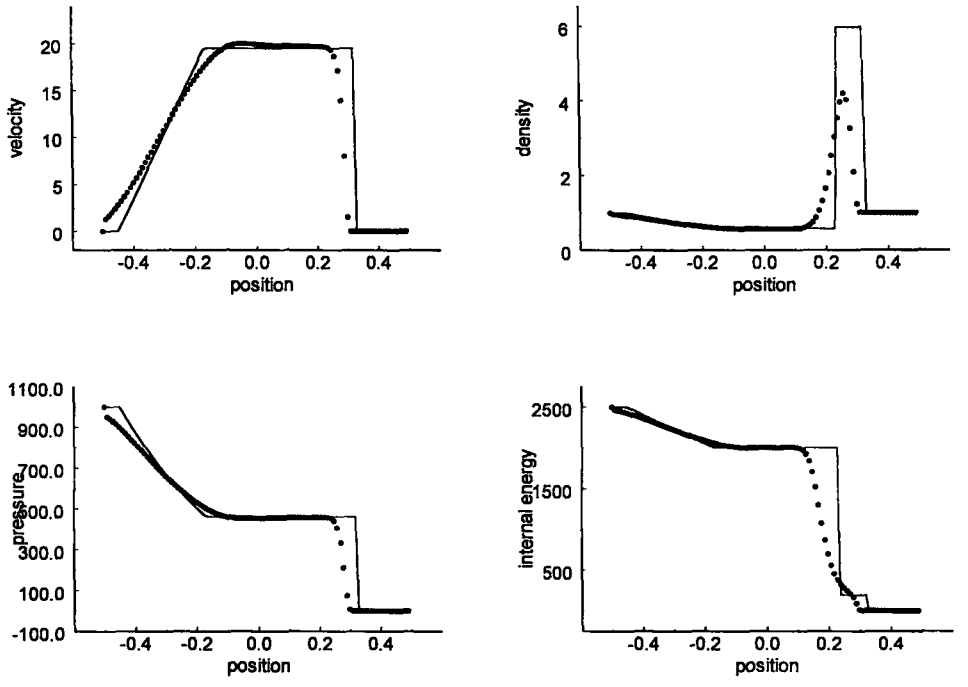

Figure A-30: Test 3 on the TVD version of the MUSCL method using Slope Limiters

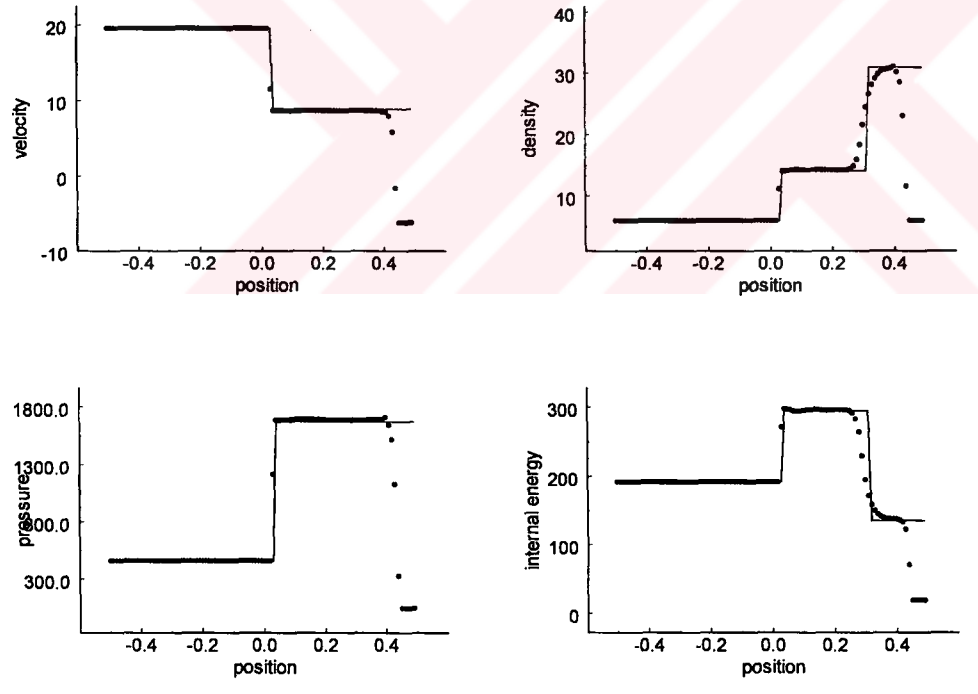

Figure A-31: Test 4 on the TVD version of the MUSCL method using Slope Limiters

APPENDIX B

THE SHOCK TUBE PROBLEM OR RIEMANN PROBLEM FOR EULER EQUATIONS

B.1 THEORY

This particular problem can be realized experimentally by the sudden breakdown of a diaphragm in a long one-dimensional tube separating two initial gas states at different pressures and densities. The initial conditions are as the following:

$$\begin{array}{lllll} u = u_L, & p = p_L, & \rho = \rho_L & x < x_0 & t = 0 \\ u = u_R, & p = p_R, & \rho = \rho_R & x < x_0 & t = 0 \end{array} \qquad (1)$$

with $p_R < p_L$ and the diaphragm is located at $x = x_0$. Two sections assumed to contain the same gas.

If viscous effects are neglected and considered is an infinite tube to avoid reflections at tube ends, the exact solutions to the Euler equations can be obtained as the waves separating regions of uniform conditions. For this case the governing equations reduces to:

$$U = \begin{bmatrix} \rho \\ \rho u \\ E \end{bmatrix}, \qquad F = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{bmatrix} \qquad (2.a)$$

for 1-D case. Also

$$E = \rho\left(\frac{1}{2}u^2 + e\right), \quad e = \frac{p}{(\gamma - 1)\rho} \qquad (2.b)$$

closes the problem for compressible flow.

Sometimes primitive variables rather than conservative ones needed in the solution. These variables are given in a vector form as:

$$W = \begin{bmatrix} \rho \\ u \\ p \end{bmatrix} \qquad (3)$$

In the absence of vacuum the exact solution of the Riemann problem has three waves, which are associated with the eigenvalues $\lambda_1 = u - a$, $\lambda_2 = u$, $\lambda_3 = u + a$. The three waves separate four constant states, which form left to right are: W_L (data on the left hand side), W_{*L}, W_{*R}, and W_R (data on the right hand side).

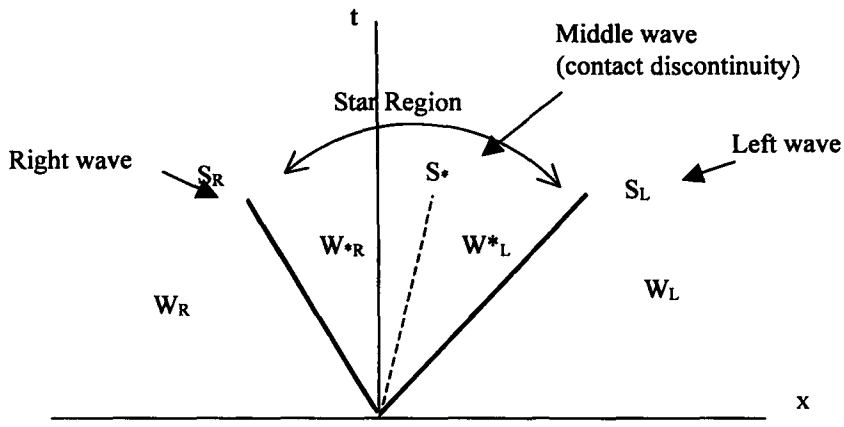

Figure B-1: The wave structure for Riemann problem for Euler equation.

The unknown region between the left and the right waves, the Star Region, is divided by the middle wave into the two subregions Star Left (W_{*L}) and Star Right (W_{*R}). The middle wave is always a contact discontinuity while the left and the right waves are either shock or rarefaction waves. Therefore according to the type of non-linear waves there are four possible wave patterns.

The solution consists of a contact discontinuity separating shocks or rarefaction, depending on the case. The four possible wave patterns are shown in figure below. Case (a) represents a left rarefaction, contact and a right shock. Case (b) is the reverse case: left shock, contact, right rarefaction, case (c) is left rarefaction, contact, right rarefaction (d) left shock, contact, right shock. That is, solution is defined for four different type of domains considering the waves are shock or rarefaction waves and they are left or right going waves. These solutions can be found in many textbooks, and the one given below is taken from Toro [19]. Note that, the solution for left or right going waves are given for the solution of middle wave. The solution for middle wave (p_* , u_*) is constant always.

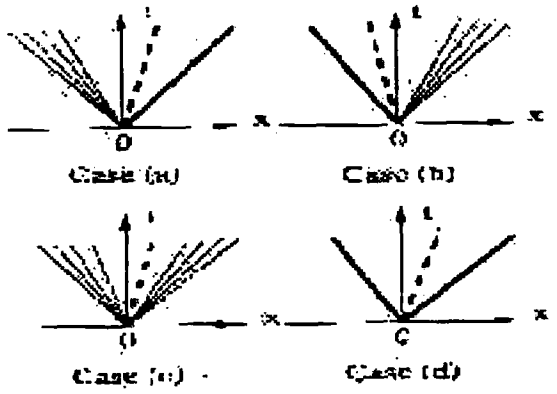

Figure B-2: Possible wave structures of Riemann problem of Gas dynamics

Left Shock: A left shock wave is identified by $p_* > p_L$

Density has been given as:

$$\rho_{*L} = \rho_L \left[\frac{\left(\frac{\gamma-1}{\gamma+1}\right) + \left(\frac{p_*}{p_L}\right)}{\left(\frac{\gamma-1}{\gamma+1}\right)\left(\frac{p_*}{p_L}\right) + 1} \right] \qquad (4)$$

Left Rarefaction Wave: A left rarefaction wave is identified by the condition $p_* \leq p_L$

$$W_{LFAN} \begin{cases} \rho = \rho_L \left[\frac{2}{(\gamma+1)} + \frac{(\gamma-1)}{(\gamma+1)a_L}\left(u_L - \frac{x}{t}\right) \right]^{\frac{2}{\gamma-1}} \\ u = \frac{2}{(\gamma+1)}\left[a_L + \frac{(\gamma-1)}{2}u_L + \frac{x}{t} \right] \\ p = p_L \left[\frac{2}{(\gamma+1)} + \frac{(\gamma-1)}{(\gamma+1)a_L}\left(u_L - \frac{x}{t}\right) \right]^{\frac{2\gamma}{\gamma-1}} \end{cases} \qquad (5)$$

Right Shock: A right wave is identified by the condition $p_* > p_R$

$$\rho_{*R} = \rho_R \left[\frac{\frac{p_*}{p_R} + \frac{\gamma-1}{\gamma+1}}{\frac{\gamma-1}{\gamma+1}\frac{p_*}{p_R} + 1} \right] \qquad (6)$$

Right rarefaction wave: A right rarefaction wave is identified by the condition $p_* \leq p_R$. The solution is given as.

$$W_{RFAN} \begin{cases} \rho = \rho_L \left[\frac{2}{(\gamma+1)} - \frac{(\gamma-1)}{(\gamma+1)a_L}\left(u_R - \frac{x}{t}\right) \right]^{\frac{2}{\gamma-1}}, \\ u = \frac{2}{(\gamma+1)} \left[-a_R + \frac{(\gamma-1)}{2} u_R + \frac{x}{t} \right], \\ p = p_R \left[\frac{2}{(\gamma+1)} - \frac{(\gamma-1)}{(\gamma+1)a_R}\left(u_R - \frac{x}{t}\right) \right]^{\frac{2\gamma}{\gamma-1}}, \end{cases} \qquad (7)$$

Solution for p_*

As it can be observed from the equation above the only unknown parameter is p_*, whose solution will give the complete solution. Defining a pressure function

$$f(p, W_L, W_R) \equiv f_L(p, W_L) + f_R(p, W_R) = 0 \qquad (8)$$

where the function f_L is given by:

$$f_L = \begin{cases} (p - p_L)\left[\frac{A_L}{p + B_L} \right]^{1/2} & \text{if } p > p_L \text{ (\textit{shock})} \\ \frac{2a_L}{(\gamma - 1)} \left[\left(\frac{p}{p_L} \right)^{\frac{\gamma-1}{2\gamma}} - 1 \right] & \text{if } p \leq p_L \text{ (\textit{rarefaction})} \end{cases} \qquad (9)$$

and the function f_R is given by

$$f_R = \begin{cases} (p - p_R)\left[\dfrac{A_R}{p + B_R}\right]^{1/2} & \text{if } p > p_R \text{ (shock)} \\ \dfrac{2a_R}{(\gamma - 1)}\left[\left(\dfrac{p}{p_R}\right)^{\frac{\gamma-1}{2\gamma}} - 1\right] & \text{if } p \le p_R \text{ (rarefaction)} \end{cases} \qquad (10)$$

and the constants, are given by

$$A_L = \frac{2}{(\gamma+1)\rho_L}, \quad B_L = \frac{(\gamma-1)}{(\gamma+1)} p_L \qquad (11.a)$$

$$A_R = \frac{2}{(\gamma+1)\rho_R}, \quad B_R = \frac{(\gamma-1)}{(\gamma+1)} p_R \qquad (11.b)$$

The solution for the particle velocity u_* is

$$u_* = \frac{1}{2}(u_L + u_R) - \frac{1}{2}[f_R(p_*) - f_L(p_*)] \qquad (12)$$

Numerical solution of p*

Now we have a function of p of the form $f(p_*) = 0$. A numerical method is required for the solution. Newton-Raphson method, which is an iterative solution can be emlployed:

$$p_{(k)} = p_{(k-1)} - \frac{f(p_{(k-1)})}{f'(p_{(k-1)})} \qquad (13)$$

where subscript indicates the iteration level. It is important that to have the initial guess for the p_* value be good to increase the convergence. There are some approximations for p_* that can be good starting points. One such approximation is so called *Two Rarefaction* approximations, which is given as:

$$p_{TR} = \left[\frac{a_L + a_R - 1/2(\gamma - 1)(u_R - u_L)}{a_L / p_L^{\frac{\gamma-1}{2}} + a_R / p_R^{\frac{\gamma-1}{2}}} \right] \qquad (14)$$

and results from the exact functions for these two waves given at (12) and (13) are rarefaction waves. If the solution actually consists of two of them, then p_{TR} is exact and no iteration is needed. A second-guess value results from a linearised solution based on primitive variables.

$$\left.\begin{aligned} p_0 &= \max(CHECK, p_{PV}) \\ p_{PV} &= 1/2(p_L + p_R) - 1/8(u_R - u_L)(\rho_R - \rho_L)(a_R - a_L) \end{aligned}\right\} \qquad (15)$$

A third guess value is given by a Two-Shock approximation as:

$$\left.\begin{aligned} p_0 &= \max(CHECK, p_{TS}) \\ p_{TS} &= \frac{g_L(\hat{p})p_L + g_R(\hat{p})p_R + \Delta u}{g_L(\hat{p}) + g_R(\hat{p})} \\ g_K(\hat{p}) &= \left(\frac{A_K}{p + B_K} \right) \end{aligned}\right\} \qquad (16)$$

here, A_K and B_K has been defined previously in (11.a) and (11.b) and $\hat{p}$ is an initial estimate for the solution. For this value $\hat{p} = p_{TR}$ given in equation (18) works well. CHECK is a non-negative value chosen that prevents negative pressures that may be produced by formulas.

As a fourth guess, the arithmetic mean of the left and right data is chosen as

$$p_0 = \frac{1}{2}(p_L + p_R) \qquad (18)$$

There are some works to compare the success of these estimates. A hybrid method is used based on Toro's work is used in this code.

B.2 TEST CASES

To test the Riemann solver, the code has been run for several test cases, solution of which is available at popular text books. First test case is taken from Hircsh [24]. The solutions of the code perfectly fits the solutions of the Hirsch's. In this test case, following initial conditions are used:

$$p_L = 10^5 \, , \, \rho_L = 1.0 \, , \, p_R = 10^4 \, , \, \rho_R = 0.125 \, , \, u_L = u_R = 0.0$$

and the solution is presented below for t=6.1msec.

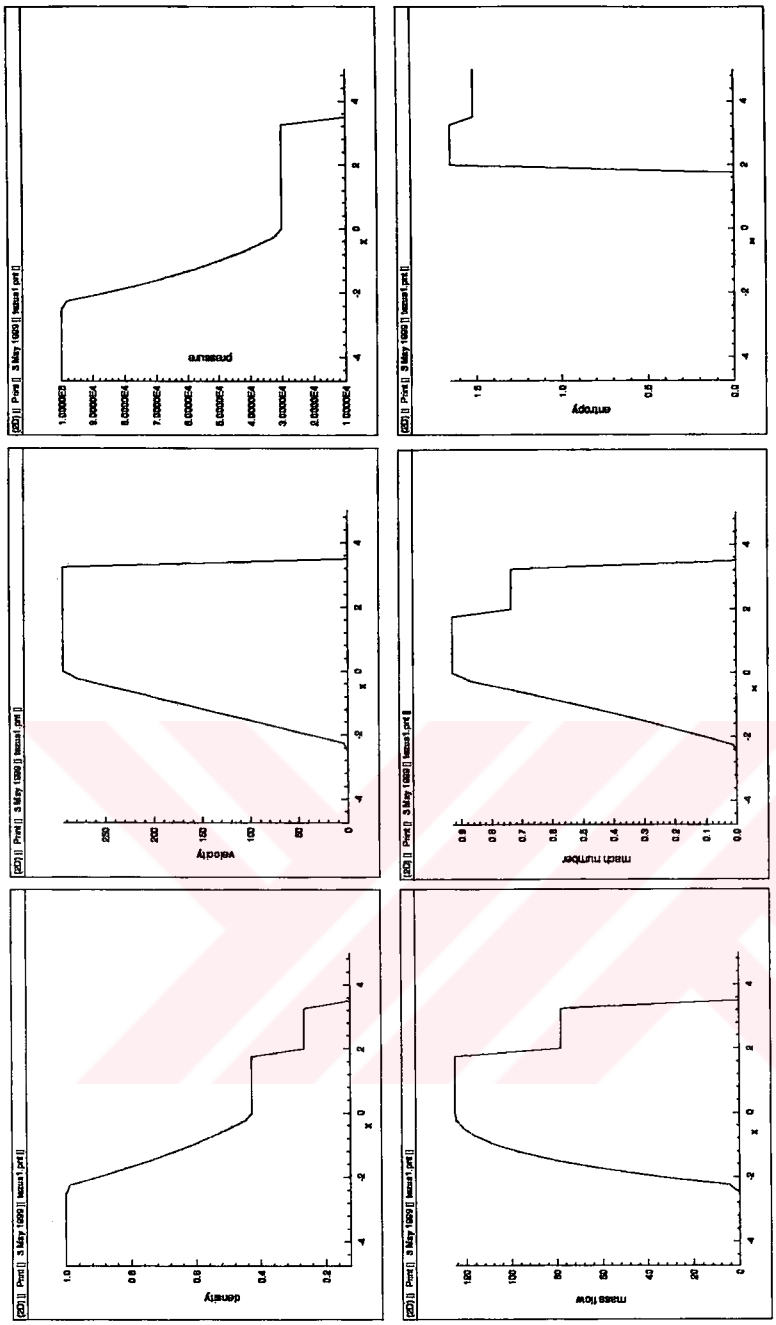

Figure B-3: Test case taken from Hirsch

The second test case is made in order to test the performance of the code for the reverse case. So the same data is considered, but this time phases are switched.

$$p_L = 10^4 \,,\ \rho_L = 0.125 \,,\ p_R = 10^5 \,,\ \rho_R = 1.0 \,,\ u_L = u_R = 0.0$$

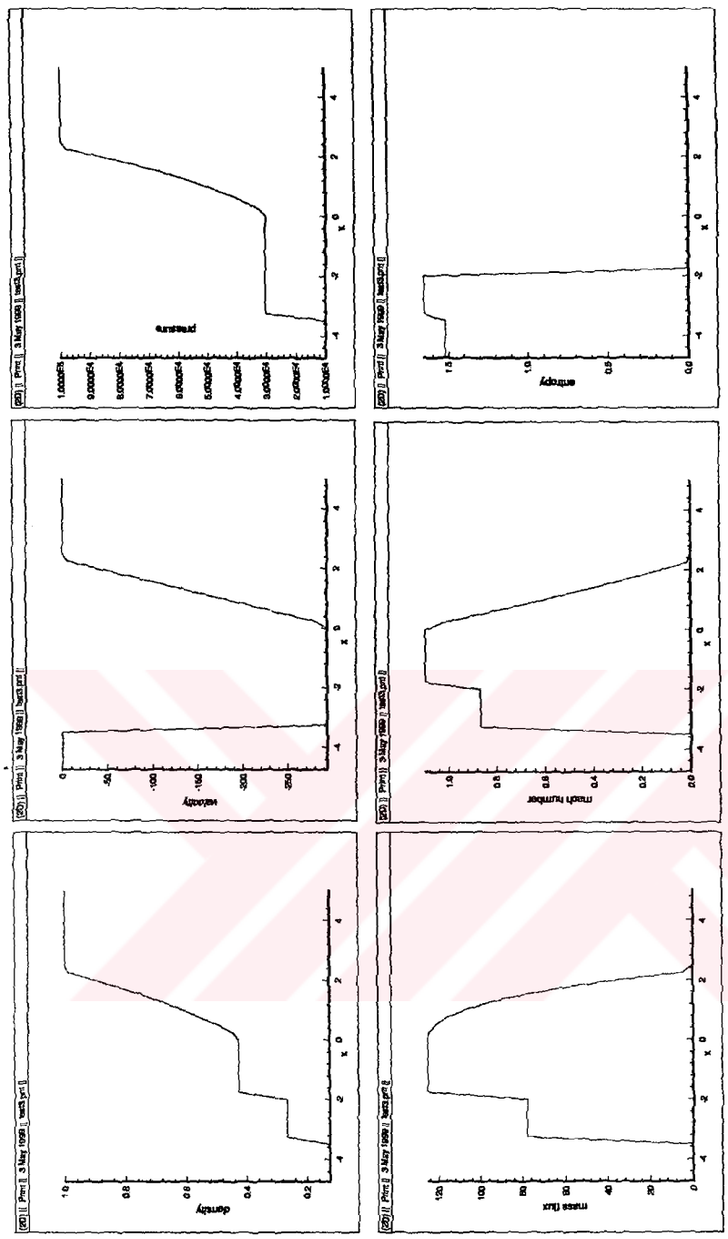

Figure B-4: Reverse test case taken from Hirsch

As is seen from the figure B-3, the results are the same, but this time in reverse order. This verifies the process of solution..

The third case consists of two strong rarefactions. Initial conditions and the solution is given below, adopted from Toro [19]:

$$p_L = 1.0\,,\ \rho_L = 1.0\,,\ p_R = 1.0\,,\ \rho_R = 1.0\,,\ u_L = -2.0\,,\ u_R = 2.0$$

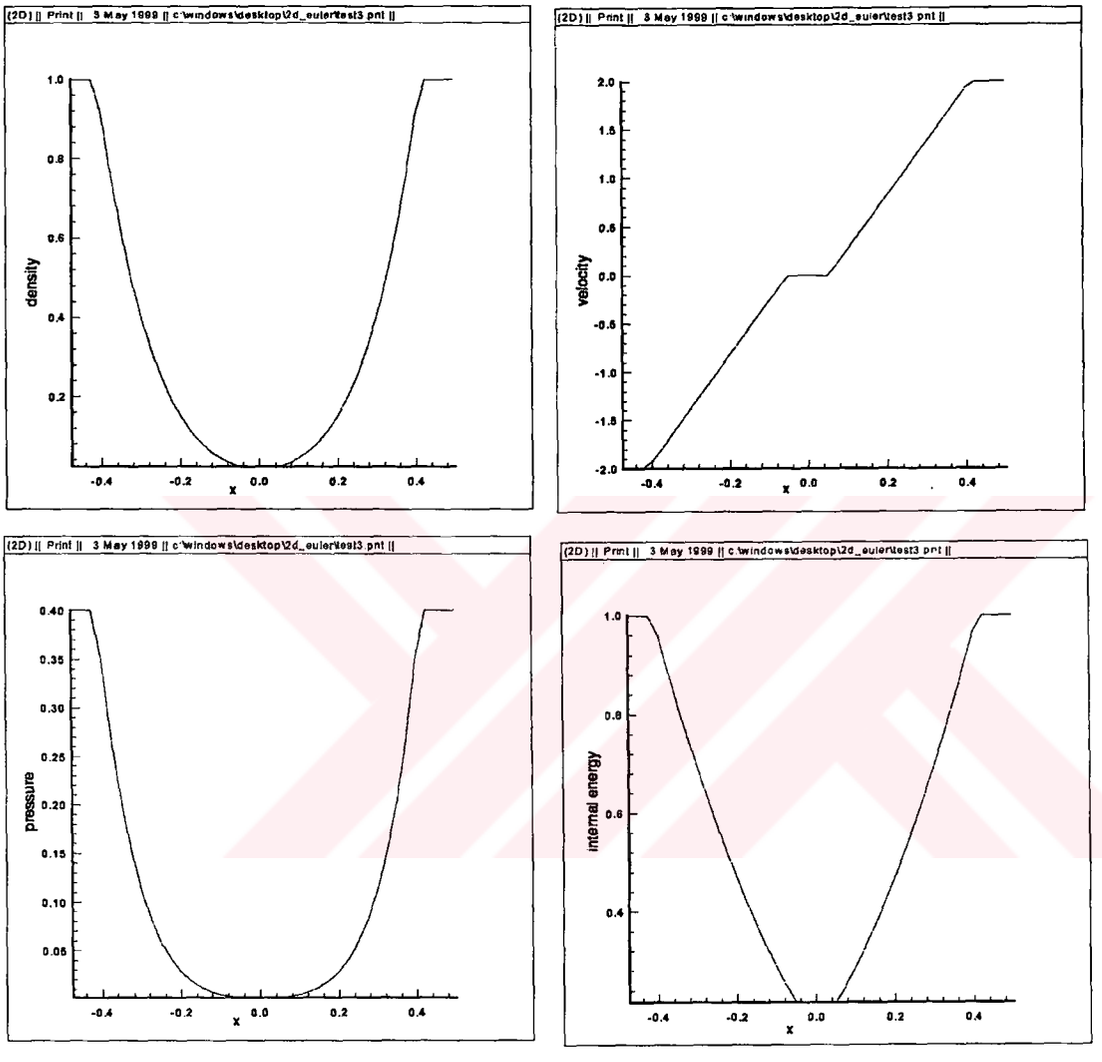

Figure B-5: Test case taken from Toro

APPENDIX C

THE ELECTRONICS OF THE DATA ACQUSITION SYSTEM

C.1 PRESSURE TRANSDUCERS

For the experiments, a PCB Piezotronics series 111A20 transducers is used. This is a piezoelectric transducer with high service pressure with a low sensitivity. The working conditions for the transducer is given in table below.

Table C-1 : Properites of the Transducer

DYNAMIC PERFORMANCE		
Dynamic range (for ±5V output)	Psi [kPa]	500 [3448]
Useful Overrange (for ±10 V output)	Psi [kPa]	1000 [6895]
Maximum Static Pressure	Psi [kPa]	5000 [34475]
Resolution	Psi [kPa]	0.01 [0.07]
Resonant frequency	KHz	> 400
Linearity	% FS	< 2.0
ENVIRONMENTAL		
Acceleration sensitivity	Psi/g [kPa/m/s^2]	< 0.002 [<0.0014]
Operating temperature range	[°C]	[-73 to +135]
Temperature coeff of sensitivity	[%/°C]	<0.03 {<0.054]
ELECTRICAL		
Sensitivity	mV/psi [mV/kPa]	10±1.0 [1.45±0.145]

Excitation voltage required	+V DC	20 to 30
Output impedance	Ohms	< 100
Discharge time constant	Sec	>50

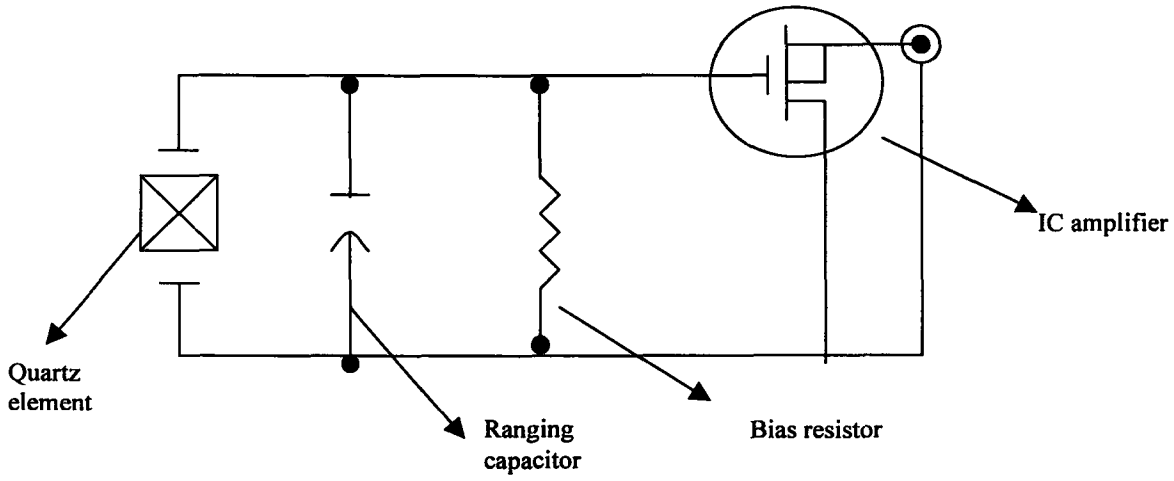

Figure C-1 : Schematic Diagram of the transducer

C.1.1 INSTALLATION OF THE TRANSDUCER

The transducers used in the experiments are flush mounted types. They are 0.218" in diameter on the diaphragm side, with a seal shoulder above them.

There are some requirements for the installation of the transducers. The surfaces they are installed on must be smooth and free from chatter marks, burrs and other irregularities, which could preclude a pressure-tight seal. Seals must always be used in order to prevent leakage and, hence, undesirable pressure changes.

C.1.2 OPERATION

It is only necessary to supply the transducer with a 2 to 20mA constant current at +18 to +24VDC through a current regulating diode or equivalent circuit. The

manufacturer recommends a proper amplifier for best operation. The recommendation of the manufacturer for the amplifier is as follows:

- Switch power on and observe reading of bias monitoring voltmeter on front panel of power unit.
- If indicator is in green section of indicator panel, the IC amplifier is producing proper bias (+11VDC), the cable connections are normal and the system is ready to operate.
- If the pointer moves into the red area of the fault monitor meter, output is zero and a short is indicated. Short could be located in amplifier cable connections or power unit.
- If pointer moves into the yellow area of the fault monitor meter, an open circuit is indicated with full power supply voltage. An open circuit could be the result of a faulty amplifier, an open cable or open connectors.
- Allow the transducer to thermally stabilise for about one minute. A signal drift may occur when a non-standard cable is connected to the readout instrument. This drift occurs during charging of the coupling capacitor in the power unit. The signal will stabilize in several minutes. After stabilisation proceed with measurements.
- Depending upon the transducer's built-in discharge time constant, repetitive output signals slowly or rapidly move toward a stable condition where the average voltage level corresponds to the zero signal position.

- In this position, the area above zero equals the area below and any references to absolute or static levels is meaningless. Such output signal behaviour is typical of an AC coupled system.

C.1.3 POLARITY

They always give positive-going output voltage for increasing pressure input.

C.1.4 CALIBRATION

Piezoelectric transducers are dynamic devices but static calibration techniques can be employed if discharge time constants are sufficiently long. Generally, static methods are not employed below several hundred seconds time constant.

For the present case, because the transducers are brand new no calibration is needed. The calibration charts obtained from the vendor are attached in the appendix D.

C.2 POWER UNIT AND AMPLIFIER

The power units used, the models 480E09 are in compliance with the specification of the PCB. They are battery powered units with gain. The voltage switch offers amplification factors of 1, 10, 100.

The amplifier operates from three self-contained 9-volt batteries and supply constant-current power to the built-in transducer amplifier in ICP transducers. The schematic diagram of the power units is shown below:

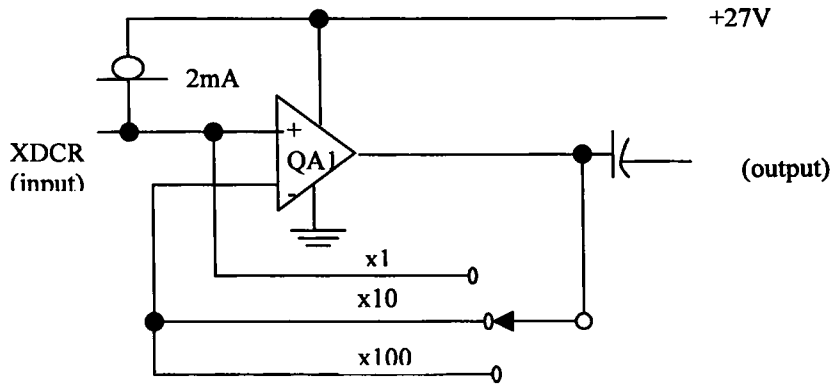

Figure C-2: Schematic diagram of the amplifier

A transducer ("XDCR") jack is located at the front panel as well as a signal output jack labelled "SCOPE". These jacks are female BNC jacks.

The units are very small and light, that is easily carried in the field and, being battery operated, are especially noise-free and unaffected by ground loops. A notable feature of these units is their low-frequency response.

C.2.1 OPERATION OF THE AMPLIFIER

- With no transducer connected them, move the rotary switch to the gain position. The front panel voltmeter will read the battery voltage. The voltmeter is scaled to read 27 volts full scale without the transducer in the system.
- When an ICP transducer is connected to the input transducer jack, the meter will indicate approximately mid-scale (+11V nominal) if the transducer's built-in amplifier is functioning properly and cables are intact.

- Immediately after connecting readout instrument to the "output" jack, the capacitor will begin to charge through the input resistance of the readout instrument. This charging will cause an apparent drifting of the output signal until the capacitor is fully charged.

C.2.2 OUTPUT VOLTAGE LIMITATIONS

The transducers used are capable of 10-volt output voltage signal. The power supplies allow the positive-going side of the signal to go to ±14 volts. The negative-going side of the signal is capable of supply –8 volts.

C.2.3 CURRENT DRIVE LIMITATIONS

The current output of the power units is fixed at 2mA. This current will adequately handle high frequency signals in cables up to 100ft (30m) long.

C.3 DATA ACQUISITION CARD

In order to collect data, a data acquisition card is used to get voltage output of transducer-amplifier system. Data card used is PCL-812 from ADVANTECH, that is a high speed, high performance, multi-function DAS card for IBM PC\XT\AT compatible computers.

C.3.1 KEY FEATURES

- 20 single ended analog input channels.

- 12-bit successive approximation converter to convert analogue inputs.
- Maximum A/D sampling rate is 30kHz in DMA mode.
- Different analogue input ranges. ±1V, ±2V, ±5V, ±10V
- Three trigger modes: programmable pacer trigger, software trigger, and external pulse trigger.
- Programmable pacer trigger with 0.5MHz to 35 minutes.

C.3.2 EXPANSION BOARD

For the expansion of the card, ie connection to multiple data, a simple PCLD-780 wiring terminal board for simple analog and/or digital I/O connections, is connected. PCLD-780 has no multiplexing capability, it is just a wiring board with 20 ends.

C.3.3 CARD OPERATION CONDITIONS

Like every computer cards, this data acquisition card provides correct jumper settings. Those settings and the values set for them are:

- Base address selection: this is the DOS and Win3.x device base address. This address is set to 22F at switch 1.
- Bipolar input range selection: this selection stands for the readable input range. This value is set to ±5V, that is the useable voltage of the transducers by SW2.
- IRQ level selection: this level is chosen as 2 by JP4, which is not conflicting the devices on the PC used.

- Other settings are not related with the case used in these experiments, like D/A conversion, DMA conversion etc.

Those drivers must be loaded before running the process:

- For DOS environment, pcl-812.exe device driver

C.3.4 SOFTWARE

The available software was ineffective and therefore a new software was developed. For this purpose, some library routines for the card are downloaded from the Internet site of the manufacturer. Two programs are developed for the general use of the laboratory and the specific needs of the experiment by C++ programming language. Detailed description will be given below. Also source code of the programs are presented in appendix G. The programs uses PCL-812.EXE DOS device driver, so that this driver must be loaded before proceeding.

Two programs are written for data acquisition. First program is created for real-time control of the input channels, while second one performs a full data acquisition task.

C.3.4.1 INPUT CHANNEL CONTROL PROGRAM

The most time consuming part of constructing a data collecting system may be the very initial steps, like connecting sensors to amplifiers, amplifiers to board, board to DA card, etc. Cabling process sometimes gets very confusing, and one sometimes cannot make sure if the cabling is correct. The diagnostic part sometimes becomes very difficult and time consuming. So that, a handy tool is strongly needed for this fault reduction step.

In order to achieve this goal, a simple but very useful program, GRP.EXE is developed. The source code of this program can be found in appendix G as GRP.CPP. Program execution is very simple, it only takes data from all of the channels in one milliseconds in every 0.3 seconds and displays them in twenty different boxes, simultaneously. These real-time channel readings can be interpreted such

- If the channel reading is 2048, then the reading is 0V and there is a short in the input channel. Remember, the empty channels most be shorted. If this channel must not be empty, then check cabling.
- If the channel reads 4096 at the beginning and reaches 2048 with high fluctuations, then, the channel is empty and not shorted. This channel must be shorted for a correct acquisition.
- If the channel reads 4096 or 0, then high voltage exists at the ends. For this case, either decrease the gain of the amplifier or increase the range of the card.
- If the channel number is not as expected, and ten channels before or after that, then, the connection cables between terminal board and DA card are in reverse order.

C.3.4.2 DATA ACQUISITION PROGRAM

In order to be used for general data collecting purposes and also to sustain the specific needs of the set-up a program called DAS.EXE is developed. The program, the source code of which is given at the appendices G as DAS.CPP, is written in C++ language. The program performs three main tasks:

C.3.4.2.1 Interactive Input Screen

This screen completes following tasks:

1. Input of the acquisition parameters: Start Channel, Stop Channel, Sampling Rate in kHz and number of data to be collected.
2. Input of the Output File name.
3. Checking the data if:
    - The numbers are given in appropriate form, in other words they are integers.
    - Start and stop channel data is in the range of 0-19, since there are 20 channels
    - Stop channel data is larger than start channel data
    - Sampling rate is less than 30000 Hz.
    - Number of data is less than 64K, since data buffer of the card is limited with this number.
    - Output filename is correct and file can be opened successively.

If one of these parameters are incorrect, then the program raises an exception, and displays an error message in an error box, then returns to the input boxes. If all those parameters are correct then, the first part closes and data acquisition stage begins.

C.3.4.2.2 Data Acquisition Stage

This is a specific stage that controls the PCL-812 data acquisition card. At this stage all the data entered in the previous stage are used. This accomplishes the following tasks:

1. Prompts the user if some jumper settings are correct for the card. If these settings are different from the desired ones used by the program, then it aborts the program and asks user to change the settings.
2. If settings are correct it makes following initializations
   - Card initialization
   - A/D conversion initialization
   - Pacer trigger initialization
3. If any of these initializations are failed, then it aborts. Else, program waits the user three seconds to run to the system and get ready to operation. Then a beep sound tells the operator the data acquisition session has began, so experimenter can run his system.
4. Collects data from the transducer.
5. After completing the task, writes data to the disk to the output file specified by user. The output is converted in voltages before it is written to the disk.

C.3.4.2.3 Graphical Display Stage

At this stage, the results are displayed in rectangular coordinates using the following visualization procedure.

1. X axis is time axis. The maximum value of the axis is the number of data. It contains major and minor thick marks and also major thick numbers.
2. Y-axis is output axis. The output is not taken as voltage, but 12-bit integer output of the card. The maximum value of the axis is 4095,

which corresponds 2^{12}-1, that designates +5.0V. In this manner, 0 reading corresponds –5.0V and 2094 corresponds to zero. It contains major and minor thick marks and also major thick numbers also. The main reason of using raw data on the scale, in other words not using real time and voltage values, is just not to loose too many times while recalculating 64 thousand data.

3. Program draws the output. Doing this it uses a different color for each channel output.
4. Prompts user to press any key. If a key is pressed, then it closes graphical VGA display environment, back to the DOS type character environment.

C.4 PROCEDURE

Before performing experiments, the system is run for several times with no electronic part mounted on it. This performed is to clean up the system in order to protect the transducers and to avoid filling up of the gate of the sensor casing.

Following experimental procedure is followed during the tests:

- Connect transducers to the amplifiers. Turn on the amplifier. Observe correct voltage on the amplifiers' screen. Wait one or two minutes that the capacitor of the transducer fills.
- Fill the pipe with water. Check the mass of water from the scale on the pipe. Do not forget to close the fill and drain valves. If those valves left open, they may cause injuries.
- Close all the valves on the air tank in order to fill it

- Close the valve between the compressor and air tank. Run the compressor.
- After pressure at the compressor raises to a treshold value open the valve and let the tank fill with pressurized air. Do not stop the compressor.
- While compressor is filling the tank, connect the amplifiers to the computer, while the computer is on and all the device drivers are loaded.
- Wait for at least five minutes, that the transducer adjusts itself for the impedance of the DA card.
- If necessary, make a sample run of the program to check if the steady state conditions of the amplifiers is reached.
- When the tank reaches to desired pressure, close the valve between compressor and the tank.
- Wait at least one minute, until the transducers give zero output for steady condition.
- Stop the compressor.
- Give the acquisition parameters to the computer.
- Begin to run the program, and go immediately behind the big valve. You have got three seconds to go and get ready. When the sound signal is heard from the computer, open the valve "VERY FAST"
- Check your results from the screen.
- Make your system ready for the next run.

APPENDIX D

CALIBRATION CHART FOR THE TRANSDUCER

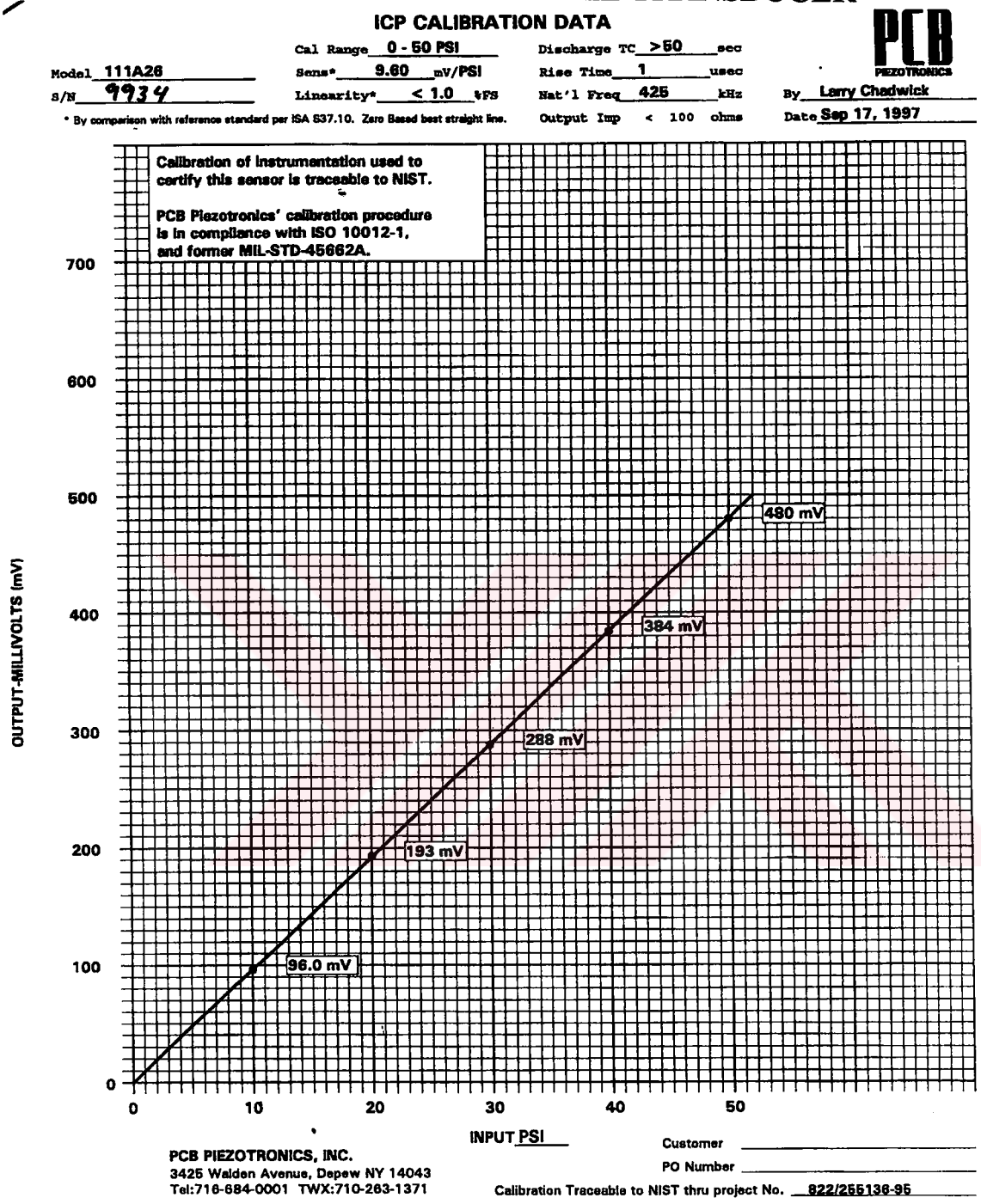

Figure D-1: Calibration chart for the transducer

APPENDIX E

SHOCKTUBE SIMULATION CODES

E.2 shcktube.cpp

```
////////////////////////////////////////////////////////////////
//                    shcktube.cpp                            //
//       this program gives the exact solution of             //
//       shock tube problem, while you compiling              //
//       make sure that Riemann.h and Vector.h                //
//       header files and also a data file is in              //
//       appropriate folder.                                  //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>
#include "Riemann.h"

Riemann shcktube;
Vector u(3), w(3);
FILE* inputfile, *outfile;
int nstep;
double dt,xl,xr,dx;

void main(void)
{
```

```
inputfile=fopen("shcktube.dat","r");
outfile=fopen("shcktube.out","w");
fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &shcktube.ul,
&shcktube.pl, &shcktube.rl,

              &shcktube.ur, &shcktube.pr, &shcktube.rr,
                                          &nstep, &dt);
fscanf(inputfile,"%lf%lf",&xl, &xr);
printf("%lf %lf %lf %lf %lf %lf %i %lf", shcktube.ul, shcktube.pl,
shcktube.rl,

              shcktube.ur, shcktube.pr, shcktube.rr,
                                          nstep, dt);
shcktube.Solvexy();
dx=(xr-xl)/nstep;
for(int i = 0; i<nstep; ++i){
      shcktube.S=(xl+i*dx)/dt;
   shcktube.soln();

fprintf(outfile,"%lf\t%lf\t%lf\t%lf\t%lf\n",(xl+i*dx),shcktube.u,shc
ktube.r,shcktube.p,(shcktube.p/0.4/shcktube.r));

}

}
```

E.3 Riemann.h

```
////////////////////////////////////////////////////////////////
//                      riemann.h                             //
//       this is the header file used in many godunov         //
//       based solvers. this file contains a c++ class        //
//       that is used to solve EXACT Riemann problem of       //
//       Euler problem. make sure that Vector.h header        //
//       file is in an appropriate folder in compilation      //
////////////////////////////////////////////////////////////////

#include <math.h>
#include "Vector.h"
#define GAMMA 1.4
#define QMIN 2.0
#define ITER 20
#define TOL 1.0E-6
#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))


class Riemann{
public:
Vector flux;
double pl,rl,ul,cl,pr,rr,ur,cr;
double ps,us,S,u,r,p,c;
double gm1b2,rgp1b2,gm1b2g,gp1b2g,g2bgm1,rgm1b2,gm1bgp1,g2bgp1;
      Riemann(){gm1b2=0.5*(GAMMA-1.0);
            S=0.;
            rgp1b2=2.0/(GAMMA+1.0);
            g2bgp1=2*GAMMA/(GAMMA+1.0);
                gm1b2g=0.5*(GAMMA-1.0)/GAMMA;
                gp1b2g=0.5*(GAMMA+1.0)/GAMMA;
            g2bgm1=1.0/gm1b2g;
            rgm1b2=1.0/gm1b2;
            gm1bgp1=(GAMMA-1.0)/(GAMMA+1.0);}
   void Solvexy();
   double initP();
      void press(double &,double &, double &, double &, double &);
   void soln();
//private declarations
private:


};
```

```
void Riemann::Solvexy()
{
      //sound speeds
   cl=sqrt(GAMMA*pl/rl);  //left state
   cr=sqrt(GAMMA*pr/rr);  //right state
   ps=initP();
   int iter=0;
   double fl,fr,dfl,dfr,pold;
   do{
      pold = ps;
      press(fl,dfl,rl,pl,cl);
      press(fr,dfr,rr,pr,cr);
      ps-=(fl+fr+ur-ul)/(dfl+dfr);
      if(ps<0.)ps=TOL;
      ++iter;
   }while((iter < ITER)&&(abs(2*(ps-pold)/ps)>TOL));
   us=0.5*(ul+ur+fr-fl);
   soln();
   flux(0)=r*u;flux(1)=r*u*u+p;flux(2)=u*(r*u*u*0.5+p*2.5+p);
}

void Riemann::press(double &f, double &df, double &dk, double &pk,
double &ck)
{
       if(ps <= pk){
      double prat=ps/pk;
      f=rgm1b2*ck*(pow(prat,gm1b2g)-1.);
      df=(1./(dk*ck))*pow(prat,-gp1b2g);
    }else{
      double a=rgp1b2/dk, b=gm1bgp1*pk;
      double qrt=sqrt(a/(b+ps));
      f=(ps-pk)*qrt;
      df=(1.-0.5*(ps-pk)/(b+ps))*qrt;
    }
}

void Riemann::soln()
{
      if(S < us){
      if(ps <= pl){
            if(S <= (ul-cl)){
            r=rl; p=pl; u=ul;}else{
                  if(S > (us-cl*pow((ps/pl),gm1b2g))){
                  r=rl*pow((ps/pl),(1.0/GAMMA));      p=ps;
      u=us;}else{
                  u=rgp1b2*(cl+gm1b2*ul+S);
      c=rgp1b2*(cl+gm1b2*(ul-S));
                  r=rl*pow((c/cl),rgm1b2);
      p=pl*pow((c/cl),g2bgm1);
               }
         }
      }else{
            if(S <= (ul-cl*sqrt(gp1b2g*ps/pl+gm1b2g))){
            r=rl; p=pl; u=ul;}else{
            r=(ps/pl);  r=rl*((r+gm1bgp1)/(r*gm1bgp1+1.0)); p=ps;
      u=us;
         }
```

```
      }
   }else{
      if(ps <= pr){
            if(S >= (ur+cr)){
            r=rr; p=pr; u=ur;}else{
                  if(S < (us+cr*pow((ps/pr),gm1b2g))){
                  r=rr*pow((ps/pr),(1.0/GAMMA));      p=ps;
      u=us;}else{
                  u=rgp1b2*(-cr+gm1b2*ur+S);    c=rgp1b2*(cr-
gm1b2*(ur-S));
                  r=rr*pow((c/cr),rgm1b2);
      p=pr*pow((c/cr),g2bgm1);
                }
         }
      }else{
            if(S >= (ur+cr*sqrt(g2bgp1*ps/pr+gm1b2g))){
            r=rr; p=pr; u=ur;}else{
            r=(ps/pr);  r=rr*((r+gm1bgp1)/(r*gm1bgp1+1.0)); p=ps;
      u=us;
         }
      }
   }
}

double Riemann::initP()
{
double pv=0.5*(pl+pr)-0.125*(ur-ul)*(rl+rr)*(cl+cr);
double pmin=min(pl,pr),pmax=max(pl,pr);
double qrat=max(pl,pr)/min(pl,pr);
if (qrat<=QMIN && ((pv<=pmax)&&(pv>=pmin))){
      return max(TOL,pv);
}else{
      if (pv<=pmin){
            return pow((cl+cr-gm1b2*(ur-
ul))/(cl/pow(pl,gm1b2g)+cr/pow(pr,gm1b2g)),g2bgm1);
   }else{
         double
gel=sqrt((rgm1b2/rl)/(gm1bgp1*pl+max(TOL,pv))),ger=sqrt((rgm1b2/rr)/
(gm1bgp1*pr+max(TOL,pv)));
      return max(TOL, (gel*pl+ger*pr-(ur-ul))/(gel+ger));
   }
}
}
```

E.4 HLLC.h

```
////////////////////////////////////////////////////////////////
//                      HLLC.h                                //
//       this is the header file used in many godunov         //
//       based solvers. this file contains a c++ class        //
//       that is used to solve APPROXIMATE Riemann problem of //
//       Euler problem using Harten-Lax and vanLeer's solver. //
//       make sure that Vector.h header                       //
//       file is in an appropriate folder in compilation      //
////////////////////////////////////////////////////////////////


#include <math.h>
#include "Vector.h"
#define GAMMA 1.4
#define QMIN 2.0
#define TOL 1.e-6
#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))

class HLLC {
double ps,us,S,u,r,p,c;
double gm1b2g,gp1b2g,gm1b2,g2bgm1,rgm1b2,gm1bgp1;

public:
double pl,rl,ul,cl,pr,rr,ur,cr;
Vector flux,Us,Ul,Ur;
      void pstar(void);
   void Solvexy();
   HLLC(){
            gm1b2g=0.5*(GAMMA-1)/GAMMA;
      g2bgm1=1./gm1b2g;
      gp1b2g=0.5*(GAMMA+1)/GAMMA;
            gm1b2=0.5*(GAMMA-1);
            rgm1b2=1./gm1b2;
            gm1bgp1=(GAMMA-1)/(GAMMA+1);
   }
};

void HLLC::Solvexy()
{
   pstar();
   double Ss;
      double ql,qr,Sl,Sr;
   if(pl <= ps)ql=sqrt(1.+gp1b2g*(ps/pl-1.));
   else ql=1.;
   Sl=ul-cl*ql;
   if(pr <= ps)qr=sqrt(1.+gp1b2g*(ps/pr-1.));
   else qr=1.;
   Sr=ur+cr*qr;
   Ss=(pr-pl+rl*ul*(Sl-ul)-rr*ur*(Sr-ur))/(rl*(Sl-ul)-rr*(Sr-ur));
```

```
   if(Sl>=0.){

      flux(0)=rl*ul;flux(1)=rl*ul*ul+pl;flux(2)=ul*(rl*ul*ul*0.5+pl*
2.5+pl);
      }else{
      if(Ss >= 0.){

      flux(0)=rl*ul;flux(1)=rl*ul*ul+pl;flux(2)=ul*(rl*ul*ul*0.5+pl*
2.5+pl);
         Us(0)=1.;Us(1)=us; Us(2)=(0.5*ul*ul+2.5*pl/rl)+(Ss-
ul)*(Ss+pl/(rl*(Sl-ul)));
         Us=(rl*(Sl-ul)/(Sl-Ss))*Us;
         flux+=Sl*(Us-Ul);
      }
   }
   if(Sr<=0){

      flux(0)=rr*ur;flux(1)=rr*ur*ur+pr;flux(2)=ur*(rr*ur*ur*0.5+pr*
2.5+pr);
      }else{
      if(Ss <= 0.){

      flux(0)=rr*ur;flux(1)=rr*ur*ur+pr;flux(2)=ur*(rr*ur*ur*0.5+pr*
2.5+pr);
         Us(0)=1.;Us(1)=us;Us(2)=(0.5*ur*ur+2.5*pr/rr)+(Ss-
ur)*(Ss+pr/(rr*(Sr-ur)));
         Us=(rr*(Sr-ur)/(Sr-Ss))*Us;
         flux+=Sr*(Us-Ur);
      }
   }
}

void HLLC::pstar()
{
double pmin=min(pl,pr),pmax=max(pl,pr);
double qrat=max(pl,pr)/min(pl,pr);
   cl=sqrt(GAMMA*pl/rl);  //left state
   cr=sqrt(GAMMA*pr/rr);  //right state
double rbar=0.5*(rl+rr), abar=0.5*(cl+cr);
ps=0.5*(pl+pr)-0.5*(ur-ul)*rbar*abar;
if(qrat<QMIN && (ps >= pmin && ps <= pmax)){     //PVRS
      us=0.5*(ul+ur)-0.5*(pr-pl)/(rbar*abar);
}else{
      if (ps < pmin){
//TRRS
            double plr=pow(pl/pr,gm1b2g);
      ps=pow((cl+cr-gm1b2*(ur-
ul))/(cl/pow(pl,gm1b2g)+cr/pow(pr,gm1b2g)),g2bgm1);
      us=(plr*ul/cl+ur/cr+5*(plr-1.0))/(plr/cl+1./cr);
   }else{                                    //TSRS
         double
gel=sqrt((rgm1b2/rl)/(gm1bgp1*pl+max(TOL,ps))),ger=sqrt((rgm1b2/rr)/
(gm1bgp1*pr+max(TOL,ps)));
      ps=(gel*pl+ger*pr-(ur-ul))/(gel+ger);
            us=0.5*((ul+ur)+(ps-pr)*ger-(ps-pl)*gel);
   }
}
```

}

E.5 GodunovR.cpp

```
////////////////////////////////////////////////////////////////
//                    GodunovR.cpp                            //
//       this program is the coding of the godunov            //
//       method using EXACT Riemann solver                    //
//       while you compiling                                  //
//       make sure that Riemann.h and Vector.h                //
//       header files and also a data file is in              //
//       appropriate folder. if you wish to solve             //
//       any riemann problem other then the Euler equations   //
//       just change the header file and type of variable F   //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "Riemann.h"
#define CFL 0.2

Vector *u,*w;
Vector *Flux;
Vector bcr,bcmr,bcml;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
Riemann F;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();


main()
{
      Setdata();
      double currtime=0.;
      do{
            settimestep();
            currtime+=dt;
      if(currtime > time)dt=currtime-time;
      for (int i = 0 ; i < ngrd ; ++i){
```

```
            Tofluxclass(i);
          F.Solvexy();
          Flux[i]=F.flux*1.;
       }
       for (int i = 1 ; i < ngrd ; ++i){
             u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
       }
       UtoW();
       bc();
       }while(currtime < time);
       output();
//    getch();
}

void bc(void)
{
       switch(bc0){
    case 0: w[0]=w[1];break;
    case 1: w[0]=w[1]*bcr;break;
    case 2: w[0]=w[1]*bcr+bcml;break;
    }
       switch(bcn){
    case 0: w[ngrd]=w[ngrd-1];break;
    case 1: w[ngrd]=w[ngrd-1]*bcr;break;
    case 2: w[ngrd]=w[ngrd-1]*bcr+bcmr;break;
    }
}

void Tofluxclass(int i)
{
       F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
       F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
}

void settimestep()
{
    double smax=0;dt=100;
       for (int i =0; i < ngrd ; ++i){
             smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
       dt=min(dt,dx[i]/smax);
    }
    dt*=CFL;
}


void WtoU(void)
{
       for (int i=0; i<=ngrd ; ++i){
       u[i](0)=w[i](0);
       u[i](1)=w[i](0)*w[i](1);
       u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
    }
}

void UtoW(void)
{
```

```
      for (int i=0; i<=ngrd ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
      fclose(inputfile);

      Flux= new Vector[ngrd+1];
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
   dx =     new double[ngrd];

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i < ngrd / 2){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

}


void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](
1),w[i](2),(w[i](2)/0.4/w[i](0)));
   }
   fclose(outfile);
}
```

E.6 GodunovH.cpp

```
////////////////////////////////////////////////////////////////
//                    GodunovH.cpp                            //
//       this program is the coding of the godunov            //
//       method using APPROXIMATE Riemann solver HLLC         //
//       while you compiling                                  //
//       make sure that HLLC.h and Vector.h                   //
//       header files and also a data file is in              //
//       appropriate folder. if you wish to solve             //
//       any riemann problem other then the Euler equations   //
//       just change the header file and type of variable F   //
//       only difference between this file and                //
//       the GodunovR.cpp file is the type of the flux.       //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLC.h"
#define CFL 0.2

Vector *u,*w;
Vector *Flux;
Vector bcr,bcmr,bcml;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();


main()
{
      Setdata();
      double currtime=0.;
      do{
            settimestep();
            currtime+=dt;
```

```
      if(currtime > time)dt=currtime-time;
      for (int i = 0 ; i < ngrd ; ++i){
            Tofluxclass(i);
         F.Solvexy();
         Flux[i]=F.flux*1.;
      }
      for (int i = 1 ; i < ngrd ; ++i){
            u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
      }
      UtoW();
      bc();
      }while(currtime < time);
      output();
//    getch();
}

void bc(void)
{
      switch(bc0){
   case 0: w[0]=w[1];break;
   case 1: w[0]=w[1]*bcr;break;
   case 2: w[0]=w[1]*bcr+bcml;break;
   }
      switch(bcn){
   case 0: w[ngrd]=w[ngrd-1];break;
   case 1: w[ngrd]=w[ngrd-1]*bcr;break;
   case 2: w[ngrd]=w[ngrd-1]*bcr+bcmr;break;
   }
}

void Tofluxclass(int i)
{
      F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
      F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
   F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
   double smax=0;dt=100;
      for (int i =0; i < ngrd ; ++i){
            smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
      dt=min(dt,dx[i]/smax);
   }
   dt*=CFL;
}


void WtoU(void)
{
      for (int i=0; i<=ngrd ; ++i){
      u[i](0)=w[i](0);
      u[i](1)=w[i](0)*w[i](1);
      u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
   }
}
```

```
void UtoW(void)
{
      for (int i=0; i<=ngrd ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
      fclose(inputfile);

      Flux= new Vector[ngrd+1];
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
   dx =     new double[ngrd];

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i < ngrd / 2){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

}


void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](
1),w[i](2),(w[i](2)/0.4/w[i](0)));
   }
   fclose(outfile);
}
```

E.7 WAF.cpp

```
////////////////////////////////////////////////////////////////
//                    WAF-TVD.cpp                             //
//       this program is the TVD version of WAF               //
//       method using APPROXIMATE Riemann solver HLLC.        //
//       while you compiling                                  //
//       make sure that HLLCwaf.h and Vector.h also           //
//       header files and also a data file is in              //
//       appropriate folder. note that, the header            //
//       file for HLLC solver is different than that of       //
//       other methods. if you wish to solve                  //
//       any riemann problem other then the Euler equations   //
//       just change the header file and type of variable F   //
////////////////////////////////////////////////////////////////


#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLCwaf.h"
#define CFL 0.90

Vector *u,*w;
Vector *Flux, **Flocal;
Vector Fold[4],Foldold[4];
Vector bcr,bcmr,bcml;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
double **c,**q,r[5];
double rold[4], roldold[4];
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();

main()
{
      Setdata();
      double currtime=0.,r0,rn;
```

```
      do{
            settimestep();
            currtime+=dt;
      printf("%lf\t%d\n",currtime);
      if(currtime > time)dt=currtime-time;
      for (int i = 0 ; i < ngrd ; ++i){
            Tofluxclass(i);
         F.Solvexy();
         c[i][1]=dt/dx[i]*F.Sl;c[i][2]=dt/dx[i]*F.Ss;
         c[i][3]=dt/dx[i]*F.Sr;
      }
      for (int i = 0 ; i < ngrd ; ++i){
            Flux[i]=(Flocal[i][0]+Flocal[i][3]);
                  if(i > 0 && i < ngrd-1){
            Flux[i]-
=(sign(c[i][j]))*limiter(r[j],fabs(c[i][j]))*(Flocal[i][j]-
Flocal[i][j-1]);
         }
            Flux[i]=0.5*Flux[i];
      }
      for (int i = 1 ; i < ngrd ; ++i){
            u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
      }
      UtoW();
      bc();
      }while(currtime < time);
      output();
//    getch();
}

void bc(void)
{
      switch(bc0){
   case 0: w[0]=w[1]*1.0;break;
   case 1: w[0]=w[1]*bcr;break;
   case 2: w[0]=w[1]*bcr+bcml;break;
   }
      switch(bcn){
   case 0: w[ngrd]=w[ngrd-1]*1.0;break;
   case 1: w[ngrd]=w[ngrd-1]*bcr;break;
   case 2: w[ngrd]=w[ngrd-1]*bcr+bcmr;break;
   }
}

void Tofluxclass(int i)
{
      F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
      F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
   F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
   double smax=0;dt=100;
      for (int i =0; i < ngrd ; ++i){
            smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
```

```
      dt=min(dt,dx[i]/smax);
   }
   dt*=CFL;
}


void WtoU(void)
{
      for (int i=0; i<=ngrd ; ++i){
      u[i](0)=w[i](0);
      u[i](1)=w[i](0)*w[i](1);
      u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
   }
}

void UtoW(void)
{
      for (int i=0; i<=ngrd ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
      fclose(inputfile);

   Flux= new Vector[ngrd+1];
      Flocal= new Vector*[ngrd+1];
      for (int j = 0; j < ngrd+1; j++)
          Flocal[j] = new Vector[4];  // STEP 2: SET UP THE COLUMNS
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
   dx =     new double[ngrd];
      c = new double*[ngrd];         // STEP 1: SET UP THE ROWS.
      for (int j = 0; j < ngrd+1; j++)
          c[j] = new double[5];  // STEP 2: SET UP THE COLUMNS*/
      q = new double*[ngrd];         // STEP 1: SET UP THE ROWS.
      for (int j = 0; j < ngrd+1; j++)
          q[j] = new double[5];  // STEP 2: SET UP THE COLUMNS*/

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i <= ngrd / 2){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();
```

```
   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

}


void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](
1),w[i](2),(w[i](2)/0.4/w[i](0)));
   }
   fclose(outfile);
}
```

E.8 MUSCL.cpp

```
////////////////////////////////////////////////////////////////
//                      MUSCL.cpp                             //
//       this program is the MUSCL                            //
//       method using APPROXIMATE Riemann solver HLLC         //
//       and flux limiters. while you compiling               //
//       make sure that HLLC.h and Vector.h                   //
//       header files and also a data file is in              //
//       appropriate folder. if you wish to solve             //
//       any riemann problem other then the Euler equations   //
//       just change the header file and type of variable F   //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLC.h"
#define CFL 0.20
#define W -1.0
#define beta 1.0

Vector *u,*w;
Vector *Flux;
Vector bcr,bcmr,bcml;
Vector delta,dplus,dminus;
Vector *ubarleft,*ubarright,uleft,uright,uold;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();


main()
{
Vector Dummy;
      Setdata();
```

```
      double currtime=0.;
      do{
            settimestep();
            currtime+=dt;
            printf("%lf\n",currtime);
      for (int i = 1 ; i < ngrd ; ++i){
         dplus=(u[i+1]-u[i]);
         dminus=(u[i]-u[i-1]);
            delta=0.5*((1.-W)*dminus+(1.+W)*dplus);
         uleft   =(u[i]*1.0)-0.5*delta;
         uright  =(u[i]*1.0)+0.5*delta;
         Dummy(0)=uleft(1)-uright(1);
         Dummy(1)=(0.8*uleft(1)*uleft(1)/uleft(0)+0.4*uleft(2));
         Dummy(1)-
=(0.8*uright(1)*uright(1)/uright(0)+0.4*uright(2));
         Dummy(2)=(1.4*uleft(1)*uleft(2)/uleft(0)-
0.2*uleft(1)*uleft(1)*uleft(1)/uleft(0)/uleft(0));
         Dummy(2)-=(1.4*uright(1)*uright(2)/uright(0)-
0.2*uright(1)*uright(1)*uright(1)/uright(0)/uright(0));
                  uleft+=(0.5*dt/dx[i])*Dummy;
                  uright+=(0.5*dt/dx[i])*Dummy;
         if(i>1){
                        F.ul=uold(1)/uold(0);F.rl=uold(0);
            F.pl=(uold(2)-0.5*F.ul*F.ul*F.rl)*0.4;
                        F.ur=uleft(1)/uleft(0);F.rr=uleft(0);
                     F.pr=(uleft(2)-0.5*F.ur*F.ur*F.rr)*0.4;
                        F.Ul=uold*1.;F.Ur=uleft*1.;
            F.Solvexy();
            Flux[i-1]=F.flux*1.;
         }
         uold=uright*1.;
      }
      for (int i = 2 ; i < ngrd-1 ; ++i){
            u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
      }
      UtoW();
      bc();
      }while(currtime < time);
      output();
   getch();
}

void bc(void)
{
      switch(bc0){
   case 0: w[0]=w[1]*1.0;break;
   case 1: w[0]=w[1]*bcr;break;
   case 2: w[0]=w[1]*bcr+bcml;break;
   }
      switch(bcn){
   case 0: w[ngrd]=w[ngrd-1]*1.0;break;
   case 1: w[ngrd]=w[ngrd-1]*bcr;break;
   case 2: w[ngrd]=w[ngrd-1]*bcr+bcmr;break;
   }
}

void Tofluxclass(int i)
```

```
{
      F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
      F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
   F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
   double smax=0;dt=100;
      for (int i =0; i < ngrd ; ++i){
            smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
      dt=min(dt,dx[i]/smax);
   }
   dt*=CFL;
}


void WtoU(void)
{
      for (int i=0; i<=ngrd ; ++i){
      u[i](0)=w[i](0);
      u[i](1)=w[i](0)*w[i](1);
      u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
   }
}

void UtoW(void)
{
      for (int i=0; i<=ngrd ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
      fclose(inputfile);

      Flux= new Vector[ngrd+1];
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
   dx =     new double[ngrd];

   ubarleft = new Vector[ngrd+1];
   ubarright = new Vector[ngrd+1];

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
```

```
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i < ngrd / 2){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

}


void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](1),w[
i](2));
   }
   fclose(outfile);
}
```

E.9 WAF-TVD.cpp

```
////////////////////////////////////////////////////////////////
//                    WAF-TVD.cpp                             //
//       this program is the TVD version of WAF               //
//       method using APPROXIMATE Riemann solver HLLC.        //
//       while you compiling                                  //
//       make sure that HLLCwaf.h and Vector.h also           //
//       flux limiter header file LimiterS.h                  //
//       header files and also a data file is in              //
//       appropriate folder. note that, the header            //
//       file for HLLC solver is different than that of       //
//       other methods. if you wish to solve                  //
//       any riemann problem other then the Euler equations   //
//       just change the header file and type of variable F   //
////////////////////////////////////////////////////////////////


#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLCwaf.h"
#include "limiterF.h"
#define CFL 0.90

Vector *u,*w;
Vector *Flux, **Flocal;
Vector Fold[4],Foldold[4];
Vector bcr,bcmr,bcml;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
double **c,**q,r[5];
double rold[4], roldold[4];
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();
double safediv(double, double);

main()
```

```
{
      Setdata();
      double currtime=0.,r0,rn;
      do{
            settimestep();
            currtime+=dt;
      printf("%lf\t%d\n",currtime);
      if(currtime > time)dt=currtime-time;
      for (int i = 0 ; i < ngrd ; ++i){
            Tofluxclass(i);
         F.Solvexy();
         c[i][1]=dt/dx[i]*F.Sl;c[i][2]=dt/dx[i]*F.Ss;
         c[i][3]=dt/dx[i]*F.Sr;
         q[i][0]=F.rl*1.;     q[i][1]=F.rsl*1.;
         q[i][2]=F.rsr*1.;    q[i][3]=F.rr*1.;
         for (int j= 0 ; j < 4 ; ++j)Flocal[i][j]=F.flux[j]*1.;
      }
      for (int i = 0 ; i < ngrd ; ++i){
            Flux[i]=(Flocal[i][0]+Flocal[i][3]);
                  if(i > 0 && i < ngrd-1){
             for(int j=1; j < 4 ; ++j){
            if(c[i][j]>=0.){
                  r[j]=safediv((q[i-1][j]-q[i-1][j-1]),(q[i][j]-
q[i][j-1]));
            }else {
                  r[j]=safediv((q[i+1][j]-q[i+1][j-1]),(q[i][j]-
q[i][j-1]));
            }
            Flux[i]-
=(sign(c[i][j]))*limiter(r[j],fabs(c[i][j]))*(Flocal[i][j]-
Flocal[i][j-1]);
                  }
         }
            Flux[i]=0.5*Flux[i];
      }
      for (int i = 1 ; i < ngrd ; ++i){
            u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
      }
      UtoW();
      bc();
      }while(currtime < time);
      output();
//    getch();
}

void bc(void)
{
      switch(bc0){
   case 0: w[0]=w[1]*1.0;break;
   case 1: w[0]=w[1]*bcr;break;
   case 2: w[0]=w[1]*bcr+bcml;break;
   }
      switch(bcn){
   case 0: w[ngrd]=w[ngrd-1]*1.0;break;
   case 1: w[ngrd]=w[ngrd-1]*bcr;break;
   case 2: w[ngrd]=w[ngrd-1]*bcr+bcmr;break;
   }
```

```
}

void Tofluxclass(int i)
{
      F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
      F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
   F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
   double smax=0;dt=100;
      for (int i =0; i < ngrd ; ++i){
            smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
      dt=min(dt,dx[i]/smax);
   }
   dt*=CFL;
}


void WtoU(void)
{
      for (int i=0; i<=ngrd ; ++i){
      u[i](0)=w[i](0);
      u[i](1)=w[i](0)*w[i](1);
      u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
   }
}

void UtoW(void)
{
      for (int i=0; i<=ngrd ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
      fclose(inputfile);

   Flux= new Vector[ngrd+1];
      Flocal= new Vector*[ngrd+1];
      for (int j = 0; j < ngrd+1; j++)
          Flocal[j] = new Vector[4];  // STEP 2: SET UP THE COLUMNS
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
   dx =     new double[ngrd];
```

```
      c = new double*[ngrd];         // STEP 1: SET UP THE ROWS.
      for (int j = 0; j < ngrd+1; j++)
          c[j] = new double[5];  // STEP 2: SET UP THE COLUMNS*/
      q = new double*[ngrd];         // STEP 1: SET UP THE ROWS.
      for (int j = 0; j < ngrd+1; j++)
          q[j] = new double[5];  // STEP 2: SET UP THE COLUMNS*/

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i <= ngrd / 2){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

}


void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](
1),w[i](2),(w[i](2)/0.4/w[i](0)));
   }
   fclose(outfile);
}

double safediv(double a, double b)
{
if(b < 1.e-10){
if(a < 1.e-10 && a > -1.e-10){return 1.;
}else{if(a > 0)return 100000.;
        else return -1000000;
     }
}
return a/b;
}
```

E.10 HLLCwaf.h

```
////////////////////////////////////////////////////////////////
//                    HLLCwaf.h                               //
//       this is the WAF version of HLLC header file          //
//       used in many godunov WAF-TVD.cpp file                //
//       this file contains a c++ class                       //
//       that is used to solve APPROXIMATE Riemann problem of //
//       Euler problem using Harten-Lax and vanLeer's solver. //
//       it also calculates flux vectors for all three states //
//       make sure that Vector.h header                       //
//       file is in an appropriate folder in compilation      //
////////////////////////////////////////////////////////////////

#include <math.h>
#include "Vector.h"
#define GAMMA 1.4
#define QMIN 2.0
#define TOL 1.e-6
#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))

class HLLC {
double ps,us,S,u,r,p,c;
double gm1b2g,gp1b2g,gm1b2,g2bgm1,rgm1b2,gm1bgp1;

public:
double pl,rl,ul,cl,pr,rr,ur,cr;
Vector flux[4],Us,Ul,Ur;
   double Ss;
      double ql,qr,Sl,Sr;
   double rsl,rsr;
      void pstar(void);
   void Solvexy();
   HLLC(){
            gm1b2g=0.5*(GAMMA-1)/GAMMA;
      g2bgm1=1./gm1b2g;
      gp1b2g=0.5*(GAMMA+1)/GAMMA;
            gm1b2=0.5*(GAMMA-1);
            rgm1b2=1./gm1b2;
            gm1bgp1=(GAMMA-1)/(GAMMA+1);
   }
};

void HLLC::Solvexy()
{
   pstar();
   if(pl <= ps)ql=sqrt(1.+gp1b2g*(ps/pl-1.));
```

```
   else ql=1.;
   Sl=ul-cl*ql;
   if(pr <= ps)qr=sqrt(1.+gp1b2g*(ps/pr-1.));
   else qr=1.;
   Sr=ur+cr*qr;
   Ss=(pr-pl+rl*ul*(Sl-ul)-rr*ur*(Sr-ur))/(rl*(Sl-ul)-rr*(Sr-ur));
//   Ss=us;

      flux[0](0)=rl*ul;flux[0](1)=rl*ul*ul+pl;flux[0](2)=ul*(rl*ul*u
l*0.5+pl*2.5+pl);
      Us(0)=1.;Us(1)=Ss; Us(2)=(0.5*ul*ul+2.5*pl/rl)+(Ss-
ul)*(Ss+pl/(rl*(Sl-ul)));
      Us=(rl*(Sl-ul)/(Sl-Ss))*Us;
      flux[1]=flux[0]+Sl*(Us-Ul);


      flux[3](0)=rr*ur;flux[3](1)=rr*ur*ur+pr;flux[3](2)=ur*(rr*ur*u
r*0.5+pr*2.5+pr);
      Us(0)=1.;Us(1)=Ss;Us(2)=(0.5*ur*ur+2.5*pr/rr)+(Ss-
ur)*(Ss+pr/(rr*(Sr-ur)));
      Us=(rr*(Sr-ur)/(Sr-Ss))*Us;
      flux[2]=flux[3]+Sr*(Us-Ur);
}

void HLLC::pstar()
{
double pmin=min(pl,pr),pmax=max(pl,pr);
double qrat=max(pl,pr)/min(pl,pr);
   cl=sqrt(GAMMA*pl/rl);  //left state
   cr=sqrt(GAMMA*pr/rr);  //right state
double rbar=0.5*(rl+rr), abar=0.5*(cl+cr);
ps=0.5*(pl+pr)-0.5*(ur-ul)*rbar*abar;
if(qrat<QMIN && (ps >= pmin && ps <= pmax)){     //PVRS
      us=0.5*(ul+ur)-0.5*(pr-pl)/(rbar*abar);
   rsl=rl+(ps-pl)/(cl*cl);
   rsr=rr+(ps-pr)/(cr*cr);
}else{
      if (ps < pmin){
//TRRS
            double plr=pow(pl/pr,gm1b2g);
      ps=pow((cl+cr-gm1b2*(ur-
ul))/(cl/pow(pl,gm1b2g)+cr/pow(pr,gm1b2g)),g2bgm1);
      us=(plr*ul/cl+ur/cr+5*(plr-1.0))/(plr/cl+1./cr);
      rsl=rl*pow((ps/pl),1./GAMMA);
      rsr=rr*pow((ps/pr),1./GAMMA);
   }else{                                    //TSRS
         double
gel=sqrt((rgm1b2/rl)/(gm1bgp1*pl+max(TOL,ps))),ger=sqrt((rgm1b2/rr)/
(gm1bgp1*pr+max(TOL,ps)));
      ps=(gel*pl+ger*pr-(ur-ul))/(gel+ger);
            us=0.5*((ul+ur)+(ps-pr)*ger-(ps-pl)*gel);
      rsl=rl*((ps/pl+gm1bgp1)/(gm1bgp1*ps/pl+1.));
      rsr=rr*((ps/pr+gm1bgp1)/(gm1bgp1*ps/pr+1.));
   }
}
}
```

E.11 LimiterF.h

```
////////////////////////////////////////////////////////////////
//                    LimiterF.h                              //
//       header file that contains flux limiters              //
//       for WAF method with TVD extension                    //
////////////////////////////////////////////////////////////////

#define SUPERBEE  0
#define VANLEER   1
#define VANALBADA 2
#define MINBEE          3
#define CHECK           4
#ifndef LIMITER
      #define LIMITER 3
#endif
#define sign(a) (((a)>=0.0)   ?     (1.0) :     (-1.0))

double superbee(double, double);
double vanleer(double, double);
double vanalbada(double, double);
double minbee(double, double);
double check(double, double);
double limiter(double, double);

double limiter(double r,double c)
{
      switch(LIMITER){
      case SUPERBEE:    return superbee(r,c);break;
      case VANLEER:     return vanleer(r,c);break;
      case VANALBADA:return vanalbada(r,c);break;
      case MINBEE:      return minbee(r,c);break;
      case CHECK: return check(r,c);break;
   }
}

double superbee(double r, double c)
{
      if(r<=0.)return 1.;
      if(r>=0. && r <= 0.5 )return 1.-2.*r*(1.-c);
      if(r>=0.5 && r<=1.)return c;
      if(r>=1. && r<= 2.)return 1.-r*(1.-c);
      if(r>=2.)return 2.*c-1.;
}

double vanleer(double r, double c)
{
      if(r<=0.)return 1.;
```

```
      if(r>=0.)return 1.-2.*r*(1.-c)/(1+r);
}

double vanalbada(double r, double c)
{
      if(r<=0.)return 1.;
      if(r>=0.)return 1.-r*(1.-c)*(1+r)/(1+r*r);
}

double minbee(double r, double c)
{
      if(r<=0.)return 1.;
      if(r>=0.&& r<=1.)return r;
      if(r>=1.)return c;
}

double check(double r, double c)
{
      return c;
}
```

E.12 MUSCLTVD.cpp

```
////////////////////////////////////////////////////////////////
//                    MUSCLSTVD.cpp                           //
//       this program is the TVD version of MUSCL             //
//       method using APPROXIMATE Riemann solver HLLC         //
//       and flux limiters. while you compiling               //
//       make sure that HLLC.h and Vector.h also              //
//       slope limiter header file LimiterS.h                 //
//       header files and also a data file is in              //
//       appropriate folder. if you wish to solve             //
//       any riemann problem other then the Euler equations   //
//       just change the header file and type of variable F   //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLC.h"
#include "limiterS.h"
#define CFL 0.80
#define W 0.0
#define beta 1.0

Vector *u,*w;
Vector *Flux;
Vector bcr,bcmr,bcml;
Vector delta,dplus,dminus;
Vector *ubarleft,*ubarright,uleft,uright,uold;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();
double safediv(double, double);


main()
{
```

```
Vector Dummy;
      Setdata();
      double currtime=0.;
      do{
            settimestep();
            currtime+=dt;
            printf("%lf\n",currtime);
      for (int i = 1 ; i < ngrd ; ++i){
         dplus=(u[i+1]-u[i]);
         dminus=(u[i]-u[i-1]);
            delta=0.5*((1.-W)*dminus+(1.+W)*dplus);
                  for (int j= 0 ; j < 3 ; ++j){
                        double r=safediv(dminus(j),dplus(j));
            double eta=2/(1-W+(1+W)*r);
            delta(j)*=limiter(r,eta);
         }
         uleft   =(u[i]*1.0)-0.5*delta;
         uright  =(u[i]*1.0)+0.5*delta;
         Dummy(0)=uleft(1)-uright(1);
         Dummy(1)=(0.8*uleft(1)*uleft(1)/uleft(0)+0.4*uleft(2));
         Dummy(1)-
=(0.8*uright(1)*uright(1)/uright(0)+0.4*uright(2));
         Dummy(2)=(1.4*uleft(1)*uleft(2)/uleft(0)-
0.2*uleft(1)*uleft(1)*uleft(1)/uleft(0)/uleft(0));
         Dummy(2)-=(1.4*uright(1)*uright(2)/uright(0)-
0.2*uright(1)*uright(1)*uright(1)/uright(0)/uright(0));
                  uleft+=(0.5*dt/dx[i])*Dummy;
                  uright+=(0.5*dt/dx[i])*Dummy;
         if(i>1){
                        F.ul=uold(1)/uold(0);F.rl=uold(0);
            F.pl=(uold(2)-0.5*F.ul*F.ul*F.rl)*0.4;
                        F.ur=uleft(1)/uleft(0);F.rr=uleft(0);
                     F.pr=(uleft(2)-0.5*F.ur*F.ur*F.rr)*0.4;
                        F.Ul=uold*1.;F.Ur=uleft*1.;
            F.Solvexy();
            Flux[i-1]=F.flux*1.;
         }
         uold=uright*1.;
      }
      for (int i = 2 ; i < ngrd-1 ; ++i){
            u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
      }
      UtoW();
      bc();
      }while(currtime < time);
      output();
   getch();
}

void bc(void)
{
      switch(bc0){
   case 0: w[0]=w[1]*1.0;break;
   case 1: w[0]=w[1]*bcr;break;
   case 2: w[0]=w[1]*bcr+bcml;break;
   }
      switch(bcn){
```

```
   case 0: w[ngrd]=w[ngrd-1]*1.0;break;
   case 1: w[ngrd]=w[ngrd-1]*bcr;break;
   case 2: w[ngrd]=w[ngrd-1]*bcr+bcmr;break;
   }
}

void Tofluxclass(int i)
{
      F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
      F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
   F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
   double smax=0;dt=100;
      for (int i =0; i < ngrd ; ++i){
            smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
      dt=min(dt,dx[i]/smax);
   }
   dt*=CFL;
}


void WtoU(void)
{
      for (int i=0; i<=ngrd ; ++i){
      u[i](0)=w[i](0);
      u[i](1)=w[i](0)*w[i](1);
      u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
   }
}

void UtoW(void)
{
      for (int i=0; i<=ngrd ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
      fclose(inputfile);

      Flux= new Vector[ngrd+1];
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
```

```
   dx =      new double[ngrd];

   ubarleft = new Vector[ngrd+1];
   ubarright = new Vector[ngrd+1];

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i < ngrd / 2){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

}


void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](1),w[
i](2));
   }
   fclose(outfile);
}

double safediv(double a, double b)
{
if(b < 1.e-10){
if(a < 1.e-10 && a > -1.e-10){return 1.;
}else{if(a > 0)return 100000.;
        else return -1000000;
      }
}
return a/b;
}
```

E.13 LimiterS.h

```
////////////////////////////////////////////////////////////////
//                    LimiterS.h                              //
//       header file that contains slope limiters             //
//       for MUSCL method with slope limiter TVD extension    //
////////////////////////////////////////////////////////////////

#define SUPERBEE  0
#define VANLEER   1
#define VANALBADA 2
#define MINBEE          3
#define CHECK           4
#ifndef LIMITER
      #define LIMITER 0
#endif
#define sign(a) (((a)>=0.0)   ?     (1.0) :     (-1.0))

double superbee(double,double);
double vanleer(double,double);
double vanalbada(double,double);
double minbee(double,double);
double check(double,double);
double limiter(double,double);

double limiter(double r,double eta)
{
      switch(LIMITER){
      case SUPERBEE:    return superbee(r,eta);break;
      case VANLEER:     return vanleer(r,eta);break;
      case VANALBADA:return vanalbada(r,eta);break;
      case MINBEE:      return minbee(r,eta);break;
      case CHECK: return check(r,eta);break;
   }
}

double superbee(double r,double eta)
{
      if(r<=0.)return 0.;
      if(r>=0. && r <= 0.5 )return 2.*r;
      if(r>=0.5 && r<=1.)return 1;
      return min(r,min(eta,2.));
}

double vanleer(double r,double eta)
{
      if(r<=0.)return 0.;
      return min(2.*r/(1.+r),eta);
}
```

```
double vanalbada(double r, double eta)
{
      if(r<=0.)return 0.;
      return min((1.+r)*r/(1.+r*r),eta);
}

double minbee(double r, double eta)
{
      if(r<=0.)return 0.;
      if(r>=0.&& r<=1.)return r;
      return min(1.,eta);
}

double check(double r, double eta)
{
      return 0.0;
}
```

E.14 MUSCLSl.cpp

```
////////////////////////////////////////////////////////////////
//                    MUSCLS1.cpp                             //
//       this program is the TVD version of MUSCL             //
//       method using APPROXIMATE Riemann solver HLLC         //
//       and limited slope. while you compiling               //
//       make sure that HLLC.h and Vector.h                   //
//       header files and also a data file is in              //
//       appropriate folder. if you wish to solve             //
//       any riemann problem other then the Euler equations   //
//       just change the header file and type of variable F   //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLC.h"
#define CFL 0.45
#define W 0.5
#define beta 1.0

Vector *u,*w;
Vector *Flux;
Vector bcr,bcmr,bcml;
Vector delta,dplus,dminus;
Vector *ubarleft,*ubarright,uleft,uright,uold;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();


main()
{
Vector Dummy;
      Setdata();
      double currtime=0.;
```

```
      do{
            settimestep();
            currtime+=dt;
      for (int i = 1 ; i < ngrd ; ++i){
         dplus=(u[i+1]-u[i]);
         dminus=(u[i]-u[i-1]);
         for(int j=0; j < 3; ++j){
            if (dplus(j)>0.){

      delta(j)=max(0,max(min(beta*dminus(j),dplus(j)),min(dminus(j),
beta*dplus(j))));
            }else{

      delta(j)=min(0,min(max(beta*dminus(j),dplus(j)),max(dminus(j),
beta*dplus(j))));
            }
         }
         uleft   =(u[i]*1.0)-0.5*delta;
         uright  =(u[i]*1.0)+0.5*delta;
         Dummy(0)=uleft(1)-uright(1);
         Dummy(1)=(0.8*uleft(1)*uleft(1)/uleft(0)+0.4*uleft(2));
         Dummy(1)-
=(0.8*uright(1)*uright(1)/uright(0)+0.4*uright(2));
         Dummy(2)=(1.4*uleft(1)*uleft(2)/uleft(0)-
0.2*uleft(1)*uleft(1)*uleft(1)/uleft(0)/uleft(0));
         Dummy(2)-=(1.4*uright(1)*uright(2)/uright(0)-
0.2*uright(1)*uright(1)*uright(1)/uright(0)/uright(0));
                  uleft+=(0.5*dt/dx[i])*Dummy;
                  uright+=(0.5*dt/dx[i])*Dummy;
         if(i>1){
                        F.ul=uold(1)/uold(0);F.rl=uold(0);
            F.pl=(uold(2)-0.5*F.ul*F.ul*F.rl)*0.4;
                        F.ur=uleft(1)/uleft(0);F.rr=uleft(0);
                     F.pr=(uleft(2)-0.5*F.ur*F.ur*F.rr)*0.4;
                        F.Ul=uold;F.Ur=uleft;
            F.Solvexy();
            Flux[i-1]=F.flux*1.;
         }
         uold=uright*1.;
//          Tofluxclass(i);
       }
       for (int i = 2 ; i < ngrd-1 ; ++i){
            u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
       }
       UtoW();
       bc();
       }while(currtime < time);
       output();
    getch();
 }

 void bc(void)
 {
       switch(bc0){
    case 0: w[0]=w[1]*1.0;break;
    case 1: w[0]=w[1]*bcr;break;
    case 2: w[0]=w[1]*bcr+bcml;break;
```

```
   }
      switch(bcn){
   case 0: w[ngrd]=w[ngrd-1]*1.0;break;
   case 1: w[ngrd]=w[ngrd-1]*bcr;break;
   case 2: w[ngrd]=w[ngrd-1]*bcr+bcmr;break;
   }
}

void Tofluxclass(int i)
{
      F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
      F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
   F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
   double smax=0;dt=100;
      for (int i =0; i < ngrd ; ++i){
            smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
      dt=min(dt,dx[i]/smax);
   }
   dt*=CFL;
}


void WtoU(void)
{
      for (int i=0; i<=ngrd ; ++i){
      u[i](0)=w[i](0);
      u[i](1)=w[i](0)*w[i](1);
      u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
   }
}

void UtoW(void)
{
      for (int i=0; i<=ngrd ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
      fclose(inputfile);

      Flux= new Vector[ngrd+1];
   u  =     new Vector[ngrd+1];
```

```
   w  =      new Vector[ngrd+1];
   x  =      new double[ngrd+1];
   dx =      new double[ngrd];

   ubarleft = new Vector[ngrd+1];
   ubarright = new Vector[ngrd+1];

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i < ngrd / 2){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

}


void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](1),w[
i](2));
   }
   fclose(outfile);
}
```

E.15 Vector.H

```
////////////////////////////////////////////////////////////////////////
//          VECTOR CLASS FOR MANAGING ONE DIMENSIONAL ARRAYS          //
//                    WRITTEN BY OZGUR UGRAS BARAN                    //
//                   LAST MODIFIED : JUNE 24TH, 1999                  //
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
//WARNING! THIS CLASS HAS BEEN WRITTEN TO COMPLETE SOME COMMON TASKS IN 1-D    /
//ARRAY CALCULATIONS. SOME CONTROL MECHANISMS, SUCH AS DIMENSION CHECK, HAS NOT/
//BEEN ADDED IN CODE, TO MAINTAIN SPEED IN NUMERICAL CALCULATIONS. ENSURE THAT /
//YOU ARE USING COMPATIBLE SIZE VECTORS IN YOUR CALCULATIONS. MOREOVER, SOME   /
//OPERATORS SEEMS MEANINGLESS OR HAVE TOTALLY DIFFERENT MEANING THAN THE       /
//COMMONLY USED ONE (SUCH AS Vector * Vector OPERATION. IT IS NOT DOT PRODUCT  /
//OR CROSS PRODUCT, BUT THE MULTIPLICATION OF EACH ELEMENT.) HAVE FUN!         /
////////////////////////////////////////////////////////////////////////


#include <mem.h>
#ifndef __VECSIZE
      #define __VECSIZE 3
#endif

class Vector{

public:
   double *data;                        //data array
   short int Size;          //size of vector
   int Dim;                                  // dimension of vector
//    Vector **dm;
      //Constructors
      Vector(short int size = __VECSIZE); //the default size is set
to be 3
//    Vector(int dim=0, short int size = 3);    //the default size
is set to be 3
      //destructor
//   ~Vector();

      //Assignment Operator
   Vector& operator = (const Vector& m);

   //Assignment and calculation operators
   Vector& operator += (const Vector& v1);
   Vector& operator -= (const Vector& v1);
   Vector& operator *= (const Vector& v1);
   Vector& operator *= (const double& v1);
```

```
      //Assignment operator for single data
      double& operator () (int row);
//    Vector& operator [] (int vc);


private:
      //basic calculation operators, since they are friend type,
they should be
   //in private declarations
      friend Vector operator + (const Vector& v1, const Vector& v2);
      friend Vector operator - (const Vector& v1, const Vector& v2);
      friend Vector operator * (const Vector& v1, const Vector& v2);
      friend Vector operator * (const double& v1, const Vector& v2);
      friend Vector operator * ( const Vector& v2,const double&
v1){return v1*v2;}
      friend Vector operator / (const Vector& v1, const Vector& v2);
      friend Vector operator / ( const Vector& v2,const double& v1);

};

//   Vector::~Vector(){ delete Size;}

inline Vector::Vector(short int size){
                  data=new double [size];
         Size=size;
}

/*inline Vector::Vector(int dim,short int size){
         *dm=new Vector* [dim];

         for(int j=0;j<dim; ++j){dm[j]=new Vector(3); }
         Size=size;
         Dim=dim;
}*/


/*inline Vector& Vector::operator [] (int vc){
                  return *this + sizeof(Vector)*vc;
} */

inline double& Vector::operator () (int row){
                  return data[row];
}

inline Vector operator + (const Vector& v1, const Vector& v2){
      Vector dummy(v1.Size);
      for(int i=0; i < v1.Size ;++i){
      dummy.data[i]=v1.data[i]+v2.data[i];
   }
   return dummy;
}

inline Vector operator - (const Vector& v1, const Vector& v2){
      Vector dummy(v1.Size);
      for(int i=0; i < v1.Size ;++i){
      dummy.data[i]=v1.data[i]-v2.data[i];
```

```
   }
   return dummy;
}

inline Vector operator * (const Vector& v1, const Vector& v2){
      Vector dummy(v1.Size);
      for(int i=0; i < v1.Size ;++i){
      dummy.data[i]=v1.data[i]*v2.data[i];
   }
   return dummy;
}

inline Vector operator / (const Vector& v1, const Vector& v2){
      Vector dummy(v1.Size);
      for(int i=0; i < v1.Size ;++i){
      dummy.data[i]=v1.data[i]/v2.data[i];
   }
   return dummy;
}

inline Vector operator * (const double& v1, const Vector& v2){
      Vector dummy(v2.Size);
      for(int i=0; i < v2.Size ;++i){
      dummy.data[i]=v1*v2.data[i];
   }
   return dummy;
}

inline Vector operator / (const Vector& v2, const double& v1){
      Vector dummy(v2.Size);
      for(int i=0; i < v2.Size ;++i){
      dummy.data[i]=v2.data[i]/v1;
   }
   return dummy;
}

inline Vector& Vector::operator = (const Vector& v1){
      data=v1.data;
   return *this;
}

inline Vector& Vector::operator += (const Vector& v1){
      for(int i=0; i < v1.Size ;++i){
   data[i]+=v1.data[i];
   }
   return *this;
}

inline Vector& Vector::operator *= (const Vector& v1){
      for(int i=0; i < v1.Size ;++i){
   data[i]*=v1.data[i];
   }
   return *this;
}

inline Vector& Vector::operator *= (const double& v1){
      for(int i=0; i < Size ;++i){
```

```
   data[i]*=v1;
   }
   return *this;
}

inline Vector& Vector::operator -= (const Vector& v1){
      for(int i=0; i < v1.Size ;++i){
   data[i]-=v1.data[i];
   }
   return *this;
}
//notice: ::Vector [] has some bugs. if you fix this bug please
repot
//to hydraulics lab civil engineering dept. Middle East Technical
University
//if you write something like
// vector a, b[ARRSIZE];
//int i;
// ...
// a=b[i]; then you just equate the adress of a to address of b[i]
// instead you have to write as a=b[i]*1.0;
```

APPENDIX F

SIMULATION CODES

F.1 SIMGOD.CPP

```
SIMGOD.cpp                                    //
//       SIMULATION OF LIQUID SLUG PROBLEM                          //
//       WITH GODUNOV METHOD.while you compiling                    //
//       make sure that HLLC.h and Vector.h                         //
//       header files and also a data file is in                    //
//       appropriate folder.                                        //
////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLC.h"
#define CFL 0.2

Vector *u,*w;
Vector *Flux;
Vector bcr,bcmr,bcml;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

int nend;
Vector tankbdFlux,tankbdFlux1;
double tankdensity,tankpressure,bvel,slugf,slugpos;
double sluglength,slugmass,f;
double dxnorm=0.1;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
```

```
void bc();
void updategrid();


main()
{
      Setdata();
      double currtime=0.;
      do{
            settimestep();
            currtime+=dt;
      if(currtime > time)dt=currtime-time;
      for (int i = 0 ; i < nend ; ++i){
            Tofluxclass(i);
         F.Solvexy();
         Flux[i]=F.flux*1.;
      }
      for (int i = 1 ; i < nend ; ++i){
          u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
      }
      UtoW();
      updategrid();
      bc();
      WtoU();
      }while(currtime < time && slugpos < 14.0);
   printf("%lf\t%lf",bvel,w[nend-1](2));
      output();
   getch();
}
void updategrid()
{
      double drivingforce=(w[nend-1](2)-pr)*7.85e-3;
   double
resistingforce=0.;//f*sluglength*0.1*bvel*bvel/8./3.14*1000+slugmass
*9.81*0.08;
      double acc=(drivingforce-resistingforce)/slugmass;
   bvel+=acc*dt;
   slugpos+=bvel*dt;
   dx[nend-1]+=bvel*dt;
      bcmr(1)=2*bvel;
      if(dx[nend-1]>(1.5*dxnorm)){
            dx[nend]=dx[nend-1]-dxnorm;
         dx[nend-1]=dxnorm;
         w[nend]=w[nend-1]*1.0;
         ++nend;
      }
}

void bc(void)
{
      switch(bc0){
   case 0: w[0]=w[1]*1.0;break;
   case 1: w[0]=w[1]*bcr;break;
   case 2: w[0]=w[1]*bcr+bcml;break;
   }
      switch(bcn){
   case 0: w[nend]=w[nend-1]*1.0;break;
```

```
   case 1: w[nend]=w[nend-1]*bcr;break;
   case 2: w[nend]=w[nend-1]*bcr+bcmr;break;
   }
   tankdensity-=w[1](0)*w[1](1)*dt*7.85e-3/0.6;
   w[0](0)=tankdensity;
      w[0](2)=tankdensity*pl/rl;
}

void Tofluxclass(int i)
{
      F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
      F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
   F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
   double smax=0;dt=100;
      for (int i =0; i < nend ; ++i){
            smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
      dt=min(dt,dx[i]/smax);
   }
   dt*=CFL;
}


void WtoU(void)
{
      for (int i=0; i<=nend ; ++i){
      u[i](0)=w[i](0);
      u[i](1)=w[i](0)*w[i](1);
      u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
   }
}

void UtoW(void)
{
      for (int i=0; i<=nend ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
   fscanf(inputfile,"%i",&nend);
   fscanf(inputfile,"%lf%lf%lf",&sluglength,&slugmass,&f);
      fclose(inputfile);
```

```
      Flux= new Vector[ngrd+1];
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
   dx =     new double[ngrd];

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i < 5){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

   tankdensity=rl;
   tankpressure=pl;
   bvel=0.;
   slugpos=nend*(xr-xl)/(ngrd*1.0)+sluglength;

}


void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i <= ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](
1),w[i](2),(w[i](2)/0.4/w[i](0)));
   }
   fclose(outfile);
}
```

F.2 SIMWAF.CPP

```
////////////////////////////////////////////////////////////////
//                    SIMwaf.cpp                              //
//       SIMULATION OF LIQUID SLUG PROBLEM                    //
//       WITH WAF METHOD.while you compiling                  //
//       make sure that HLLCWAF.h and Vector.h                //
//       header files and also a data file is in              //
//       appropriate folder.                                  //
////////////////////////////////////////////////////////////////


#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLCwaf.h"
#include "limiterF.h"
#define CFL 0.90

Vector *u,*w;
Vector *Flux, **Flocal;
Vector Fold[4],Foldold[4];
Vector bcr,bcmr,bcml;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
double **c,**q,r[5];
double rold[4], roldold[4];
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

int nend;
Vector tankbdFlux,tankbdFlux1;
double tankdensity,tankpressure,bvel,slugf,slugpos;
double sluglength,slugmass,f;
double dxnorm=0.1;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();
double safediv(double, double);
void updategrid(void);

main()
{
      Setdata();
```

```
      double currtime=0.;
      do{
            settimestep();
            currtime+=dt;
      if(currtime > time)dt=currtime-time;
      for (int i = 0 ; i < nend ; ++i){
            Tofluxclass(i);
         F.Solvexy();
         c[i][1]=dt/dx[i]*F.Sl;c[i][2]=dt/dx[i]*F.Ss;
         c[i][3]=dt/dx[i]*F.Sr;
         q[i][0]=F.rl*1.;     q[i][1]=F.rsl*1.;
         q[i][2]=F.rsr*1.;    q[i][3]=F.rr*1.;
         for (int j= 0 ; j < 4 ; ++j)Flocal[i][j]=F.flux[j]*1.;
      }
      for (int i = 0 ; i < nend ; ++i){
            Flux[i]=(Flocal[i][0]+Flocal[i][3]);
                  if(i > 0 && i < nend-1){
             for(int j=1; j < 4 ; ++j){
            if(c[i][j]>=0.){
                  r[j]=safediv((q[i-1][j]-q[i-1][j-1]),(q[i][j]-
q[i][j-1]));
            }else {
                  r[j]=safediv((q[i+1][j]-q[i+1][j-1]),(q[i][j]-
q[i][j-1]));
            }
            Flux[i]-
=(sign(c[i][j]))*limiter(r[j],fabs(c[i][j]))*(Flocal[i][j]-
Flocal[i][j-1]);
                  }
         }
            Flux[i]=0.5*Flux[i];
      }
      for (int i = 1 ; i < nend ; ++i){
            u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
      }
      UtoW();
      updategrid();
      bc();
      WtoU();
      }while(currtime < time && slugpos < 14.0);
   printf("%lf\t%lf",bvel,w[nend-1](2));
      output();
   getch();
}
void updategrid()
{
      double drivingforce=(w[nend-1](2)-pr)*7.85e-3;
   double
resistingforce=0.;//f*sluglength*0.1*bvel*bvel/8./3.14*1000+slugmass
*9.81*0.08;
      double acc=(drivingforce-resistingforce)/slugmass;
   bvel+=acc*dt;
   slugpos+=bvel*dt;
   dx[nend-1]+=bvel*dt;
      bcmr(1)=2*bvel;
      if(dx[nend-1]>(1.5*dxnorm)){
            dx[nend]=dx[nend-1]-dxnorm;
```

```
            dx[nend-1]=dxnorm;
            w[nend]=w[nend-1]*1.0;
            ++nend;
        }
}

void bc(void)
{
        switch(bc0){
    case 0: w[0]=w[1]*1.0;break;
    case 1: w[0]=w[1]*bcr;break;
    case 2: w[0]=w[1]*bcr+bcml;break;
    }
        switch(bcn){
    case 0: w[nend]=w[nend-1]*1.0;break;
    case 1: w[nend]=w[nend-1]*bcr;break;
    case 2: w[nend]=w[nend-1]*bcr+bcmr;break;
    }
    tankdensity-=w[1](0)*w[1](1)*dt*7.85e-3/0.6;
    w[0](0)=tankdensity;
        w[0](2)=tankdensity*pl/rl;
}

void Tofluxclass(int i)
{
        F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
        F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
    F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
    double smax=0;dt=100;
        for (int i =0; i < nend ; ++i){
              smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
        dt=min(dt,dx[i]/smax);
    }
    dt*=CFL;
}


void WtoU(void)
{
        for (int i=0; i<=nend ; ++i){
        u[i](0)=w[i](0);
        u[i](1)=w[i](0)*w[i](1);
        u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
    }
}

void UtoW(void)
{
        for (int i=0; i<=nend ; ++i){
        w[i](0)=u[i](0);
        w[i](1)=u[i](1)/u[i](0);
        w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
```

```
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
   fscanf(inputfile,"%i",&nend);
   fscanf(inputfile,"%lf%lf%lf",&sluglength,&slugmass,&f);
      fclose(inputfile);

   Flux= new Vector[ngrd+1];
      Flocal= new Vector*[ngrd+1];
      for (int j = 0; j < ngrd+1; j++)
          Flocal[j] = new Vector[4];  // STEP 2: SET UP THE COLUMNS
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
   dx =     new double[ngrd];
      c = new double*[ngrd];         // STEP 1: SET UP THE ROWS.
      for (int j = 0; j < ngrd+1; j++)
          c[j] = new double[5];  // STEP 2: SET UP THE COLUMNS*/
      q = new double*[ngrd];         // STEP 1: SET UP THE ROWS.
      for (int j = 0; j < ngrd+1; j++)
          q[j] = new double[5];  // STEP 2: SET UP THE COLUMNS*/

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i < 5){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

   tankdensity=rl;
   tankpressure=pl;
   bvel=0.;
   slugpos=nend*(xr-xl)/(ngrd-1.0)+sluglength;

}


void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](
1),w[i](2),(w[i](2)/0.4/w[i](0)));
```

```
   }
   fclose(outfile);
}

double safediv(double a, double b)
{
if(b < 1.e-10){
if(a < 1.e-10 && a > -1.e-10){return 1.;
}else{if(a > 0)return 100000.;
        else return -1000000;
     }
}
return a/b;
}
```

F.3 SMSL.CPP

```
////////////////////////////////////////////////////////////////
//                      SMSL.cpp                              //
//       SIMULATION OF LIQUID SLUG PROBLEM                    //
//       WITH muscl METHOD Via limiter slope.                 //
//       while you compiling                                  //
//       make sure that HLLC.h and Vector.h                   //
//       header files and also a data file is in              //
//       appropriate folder.                                  //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLC.h"
#define CFL 0.2
#define W 0.0
#define beta 1.0

Vector *u,*w;
Vector *Flux;
Vector bcr,bcmr,bcml;
Vector delta,dplus,dminus;
Vector *ubarleft,*ubarright,uleft,uright,uold;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

int nend;
Vector tankbdFlux,tankbdFlux1;
double tankdensity,tankpressure,bvel,slugf,slugpos;
double sluglength,slugmass,f;
double dxnorm=0.1;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();
void updategrid(void);


main()
{
Vector Dummy;
```

```
      Setdata();
      double currtime=0.;
      do{
            settimestep();
            currtime+=dt;
      uold  = u[0]*1.0;
      for (int i = 1 ; i <= nend ; ++i){
         dplus=(u[i+1]-u[i]);
         dminus=(u[i]-u[i-1]);
         for(int j=0; j < 3; ++j){
            if (dplus(j)>0.){

      delta(j)=max(0,max(min(beta*dminus(j),dplus(j)),min(dminus(j),
beta*dplus(j))));
            }else{

      delta(j)=min(0,min(max(beta*dminus(j),dplus(j)),max(dminus(j),
beta*dplus(j))));
            }
         }
         uleft   =(u[i]*1.0)-0.5*delta;
         uright  =(u[i]*1.0)+0.5*delta;
         Dummy(0)=uleft(1)-uright(1);
         Dummy(1)=(0.8*uleft(1)*uleft(1)/uleft(0)+0.4*uleft(2));
         Dummy(1)-
=(0.8*uright(1)*uright(1)/uright(0)+0.4*uright(2));
         Dummy(2)=(1.4*uleft(1)*uleft(2)/uleft(0)-
0.2*uleft(1)*uleft(1)*uleft(1)/uleft(0)/uleft(0));
         Dummy(2)-=(1.4*uright(1)*uright(2)/uright(0)-
0.2*uright(1)*uright(1)*uright(1)/uright(0)/uright(0));
                  uleft+=(0.5*dt/dx[i])*Dummy;
                  uright+=(0.5*dt/dx[i])*Dummy;
         if(i==nend)uleft=u[nend]*1.0;
//         if(i>1){
                        F.ul=uold(1)/uold(0);F.rl=uold(0);
            F.pl=(uold(2)-0.5*F.ul*F.ul*F.rl)*0.4;
                        F.ur=uleft(1)/uleft(0);F.rr=uleft(0);
                     F.pr=(uleft(2)-0.5*F.ur*F.ur*F.rr)*0.4;
                        F.Ul=uold;F.Ur=uleft;
            F.Solvexy();
            Flux[i-1]=F.flux*1.;
//          }
         uold=uright*1.;
//          Tofluxclass(i);
      }
      for (int i = 1 ; i < nend ; ++i){
            u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
      }
      UtoW();
      updategrid();
      bc();
      WtoU();
      }while(currtime < time && slugpos < 14.0);
   printf("%lf\t%lf",bvel,w[nend-1](2));
      output();
   getch();
}
```

```
void updategrid()
{
      double drivingforce=(w[nend-1](2)-pr)*7.85e-3;
   double
resistingforce=0.;//f*sluglength*0.1*bvel*bvel/8./3.14*1000+slugmass
*9.81*0.08;
      double acc=(drivingforce-resistingforce)/slugmass;
   bvel+=acc*dt;
   slugpos+=bvel*dt;
   dx[nend-1]+=bvel*dt;
      bcmr(1)=2*bvel;
      if(dx[nend-1]>(1.5*dxnorm)){
            dx[nend]=dx[nend-1]-dxnorm;
         dx[nend-1]=dxnorm;
         w[nend]=w[nend-1]*1.0;
         ++nend;
      }
}

void bc(void)
{
      switch(bc0){
   case 0: w[0]=w[1]*1.0;break;
   case 1: w[0]=w[1]*bcr;break;
   case 2: w[0]=w[1]*bcr+bcml;break;
   }
      switch(bcn){
   case 0: w[nend]=w[nend-1]*1.0;break;
   case 1: w[nend]=w[nend-1]*bcr;break;
   case 2: w[nend]=w[nend-1]*bcr+bcmr;break;
   }
   tankdensity-=w[1](0)*w[1](1)*dt*7.85e-3/0.6;
   w[0](0)=tankdensity;
      w[0](2)=tankdensity*pl/rl;
}

void Tofluxclass(int i)
{
      F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
      F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
   F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
   double smax=0;dt=100;
      for (int i =0; i < nend ; ++i){
            smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
      dt=min(dt,dx[i]/smax);
   }
   dt*=CFL;
}


void WtoU(void)
{
```

```
      for (int i=0; i<=nend ; ++i){
      u[i](0)=w[i](0);
      u[i](1)=w[i](0)*w[i](1);
      u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
   }
}

void UtoW(void)
{
      for (int i=0; i<=nend ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
   fscanf(inputfile,"%i",&nend);
   fscanf(inputfile,"%lf%lf%lf",&sluglength,&slugmass,&f);
      fclose(inputfile);

      Flux= new Vector[ngrd+1];
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
   dx =     new double[ngrd];

   ubarleft = new Vector[ngrd+1];
   ubarright = new Vector[ngrd+1];

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i < 5){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

   tankdensity=rl;
   tankpressure=pl;
   bvel=0.;
   slugpos=nend*(xr-xl)/(ngrd*1.0)+sluglength;

}
```

```
void output()
{
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](1),w[
i](2));
   }
   fclose(outfile);
}
```

F.4 SIMMUSCL

```
////////////////////////////////////////////////////////////////
//                    SIMMUSCL.cpp                            //
//       SIMULATION OF LIQUID SLUG PROBLEM                    //
//       WITH muscl METHOD Via slope limiter.                 //
//       while you compiling                                  //
//       make sure that HLLC.h and Vector.h                   //
//       header files and also a data file is in              //
//       appropriate folder.                                  //
////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <conio.h>
#define __VECSIZE 3
#include "HLLC.h"
#include "limiterS.h"
#define CFL 0.80
#define W 0.0
#define beta 1.0

Vector *u,*w;
Vector *Flux;
Vector bcr,bcmr,bcml;
Vector delta,dplus,dminus;
Vector *ubarleft,*ubarright,uleft,uright,uold;
double *x, *dx;
double dt,time,ubl,ubr;
double ul,pl,rl,ur,pr,rr,xl,xr;
int ngrd,bc0,bcn;
FILE* inputfile, *outfile;
HLLC F;

int nend;
Vector tankbdFlux,tankbdFlux1;
double tankdensity,tankpressure,bvel,slugf,slugpos;
double sluglength,slugmass,f;
double dxnorm=0.1;

void Setdata(void);
void WtoU(void);
void UtoW(void);
void settimestep(void);
void Tofluxclass(int);
void output(void);
void bc();
double safediv(double, double);
void updategrid(void);

main()
```

```
{
Vector Dummy;
      Setdata();
      double currtime=0.;
      do{
            settimestep();
            currtime+=dt;
            printf("%lf\n",currtime);
      uold  = u[0]*1.0;
      for (int i = 1 ; i <= nend ; ++i){
         dplus=(u[i+1]-u[i]);
         dminus=(u[i]-u[i-1]);
            delta=0.5*((1.-W)*dminus+(1.+W)*dplus);
                  for (int j= 0 ; j < 3 ; ++j){
                        double r=safediv(dminus(j),dplus(j));
            double eta=2./(1-W+(1+W)*r);
            delta(j)*=limiter(r,eta);
          }
          uleft   =(u[i]*1.0)-0.5*delta;
          uright  =(u[i]*1.0)+0.5*delta;
          Dummy(0)=uleft(1)-uright(1);
          Dummy(1)=(0.8*uleft(1)*uleft(1)/uleft(0)+0.4*uleft(2));
          Dummy(1)-
=(0.8*uright(1)*uright(1)/uright(0)+0.4*uright(2));
          Dummy(2)=(1.4*uleft(1)*uleft(2)/uleft(0)-
0.2*uleft(1)*uleft(1)*uleft(1)/uleft(0)/uleft(0));
          Dummy(2)-=(1.4*uright(1)*uright(2)/uright(0)-
0.2*uright(1)*uright(1)*uright(1)/uright(0)/uright(0));
                    uleft+=(0.5*dt/dx[i])*Dummy;
                    uright+=(0.5*dt/dx[i])*Dummy;
//          if(i>1){
                          if(i==nend)uold=u[nend]*1.0;
                          F.ul=uold(1)/uold(0);F.rl=uold(0);
             F.pl=(uold(2)-0.5*F.ul*F.ul*F.rl)*0.4;
                          F.ur=uleft(1)/uleft(0);F.rr=uleft(0);
                       F.pr=(uleft(2)-0.5*F.ur*F.ur*F.rr)*0.4;
                          F.Ul=uold*1.;F.Ur=uleft*1.;
             F.Solvexy();
             Flux[i-1]=F.flux*1.;
//          }
          uold=uright*1.;
        }
        for (int i = 1 ; i < nend ; ++i){
              u[i]+=dt/dx[i]*(Flux[i-1]-Flux[i]);
        }
        UtoW();
        updategrid();
        bc();
        WtoU();
        }while(currtime < time && slugpos < 14.0);
     printf("%lf\t%lf",bvel,w[nend-1](2));
        output();
     getch();
 }
 void updategrid()
 {
        double drivingforce=(w[nend-1](2)-pr)*7.85e-3;
```

```
   double
resistingforce=0.;//f*sluglength*0.1*bvel*bvel/8./3.14*1000+slugmass
*9.81*0.08;
      double acc=(drivingforce-resistingforce)/slugmass;
   bvel+=acc*dt;
   slugpos+=bvel*dt;
   dx[nend-1]+=bvel*dt;
      bcmr(1)=2*bvel;
      if(dx[nend-1]>(1.5*dxnorm)){
            dx[nend]=dx[nend-1]-dxnorm;
         dx[nend-1]=dxnorm;
         w[nend]=w[nend-1]*1.0;
         ++nend;
      }
}

void bc(void)
{
      switch(bc0){
   case 0: w[0]=w[1]*1.0;break;
   case 1: w[0]=w[1]*bcr;break;
   case 2: w[0]=w[1]*bcr+bcml;break;
   }
      switch(bcn){
   case 0: w[nend]=w[nend-1]*1.0;break;
   case 1: w[nend]=w[nend-1]*bcr;break;
   case 2: w[nend]=w[nend-1]*bcr+bcmr;break;
   }
   tankdensity-=w[1](0)*w[1](1)*dt*7.85e-3/0.6;
   w[0](0)=tankdensity;
      w[0](2)=tankdensity*pl/rl;
}

void Tofluxclass(int i)
{
      F.rl=w[i](0);F.ul=w[i](1);F.pl=w[i](2);
      F.rr=w[i+1](0);F.ur=w[i+1](1);F.pr=w[i+1](2);
   F.Ul=u[i];F.Ur=u[i+1];
}

void settimestep()
{
   double smax=0;dt=100;
      for (int i =0; i < nend ; ++i){
            smax=max(smax,fabs(w[i](1))+sqrt(GAMMA*
w[i](2)/w[i](0)));
      dt=min(dt,dx[i]/smax);
   }
   dt*=CFL;
}


void WtoU(void)
{
      for (int i=0; i<=nend ; ++i){
      u[i](0)=w[i](0);
      u[i](1)=w[i](0)*w[i](1);
```

```
      u[i](2)=w[i](0)*w[i](1)*w[i](1)*0.5+w[i](2)*2.5;
   }
}

void UtoW(void)
{
      for (int i=0; i<=nend ; ++i){
      w[i](0)=u[i](0);
      w[i](1)=u[i](1)/u[i](0);
      w[i](2)=(u[i](2)-w[i](0)*w[i](1)*w[i](1)*0.5)*0.4;
   }
}

void Setdata(void)
{
      inputfile=fopen("godunovr.dat","r");
      fscanf(inputfile,"%lf%lf%lf%lf%lf%lf%i%lf", &ul, &pl, &rl,
&ur, &pr, &rr, &ngrd, &time);
      fscanf(inputfile,"%lf%lf%i%i",&xl, &xr, &bc0, &bcn);
   if(bc0==2)fscanf(inputfile,"%lf", &ubl);
   if(bcn==2)fscanf(inputfile,"%lf", &ubr);
   fscanf(inputfile,"%i",&nend);
   fscanf(inputfile,"%lf%lf%lf",&sluglength,&slugmass,&f);
      fclose(inputfile);


      Flux= new Vector[ngrd+1];
   u  =     new Vector[ngrd+1];
   w  =     new Vector[ngrd+1];
   x  =     new double[ngrd+1];
   dx =     new double[ngrd];

   ubarleft = new Vector[ngrd+1];
   ubarright = new Vector[ngrd+1];

   for (int i=0; i <= ngrd ; ++i){
      x[i] = xl + (xr - xl)*i/((ngrd-1)*1.);
      if(i<ngrd)dx[i]=(xr - xl)/((ngrd-1)*1.);
      if( i < 5){w[i](0)=rl;w[i](1)=ul;w[i](2)=pl;}
      else{w[i](0)=rr;w[i](1)=ur;w[i](2)=pr;}
   }
   WtoU();

   bcr(0)=1.0;bcr(1)=-1.0;bcr(2)=1.0;
   bcmr(0)=0.0;bcmr(1)=2*ubr;bcmr(2)=.0;
   bcml(0)=0.0;bcml(1)=2*ubl;bcml(2)=.0;

   tankdensity=rl;
   tankpressure=pl;
   bvel=0.;
   slugpos=nend*(xr-xl)/(ngrd*1.0)+sluglength;

}


void output()
{
```

```
      outfile=fopen("godunovr.out","w");
   for(int i = 0;i < ngrd ; ++i){

      fprintf(outfile,"%lf\t%lf\t%lf\t%lf\n",x[i],w[i](0),w[i](1),w[
i](2));
   }
   fclose(outfile);
}

double safediv(double a, double b)
{
if(b < 1.e-10){
if(a < 1.e-10 && a > -1.e-10){return 1.;
}else{if(a > 0)return 100000.;
        else return -1000000;
     }
}
return a/b;
}
```

APPENDIX G

DATA ACQUISITION SOFTWARE CODES

G.1 grp.cpp

```
////////////////////////////////////////////////////
//    this program takes data at 20 input         //
//    channels from ADVANTECH PCL-812             //
//    A/D D/A card at every 0.3 seconds and       //
//    displays simultaneously.                    //
//    use this program at diagnosis level         //
//    note that this program uses pcl812          //
//    device driver for C++. you can obtain       //
//    this file at ADVANTECH's web site           //
//    www.advantech.tw.com                        //
////////////////////////////////////////////////////

#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <stdlib.h>

unsigned int param[60];       /* If two boards installed, need to
declare

      the second parameter array                */
unsigned int data[6400];       /* Conversion data buffer
*/
unsigned int far * dat;

void check(void);
void cardinit(int,int,int,int);
void channel();

extern pcl812(int, unsigned int *);

main()
{
```

```
cardinit(100,0,19,20);
check();
channel();
getch();
}

void channel()
{
char dummy[30];
int x=getmaxx(),y=getmaxy();
do{
            pcl812(5, param);          /* Func 5 : "N" times of A/D
trigger     */
            if (param[45] != 0) {
                        printf(" A/D SOFTWARE DATA TRANSFER FAILED
!");
                        exit(1);
                        }
            for(int i=0;i<=1;++i){
            sprintf(dummy,"  %i",data[i]);
            outtextxy(x/4+30,y/10*(i+0.5),dummy);
            }
            delay(1000);
}while(getch());
}


void cardinit(int pacer_rate,int start_ch, int stop_ch,int sampleno)
{
            unsigned int i;
            float        DataBuf;
            char         y;

            printf("\n");
            printf("This demo program needs PCL-812 jumper
setting");
            printf(" as following: \n");
            printf("I/O PORT BASE ADDRESS (SW1)                    :
HEX 220 \n");
            printf("A/D INPUT RANGE (812 SW2)                      :
+/- 5V\n");
            printf("TRIGGER MODE (JP1)                             :
INTERNAL\n");
            printf("IRQ LEVEL JP4)                                 :
X \n");
            printf("Is the setting correct ?(Y/N) ");
            y=getch();
            printf("\n\n");
            if ((y != 'Y') && (y != 'y' ))
                 {
                  printf("Set the PCL-812 card to correct
configuration before \n");
                  printf("running this program.\n");
                  exit(0);
                  }

            printf("ub1");getch();
```

```
            clrscr() ;

            dat = data;
            param[0]  = 0;            /* Board number
*/
            param[1]  = 0x220;        /* Base I/O address
*/
            param[5]  = 20000/pacer_rate; /* Pacer rate = 2M / (50
* 100) = 400 Hz  */
            param[6]  = 100;
            param[7]  = 0;            /* Trigger mode, 0 : pacer
trigger        */
            param[10] = FP_OFF(dat);  /* Offset of A/D data buffer
A            */
            param[11] = FP_SEG(dat);  /* Segment of A/D data buffer
A           */
            param[12] = 0;            /* Data buffer B address, if
not used,    */
            param[13] = 0;            /* must set to 0.
*/
            param[14] = sampleno;     /* A/D conversion number
*/
            param[15] = start_ch;     /* A/D conversion start
channel          */
            param[16] = stop_ch;      /* A/D conversion stop
channel           */
            param[17] = 0;            /* Overall gain code, 0 : +/-
5V          */

             /* param[18] = FP_OFF(gain_array);
                  param[19] = FP_SEG(gain_array);  */

             /* param[45] : Error code
                  param[46] : Return value 0
                  param[47] : Return value 1 */

            printf("ub2");getch();
             pcl812(3, param);        /* Func 3 : Hardware
initialization       */
             if (param[45] != 0) {
                        printf(" DRIVER INITIALIZATION FAILED !");
                        exit(1);
                        }
            printf("ub3");getch();

             pcl812(4, param);        /* Func 4 : A/D
initialization          */
             if (param[45] != 0) {
                        printf(" A/D INITIALIZATION FAILED !");
                        exit(1);
                        }

}


void check(void)
```

```
{
      int gdriver = DETECT, gmode, errorcode;
      struct viewporttype viewinfo;
      char dummy[30];
      initgraph(&gdriver, &gmode, "");
      errorcode = graphresult();
      if (errorcode != grOk) {  /* an error occurred */
            printf("Graphics error: %s\n",
grapherrormsg(errorcode));
            printf("Press any key to halt:");
            getch();
 //         exit(1);               /* terminate with an error code
*/
      }
            cleardevice();
      int x=getmaxx(),y=getmaxy();
//    setbkcolor(BLUE);
      setfillstyle(1,BLUE);
      setcolor(WHITE);
      for(int i=0;i<10;++i){
      bar(x/4,y/10*i,x/2-5,y/10*(i+1)-5);
      rectangle(x/4+5,y/10*i+5,x/2-10,y/10*(i+1)-10);
      sprintf(dummy,"channel %i",i);
      outtextxy(30,y/8*(i+0.5),dummy);
      }
      for(i=0;i<10;++i){
      bar(x/4*3,y/10*i,x-5,y/10*(i+1)-5);
      rectangle(x/4*3+5,y/10*i+5,x-10,y/10*(i+1)-10);
      sprintf(dummy,"channel %i",i+8);
      outtextxy(x/2+30,y/8*(i+0.5),dummy);
      }
//    getch();
//    closegraph();
}
```

G.2 DAS.cpp

```
////////////////////////////////////////////////////
//    this program takes data at desired input    //
//    channels from ADVANTECH PCL-812             //
//    A/D D/A card at desired number and frequency,//
//    then displays them graphically.             //
//    do not forget that program waits 3 secs     //
//    after you hit go! key.                      //
//    note that this program uses pcl812          //
//    device driver for C++. you can obtain       //
//    this file at ADVANTECH's web site           //
//    www.advantech.tw.com                        //
////////////////////////////////////////////////////

#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>
#include <math.h>
#include <string.h>

#include "cardinp.h"

//extern pcl818(int, unsigned int *);
unsigned int param[60];       /* If two boards installed, need to
declare

      the second parameter array                */
unsigned int huge data[64000];       /* Conversion data buffer
*/
unsigned int far * dat;

void acquision(void);
void graph(void);
extern pcl812(int, unsigned int *);
main()
{

             printf("\n");
             printf("This demo program needs PCL-812 jumper
setting");
             printf(" as following: \n");
             printf("I/O PORT BASE ADDRESS (SW1)                    :
HEX 220 \n");
```

```
            printf("A/D INPUT RANGE (812 SW2)                    :
+/- 5V\n");
            printf("TRIGGER MODE (JP1)                           :
INTERNAL\n");
            printf("IRQ LEVEL JP4)                               :
X \n");
            printf("Is the setting correct ?(Y/N) ");
            y=getch();
            printf("\n\n");
            if ((y != 'Y') && (y != 'y' ))
                 {
                  printf("Set the PCL-812 card to correct
configuration before \n");
                  printf("running this program.\n");
                  exit(0);
                  }


inputscr();
acquision();
graph();
getch();
return 0;
}



void acquision(void)
{
            unsigned int i;
            float        DataBuf;
            char         y;
            clrscr() ;

            dat = data;
            param[0]  = 0;            /* Board number
*/
            param[1]  = 0x220;        /* Base I/O address
*/
            param[5]  = 20000/pacer_rate; /* Pacer rate = 2M / (50
* 100) = 400 Hz  */
            param[6]  = 100;
            param[7]  = 0;            /* Trigger mode, 0 : pacer
trigger       */
            param[10] = FP_OFF(dat);  /* Offset of A/D data buffer
A           */
            param[11] = FP_SEG(dat);  /* Segment of A/D data buffer
A          */
            param[12] = 0;            /* Data buffer B address, if
not used,    */
            param[13] = 0;            /* must set to 0.
*/
            param[14] = sampleno;     /* A/D conversion number
*/
            param[15] = start_ch;     /* A/D conversion start
channel          */
```

```
            param[16] = stop_ch;       /* A/D conversion stop
channel             */
            param[17] = 0;             /* Overall gain code, 0 : +/-
5V          */

            /* param[18] = FP_OFF(gain_array);
                  param[19] = FP_SEG(gain_array);  */

            /* param[45] : Error code
                  param[46] : Return value 0
                  param[47] : Return value 1 */

            pcl812(3, param);          /* Func 3 : Hardware
initialization       */
            if (param[45] != 0) {
                        printf(" DRIVER INITIALIZATION FAILED !");
                        exit(1);
                        }

            pcl812(4, param);          /* Func 4 : A/D
initialization            */
            if (param[45] != 0) {
                        printf(" A/D INITIALIZATION FAILED !");
                        exit(1);
                        }

            delay(3000);
      // wait three seconds
            sound(2000);delay(200);nosound();  // beginning signal
of acq. of .2 sec
            pcl812(5, param);          /* Func 5 : "N" times of A/D
trigger      */
            if (param[45] != 0) {
                        printf(" A/D SOFTWARE DATA TRANSFER FAILED
!");
                        exit(1);
                        }
            printf("here1 %i",param[14]);
            for (i = 0; i < param[14]; i+=stop_ch-start_ch+1)    /*
Display data                  */
                 {
                 fprintf(out,"%5d ",i);
                  for (j=0;j<stop_ch-start_ch+1;++j){
                        DataBuf = data[i+j] & 0xFFF;
                        DataBuf =( (5 - (-5)) * DataBuf / 4096) + (-
5);
                        /*
                              (5 - (-5)) : A/D input range (-5V  to
5V)
                        4096       : Full scale 12 bit A/D data
                                   DataBuf    : A/D input data
                                   (-5)       : A/D input range "-5" V
                        */
                        fprintf(out,"%1.8f ", DataBuf);
                   }
                        fprintf(out,"\n");
                  }
```

```
                    printf("here");

}


int
xbeg=50,xend=550,ybeg=40,yend=440,Mthick_x=50,mthick_x=10,Mthick_y=5
0,mthick_y=25;
double x1=0.,xe=1000,y1=0.,ye=4096.;
void graph(void)
{
                    double constx,consty;
              int chnum;
       char num[25];
       int thicknum=500/mthick_x,thpos=xbeg;
       int bkcol=WHITE,x,y;
       int gdriver = DETECT, gmode, errorcode;
       struct viewporttype viewinfo;
       int midx, midy, ht, i,j;
       chnum=stop_ch-start_ch+1;
       xe=sampleno;
//     char topstr[80], botstr[80], clipstr[80],msg[100];

       /* initialize graphics and local variables */
       initgraph(&gdriver, &gmode, "");

  /* read result of initialization */
       errorcode = graphresult();
       if (errorcode != grOk) {  /* an error occurred */
             printf("Graphics error: %s\n",
grapherrormsg(errorcode));
             printf("Press any key to halt:");
             getch();
             exit(1);               /* terminate with an error code
*/
       }
//     else{printf("ok!");}

        /* clear the screen */
             cleardevice();

             /* select a new background color */
 /*          setbkcolor(bkcol);
         x = getmaxx() / 2;
         y = getmaxy() / 2;


         /* output a messsage */
                    setcolor(BLUE);
//           sprintf(msg, "Background color: %d", bkcol);
//           outtextxy(x, y, msg);
             line(xbeg,ybeg,xbeg,yend);
             line(xbeg,yend,xend,yend);
             moveto(100,100);

             for(i=0;i<=thicknum;++i){
```

```
            line(thpos,yend-2,thpos,yend+2);
            thpos+=mthick_x;
            }
            settextjustify(1,1);
            thicknum=500/Mthick_x,thpos=xbeg;
            for(i=0;i<=thicknum;++i){
            line(thpos,yend-5,thpos,yend+5);
            gcvt((1.*x1+(xe-x1)/thicknum*i),5,num);
            outtextxy(thpos,yend+15,num);
            thpos+=Mthick_x;
            }

            thicknum=400/mthick_y,thpos=yend;
            for(i=0;i<=thicknum;++i){
            line(xbeg+2,thpos,xbeg-2,thpos);
            thpos-=mthick_y;
            }

            settextjustify(2,1);
            thicknum=400/Mthick_y,thpos=yend;
            for(i=0;i<=thicknum;++i){
            line(xbeg+5,thpos,xbeg-5,thpos);
            gcvt((1.*y1+(ye-y1)/thicknum*i),5,num);
            outtextxy(xbeg-5,thpos,num);
            thpos-=Mthick_y;
            }


//    this part draws the output desired on the graph
//          note it does not draw the values below the minimum y
value!!!
            constx=chnum*1./(xe-x1)*500;consty=1./(ye-y1)*400;

            for(j=0;j<chnum;++j){
            x=(j-x1)*constx+50;
            y=440-(data[j]-y1)*consty;
            setcolor(i+1);
            moveto(x,y);
            for(i=j+chnum;i<=sampleno;i+=chnum){
                  x=(i-x1)*constx+50;
                  y=440-(data[i]-y1)*consty;
                  lineto(x,y);
            }
            }

            getch();

      /* clean up */
      closegraph();
```