DISCRETIZATION **E**RROR **E**STIMATION **U**SING **T**HE **M**ETHOD **O**F **N**EARBY **P**ROBLEMS: **O**NE-**D**IMENSIONAL **C**ASES

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee.

_____

ANIL RAJU

Certificate of Approval:

_____	_____
Anwar Ahmed Associate Professor Aerospace Engineering	Christopher. J. Roy, Chair Assistant Professor Aerospace Engineering
_____	_____
Brian Thurow Assistant Professor Aerospace Engineering	Stephen L. McFarland Dean Graduate School

DISCRETIZATION **E**RROR **E**STIMATION **U**SING **T**HE **M**ETHOD **O**F **N**EARBY **P**ROBLEMS: **O**NE-**D**IMENSIONAL **C**ASES

Anil Raju

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama

December 16, 2005.

DISCRETIZATION **E**RROR **E**STIMATION **U**SING **T**HE **M**ETHOD **O**F **N**EARBY **P**ROBLEMS: **O**NE-**D**IMENSIONAL **C**ASES

Anil Raju

_____

Signature of Author

_____

Date

VITA

Anil Raju, son of Mr. and Mrs. Thomas Raju, was born on September 3, 1979, in Trivandrum, India. He graduated with a Bachelors degree in Industrial Engineering from University of Kerala, Trivandrum, India in May 2002. He joined the graduate program in Aerospace Engineering at Auburn University in the fall of 2003.

THESIS ABSTRACT

DISCRETIZATION **E**RROR **E**STIMATION **U**SING **T**HE **M**ETHOD **O**F **N**EARBY **P**ROBLEMS: **O**NE-**D**IMENSIONAL **C**ASES

Anil Raju

Master of Science, December 16, 2005

(Bachelor of Technology, Industrial Engineering, India, May 2002)

91 TYPED PAGES

Directed by Dr. Christopher J. Roy

Discretization error is defined as the difference between the solution of the discretized equation and the exact solution of the original partial differential equation. There are two main goals in this study. The first goal is to use of the method of nearby problems to generate exact solutions to realistic problems so that we can asses the performance of discretization error estimators can be assessed. The second goal is to develop and use method of nearby problems itself as an error estimator. Different polynomial curve fitting techniques are examined and fifth-order Hermite splines are identified as the best approach for the method of nearby problems. Steady-state Burgers equation and a modified form of Burgers equation are used as test cases. The analytical curve fits are then the exact solution to a problem nearby the original problem. Results are presented for Burgers equation corresponding to a viscous shock wave for Reynolds

numbers of 8 and 64, as well as for a modified version of Burgers equation with a variable viscosity at a nominal Reynolds number of 64. Various discretization error estimators are evaluated for the original Burgers equation, the nearby problem, and the modified version of Burgers equation which includes a nonlinear viscosity term. It is also observed that the method of nearby problems itself performs well as a discretization error estimator even on coarse meshes.

ACKNOWLEDGEMENT

I would like to thank the Sandia National Laboratories for financial support extended to this project.

I am sure this is a great opportunity for me to acknowledge several key people I came across during my educational career in Aerospace Engineering. I would like to thank my advisor Dr. Christopher J. Roy for honoring me with the project ‘Discretization error estimation using the method of nearby problems: one-dimensional cases’, and allowing me to work with him. I will be always thankful to him for guiding me and supervising my work. I will also be obligated to him for the resources he spent on me. His research and work practices will certainly have significant influence on my career.

I want to express sincere gratitude towards my Masters committee members Dr.Anwar Ahmed and Dr. Brian Thurow, for their comments and suggestions on my Masters thesis.

I would like to thank Dr. Matthew Hopkins of Sndia National Laboratories for all his comments during my work on this project.

Finally, everything I own is dedicated to my Mom, sister Simi, brother-in-law John, nephew Jeff and last but not the least, Sherin. They mean everything to me.

Anil Raju.

Date: December 16, 2005.

aniraju.c@gmail.com,anilraju2524@yahoo.com

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

1.1 *Background on computational fluid dynamics*

Computational fluid dynamics (CFD) is a term given to a variety of numerical techniques applied to solve the equations that govern fluid flow. Before computers, theory and experiments were the only methods for gaining insight into physical phenomena. Most of these physical phenomena can be modeled using differential equations. The ideal approach would be to solve these equations via analytical techniques and obtain exact solutions. But in most cases, these equations are complex and therefore difficult to solve analytically. For such cases we have to resort to approximate solutions. Using computer simulations is one method to obtain these approximate solutions.

The seventeenth century saw the growth of experimental fluid dynamics in Europe. Experimental fluid dynamics is considered as the first approach [1] in the study and development of fluid dynamics. The eighteenth and nineteenth centuries saw the development of the second approach which is theoretical fluid dynamics. The latter part of the twentieth century saw the development of CFD which is the third approach. The growth of CFD can be attributed to the development of high-speed computers and accurate numerical algorithms to solve various problems. CFD complements the other two approaches of pure theory and pure experiment. There is still a need for theory and

experiments, and the future of fluid dynamics depends on the balance of all three approaches as shown in Fig 1.1.

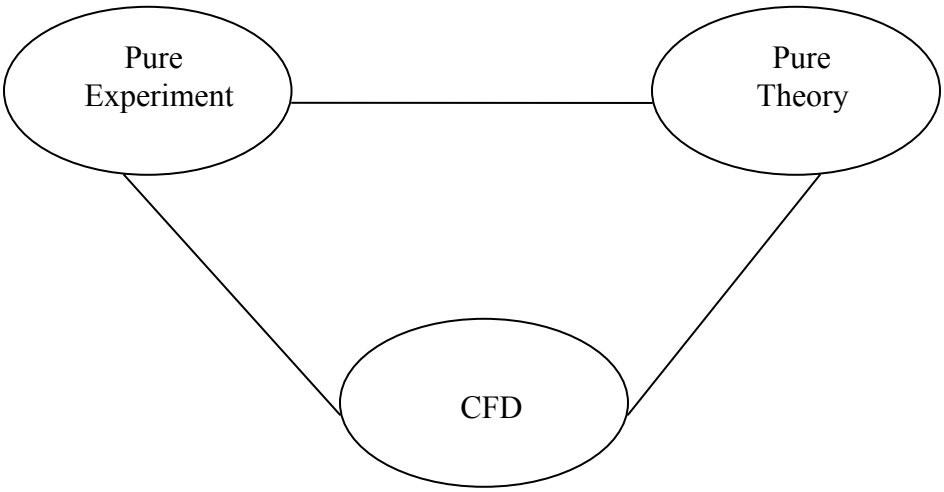

Fig 1.1 Three approaches to fluid dynamics

CFD has established itself as a research tool. Moreover, CFD is now establishing its use as a design tool as well. CFD can be used to predict the presence of vortices in the flow over vehicles [1] and by studying the behavior of these vortices and their interaction, one can come up with an optimal aerodynamic design for the vehicle. This is just one of many uses of CFD as a design tool.

CFD is being used today in a wide variety of areas [1], for example, engine and automobile applications. By assessing the flow over the body of the vehicle, an aerodynamic shape can be determined. Today, automotive engineers use CFD to study details of flow in engines. CFD can be applied in industrial manufacturing as well, for example, to calculate the flow field in a mold filled by liquid metal or polymer. Civil engineering uses CFD to tackle problems related to lakes, rivers, estuaries, etc. CFD is

also used in other applications relating to heating, air conditioning, and general air circulation in buildings.

1.2. ***Verification and validation in CFD***

Verification is the process of determining whether the implementation of the model, accurately represents the original problem of interest [2]. Validation is the process of determining the extent to which the model accurately represents reality. In other words, verification checks whether the model is solved correctly while validation ensures that the correct model is solved. In yet another way, verification can be described as dealing with the mathematical correctness of the solution, while validation deals with the physical correctness of the model. Both verification and validation are compared with a reference standard. In the case of verification the standard is exact solution of the partial differential equation while in the case of validation, the standard is real world observations.

Verification is a two step process [3]. The first step is code verification where the computer code is verified and made free of unacknowledged errors, such as errors due to mistakes in code or inconsistent numerical schemes. The second step, solution verification, is the step in which the acknowledged errors such as round-off errors or discretization errors are assessed.

Code verification [3] can be broadly classified into two areas: numerical algorithm verification and software quality assurance practices. Some types of algorithm testing are the method of manufactured solutions, benchmark solutions, iterative convergence tests, conservation tests, symmetry tests, and so forth. Software quality

assurance [4] involves the entire software development process: monitoring the process and trying to improve it, making sure that all standards and procedures are followed, and finally making sure that all the defects are found out and properly dealt with. Some of the software quality analysis tools are static analysis, dynamic testing, and formal testing.

Solution verification has three main aspects [3]. The first aspect, verification of input data, makes sure that the data that is provided is correct. The second aspect, numerical error estimation, calculates the acknowledged errors in simulation. The third aspect, verification of output data, makes sure that the correct post-processing steps are used.

1.3. ***Sources of numerical error***

In addition to the errors that can come in during the development of the solution algorithm, there are some acknowledged errors that occur in every computational simulation. These errors are called numerical errors.

The main types of numerical errors are round-off errors, iterative errors, and discretization errors. Round-off errors [3] occur due to finite arithmetic in digital computers. An example of round-off error is $3.0 \times (1.0/3.0) = 0.9999999$ for single precision. Here we see that the computation of $1.0/3.0$ in single precision gives 0.3333333, which leads to the final result of 0.9999999 as opposed to 1.0. Round-off error can be very important in the case of ill-conditioned problems or time accurate simulations where a large number of time steps can result in error accumulation. The way by which the round-off error can be reduced is by using more digits in the computation.

For example, round-off error can be reduced by using double precision instead of single precision.

Iterative error [5] is the difference between the current iterative solution and the exact solution of the discretized equations. When the governing equations for fluid dynamics are discretized, they often result in a set of non-linear equations. The usual procedure that is followed to solve these equations is to first linearize them and then solve them using an iterative method. To stop the iterative process, a convergence criterion has to be introduced. The iteration goes on until the residual, which is calculated by substituting the current iterative solution into the discrete equations, is less than this convergence criterion. If the iterative process is run until the residual is as small as possible (i.e. machine zero), then the iterative error will be minimized. In this research, a small iterative error was desired and so a convergence criterion of 10^{-14} was used.

Discretization error is defined as the solution of the algebraic system of equations which is obtained by discretizing the conservation equation and the difference between the exact solutions of the conservation equations. It is important to estimate the discretization error before the CFD predictions can be compared with the experimental data. The first step to solve a set of governing equations numerically is to discretize them. Discretization is the process of converting the original partial differential equations to an algebraic set of equations. This algebraic set of equations is then solved on a discrete mesh to obtain numerical solutions. These solutions are approximate and are generally different from the exact solution of the governing equations. This difference is the discretization error. If the discretization approach is consistent, then the discretization error will decrease as the mesh is refined.

1.4. ***Discretization error estimators***

There are a number of ways to estimate the discretization error. Richardson extrapolation involves the computation of numerical solutions on two or more meshes. Solutions on these different meshes are then used to compute a higher-order estimate of the exact solution. This estimate of the exact solution can then be used to estimate the discretization error. There are certain assumptions that are used in Richardson extrapolation. The solution is assumed to be smooth, uniform meshes are assumed and the higher order terms are neglected. The discretization error [3] can be written as

$$DE_k = f_k - f_{exact} = g_1 h_k + g_2 h_k^2 + g_3 h_k^3 + Higher\,Order\,Terms \tag{1.1}$$

where, f_k is the discrete solution on mesh k, f_{exact} is the exact solution to the partial differential equation, g_i is the coefficient of the i^{th} order term, and h is the measure of the element size. Consider a second order accurate scheme with solutions on two different meshes h_1 and h_2, with $h_2 = 2h_1$. Neglecting the higher order terms, the discretization error equation can be written as

$$f_1 = f_{exact} + g_2 h_1^2 \tag{1.2}$$

$$f_2 = f_{exact} + g_2 (2h_1)^2 \tag{1.3}$$

Solving these two equations for g_2 and f_{exact} we get

$$f_{exact} = f_1 + \frac{f_1 - f_2}{3} \tag{1.4}$$

In general, if we consider a p^{th} order accurate scheme with solutions on a fine mesh (h_1) and a course mesh (h_2), f_{exact} can be approximated as

$$f_{exact} = f_1 + \frac{f_1 - f_2}{r^p - 1} \tag{1.5}$$

where r is the grid refinement factor given by $r = h_2 / h_1$. Once f_{exact} is estimated, then the relative discretization error (RDE) in the fine grid can be calculated as

$$RDE_1 = \frac{f_1 - f_{exact}}{f_{exact}} \tag{1.6}$$

1.5. *Objective*

The current study concentrates on verification. In particular, we focus on solution verification and use the method of nearby problems (MNP) as an error estimator. We have also used MNP to come up with an exact solution for a problem that is very close to (i.e. nearby) the original problem, and have evaluated various error estimators on those nearby problems.

CHAPTER TWO

BACKGROUND

2.1. *Method of manufactured solutions (MMS)*

MMS [7] provides a general procedure for generating an analytical solution for code verification. It is a general approach to find coding mistakes/bugs or inconsistent algorithms. The goal of MMS is code verification. It involves manufacturing an exact solution to a set of equations which are a modified form of the original partial differential equations. The solution obtained to this set of modified equations is not physically realistic. This method is used only to verify the mathematics involved in solving the original equations, and does not verify the solution obtained by solving the original equations. This procedure is used when the method of exact solutions cannot be used. The method of exact solutions [3] is one in which the numerical solution is compared to an exact solution to the partial differential equation. In the method of exact solution, the discretization error is computed and then the observed order of accuracy is calculated. This observed order of accuracy is compared with the formal order of accuracy. This method [2] is usually not followed for complex cases (geometric complexity, physical complexity, etc.) because of the limited number of exact solutions.

Once the problem of interest is identified then MMS is conducted for code verification. MMS is a five step process [3].

1. The first step is to choose a manufactured solution. There are certain guidelines [8] that should be followed in choosing a manufactured solution.
    - The manufactured solution should be sufficiently smooth so that the theoretical order of accuracy can be matched by the observed order of accuracy on relatively coarse meshes.
    - The solution should exercise all the terms of the governing equation. For example, for the unsteady heat equation, the temperature cannot be chosen as a function which is independent of time.
    - The solution should be such that it has a number of nontrivial derivatives. For example, in the heat equation which is a second-order equation in space, picking temperature as a linear function of the spatial coordinate will not provide a sufficient test.
    - The chosen solution should consist of simple analytic functions like polynomials, trigonometric functions, etc.
    - It is better to avoid exponential growth of the solution in time to avoid confusion with numerical instability.
2. The second step in MMS is to derive the modified governing equation. Here the governing equations are applied to the chosen manufactured solution. This will result in the generation of analytical source terms. These analytic source terms are then added to the governing equations, resulting in a modified form of the original equations.
3. Once the modified equations are obtained, then they are solved numerically on different meshes.

4. The next step is to evaluate the global discretization error, which can be calculated as a global norm of the difference between the numerical solution and the exact solution of the modified equation over the whole domain. The exact solution of the modified equation is nothing but the manufactured solution chosen in step 1.
5. The last step in MMS is to compute the observed order of accuracy. If the exact solution is known, then the observed order of accuracy is calculated as

$$p = \frac{\ln(\frac{DE_2}{DE_1})}{\ln(r)} \tag{2.1}$$

where, DE_2 and DE_1 are the discretization error on the coarse and fine meshes respectively, and r is the grid refinement factor. Once the observed order of accuracy is calculated, it is compared to the formal order of accuracy. If they match, then the code is considered to be free of bugs or mistakes in the discretization. If they do not match, then it suggests that there is some problem in the code.

2.2. *Prior work in MNP*

Standard benchmark problems are often used for testing codes. True solutions of these standard benchmark problems are often known, but it is hard to ascertain the relationship between the behavior of an algorithm on a standard benchmark problem and the behavior of the algorithm on the true problem of interest.

Lee and Junkins [9] described the use of a problem near an original ordinary differential equation (ODE) to serve as a benchmark problem. They constructed a

benchmark problem near the original ODE which exactly satisfied the original ODE with a small, known forcing function. Their work can be summarized as follows:

- Compute a numerical solution on a very refined mesh.
- Generate a polynomial fit for the fine numerical solution by the least squares approximation using Chebyshev polynomials. A global polynomial was used instead of a local polynomial to avoid discontinuities.
- Generate the analytic solution from the global fit. This analytic solution becomes the exact solution of the nearby benchmark problem.
- Use symbolic manipulation to plug the analytic solution into the original problem and generate small source terms.
- Add these small source terms to the original ODE to form the benchmark problem.

Since the exact solution of the benchmark problem was known, it was easy to compute the global error and also to determine optimal integration parameters.

Junkins and Lee [10] later extended the previous methodology to construct exact special-case solutions for hybrid ODE/PDE systems. This hybrid ODE/PDE system was able to serve as a benchmark problem to test approximate solution methods. In their work, they have described a method for coming up with a benchmark problem to determine optimal time integration parameters, while in our work we use method of nearby problems to come up with an exact solution to a nearby problem and also as an error estimator.

Roy and Hopkins [11] examined the generation of exact solutions to problems near the original problem of interest. They studied two examples: fully developed laminar

flow in a channel and a lid-driven cavity. Two codes were used in their work. The SACCARA code was used to establish a refined numerical solution to the original problem while the Premo code was used for the implementation of source terms and generalized boundary conditions. In the first example, the fully developed laminar flow in a channel they employed a third-order polynomial for the polynomial fit and, Mathematica was used to compute the analytic source terms. Once the source terms were computed, they formulated the nearby problem and used the Premo code for the solution of the nearby problem. Computed solutions on various meshes were compared to the analytic solution. The L_2 norms of the source term as a function of the underlying mesh solution are shown in Fig 2.1. It is seen that as the mesh becomes finer (i.e. as h goes to 1), the magnitude of the source terms decrease. Since the magnitude of the source terms is small, the nearby problem that is formulated will be close to the original problem of interest. The method of nearby problems was successfully demonstrated for fully developed laminar flow in a channel.

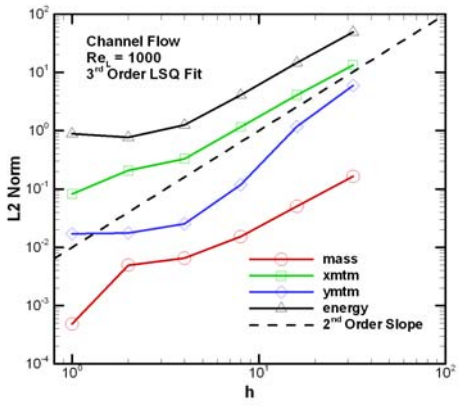

Fig 2.1. L_2 norm of the source term for the third-order polynomial fit (reproduced from Roy and Hopkins [11])

In the second example, a lid-driven cavity, the analytic source terms were generated using a fourth-order least square polynomial based on different underlying mesh refinement levels. To minimize the errors that arise due to singularities at the corners, a truncated domain was used. The L_2 norm of the source term did not get smaller with mesh refinement as is seen in Fig 2.2. The magnitude of the source term was found to be large near the boundaries. For this example, the global polynomial did not capture the solution well.

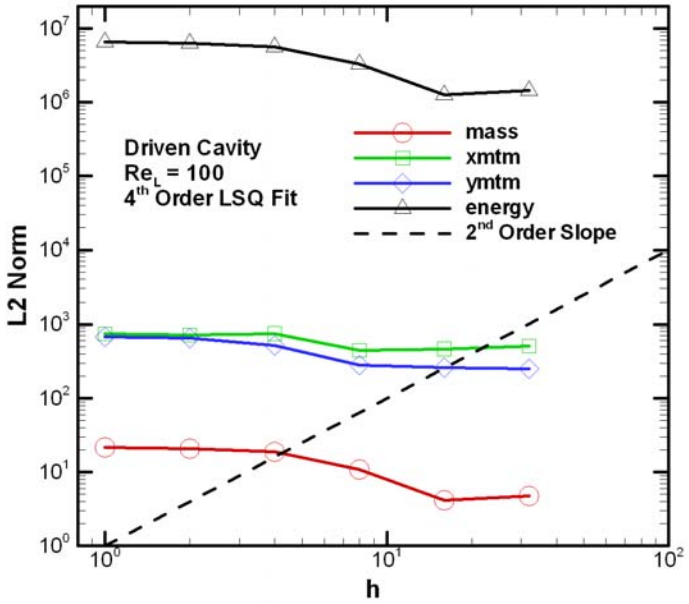

Fig 2.2. L_2 norm of the source term for the fourth-order polynomial fit (reproduced from Roy and Hopkins [11])

CHAPTER THREE

BURGERS EQUATION

3.1. *Introduction to Burgers equation*

Burgers equation is a quasi-linear, one-dimensional, parabolic partial differential equation of the form

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2} \tag{3.1}$$

where *u(x,t)* is a two-dimensional scalar field. As an example it can be assumed as representing velocity as a function of position '*x*' and time '*t*', and 'υ' is the viscosity. A quasi-linear equation is one in which the highest-order derivative occurs linearly. A second-order partial differential equation of the form

$$A\frac{\partial^2 u}{\partial x^2} + 2B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D\frac{\partial u}{\partial x} + E\frac{\partial u}{\partial y} + F = 0 \tag{3.2}$$

is said to be parabolic if the matrix $Z \equiv \begin{bmatrix} A & B \\ B & C \end{bmatrix}$ satisfies the determinant $|Z| = 0$

Burgers equation can be related to shock wave theory. The solutions of Burgers equation can describe the formation and decay of shocks in a compressible fluid. Burgers equation can also be used as a mathematical model for turbulence. The Navier-Stokes equation and Burgers equation are quite similar since both contain a non-linear term and a second-order term multiplied by a small parameter.

3.2. *Solution to Burgers equation*

Benton and Platzmann [12] describe thirty five solutions to Burgers equation. Out of the thirty five solutions we have chosen three that are smooth and real:

1. The steady state solution in dimensionless form (denoted by primes) is given by $u'(x',t') = -2\tanh(x')$. This is a solution of Burgers equation when $\frac{\partial u'}{\partial t} = 0$. It can model a steady shock, and it is smooth, non trivial, and in the real plane. The plot for this solution is given in Fig. 3.1. for a Reynolds number of 8. The abscissa is the spatial coordinate *x*, while the ordinate is the value of *u*.

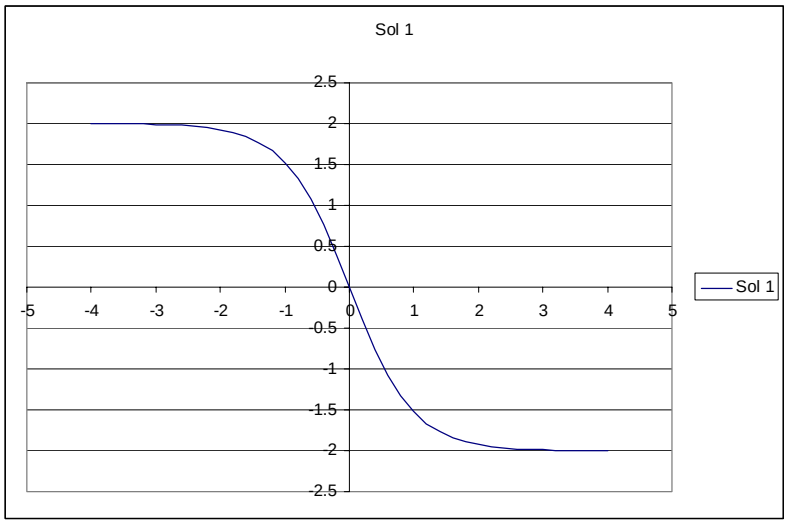

Fig 3.1. Steady state solution of Burgers equation

2. An unsteady solution to Burgers equation is given as $u'(x',t') = -\frac{2\sinh(x')}{\cosh(x') + e^{-t'}}$ and this solution can model the coalescence of two equal, unsteady shocks and is shown in Fig. 3.2 for different time levels.

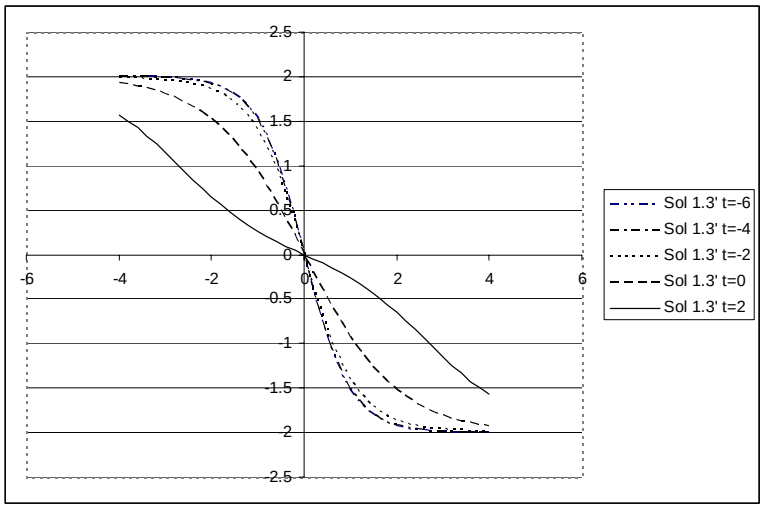

Fig 3.2. Unsteady solution of Burgers equation for shock coalescnese

3. Another unsteady solution to Burgers equation is given as $u'(x',t') = \dfrac{x'/t'}{1+t'^{1/2}\, e^{x'^2/4t'}}$. This solution can model the decay of a solitary pair of unsteady, equal compression and expansion pulses, as shown in Fig. 3.3.

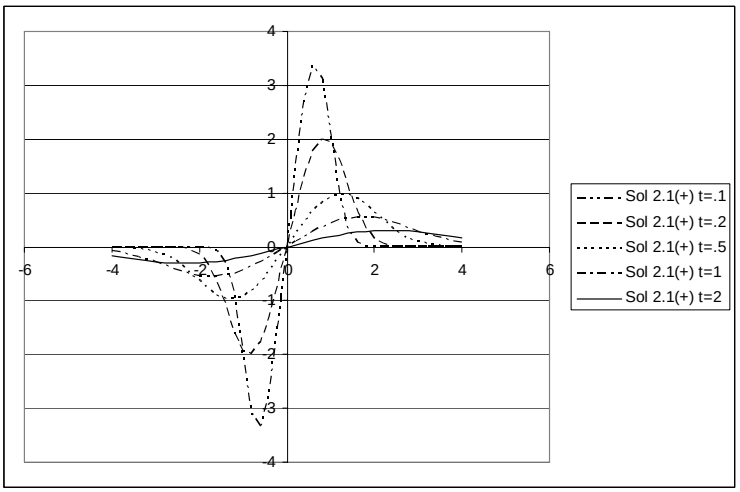

Fig 3.3. Unsteady solution of Burgers equation for pulse decay

3.3. *Conversion to dimensional quantities and scaling factors*

The solution of Burgers equation can be converted to dimensional quantities via transformations given by $x' = x/l,\ t' = \upsilon t/l^2,$ and $u' = ul/\upsilon$. A scaling factor α can also be used to scale the solution. The scaling factor was used in the forms $\bar{x} = x/\alpha,$ $\bar{t} = t/\alpha^2,$ and $\bar{u} = \alpha u$. The value of the scaling factor depends on the Reynolds number used. For a Reynolds number of 8, the scaling factor used was 2. The value of the scaling factor used for a Reynolds number of 64 was 16.

CHAPTER FOUR

METHOD OF NEARBY PROBLEMS

The method of nearby problems (MNP) is used to generate exact solutions to realistic problems, which in turn allows assessment of how discretization error estimators will perform on the original problem of interest. MNP can also be used as an *a posteriori* error estimation technique [13]. It is based on constructing a problem close to the original problem called the nearby problem. This nearby problem has an exact solution, and also is representative of the original problem if the source term is sufficiently small. The nearby problem is numerically solved just like the original problem. Since the exact solution of the nearby problem is known, the error in its numerical solution can be evaluated exactly. This information can then be used to estimate the discretization error in the original problem.

4.1 *MNP as an evaluator of discretization error estimators*

The method of nearby problems (MNP) involves five steps. These steps can be summarized as given below:

- Establish an accurate numerical solution
- Generate an analytical curve fit for the above accurate numerical solution
- Generate analytic source terms
- Numerically solve the nearby problem (original problem plus analytical source term)

- Evaluate the discretization error in the nearby problem

An explanation to all of these five steps is given below.

Accurate numerical solution

Once the problem of interest is identified, the first step is to discretize the problem and come up with an accurate numerical solution.

Analytic curve fit

Once an accurate numerical solution is computed, the second step involves generating an analytic curve fit to this numerical solution. A curve fitting tool is used to generate this curve fit. It should be kept in mind that the tool used for the curve fitting should provide a particular level of continuity which is problem dependent. Once the curve fit is generated, it should be examined to see how good the fit approximates the numerical solution. This analytic curve fit will serve as the exact solution to the nearby problem.

Generation of analytic source terms

The nearby problem differs from the original problem by a (hopefully) small source term. This source term is obtained by operating the original equation on the analytic curve fit obtained from the previous step. In the limit as the size of the source terms approaches zero, the nearby problem approaches the original problem. The nearness of the nearby problem to the original problem can be judged by calculating the magnitude of the source term. A more rigorous assessment of the nearness of the nearby problem is presented in [13] for ordinary differential equations. Such an assessment for Burgers equation, a nonlinear PDE is beyond the scope of this work.

Numerical solution to the nearby problem

The next step involves discretization of the nearby problem and then computation of the numerical solution on a series of meshes. For a consistent numerical scheme and sufficiently refined meshes, the formal order of accuracy of the scheme should be observed, even on the perturbed equations. In general, the discretization error should drop as $1/r^p$, where r is the grid refinement factor and p the formal order of accuracy. In order to examine the global discretization error behavior, we define the discrete error function as:

$$E(\phi_k) = \left(\frac{1}{N} \sum_{n=1}^{N} \left| \phi_{k,n} - \phi_{exact,n} \right|^2 \right)^{1/2} \tag{4.1}$$

where k refers to the discrete mesh level and N is the number of mesh nodes in space including both interior and boundary nodes with the exception of any Dirichlet boundary nodes for which the discretization error is identically zero. Here, $\phi_{exact,n}$ refers to the exact solution (i.e., the curve fit) evaluated at node n.

Evaluation of discretization error

Since the exact solution to the nearby problem is now known, the discretization error in the numerical solution to the nearby problem no longer has to be estimated, but can be evaluated exactly.

4.2. E*xample of MNP as an evaluator of discretization error estimators*

The steps involved in MNP can be best explained by simple example. Consider that

$$L(u) = \frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2} = 0 \tag{4.2}$$

is the differential operator of interest. The first step in MNP involves obtaining a highly refined numerical solution to this original problem. Any discretization scheme can be used, and the numerical solution can be obtained. The second step involves fitting an analytic curve fit to the refined numerical solution that we have from the first step. This problem demands C^3 continuity as the highest order of the differential operator is two and the source term that we develop should be slope continuous. So the tool that we use for curve fitting should be able to provide this continuity criterion. Consider that the resulting analytic curve fit is

$$\bar{u}(x) = a + bx + cx^2 + dx^3 + ex^4 \tag{4.3}$$

Now the third step of MNP is to operate the original equation on the analytic curve fit and come up with an analytic source term. By operating the original problem of interest on the analytic curve fit, we get

$$L(\bar{u}) = b + 2c(x+1) + 3dx(x+2) + 4ex^2(x+3) \tag{4.4}$$

This is the source term, $s(x)$ and it is not equal to zero. Now this $\bar{u}(x)$ becomes the exact solution of a modified equation or the nearby equation,

$$L(u) = s(x) \quad \text{or} \quad L(u) - s(x) = 0 \tag{4.5}$$

It should be noted that as *s(x)* approaches zero, the nearby problem approaches the original problem. The next step is to come up with a numerical solution to the nearby problem. Since we have an exact solution to the nearby problem, we can evaluate the discretization error exactly.

CHAPTER FIVE

POLYNOMIAL FITTING PROCEDURES

5.1. *Polynomial fitting in MNP*

Obtaining a good polynomial fit is the most difficult task in MNP. Two conditions have to be kept in mind while fitting the numerical solution. The first condition is the continuity criterion which is problem specific. For our example case, Burgers equation, C^3 continuity is needed in the solution to maintain slope continuity of the source term. The second condition is that the fit should approximate the numerical solution fairly well to obtain small source term. The magnitude of the source terms depends on the approximation that is used. If the approximation that is used is not good, then the magnitude of the source term will increase and the nearness of the nearby problem is affected.

5.2. *Standard polynomial using MATLAB*

MATLAB [14] uses the function *polyfit* to fit a polynomial to a given set of data. Polyfit(X, Y, N) returns the coeffecients of a polynomial P(X) of degree N that fits the data P(X(I))~=Y(I) in a least-squares sense. Y = Polyval(P,X) gives the value of the polynomial evaluated at X.

These functions were used to fit a twentieth order polynomial to numerical solutions of steady-state Burgers equation for various Reynolds numbers. As shown in Fig 5.1 for the low Reynolds number case, the polynomial fits the data well. But for the

high Reynolds number case (Re=64) as shown in Fig 5.2, the global polynomial does not fit the data well. As a result, the source term will not be sufficiently small, thus this approach is limited to low Reynolds number (i.e., smoothly varying) cases only.

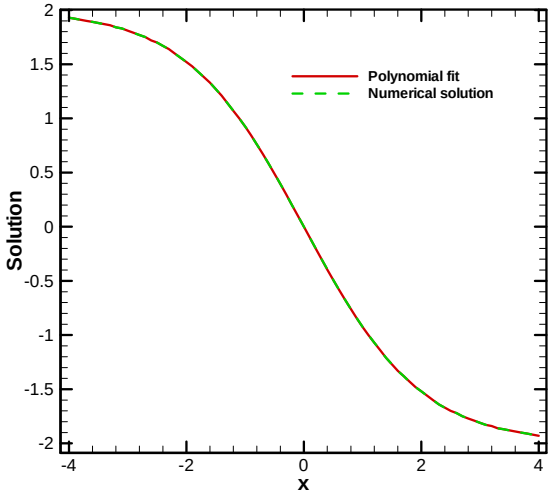

Fig 5.1. Fitting the numerical solution of steady state Burgers equation with a standard 20^{th} order polynomial: Re=8

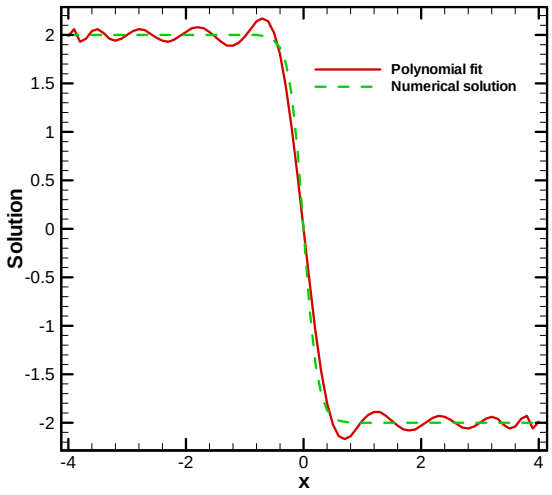

Fig 5.2. Fitting the numerical solution of steady state Burgers equation with a standard 20^{th} order polynomial: Re=64

5.3. *Legendre polynomials*

Legendre polynomials are an orthogonal set of polynomials which can be used to represent a given function. The first few Legendre polynomials are given in equations 5.1 to 5.5:

$$p_0(x) = 1 \tag{5.1}$$

$$p_1(x) = x \tag{5.2}$$

$$p_2(x) = \frac{1}{2}(3x^2 - 1) \tag{5.3}$$

$$p_3(x) = \frac{1}{2}(5x^3 - 3x) \tag{5.4}$$

$$p_4(x) = \frac{1}{8}(35x^4 - 30x^2 + 3) \tag{5.5}$$

These first five Legendre polynomials are shown graphically in Fig 5.3.

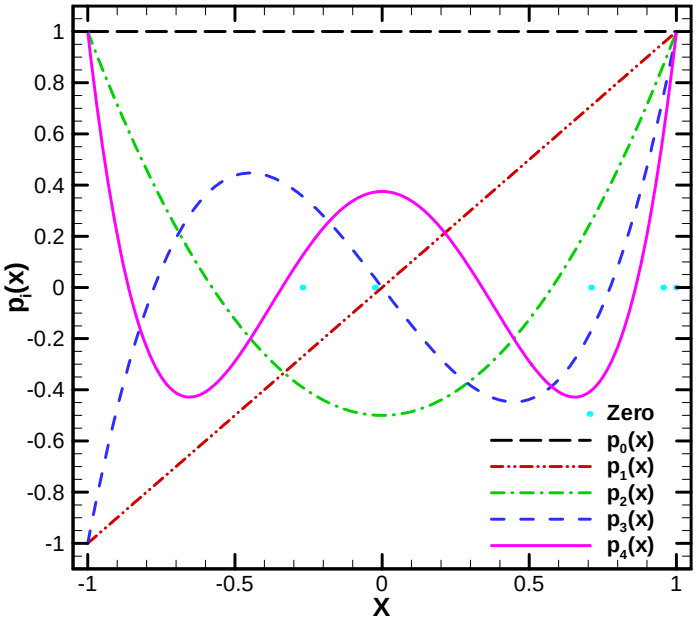

Fig 5.3. First five Legendre polynomials

The Legendre polynomials can be determined by using the following generating function:

$$p_n(x) = \sum_{k=0}^{n/2} \frac{(-1)^k (2n-2k)! x^{n-2k}}{2^n k!(n-k)!(n-2k)!} \tag{5.6}$$

We can approximate a function *f(x)* with a truncated Legendre expansion $f_n(x)$ by

$$f_n(x) = \sum_{i=0}^{n} c_i p_i(x) \tag{5.7}$$

where the coefficients c_i can be found by making use of the orthogonality of the Legendre polynomials

$$c_i = \frac{\int_{-1}^{1} f(x) p_i(x) dx}{\int_{-1}^{1} p_i(x) p_i(x) dx} \tag{5.8}$$

The main reason for using Legendre polynomials instead of standard polynomials is that the Legendre polynomial-based procedure is more stable and robust. That is, the approximations are guaranteed not to get worse as more terms are included. In Fig 5.4, the numerical solution of steady-state Burgers equation for Reynolds number of 8 is well approximated by Legendre polynomials. But when we increase the Reynolds number, this global polynomial fitting technique again fails to provide a good approximation as is shown in Fig 5.5 and Fig 5.6. In the presence of a sharp gradient region, the global polynomial approximation does not perform well.

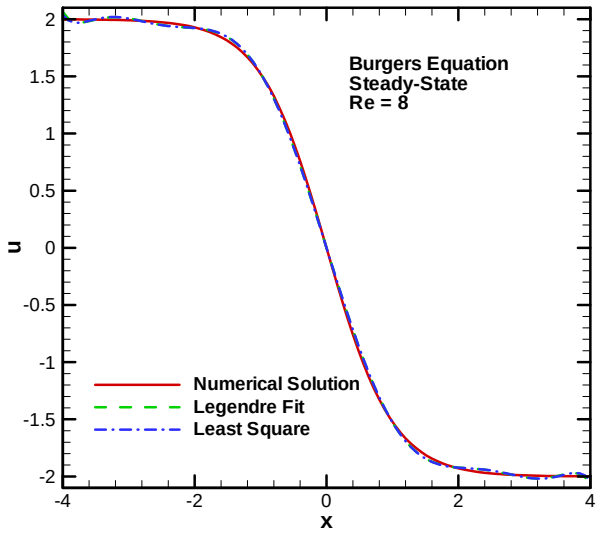

Fig 5.4 Numerical solution and 10th order Legendre and standard polynomial fits for steady-state Burgers equation at Re=8

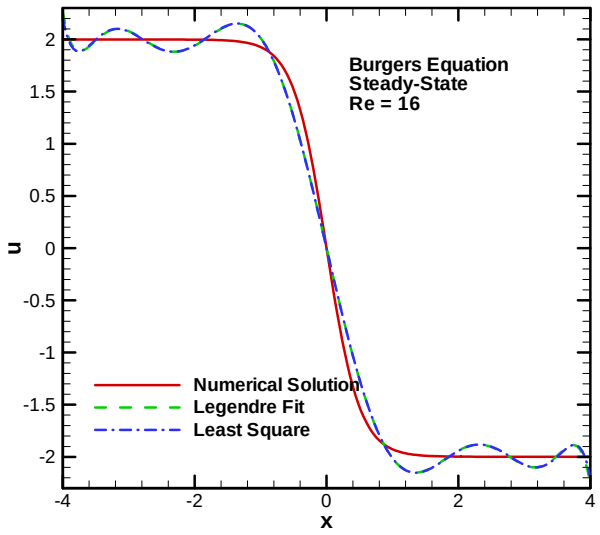

Fig 5.5 Numerical solution and 10th order Legendre and standard polynomial fits for steady-state Burgers equation at Re=16

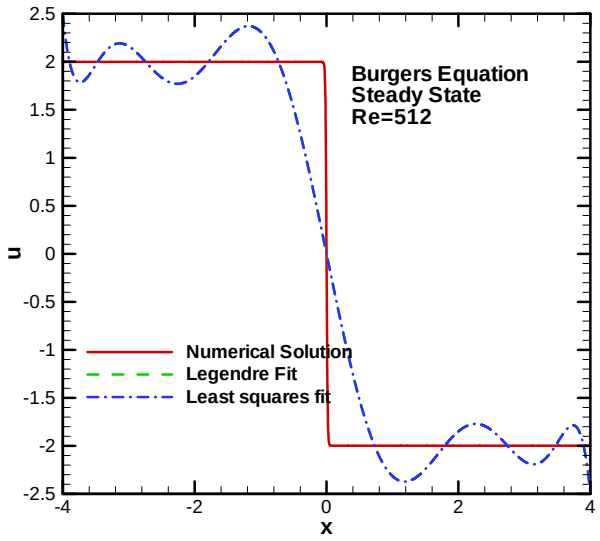

Fig 5.6 Numerical solution and 10^{th} order Legendre and standard polynomial fits for steady-state Burgers equation at Re=512

Since the Legendre polynomial does not approximate the numerical solution properly, the source term will not be small as shown in the Fig 5.7 for Re=8 and Fig 5.8 for Re=16. In these cases, the magnitude of the source term is unacceptably large, thus we will not be able to construct a problem which will be nearby the original problem. So global polynomials are not good candidates for curve fitting in MNP.

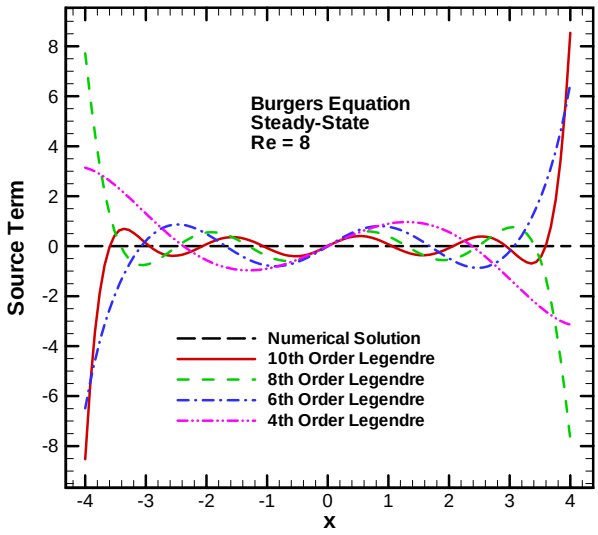

Fig 5.7 Size of source term using different order Legendre polynomial fits: Re=8

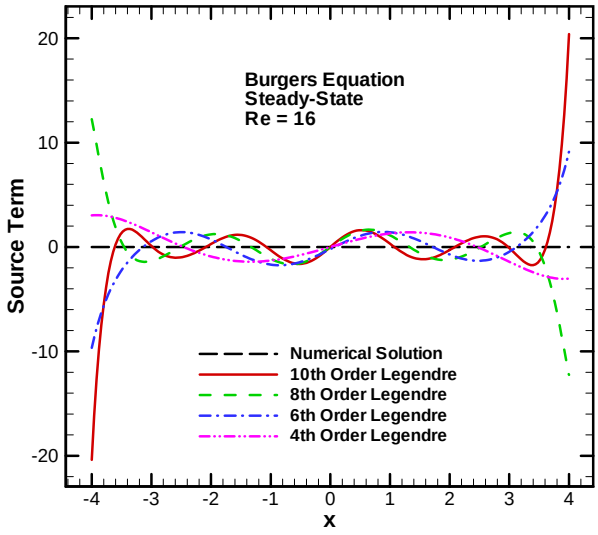

Fig 5.8 Size of source term using different order Legendre polynomial fits: Re=16

5.4. *Cubic splines*

A cubic spline [15] is a spline polynomial constructed of piecewise third-order polynomials. A cubic polynomial spline is twice continuously differentiable and depends on four parameters. It can be written as:

$$S_i(x) := a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \text{ for } x \in [x_i, x_{i+1}], i = 0,..., n-1 \quad (5.9)$$

and the setup of the system is given diagrammatically in Fig 5.9. This system has *n+1* spline points ranging from *0* to *n* and *n* spline zones.

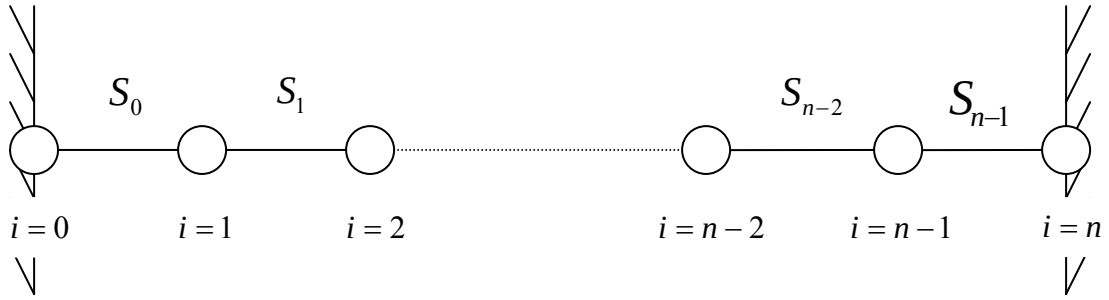

Fig 5.9. Schematic of the spline fitting system

The conditions that are used to construct the polynomials are

1. $S_i(x_i) = y_i, i = 0,...,n$
2. $S_i(x_i) = S_{i-1}(x_i), i = 1,...,n$
3. $S'_i(x_i) = S'_{i-1}(x_i), i = 1,...,n-1$
4. $S''_i(x_i) = S''_{i-1}(x_i), i = 1,...,n$

The first constraint sets the values at each node. The second constraint sets the continuity of the values at each node. The third constraint takes care of the continuity of the first derivative at each interior node and the fourth constraint takes care of the continuity of

the second derivative at each interior node. Here we also set $S_n(x_n) = a_n$ and $S''_n(x_n) = 2c_n$ for convenience. The first derivatives at end points are also specified which gives two additional conditions. As we have already seen, global polynomials do not do a good job approximating sharp gradients. The numerical solution was fit using cubic splines and the results were encouraging as shown in Figs 5.10, and 5.11.

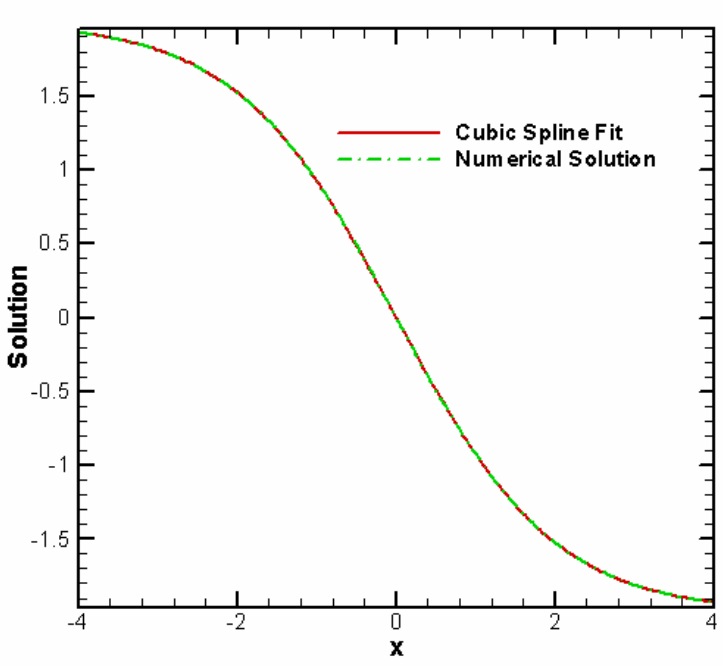

Fig 5.10. Fitting the numerical solution with cubic spline fits using 257 nodes and 9 spline points, Re=8

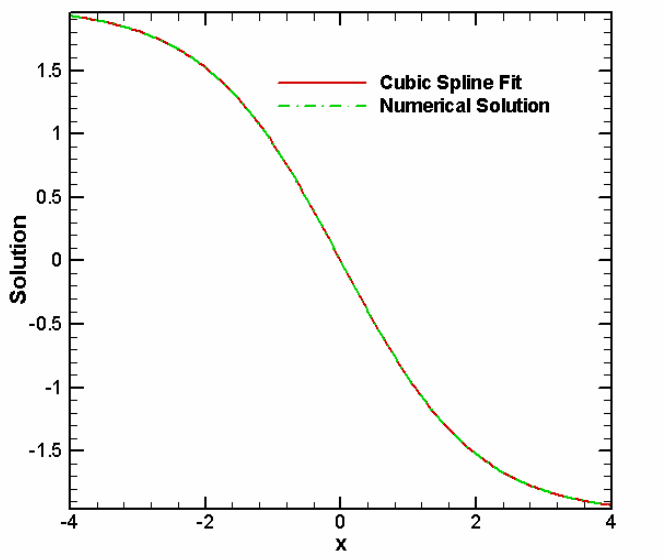

Fig 5.11. Fitting numerical solution with cubic spline fits using 257 nodes and 17 spline points, Re=8

Since the cubic splines perform well in fitting the numerical solution, one can expect that the source term that result will be small as desired. Plots of the source term using cubic splines are given in Figs 5.12 and 5.13. Close examination of these figures indicates that the source terms exhibit slope discontinuities at the spline points. Since cubic splines are only C^2 continuous and Burgers equation contains a second derivative, the first criteria is not satisfied and we cannot use cubic splines

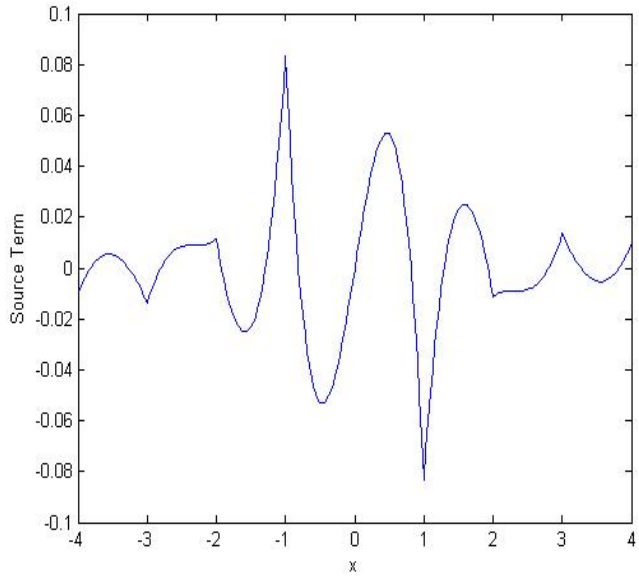

Fig 5.12. Source term distribution using 9 spline points and 257 nodes

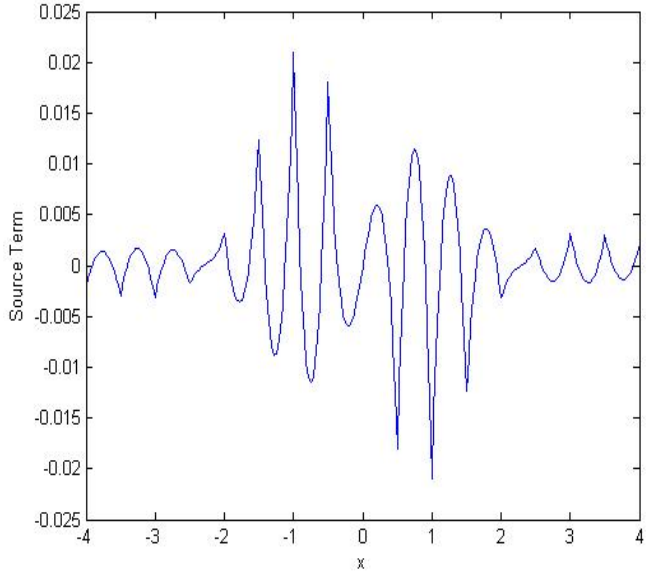

Fig 5.13. Source term distribution using 17 spline points and 257 nodes

5.5. *Fifth order Hermite spline*

A fifth-degree Hermite spline [15] is a spline polynomial constructed of piecewise fifth-order polynomials. This system also has *n+1* spline points ranging from *0* to *n* and *n* spline zones. This can be given by

$$S_i(x) := a_i + b_i(x-x_i) + c_i(x-x_i)^2 + d_i(x-x_i)^3 + e_i(x-x_i)^4 + f_i(x-x_i)^5$$

$$\text{for } x \in [x_i, x_{i+1}], i = 0,...,n-1 \tag{5.10}$$

where the same spline system shown in Fig. 5.9 is used. The conditions used to construct a fifth-degree Hermite spline are:

1. $S_i(x_i) = y_i, i = 0,...,n$
2. $S'_i(x_i) = y'_i, i = 0,...,n$
3. $S_i(x_i) = S_{i-1}(x_i), i = 1,...,n$
4. $S'_i(x_i) = S'_{i-1}(x_i), i = 1,...,n-1$
5. $S''_i(x_i) = S''_{i-1}(x_i), i = 1,...,n-1$
6. $S'''_i(x_i) = S'''_{i-1}(x_i), i = 1,...,n-1$

and also set $S_n(x_n) = a_n$ and $S'_n(x_n) = b_n$ as two extra constraints. The first constraint sets the values at each node. The second constraint sets the first derivative at each node. The third constraint sets the continuity of the solution at each node. The fourth constraint takes care of the continuity of the first derivative at each interior node. The fifth constraint takes care of the continuity of the second derivative at each interior node, and the sixth constraint sets the third derivative continuity at each interior node.

As is seen in Figs 5.14, 5.15 and 5.16, fifth-order Hermite splines do a good job in approximating the numerical solution for the various Reynolds number cases. Unlike

global polynomial fitting techniques, fifth-order Hermite splines approximates even the high Reynolds number cases well, including sharp gradient region. Fifth-order Hermite splines were found to be the best fitting tool that can be used for this problem. The chosen example of Burgers equation demanded C^3 continuity which is met by fifth-order Hermite splines.

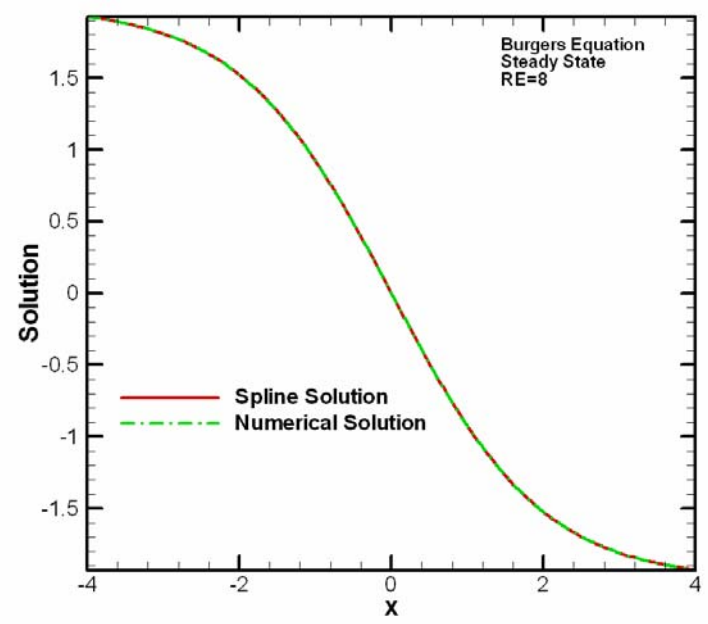

Fig 5.14. Fifth order Hermite spline approximation for Re=8, using 17 spline points

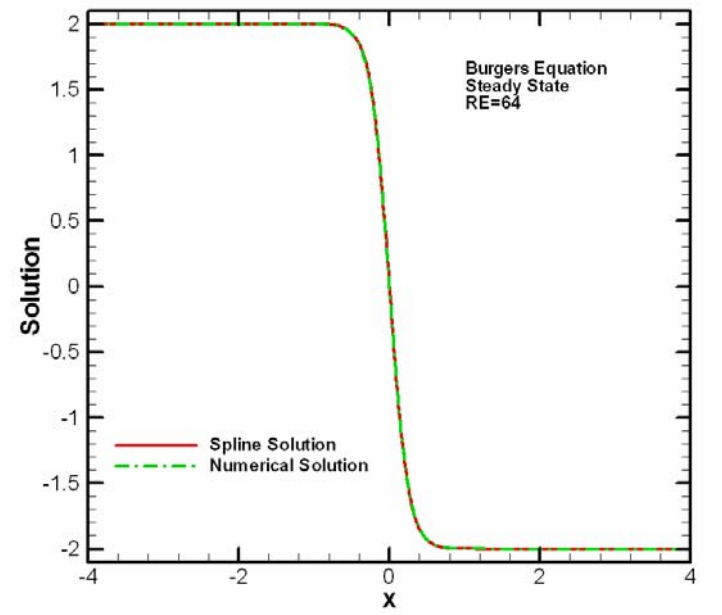

Fig 5.15. Fifth order Hermite spline approximation for Re=64, using 65 spline points

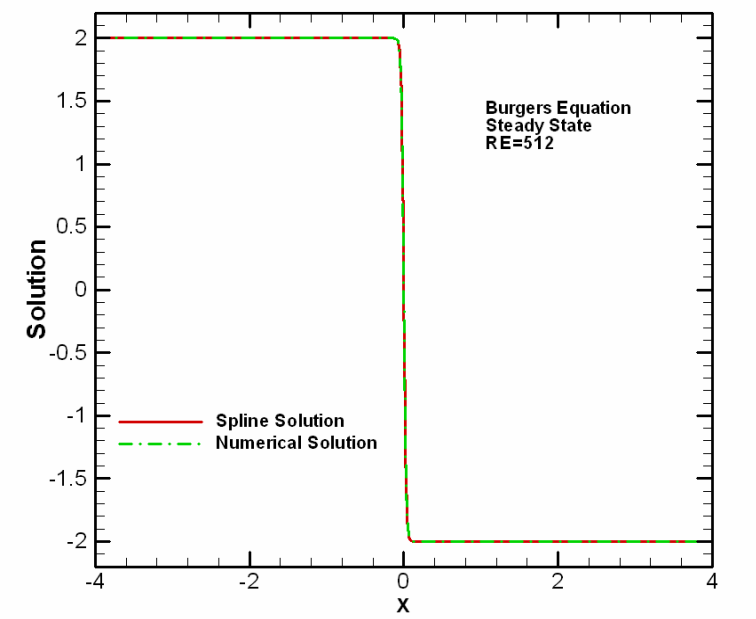

Fig 5.16. Fifth order Hermite spline approximation for Re=512, using 129 spline points

The numerical solution was fit accurately enough to provide small source terms as shown in Fig 5.17. for Re=8. In addition, the source term is slope continuous (C^1 continuous) over the entire domain. The size of the source terms for the fifth-order Hermite splines will be discussed in Chapter 7. A fortran program was developed to compute the coefficients of the fifth-order Hermite splines (see Appendix D).

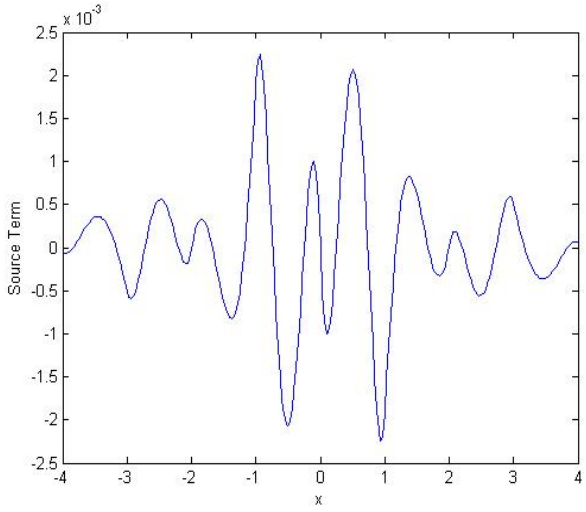

Fig 5.17 Source term distribution using fifth-order Hermite splines with 9 spline points and 257 nodes, Re=8

CHAPTER SIX

DISCRETIZATION ERROR ESTIMATORS

Discretization error arises due to the fact that the original partial differential equations must be discretized to numerically solve them. There are various methods to compute this discretization error and in this chapter we discuss those methods.

6.1. *Discretization error estimator using local order of accuracy*

The discretization error on the fine grid is given as:

$$DE = f_1 - f_{exact} \tag{6.1}$$

where f_{exact} is the exact solution to the partial differential equation and f_1 is the numerical solution. We can estimate f_{exact} using Richardson extrapolation [3] which is given by:

$$f_{exact} = f_1 + \frac{f_1 - f_2}{r^p - 1} \tag{6.2}$$

where r is the grid refinement factor, p is the order of accuracy, and f_2 and f_1 are the solutions on the coarse and fine meshes, respectively. Here the order of accuracy can be computed using solutions on three meshes as

$$p = \frac{\ln\left(\frac{f_3 - f_2}{f_2 - f_1}\right)}{\ln(r)} \tag{6.3}$$

where f_1, f_2 and f_3 are the solutions on fine, medium and coarse meshes, respectively. To use Richardson extrapolation with the local order of accuracy as an error estimator, we need numerical solution on three different meshes.

6.2. *Discretization error estimator using global order of accuracy*

This technique is different from Richardson extrapolation with local order of accuracy in the sense that we use the formal order of accuracy instead of computing the local order of accuracy. As a result, this technique uses numerical solutions on just two meshes.

6.3. *Mixed order error estimator*

The mixed order error estimator [16] involves approximation of an exact solution using three different meshes. In this technique, instead of assuming a single dominant error term, both first and second order error terms are considered.

$$f_k = f_{exact} + g_1 h_k + g_2 h_k^2 + HOT \tag{6.4}$$

Three discrete solutions are used to solve a linear set of equations (see [2] for details) and finally arrive at an approximation for the exact solution given by

$$f_{exact} = \frac{(f_3 - f_2) - (r^2 + r - 1)(f_2 - f_1)}{(r+1)(r-1)^2} \tag{6.5}$$

Once the exact solution is approximated, the discretization error relation is used to compute the error, which is given as:

$$Mixed\ Order\ Error = f_1 - f_{exact} \tag{6.6}$$

6.4. *Method of nearby problems*

The Method of nearby problems (MNP) itself can be used as an error estimator. The discretization error on any given mesh can be evaluated exactly for the nearby problem as the MNP approach involves the generation of an exact solution to the nearby problem. If the nearby problem is "close enough" to the original problem of interest, then the error on a given mesh for the nearby problem is expected to be very close to the error in the original problem on the same mesh. The expression for using MNP as an error estimator is given as:

$$MNP_1 = f_{1,MNP} - f_{exact,MNP} \tag{6.7}$$

where $f_{1,MNP}$ is the numerical solution of the nearby problem on a mesh and $f_{exact,MNP}$ is the exact solution of the nearby problem. Here we see that for using this technique, we use numerical solutions on only one mesh unlike other schemes which use multiple meshes. MNP thus requires two solutions on the same mesh, one for the original problem and the second one for the nearby problem.

CHAPTER SEVEN

RESULTS

We have applied MNP to three different one-dimensional test problems:

1. Original Burgers equation (steady state): The exact solution to the steady state Burgers equation was given by Benton & Platzman [12].
2. "Nearby" problem to Burgers equation: The exact solution to the nearby problem to Burgers equation is the fifth order Hermite spline fit to the numerical solution of the original Burgers equation.
3. Modified form of Burgers equation: A modified form of Burgers equation was generated which includes a nonlinear viscosity which varies as a function of both u and x thus giving a nominal Reynolds number of 64.

7.1. *Steady-state Burgers equation*

An implicit scheme was used to obtain the numerical solution to Burgers equation (see Appendix A). Two different Reynolds number cases were run for the original Burgers equation: Re=8 and Re=64. Both numerical solutions and exact solutions for these two cases are shown in Figs. 7.1, and 7.2. The numerical solution is right on top of exact solution which suggests that the implicit scheme used to obtain the numerical solution is performing well.

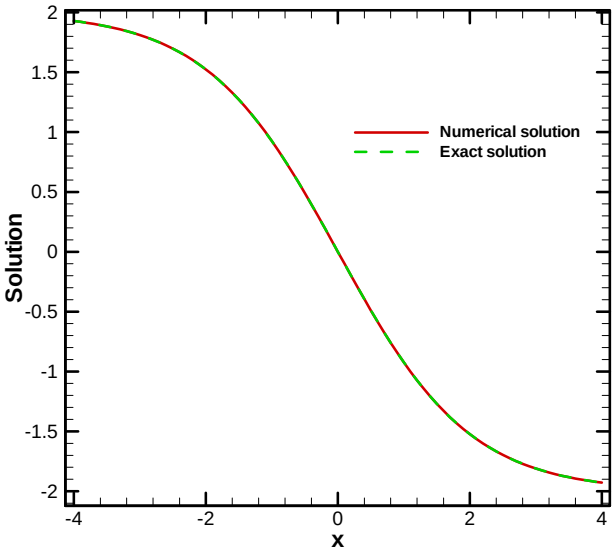

Fig 7.1 Comparison of the numerical solution with exact solution, Re=8

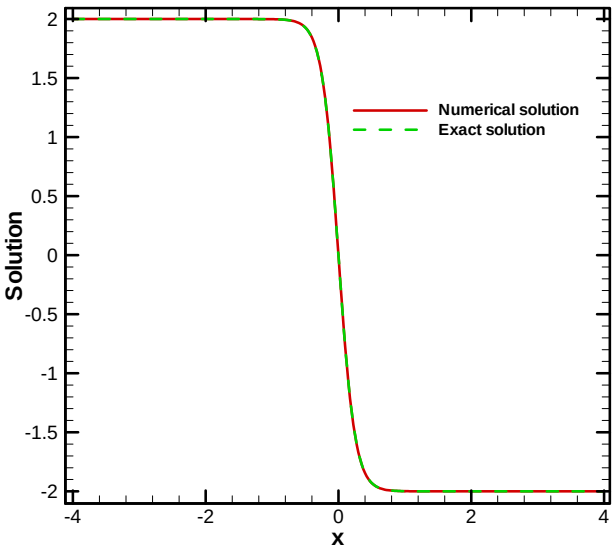

Fig 7.2 Comparison of the numerical solution with exact solution, Re=64

Since the exact solution of the steady Burgers equation is known, we can evaluate the discretization error exactly. The discretization error for two different mesh spacing ($\Delta x = 0.25$ and $\Delta x = 0.0625$) is shown in Fig. 7.3. As expected, the discretization error increases when the mesh spacing increases.

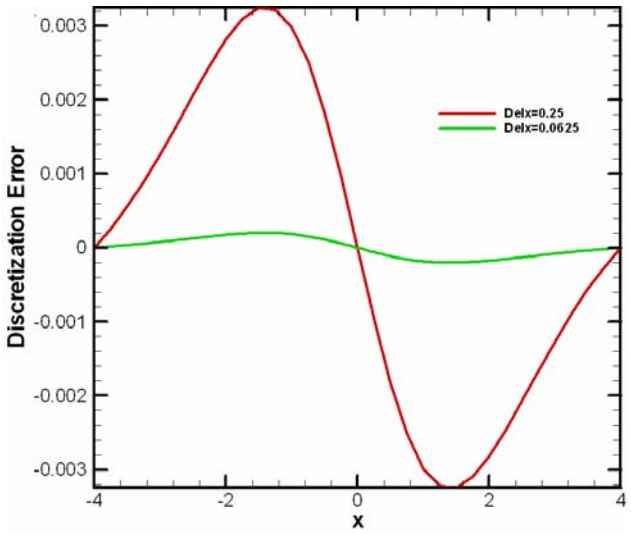

Fig 7.3 Discretization error for the steady state Burgers equation, Re=8

The observed order of accuracy was computed for the various mesh solutions. Since the exact solution is known, the observed order of accuracy can be computed by the relation

$$p = \frac{\ln(\frac{DE_2}{DE_1})}{\ln(r)} \tag{7.2}$$

where r is the grid refinement factor and DE_2 and DE_1 are the L_2 norms of the discretization errors for the coarse and fine mesh, respectively. The plot for the observed order of accuracy is given in Fig. 7.4. The observed order of accuracy is seen to be

approaching two, which is the formal order of accuracy. This suggests that the numerical solution that was computed is good.

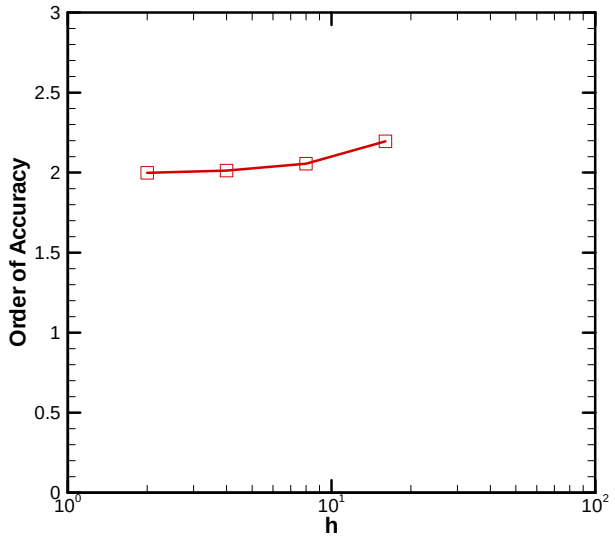

Fig 7.4 Observed order of accuracy for different meshes, Re=8

7.2 *Nearby problem to Burgers equation*

The nearby problem to Burgers equation is constructed by using fifth-order Hermite spline to fit the original numerical solution, then operating Burgers equation on the spline fit to generate analytical source terms. The source terms are then added to the original problem of interest to give the nearby problem (see Appendix B). When the source term approaches zero, the nearby problem approaches the original problem. The same implicit scheme is used to solve the nearby problem. Two different Reynolds number cases were run for the problem nearby Burgers equation: Re=8 and Re=64. In both cases an underlying numerical solution from a 1025 node mesh is used. A

description of process of choosing the number of spline points is given in detail in chapter 7.4. The numerical solutions of the nearby problem for Re=8 and Re=64 are shown in Figs. 7.5 and 7.6. For Re=8 case, 17 spline points were used and for the Re= 64 case, 65 spline points were used. The numerical solution to the nearby problem is compared with the exact solution to the nearby problem. The figures show that the numerical solution obtained is good.

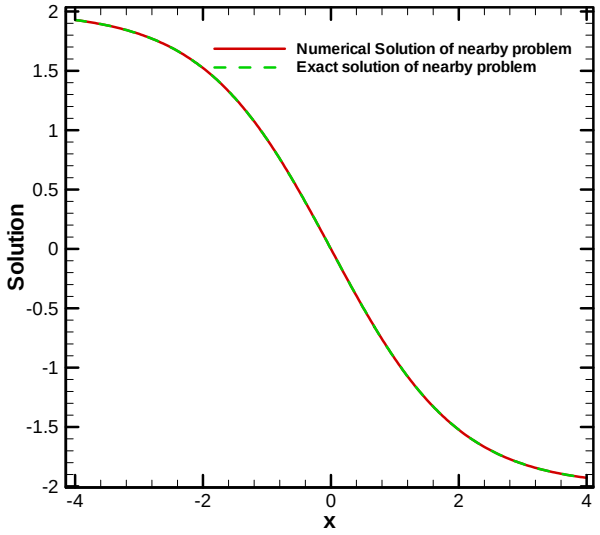

Fig 7.5 Numerical solution of the nearby problem to Burgers equation, Re=8

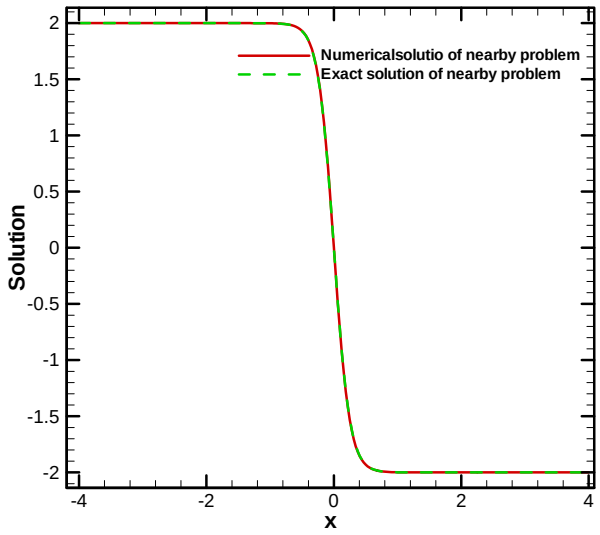

Fig 7.6 Numerical solution of the nearby problem to Burgers equation, Re=64

The order of accuracy of the nearby problem was also computed by comparing numerical solutions on different meshes. The plot of the order of accuracy of the nearby problem and the original Burger equation is shown in Fig. 7.7 for the Re = 8 case and Fig. 7.8 for the Re = 64 case. In both the cases, as the mesh becomes fine, the order of accuracy approaches two which is the formal order of accuracy. This suggests that the solution that we have obtained is good.

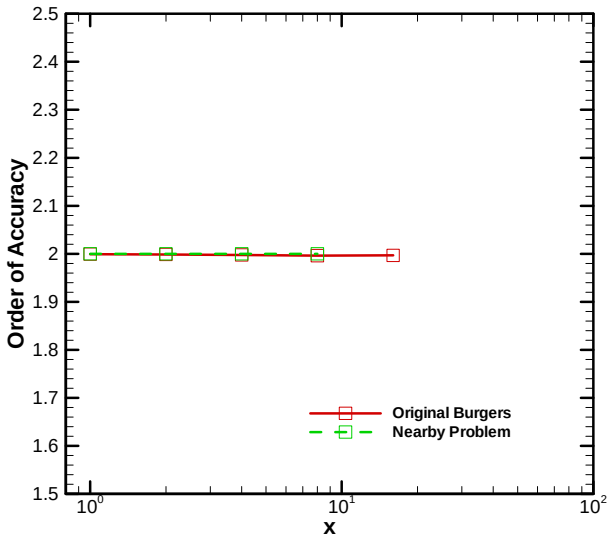

Fig 7.7 Observed order of accuracy for Burgers equation and the nearby problem, Re=8

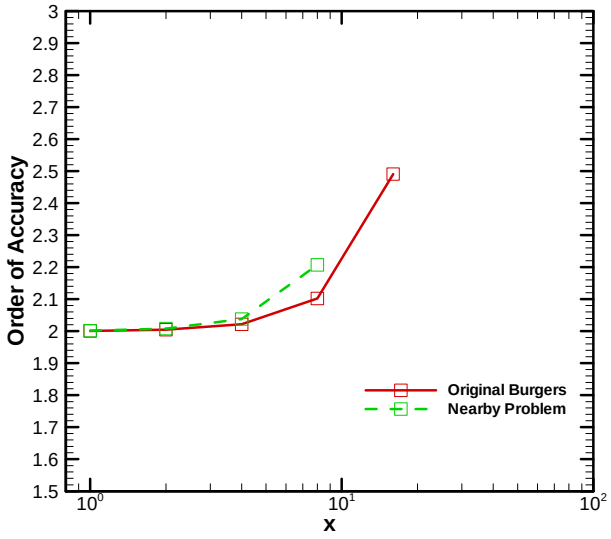

Fig 7.8 Observed order of accuracy for Burgers equation and the nearby problem, Re=64

7.3 *Modified form of Burgers equation*

A modified form of Burgers equation was generated which includes a nonlinear viscosity which varies as a function of both u and x:

$$\frac{\upsilon}{\upsilon_0} = \left(\frac{u}{u_0}\right)^2 + \left(\frac{x - x_L}{x_R - x_L} + \frac{1}{4}\right)^{1/4} \tag{7.3}$$

The constants were chosen as $\upsilon_0 = 0.25$ m^2/s, $u_0 = 2$ m/s, $x_L = -4$ m, and $x_R = 4$ m, thus giving a nominal Reynolds number of 64. This modified form of Burgers equation was solved numerically using a mesh with 1025 spatial points. The numerical solution and the viscosity distribution for this modified form of Burgers equation are given in Fig 7.9. Implicit scheme was used to obtain the numerical solution to the modified form of Burgers equation (see Appendix C).

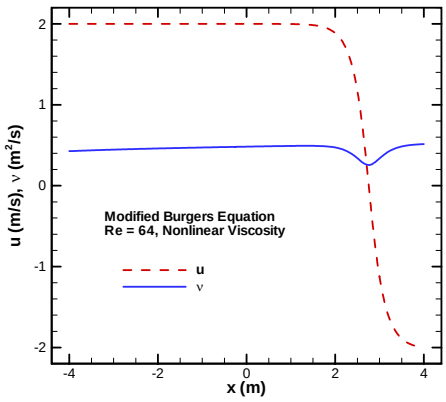

Fig 7.9 Solution and viscosity variation for the modified form of Burgers equation

The source terms were computed and added to the modified form of Burgers equation, which then resulted in a nearby problem the modified form of Burgers equation. The numerical solution is given in Fig 7.10. The plots shows the numerical

solution of modified Burgers equation with the numerical solution of the nearby problem to modified Burgers equation. The two solutions are very close to each other.

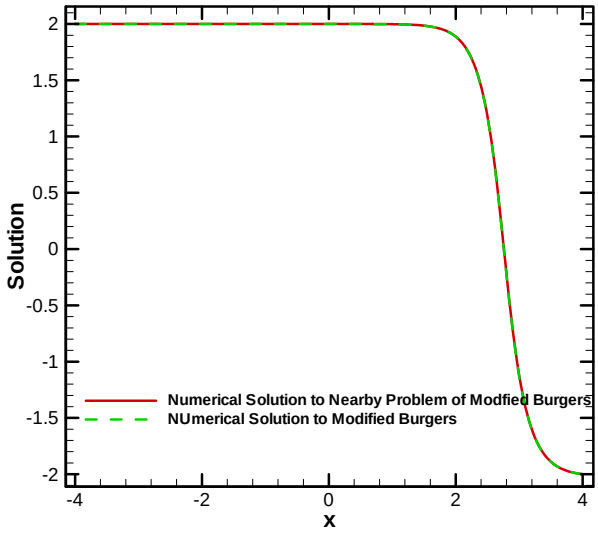

Fig 7.10 Numerical solution of the nearby problem to modified form of Burgers equation

7.4. *Nearness of the nearby problem*

The nearness of the nearby problem can be measured using the source terms. The source terms are computed by operating the governing equations on the analytical curve fit. The source term should be small for the nearby problem to be close to the original problem. As the source term approaches zero, the nearby problem approaches the original problem of interest. Source terms were calculated for the nearby problem to the Burgers equation for three different cases: Re=8, Re=64, and Re=512 as shown in Figs. 7.11, 7.12, and 7.13, respectively. In these figures we see that as the number of spline points increases, the magnitude of the source term decreases. We require that the source

term be small. Examining the Figs. 7.11, 7.12, and 7.13, the source term is the smallest for 17 spline point case and so this case will be the best option.

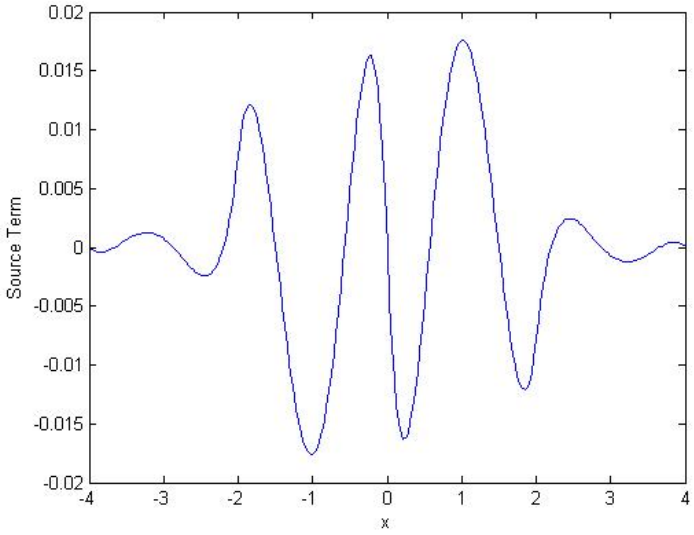

Fig 7.11 Magnitude of the source term for the nearby problem with 5 spline points, Re=8

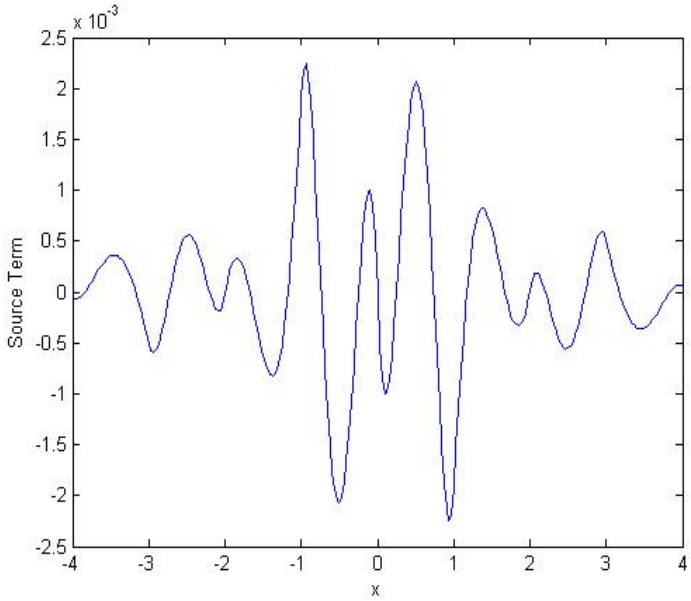

Fig 7.12 Magnitude of the source term for the nearby problem with 9 spline points, Re=8

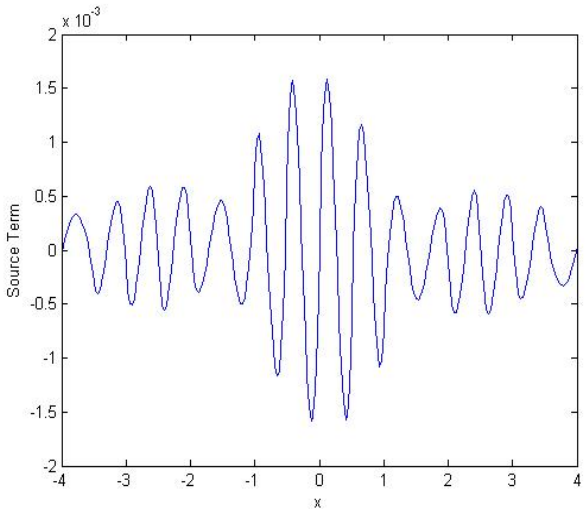

Fig 7.13 Magnitude of the source term for the nearby problem with 17 spline points, Re=8

For the Reynolds number 64 case, the distribution of the source term over the domain is given in Figs. 7.14, 7.15, and 7.16. Here again as the number of spline points increases the magnitude of the source term decreases. The smallest source term is obtained for 65 spline points and so we choose 65 spline points for Re= 64.

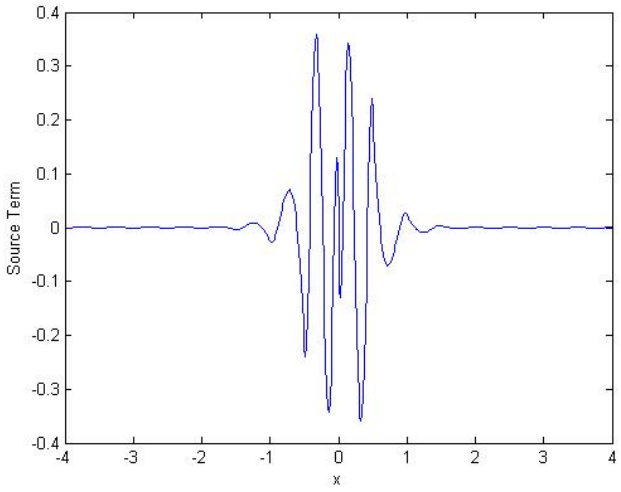

Fig 7.14 Magnitude of the source term for the nearby problem with 17 spline points, Re=64

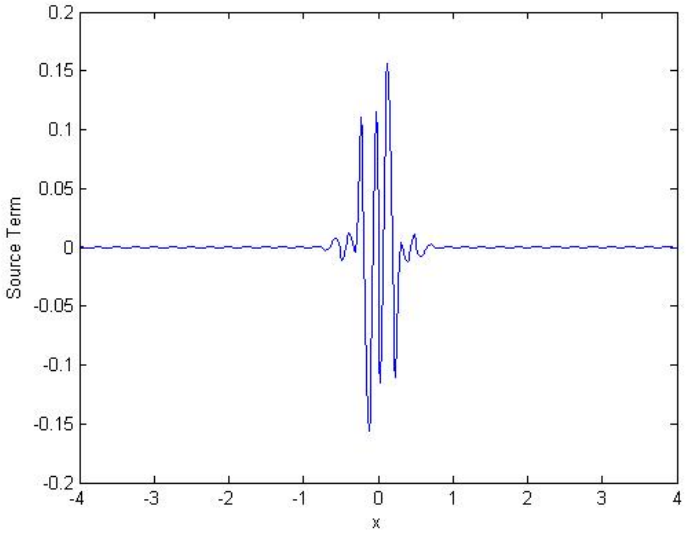

Fig 7.15 Magnitude of the source term for the nearby problem with 33 spline points, Re=64

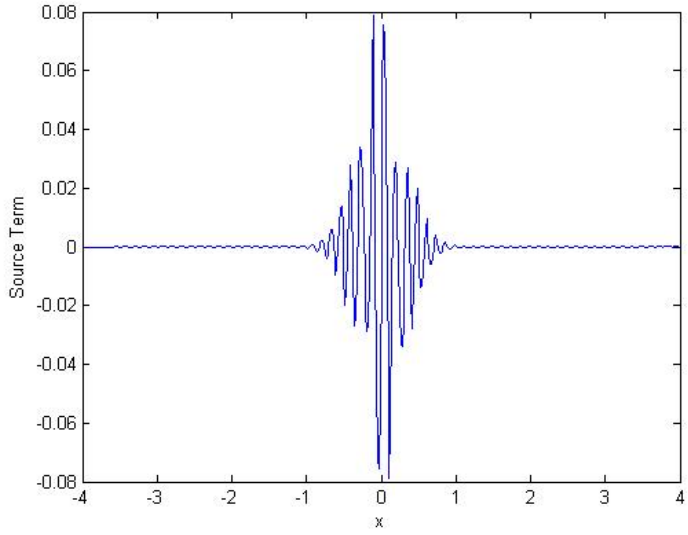

Fig 7.16 Magnitude of the source term for the nearby problem with 65 spline points, Re=64

For Re=512, since there is a very sharp gradient region, the number of spline points to capture the solution should be large. The distribution of the source term for Re=512 is presented in Figs. 7.17, 7.18, and 7.19. Here we see that for 129 and 257 spline points, the source term is still relatively large and is not going to result in a problem close to the original problem. But using 1025 spline points, the size of the source term is decreased. This does not necessarily mean that for higher Reynolds number cases we have to use a large number of spline points. The number of spline points could be reduced by the use of spline points with variable spacing. That is, if we can recognize the areas of large variation, and include a large number of spline points to capture that variation and fewer points elsewhere, then the number of spline points can be significantly reduced.

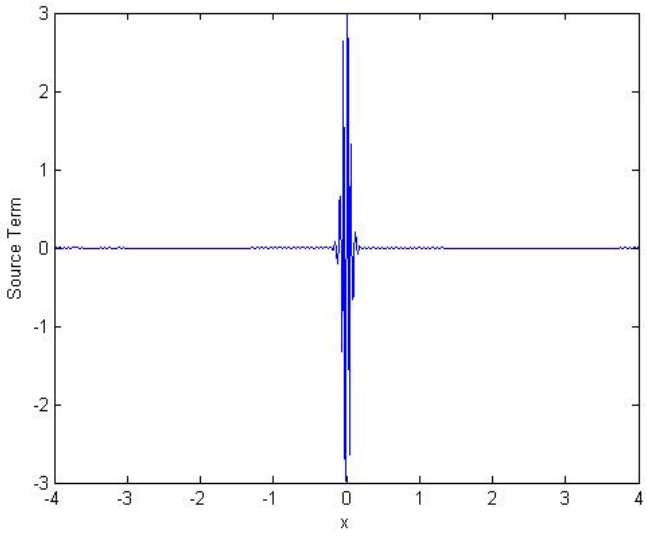

Fig 7.17 Magnitude of the source term for the nearby problem with 129 spline points, Re=512

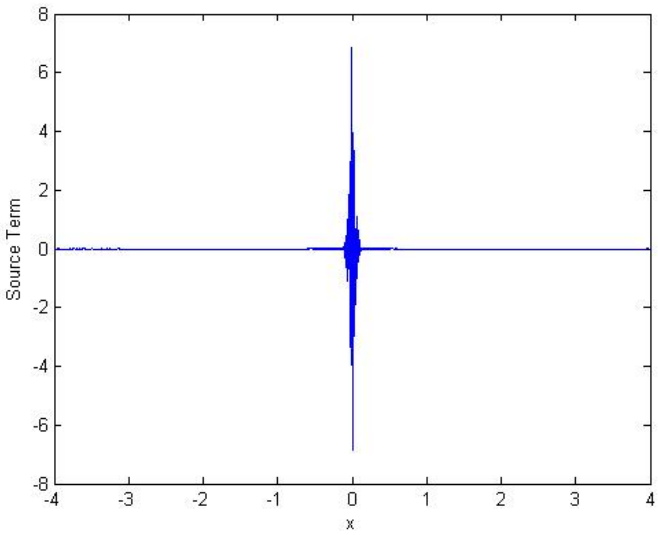

Fig 7.18 Magnitude of the source term for the nearby problem with 257 spline points, Re=512

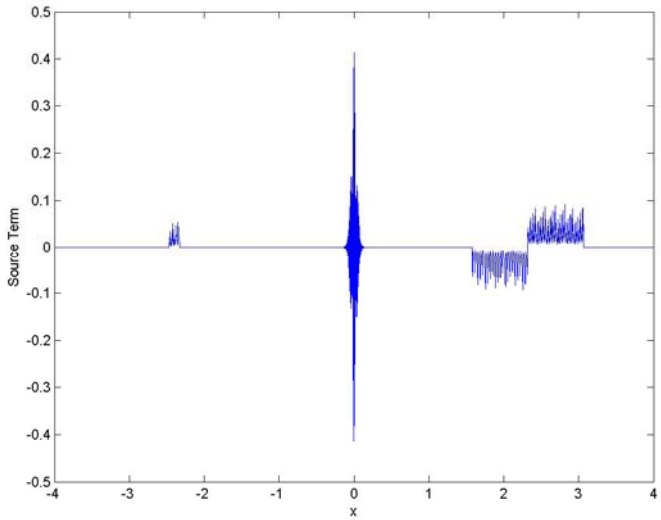

Fig 7.19 Magnitude of the source term for the nearby problem with 1025 spline points, Re=512

Source terms were also computed for the nearby problem to the modified form of Burgers equation for a Reynolds number of 64. The magnitude of the source terms is given in Figs. 7.20, 7.21, and 7.22. Here it is found that the magnitude of the source term decreases when we go from 33 spline point to 65, and then increases slightly when we go from 65 to 129 spline points. So in this case we should use 65 spline points to formulate the nearby problem.

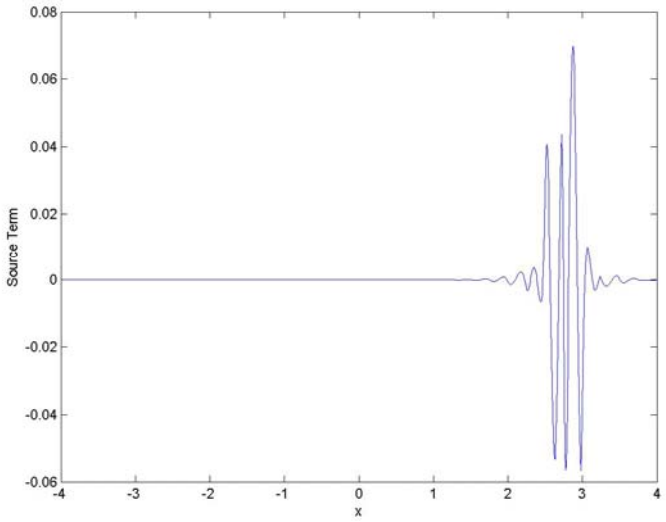

Fig 7.20 Magnitude of the source term for the nearby problem to modified Burgers equation with 33 spline points, Re=64

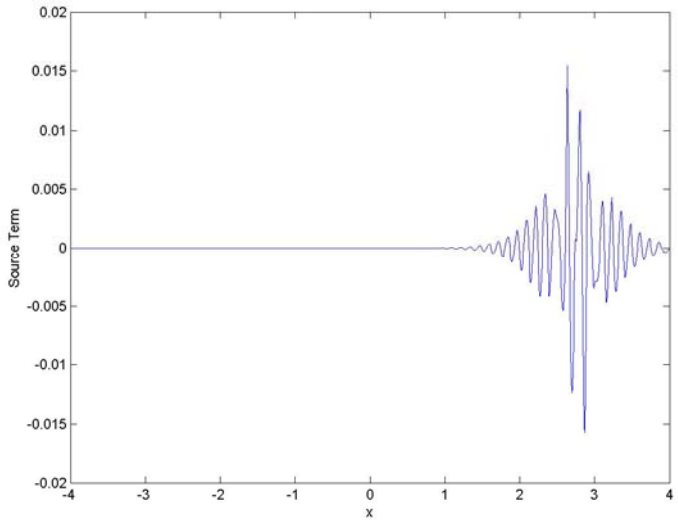

Fig 7.21 Magnitude of the source term for the nearby problem to modified Burgers equation with 65 spline points, Re=64

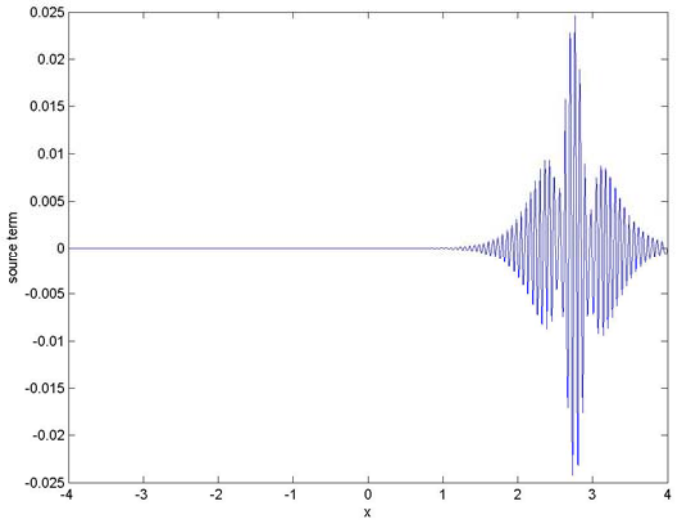

Fig 7.22 Magnitude of the source term for the nearby problem to modified Burgers equation with 129 spline points, Re=64

7.5. *Evaluation of discretization error estimators*

Now that the nearness of the nearby problem has been established, we are in a position to evaluate the various discretization error estimators. We now examine discretization error estimates on various grid levels using four different methods: 1) Richardson extrapolation with global p, i.e. assuming the formal order of accuracy (requiring two grids), 2) Richardson extrapolation with local p, i.e. Richardson extrapolation employing the locally calculated order of accuracy (requiring three grids), 3) a mixed-order error estimator (requiring three grids), and 4) the Method of Nearby Problems (MNP) (requiring only one grid). Numerical solutions are computed on a wide range of grid levels. In cases where multiple mesh levels are required to obtain the error estimate, the error is reported for the finest grid only. Grid refinement is performed by doubling the node spacing (i.e., grid doubling) in all cases.

Fig 7.23 gives the discretization error estimates of all the error estimators described earlier for meshes using 1025, 513 and 257 points for Re=8 case. For these fine meshes we see that all the error estimators do a good job in estimating the error. Fig. 7.24 gives the discretization error estimates for meshes using 257, 129 and 65 points. All the estimators again match the true error. Fig. 7.25 gives the discretization error estimates for the relatively coarse meshes using 65, 33 and 17 points. The mixed-order error estimator and Richardson extrapolation using the local order of accuracy underestimates the error, while MNP and Richardson extrapolation using the global order of accuracy match the true error. Fig. 7.26 gives the discretization error estimates for very coarse meshes using 33, 17 and 9 points. The mixed-order error estimator and Richardson extrapolation using local order of accuracy do not provide good estimates, while MNP and Richardson

extrapolation using the global order of accuracy are in good agreement with the true error.

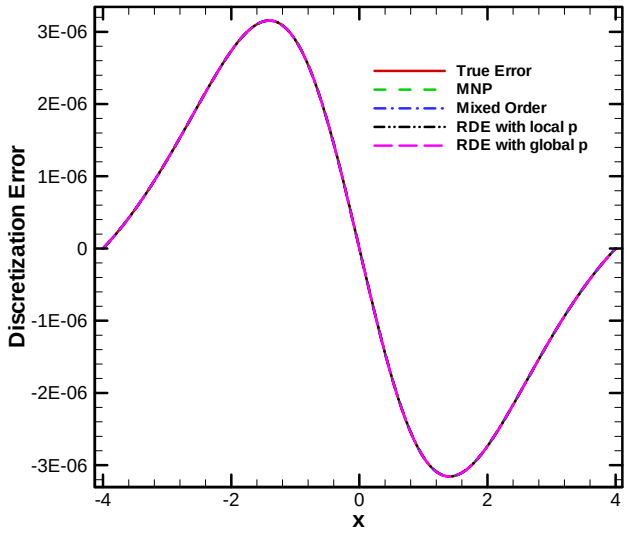

Fig 7.23 Discretization error estimators for Burgers equation, Re=8, finest mesh =1025 points

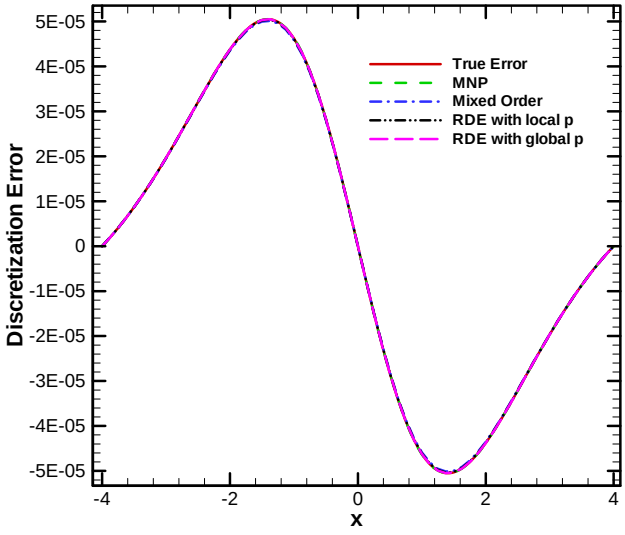

Fig 7.24 Discretization error estimators for Burgers equation, Re=8, finest mesh = 257 points

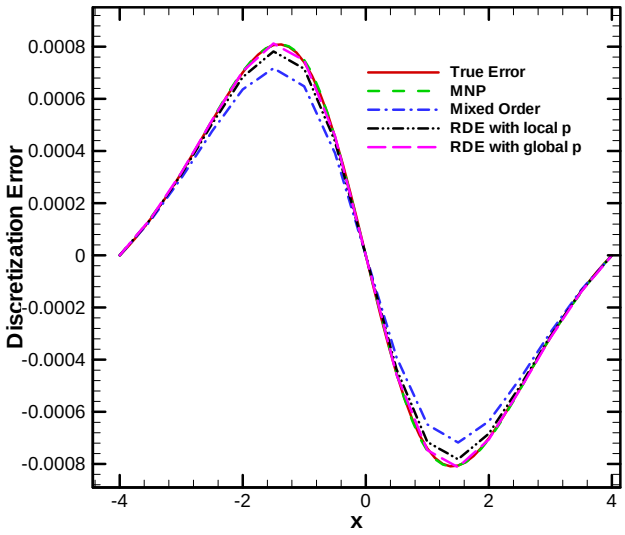

Fig 7.25 Discretization error estimators for Burgers equation, Re=8, finest mesh = 65 points

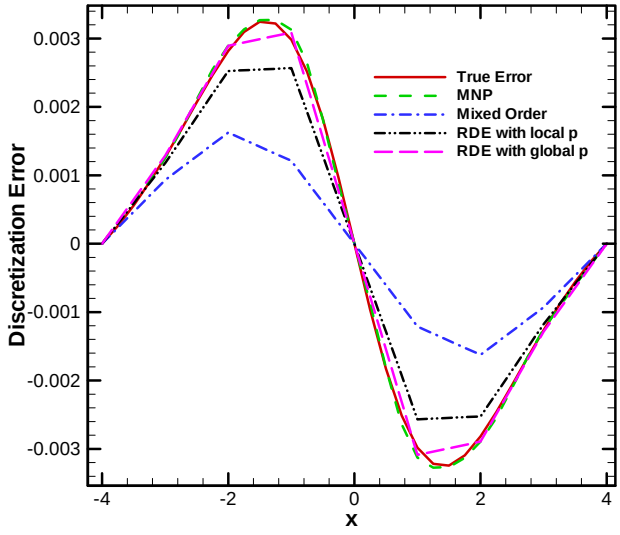

Fig 7.26 Discretization error estimators for Burgers equation, Re=8, finest mesh = 33 points

Next we examine the higher Reynolds number case with Re=64. Here also we expect that all the error estimators will perform well on the finer meshes, while on coarser meshes, some of the error estimators will break down. The plots of the discretization error estimators for the Reynolds number of 64 are given in Figs. 7.27, 7.28, 7.29, and 7.30. In Fig 7.27, where the finest mesh used is 1025 points, all the error estimators perform well. In Fig 7.28, where the finest mesh is 257 points, the mixed-order error estimator and Richardson extrapolation with the global order of accuracy begin to deviate from true error. In Fig. 7.29, only the error estimators using two or fewer mesh levels can be used, as the coarsest mesh that could be computed was 33, below which the solution was found to be numerically unstable. Here both MNP and Richardson extrapolation using the global order of accuracy gives a slightly higher estimate (but in reasonable limit). Due to the same reason, in Fig 7.30, only MNP could be used as all other estimators needed additional coarse meshes. MNP gives a slightly higher estimate of the error.

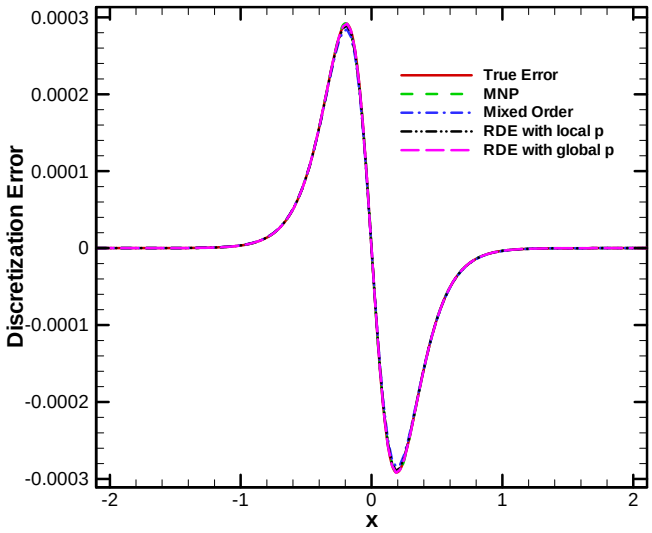

Fig 7.27 Discretization error estimators for Burgers equation, Re=64, finest mesh =1025

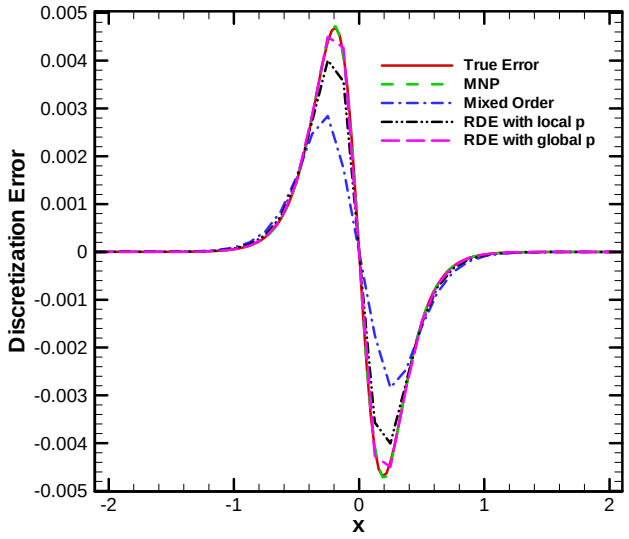

Fig 7.28 Discretization error estimators for Burgers equation, Re=64, finest mesh =257

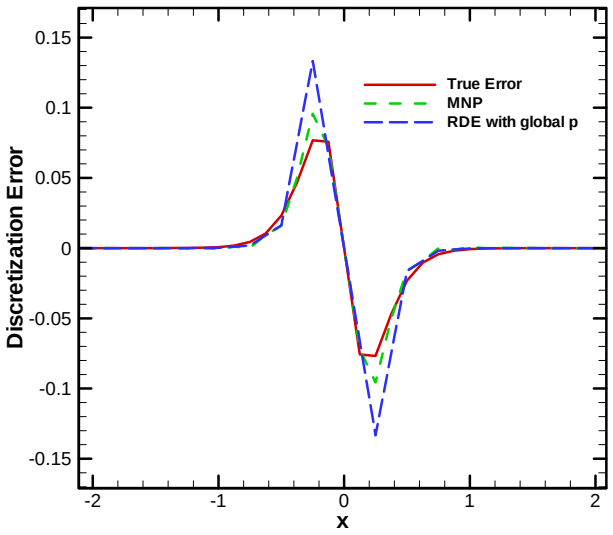

Fig 7.29 Discretization error estimators for Burgers equation, Re=64, finest mesh =65

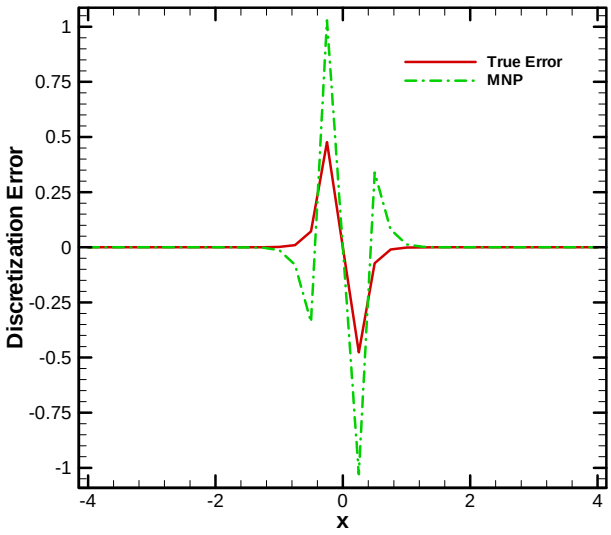

Fig 7.30 Discretization error estimators for Burgers equation, Re=64, finest mesh =33

We now turn our attention to the discretization error in the nearby problem. Again different meshes are used for each Reynolds number case. For a Reynolds number of 8, we used 17 spline fit points. Fig 7.31 shows discretization error estimates with the finest mesh being 1025 points. In this case, we see that all the error estimators perform well. In Fig. 7.32, the finest mesh used was 257 points. Again all the estimators give a good estimate of the discretization error. In Fig. 7.33, the finest mesh that was used was 129 points. Here we see that the mixed-order error estimator begins to under estimate the error. In Fig. 7.34, both the mixed-order error estimator and Richardson extrapolation using the local order of accuracy underestimate the error.

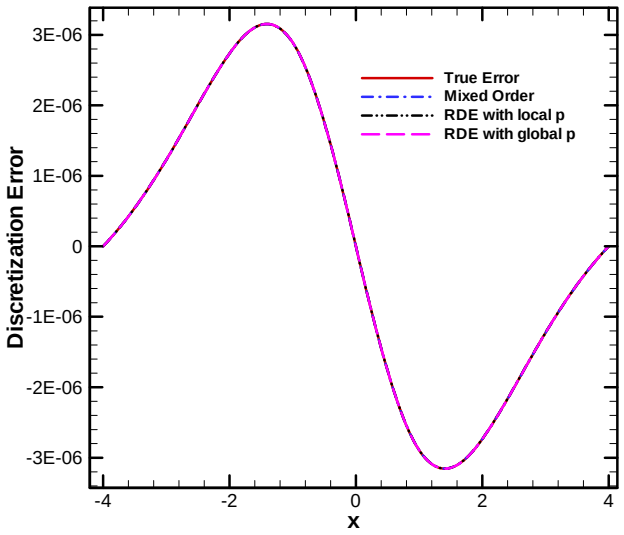

Fig 7.31 Discretization error estimators for the nearby problem, Re=8, finest mesh=1025

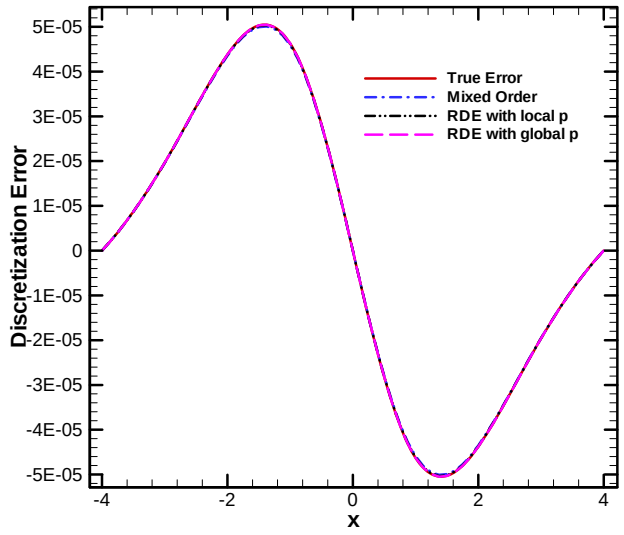

Fig 7.32 Discretization error estimators for the nearby problem, Re=8, finest mesh=257

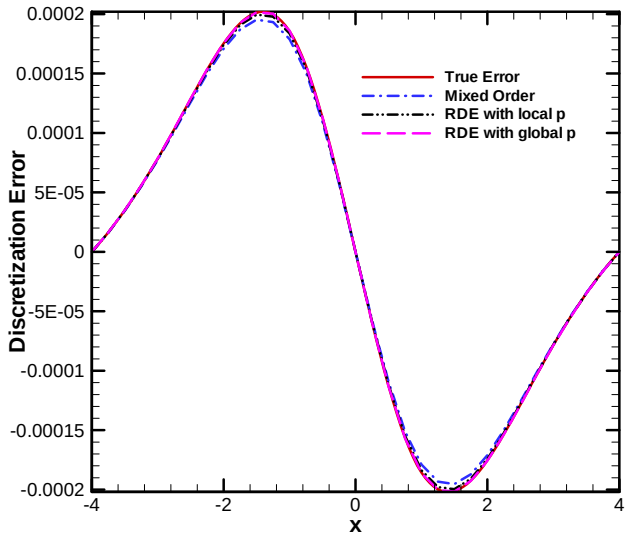

Fig 7.33 Discretization error estimators for the nearby problem, Re=8, finest mesh=129

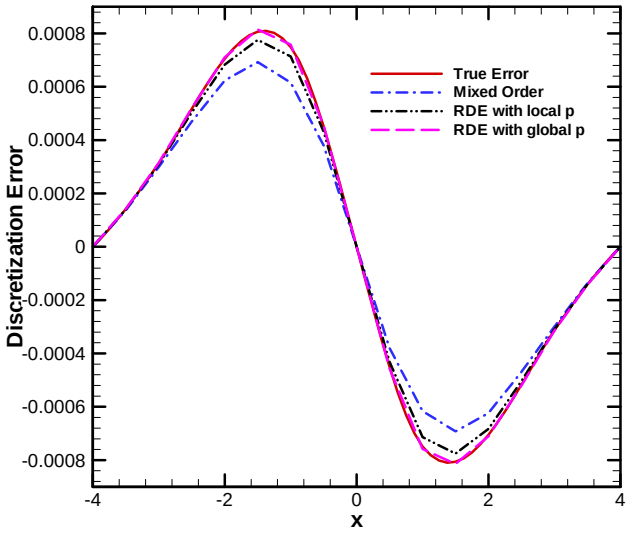

Fig 7.34 Discretization error estimators for the nearby problem, Re=8, finest mesh=65

For the Reynolds number of 64 case, we used 33 spline points. Fig. 7.35 shows the behavior of the error estimators with the finest mesh being 1025 points. In this case, we see that all the error estimators perform well. In Fig. 7.36, the finest mesh used was 257 points. The mixed-order error estimator and Richardson extrapolation with the local order of accuracy underestimate the error. In Fig. 7.37, the finest mesh that was used was 129 points. Here the mixed-order error estimator gives very poor estimates and Richardson extrapolation with the local order of accuracy also begins to fail. In Fig. 7.38, only Richardson extrapolation using the global order is used, because the other methods need three mesh solutions, but we cannot go coarser as the numerical solutions become unstable.

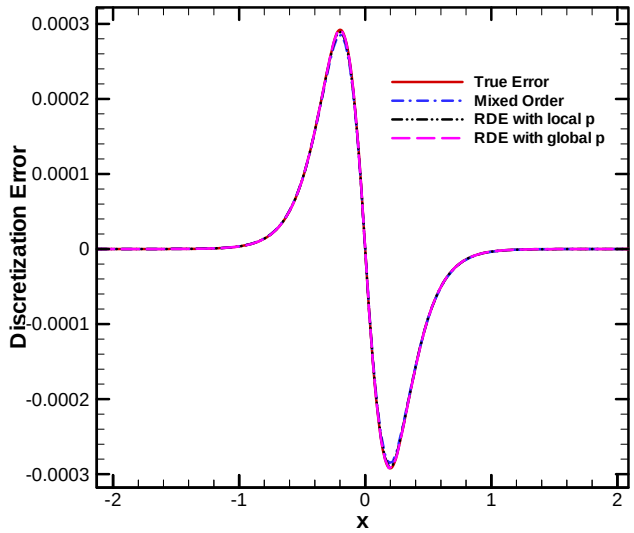

Fig 7.35 Discretization error estimators for the nearby problem, Re=64, finest mesh=1025

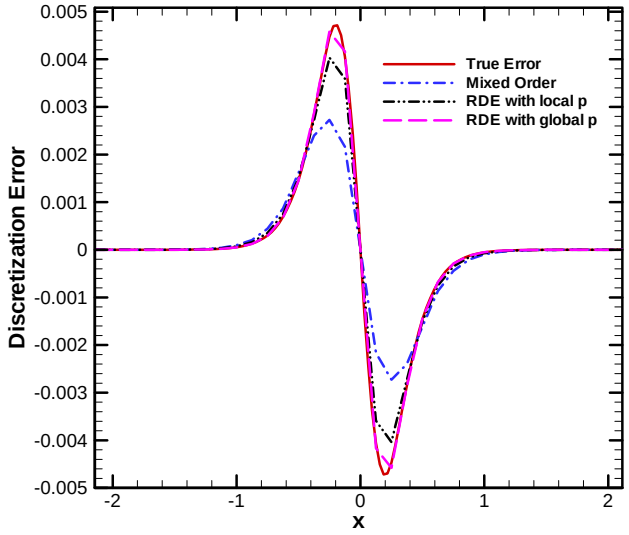

Fig 7.36 Discretization error estimators for the nearby problem, Re=64, finest mesh=257

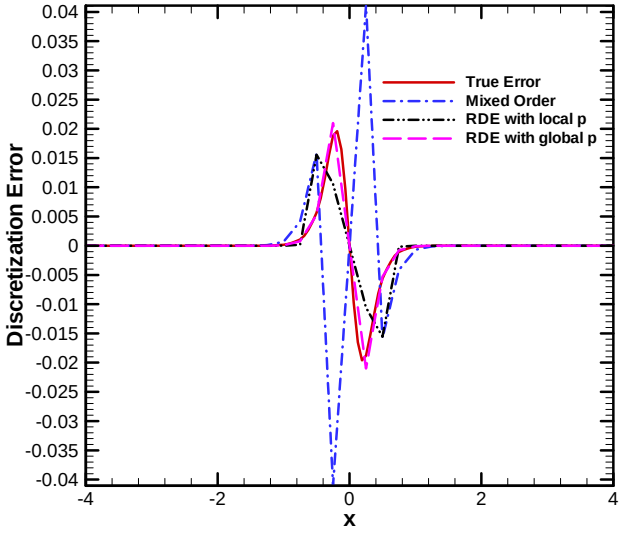

Fig 7.37 Discretization error estimators for the nearby problem, Re=64, finest mesh=129

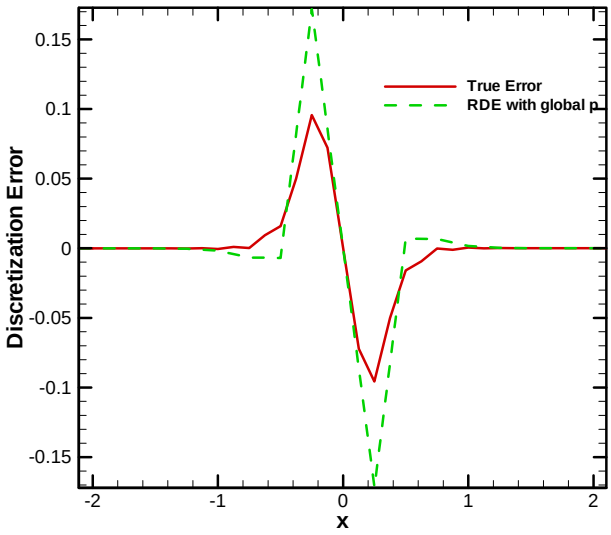

Fig 7.38 Discretization error estimators for the nearby problem, Re=64, finest mesh=65

If we compare the performance of different error estimators on original Burgers equation and the nearby problem to Burgers equation, we see that they are very similar. For example, for Re=8 with finest mesh being 65 points, the error estimator on the original Burgers equation (Fig. 7.25) are quite similar to the estimates on the nearby problem (Fig. 7.34). This is also true for the Re=64 case for original Burgers equation (Fig. 7.28) and the nearby problem (Fig. 7.36). This similarity justifies evaluating error estimators on the nearby problem.

The nearby problem to the modified form of the Burgers equation was solved and different error estimators were used to estimate the discretization error. In the modified form of Burgers equation, we ran only the Re = 64 case. Examining at Figs. 7.39 and 7.40, we see that as we reduce the number of nodes, Richardson extrapolation using local order of accuracy and the mixed order error estimator breaks down, while Richardson extrapolation using the global order of accuracy gives the best estimate. In Fig. 7.40,

mixed-order error estimator fails while Richardson extrapolation using the local order also gives bad estimates.

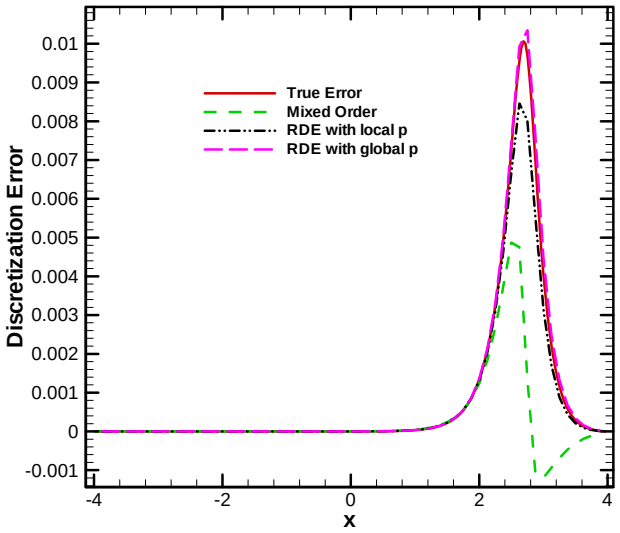

Fig 7.39 Discretization error estimators for nearby problem to modified Burgers equation, Re=64, nodes=257

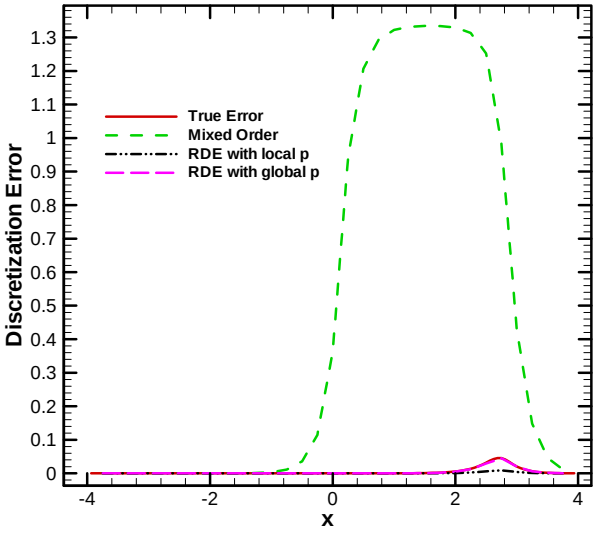

Fig 7.40 Discretization error estimators for nearby problem to modified Burgers equation, Re=64, nodes=129

CHAPTER EIGHT

CONCLUSIONS AND RECOMMENDATIONS

8.1. *Conclusions*

The current work on MNP has focused on two major goals. Using MNP to generate exact solutions to a problem nearby the steady-state Burgers equation in order to evaluate error estimators was the first goal. The second goal was to use MNP itself as an error estimator. To fulfill the first goal, a fifth-order Hermite spline was shown to be the best available curve fitting tool for Burgers equation. The polynomials fits were demonstrated to be accurate, and we were thus able to generate very small source terms. MNP was then used to evaluate various discretization error estimators and we found that Richardson extrapolation using the global order of accuracy provided the best estimates.

MNP itself was used as an error estimator. It was shown that MNP provided error estimates that were at least as good as Richardson extrapolation with the global order of accuracy, and often better. The use of MNP as an error estimator requires numerical solutions on only one mesh. This is an advantage of using MNP, as other error estimators required multiple meshes. The cost of MNP alone will be approximately the same as the cost of solving the original problem, with some additional overhead involved in the curve fitting procedure. So the total cost of using MNP as an error estimator will be approximately twice the cost of numerically solving the original problem of interest alone.

8.2. *Future work*

This research was focused on one-dimensional problems. The next step is to extend the MNP methodology to two-dimensions, three-dimensions, and four-dimensions (three spatial dimensions plus time). The biggest challenge is to develop a two-dimensional or higher-dimensional fit which has C^3 continuity. Rouff [17] has developed an approach for generating n-dimensional, C^k continuous splines. Implementation of Rouff's approach will allow the extension of MNP to more realistic two-dimensional and three-dimensional applications.

REFERENCES

1. Anderson, J. D., "Computational Fluid Dynamics: The Basics With Applications," McGraw-Hill, New York, 1995
2. Oberkampf, W. L., Roy, C. J., short course for Moldflow corporation on "Verification and Validation in Computational Simulation," Jul 22-23, 2004, Ithaca, New York
3. Roy, C. J., "Review of Code and Solution Verification Procedures for Computational Simulation" *Journal of Computational Physics*, Vol. 205, 2005, pp. 131-156
4. www.softwareqatest.com, access date: June 29, 2005
5. Ferziger, J. H., Peric, M., "Computational Methods of Fluid Dynamics," Springer-Verlag, Berlin, 1999
6. Raju, A., Roy, C. J., Hopkins, M. M., "Evaluation of Discretization Error Estimators using the Method of Nearby Problems," AIAA 2005-0684, 35th AIAA Fluid Dynamics Conference & Exhibit, Toronto, Canada
7. Roache, P. J., "Code Verification by the Method of Manufactured Solutions," *Journal of Fluids Engineering*. 124(1) (2002) 4-10
8. Knupp, P., Salari, K., "Verification of Computer Codes in Computational Science and Engineering," Chapman & Hall/CRC, New York, 2002
9. Lee, S., Junkins, J. L., "Construction of Benchmark Problems for Solution of ordinary Differential Equations," *Journal of Shock and Vibration*, Vol. 1, No. 5, 1994, pp. 403-414

10. Junkins, J. L, Lee, S., "Validation of Finite-Dimensional Approximate Solutions for Dynamics of Distributed-Parameter Systems," *Journal of Guidance, Control, and Dynamics*, Vol. 18, No. 1, 1995, pp. 87-95
11. Roy, C. J., and Hopkins, M. M., "Discretization Error Estimates using Exact Solutions to Nearby Problems," AIAA Paper 2003-0629, January 2003
12. Benton, E. R., and Platzman, G. W., "A Table of Solutions of the One-Dimensional Burger's Equation," *Q. Appl Math*, Vol. 30. pp. 195-212, 1972
13. Hopkins, M. M., Roy, C. J., "Introducing The Method of Nearby Problems," European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS 2004, P. Neittaanmaki, T. Rossi, S. Korotov, E. Onate, J. Periaux, and D. Knorzer (eds.), July 24-28, 2004
14. Matlab user manual
15. Mullges, G. E., Uhlig, F., "Numerical Algorithms with Fortran," Springer-Verlag, Berlin, 1996
16. Roy, C. J., "Grid Convergence Error Analysis for Mixed-Order Numerical Schemes," *AIAA journal*, Vol. 41, No. 4, April 2003
17. Rouff, M., "The Computation of Ck Spline Functions," Computers in Mathematical Applications, Vol 23, No. 1, 1992, pp. 103-110

APPENDICES

A. *Fortran program for solving original Burgers equation*

```
Program Steady_implicit
      implicit doubleprecision(a-h,o-z)

      Parameter(imax=81,lmax=imax-2)
      Dimension uold(imax),up(imax),unew(imax),x(imax)
      Dimension
uexact(imax),Disc_error(imax),AA(imax),BB(imax),CC(imax)
      Dimension G(imax),y(imax),ub(imax),unew1(imax),unew2(imax)


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxx    Initialization of constants   xxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
      rLen = 4.0d0
      a=-4.0d0
      b=4.0d0
      rNu=2.0d0
      rH=rLen*2.0d0/dfloat(imax-1)
      cfl=200.0d0
      rK=cfl*((rH*rH)/(rH + 2.0d0*rNu))
      n=imax-1
      th=1.0d0
      t=0.0d0
      Tol = 2.7e-12
      jmax = 400000000
      al=2.0d0
      alp=rK/(2.0d0*rH)
      bet=rNu*rK/rH**2

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxx     Setting up of Initial Conditions    xxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      Do i = 1,imax
         x(i) = a + (b-a)*dfloat(i-1)/dfloat(n)
!         uold(i)=0.0
         uold(i) = (-2.0d0*rNu*al/rLen)*(dsinh(x(i)*al/rLen))/
     &    (dcosh(x(i)*al/rLen)+dexp(-(t*rNu*al**2/rLen**2)))
!         Write(*,*) x(i)
      End Do
```

```
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxx     Setting up of Boundary Conditions     xxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      uold(1) = -2.0d0*al*rNu/rLen*dtanh(a*al/rLen)
      uold(n+1)=-2.0d0*al*rNu/rLen*dtanh(b*al/rLen)


      Do i=1,imax
         uexact(i)=-2.0d0*rNu*al/rLen*dtanh(x(i)*al/rLen)
      End Do


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxxxxxx      Main Loop    xxxxxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      Do j=1,jmax
         Do i=1,imax
             ub(i)=uold(i)
         End do


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxx     Setting up of Tridiagonal Matrix  xxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


         Do i=3,imax-2
             k=i-1
             AA(k)=-1.0d0*th*(alp*ub(i)+bet)
             BB(k)=(1.0d0+2.0d0*th*bet)
             CC(k)=th*(alp*ub(i)-bet)
             G(k)=uold(i)-((1.0d0-th)*alp*ub(i)*(uold(i+1)-uold(i-1)))+
     &        ((1.0d0-th)*bet*(uold(i+1)-2.0d0*uold(i)+uold(i-1)) )
         End Do

         BB(1)=(1.0d0+2.0d0*th*bet)
         CC(1)= (th*alp*ub(2)-th*bet)
         AA(imax-2)=-1.0d0*(th*alp*ub(imax-1)+th*bet)
         BB(imax-2)=(1.0d0+2.0d0*th*bet)
         G(1)=uold(2)-((1.0d0-th)*alp*ub(2)*(uold(3)-uold(1)))+
     &    ((1.0d0-th)*bet*(uold(3)-2.0d0*uold(2)+uold(1)))+(th*alp*
     &    ub(2)+th*bet)*uold(1)
         G(imax-2)=uold(imax-1)-((1.0d0-th)*alp*ub(imax-1)*(uold(imax)-
     &    uold(imax-2)))+((1.0d0-th)*bet*(uold(imax)-2.0d0*uold(imax-
1)+
     &    uold(imax-2)))-(th*alp*ub(imax-1)-th*bet)*uold(imax)

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxx     Calling the Tridiagonal Solver   xxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

         CALL TRDIAG (lmax,AA,BB,CC,y,G)
```

```
         Do i=1,imax-2
             unew(i+1)=y(i)
         End do

         unew(1)=uold(1)
         unew(n+1)=uold(n+1)
         Residue = 0.0d0

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxx    Computation of Residual    xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


         Do i=2,imax-1
           Res = unew(i)/(2.0d0*rH)*(unew(i+1)-unew(i-1))-rNu/(rH**2)*
     &      (unew(i+1)-2.0d0*unew(i)+unew(i-1))
           Residue = Residue + Res**2
         End Do
         rNorm=Residue
         rL2Norm =dsqrt(rNorm/dfloat(imax-2))
         if (j.eq.1) r1 = rL2Norm
         rL2Norm=rL2Norm/r1
         write(11,*) j,rK*dfloat(j),rK,rL2Norm
         if(mod(j,10).eq.0.0d0) then
             Write(19,*) j,rL2Norm
         End if

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxx    Checking Convergence    xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


         If (rL2Norm.LT.Tol) Goto 100

 200     Do i =1,imax
            uold(i) = unew(i)
         End Do
      End Do


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxx    End of Main Loop    xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


 100  Write(*,*) j

      do i=2,imax-1
        unew1(i)=(unew(i+1)-unew(i-1))/(2.0d0*rH)
      End do
      unew1(1)=(-3.0d0*unew(1)+4.0d0*unew(2)-unew(3))/(2.0d0*rH)
      unew1(imax)=(3.0d0*unew(imax)-4.0d0*unew(imax-1)+unew(imax-2))/
     &            (2.0d0*rH)
      unew2(1)=(-unew(4)+4.0d0*unew(3)-5.0d0*unew(2)+2.0d0*unew(1))/
     &         rH**2
      unew2(imax)=(2.0d0*unew(imax)-5.0d0*unew(imax-1)+4.0d0*
```

```
     &         unew(imax-2)-unew(imax-3))/rH**2
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxx  Writing out Solution    xxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      open(30,file='velocity_der_new_17.dat',status='unknown')
      open(40,file='distance_new_17.dat',status='unknown')
      open(50,file='velocity_new_17.dat',status='unknown')
      Do i=1, imax,5
         write(30,6) unew1(i)
         write(40,6) x(i)
         write(50,6) unew(i)
6        Format (e30.20)
      end do

       open(15,file='velocity.dat',status='unknown')
       open(10,file='distance1.dat',status='unknown')

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxx    Computation of Discretization Error    xxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      Do i=1,imax
        Write (10,500) x(i)
        Write (15,500) unew(i)
500     Format(e30.20)
        Write (16,501) x(i),unew(i),uexact(i)
        Disc_error(i)=(unew(i)-uexact(i))
        Write(14,*) x(i),Disc_error(i)
 501    Format(3(e30.20))
        Write(99,500) uexact(i)
      End Do

      rl2 = 0.0d0
      rl1 = 0.0d0

      Do i=1,imax
        error = dabs(uexact(i) - unew(i))
        error2 = error*error
        rl1 = rl1 + error
        rl2 = rl2 + error2
      End Do

      rl1 = rl1/dfloat(n+1)
      rl2 = dsqrt(rl2/dfloat(n+1))

      Write(*,*) 'L2Norm=',rl2
      Write(*,*) 'L1Norm=',rl1
      Write(*,8) unew2(1)
      Write(*,8) unew2(imax)
 8    Format(e30.20)
      close(30)
      close(40)
      close(50)
      End
```

```
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxx    Tridiagonal Solver Subroutine    xxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      SUBROUTINE TRDIAG (N,A,B,C,X,G)
      implicit doubleprecision(a-h,o-z)
      DIMENSION A(2000),B(2000),C(2000),X(2000),G(2000),BB(2000)
C!.....THIS SUBROUTINE SOLVES TRIDIAGONAL SYSTEMS OF EQUATIONS
C!.....BY GAUSS ELIMINATION
C!.....THE PROBLEM SOLVED IS MX=G WHERE M=TRI(A,B,C)
C!.....THIS ROUTINE DOES NOT DESTROY THE ORIGINAL MATRIX
C!.....AND MAY BE CALLED A NUMBER OF TIMES WITHOUT REDEFINING
C!.....THE MATRIX
C!.....N = NUMBER OF EQUATIONS SOLVED (UP TO 1000)
C!.....FORWARD ELIMINATION
C!.....BB IS A SCRATCH ARRAY NEEDED TO AVOID DESTROYING B ARRAY
      DO 1 I=1,N
      BB(I) = B(I)
    1 CONTINUE
      DO 2 I=2,N
      T = A(I)/BB(I-1)
      BB(I) = BB(I) - C(I-1)*T
      G(I) = G(I) - G(I-1)*T
    2 CONTINUE
C!.....BACK SUBSTITUTION
      X(N) = G(N)/BB(N)
      DO 3 I=1,N-1
      J = N-I
      X(J) = (G(J)-C(J)*X(J+1))/BB(J)
    3 CONTINUE
      RETURN
      END
```

B. ***Fortran program for solving the nearby problem to Burgers equation***

```
                        Program Steady_implicit
      implicit doubleprecision(a-h,o-z)

      Parameter(imax=81)
      Dimension uold(imax),unew(imax),x(imax),u(imax)
      Dimension
source(imax),Disc_error(imax),AA(imax),BB(imax),CC(imax)
      Dimension G(imax),ub(imax),y(imax)

      rLen = 4.0d0

 10    Format (e25.14)
```

```
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxx    Initialization of constants   xxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
      a=-4.0d0
      b=4.0d0
      rNu=2.0d0
      rH=rLen*2.0d0/dfloat(imax-1)
      cfl=100.0d0
      rK=cfl*((rH*rH)/(rH + 2.0d0*rNu))
      n=imax-1
      th=1.0d0
!     write(*,*) 'n = ',n
      t=0.0d0
      Tol = 1e-12
      jmax = 400000000
      al=2.0d0!255.828300733!7.99463621669!4.0!255.828300733
      alp=rK/(2.0d0*rH)
      bet=rNu*rK/rH**2

      open(60,file='distance1.dat',status='unknown')
      open(70,FILE='velocityterm.dat',status='unknown')
      open(66,file='source_17.dat',status='unknown')

      Do i=1,imax
          Read(60,*) x(i)
          Read(70,*) u(i)
          Read(66,*) source(i)
      End do

      close(60)
      close(70)
      close(66)

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxx     Setting up of Initial Conditions    xxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      Do i = 1,imax
         x(i) = a + (b-a)*dfloat(i-1)/dfloat(n)
         uold(i) = (-2.0d0*rNu*al/rLen)*(dsinh(x(i)*al/rLen))/
     &    (dcosh(x(i)*al/rLen)+dexp(-(t*rNu*al**2/rLen**2)))
      End Do

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxx     Setting up of Boundary Conditions     xxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      uold(1) = u(1)!-2.0d0*al*rNu/rLen*dtanh(a*al/rLen)
      uold(n+1)=u(imax)!-2.0d0*al*rNu/rLen*dtanh(b*al/rLen)


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxxxxxx      Main Loop    xxxxxxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
```

```
      Do j=1,jmax

         Do i=1,imax
             ub(i)=uold(i)
         End do


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxx     Setting up of Tridiagonal Matrix  xxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

         Do i=2,imax-1
             AA(i)=-1.0d0*th*(alp*ub(i)+bet)
             BB(i)=(1.0d0+2.0d0*th*bet)
             CC(i)=th*(alp*ub(i)-bet)
             G(i)=source(i)*rK+uold(i)-((1.0d0-
th)*alp*ub(i)*(uold(i+1)-
     &        uold(i-1)))+((1.0d0-th)*bet*(uold(i+1)-2.0d0*uold(i)+
     &        uold(i-1)) )
         End Do

         BB(1)=1.0d0
         CC(1)= 0.0d0
         AA(imax)=0.0d0
         BB(imax)=1.0d0
         G(1)=u(1)
         G(imax)=u(imax)

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxx     Calling the Tridiagonal Solver   xxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


         CALL TRDIAG (imax,AA,BB,CC,y,G)

         Do i=1,imax
             unew(i)=y(i)
         End do


         Residue = 0.0d0

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxx    Computation of Residual    xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!



         Do i=2,imax-1
           Res =unew(i)-uold(i)! unew(i)/(2.0d0*rH)*(unew(i+1)-unew(i-
1))-rNu/(rH**2)*
           Residue = Residue + Res**2
         End Do

         rNorm=Residue
         rL2Norm =dsqrt(rNorm/dfloat(imax-2))
```

```
         if (j.eq.1) r1 = rL2Norm

         rL2Norm=rL2Norm/r1
         write(11,*) j,rK*dfloat(j),rK,rL2Norm

         if(mod(j,10).eq.0.0d0) then
             Write(19,*) j,rL2Norm
         End if

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxxx    Checking Convergence    xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


         If (rL2Norm.LT.Tol) Goto 100

         Do i =1,imax
            uold(i) = unew(i)
         End Do

      End Do

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxxx    End of Main Loop    xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


 100  Write(*,*) j
       open(14,file='MNP.dat',status='unknown')

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxx    Computation of Discretization Error    xxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      Do i=1,imax
        Disc_error(i)=(unew(i)-u(i))/u(i)
        Write(14,8) x(i),Disc_error(i)
        Write (16,9) x(i),unew(i),u(i)
      End Do

 8    Format(2(e30.20))
 9    Format(3(e30.20))


      rl2 = 0.0d0
      rl1 = 0.0d0

      Do i=1,imax
        error = dabs(u(i) - unew(i))
        error2 = error*error
        rl1 = rl1 + error
        rl2 = rl2 + error2
      End Do

      rl1 = rl1/dfloat(imax)
      rl2 = dsqrt(rl2/dfloat(imax))
      Write(*,99) rl1
```

```
      Write(*,99) rl2
99    Format(e30.20)
      End



!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxx    Tridiagonal Solver Subroutine    xxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      SUBROUTINE TRDIAG (N,A,B,C,X,G)
      implicit doubleprecision(a-h,o-z)
      DIMENSION A(2000),B(2000),C(2000),X(2000),G(2000),BB(2000)
C!.....THIS SUBROUTINE SOLVES TRIDIAGONAL SYSTEMS OF EQUATIONS
C!.....BY GAUSS ELIMINATION
C!.....THE PROBLEM SOLVED IS MX=G WHERE M=TRI(A,B,C)
C!.....THIS ROUTINE DOES NOT DESTROY THE ORIGINAL MATRIX
C!.....AND MAY BE CALLED A NUMBER OF TIMES WITHOUT REDEFINING
C!.....THE MATRIX
C!.....N = NUMBER OF EQUATIONS SOLVED (UP TO 1000)
C!.....FORWARD ELIMINATION
C!.....BB IS A SCRATCH ARRAY NEEDED TO AVOID DESTROYING B ARRAY
      DO 1 I=1,N
      BB(I) = B(I)
    1 CONTINUE
      DO 2 I=2,N
      T = A(I)/BB(I-1)
      BB(I) = BB(I) - C(I-1)*T
      G(I) = G(I) - G(I-1)*T
    2 CONTINUE
C!.....BACK SUBSTITUTION
      X(N) = G(N)/BB(N)
      DO 3 I=1,N-1
      J = N-I
      X(J) = (G(J)-C(J)*X(J+1))/BB(J)
    3 CONTINUE
      RETURN
      END
```

C. *Fortran program to numerically solve the modified form of Burgers equation*

```
   Program Steady_implicit
      implicit doubleprecision(a-h,o-z)

      Parameter(imax=1025,lmax=imax-2)
      Dimension uold(imax),up(imax),unew(imax),x(imax)
      Dimension
uexact(imax),Disc_error(imax),AA(imax),BB(imax),CC(imax)
      Dimension G(imax),y(imax),ub(imax),unew1(imax),unew2(imax)
```

```
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxx    Initialization of constants   xxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      rLen = 4.0d0
      a=-4.0d0
      b=4.0d0
      rNu=0.25d0      !2.0d0,  0.25d0, 0.03125d0
      rH=rLen*2.0d0/dfloat(imax-1)
      cfl=1000.0d0
      rK=cfl*((rH*rH)/(rH + 2.0d0*rNu))
      n=imax-1
      th=1.0d0
      t=0.0d0
      Tol = 3e-12
      jmax = 400000000
      al=16.0d0
      alp=rK/(2.0d0*rH)
      bet=rNu*rK/rH**2
      xxp = 0.25

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxx     Setting up of Boundary Conditions    xxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      x(1) = a
      x(imax) = b

      uold(1) = 2.
      uold(n+1) = -2.


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxx     Setting up of Initial Conditions    xxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


     Do i = 2,imax-1

         x(i) = a + (b-a)*dfloat(i-1)/dfloat(n)
         uold(i) = uold(1)+(uold(n+1)-uold(1))*dfloat(i-1)/dfloat(n)
     End Do



      Write(11,*) 'TITLE = "Solution Profiles"'
      Write(11,*) 'variables="Iterations""rk*float(j)""rK""L2Norm"'

      Do i=1,imax
         uexact(i)=-2.0d0*rNu*al/rLen*dtanh(x(i)*al/rLen)
      End Do
```

```
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxxxxxx      Main Loop    xxxxxxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      Do j=1,jmax
         Do i=1,imax
             ub(i)=uold(i)
         End do

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxx     Setting up of Tridiagonal Matrix  xxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


         Do i=3,imax-2
             k=i-1
             betvar=bet*( (ub(i)/2.)**2 + ( (x(i)-a)/(b-a)+0.25)**xxp )
             AA(k)=-1.0d0*th*(alp*ub(i)+betvar)
             BB(k)=(1.0d0+2.0d0*th*betvar)
             CC(k)=th*(alp*ub(i)-betvar)
             G(k)=uold(i)-((1.0d0-th)*alp*ub(i)*(uold(i+1)-uold(i-1)))+
     &        ((1.0d0-th)*betvar*(uold(i+1)-2.0d0*uold(i)+uold(i-1)) )
         End Do

         betvar=bet*( (ub(2)/2.)**2 + ( (x(2)-a)/(b-a) + 0.25)**xxp )
         BB(1)=(1.0d0+2.0d0*th*betvar)
         CC(1)= (th*alp*ub(2)-th*betvar)
         G(1)=uold(2)-((1.0d0-th)*alp*ub(2)*(uold(3)-uold(1)))+
     &    ((1.0d0-th)*betvar*(uold(3)-2.0d0*uold(2)+uold(1)))+(th*alp*
     &    ub(2)+th*betvar)*uold(1)
         betvar=bet*( (ub(imax-1)/2.)**2 +
     &         ( (x(imax-1)-a)/(b-a) + 0.25)**xxp )
         AA(imax-2)=-1.0d0*(th*alp*ub(imax-1)+th*betvar)
         BB(imax-2)=(1.0d0+2.0d0*th*betvar)
         G(imax-2)=uold(imax-1)-
     &    ((1.0d0-th)*alp*ub(imax-1)*(uold(imax)-
     &    uold(imax-2)))+((1.0d0-th)*betvar*(uold(imax)
     &     -2.0d0*uold(imax-1)+
     &    uold(imax-2)))-(th*alp*ub(imax-1)-th*betvar)*uold(imax)

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxx     Calling the Tridiagonal Solver   xxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

         CALL TRDIAG (lmax,AA,BB,CC,y,G)

         Do i=1,imax-2
             unew(i+1)=y(i)
         End do

         unew(1)=uold(1)
         unew(n+1)=uold(n+1)
         Residue = 0.0d0
```

```
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxx    Computation of Residual    xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!



         Do i=2,imax-1
           rnuvar=rNu*( (unew(i)/2.)**2 + ( (x(i)-a)/(b-a)+0.25)**xxp )
           Res = unew(i)/(2.0d0*rH)*(unew(i+1)-unew(i-1))-
     &       rnuvar/(rH**2)*(unew(i+1)-2.0d0*unew(i)+unew(i-1))
           Residue = Residue + Res**2
         End Do
         rNorm=Residue
         rL2Norm =dsqrt(rNorm/dfloat(imax-2))
         if (j.eq.1) r1 = rL2Norm
         rL2Norm=rL2Norm/r1
         write(11,*) j,rK*dfloat(j),rK,rL2Norm
         if(mod(j,100).eq.0.0d0) then
             Write(19,*) j,rL2Norm
             write(*,*) j,rL2Norm
         End if

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxx    Checking Convergence    xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


         If (rL2Norm.LT.Tol) Goto 100

 200     Do i =1,imax
            uold(i) = unew(i)
         End Do
      End Do

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxx    End of Main Loop    xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

 100  Write(*,*) j


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxx  Writing out Solution    xxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      do i=2,imax-1
        unew1(i)=(unew(i+1)-unew(i-1))/(2.0d0*rH)
      End do
      unew1(1)=(-3.0d0*unew(1)+4.0d0*unew(2)-unew(3))/(2.0d0*rH)
      unew1(imax)=(3.0d0*unew(imax)-4.0d0*unew(imax-1)+unew(imax-2))/
     &            (2.0d0*rH)
      unew2(1)=(-unew(4)+4.0d0*unew(3)-5.0d0*unew(2)+2.0d0*unew(1))/
     &         rH**2
      unew2(imax)=(2.0d0*unew(imax)-5.0d0*unew(imax-1)+4.0d0*
     &         unew(imax-2)-unew(imax-3))/rH**2
      open(30,file='velocity_der_new_33_F.dat',status='unknown')
      open(40,file='distance_new_33_F.dat',status='unknown')
```

```
      open(50,file='velocity_new_33_F.dat',status='unknown')



 Do i=1, imax,32
         write(30,6) unew1(i)
         write(40,6) x(i)
         write(50,6) unew(i)
6        Format (e30.20)
      end do

      open(15,file='velocity_F.dat',status='unknown')
      open(10,file='distance1_F.dat',status='unknown')
      open(51,file='distance1_M.dat',status='unknown')
      open(52,file='distance1_C.dat',status='unknown')
      open(11,file='ubar_F.dat',status='unknown')

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxx    Computation of Discretization Error    xxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      Do i=1,imax
        Write (10,500) x(i)
        Write (15,500) unew(i)
        write(11,500) ub(i)
500     Format(e30.20)
        Disc_error(i)=(unew(i)-uexact(i))
        Write(14,*) x(i),Disc_error(i)
      End Do

      Do i=1,imax,2
        Write (51,501) x(i)
501     Format(e30.20)
      Enddo

      Do i=1,imax,4
        Write (52,502) x(i)
502     Format(e30.20)
      Enddo


      rl2 = 0.0d0
      rl1 = 0.0d0

      Do i=1,imax
        error = dabs(uexact(i) - unew(i))
        error2 = error*error
        rl1 = rl1 + error
        rl2 = rl2 + error2
      End Do

      rl1 = rl1/dfloat(n+1)
      rl2 = dsqrt(rl2/dfloat(n+1))
      write(*,*) rl1,rl2
```

```
      Write(*,8) rl2
!      Write(*,*) 'L1Norm=',rl1
      Write(*,8) unew2(1)
      Write(*,8) unew2(imax)
 8    Format(e30.20)
      close(30)
      close(40)
      close(50)
      End



!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxx    Tridiagonal Solver Subroutine    xxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      SUBROUTINE TRDIAG (N,A,B,C,X,G)
      implicit doubleprecision(a-h,o-z)
      DIMENSION A(2000),B(2000),C(2000),X(2000),G(2000),BB(2000)
C!.....THIS SUBROUTINE SOLVES TRIDIAGONAL SYSTEMS OF EQUATIONS
C!.....BY GAUSS ELIMINATION
C!.....THE PROBLEM SOLVED IS MX=G WHERE M=TRI(A,B,C)
C!.....THIS ROUTINE DOES NOT DESTROY THE ORIGINAL MATRIX
C!.....AND MAY BE CALLED A NUMBER OF TIMES WITHOUT REDEFINING
C!.....THE MATRIX
C!.....N = NUMBER OF EQUATIONS SOLVED (UP TO 1000)
C!.....FORWARD ELIMINATION
C!.....BB IS A SCRATCH ARRAY NEEDED TO AVOID DESTROYING B ARRAY
      DO 1 I=1,N
      BB(I) = B(I)
    1 CONTINUE
      DO 2 I=2,N
      T = A(I)/BB(I-1)
      BB(I) = BB(I) - C(I-1)*T
      G(I) = G(I) - G(I-1)*T
    2 CONTINUE
C!.....BACK SUBSTITUTION
      X(N) = G(N)/BB(N)
      DO 3 I=1,N-1
      J = N-I
      X(J) = (G(J)-C(J)*X(J+1))/BB(J)
    3 CONTINUE
      RETURN
      END
```

D. ***Fortran program to compute the coefficients of the Hermite spline polynomial***

```
                    Program Spline
      implicit doubleprecision(a-h,o-z)
      Parameter (imax=257,kmax=2000,nmax=33,lmax=nmax-2)
      Dimension x(kmax),u(kmax),a(kmax),b(kmax),c(kmax),d(kmax),h(kmax)
      Dimension
g(kmax),aa(kmax),bb(kmax),cc(kmax),u_num(kmax),u_1(kmax)
      Dimension e(kmax),f(kmax),y(kmax),z(kmax),p(kmax),q(kmax)
      Dimension term(kmax),de2(imax)


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxx  Reading Input Files    xxxxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      open(60,file='distance_new_33.dat',status='unknown')
      open(70,FILE='velocity_new_33.dat',status='unknown')
      open(80,FILE='velocity_der_new_33.dat',status='unknown')
      open(66,FILE='distance1.dat',status='unknown')
      open(55,FILE='velocity.dat',status='unknown')

      Do i=1,nmax
          Read(60,*) x(i)
          Read(70,*) u(i)
          Read(80,*) u_1(i)
!100       Format (F7.4)
      End do
      Do i=1,imax
          Read(55,*) u_num(i)
          Read(66,*) y(i)
      Enddo

      close(60)
      close(70)
      close(80)
      close(66)
      close(55)

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxx  Initializing Constants    xxxxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      toler=0.25d0
      rh=0.25d0
      rnu=0.250d0
      alp=1.0d0
      bet1=-0.5d0*0.9094947017729282400E-12/rh
      bet2=-0.5d0*0.2046363078989088500E-11/rh
```

```
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxx    Initializing ‘a’ and ‘b’ Coefficients    xxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
do i = 1,nmax
          a(i)=u(i)
          b(i)=u_1(i)
      end do

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxx    Setting up Tridiagonal Matrix for the Coefficient ‘c’    xxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      c(1)=-0.5d0*0.90949470177292824000E-12!0.5/h(1)*(3.0/h(1)*(a(2)-
a(1))-3.0*ral1-ral1/2.0
      c(nmax)=-0.5d0*0.20463630789890885000E-11!0.5*0.0101!ral2/2.0


      i=2
      k=i-1



      g(k)=10.0d0/rh**3*(a(3)-2.0d0*a(2)+a(1))+4.0d0/rh**2*(b(1)-b(3))+
     &      1.0d0/rh*c(1)

      i=nmax-1
      k=i-1


      g(k)= 10.0d0/rh**3*(a(nmax)-2.0d0*a(nmax-1)+a(nmax-2))+
     &      4.0d0/rh**2*(b(nmax-2)-b(nmax))+1.0d0/rh*c(nmax)


      do i=3,nmax-2
          k=i-1
          g(k)=10.0d0/rh**3*(a(i+1)-2.0d0*a(i)+a(i-1))+4.0d0/rh**2*
     &         (b(i-1)-b(i+1))
           aa(k)= -1.0d0/rh
           bb(k)= 6.0d0/rh
           cc(k)= -1.0d0/rh
      end do


      bb(1)=6.0d0/rh
      cc(1)=-1.0d0/rh
      aa(lmax)=-1.0d0/rh
      bb(lmax)=6.0d0/rh

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxx  Calling Tridiagonal Solver    xxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


      CALL TRDIAG (lmax,aa,bb,cc,z,g)

      Do i=1,nmax-2
```

```
          c(i+1)=z(i)
       End do

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxx    Computing'd', 'e', and 'f' Coefficients    xxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


       Do i=1,nmax-1
          d(i)=10.0d0/rh**3*(a(i+1)-a(i))-2.0d0/rh**2*(2.0d0*b(i+1)+
     &          3.0d0*b(i))+1.0d0/rh*(c(i+1)-3.0d0*c(i))
          e(i)=5.0d0/rh**4*(a(i+1)-a(i))-
1.0d0/rh**3*(b(i+1)+4.0d0*b(i))
     &      -3.0d0/rh**2*c(i)-2.0d0/rh*d(i)
          f(i)=1.0d0/(10.0d0*rh**3)*(c(i+1)-c(i)-3.0d0*d(i)*rh-6.0d0*
     &    e(i)* rh**2)
       End do

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxx  Writing out Coefficients xxxxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      open(20,file='OUTPUT_hermite_new_33.dat',status='unknown')


       Do i=1,nmax-1
          write(20,200) a(i), b(i), c(i), d(i), e(i), f(i)
200       Format(6(e30.20))
       End do

       close(20)

       open(77,file='G.dat',status='unknown')
       open(88,file='H.dat',status='unknown')

       Do i=1,imax-1
            Read(77,*) p(i)
            Read(88,*) q(i)
            Write(*,*) p(i),q(i)
       End do

       close(77)
       close(88)

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxx  Computing Value of the Polynomial  xxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


       Do i=1,imax
          Do j=1,nmax-1
              if((y(i)-x(j)).le.toler) goto 300
          enddo

300      term(i)=a(j)+b(j)*(y(i)-x(j))+c(j)*(y(i)-x(j))**2+d(j)*
     &         (y(i)-x(j))**3+e(j)*(y(i)-x(j))**4+f(j)*(y(i)-x(j))**5
       enddo
```

```
      rl1norm=0.0d0

      Do i=1,imax-1
         Do j=1,nmax
             if ((y(i)-x(j)).le.toler) goto 400
         enddo

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxx  Computing the Norm    xxxxxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!


400     rl1norm = rl1norm + dabs(((p(i)*y(i+1)+q(i)/2.0d0*y(i+1)**2)-
     &     (a(j)*y(i+1)+b(j)/2.0d0*(y(i+1)-x(j))**2+c(j)/3.0d0*(y(i+1)-
     &     x(j))**3+d(j)/4.0d0*(y(i+1)-x(j))**4+e(j)/5.0d0*(y(i+1)-
     &     x(j))**5+f(j)/6.0d0*(y(i+1)-x(j))**6))-((p(i)*y(i)+q(i)/
     &     2.0d0*y(i)**2)-(a(j)*y(i)+b(j)/2.0d0*(y(i)-x(j))**2+c(j)/
     &     3.0d0*(y(i)-x(j))**3+d(j)/4.0d0*(y(i)-x(j))**4+e(j)/5.0d0*
     &     (y(i)-x(j))**5+f(j)/6.0d0*(y(i)-x(j))**6)))

      end do

      rl1norm=rl1norm/8
      Write(*,145) rl1norm
145   Format(e30.20)
      open(22,file='term_33.dat',status='unknown')
      open(44,file='velocityterm.dat',status='unknown')
      Do i=1,imax
         Write(44,141) term(i)
141      format(e30.20)
         Write(22,140) y(i),term(i),u_num(i)
140      Format (3(e30.20))
      End do
      Do i=1,imax
         de2(i)=u_num(i)-term(i)
      End do
      open(23,file='de2.dat',status='unknown')
      Do i=1,imax
         Write(23,333) y(i),term(i),u_num(i),de2(i)
333      Format (4(e30.20))
      End do

      close(22)

      end



!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxx  Tridiagonal Solver Subroutine  xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

      SUBROUTINE TRDIAG (N,A,B,C,X,G)
      implicit doubleprecision(a-h,o-z)
      DIMENSION A(2000),B(2000),C(2000),X(2000),G(2000),BB(2000)
C!.....THIS SUBROUTINE SOLVES TRIDIAGONAL SYSTEMS OF EQUATIONS
```

```
C!.....BY GAUSS ELIMINATION
C!.....THE PROBLEM SOLVED IS MX=G WHERE M=TRI(A,B,C)
C!.....THIS ROUTINE DOES NOT DESTROY THE ORIGINAL MATRIX
C!.....AND MAY BE CALLED A NUMBER OF TIMES WITHOUT REDEFINING
C!.....THE MATRIX
C!.....N = NUMBER OF EQUATIONS SOLVED (UP TO 1000)
C!.....FORWARD ELIMINATION
C!.....BB IS A SCRATCH ARRAY NEEDED TO AVOID DESTROYING B ARRAY
      DO 1 I=1,N
      BB(I) = B(I)
    1 CONTINUE
      DO 2 I=2,N
      T = A(I)/BB(I-1)
      BB(I) = BB(I) - C(I-1)*T
      G(I) = G(I) - G(I-1)*T
    2 CONTINUE
C!.....BACK SUBSTITUTION
      X(N) = G(N)/BB(N)
      DO 3 I=1,N-1
      J = N-I
      X(J) = (G(J)-C(J)*X(J+1))/BB(J)
    3 CONTINUE
      RETURN
      END
```

E. ***Matlab program to calculate the source terms***

```
clc
clear all

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxxx  Reading in Coefficients xxxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

op=load('OUTPUT_hermite_new_17.dat');
y=load('distance1.dat');
x=load('distance_new_17.dat');
a=op(:,1);
b=op(:,2);
c=op(:,3);
d=op(:,4);
e=op(:,5);
f=op(:,6);
tol=0.50;
nu=2.0;


!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxx  Computation of Source Term   xxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

for i=1:81
    for j=1:16
        if y(i)-x(j)<=tol ,break,end
    end
```

```
    source(i)=(a(j)+b(j)*(y(i)-x(j))+c(j)*(y(i)-x(j))^2+d(j)*(y(i)-
x(j))^3+e(j)*(y(i)-x(j))^4+f(j)*(y(i)-x(j))^5)*(b(j)+2*c(j)*(y(i)-
x(j))+3*d(j)*(y(i)-x(j))^2+4*e(j)*(y(i)-x(j))^3+5*f(j)*(y(i)-x(j))^4)-
nu*(2*c(j)+6*d(j)*(y(i)-x(j))+12*e(j)*(y(i)-x(j))^2+20*f(j)*(y(i)-
x(j))^3);
end

!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!
 xxxxxxxxxxxxxxxxxxx  Plotting Source Term    xxxxxxxxxxxxxxxxxxx
!xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx!

source=source';
plot(y,source);
l1=0;
l2=0;
for i=1:81
    res2=source(i)^2;
    res1=abs(source(i));
    l1=l1+res1;
    l2=l2+res2;
end
l1=l1/81;
l2=(l2/81)^0.5;
```