COMPARISON OF FOUR CODES FOR SOLVING BOUNDARY VALUE PROBLEMS FOR ORDINARY DIFFERENTIAL EQUATIONS

by

Min Cao

B.Sc., Lanzhou University, 1985

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENT FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department
of
Mathematics & Statistics

Simon Fraser University

November, 1990

APPROVAL

Name: Min Cao

Degree: Master of Science (Numerical Analysis)

Title of Thesis: Comparison of Four Codes for Solving Boundary Value Problems for Ordinary Differential Equations

Examining Committee:

Chairman: Dr. A. Lachlan

[illegible]

Dr. R.D. Russell
Senior Supervisor

[illegible]

Dr. B. Bhattacharya

[illegible]

Dr. M. Trummer

[illegible]

Dr. T. Tang
External Examiner

Date Approved: November 8, 1990

Comparison of Four Codes for Solving Boundary Value
Problems for Ordinary Differential Equations

Abstract

The focus of this thesis is to compare four *codes* for solving boundary value problems (BVP) for ordinary differential equations (ODE), namely COLNEW, COLSYS, HAGRON and MUTS. Background information concerning the numerical methods that underlay these four codes, as well as information concerning how to use the four codes, is included. The four codes are compared according to many important criteria such as robustness, timing, and ease of use.

A considerable portion of this thesis is devoted to the discussion of various issues concerning the comparison of codes, such as the validity of conclusions resulting from a comparison of codes and the methods for making a comparison. The need for and the difficulties of comparing codes, in particular in our context of solving BVP for ODE, are discussed. A traditional method of comparison where the performance of the codes are compared under similar input parameter settings is examined. A different approach that compares the performance of the codes under the condition that the numerical solutions produced by the codes are of the similar quality is proposed. The relationship between these two approaches, as well as their relative merits, are also discussed.

To my parents

Acknowledgements

I am deeply indebted to my family, especially to my mother Kun-Xiang Lan, my father Kan Cao, for their constant encouragement, and to my wife Xiu-Qing Chen, for her full support and her help in typing this thesis.

I would also like to express my appreciation for the useful comments I received from Dr. B. Bhattacharya, Dr. M. Trummer, and Dr. T. Tang. I sincerely thank Mrs. S. Holmes for her keeping me informed of the important deadlines related to my defence and submission of this thesis, and Mrs. C.E. Dwyer for her help with my English.

Finally, I would like to thank my supervisor Dr. R.D. Russell for his patience, suggestions and many important discussions I have had with him in the one year period of writing this thesis.

Table of Contents

Introduction

The proliferation of mathematical software has increased the need to develop methods of evaluating the software, in order to determine their relative merits. The software user, on one hand, certainly needs information concerning the relative merits of the software in order to use them properly. The numerical analyst and the software developer, on the other hand, need to evaluate their own software in order to compare with others'. It becomes a common practice among the numerical analysts to use software evolution and comparison to provide justification for their methods and programs [6], [17].

a) Three commonly used methods of comparison

The evolution of software usually has three stages as illustrated by Figure 1. Naturally, three types of comparison are common: *1)* The comparison of methods by using some theoretical criteria, *2)* The comparison of algorithms (*e.g.* in terms of operation counts), *3)* The comparison of codes.

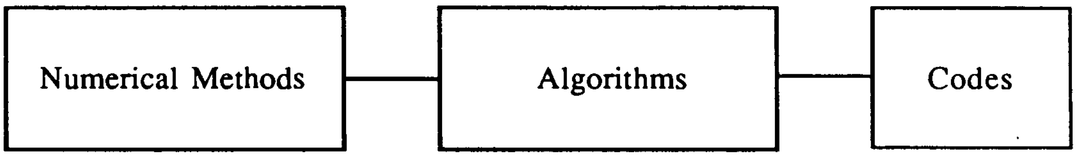


Figure 1

In a field of high complexity such as the numerical solution of boundary value problems for ordinary differential equations, the comparison of methods suffers technically from the fact that the numerical methods can be so different that they may have little common ground to be compared with. The comparison of algorithms also has serious drawbacks. Pereyra and Russell gave the following good example of the drawback of comparison of algorithms in [17]. Operation counts for collocation and Ritz method for solving elliptic PDE (partial differential equation) have been made in [18], [11], [9] and [26]. Each contains modifications of the algorithms giving improved operation counts over the previous results. While the second and the third view collocation as more efficient, both in terms of the counts and the authors' resulting codes, the fourth improves upon the counts for Ritz method and concludes that Ritz is more efficient.

Apart from the drawbacks of the first two types of comparison, the need for the comparison of codes is also reflected by the facts that neither the comparison of methods nor the comparison of algorithms provide certain information about the codes. Figure 2 further illustrates this point. The fact that there are different ways in which a method or an algorithm can be implemented severely weakened the connection between the observations resulting from the first two types of comparison and the observations from the comparison of two specific codes. The comparison of codes therefore cannot be replaced by a trivial extension of the observations from either one or both of the first two types of comparison. It is our view that the results from the three different types of comparison are not that closely related and consistent as they appear to be. Neither does the superiority of an algorithm resulting from the second type of comparison imply the superiority of some codes based upon that algorithm, nor does a superior method necessarily lead to a superior code. In one words, in the process of software evolution, superiority at one stage may not be inherited by the stages that follows. This puts the comparison of codes into an irreplaceable position of its own and it often makes the first two types of comparison less attractive, especially to the user of the codes, since for them, it is the performance of the codes, not the superiority of the methods or the algorithms, that counts the most.

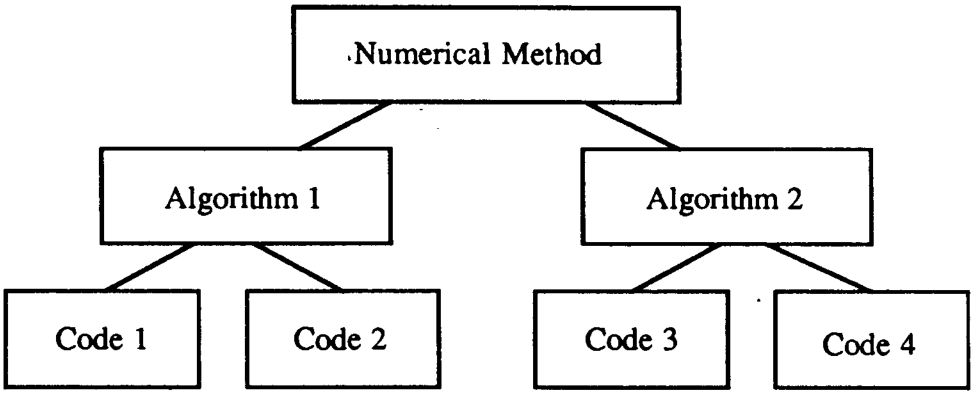


Figure 2

One may notice that so far we have not fully explained what we mean by 'superior' and what we mean by the comparison of methods or the comparison of algorithms. Unfortunately, the literature of software evaluation is by no means near maturity. There are few criteria of comparison that everybody agrees on. Thus 'superior' and 'comparison' are very much subjective concepts. While it seems unrealistic to have a set of criteria for evaluating all kinds of software, if we restrict ourselves in a specific field, meaningful comparison is not impossible.

b) An overview of this thesis

In this thesis, we shall concentrate on comparison of four codes* for solving boundary value problem (BVP) for ordinary differential equation (ODE). Our purpose is to discuss various issues concerning the comparison of codes in this specific field through actually comparing four of them. Users of these codes may also benefit from our discussion.

The first chapter of this thesis covers some background material concerning the boundary value problem for ordinary differential equation and the basic numerical methods that eventually lead to the four codes. The last three chapters discuss the criteria for comparison and describe the four codes that we are interested in. Comparison of these codes centred on the discussed criteria is then conducted.

More specifically, Chapter One serves as an introduction to BVP for ODE and the initial value problem (IVP) for ODE. It also contains description of some simple numerical methods that aimed at solving IVP and BVP for ODE. The difficulties of BVP for ODE are examined from a numerical analysis point of view.

Chapter Two is devoted to various criteria and difficulties for comparing mathematical software, in particular, the criteria and difficulties for comparing ODEBVP codes. It also carries on the discussion about the relationships among the three types of comparison we mentioned in the last section. No sophisticated mathematics is involved in this chapter. Users of BVP codes may find this chapter useful.

Chapter Three consists mainly of the description of the four codes we compare. Since all of them can be fairly difficult to use, especially to users who have little experience with BVP for ODE, and some of them do not have complete documentation available due to the undergoing changes, our description is aimed at diminishing this problem by providing guidance on how to use them.

Chapter Four is a detailed report of the comparison of the four codes. It covers both how the comparison is conducted and what we observed through our comparison. It is important that the observations are only valid for the versions of the codes we currently have. However, some of the observations are unlikely to be affected by any future implementations. We shall point this out whenever we go through situations of this kind.

* The list of the four codes and the date they were received:

COLNEW by Bader, G. and Ascher, U.. Received in October 1988.
COLSYS by Ascher, U., Christiansen, J., and Russell, R. D.. Received in July 1988.
HAGRON by J.R. Cash and Margaret H. Wright. Received in May, 1990.
MUTS by R. M. M. Mattheij and G. W. M. Staarink. Received in November 1988.

Chapter 1 BVP for ODE and Some Related Issues

This chapter is concerned with background material about BVP for ODE including the general formulation of a BVP for ODE, the numerical methods for BVP and the numerical methods for the related problem of solving IVP for ODE. Section 1 discusses various forms of ordinary differential equations and boundary conditions. Section 2 describes briefly how a few typical IVP methods work. Section 3 describes the basic idea of those numerical methods that are related to the four codes. We do not intend to provide detailed discussion about IVP and BVP theory. Also we assume that the reader has certain numerical analysis background. Thus we merely describe some basic ideas to acquaint the reader with how IVP and BVP are solved numerically. We shall also emphasize those issues that separate BVP methods from IVP methods.

1.1 BVP for Ordinary Differential Equation

Example *1.1.1*

A simple second order IVP has the form

$$y'' = f(x, y, y') \qquad x>a \qquad (1.1.1a)$$

$$y(a) = \alpha_1, \ y'(a) = \alpha_2 \qquad (1.1.2a)$$

Similarly, a simple second order BVP has the form

$$y''=f(x,y,y') \qquad a<x<b \qquad (1.1.1a)$$

$$y(a) = \beta_1, y(b) = \beta_2 \qquad (1.1.2b)$$

If $f(x,y,y')$ is linear in y and y', then we have a linear equation. It takes a simpler form

$$y''(x) - c_1(x)y'(x) - c_0(x)y(x) = q(x). \qquad (1.1.1b)$$

where $c_0(x)$, $c_1(x)$ and $q(x)$ are functions of x. Usually one assumes that general ordinary differential equation can be written as a first order system

$$y'=f(x,y) \qquad a<x<b \qquad (1.1.3a)$$

where $y(x)=(y_1(x),...,y_n(x))^T$ is the unknown function and

$$f(x,y)=(f_1(x,y),\cdots f_n(x,y))^T$$

is generally a non-linear right hand side. For linear problems the ODE simplifies to

$$y'=A(x)y+q(x) \qquad a<x<b \qquad (1.1.3b)$$

where the matrix $A(x)\in R^{n\times n}$ and the vector $q(x)\in R^n$ are functions of x.

Some of the BVPODE codes deal with first order systems exclusively. Among the four codes we are interested in, HAGRON and MUTS require the user to change high order equations into a first order system before they can be applied. To show how to convert a high order ODE into a first order system (also see [2]), we consider the general form of the mixed order ODE system that has d equations and the i^{th} equation is of order m_i

$$u_n^{(m_n)}(x) = f_n(x;\, u_1,\, u_1',\, \cdots,\, u_1^{(m_1-1)},\, u_2,\, u_2',\, \cdots,\, u_d,\, \cdots,\, u_d^{(m_d-1)}) \qquad (1.1.4)$$

$$a<x<b, \qquad n=1,2,\cdots,d$$

where m_n's are integers and f_n's are generally nonlinear functions.

Let $z(u) = (u_1,u'_1,\cdots,u_1^{(m_1-1)},u_2,\cdots,u_d,\cdots,u_d^{(m_1-1)})^T$, then $(1.1.4)$ is converted to the form of $(1.1.3a)$

$$z'_1 = z_2$$

$$z'_2 = z_3$$

$$\cdots\cdots$$

$$z'_{m_1} = f_1(x;z)$$

$$z'_{m_1+1} = z_{m_1+2}$$

$$\cdots\cdots$$

$$z'_M = f_d(x;z(u)) \qquad (M = \sum_{k=1}^{d} m_k)$$

A first order system of ODE *(1.1.3a)* is normally supplemented by a two-point boundary condition

$$g(y(a), y(b)) = \mathbf{0} \qquad (1.1.5a)$$

where $g = (g_1, g_2, \cdots, g_n)^T$ is generally a nonlinear function and $\mathbf{0}$ is the zero vector in R^n. One may notice that *(1.1.5a)* is a special case since only two end points are involved. Nevertheless, it is the most popular form of boundary condition for an ODE, and most BVPODE codes can only be applied when the boundary condition for an ODE is in this form. When g is linear in $y(a)$ and $y(b)$, we have linear boundary condition. The general form of linear two-point boundary condition for a first order system is

$$B_a y(a) + B_b y(b) = \beta \qquad (1.1.5b)$$

where $B_a, B_b \in R^{n\times n}, \beta \in R^n$. A very important case arises when *(1.1.5b)* simplify to (also see Chapter One in [2])

$$\begin{aligned} B_a y(a) &= \beta_1 \\ B_b y(b) &= \beta_2 \end{aligned} \qquad (1.1.5c)$$

(1.1.5c) is called *separated linear boundary conditions*. Similarly, if *(1.1.5a)* simplify to

$$\begin{aligned} g_1(y(a)) &= \mathbf{0} \\ g_2(y(b)) &= \mathbf{0} \end{aligned} \qquad (1.1.5d)$$

then we have *separated nonlinear boundary condition*. A significant portion of the currently available software for BVP assumes that the boundary conditions are separated. In fact, general boundary conditions *(1.1.5a)* can always be converted to one with separated boundary conditions [2]. More general boundary conditions arise when there are some other points rather than just two end points that are involved in a boundary condition. We call this kind of boundary condition a multipoint boundary condition. Linear multipoint boundary conditions have the form

$$\sum_{j=1}^{J} B_j y(x_j) = \beta \qquad (1.1.5e)$$

where $B_1, B_2, \cdots, B_J \in R^{n\times n}, \beta \in R^n$ and $a = x_1 < x_2 < \cdots < x_J = b$. The most general form of multipoint boundary condition of interest is of the form

$$g_j(z(y(x_j))) = 0 \qquad 1 < j < J \qquad (1.1.5f)$$

where $a = x_1 < x_2 < \cdots < x_J = b$ and g_i's are *linear* functions.

(1.1.4) together with *(1.1.5f)* form the most complicated BVP for ODE that can be directly handled by the available BVPODE codes so far.

A multipoint BVP, like a high order ODE, can be converted to simpler form [2]. By transforming each of the subintervals $[x_j, x_{j+1}]$ onto a fixed interval, say, $[0,1]$ and writing the ODE for the independent variable

$$t = \frac{x - x_j}{x_{j+1} - x_j} \qquad for\ j = 1,2,\cdots,n$$

one can transfer a multipoint BVP into a two-point BVP. To see how this works, We consider the following example:

$$y' = A(x)y + q(x) \qquad a<x<b \qquad (1.1.6)$$

$$\sum_{j=1}^{J} Bj\, y(xj) = \beta \qquad (1.1.7)$$

where $A(x),\ B_1,\ldots,B_J \in R^{n\times n};\ q(x),\ \beta \in R^n$, and $a = x_1 < x_2 < \ldots < x_J = b$ with $t = \frac{x - x_j}{x_{j+1} - x_j}$ for $j = 1,\cdots,J-1$. *(1.1.6)* becomes n ODEs on $[0,1]$. We thus have in total $(J-1)\times n$ ODEs. With new variable $t \in [0,1]$, *(1.1.7)* becomes n B.C. which are specified at interval endpoints 0 and 1. These n new B.C. together with the $n\times(J-2)$ additional B.C. resulting from the interior break points $x_j, j = 2,3,\cdots,J-1$ will be the new B.C. for the $(J-1)\times n$ ODEs resulting from the transformation.

HAGRON and MUTS solve only first order systems with two-point boundary conditions. Using the above transformation, one can change any multipoint BVP into a two-point BVP and then apply the codes. Thus the above transformation in some sense justified these codes that deal with two-point BVP only. However, such a transformation cannot be done without cost. As one can see from the example above, the size of the system is dramatically increased after the transformation. When it comes to solving the BVP by numerical means, this kind of size increase means that the process will be more expensive than solving a similar system in its original size. Sometimes the transformed system can be simply out of the reach of the BVPODE codes, because of various reasons such as a code may be designed to handle an ODE system of limited size.

1.2 A Few Methods for Solving IVP for ODE

Compared with the development of numerical methods for solving BVP, literature for solving IVP was much earlier in maturing. Robust codes for IVP appeared long before the births of the four codes we are about to discuss. Big software packages such as NAG or IMSL have many reliable routines for IVP, but have only a few routines for BVP. This is mainly due to the facts that BVP are generally more difficult than IVP in the context of numerical analysis, and the difficulties of BVP was not fully recognized until the last decade. To understand the difficulties of BVP, it is essential to know how an IVP and a BVP may be solved numerically. In this section, we will discuss how some numerical methods for IVP work.

Consider a simple initial value problem

$$u' = f(t,u) \qquad x>0 \qquad (2.2.1a)$$

$$u(0) = u_0 \qquad (2.2.1b)$$

A partition of the domain $0 = t_0 < t_1 <\cdots<t_k<\cdots$ generally has varied step sizes $h_i = t_{i+1} - t_i$, *i.e.* the value of h_i depends on i. For simplicity, we will only consider the special cases where h_i's are all equal to a constant h. There are many IVP methods that can be used to solve *(2.2.1a&b)*. Roughly, they can be divided into two types. The first type of methods is the so called 'one step methods'. These one step methods calculate u_{i+1}, the numerical solution at point t_{i+1}, using only information at t_i. The second type of methods is the multistep methods by which not only information at the point t_i but also information at $t_{i-1}, t_{i-2},\cdots,t_{i-m} (m >1)$ is used to calculate u_{i+1}.

Example 1.2.1: Euler's Method

The well known Euler's method is an explicit one step method where

$$u_{i+1} = u_i + hf(t_i,u_i). \qquad (2.2.2)$$

The motivation for this formula is linear extrapolation [25], as suggested in Figure 1.2.1. If u_0 is given (set equal to the initial value $u(0)$), it is a straightforward matter to apply *(2.2.2)* to compute successive values $u_1, u_2, \cdots$, as is illustrated by the simple algorithm below.

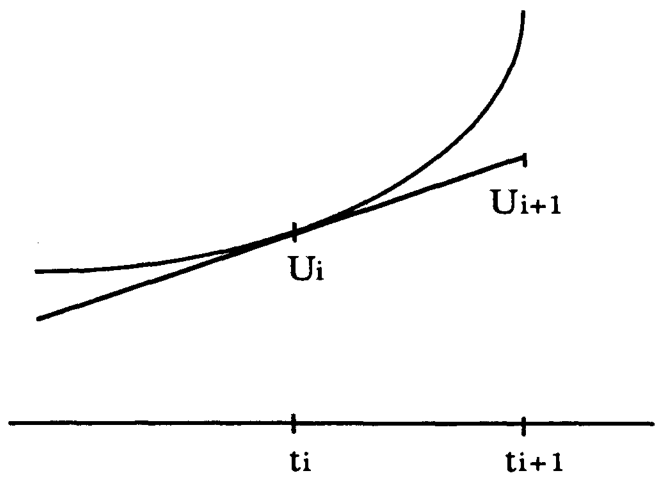


Figure 1.2.1 Euler's Method

Algorithm for Euler's method:

Step 1: $u_0 = u(0)$

Step 2 : $u_{i+1} = u_i + hf(t_i,u_i)$ $\qquad i = 1,2,\cdots,n$

There are two common ways that a one step method may be derived. One way is to use Taylor expansion and another way is to use numerical quadrature. To see how Euler's formula is derived, we expand the exact solution for *(2.2.1a&b)* u at t_i

$$u(t_{i+1}) = u(t_i) + (t_{i+1} - t_i)u'(t_i) + R(t_{i+1}),$$

that is
$$u(t_{i+1}) = u(t_i) + hf(t_i,u(t_i)) + R(t_{i+1}).$$

As $R(t_{i+1})$ is of order of h^2, it is negligible provided that step size h is small. When $R(t_{i+1})$ is negligible, the above equation is well approximated by *(2.2.2)* in the sense that u_i 's satisfying *(2.2.2)* are good approximations of $u(t_i)$'s. Assuming that h is small and $R(t_{i+1})$ is negligible, by dropping $R(t_i)$ and replacing the exact solution $u(t_k)$ by numerical approximation u_k in the Taylor expansion above, we get Euler's formula. One the other hand, one can use the fundamental theorem of calculus and use the following equation

$$u(t_{i+1}) - u(t_i) = \int_{t_{i+1}}^{t_i} u'(t)\, dt$$

that is

$$u(t_{i+1}) - u(t_i) = \int_{t_{i+1}}^{t_i} f(t,u(t))\, dt$$

Replacing the integral in the right hand side of the above equation by the quadrature formula that approximates the integral using $(t_{i+1}-t_i)f(t_i,u(t_i))$ and replacing the $u(t_i)$'s by u_i 's as we did above, we then have the Euler's formula.

Incidentally, most of the numerical methods for solving ODE (not just one step methods) can also be derived by the two approaches we briefly explained above. For more concerning the derivation of the numerical methods and the theoretical aspects of the methods, please see [2] and [25].

Assume we know u_i's at the first m - 1 meshpoints:

$$\begin{array}{cccc} t_0 & t_1 & \cdots & t_{m-1} \\ u_0 & u_1 & \cdots & u_{m-1} \end{array}$$

An explicit m step method has the form

$$u_{k+1} = \varphi(t_k,h,u_k,u_{k-1},\cdots,u_{k-m})$$

and an implicit m step method has the form

$$u_{k+1} = \varphi(t_k,h,u_{k+1},u_k,u_{k-1},\cdots,u_{k-m})$$

where $\varphi(t_k,h,u_k,u_{k-1}\ldots,u_{k-m})$ and $\varphi(t_k,h,u_{k+1},u_k,u_{k-1}\ldots,u_{k-m})$ are known functions.

Example 2.2.2: Some examples of multistep schemes are given below.

The Adams-Bashforth explicit four step method is

$$u_{i+1} = u_i + \frac{h}{24}[\ 55f(t_i,u_i) - 59f(t_{i-1},u_{i-1}) + 37f(t_{i-1},u_{i-2}) - 9f(t_{i-3},u_{i-3})\ ]$$

The Adams-Moulton implicit four step method is

$$u_{i+1} = u_i + \frac{h}{24}[\ 9f(t_{i+1},u_{i+1}) + 19f(t_i,u_i) - 5f(t_{i-1},u_{i-1}) + f(t_{i-2},u_{i-2})\ ]$$

In order to use an explicit *m* step method to solve *(1.2.1a&b)*, one needs to know $u_1,u_2,\ldots,u_{m-1}$ to proceed. This is usually done by using a one step method, e.g. Euler's method. A simple algorithm for an Adams-Bashforth method would be

Step 1: Calculate u1,u2,u3 by using some one step method

Step 2: $u_{i+1} = u_i + \frac{h}{24}[\ 55f(t_i,u_i) - 59f(t_{i-1},u_{i-1}) + 37f(t_{i-1},u_{i-2}) - 9f(t_{i-3},u_{i-3})\ ]$

for $i = 4,5,6,\cdots$

The implementation of an implicit *m* step method is more complicated. While one can still rely on a one step method such as Euler's method to calculate $u_1, u_2,\cdots, u_{m-1}$, more efforts are needed in the second step in order to calculate $u_m,u_{m+1},\cdots$. One commonly used strategy is known as the Predictor-Corrector method. We use the following simple algorithm for an Adams-Moulton method as an example to illustrate how an implicit multistep method with a Predicator-Corrector technique is typically implemented

Step 1: Calculate u_1, u_2, u_3 *by using some one step method*

Step 2: Calculating u_i *for* $i = 4,5,6,\cdots$

2.1 Estimate u^0_{i+1} *by some method, e.g. Adams-Bashforth scheme.*

2.2 Calculate $f^n = f(t_{i+1},u^n_{i+1})$

2.3 $u^{n+1}_{i+1} = u_i + \frac{h}{24}[\ 9f^n + 19f(t_i,u_i) - 5f(t_{i-1},u_{i-1}) + f(t_{i-2},u_{i-2})\ ]$

2.4 Go back to 2.2 until u^{n+1}_{i+1} *converges to a satisfactory accuracy.*

Generally, a simple one step method like Euler's method can be easily carried out. A multistep method, especially an implicit multistep method, needs more effort. But its order of truncation error is usually higher than that of Euler's method. Euler's method has a local truncation error of order 1. In our examples, both Adams-Bashforth and Adams-Moulton's methods are of order two [25]. Among the numerical methods for ordinary differential equations, one family of methods, known as Runge-Kutta methods, have been very widely used since it has the advantages of both a simple one step method and a multistep method, *i.e.* it is a family of high order one step methods. We only look at a simple example of an explicit Runge-Kutta method for IVP in this section, and leave the discussion about general Runge-Kutta methods to the next section. Like the one step Euler's method, an explicit Runge-Kutta scheme also has the form

$$u_{i+1} = u_i + h\varphi(t_i,u_i,h).$$

but the function $\varphi(t_i,u_i,h)$ is no longer necessarily a linear function of u_i and f. The following example is a second order Runge-Kutta method known as Heun's method.

Example 2.2.3: Heun's Method is

$$u_{i+1} = u_i + \frac{h}{4}[\,f(t_i,u_i) + 3f(t_i + \frac{2}{3}h,\, u_i + \frac{2}{3}hf(t_i,u_i))].$$

Since it is an explicit method, in order to apply it to solve IVP, the algorithm described in *example 2.2.1* can also be used for its implementation.

The above are a few well known IVP methods. What they have in common is that they are all *local* in nature. Namely the methods are based on the relationships of the numerical solution at only a few neighbouring (*local*) mesh points. *e.g.* an Adams-Bashforth method is solely based on the a relationship among u_{i+1}, u_i, u_{i-1}, u_{i-2}, u_{i-3}. The fact that complete information about the solution at the initial point is known enables these methods to be proceeded iteratively in a fixed direction in the sense that the numerical solution values at the mesh points $u_1, u_2,\cdots, u_k,\cdots$ are calculated *one at a time* in the order of their corresponding mesh points. It turns out that this is what separates BVP methods from IVP methods (also see [2]). It is also the major reason that an IVP is generally easier to solve numerically than BVP as one will see as we continue our discussion.

1.3 Some ideas about solving BVP

The detailed methods and techniques involved in designing the four codes we are interested in varies from code to code to different degrees. But the basic ideas that lies behind them can be roughly divided into two types. The first type of ideas are those that suggest one solve a BVP for ODE via solving its related IVP. Numerical methods based on these ideas are referred to as initial value methods (for BVP). The second type of ideas involve spline-collocation or implicit Runge-Kutta approach, and the related methods are referred to as finite difference methods. Among the four codes we are interested in, only MUTS is based on a method that belongs to the first type [15]. COLNEW and COLSYS use spline-collocation [3], [1]. HAGRON is based on a special implicit Runge-Kutta methods with deferred correction [4], [5], [6].

The first type of ideas is very natural in the sense that to construct a method for solving BVP by relating a BVP to its corresponding IVP, one can then take full advantage of the existing numerical tools for solving IVP. To see how this might be done, we consider the following simple example

Example 1.3.1 Single shooting method for general linear two-point BVP

Consider a general linear two-point BVP

$$y' = A(x)y + q(x) \qquad a<x<b \qquad (1.3.1a)$$

$$B_a y(a) + B_b y(b) = \beta \qquad (1.3.1b)$$

where $A(x), B_a, B_b \in R^{n \times n}$ and $y(x), q(x), \beta \in R^n$

If the general solution of *(1.3.1a)* $y(x)$ can be expressed as

$$y(x) = Y(x)s + v(x) \qquad a \leq x \leq b \qquad (1.3.2)$$

where $Y(x)$ is a matrix function, s is a parameter vector $(s \in R^n)$ and $v(x)$ is a particular solution of *(1.3.1a)*, and if *(1.3.1a&b)* has a unique solution, then there must exist an ***unique*** s that corresponds to this solution. To find such a s, let's substitute *(1.3.2)* into *(1.3.1b)*

$$B_a[Y(a)s + v(a)] + B_b[Y(b)s + v(b)] = \beta$$

or $$[B_aY(a) + B_bY(b)]s + B_av(a) + B_bv(b) = \beta.$$

Letting $$Q = B_a Y(a) + B_bY(b)$$

$$\beta^* = \beta - B_a v(a) - B_b v(b)$$

then we have $$Qs = \beta^*. \qquad (1.3.3)$$

Thus s can be calculated by using *(1.3.3)* and *(1.3.1a&b)* is then solved. However, there still remain a few important questions that need to be answered: *1)* Does there exist a general solution of the form *(1.3.2) ? 2)* Is there a unique solution to *(1.3.3)*, *i.e.* is the Q matrix nonsingular ? *3)* What does the above process of solving BVP have to do with solving IVP ?

Under the condition that $A(x)$, $q(x)$ are continuous on $[a,b]$, and *(1.3.1a&b)* has an unique solution, one can show that the n homogeneous first-order systems

$$Y'(x) = A(x)Y(x) \qquad a<x<b \qquad (1.3.4a)$$

$$Y(a) = I \qquad (1.3.4b)$$

where $Y(x), A(x), I \in R^{n \times n}$, I is identity matrix and the first-order linear system

$$v'(x) = A(x)v(x) + q(x) \qquad a<x<b \qquad (1.3.5a)$$

$$v(a) = \alpha \qquad (\alpha \in R^n) \qquad (1.3.5b)$$

will all have unique solutions [2]. Using the solutions of *(1.3.4a&b)* and *(1.3.4a&b)*, one can form a general solution of *(1.3.1a)* by substituting the solutions into *(1.3.2)*. Furthermore, the resulting Q matrix in *(1.3.3)* is then a nonsinguler matrix. To answer the third question, we notice that both *(1.3.4a&b)* and *(1.3.5a&b)* are IVPs. Thus the method described above can be implemented by using the following algorithm that involves heavy use of an IVP code.

Step 1: Integrate *(1.3.4a&b)*, *(1.3.5a&b)* numerically by using an IVP code, and obtain $Y^h(b)$, $v^h(b)$ (the numerical solution of Y and v at b).

Step 2: Form Q and β^* and solve *(1.3.3)* for s^h.

Step 3: Integrate *(1.3.1a)* with the following initial condition

$$y^h(a) = s^h + \alpha \qquad (1.3.1c)$$

numerically using an IVP code.

The numerical solution of (1.3.1a&c) is then the numerical solution of (1.3.1a&b).

We conclude this example by pointing out that despite its mathematical elegance, the method described above is by no means always practical. One major reason for this is the fact that $y^h(a)$ is only an approximation of $y(a)$. When the IVP for *(1.3.1a)* is very sensitive to the change of the initial value, use of $y^h(a)$ and $y(a)$ as initial conditions will lead to totally different solutions. In this case, the solution of *(1.3.1a&c)* cannot be used to approximate the solution of the IVP *(1.3.1a)* with initial value $y(a)$ and is therefore not the solution of *(1.3.1a&b)*. The techniques and methods behind MUTS are far more sophisticated than what was described above [14]. Nonetheless, one can get some basic idea concerning how a BVP can be solved via solving IVP from this example. See [2] for a complete coverage of shooting methods.

One may have already noticed that whether a problem is linear or not had little effect on the IVP methods we discussed above. But nonlinear BVP have to be treated quite differently and are usually considerably more expensive to solve than linear BVP. The general idea of approaching a nonlinear BVP is to adapt numerical methods for linear BVP and use Newton's iteration. To see how this can be done, we consider a simple finite difference method—the trapezoidal scheme for solving *(1.1.3a)* with general nonlinear two-point boundary conditions *(1.1.5)*

$$y' = f(x,y) \qquad a<x<b \qquad (1.1.3a)$$

$$g(y(a),y(b)) = 0 \qquad (1.1.5b)$$

Example 1.3.2: Trapezoidal Scheme

We first discuss how the *trapezoidal scheme* works on the linear problem *(1.3.1a&b)*. Given a mesh π: $a = x_1 < x_2 < \cdots < x_{N+1} = b$, the so called *trapezoidal scheme* is the following discretization of *(1.3.1a)*

$$\frac{y_{i+1} - y_i}{h_i} = \frac{1}{2}[A(x_{i+1})y_{i+1} + A(x_i)y_i] + \frac{1}{2}[q(x_{i+1}) + q(x_i)] \quad i = 1,2,\cdots,N. \qquad (1.3.6a)$$

Letting $$S_i = -h_i^{-1}I - \frac{1}{2}A(x_i),\ R_i = h_i^{-1}I - \frac{1}{2}A(x_{i+1}),\ q_i = \frac{1}{2}[q(x_{i+1}) + q(x_i)] \qquad (1.3.6b)$$

then we can write *(1.3.6a)* in matrix form

$$\begin{bmatrix} S_1 & R_1 & & & \\ & S_2 & R_2 & & \\ & & \cdot & \cdot & \\ & & & \cdot & \cdot \\ & & & S_N & R_N \\ B_a & & & & B_b \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ \cdot \\ \cdot \\ q_N \\ \beta \end{bmatrix}. \qquad (1.3.7)$$

(1.3.7) gives the numerical solution of *(1.3.1a&b)* at the mesh points.

Recall that Newton's method for solving system of equation

$$F(s) = 0 \qquad (1.3.8a)$$

involves an iterative procedure

$$s^{m+1} = G(s^m) \qquad m = 0, 1, 2, \cdots$$

where $G(s) = s - [F'(s)]^{-1}F(s)$, and $F'(s) = \frac{\partial F(s)}{\partial s}$ is the Jacobian matrix. This can also be written as

$$F'(s^m)\, w = -F(s^m) \qquad (1.3.8b)$$

$$s^{m+1} = s^m + w \qquad (1.3.8c)$$

To illustrate Newton's method for solving nonlinear BVP, consider the following example. The trapezoidal scheme for *(1.1.3a)* with boundary condition *(1.1.5b)* is given by *(1.3.9a&b)* below.

$$\frac{y_{i+1} - y_i}{h_i} = \frac{1}{2}[f(x_{i+1},y_{i+1}) + f(x_i,y_i)] \qquad i = 1,2,\cdots,N. \qquad (1.3.9a)$$

$$g(y_1,y_{N+1}) = 0 \qquad (1.3.9b)$$

Let the nonlinear algebraic equations *(1.3.8a)* be *(1.3.9a&b)* with

$$s \equiv y_\pi = (y_1,y_2,\cdots,y_{N+1})^T, \qquad s \in R^{N+1}$$

Using difference operator notation for *(1.3.9a)* we obtain

$$N_\pi y_i = \frac{y_{i+1} - y_i}{h_i} - \frac{1}{2}[f(x_{i+1},y_{i+1}) + f(x_i,y_i)] \qquad (1.3.10a)$$

and $F(s) = (N_\pi y_1, \cdots, N_\pi y_N, g(y_1, y_{N+1}))^T$. Newton's iteration *(1.3.8b)* gives

$$\frac{w_{i+1} - w_i}{h_i} - \frac{1}{2}[A(x_{i+1})w_{i+1} + A(x_i)w_i] = -N_\pi y_i^m \qquad 1 \le i \le N \qquad (1.3.10b)$$

$$B_a w_1 + B_b w_{N+1} = -g(y_1^m, y_{N+1}^m). \qquad (1.3.10c)$$

Here, y_π^m are known from values from a previous iteration (y_π^0 is an initial guess) and

$$A(x_j) = \frac{\partial f}{\partial y}(x_j, y_j^m) \qquad (1.3.11a)$$

$$B_a = \frac{\partial g(u,v)}{\partial u}, \quad B_b = \frac{\partial g(u,v)}{\partial v} \qquad at\ u = y_1^m,\ v = y_{N+1}^m. \quad (1.3.11b)$$

The next iteration is given, according to *(1.3.8c)*, by $y_i^{m+1} = y_i^m + w_i$, $i = 1, 2, \cdots, N+1$. The system *(1.3.10)* for the correction vector w_π is a linear system of equations which looks like a trapezoidal discretization of some linear problem. In each iteration, we performed two operation in succession—discretization and linearization. This method can be implemented by using the following algorithm:

Algorithm: *Trapezoidal scheme with Newton's method*

Input: *A BVP (1.1.3a) and (1.1.5b), a mesh π, an initial guess of solution values y_π^0 at mesh points, and a tolerance TOL.*

Output: *Solution values at mesh points.*

```
Repeat

1. Generate $B_a$, $B_b$ by (1.3.11b) and set $\beta = -g(y_1, y_{N+1})$.

2: For $i = 1, 2, \cdots, N$ DO

        Generate $S_i$, $R_i$ and $q_i$ of (1.3.6b) using (1.3.11a) and $q_i = -N_\pi y_i$. At the
        end of this iteration, the matrix A and the right hand side vector $\beta$ have
        been generated.

3: Solve $A w_\pi = \beta$ for $w_\pi$

4: For $i = 1, 2, \cdots, N+1$ DO

        $y_i = y_i + w_i$
```

5: Stop if $|w_J| \leq TOL$ or the iteration limit is exceeded.

The trapezoidal scheme discussed above is one of the simplest finite difference methods for solving BVP. Like the single shooting method, this method is simple but is rarely used in practice. Its biggest disadvantage is that this method has a local truncation error of order 2 (see [2] for more details). If high accuracy is desired, then a higher order scheme is more effective. Nevertheless, from this example, one can see that the nonlinear BVP are indeed more complicated to solve than the linear BVP since it involves Newton's iteration. But for most of the IVP methods, as we mentioned earlier, nonlinearity has little effect on them.

To continue our discussion about the nature of the numerical methods for IVP and BVP with respect to the formulations of BVP and IVP, we point out that since the information about the solution is given at the (at least) two boundary points for a BVP, but at none of these points the information is completely known, it is then impossible to use *local* methods such as those methods for IVP we discussed above. Consequently, the numerical solution y_i's are not determined one at a time in some linear order, rather they are determined *simultaneously* by some *global* relation such as *(1.3.7)* that connects all y_i's through out the entire mesh [2].

Let us now carry on our discussion about Runge-Kutta methods we started in the last section. A general *k-stage Runge-Kutta scheme* for $y' = f(x,y)$ is defined by

$$y_{i+1} = y_i + h_i \sum_{j=1}^{k} \beta_j f_{ij} \qquad 1 \leq i \leq N \qquad (1.3.12a)$$

where
$$f_{ij} = f(x_{ij}, y_i + h_i \sum_{l=1}^{k} \alpha_{jl} f_{il}) \qquad 1 \leq j \leq k. \qquad (1.3.12b)$$

The points x_{ij} are given by

$$x_{ij} = x_i + h_i \rho_j \qquad 1 \leq j \leq k,\ 1 \leq i \leq N \qquad (1.3.12c)$$

with
$$0 \leq \rho_1 \leq \rho_2 \leq \cdots \leq \rho_k \leq 1 \qquad (1.3.12d).$$

the "canonical points". Thus the points x_{ij}, which are sometimes called *collocation points*, are N scaled translations of the canonical set of k points $\rho_1, \rho_2, \cdots, \rho_k$ into each subinterval of the mesh π. With $k(k+2)$ free parameters, a *k-stage Runge-Kutta* scheme can archive a high order of accuracy. A Runge-Kutta scheme is explicit if $\rho_l = 0$ and $\alpha_{jl} = 0$ for all $j \leq l$, and implicit otherwise. For initial value problems, explicit methods have obvious advantages over implicit ones even though they have fewer free parameters to choose from. With an implicit method, one can still use a simple *"marching algorithm"*, such as the algorithm for Euler's method, to calculate u_i's from the left to the right *one at a time*. But for boundary value problems, implicitness are inherent in the problems in the sense that numerical solution values on π are obtained simultaneously. Thus the *"marching algorithm"* becomes impossible and the biggest advantage of explicit methods disappears. It therefore makes sense to use implicit ones by which one can make use of all the $k(k+2)$ free parameters and get the most out of the methods in terms of accuracy and efficiency [2]. Implicit Runge-Kutta methods play an important role in the literature of finite difference methods for solving BVP. The methods behind COLNEW, COLSYS and HAGRON all have close ties with it.

The above are just some basic ideas for solving BVP for ODE. As our interest is in comparing the codes rather than the methods, we will not dig further into the numerical methods for BVP for ODE. To finish this chapter, we point out once again that the methods we had discussed above are only illustrative but by no means complete or close to the methods that are behind the four codes. We hope that this chapter can get those reader who are not familiar with the BVP for ODE started getting to know the basic types of BVP for ODE and the basic ideas for solving BVP for ODE, as well as the difference between IVP and BVP from a numerical methods point of view. The four codes we are about to compare are based on the most advanced developments in the field of BVP for ODE. The methods and techniques involved are so sophisticated that it is difficult for us to come up with some simple versions that can be included here. For those who are interested in details of the theoretical background of the codes, [2] has the most complete and up-to-date coverage of the background information.

Chapter 2 The Comparison of Codes (I)
—General Discussion on the Difficulties and Criteria of Comparing Codes for BVPODE

The lack of sufficient commonly agreed upon criteria is indubitably the most serious problem of the comparison of codes. This problem not only causes technical difficulties for the comparison, but also limits the validity of conclusions resulting from any comparison. Even if one restricts himself in the comparison of codes in a narrow field, one still cannot totally get away with this problem. Both [5], [6] for HAGRON and [15] for MUTS have some discussion involving COLSYS, but the authors are very cautious in making any explicit comparison between their codes and COLSYS. People have been avoiding making direct comparison because of its problems. The common notion about comparison that it must end up with telling the good ones from the bad ones also contributed to people's reluctance of making direct comparison. Thus it seems important to make it clear what we mean by "comparison", in particular, the "comparison of BVPODE codes".

2.1 The Comparison of Codes and the Validity of Its Results

There exist many criteria that are relevant to the comparison of codes. The problem of the lack of commonly agreed criteria is mainly due to the fact that people have different opinions about the weights each of those possible criteria should receive. CPU time is definitely a possible criterion, so is the accuracy. For people with very limited computing resources, CPU time may be just as important as accuracy. On the other hand, for people who need high accuracy and have abundant computing resources, CPU time may not be what they are concerned about. A code may be too slow to be competitive today, but with faster computers that are bound to come tomorrow, it may become very competitive due to its advantages in some other aspects. Realizing that the importance for each of the possible criteria varies with individuals and time, it make sense to simply put the weights issue aside and not to judge the overall performance of a code according to several simple criteria. "Linear Ordering" does not apply in the context of the comparison of codes [17]. Realistically, what one can archive is to use as many relevant criteria as possible and compare the codes by using these criteria separately. Thus a meaningful comparison of codes may simply be a collection of information resulting in comparing the codes according to many specific criteria. It is not an attempt to tell the good ones from the bad ones, and it does not assign the weights to the criteria. Rather, it simply provides the user with plain facts about the codes and enables them to assess the codes' relative overall performances according to the criteria that they are most concerned about. However, we do not rule out the possibility of having a meaningful overall assessment as a part of a general comparison. In fact, there have previously been quite useful comparisons of codes for numerical quadrature, scalar nonlinear equation solvers, and comparisons of codes for IVPODE (see [12], [24]). In these cases, however, the codes were intended to solve the same problems and the design criteria were basically the same.

The solution of BVPODE necessitates many types of numerical consideration. In Figure 2.1 below, BVP for ODE is connected to the other areas such as approximation theory, numerical linear algebra and optimization. It is in this sense we say that solving BVP for ODE is of high complexity. This complexity of BVP for ODE makes 'the method for BVP for ODE' or 'the algorithm for a BVP for ODE code' very vague. If we interpret 'the method for BVP for ODE' as the basic numerical scheme that can solve BVP for ODE in theory (and do not include those considerations for actual implementation), and interpret an algorithm as a detailed scheme based on a numerical method that is ready for coding by using some computer language, then this complexity certainly prevents us from extending the observations resulting from comparing the codes to the comparison of the methods or comparison of algorithms. COLNEW and COLSYS are based on the same method, *i.e.* the collocation method for BVP, but COLNEW is generally faster than COLSYS in terms of CPU time needed for solving the same problem. Codes based on different implementations of a certain numerical method can be so different in many aspects of their performance that even with a considerable amount of expertise, it is still hard for one to tell whether a relative merit of a code is due to the code's underlying method. In the example mentioned earlier, COLNEW is only different from COLSYS in terms of the types of spline used for the representation of the numerical solutions and the linear system solvers. The difference between COLNEW and COLSYS in terms of speed is often simply due to their different linear system solvers. If a code is faster than COLSYS, and is slower than COLNEW, we probably cannot say anything concerning the comparison between the numerical method that lies behind that code and the collocation method in terms of speed. In fact, every part that attached to BVPODE in Figure 2.1 is very important to a BVPODE code that involves it. An improvement over any participating part can dramatically improve the performance of a code. Thus the validity of the results resulting from a comparison of codes is very much limited to the related codes themselves.

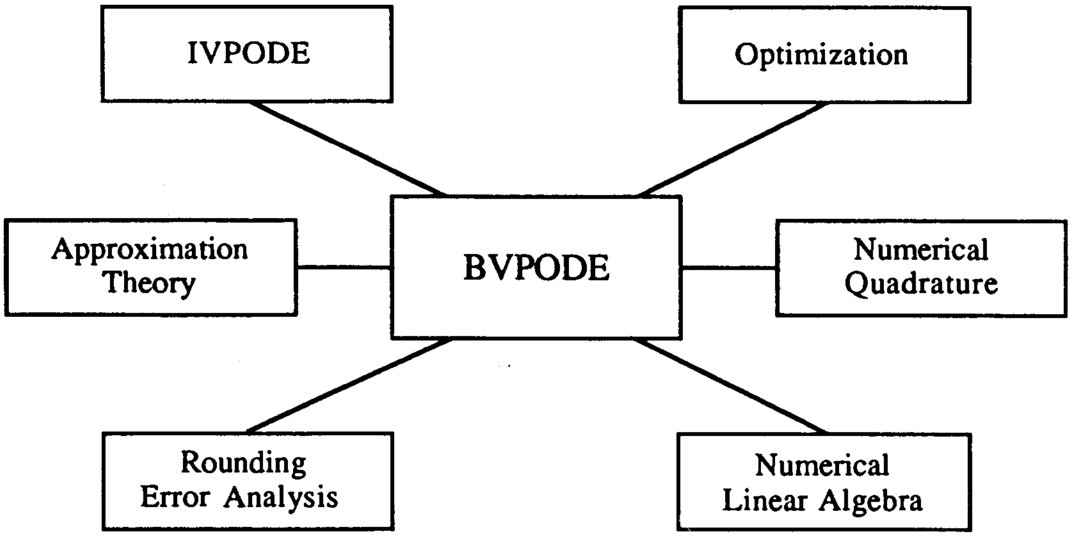


Figure 2.1 Reproduced from [17] by permission of the authors

Even if we restrict the validity of the comparison to the participating codes, we still have to answer to what degree our comparison represents the true relationships among the codes. The problem that still challenges the validity of a comparison is that many criteria can only be applied with some test problems. Accuracies of a code (or accuracy of the numerical solutions that a code can obtain), for example, usually cannot be determined without some test problems, and it also varies from test problems to test problems. The accuracy of one code may be better on one test problem than that of the other codes, and may be worse on another test problem. The validity of a general conclusion concerning accuracy is very much based upon the hope that the test problems involved are somewhat 'typical' and can represent. (if not every) almost every type of problems we may encounter in practice. While a standard set of test problems that have all the desirable properties, such as be typical or representative, might be possible for comparison of codes in some other fields, it is not a realistic idea in our context of comparing BVPODE codes (see [17], [21]).

Nonetheless, a general comparison of codes is usually subjective in nature and the real strength of a comparison often lays in the correct subjective input (or expertise) from those who make the comparison. A comparison of codes can involve only a limited number of test problems, but the observations resulting from these test problems can reflect the true relationships among the *codes* to some degree. With correct subjective input, the results of a comparison can be valid far beyond the limited number of test problems involved.

2.2 Criteria for Comparing Codes for BVPODE and Their Classification

Pereyra and Russell divided the criteria related to the comparison of BVPODE codes into the following three categories 1) general "objective" (more quantitative) criteria, 2) general "subjective" (more qualitative) criteria and 3) subjective criteria particularly relevant to BVPODE codes [17]. The first category includes timing (speed), storage, portability and program correctness. The second category includes ease of use and robustness, and the third category includes user feedback, error estimation, termination criteria, program parameters and program driver. Ironically, the objective criteria are among those criteria that are most difficult to be implemented and very often, the comparison related to these criteria need a considerable amount of subjective input in order to be complete. The speeds of the codes, for example, are generally not comparable except on some concrete examples (or test problems as mentioned above). To draw general conclusions concerning the relative speeds of the codes through some concrete examples certainly needs a considerable amount of subjective input.

In our comparison in the last chapter, apart from most of the criteria mentioned above, we will also take into consideration the number of points in the final mesh, the distribution of the final mesh points and the location of the maximum absolute error. A meaningful classification, like the one provided by Pereyra and Russell in [17], can help us understand the nature of the criteria and therefore the importance and validity of the observations related to the criteria. Indeed, there are many common aspects of the criteria that are worth noticing and can be used to make useful classification. The following are two classifications we will refer to later in our discussion. 1) According to whether or not a criterion has to depend on test problems, one may classify the criterion by 'test problem dependent' or 'non-test problem dependent'. Among those criteria mentioned above, timing, storage, accuracy, robustness, the number of mesh points in the final mesh, the distribution of the final mesh and the location of the maximum error are test problem dependent. The rest, *e.g.* ease of use, termination criteria and program parameter, can be considered as non-test problem dependent. 2) According to whether or not the criteria are about the technical details of codes' design or codes' performance, one may classify them as 'structure type criteria' and 'performance type criteria'. All the test problem dependent criteria, timing, storage, accuracy, plus ease of use can be considered as performance type criteria. The rest, such as termination criteria and error estimation are structure type criteria.

2.3 Comments On The Criteria We Choose

Most of the criteria we mentioned above will be used by us and they can be well understood without further explanation. But some of them deserve some explanation. The following is the list of criteria we will use in the comparison in the last chapter. In order to be precise, we also provide a short description of those criteria that may not be clear to everyone.

a) Codes' Drivers Related criteria

1: The form of the BVPODE that can be directly dealt with by each code.

We choose this as one of our criteria because some of the codes can only solve BVPODE that are in a very specific form. Information concerning this aspect is supposed to be useful to code users.

2: Input information

2.1 Input parameters

Each code has many parameters through which useful information is conveyed to the code. Some of them, such as a linear (nonlinear) indicator parameter, are easy to determine. But the others, such as the tolerance parameter, are more difficult to choose. We will have some general description concerning the use of the input parameters in the third chapter, and we will comment on those critical parameters in chapter four when we make our comparison.

2.2 Input subroutines

Input subroutines are the major channels to convey the information concerning a BVPODE to the codes. Each input subroutine usually describes one aspect of the problem. For example, most of the four codes involve an input subroutine that conveys the information of the ODE (or ODE system) to the codes and an input subroutine that conveys the information of the boundary value equations to the codes. It is plausible for a code to have more subroutines in order to channel more information through to the code. However, not only the number of necessary input subroutines is often restricted by the underlying method, but also there exists the problem of trade off between the amount of information one wants to channel through and the complexity (or efficiency) of the driver. This is a useful criterion since it is related to both the least amount of information a code uses to solve the problem and the complexity of the driver.

3: Ease of use

It is useful to have the difficulties we had encountered from time to time when we used the codes included here so that those who do not have much experience with BVPODE codes can learn to avoid them. The relative ease of use of the four codes will be assessed through comparing these difficulties.

b) Other performance type criteria

4: Timing (we will only use CPU time on some specific test problems)
5: Storage
6: Accuracy (measured by the maximum error. see Appendix II)
7: Portability
8: Robustness

There are a few things that are worth commenting on here. Among the above list, most of the criteria are test problem dependent. Timing, accuracy and storage are not only test problem dependent, but also 'input parameter dependent' and depend on each other in the sense that even with a fixed test problem, they may still vary with different combinations of input parameters and they are also related to each other. COLSYS and COLNEW, for example, will use different mesh selection strategies, depending upon whether or not the allotted storage is a limiting factor. If it is, the codes may solve a problem using less storage than if unlimited storage is provided. At the same time, the accuracy of the solution may be worse than it would be when unlimited storage is provided. With so many factors related to each of these criteria, it is not realistic to find the best set of parameters that would give the code the best performance in terms of all these criteria. This is an important issue and we will have technical details about the implementation of these criteria later in the last chapter and Appendix II.

c) Criteria related to the design of the codes.

9: Termination criteria

A code may be terminated for many reasons such as a satisfactory numerical solution is found (this is called normal termination), the allowed storage space is insufficient, code overflow or even the elapsed time since the code started running exceeds some limit. All these reasons except the first one are called abnormal terminations. By termination criteria here we simply mean the mechanisms that are built in the code that lead to the normal returns from the codes. We will examine these mechanisms as well as their relationship with tolerance in the fourth chapter. The other abnormal causes for terminations will be discussed when we compare the robustness of the codes.

10: User feedback

By user feedback, we mean the information about the procedures for solving a BVPODE and the correctness of the driver provided by the codes. This information might be an abnormal return from the code like a warning message indicating a certain parameter value is not properly set, or simply feedback information like the number of points in the current mesh. With more than one choice of a code, a user (especially if he/she is not familiar with the merits of various BVP codes) is likely to choose the one that is the most 'user friendly' one. When writing a code, knowing what kind of feedback information the user may need to know the most and providing the user with the *access* to this information can often result in a user friendly code. It is also important not to burden the user with feedback information that is not of great importance to them.

d) Solution related criteria

13: The number and the distribution of the final mesh points
14: The form of the solution
15: Error analysis (Location of the maximum error and the graph of the total error)

It would not be of such a great importance to look at these issues if all the codes we are interested in provide solutions on the entire domain upon normal returns from the codes. Since some of the codes return with a solution at only a finite number of final mesh points, and the gap between a continuous solution and a discrete one cannot be bridged by simple interpolation [19], these issues become very important. One can say nothing about the relative efficiency concerning two codes if one takes ten CPU seconds and returns with a solution at one hundred points and another one uses only five CPU seconds but returns with a solution at only, say, forty points for the same objective problem. Neither can one say anything concerning the relative efficiency if one code uses more time or storage and returns with a solution that characterize the true solution very well and another one uses less time and storage but returns with a solution from which only part of the true solution can be read off. An exaggerated example of this type is that a code terminates with a set of final mesh point that are all in the upper half of the domain except the lower end point. It is then very unlikely that one can tell the behaviour of the solution at the lower half of the domain by looking at the discrete solution at these mesh points.

Thus a comparison of codes with respect to only time or storage is incomplete since the ultimate goal of a code is to provide a good solution to problems. The quality of the numerical solutions from each code must also be taken into consideration. In our view, this quality should have at least the following aspects 1) accuracy of the solution, 2) the number of the final mesh points, 3) the distribution of the final mesh points (versus the shape of the true solution on test problems). We will look into these three issues in our comparison in Chapter Four.

2.4 Numerical Methods Related Issues and the Principle of Our Comparison

There are differences among the codes that can be directly traced back to the methods which the codes are built upon. The types of BVPODE that the codes can be directly applied to, the forms of solutions provided by the codes and whether or not a code has built in adapted mesh selection strategy are three most important differences of this kind.

The difficulty of comparing these differences is that they are often related to the designers' original purposes of writing the codes. If a code is written to solve two point boundary value problems only, for example, is it still a disadvantage of the code that it can not be directly used for a multipoint boundary value problem? Should not one compare this code to other codes with respect to the type of problems a code is addressed to at all?

When making a general comparison, we believe it is important to emphasize the common side of these codes' goals. The types of BVPODE these codes are aimed at or the forms of solution they provide may differ to some degree, but they all share the same original motivation, *i.e.* they are all motivated to solve BVP for ODE. This is the 'lowest common denominator' of their goals and it is on this ground that we are comparing the four codes.

In this thesis, we will treat all the four codes as general purpose codes for solving boundary value problems for ordinary differential equations regardless of the differences mentioned above. *It is our principle that what we are comparing are simply BVPODE codes, not a code aimed at a certain type of BVPODE that results in a solution in a certain form and another code aimed at another type of BVPODE that provides a solution in another form.* Every aspect of the codes we included in the last section, whether or not it is related to the designer's original purposes of writing the code, will be compared.

2.5 The Selection of Test Problems

We finish this chapter with a few words on how test problems are usually selected by many experts in this BVPODE field. We will follow their expertise on selecting the test problems. For the complete set of test problems we are going to use, please see appendix (I).

2.5.1 What do we want from the test problems

First, we need to use test problems in order to carry out the comparison concerning those test problem dependent criteria. Second, we want the test problems to be somewhat representative so that the comparison related to these test problems is somewhat trustable. These concerns are our guidance for selecting test problems.

To convert the above concerns into concrete criteria of selecting test problems, we notice that the test problem dependent criteria, especially robustness, require that the test problem includes nontrivial problems, as well as BVPODE of different types.

2.5.2 Test problems involving parameters

It is a common practice among the BVPODE experts to use test problems that involve one or more parameters which control the difficulty of the problem (see [1], [2], [4], [6], [8], [17]).

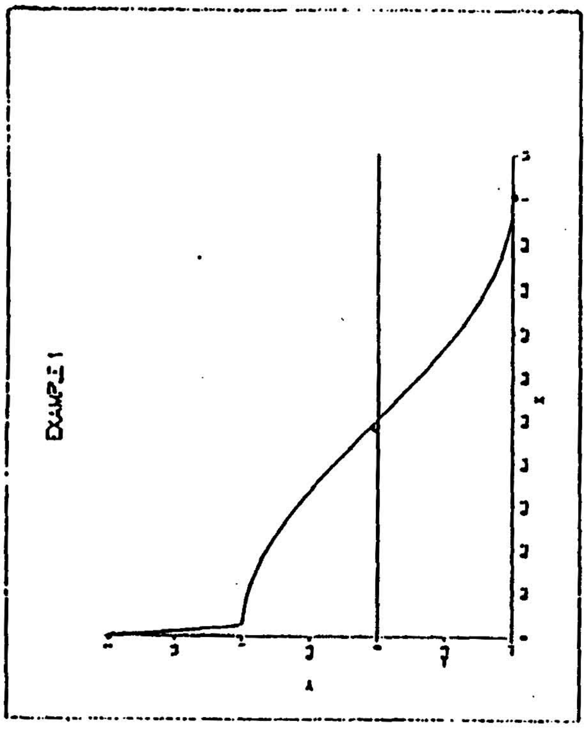
For a BVPODE code, the difficulty of a problem is usually represented by the nonsmoothness of its solution. Since all the codes are based on numerical methods that rely on the assumption that the object problems have somewhat smooth solutions to work properly, we expect that as the problem gets rougher and rougher, the performance of the codes will become worse and worse, and the codes will eventually fail to solve the problem. Thus it is ideal to select test problems with different degrees of difficulties to test the robustness of the codes. One reason for using test problems with parameters is that they provide us with the different degrees of difficulties.

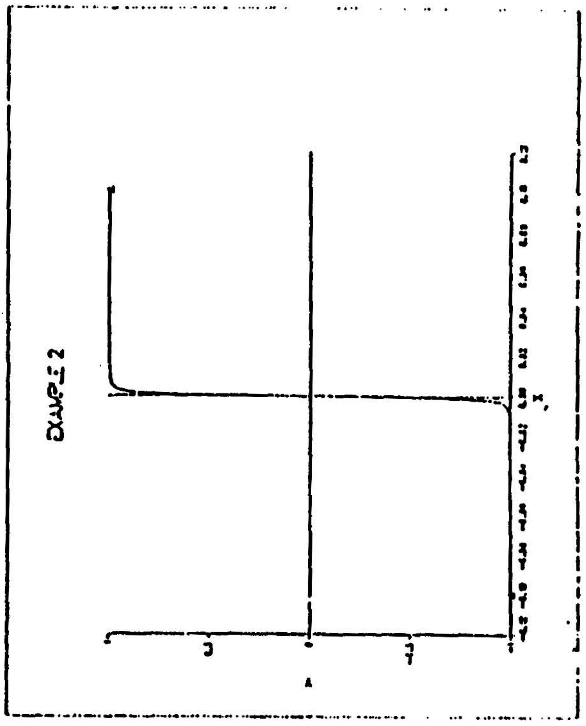
It is generally difficult to compare the degrees of difficulties of different problems, but is relatively easy to predicate the change of the degree of difficulty for a problem involving parameters when one changes the values of its parameters. This is another reason we use test problems involving parameters.

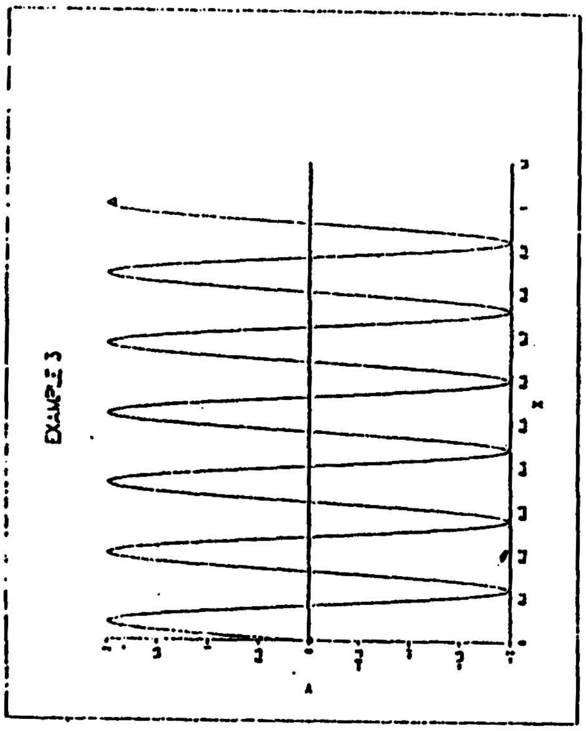
2.5.3 A few types of common nontrivial BVPs for ODE

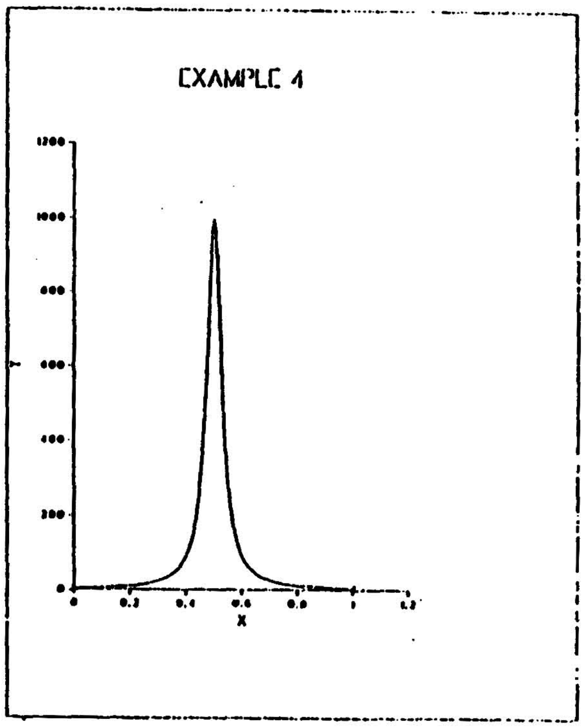
While we will only use those test problems that we believe are somewhat representative, developing strategies for finding a set of representative problems for boundary value problems for ordinary differential equation is too big a subject to be dealt with here. We will not spend too much effort on this, rather we merely mention the four basic types of nonsmoothness behaviour of the solutions for ODE for BVP.

The first type is boundary layer type (BL), such as the one shown in example one on the next page. The second type is the turning point type (TPT) such as the one shown in example two on the next page. The third type is oscillatory type (OSC) such as a high frequency sine or cosine wave, *e.g.* example three on the next page. The fourth type is spike type (SPK), such as the one shown in example four on the next page [17]. It follows naturally that a solution might have a boundary layer and have a turning point at the same time. Due to various expenses, we will only have test problems that have exact solutions that belong to each one of the four types above.


EXAMPLE 1
X
A


EXAMPLE 2
X
A


EXAMPLE 3
X
A


EXAMPLE 4
1200
1000
800
600
400
200
0
Y
0
0.2
0.4
0.6
0.8
1
1.2
X

Chapter 3 How to Use the Four Codes

The purpose of this chapter is to acquaint the reader with the procedures for running the four codes. It also serves as preliminaries for comparing the codes in terms of the criteria related to the form of the driver such as 'input parameters' and 'ease of use ' etc. As some of the codes are still undergoing changes, the procedures described in this chapter are strictly for the versions of the four codes we currently have. The procedure for running each code we provide below include the type of BVPODE each code is addressed to, information for how to set up input parameters, input (user supplied) subroutines for each code and user feedback that is available from each code. A sample driver for solving the following nonlinear two point boundary value problem for each code is also included.

Example 3.1 A two point BVPODE

$$y_1' = \frac{-2}{y_2^2} \qquad 0<t<1 \qquad (3.1a)$$

$$y_2' = y_2^2 - \frac{1}{y_1} + e^t \qquad (3.1b)$$

$$y_1(0) = 1, \; y_2(1) = e^1 \qquad (3.1c)$$

3.1: COLNEW and COLSYS

COLNEW is a modified version of COLSYS where the linear solver and the bases for representing the numerical solution are different from COLSYS. Despite these differences, the two codes have exactly the same set of input parameters, input subroutines and the forms of the drivers for running them can be exactly the same. Since the two codes also provide the same user feedback information and their difference is not what we are concerned about in this chapter, we will treat them as if they are the same and the procedure described in this section is therefore for running both of them.

3.1.1 The classes of BVPODE COLNEW and COLSYS are addressed to

COLNEW and COLSYS solve a mixed-order system of ODE subject to separated, multipoint boundary conditions given by

$$u_i^{(m_i)} = f_i(x;\, z(u(x))) \qquad i=1,\cdots,d \qquad a \le x \le b \qquad (3.2a)$$

$$g_j(z(u(\zeta_j))) = 0 \qquad j=1,\cdots,m^* \qquad (3.2b)$$

where $u(x) = (u_1(x),u_2(x),\ldots,u_d(x))^T$ is the exact solution vector, $m^* = \sum_{j=1}^{d} m_j$, the boundary points satisfy $a = \zeta_1 \le \zeta_2 \le \cdots \le \zeta_{m^*} = b$ and

$$z(u(x))=(u_1(x),u_1'(x),\cdots,u_1^{(m_1-1)}(x);u_2(x),u_2'(x),\cdots,u_2^{(m_2-1)}(x);\cdots;u_d(x),u_d(x)',\cdots,u_d^{(m_d-1)}(x))^T.$$

$m_i\ (i=1,2,\cdots,d)$, the order of the i^{th} differential equation satisfy $1\le m_i \le 4$. The functions f_i and g_i are generally nonlinear.

3.1.2 Code Parameters

Both COLSYS and COLNEW are headed by

```
SUBROUTINE COLSYS(NCOMP, M, ALEFT, ARIGHT, ZETA, IPAR,
+                  LTOL,TOL, FIXPNT, ISPACE, FSPACE, IFLAG,
+                  FSUB, DFSUB, GSUB, DGSUB, GUESS)
```

The variables in the first two lines of the heading are input parameters. The last five in the third line of the heading are the names of the input subroutines. We now explain how to set up these parameters one by one according to the order they appear in the calling sequence. Unless specified otherwise, the parameters are input parameters.

NCOMP: $= d$ — the number differential equations (≤ 20).

M(j): order of the j^{th} differential equation, $1 \le j \le NCOMP$.

ALEFT $= a$, *ARIGHT* $= b$: interval end points.

ZETA(j): $= \zeta_j,\ 1 \le j \le \sum_{j=1}^{NCOMP} m_i$, Must be mesh points in all meshes used.

See description of *IPAR(11)* and *FIXPNT* below.

IPAR:	An integer array of dimension *11*. A list if the parameters in *IPAR* and their meaning follows:
IPAR(1):	= *0* if the problem *(3.1a)* is linear in $z(\mathbf{u}(x))$. = *1* if the problem *(3.1a)* is nonlinear in $z(\mathbf{u}(x))$.
IPAR(2):	= number of collocation points per subinterval and . $m_{max} = max\{m_j, j=1,\cdots,d\} \le IPAR(2) \le 7$ (Recall that $m_{max} \le 4$).
IPAR(3):	= number of subintervals in the initial mesh *(>0)*. If on entry *IPAR(3)* is equal to *0*, then COLSYS arbitrarily sets *IPAR(3)* to be *5*.
IPAR(4):	= number of solution and derivative tolerances. $0 \le IPAR(4) \le m^*$
IPAR(5):	= dimension of *FSPACE* (see description of *FSPACE*).
IPAR(6):	= dimension of *ISPACE*. (see description of *ISPACE*).
IPAR(7):	output control = *-1* for full diagnostic printout = *0* for selected printout = *+1* for no printout
IPAR(8):	= *0* causes COLSYS to generate a uniform initial mesh. = *1* if the initial mesh π: $a = x_1 \le x_2 \le \cdots \le x_{IPAR(3)+1} = b$ is provided by the user. In this case, the initial mesh must be defined in *FSPACE* by $FSPACE(j) = x_j$. = *2* if the initial mesh is supplied by the user as with *IPAR(8) =1*, and in addition no adaptive mesh selection is to be done.
IPAR(9):	=*0* if no initial guess for the solution is provided. =*1* if an initial guess is provided by the user in subroutine *GUESS*.

	=2 if an initial mesh and approximate solution coefficients are provided by the user in *FSPACE*. (The former and new mesh are the same.)
	=3 if a former mesh and approximate solution coefficients are provided by the user in *FSPACE*, and the new mesh is to be taken twice as coarse, i.e.,every second point from the former mesh.
	=4 if in addition to a former initial mesh and approximate solution coefficients, a new mesh is provided in *FSPACE* as well.
	*** See description of output for further details on *IPAR(9) = 2, 3, 4*
IPAR(10):	= *0* if the problem is regular
	= *1* if the first relax factor if small, and the nonlinear iteration does not rely on past convergence (use for an extra sensitive nonlinear problem only).
	= 2 if we are to return immediately upon *(a)* two successive nonconvergences,or *(b)* after obtaining an error estimate for the first time.
IPAR(11):	= number of fixed points in the mesh ***other than*** *ALEFT* and *ARIGHT*.
LTOL:	an **integer** array of dimension *IPAR(4)*. *LTOL(j) = k* specifies that the j^{th} tolerance in *TOL* controls the error in the k^{th} component of *z(u(x))*.
TOL:	a **real** array of dimension *IPAR(4)*. *TOL(j)* is the error tolerance on the $LTOL(j)^{th}$ component of z(**u**). The code will attempts to satisfy on each subinterval $\|(z(v) - z(\mathbf{u}))_{LTOL(j)}\| \le TOL(j)\ (\|z(\mathbf{u})_{LTOL(j)}\| + 1)$ if *v(x)* is the approximate solution vector. ***(u(x)*** *is the exact solution of (3.1a&b))*
FIXPNT:	an array of dimension *IPAR(11)*. It contains the points, other than *ALEFT* and *ARIGHT*, which are to be included in every mesh.
ISPACE:	an **integer** work array of dimension *IPAR(6)*. Its size provides a constraint on the maximum mesh points.

FSPACE: a **real** work array of dimension *IPAR(5)*. Its size provides a constraint on the maximum mesh size.

IFLAG: the mode of return from COLSYS, output parameter.
= *1* for normal return.
= *0* if the collocation matrix is singular.
=*-1* if the expected number of subintervals exceeds storage specifications.
=*-2* if the nonlinear iteration has not converged.
=*-3* if there is an input error.

3.1.3 Input (user supplied) Subroutines

The following five subroutines must be declared external in the main program which calls either COLSYS or COLNEW.

SUBROUTINE FSUB:

This subroutine is for evaluating $f_i(x,z(u(x)))$. It should have the heading

SUBROUTINE FSUB (X, Z, F)

where $X = x$, $Z = z$ and F is the vector containing the values of f_i, as defined in *(3.2a&b)* above.

SUBROUTINE DFSUB:

This subroutine is for evaluating the *Jacobian* of *F* at a point *X*. It should have the heading

SUBROUTINE DFSUB(X, Z, DF)

where $Z = z(u(x))$ is defined as for *FSUB* and the $d{\times}m^*$ array *DF* should be filled by the partial derivatives of *F*, *i.e.* for a particular call, the subroutine returns with

$$DF(i,j) = \frac{\partial f_i}{\partial z_j} \; (i = 1, 2, \ldots, d, \; j = 1, 2, \cdots, m^*) \quad \text{at point } X.$$

SUBROUTINE GSUB:

This subroutine is for evaluating the j^{th} side condition g_i at a point $x = ZETA(j)$ $(1 \leq j \leq m^*)$. It should have the heading

SUBROUTINE GSUB(J, Z, G)

where Z is, as for *FSUB*, $z(u(x))$, G is a **scalar** containing g_j as defined in *(3.2b)*.

SUBROUTINE DGSUB:

This subroutine is for evaluating the partial derivatives of g_j's w.r.t $z(u(x))$. It should have the heading

SUBROUTINE DGSUB(J, Z, DG)

where Z is again $z(u(x))$. J is, as for *GSUB*, the index of the side condition. DG is a m^* dimensional vector that contains the partial derivatives $DG(k) = \frac{\partial g_i}{\partial z_k}, k=1,\cdots,m^*$.

SUBROUTINE GUESS:

This subroutine is for evaluating the initial approximation for $Z = z(u(x))$ and for evaluating the vector *DMVAL* which contains the $m_j{}^{th}$ derivative of the j^{th} component of the initial approximation $u(x)$. *i.e.* $DMVAL = \frac{\partial^{m_j} u_i}{\partial x^{m_j}}$, where m_j is the order of the j^{th} equation in *(3.2a)* and $j = 1, 2, \cdots, d$ *(NCOMP)*. ****Note* that this subroutine is needed only for nonlinear problems if using *IPAR(9) = 1*. It should have the heading

SUBROUTINE GUESS(J, Z, DMVAL).

3.1.4 User feedback from COLSYS and COLNEW

Users can get a various amount of feedback via the three options of IPAR(7) and IFLAG which is the mode of returning from COLSYS and COLNEW. When IFLAG(7) is set to *-1*, one gets the maximum feedback which includes the following:

1: Verification of some key input information including *a)* the number of differential equations, *b)* if the system is nonlinear, *c)* side condition points,and *d)* number of collocation points per interval.

2: The possible maximum number of subintervals (determined by the dimensions of both FSPACE and ISPACE).

3: The current mesh and approximate solution values at the mesh points.

4: Solution error estimates. [17].

5: If the problem is nonlinear, the programs also provide an account of how the nonlinear iteration is proceeding.

6: The five modes of return from *IFLAG* (see IFLAG above).

3.1.5 Solution Evaluation and Simple Continuation(with *IPAR(9)* $\geq$ *2)*

On normal return from COLSYS, the arrays *FSPACE* and *ISPACE* contain information specifying the approximate solution. In particularly, the final mesh points are contained by the first *ISPACE(1)+1* components of *FSPACE*. To produce the solution vector $z(u(x))$ at any point x $(a \leq x \leq b)$, one should use the following statement

CALL APPSLN(X, Z, FSPACE, ISPACE)

where $X = x$, $Z = z(u(x))$ and *APPSLN* is a subroutine (comes with COLSYS or COLNEW) for evaluating $z(u(x))$ that when given x, *FSPACE* and *ISPACE*, returns with $Z(x)$ (*i.e.* $z(u(x))$).

When using COLSYS or COLNEW, the resulting solution is defined by the first *(7+NCOMP)* components of *ISPACE* and the first *ISPACE(7)* components of *FSPACE*, *i.e. ISPACE(1), ... ,ISPACE(7+NCOMP)* and *FSPACE(1), ... ,FSPACE(ISPACE(7))*. Thus when evaluating the approximate solution at a specific point x, *APPSLN* only uses these components of *ISPACE* and *FSPACE*.

A formerly obtained solution can be used as the first approximation for the *nonlinear iteration* for a new problem by setting *IPAR(9) = 2, 3, or 4*. This is called *continuation*. When *IPAR(9)* is 2 or *3*, in order to do continuation, one only need to initialize *ISPACE* and *FSPACE* for the new problem by using *ISPACE(1), ... , ISPACE(7+NCOMP)* and *FSPACE(1), ... ,FSPACE(ISPACE(7))*, which define the former solution and set *IPAR(3) = ISPACE(1)* (the size of the former mesh). When *IPAR(9)* is *4*, one has to provide an initial mesh of size *IPAR(3)* and put the initial mesh into the first *IPAR(3)* components of *FSPACE*. In this case, *ISPACE* for the new problem is still initialized by using the first *(7+NCOMP)* components of *ISPACE* which define the former solution. The initial *FSPACE* for the new problem must contain the *IPAR(3)* new mesh points as its first *IPAR(3)* components followed by the first *ISPACE(7)* values in *FSPACE* that define the former solution (see *IPAR(9)* for more details).

3.1.6 Sample Driver for Example *(3.1a&b&c)* with COLSYS or COLNEW

```
C SAMPLE DRIVER PROGRAM.
C
C SOLVING EXAMPLE 3.1 BY USING COLSYS OR COLNEW.
C
C MAIN PROGRAM
C
    IMPLICIT REAL*8 (A-H,O-Z)
C
C SET UP PARAMETERS
C
      PARAMETER ( NCOMP=2       )
      PARAMETER ( IPAR1=1       )
      PARAMETER ( IPAR2=4       )
      PARAMETER ( IPAR3=0       )
      PARAMETER ( IPAR4=2       )
      PARAMETER ( IPAR5=6000    )
      PARAMETER ( IPAR6=300     )
      PARAMETER ( IPAR7=-1      )
      PARAMETER ( IPAR8=0       )
      PARAMETER ( IPAR9=1       )
      PARAMETER ( IPAR10=0      )
      PARAMETER ( IPAR11=0      )
      PARAMETER ( MSTAR=2       )
C
C SET UP ARRAYS
C
      REAL*8        ALEFT,ARIGHT,TOL(IPAR4),FSPACE(IPAR5),X
      REAL*8        EPS,FIXPNT,ZETA(MSTAR),Z(MSTAR),U(MSTAR)
      INTEGER       M(NCOMP),ISPACE(IPAR6),LTOL(IPAR4),IPAR(11),IFLAG
      INTEGER       KEY,PR,RES(2)
      EXTERNAL      FSUB,DFSUB,GSUB,DGSUB,GUESS
      COMMON        /PARAME/MPARA1,MPARA2
```

```
C
      MPARA1=MSTAR
      MPARA2=NCOMP
      IPAR(1)=IPAR1
      IPAR(2)=IPAR2
      IPAR(3)=IPAR3
      IPAR(4)=IPAR4
      IPAR(5)=IPAR5
      IPAR(6)=IPAR6
      IPAR(7)=IPAR7
      IPAR(8)=IPAR8
      IPAR(9)=IPAR9
      IPAR(10)=IPAR10
      IPAR(11)=IPAR11
C
      DO 3 I=1,11
 3    PRINT*,'I=',I,' IPAR=',IPAR(I)
C
      ALEFT=0.0D0
      ARIGHT=1.0D0
C
      ZETA(1)=ALEFT
      ZETA(2)=ARIGHT
C
      M(1)=1
      M(2)=1
C
      LTOL(1)=1
      LTOL(2)=2
      TOL(1)=1.D-4
      TOL(2)=1.D-4
C
      CALL  COLSYS(NCOMP,M,ALEFT,ARIGHT,ZETA,IPAR,LTOL,
     +                  TOL,FIXPNT,ISPACE,FSPACE,IFLAG,
     +                  FSUB,DFSUB,GSUB,DGSUB,GUESS)
C
      PRINT*,'*** FILAG = ',IFLAG
C
C CALCULATE THE MAXIMUM ERROR ON 200 EQUAL DISTANCE POINT
C
      SP=0.D0
      SQ=0.D0
      X=ALEFT
      RINCRE=(ARIGHT-ALEFT)/200.D0
      ENDPNT=ARIGHT+RINCRE/2.D0
30    CALL APPSLN (X, Z, FSPACE, ISPACE)
      CALL ACCURA (X,U)
      P=DABS(U(1)-Z(1))
      Q=DABS(U(2)-Z(2))
      SP=DMAX1(SP,P)
      SQ=DMAX1(SQ,Q)
C     PRINT 40, X,P,Q
      X=X+RINCRE
      IF(X.LT.ENDPNT) GOTO 30
C
      PRINT*,'THE MAXIMUM ERROR1 IS: ',SP
      PRINT*,'THE MAXIMUM ERROR2 IS: ',SQ
C40      FORMAT(1X,F5.2,4X,'ERROR U1: ',D14.6,'  ERROR2: ',D14.6)
      STOP
      END
C
C     *** SUBROUTINE GUESS ***
C
```

```
      SUBROUTINE  GUESS(X,Z,DMVAL)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/MSTAR,NCOMP
      REAL*8   X,Z(MSTAR),DMVAL(MSTAR)
C
      Z(2)=1.D0+X
      Z(1)=1.D0/Z(2)
      DMVAL(1)= -1.D0/(Z(2)*Z(2))
      DMVAL(2)= 1.D0
      RETURN
      END
C
C     *** SUBROUTINE FSUB ***
C
      SUBROUTINE FSUB(X,Z,F)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/MSTAR,NCOMP
      REAL*8  Z(MSTAR),F(NCOMP)
      F(1)=-2.D0/(Z(2)*Z(2))
      F(2)=Z(2)*Z(2)-1.D0/Z(1)+DEXP(X)
      RETURN
      END
C
C     *** SUBROUTINE DFSUB ***
C
      SUBROUTINE  DFSUB(X,Z,DF)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/MSTAR,NCOMP
      REAL*8  Z(MSTAR),DF(NCOMP,MSTAR),X
      DF(1,1)=0.D0
      DF(1,2)=4.D0/(Z(2)*Z(2)*Z(2))
      DF(2,1)=1.D0/(Z(1)*Z(1))
      DF(2,2)=2.D0*Z(2)
      RETURN
      END
C
C     *** SUBROUTINE GSUB ***
C
      SUBROUTINE GSUB(I,Z,G)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/MSTAR,NCOMP
      REAL*8  Z(MSTAR),G
C
      GO TO (1,2),I
1     G=Z(1)-1.D0
      RETURN
2     G=Z(2)-DEXP(1.0D0)
      RETURN
      END
C
C     *** SUBROUTINE DGSUB ***
C
      SUBROUTINE DGSUB(I,Z,DG)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/MSTAR,NCOMP
      REAL*8  Z(MSTAR),DG(MSTAR)
C
      IF(I.EQ.1) THEN
        DG(1)=1.D0
        DG(2)=0.D0
      ELSE
        DG(1)=0.D0
        DG(2)=1.D0
      ENDIF
      RETURN
      END
C
C     *** SUBROUTINE ACCURA ***
C
      SUBROUTINE ACCURA(X,U)
C
C     SUBROUTINE FOR EVALUATING THE EXACT SOLUTION
C
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/MSTAR,NCOMP
      REAL*8  X,U(MSTAR)
      U(1)=DEXP(-2.D0*X)
      U(2)=DEXP(X)
      RETURN
      END
```

3.2 HAGRON

HAGRON is designed to solve first order systems of two point boundary value problems. It is based on an implicit Runge-Kutta method (see [4], [5], [6]) and is still undergoing changes. The current version of HAGRON we have is a preliminary version which we have gratefully received from the authors. We do not yet have a complete code documentation for this version. The following is some information about the functions of the code's parameters and input subroutines we gathered when we ran the code.

3.2.1 The type of BVPODE HAGRON is addressed to

HAGRON solves a two-point boundary value problem for a system of first order ordinary differential equations given by

$$u_i' = f_i(x, \boldsymbol{u}(x)), \qquad i = 1,2, \cdots ,d \qquad a \le x \le b \tag{3.3a}$$

$$g_j(\zeta_j, \boldsymbol{u}(\zeta_j)) = 0 \qquad j = 1,2, \cdots ,d \tag{3.3b}$$

where $\boldsymbol{u}(x) = (u_1(x), u_2(x), \ldots , u_d(x))^T$ is the exact solution, $f_i(x, \boldsymbol{u}(x))$ and $g_j(\zeta_j, \boldsymbol{u}(\zeta_j))$ are generally nonlinear functions, and there is a integer k $(1 \le k \le d)$ such that

$$a = \zeta_1 = \zeta_2 = \cdots = \zeta_k < \zeta_{k+1} = \zeta_{k+2} = \cdots = \zeta_d = b \tag{3.3c}$$

3.2.2 Code Parameters

HAGRON is headed by

```
      SUBROUTINE HAGRON (ICOMP, ZETA, IPAR, LTOL, TOL, FIXPNT,
    +                         ISPACE, FSPACE, U, IFLAG,
    +                         FSUB, DFSUB, GSUB, DGSUB, SOLUTN)
```

Like COLSYS and COLNEW, the variables in the first two lines of the heading are input parameters. The last five in the third line are the names of the input subroutines. The following is a list of these parameters with explanation. Unless specified otherwise, the parameters are input parameters.

ICOMP:	$= d$ — the number of differential equations.
ZETA(J):	j^{th} side condition point (boundary point ζ_j). Must satisfy *(3.2c)*.
IPAR:	An integer array of dimension *16*. A list of the parameters in *IPAR* and their meaning follows:
IPAR(1):	= *0* if system *(3.2a&b)* is linear. = *1* if system *(3.2a&b)* is nonlinear.
IPAR(2):	= the number of side conditions at the left hand end of the region *(a)*.
IPAR(3):	= the number of subintervals in the initial mesh *(>0)*. If on entry *IPAR(3)* is equal to *0*, HAGRON arbitrarily set *IPAR(3)* to be *6*.
IPAR(4):	= number of solution tolerances.
IPAR(5):	= dimension of *FSPACE* (see description of *FSPACE*).
IPAR(6):	= dimension of *ISPACE* (see description of *ISPACE*).
IPAR(7):	output control

= *-1* for full diagnostic printout
= *0* for selected printout
= *+1* for no printout

IPAR(8): = *0* causes HAGRON to generate a uniform initial mesh.

= *1* if the initial mesh $\pi: a = x_1 \leq x_2 \leq \cdots \leq x_{IPAR(3)+1} = b$ is provided by the user. In this case, the initial mesh must be defined in *FSPACE* by *FSPACE(j)* $= x_j$.

IPAR(9): =*0* if no initial guess for the solution is provided.

=*1* if an initial guess is provided by the user in subroutine *SOLUTN*.

=*2* if an initial mesh and approximate solution are provided by the user. The mesh is in *FSPACE,* the solution is in ***u*** (see description for ***u***).

IPAR(10): = number of fixed points in the mesh ***other than a*** and *b* in *(3.2a)*. It is the dimension of *FIXPNT*.

IPAR(11): Currently not in use.

IPAR(12): = *0* unscaled merit function is used for nonlinear iteration.
=*1* scaled merit function and watchdog are used for nonlinear iteration.

IPAR(13): Currently not in use.

IPAR(14): Currently not in use.

IPAR(15): Only effective when *IPAR(12)* is set to *1*. This parameter specifies the maximum number of ***consecutive*** iterations in a Newton iteration procedure during which the unscaled merit function is allowed to increase (***consecutively***). For users who are not familiar with Newton's iteration with watchdog technique, the default value for this parameter is recommended (in the present code, this default value is *8*) .

IPAR(16): Only effective when *IPAR(12)* is set to *1*. This parameter specifies the number of iterations in the beginning of Newton's iteration procedure at which the watchdog does not bark at any "substantial increase" in unscaled merit function. For users who are not familiar with Newton's iteration with watchdog technique, the default value for this parameter is recommended. (In the current version of HAGRON, "substantial increase" means " increase by a factor of *100*", and the default value for *IPAR(16)* is *5*.)

LTOL: an **integer** array of dimension *IPAR(4)*. $LTOL(j) = k$ specifies that the j^{th} tolerance in *TOL* controls the error in the k^{th} component of $u(x)$. We also need that $1 \leq LTOL(1) \leq LTOL(2) \leq \ldots \leq LTOL(IPAR(4)) \leq ICOMP$.

TOL: a **real** array of dimension *IPAR(4)*. *TOL(j)* is the error tolerance on the $LTOL(j)^{th}$ component of $u(x)$. The code attempts to satisfy at each grid point x

$$|(v(x) - u(x))_{LTOL(j)}| \leq TOL(j)\ max(|v(x)_{LTOL(j)}|, 1)$$

where $v(x)$ is the approximate solution vector at the grid point x ($u(x)$ is the exact solution of *(3.2a&b)*).

FIXPNT: an array of dimension *IPAR(11)*. It contains the points, other than *ALEFT* and *ARIGHT*, which are to be included in every mesh.

ISPACE: an **integer** work array of dimension *IPAR(6)*. Its size provides a constraint on the maximum mesh points.

FSPACE: a **real** work array of dimension *IPAR(5)*. Its size provides a constraint on the maximum mesh size.

U: A *1* dimensional vector that holds the approximate solution for *(3.2a&b)*.

IFLAG: the mode of return from HAGRON. A output parameter.
= *1* for normal return.
= *0* if the collocation matrix is singular.
=*-1* if the expected number of subintervals exceeds storage specifications.

=*-2* if the nonlinear iteration has not converged.

=*-3* if there is an input error.

3.2.3 Input (user supplied) Subroutines

The following five subroutines must be declared external in the main program which calls HAGRON.

SUBROUTINE FSUB:

This subroutine is for evaluating $f_i(x,u(x))$. It should have the heading

SUBROUTINE FSUB (X, U, F)

where $X = x$, $U = u(x)$ and F is the vector containing the values of f_i, as defined in *(3.3a&b)* above.

SUBROUTINE DFSUB:

This subroutine is for evaluating the Jacobian of F at a point X. It should have the heading

SUBROUTINE DFSUB(X, U, DF)

where $U = u(x)$ and the $d \times d$ array DF should be filled by the partial derivatives of F, *i.e.* for a particular call, the subroutine returns with

$$DF(i,j) = \frac{\partial f_i}{\partial u_j} \ (i = 1, 2, \cdots, d,\ j = 1, 2, \cdots, d) \text{ at point } X.$$

SUBROUTINE GSUB:

This subroutine is for evaluating the j^{th} side condition g_i at a point $x = ZETA(j)$ $(1 \le j \le d)$. It should have the heading

SUBROUTINE GSUB(J, U, G)

where $U = u(x)$ and G is a scalar containing g_j as defined in *(3.3b)*.

SUBROUTINE DGSUB:

This subroutine is for evaluating the partial derivatives of g_j's w.r.t $u(x)$. It should have the heading

SUBROUTINE DGSUB(J, U, DG)

where *U* is again $u(x)$. *J* is, as for *GSUB*, the index of the side condition. *DG* is a d dimensional vector that contains the partial derivatives $DG(k) = \frac{\partial g_i}{\partial u_k}$, $k=1,\cdots,d$.

SUBROUTINE SOLUTN:

This subroutine is for evaluating the initial approximation for $u(x)$. It is only needed when *IPAR(9) = 1* and it should have the heading

SUBROUTINE SOLUTN(X, U)

where $X = x$ and $U = u(x)$.

3.2.4 User feedback from HAGRON

Like using COLSYS and COLNEW, users can get a various amount of feedback via the three options of *IPAR(7)* and *IFLAG* which is the mode of returning from HAGRON. When *IFLAG(7)* is set to *-1*, one get the maximum feedback which includes the following:

1: Verification of some key input information including *a)* the number of differential equations, *b)* if the system is nonlinear, *c)* side condition points,and *d)* components of u that require tolerances.

2: The possible maximum number of subintervals (determined by the dimensions of both *FSPACE* and *ISPACE*).

3: The number of points in the current mesh.

4: Parameters concerning deferred correction procedure.

5: If the problem is nonlinear, the programs also provide an account of how the nonlinear iteration is proceeding.

6: The five modes of return from *IFLAG* (see *IFLAG* above).

3.1.5 Output and Simple Continuation (with *IPAR(9) = 2)*

Unlike COLSYS and COLNEW, HAGRON does not have a subroutine that evaluates the solution at any point x $(a \le x \le b)$. HAGRON only provides an approximate solution at a finite number of final mesh points. On normal return from HAGRON, the array *FSPACE* contains the final mesh while the one dimensional array *U* contains the solution. More specifically, the first *d (or ICOMP)* components of *U* is just *v(FSPACE(1)),* the second *d* components that follows is *v(FSPACE(2))* and so on. Like COLSYS and COLNEW, the number of final mesh points is *ISPACE(1)+1, i.e.*

$$a = FSPACE(1) < FSPACE(2) < \cdots < FSPACE(ISPACE(1)+1) = b$$

is the final mesh.

To do *continuation* with HAGRON, one has to setting *IPAR(9)=2* and put the starting approximate solution in *U* in the way we described above and put the corresponding mesh points in *FSPACE.*

3.1.6 Sample Driver for solving *(3.1a&b&c)* with HAGRON

```
C     PROGRAM  DRIVER
C
C     THIS IS THE SAMPLE DRIVER PROGRAM FOR HAGRON
C
      IMPLICIT  REAL*8 (A-H,O-Z)
C
      PARAMETER(     NCOMP = 2       )
      PARAMETER(     IPAR1 = 1       )
      PARAMETER(     IPAR2 = 1       )
      PARAMETER(     IPAR3 = 0       )
      PARAMETER(     IPAR4 = 2       )
      PARAMETER(     IPAR5 = 25000   )
      PARAMETER(     IPAR6 = 15000   )
      PARAMETER(     IPAR7 = -1      )
      PARAMETER(     IPAR8 = 0       )
      PARAMETER(     IPAR9 = 0       )
      PARAMETER(     IPAR10 = 0      )
      PARAMETER(     IPAR11 = 0      )
      PARAMETER(     IPAR12 = 1      )
      PARAMETER(     IPAR13 = 0      )
      PARAMETER(     IPAR14 = 0        )
      PARAMETER(     IPAR15 = 8        )
      PARAMETER(     IPAR16 = 5        )
C
      REAL*8  FSPACE(IPAR5),ZETA(NCOMP),TOL(IPAR4)
      REAL*8  U(15000),FIXPNT(2),UU(NCOMP,4000)
      INTEGER  ISPACE(IPAR6),IPAR(20),LTOL(IPAR4),RES(2)
      EXTERNAL FSUB, DFSUB, GSUB, DGSUB, SOLUTN
      COMMON/PARAME/MPARA1
C
C     SET UP IPAR AND SOME CONSTANTS
C
      MPARA1  =   NCOMP
      IPAR(1)    =   IPAR1
      IPAR(2)    =   IPAR2
      IPAR(3)    =   IPAR3
      IPAR(4)    =   IPAR4
      IPAR(5)    =   IPAR5
      IPAR(6)    =   IPAR6
      IPAR(7)    =   IPAR7
      IPAR(8)    =   IPAR8
      IPAR(9)    =   IPAR9
      IPAR(10)  =   IPAR10
      IPAR(11)  =   IPAR11
      IPAR(12)  =   IPAR12
      IPAR(13)  =   IPAR13
      IPAR(14)  =   IPAR14
      IPAR(15)  =   IPAR15
      IPAR(16)  =   IPAR16
      IPRINT     =   IPAR(7)
      IMERIT     =   IPAR(12)
      IWATCH    =   IPAR(15)
      KWATCH  =   IPAR(16)
C
C     SET BOUNDARY VALUE CONDITION
C
      ALEFT = 0.0D0
      ARIGHT = 1.0D0
C
      ZETA(1) = ALEFT
      ZETA(2) = ARIGHT
C
C     SET TOLERANCES FOR U
C
      LTOL(1) = 1
      LTOL(2) = 2
C
      TOL(1) = 1.D-4
      TOL(2) = 1.D-4
C
      WRITE(6,130) NCOMP
      IF(IPAR(1).EQ.1) THEN
      IF(IMERIT .EQ. 1)  WRITE(6,110)
      IF(IMERIT .EQ. 0)  WRITE(6,120)
      WRITE(6,150) IWATCH, KWATCH
      ENDIF
C
      CALL  TIME(0,0,RES)
      CALL  HAGRON(NCOMP,ZETA,IPAR,LTOL,TOL,FIXPNT,ISPACE,
     *          FSPACE,U,IFLAG,FSUB,DFSUB,GSUB,DGSUB,SOLUTN)
      CALL  TIME(3,-1,RES)
      WRITE(6,*) 'CPU IN MILLISECONDS: ',RES(1)
      WRITE(6,*) 'ELT IN MILLISECONDS: ',RES(2)
```

```
C
C     NP1 IS THE TOTAL NUMBER OF POINTS IN THE FINAL MESH
C
      NP1=ISPACE(1)+1
C     PRINT*,'FSPACE(NP1)=',FSPACE(NP1)
      WRITE(6,170) IFLAG, NP1
      NTOL=IPAR(4)
      INCP = 1
      IF(NP1.GT.45) INCP = 5
      IF(NP1.GT.80) INCP = 10
      IF(NP1.GT.400) INCP = 50
      IF(NP1.GT.1000) INCP = 75
      CALL JSJAI(U,NP1,NCOMP,UU)
100   PRINT*,'IF YOU WANT THE OUTPUT ON U(KK), INPUT KK PLEASE'
      PRINT*,'OR ENTER ZERO FOR EXIT'
      READ(*,*) KK
      IF(KK.EQ.0) GOTO 200
      ERRMAX=0.D0
      DO 20 I=1,NP1,INCP
      XX=FSPACE(I)
      CALL  EXACT(KK,XX,SOL)
      ER=DABS(SOL-UU(KK,I))
      ERRMAX=DMAX1(ERRMAX,ER)
      WRITE(6,180) I,XX,SOL,UU(KK,I),ER
20    CONTINUE
      PRINT*,'*** THE MAXIMUE A-ERROR AT MESH PTNS IS: ',ERRMAX
      GOTO 100
C
110   FORMAT(' SCALED MERIT FUNCTION')
120   FORMAT(' UNSCALED MERIT FUNCTION')
130   FORMAT(' NUMBER OF COMPONENTS = ',I5)
150   FORMAT(' WATCHDOG ITERATION LIMIT',I5,5X,'WATCHDOG MIN',I5)
170   FORMAT(1H , 6HIFLAG=, I5,5X,16HNUMBER OF POINTS,I5)
180   FORMAT(1X,I5,3(1PG17.7),1PG20.10)
200   STOP
      END
C
C     *** SUBROUTINE SOLUTN ***
C
      SUBROUTINE  SOLUTN(X,Z)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/NCOMP
      REAL*8  Z(NCOMP)
C
      Z(2)=1.D0+X
      Z(1)=1.D0/Z(2)
      RETURN
      END
C
C   *** SUBROUTINE FSUB ***
C
      SUBROUTINE FSUB(X,Z,F)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON  /PARAME/NCOMP
      REAL*8  Z(NCOMP),F(NCOMP)
      F(1)=-2.D0/(Z(2)*Z(2))
      F(2)=Z(2)*Z(2)-1/Z(1)+DEXP(X)
      RETURN
      END
C
C     *** SUBROUTINE DFSUB ***
C
      SUBROUTINE DFSUB(X,Z,DF)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/NCOMP
      REAL*8  Z(NCOMP),DF(NCOMP,NCOMP)
C
      DF(1,1)=0.0D0
      DF(1,2)=4.D0/(Z(2)*Z(2)*Z(2))
      DF(2,1)=1.D0/(Z(1)*Z(1))
      DF(2,2)=2.0D0*Z(2)
      RETURN
      END
C
C     *** SUBROUTINE GSUB ***
C
      SUBROUTINE GSUB(I,Z,G)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/NCOMP
      REAL*8  Z(NCOMP),G
      GO TO (1,2),I
1     G=Z(1)-1.D0
      RETURN
2     G=Z(2)-DEXP(1.D0)
      RETURN
      END
C
C     *** SUBROUTINE DGSUB ***
C
      SUBROUTINE DGSUB(I,Z,DG)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAME/NCOMP
      REAL*8  Z(NCOMP),DG(NCOMP)
C
      DO 10 J=1,2
10    DG(J)=0.0D0
      GO TO (1,2),I
1     DG(1)=1.0D0
      RETURN
2     DG(2)=1.0D0
      RETURN
      END
C
C     *** SUBROUTINE EXACT ***
C
      SUBROUTINE  EXACT(KK,XX,SOL)
C
C     SUBROUTINE FOR  EVALUATING THE EXACT SOLUTION
C
      IMPLICIT REAL*8 (A-H,O-Z)
      GO TO (1,2), KK
1     SOL=DEXP(-2.D0*XX)
      RETURN
2     SOL=DEXP(XX)
      RETURN
      END
C
C     *** SUBROUTINE JSJAI ***
C
      SUBROUTINE JSJAI(U,NP1,NCOMP,UU)
C
C     SUBROUTINE FOR PUTTING THE SOLUTION INTO A NCOMP BY NP1 ARRAY
C
      IMPLICIT REAL*8 (A-H,O-Z)
      REAL*8  U(25002),UU(NCOMP,NP1)
C
```

```
        DO 10 I=1,NP1
        DO 10 J=1,NCOMP
        UU(J,I)=U(NCOMP*(I-1)+J)
10      CONTINUE
        RETURN
        END
```

3.3 MUTS

MUTS is based on a multiple shooting method for two point boundary value problems for ODE [15]. It consists of two subroutines, namely MUSL for linear two point boundary value problems and MUSN for nonlinear problems. The driver programs for MUSL and MUSN are not exact the same. The following are some details about how to use them.

3.3.1 MUSL

3.3.1a The classes of problem that MUSL is addressed to

MUSL solves a linear two-point boundary value problem

$$y'(t) = L(t)\,y(t) + r(t) \qquad a<t<b \qquad (3.4a)$$

$$M_a y(a) + M_b y(b) = \beta \qquad (3.4b)$$

where $y, \beta, r \in R^n$, and $L, M_a, M_b \in R^{n \times n}$.

3.3.1b Input Parameters

Subroutine MUSL is headed by

```
SUBROUTINE MUSL(FLIN, FDIF, N, IHOM, A, B, MA, MB, BCV,
+                  AMP,ER, NRTI, TI, NTI, Y, U, NU, Q, D, KPART,
                   PHIREC, W, LW, IW, LIW, IERROR)
```

The following is information about the parameters in the heading.

N:	The order of the system *(3.4a&b)*.
IHOM:	= *0* if the system *(3.4a&b)* is homogeneous. = *1* if the system *(3.4a&b)* is inhomogeneous.

A, B:	The two boundary points.
M_a, M_b:	The *N×N* matrices in *(3.4b)*.
BCV:	An real *N* dimensional array containing β in *(3.4b)*.
AMP:	On entry *AMP* must contain the allowed incremental factor of the homogeneous solutions between two successive output points. If the increment of a homogeneous solution between two successive output points becomes greater than *2×AMP*, a new output point is inserted. When the input value of *AMP* is less than or equal to *1*, a default value is assumed. The default value of *AMP* varies with the value of *NRTI*. If *NRTI = 0,* then the default value of *AMP* is

max{ ER(1), ER(2) } / ER(3)

If *NRTI ≥1,* then the default value is infinity.

ER: An real array of dimension *5*.

On entry *ER(1)* must contain a relative tolerance for solving the differential equation. If the relative tolerance is smaller than 10^{-12} the subroutine will change *ER(1)* into 10^{-12} + *ER(3)*.

On entry *ER(2)* must contain an absolute tolerance for solving the differential equation, *ER(3)* must contain the machine precision.

On exit *ER(2)* and *ER(3)* are unchanged.

See *3.3.1e* for *ER(4)* and *ER(5)*.

NRTI: On entry *NRTI* is used to specify the required output points. There are three ways to specify the required output points:

1) *NRTI = 0*, the subroutine automatically determines the output points using the allowed incremental factor *AMP*.

2) *NRTI = 1*, the output points are supplied by the user in the array *TI*.

3) *NRTI* > 1, the subroutine computes the *(NRTI+1)* output points *TI(K)* by

$$TI(k) = A + \frac{(k-1)\times(B-A)}{NRTI};$$

so $TI(1) = A$ and $TI(NRTI+1) = B$.

Depending on the allowed incremental factor *AMP*, more output points may be inserted in cases *2* and *3*.

Also see *3.3.1e*.

TI: A real array of dimension *NTI*. On entry: if $NRTI = 1$, *TI* must contain the required output points in monotone order:

$$A = TI(1) < \cdots < TI(l) = B$$

l denotes the total number of required output points.

Also see *3.3.1e*.

NTI: *NTI* is the dimension of *TI* and one of the dimensions of the arrays *X, U,Q, D, PHIREC*. *NTI* must satisfy

NTI $\geq$ *the total number of output points* $+ 3$.

i.e. if the routine was called with $NRTI > 1$ and $AMP \leq 1$ the total number of output points is the entry value of $NRTI + 1$, so *NTI* should be at least the entry value of $NRTI + 4$. Unchanged on exit.

Y: A real array of dimension *(N,NTI)*. Also see *3.3.1e*.

U: A real array of dimension *(NU,NTI)*. Also see *3.3.1e*.

NU: *NU* is one of the dimensions of *U* and *PHIREC*. *NU* must satisfy

$$NU \geq \frac{N\times(N+1)}{2}.$$

Unchanged on exit.

Q:	A real array of dimension *(N,N,NTI)*. See *3.3.1e* for more details.
D:	A real array of dimension *(N,NTI)*. If *IHOM* = *0*, the array *D* has no real use and the user is recommended to use the same array for the *Y* and *D*. If *IHOM* = *1*, on exit *D(i,k)* $i=1,2,\cdots,N$ contains the inhomogeneous term *d(k)*, $k=1,2,\cdots,NRTI$, of the multiple shooting recursion. Also see *3.3.1e*.
KPART:	Integer. Also see *3.3.1e*.
PHIREC:	A real array of dimension *(NU,NTI)*. Also see *3.3.1e*.
W:	A real array of dimension *(LW)*. Used as work space.
LW:	*LW* is the dimension of *W* and $LW \geq 8 \times N + 2 \times N \times N$. Unchanged on exit.
IW:	An integer array of dimension *(LIW)*. Used as work space.
LIW:	*LIW* is the dimension of *IW*. $LIW \geq 3 \times N$. Unchanged on exit.
IERROR:	Integer. Error indicator (see user feedback below for details).

3.3.1c Input Subroutines

SUBROUTINE FLIN:

This subroutine evaluates the homogeneous part of the differential equation $L(t)y(t)$ in *(3.4a)*. It must have the heading

SUBROUTINE FLIN(T, Y, F)

where $t = T$, $y(t) = Y$ and *F* is the *N* dimensional vector containing $L(t)y(t)$. *FLIN* must be declared as *EXTERNAL* in the program from which MUSL is called.

SUBROUTINE FDIF:

This subroutine evaluates the right-hand-side of the inhomogenous differential equation $L(t)y(t) + r(t)$ in *(3.4a)*. It must have the heading

SUBROUTINE FDIF(T, Y, F)

where $t = T$, $y(t) = Y$ and F is the N dimensional vector containing $L(t)y(t) + r(t)$, and it must be declared *EXTERNAL* in the program from which MUSL is called.

In case the system *(3.4a)* is homogeneous, *FDIF* is the same as *FLIN*.

3.3.1d User feedback from MUSL

MUSL provide a wide range of user feedback through an error indicator *IERROR*. This indicator indicates *15* different kinds of mode of return from MUSL. These modes are either specific terminal errors or specific warning messages. The following are these *15* modes

***IERROR**:* Integer. Error indicator.

= *0*. No errors detected .

= *100*. Input error. This is caused by at least one of the following:
$N < 1, IHOM < 0 , NRTI < 0, NTI < 5, NU < \frac{N\times(N+1)}{2} ,or\ A=B$.
Terminal error.

= *101*. Input error: either *ER(1)* or *ER(2)* or *ER(3)* is negative.
Terminal error.

= *103*. Input error: either $LW < 8\times N + 2\times N\times N$ *or* $LIW < 3\times N$.
Terminal error.

= *120*. Input error: the routine was called with $NRTI = 1$, but the given output points in the array *TI* are not in monotone order.
Terminal error.

= *121*. Input error: the routine was called with $NRTI = 1$, but the first given output point or the last output point is not equal to A or B. Terminal error.

= *122*. Input error: the value of *NTI* is too small; the number of output points is greater than *NTI - 3*. Terminal error.

= *200*. This indicates that there is a minor shooting interval on which the incremental growth is greater than the *AMP*. This is to be attributed to the used method for computing the fundamental solution, and may jeopardize the global accuracy if

$$ER(3)\times AMP > \max\{ER(1), ER(2)\}.$$

Warning error.

= *213*. This indicates that the relative tolerance was too small. The subroutine has changed it into a suitable value. Warning error.

= *215*. This indicates that during integration the particular solution or a homogeneous solution has vanished, making a pure relative error test impossible. Must use non-zero absolute tolerance to continue. Terminal error.

= *216*. This indicates that during integration the requested accuracy could not be achieved. User must increase error tolerance. Terminal error.

= *218*. This indicates that the input parameter $N \leq 0$, or that either the relative tolerance or the absolute tolerance is negative.Terminal error.

= *240*. This indicates that the global error is probably larger than the error tolerance due to instabilities in the system. Most likely the problem is ill-conditioned. Output value is the estimated error amplification factor *ER(5)*. Warning error.

= *250*. This indicates that one of the *U(k)* is singular. Terminal error.

= *260*. This indicates that the problem is probably too ill-conditioned with respect to the boundary condition. Terminal error.

3.3.1e Output from MUSL

On normal return from MUSL, there are two types of output available. They are the approximate solution related outputs and the others.

Approximate Solution related Outputs:

NRTI:	On exit, *NRTI* contains the total number of output points.
TI:	On exit, *TI(i), i = 1,2, ···,NRTI* contains the output points.
Y:	On exit *Y(i,k) , i=1,2,···,N* contains the solution of the BVP at the output points *TI(k), k=1,2, ···,NRTI.*

Other Outputs:

The following output may not be of great importance to those who are not interested in the details of solving BVPODE with multiple shooting method. Please see [14] and [2] for more details.

ER(4):	On exit *ER(4)* contains an estimate of the condition number of the boundary value problem.
ER(5):	On exit *ER(5)* contains an estimated error amplification factor.
U:	On exit *U(i,k) i=1,2,···,NU* contains the relevant elements of the upper triangular matrix *U(k), k=2,···,NRTI.* The elements are stored column-wise, the j^{th} column of *U(k)* is stored in $U(n_j+1, k), U(n_j+2, k), \cdots, U(n_j+j, k)$, where $n_j = \frac{(j-1)\times j}{2}$. (See [14] for *U(k)*.)

Q: On exit $Q(i,j,k)$ $i=1,2,\cdots,N$, $j=1,2,\cdots,N$ contains the N columns of the orthogonal matrix $Q(k)$, $k=1,\cdots,NRTI$. (See [14] for Q.)

D: If $IHOM = 0$ the array D has no real use and the user is recommended to use the same array for the Y and D. If $IHOM = 1$, on exit $D(i,k)$ $i=1,2,\cdots,N$ contains the inhomogeneous term $d(k)$, $k=1,2,\cdots,NRTI$, of the multiple shooting recursion. (See [14] for $d(k)$.)

KPART: On exit *KPART* contains the ***global k-partition*** of the upper triangular matrices $U(k)$.

PHIREC: On exit *PHIREC* contains a fundamental solution of the multiple shooting recursion. The fundamental solution is upper triangular and is stored in the same way as the $U(k)$.

3.3.1f Sample driver for solving *(3.5a&b)* using MUSL

$$y'' = \frac{2}{(t+\varepsilon)^3} + \frac{2}{(t-\varepsilon-1)^3} \qquad 0<t<1 \qquad (3.5a)$$

$$y(0) = \frac{1}{\varepsilon} - \frac{1}{(\varepsilon+1)}\,, \quad y(1) = \frac{1}{(\varepsilon+1)} - \frac{1}{\varepsilon} \qquad (3.5b)$$

The following is a sample driver program for solving *(3.5a&b)* using MUSL.

```
C
C SAMPLE DRIVER PROGRAM
C
C SOLVING EXAMPLE (3.5a&b) BY USING MUSL
C
C PROGRAM MAIN
C
      IMPLICIT REAL*8 (A-H,O-Z)
C
C SET UP OBJECTIVE PROBLEM RELATED PARAMETERS
C
      PARAMETER (A=0.0D0, B=1.0D0, IHOM=1, N=2 )
C
C SET UP PROGRAM ARRAYS' DIMENSION RELATED PARAMETERS
C
      PARAMETER (NTI=400, NU=10 , LW=40 , LIW=20 )
C
C SET UP ARRIES AND CONSTANTS
C
```

```
      REAL*8        MA(N,N),MB(N,N),BCV(N),SOL(N)
      REAL*8        ER(5),TI(NTI),Y(N,NTI),Q(N,N,NTI),U(NU,NTI)
      REAL*8        D(N,NTI),PHIREC(NU,NTI),W(LW)
      INTEGER       NRTI,KP,IW(LIW),IERRO,KPART
      INTEGER       RES(2)
      COMMON        /PARAM1/EPS,NZ
      EXTERNAL      FLIN,FDIF
C
      PRINT*,'MUS EXAMPLE (3.5a&b)    OUTPUT POINTS 300'
      PRINT*,'INPUT EPS'
      READ(6,*) EPS
C
C     MORE  PROGRAM  PARAMETERS
C
      NZ=N
      AMP=100
      NRTI=299
C
C     ER1: R-TOL,ER2: A-TOL,ER3: M-EPS
C
      ER(1)=1.D-10
      ER(2)=1.D-6
      ER(3)=0.2D-15
C
C     SET UP B.C. MATRIX AND VECTOR
C
      DO 5 I=1,N
      BCV(I)=0.D0
      DO 5 J=1,N
      MA(I,J)=0.D0
      MB(I,J)=0.D0
5     CONTINUE
      MA(1,1)=1.D0
      MB(2,1)=1.D0
      XX=1.0D0/EPS
      BCV(1)=XX-1.D0/(EPS+1.D0)
      BCV(2)=-XX+1.D0/(1.D0+EPS)
C
      CALL  TIME(0,0,RES)
      CALL  MUSL(FLIN,FDIF,N,IHOM,A,B,MA,MB,BCV,AMP,ER,NRTI,TI,
     +         NTI,Y,U,NU,Q,D,KPART,PHIREC,W,LW,IW,LIW,IERROR)
      CALL  TIME(3,-1,RES)
      WRITE(*,*) '**CPU IN MILLISECONDS: ',RES(1)
      WRITE(*,*) '**ELT IN MILLISECONDS: ',RES(2)
C
C     CALCULATE  THE  ERROR
C
      SP=0.D0
      SS=0.D0
      DO 50 J=1,NRTI
      XX=TI(J)
      CALL  EXACT(XX,SOL)
      P=DABS(SOL(1)-Y(1,J))
      S=DABS(SOL(2)-Y(2,J))
      SP=DMAX1(P,SP)
      SS=DMAX1(S,SS)
      WRITE(6,60)  XX,P,S
50    CONTINUE
60    FORMAT(1X,F8.6,'   ER1: ',D16.8,'   ER2: ',D16.8)
      PRINT*,'NRTI=',NRTI
      PRINT*,'** MAXIMUM E1: ',SP
      PRINT*,'** MAXIMUM E2: ',SS
      STOP
      END
C
C     *** SUBROUTINE FLIN ***
C
      SUBROUTINE FLIN(T,Y,F)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAM1/EPS,NZ
      REAL*8  Y(NZ),F(NZ)
C
C     HOMOGENOUS PART OF THE R-H SIDE
C
      F(1)=Y(2)
      F(2)=0.D0
      RETURN
      END
C
C     *** SUBROUTINE FDIF ***
C
      SUBROUTINE FDIF(T,Y,F)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAM1/EPS,NZ
      REAL*8  Y(NZ),F(NZ)
C
C     R-H FUNCTION EVALUTION
C
      PP=(T+EPS)
      XX=PP*PP*PP
      PP=T-EPS-1.D0
      YY=PP*PP*PP
      F(1)=Y(2)
      F(2)=2.D0/XX+2.D0/YY
      RETURN
      END
C
C     *** SUBROUTINE EXACT ***
C
      SUBROUTINE  EXACT(T,SOL)
      IMPLICIT REAL*8 (A-H,O-Z)
      COMMON/PARAM1/EPS,NZ
      REAL*8  SOL(NZ)
C
      PP=1.D0/(EPS+T)
      QQ=1.D0/(T-EPS-1.D0)
      SOL(1)=PP+QQ
      SOL(2)=-PP*PP-QQ*QQ
      RETURN
      END
```

3.3.2 MUSN

3.3.2a The classes of BVPODE that MUSL is addressed to

MUSN solves a nonlinear two-point boundary value problem

$$y' = f(t, y) \qquad a<t<b \qquad (3.6a)$$

$$g(y(a), y(b)) = 0 \qquad (3.6b)$$

where $y, f, g, 0 \in R^n$ and y, f, g are n dimensional vector functions.

3.3.2b Input Parameters

Subroutine MUSN is headed by

```
SUBROUTINE MUSN(FDIF, YOT, G, N, A, B, ER, TI, NTI, NRTI, AMP,
+                    ITLIM, Y, Q, U, NU, D, PHI, KP, W, LW, IW, LIW,
                     WG, LWG, IERROR)
```

The following is information about the parameters in the heading.

N:	$= n$. The order of the system *(3.6a&b)*.
A, B:	The two boundary points. *i.e.* $a = A$, $b = B$.
ER:	A real array of dimension *5*.

On entry *ER(1)* must contain the required tolerance for solving the differential equation.

On entry *ER(2)* must contain the initial tolerance with which a first approximate solution will be computed. This approximate solution is then used as an initial approximation for the computation of a solution with a tolerance *ER(2)×ER(2)* and so on until the required tolerance is reached. The initial tolerance that is actually used by the code is not necessarily the input value of *ER(2)*. To avoid an inappropriate input value of *ER(2)*, the code always uses $max\{ ER(1), min(ER(2), 10^{-2})\}$ as the initial tolerance.

On entry *ER(3)* must contain machine precision.

On exit *ER(1)*, *ER(2)* and *ER(3)* are unchanged.

See *3.3.2e* for *ER(4)* and *ER(5)*.

NRTI: On entry *NRTI* is used to specify the required output points. There are three ways to specify the required output points:

1) *NRTI* = *0*, the subroutine automatically determines the output points using the allowed incremental factor *AMP* (see *AMP* below).

2) *NRTI* = *1*, the output points are supplied by user in the array *TI*.

3) *NRTI* > *1*, the subroutine computes the *(NRTI+1)* output points *TI(k)* by

$$TI(k) = A + \frac{(k-1)\times(B-A)}{NRTI};$$

so *TI(1)* = *A* and *TI(NRTI+1)* = *B* .

Depending on the allowed incremental factor *AMP*, more output points may be inserted in cases *2* and *3*.

Also see *3.3.2e*.

TI: A real array of dimension *NTI*. On entry: if *NRTI* = *1* , *TI* must contain the required output points in monotone order:

$$A = TI(1) < \cdots < TI(l) = B$$

l (determined by *NRTI*) is the total number of required output points.

Also see *3.3.2e*.

NTI: Integer. *NTI* is one of the dimensions of *TI, Y, S, Q, U* and *PHI*. It must satisfy

NTI ≥ *the total number of output points + 1.*

i.e. if the routine was called with $NRTI > 1$, NTI may be equal to the entry value of $NRTI + 1$. Unchanged on exit.

AMP: On entry *AMP* must contain the allowed increment between two successive output points. *AMP* is used to determine output points and to assure that the increment between two output points is at most $AMP \times AMP$. A small value for *AMP* may result in a large number of output points.

Unless $1 < AMP < 0.25 \times (ER(1)/ER(3))^{0.5}$, the default value

$$AMP = 0.25 \times (ER(1)/ER(3))^{0.5} \text{ is used.}$$

Unchanged on exit.

ITLIM: Integer. Maximum number of iterations allowed.

Y: A real array of dimension *(N,NTI)*. Also see *3.3.2e*.

U: A real array of dimension *(NU,NTI)*. Also see *3.3.2e*.

NU: *NU* is one of the dimensions of *U* and *PHI*. *NU* must satisfy

$$NU \geq \frac{N \times (N+1)}{2}$$

Unchanged on exit.

Q: A real array of dimension *(N,N,NTI)*. See *3.3.2e* for more details.

D: A real array of dimension *(N,NTI)*. On exit $D(.,i)$ $i=1,2,\cdots,NRTI$ contain the inhomogeneous term of the incremental recursion.

KP: Integer. Also see *3.3.1e*.

PHI: A real array of dimension *(NU,NTI)*. Also see *3.3.1e*.

W:	A real array of dimension *(LW)*. Used as work space.
LW:	*LW* is the dimension of *W* and $LW \geq 7 \times N + 3 \times N \times NTI + 4 \times N \times N$. Unchanged on exit.
IW:	An integer array of dimension *(LIW)*. Used as work space.
LIW:	*LIW* is the dimension of *IW*. $LIW \geq 3 \times N + NTI$. Unchanged on exit.
WG:	A real array of dimension *LWG*. *WG* is used to restore the integration grid points.
LWG:	Integer. *LWG* is the dimension of *WG*. *LWG* must satisfy $LWG \geq \frac{1}{5} \times (\textit{total number of grid points})$. The minimum number of grid points between two successive output points is *5*, so the minimum value for *LWG* is the number of actually used output points. Initially a crude estimate for *LWG* has to be made. Also see *IERROR 219* in *3.3.2e*.
IERROR:	Integer. Error indicator (see user feedback below for details).

3.3.2c Input Subroutines

SUBROUTINE FDIF:

This subroutine evaluates the right-hand-side $f(t,y)$ in *(3.6a)*. It must have the heading

SUBROUTINE FDIF(T, Y, F)

where $t = T$, $y(t) = Y$ and *F* is the *N* dimensional vector containing $f(t,y)$. *FDIF* must be declared *EXTERNAL* in the program from which MUSN is called.

SUBROUTINE YOT:

This subroutine evaluates the initial approximate solution $y0(t)$ supplied by the user for any $t = T$. It must have the heading

SUBROUTINE YOT(T, Y)

where $t = T$ and Y is an N dimensional vector that $y0(t) = Y$. *YOT* must be declared as *EXTERNAL* in the program from which MUSN is called.

SUBROUTINE G:

This subroutine evaluates $g(y(a),y(b))$ in *(3.6b)* as well as the *Jacobians*

$$\frac{\partial g(u,v)}{\partial u} \text{ at } u = y(a)\,, \quad \frac{\partial g(u,v)}{\partial v} \text{ at } v = y(b)$$

It must have the heading

SUBROUTINE G(N, YA, YB, FG, DGA, DGB)

where $YA, YB, FG \in R^N$, $DGA, DGB \in R^{N \times N}$. $y(a) = YA$, $y(b) = YB$, $FG = g(y(a),y(b))$ and *DGA, DGB* contain the first and second *Jacobians* shown above, respectively. *G* must be declared as *EXTERNAL* in the program from which MUSN is called.

3.3.2d User feedback from MUSN

Like MUSL, MUSN also provides a wide range of user feedback through error indicator *IERROR*. This indicator indicates *15* different kinds of mode of return from MUSN. These modes are either specific terminal errors or specific warning messages. The following are these *15* modes

IERROR: Integer. Error indicator.

$= 0$. No errors detected.

= *01*. Input error: either *ER(1)* or *ER(2)* or *ER(3)* is negative.
Terminal error.

= *05*. Input error: either $N<1$ or $NTI<3$ *or* $NRTI<0$ or $NU < \frac{N\times(N+1)}{2}$
or $A = B$. Terminal error.

= *06*. Input error: either $LW < 7\times N + 3\times N\times NTI + 4\times N\times N$ *or*
$LIW < 3\times N + NTI$. Terminal error.

= *20*. Input error: the routine was called with $NRTI = 1$, but the given output points in the array *TI* are not in monotone order.
Terminal error.

= *21*. Input error: the routine was called with $NRTI = 1$, but the first given output point or the last output point is not equal to A or B.
Terminal error.

= *22*. Input error: the value of *NTI* is too small, the number of output points is greater than $NTI - 1$. Terminal error.

= *23*. Input error: the value of *LWG* is less than the number of output points. Increase the dimension of the array *WG* and the value of *LWG*.
Terminal error.

= *216*. This indicates that during integration the requested accuracy could not be achieved. User must increase error tolerance.
Terminal error.

= *219*. This indicates that the routine needs more space to store the integration grid points. An estimation for the required workspace (*i.e.*

the value of *LWG)* is given. Terminal error.

= *230*. This indicates that the Newton iteration fails to converge. Terminal error.

= *231*. This indicates that the number of iterations has become greater than *ITLIM*. Terminal error.

= *240*. This indicates that the global error is probably larger than the error tolerance due to instabilities in the system. Most likely the problem is ill-conditioned. Output value is the estimated error amplification factor *ER(5)*. Warning error.

= *250*. This indicates that one of the upper triangular matrices *U* is singular. Terminal error.

= *260*. This indicates that the problem is probably too ill-conditioned with respect to the boundary conditions. Terminal error.

3.3.2e Output from MUSN

On normal return from MUSN, like MUSL, there are two types of outputs available. They are the approximate solution related outputs and the others.

Approximate Solution related Outputs:

NRTI:	On exit, *NRTI* contains the total number of output points.
TI:	On exit, *TI(i), i = 1,2, ⋯,NRTI* contains the output points.
Y:	On exit *Y(i,k)* , *i=1,2,⋯,N* contains the solution of the BVP at the output point *TI(k), k=1,2, ⋯,NRTI*.

Other Outputs:

The following output may not be of importance to those who are not interested in the details of solving BVPODE with the multiple shooting method. Please see [14] and [2] for more details.

ER(4): On exit *ER(4)* contains an estimation of the condition number of the boundary value problem (see [2], [14]).

ER(5): On exit *ER(5)* contains an estimated error amplification factor.

U: On exit *U(.,i) i=1,2,⋯,NRTI* contains the the upper triangular factors of the incremental recursion. The elements are stored column wise. The j^{th} column of *U* is stored in $U(n_j+1, k), U(n_j+2, k), \cdots, U(n_j+j, k)$, where $n_j = \frac{(j-1)\times j}{2}$. (See [14] for *U*.)

Q: On exit *Q(.,.,i), i=1,2, ⋯,NRTI* contains the orthogonal factors of the incremental recursion.

D: On exit *D(.,i) i=1,2,...,NRTI* contain the inhomogeneous term of the incremental recursion.

KP: On exit *KP* contains the dimension of the increasing solution space.

PHI: On exit *PHI(.,i), i = 1,2,...,NRTI* contains the fundamental solution of the incremental recursion. The fundamental solution is upper triangular and stored in the same way as the upper triangular *U*.

3.3.2f Sample driver program for solving *(3.1a&b)* using MUSN

```
C
C SAMPLE DRIVER PROGRAM
C
C SOLVING EXAMPLE (3.1a&b) USING MUSN
C
C MAIN PROGRAM
C
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
C
C     SET UP PARAMETERS
C
      PARAMETER (A=0.D0,B=1.D0,N=2,ITLIM=50 )
```

```
        PARAMETER  (NRTI=299,NTI=400,NU=20,LW=2600,LIW=600,LWG=400 )
C
        REAL*8          ER(5),TI(NTI),Y(N,NTI),Q(N,N,NTI),U(NU,NTI)
        REAL*8          D(N,NTI),PHIREC(NU,NTI),W(LW),WG(LWG),SOL(N)
        INTEGER         KPART,IW(LIW),IERROR
        INTEGER         RES(2)
        EXTERNAL        FDIF,YOT,G
        COMMON          /PARAME/NZ
C
        ER(1)=1.D-6
        ER(2)=1.D-2
        ER(3)=0.20D-15
        AMP = 100
        NZ = N
C
        CALL  TIME(0,0,RES)
        CALL  MUSN(FDIF,YOT,G,N,A,B,ER,TI,NTI,NRTI,AMP,ITLIM,Y,Q,U,NU,
         +        D,PHIREC,KPART,W,LW,IW,LIW,WG,LWG,IERROR)
        CALL  TIME(3,-1,RES)
        PRINT*,'CPU IN MILISECONDS: ',RES(1)
        PRINT*,'ELT IN MILISECONDS: ',RES(2)
C
        SP=0.D0
        SR=0.D0
        DO 20 I=1,NRTI
        X=TI(I)
        CALL  EXACT(X,N,SOL)
        P=DABS(SOL(1)-Y(1,I))
        R=DABS(SOL(2)-Y(2,I))
        SP=DMAX1(SP,P)
        SR=DMAX1(SR,R)
        WRITE(6,40)  X,P,R
40      FORMAT(1X,F8.4,4X,'ERROR1:',D12.6,'     ERROR2:',D12.6)
20      CONTINUE
        PRINT*,'THE  MAXIMUM  ERROR1  IS:  ',SP
        PRINT*,'THE  MAXIMUM  ERROR2  IS:  ',SR
        STOP
        END
C
C       *** SUBROUTINE FDIF ***
C
        SUBROUTINE FDIF(T,Y,F)
        IMPLICIT REAL*8 (A-H,O-Z)
        COMMON/PARAME/N
        REAL*8  Y(N),F(N)
C
        F(1) =-2.D0/(Y(2)*Y(2))
        F(2) =Y(2)*Y(2)-1.D0/Y(1)+DEXP(T)
        RETURN
        END
C
C       *** SUBROUTINE YOT ***
C
        SUBROUTINE  YOT(T,Y)
        IMPLICIT REAL*8 (A-H,O-Z)
        COMMON/PARAME/N
        REAL*8  Y(N)
C
        Y(1) = 1.D0/(1.D0+X)
        Y(2) = 1.D0
        RETURN
        END
C
```

```
C         *** SUBROUTINE G ***
C
          SUBROUTINE G(N,XA,XB,FG,DGA,DGB)
          IMPLICIT REAL*8 (A-H,O-Z)
          REAL*8  XA(N),XB(N),FG(N),DGA(N,N),DGB(N,N)
C
          FG(1)=XA(1)-1.D0
          FG(2)=XB(2)-DEXP(1.D0)
C
          DO 20 I=1,N
          DO 20 J=1,N
          DGA(I,J)=0.D0
          DGB(I,J)=0.D0
20        CONTINUE
C
          DGA(1,1)=1.D0
          DGB(2,2)=1.D0

          RETURN
          END
C
C         *** SUBROUTINE EXACT ***
C
          SUBROUTINE  EXACT(X,N,SOL)
          IMPLICIT REAL*8 (A-H,O-Z)
          REAL*8  SOL(N)
C
          SOL(1)=DEXP(-2.D0*X)
          SOL(2)=DEXP(X)
C
          RETURN
          END
```

Chapter 4: The Comparison of the Codes (II)
—The Basic Design of Our Comparison and the Results of Comparing the Four Codes

This chapter is mainly concerned with comparing the four codes with respect to those issues discussed in Chapter Two. The date that we received each one of the four codes is shown in the end the Introduction. The comparison conducted here is based on the versions of the four codes we received and the resulting conclusions may not be applied to different versions of these codes. We will have some general discussion about the design of the comparison and how our comparison is conducted in the first section. In the second section we will have detailed comparison with respect to the criteria discussed in Chapter Two. We will conclude this thesis by summarizing our observations which resulted from comparing the codes in the third section of this chapter.

4.a The Basic Design of Our Comparison

In Chapter Two, we have discussed many criteria the are relevant to the comparison of the codes. But how to design a comparison so that all these criteria can be fully utilized is still a problem. In this section, we focus on explaining how we are going to use those test problem dependent criteria to develop a quality of solution oriented comparison. Furthermore, we will discuss our basic strategy of conducting a comparison in this type by using a critical input parameter — the tolerance.

4.a1 Quality of solution oriented comparison

One of the most important parts of comparison of codes is to evaluate the relative efficiencies of the codes. The relative efficiencies of the codes on a certain test problem, though still a vague concept, can usually be determined by using the test problem dependent criteria such as the time and storage they require to solve the problem. Test problems without the known exact solutions have been used before. But in order to fully utilize criterion accuracy, we purposely choose the test problems so that they all have known exact solutions. Up to now, relative efficiency remains to be a concept that has be widely used but not very well defined. A quite common strategy of measuring the relative efficiency of the codes has been that of first setting up the input parameters for each code such that all the codes have more or less the same parameter setting, and then compare the resulting timing, storage and accuracy from each code. The underlying motivation for this, we believe, is that when the codes have the same parameter setting, they are given the same amount of input, and therefore it is justifiable to compare the efficiencies of the codes in terms of the resulting CPU time and storage used since these efficiencies are the yields of the same input.

But it seems to us that this plausible strategy may not be appropriate for our purpose of comparing the four codes because of the fact that parameters that bear the same names, are supposed to have similar functions and were assigned the same values can play quite different roles in different programs. More importantly, not all the parameters are relevant to the efficiencies of the codes. Even if two different programs have exact the same parameter setting, the comparison of the results from the two programs that correspond to that setting may not be very meaningful. Let's use an exaggerated example to further illustrate this point: Two codes A and B have exactly the same set of parameters. When we apply them to solve the same problem, all the parameters for both programs are set to be the same constants except, say, the tolerance. When varying the tolerance, we observe that for both codes, the results get strictly better and better as the tolerance decreases. Furthermore, the two codes would have *exactly* the same results when the tolerance for code A is *TOL* and the tolerance for code B is *10×TOL*. Thus a comparison according to the same setting of parameters would result in a consistently better performance of code B over code A. However, in our opinion, this is not a fair comparison since tolerance is just a parameter and it is not related to the cost of running the codes in any way. Furthermore, the two codes A and B have exact the same *capabilities* and *efficiencies* under our assumption in the sense that no matter how well code B can perform, code A can achieve exactly the same performance at no extra cost and vice versa. The only thing different is that the two codes always attain the same level of performance with different tolerances. With a minor modification to code A, *i.e.* set *TOL* to be equal to *TOL/10,* the two codes would then perform exactly the same.

Whether the accuracy of the solution each code produces responds well to the tolerance that the user provides is an important aspect of the code and we will come to this point later on. But we felt that tolerance itself may not be considered as a performance index and is not important when it comes to timing efficiency or storage efficiency of the code. We are not in an 'input output' situation where the tolerance is the input and the result is the output when we compare the timing and storage efficiencies, even though it appears that we are.

In this thesis, we are mainly concerned with the potential of the performance of the codes rather than how well the codes can perform at a certain input parameter setting which, as was indicated by the example above, is not always relevant to the potential efficiencies of the codes. In order to design a comparison that focuses on how well the codes *can* perform (not how well they can perform under the similar input parameter settings), we split the performance related criteria into two groups according to the nature of these criteria. It is clear that when writing or running a code, what we are really after is a good quality solution to the problems we want to solve. In order to calculate the numerical solution, we need computing time and storage. One may consider the numerical solution we are after as the 'goods' and the timing and storage as the 'cost'. We intend to make a split of the criteria according to whether the criteria are about the 'goods' or 'cost'. From now on, except those that are related to the quality of the solution, all performance type criteria, most importantly timing and storage, will be referred to as cost indexes. They, as a group, define the cost needed by the codes to solve a problem. Those that are related to the quality of the numerical solution form another group and determine the quality of the solution produced by the codes. Based on this split, there are two possible aspects of the codes that can be revealed by a comparison. One may compare the best quality of the solution that is attained by each code during an experiment. More practically, one may gather those runs from the codes that yield solutions with similar quality and then compare the codes for these runs according to the cost indexes to reveal the relative efficiency. In another words, it is then possible to use peak quality solutions produced by each code and the relative efficiencies defined by cost versus solution quality to make a comparison. We call this the quality of solution oriented comparison. This new way of comparing the test problem related efficiencies will be referred to as QSO approach later in our discussion.

The advantage of a QSO approach is that it gives the relative efficiency a clear meaning. More importantly, the information concerning the connection between the cost and the quality contained by the data is utilized by comparisons of this type. The comparison where codes are compared under similar input parameter settings, however, is weak in making use of this kind of information.

However, the appealing ideas we have above suffer from the problem of being not very practical and are usually difficult to carry out. The peak quality solution, for example, will remain to be just an idea in this thesis and will not be an issue of our comparison due to the difficulties of getting the peak quality solution of the codes on any test problem. On the other hand, the major obstacle for making use of the relative efficiency described above is that there may not be a commonly agreed upon way of setting up numerical solution quality levels. It is unrealistic for us to find some kind of 'standard' that can quantify the solution quality levels in a way that it makes sense to everybody. An individual user that is particularly concerned with a specific aspect of the solution can set up a quality scale according to what he/she is concerned about. When one is only concerned about the maximum error at the mesh points or the average of the squared errors at the mesh points, for example, one can then set up some quality levels associated with the maximum error or the average mentioned above. The quality of the solutions is then quantified. Despite the fact that there exist many different quantitative features of the numerical solution, the most common focus for many experts has always been the maximum absolute error at the mesh points. Though we intend to look beyond the quantitative features of the numerical solutions as was indicated in Chapter Two, when evaluating the relative efficiencies of the codes, it is difficult for us to take into consideration the qualitative aspects of the solutions. Thus we will only use the maximum absolute error at the mesh points (for codes that provide continuous solution, the maximum absolute error at thirty equally spaced points in the domain will also be considered sometimes) to judge the quality of the solution. The qualitative features we mentioned in Chapter two, *i.e.* the form of the solution and the distribution of the final mesh points are not test problem dependent and will be compared separately.

4.a3 Collecting test problems related information for comparison

How do we collect test problem related information so that we can carry out the comparison concerning the test problem dependent criteria ? In particular, how can we find out the costs each code needs to obtain solutions at different quality levels to reveal the relative efficiencies ?

Each one of the four codes has more than ten input parameters that has to be set by the user. Some of the parameters can assume infinitely many different values. Thus it is virtually impossible to exhaust all the possible combinations of these parameters and collect information such as timing, storage and accuracy that follows. Having ruled out the possibility of looking into every combination of the input parameters, it is clear that we have to find a way to collect information such that not only the way is feasible and practical, but also the information collected this way can best represent the capabilities of the codes and is sufficient to support our comparison. Such a way is often referred to as 'testing'.

Fortunately, among the many input parameters, only a few may influence the timing, storage and accuracy that related to the codes. In many cases, the influence of a single parameter can be so dramatic that the influence of the other parameters is negligible. Based on our experience with the four codes, when compared with other parameters, tolerance, the parameter that is supposed to impose the desired degree of accuracy on the numerical solution in some way, is one of the input parameters that consistently has a dramatic influence on the performance of the codes, no matter what kind of test problems we use. Furthermore, it is designed to influence the solution in a fairly predictable way (we will come to this in the next paragraph) and it is the *only* such a parameter that every code has in common. Despite many problems it may cause, in order to get rid off the difficulties of being entangled by the infinitely many combinations of the input parameters and infinitely many solutions that follows, some kind of 'clear cut' approach is inevitable. Our 'clear cut' approach in this thesis is that when using the codes to solve a test problem, we set all the input parameters except the tolerance in a way that we believe will best serve the codes in terms of producing quality solutions by them. We then vary the tolerances from *1.D-2, 1.D-4, 1.D-6* to *1.D-8* and run the codes with these values of tolerance to collect information such as accuracy, timing and storage that we need to reveal the relative efficiencies. We will provide more details about this approach in Appendix (I). As one will see from there, this approach is feasible and practical.

The four codes differ from one another in the number of tolerances allowed. COLNEW, COLSYS and HAGRON allow the user to specify a tolerance for each variable (in the case of COLNEW and COLSYS, these variables may be any component of z in *3.2a&b*), while MUTS allows a user to provide only two tolerances. The impact the tolerances have on the performance of the codes is usually test problem dependent and varies from code to code. Based on our experience with the four codes, when we set all the input parameters except the tolerance in the way we discussed above, the CPU time and storage needed by each code for solving a problem generally increases as the tolerance decreases, but the accuracy of the solution generally gets better, provided that the codes return normally. Thus when we take the above approach, we can get solutions in *different* quality level as well as the related costs upon normal returns from the codes. On the other hand, if the codes fail at some small tolerance, we then know the limits of the codes in terms of the best quality solution the codes can provide. These limits, as well as the reasons for failure when the codes are brought to these limits, are important for comparing the codes in terms of the robustness of the codes, and to what degrees a BVP for ODE can be solved by the codes.

Being able to find out the solutions at *different* quality levels together with their accompanying costs, and detect the limits that are mentioned above enables us to collect information about these solutions and costs and carry out the test problem related comparison. The information collected by this approach is exactly what a QSO type of comparison needs.

4.b The Comparison of the Codes

4.b1 Codes' driver related comparison

1) The form of the BVPODE that can be directly dealt with by each code

COLNEW and COLSYS can be directly applied to a system of ODE with high order (greater than 1) equations while HAGRON and MUTS solve first order system exclusively. When using HAGRON and MUTS for a system of ODE with high order equations, one must first rewrite the system as a first order system. For details on how to change a higher order equation into a first order system, please see section one of the first chapter.

COLNEW and COLSYS accept multipoint boundary conditions but require that they are separated. The current version of HAGRON accepts two point separated boundary conditions only. MUTS is not restricted to separated boundary conditions but like HAGRON, it accepts two point boundary conditions only.

The form of the BVPODE that can be directly dealt with by each code is not critical in theory since a mixed order system with general multipoint boundary conditions can be recast into a first order system with separated two point boundary conditions, which all the four codes can be directly applied to [2]. However, when a higher order equation is changed into an equivalent first order system, a multipoint boundary condition is changed into an equivalent two point boundary condition, or a non-separated boundary condition is changed into a separated one, the number of equations will be increased and the transformation needed may be rather cumbersome. Thus this issue does reveal the disadvantages and advantages of each code in the sense that when a problem is not in the form that the a code can be directly applied to, the transformed problem may be much more expensive to solve due to the increase in dimension of the problem. The transformation needed can also be a great difficulty to many users.

Whether this issue brings advantages or disadvantages to a code will certainly change according to different users. If you only want to solve a two point boundary problem, being able to solve a multipoint boundary problem is not an advantage to you. Otherwise, it is. Nevertheless, we noticed that while the types of problems that can be directly handled by COLNEW, COLSYS and MUTS are not entirely overlapping, they contain the type of problems that HAGRON can be directly applied to.

2) Input parameters

There are roughly two types of input parameters. The first type are those that are used to describe the objective problem, such as the dimension of problem or whether the problem is linear or nonlinear. These parameters are objective problem dependent, and the user cannot choose these parameters freely. The second type are those that have to be set by the user when using the codes. This type of parameters includes the tolerances, the dimensions of some working arrays, etc. The setting of this type of parameters, like tolerance we discussed in the last section, is usually critical to the performance of a code.

3) Input subroutines

COLNEW, COLSYS and HAGRON all need four subroutines for linear problems and an optional fifth subroutine for nonlinear problems (see Chapter Three for the description of the subroutines). Through these subroutines, the right hand side of the equation, the boundary conditions and their the first derivatives are evaluated and conveyed to the codes. When solving a nonlinear problem with the optional fifth subroutine, the initial solution as well as some of its derivatives are also conveyed to the codes.

MUTS, on the other hand, needs only two subroutines when solving a linear problem and three subroutines when solving a nonlinear one. When solving a linear problem, it differs from the other three by having a parameter to tell the code whether or not the problem is homogeneous. The two subroutines needed are used to evaluate the homogeneous part of the system and the whole right hand side of the system. In the case of a homogeneous problem, one only needs one subroutine. When solving nonlinear problems, the three subroutines needed are used to evaluate the right hand side of the system, the initial solution, and the boundary conditions together with the derivatives of the boundary conditions.

When the derivatives of the right hand side of the system are easily available, the first three codes appear to have the advantage of making use of more information. But when the first derivatives are not easily available, MUTS has the advantage of not depending on these derivatives, and in this case it is the easiest choice.

4) Ease of Use (I)

If compared with codes for many other purposes, all the four codes are fairly difficult to use in general. In particular, COLNEW, COLSYS and HAGRON require the derivatives for both the right hand side of the system and boundary conditions. When the right hand side gets complicated, this may be the place to watch out for the errors. The current version of HAGRON puts the numerical solution into a one dimensional array that is not well explained by the available documents. MUTS is relatively easier to use in terms of the complexity of its driver. This is particularly the case when one solves a linear problem, and needs only two simple subroutines.

All the codes appear to have a common problem of having too many options or too much output information for ordinary users. The setting of *IPAR(10)* for COLNEW and COLSYS, *IPAR(15), IPAR(16)* for HAGRON and output arrays *U, Q, D* from

MUTS, for example, may not be of great importance to an ordinary user, but their presence surely makes the codes more difficult to run since it is hard to decide what to do with them. While we noticed that the codes, such as HAGRON, are still in an experimental stage, a practical idea for diminishing this problem may be producing different versions of a code that suit different groups of users.

4.b2 Comparison concerning the qualitative aspects of the solution

1) The form of the solution

The four codes offer two different types of solutions. COLNEW and COLSYS produce continuous solutions on the entire domain. HAGRON and MUTS produce solutions at some mesh points that are either chosen by the user or automatically determined by the code.

When one is only concerned with the solution at some discrete points, one may not care about whether a code produces continuous or discrete solution. But the same thing cannot be said when one wants a continuous solution on the entire domain. The ability of being able to produce a continuous solution is clearly an advantage. In this sense, the solutions from COLNEW and COLSYS are more desirable.

2) The number and the distribution of the final mesh points

It had been considered to be more efficient if a code can solve a problem with fewer final mesh points. Since the number of final mesh points is directly related to the storage that is needed to produce a solution, the *minimum* number of final mesh points on which a code can provide a satisfactory solution may be used to reveal the minimum amount of storage each code needs in order to solve the problem (the solution one gets with this minimum storage is usually not as good as the solution when more storage is supplied and be used by a code). But it is not appropriate to compare the number of final mesh points needed by a code that produces a discrete solution with that of a code that produces a continuous solution. If we put aside other issues concerning the quality of solution, it is true that if a code produces a continuous solution, the fewer the number of final mesh points are, the less the amount of storage the code needs to solve the problem, and then the more efficient the code is. When you have a continuous solution, the solution at the mesh points (though it is usually of higher accuracy) is not of particular importance. But when you only have a solution at some discrete mesh points, the fewer the number of final mesh points there are, the less the program tells the user. Thus a big or small number of final mesh points has both an advantage and disadvantage if a code only produces a discrete solution. When two codes can solve a problem with the similar amount of storage, the one with the bigger number of final mesh points is clearly more efficient in the sense that it tells the user more.

The distribution of the final mesh points is also very important. When solving a real problem where the exact solution is not known, users have to rely on the code to provide a numerical solution that can best characterize the unknown exact solution. Our first test problem has a boundary layer near zero (see 114). Assuming that we are not aware of this boundary layer, when using COLNEW, COLSYS and HAGRON to solve this problem, the mesh points can be automatically determined by the codes. One can see from one of the graphs on 114, 115, 117, 120-125 that the final mesh points from these three codes are very reasonably distributed in the sense that the density of the points in the small region where one of the solutions changes rapidly is much higher than the density in any other places. With the distributions of the final mesh points like this, the behaviour of the exact solutions are then well characterized. But when we use MUTS, we have to set the mesh points all by ourselves. Given that we do not know the existence of the boundary layer, we are simply not able to set the points reasonably as was automatically done by the other three codes. The only thing we can do is to choose a set of equally spaced points in the domain. But the boundary layer can be easily missed if the number of points we choose is not big enough. However, a big number of mesh points usually results in not only a big storage requirement, but also a big amount of CPU time. We do not want to go that far as to discuss the mesh selection strategy used by COLNEW, COLSYS and HAGRON and whether this strategy can or cannot be adapted by MUTS. From a pure user point of view, the first three have a clear advantage over MUTS for their ability of detecting the important features of the solution by themselves and their ability of distributing the mesh points efficiently. It is possible, however, to let MUTS determine the output points by itself by setting NRTI to zero and set AMP to some small value. But the distribution of the output points is not determined by the shape of the exact solution of the problem as was by the first three, rather it is determined by the shape of the corresponding fundamental solution [2]. Consequently, the exact solution usually cannot be well characterized by the numerical solution from MUTS. See 118, 126 and 127 for examples.

3) Error analysis

By looking at the location of the maximum absolute error and the graph of the absolute errors at the final mesh points, we expected to see some kind of patterns regarding these two aspects to emerge from our experiments that might be related to each code. We anticipated that the errors at the points in those regions where the solution changes dramatically would be bigger and the maximum error would be located at these points. But to our surprise, this did not always happen. From the graphs in Appendix II, one can see that the maximum errors are not always located at the places that we thought they should be although in some cases (see 120-122) they are. The overall error curves for the four codes are also somewhat random except that the curves for HAGRON appears to have more oscillations.

We intended to include error analysis as a part of the quality of solution. But the observation that there is unlikely to exist any pattern that the errors follow makes this issue incompatible. Should more test problems and data become available, some useful patterns might be found.

4.b3 Relative efficiencies and robustness

The following is a comparison concerning the test problem dependent criteria, mainly the relative efficiencies defined in the first section, as well as robustness, of the four codes on the eleven test problems we chose. Though a quantified comparison is more desirable, we felt that the testing we conducted is more supportive to a qualitative comparison, and the comparison we have below is more qualitative than quantitative.

1) Quality of solution oriented efficiency

1.1) Timing

The comparison between COLNEW and COLSYS is straightforward because they both use the same mesh selection strategy and both produce continuous solutions. If the accuracies of their solutions are the same, then the solutions can usually be considered to be of the same quality. Even though there are some exceptions, COLNEW is generally more efficient in terms of CPU time needed to produce a solution of certain degree of accuracy than COLSYS. In most cases, the ratio of COLNEW's and COLSYS's CPU times that correspond to the same accuracy is between 0.9 and 0.7 (see Appendix III *1.a*).

About one and half year ago, we did a separate study comparing COLNEW to COLSYS. One of the main purposes of that study is to find out the reason why COLNEW is usually faster than COLSYS. For each test run, we monitored the amount of CPU time a code spent on its linear system solver and the total amount of CPU time it spent on the test run. We then compared these CPU time for the two codes under the condition that the input parameter settings (for the two code) associated with these CPU time are the same (this is easy to do since COLSYS and COLNEW have exactly the same set of input parameters). From the data we collected, we found that the proportion of CPU time used by the linear system solver in COLSYS is much higher than that proportion of COLNEW. The difference between the amount of CPU time used by the two linear system solvers is often close to the difference between the total amount of CPU time used by the two codes. Based on these observations, we believe that the linear system solver in COLSYS is the major reason for COLSYS being slower than COLNEW. Decisive evidence for this may be found if one replaces the linear system solver in COLSYS with the linear system solver in COLNEW.

When we compare COLNEW, COLSYS with HAGRON, it appears to us that HAGRON is competitive with COLNEW and COLSYS in terms of the CPU times needed in order to produce solutions of the same quality, regardless of the type of maximum error for COLNEW and COLSYS that is used for measuring the solution quality (see Appendix I for the description of error types). In Appendix III, we compared the four codes in term of timing a pair at a time on all the test problems. By and large, when the first type of error for COLNEW and COLSYS is used, HAGRON outperforms COLNEW and COLSYS in terms of this criterion on L2, L5 and L6. COLNEW outperforms HAGRON on L1, L3, L4, N2 and N4. COLSYS outperforms

HAGRON on L1, L3, L4, and N2 (see the related tables in Appendix III 1.a). A comparison based on the second type of maximum error for COLNEW and COLSYS also gave us similar results (also see Appendix III 1.a). With limited test problems, it is hard to tell whether this relative efficiency is related to the dimensions or the behaviour of the exact solutions of the test problems. Taking into consideration that our test problems were independently selected in the sense that they were chosen before we ran them with the four codes, it is fair to say that HAGRON is competitive with COLNEW and COLSYS on this issue.

It is most difficult to compare the rest of the four codes with MUTS mainly because it requires the user to specify the number of output points, and both the CPU time needed by MUTS and the accuracy of the solution depend not only upon the value of tolerances but also the number of output points. When running MUTS on L3 with *a=11, TOL=1.D-4*, for example, the following table shows what we get when we vary the number of output points:

NRTI	CPU	ERROR
20	40	.32D-2
100	147	.23D-5
200	289	.70D-7

CPU is measured in milliseconds.

Even though the code can determine the output points by itself when NRTI is set to zero, we found that no matter how small we set AMP to be (as required, it has to be greater than 1), we often only got output at the two boundary points. Such examples are those homogenuous linear problems with right hand sides equal to zero. In these cases, the fundamental solutions are all constants and no matter how small AMP one set, one cannot get additional output points. With only two output points, we simply cannot say that the quality of the solution is compatible with that of the other three. Thus it is necessary to set it to some value that is reasonably bigger than zero. According to our experience with MUTS, when the required number of output points is big (more than 300), MUTS is considerably slower than the other three. We thought about using the final mesh points from COLNEW, COLSYS or HAGRON as the required output points for MUTS and then compare the resulting CPU time and accuracy with that of the others, but we decided not to do so because this will put

MUTS in an secondary position and there is no sensible justification for this. In the meantime, it is also difficult for us to collect the data if values of NRTI are allowed to vary among all the positive integers. We eventually decided to set NRTI to be the smallest number of in the set *{20,50,100,200,300}* that produces an accuracy that is about the same size as the value of the corresponding tolerance (also see Appendix I). With the input parameter settings described above, we found that MUTS runs slower than the other three on most of the ten test problems (one of the eleven test problem is not used except for comparing the robustness) in terms of CPU time needed to reach a certain degree of accuracy (see Appendix III *1.a*). While this is observed through using a particular set of input parameters on the ten test problems we selected, we believe that MUTS, when compared to the other three, is generally less efficient in terms of timing efficiency we described above.

1.2) Storage

It is extremely difficult to keep track of exactly how much storage a code needs in order to produce a solution at a certain degree of accuracy. When using COLNEW, COLSYS and HAGRON, one has to provide more than what the codes need in order to run the codes. Because one never knows how much storage they will really need beforehand, when writing the driver, one usually can only take a guess and the question that follows is that there are two storages here, one is that provided by the user, another one is that actually needed by the code. Which one should we compare ? Furthermore, like MUTS where the number of output point is related to the quality of solution, the quality of solution of the three codes may be related to the amount of storage available. COLSYS, for example, can often produce decent solutions even when it needs more storage to insure that the required tolerance is achieved. When solving L3 with *a=55, TOL=1.D-8,* with sufficient and insufficient storage, the solutions differ in quality but they are both acceptable.

	I. S.	S. S.
FMP	375	1281
ERR1	.115D-9	.696D-11
ERR2	.273D-8	.562D-11

I.S. = insufficient storage, S.S. = sufficient storage, FMP = number of final mesh points

ERR1 = the max error at mesh points, ERR2 = the max error at 30 equidistant points

We did record the numbers of final mesh points when we ran the codes. It is possible to detect the maximum number of final mesh points allowed by a code when the amount of storage as well as the dimension of the object problem are known. But since the same storage allocation for different codes may result in a different maximum number of final mesh points, and different codes may need different amounts of storage in order to produce the same number of final mesh points, the relationship between the storage that is actually used by each code and the number of points in the final mesh is not clear. This makes it difficult to compare the codes' storage efficiency via the number of final mesh points we recorded. However, if we look at only a single code with a fixed input parameter setting, more final mesh points or output points always goes with a bigger storage requirement needed by the code. See section Testing (Appendix I) for details about how the storage for each code is set when we run the test problems.

Due to the above difficulties, we felt that with the set of data we collected on the ten test problems, a detailed comparison of the relative efficiency in terms of the storage versus the quality of the solution is not possible.

However, it is possible to comment on the flexibility each code has on the use of storage. When using COLNEW, COLSYS or HAGRON, if *IPAR(11)* is set to zero, the final mesh is totally determined by the codes themselves. Even when IPAR(11) is not equal to zero, usually there are still many points in the final mesh that are determined by the codes. Thus the final meshes for these three codes are more or less beyond the user's control. Since the final mesh is very closely related to the storage requirements of each code, the storage requirements for the codes are also beyond the user's control. Recall that the ability of automatically determining the final mesh points brought advantages to the three codes when we discuss the quality of solutions. This same issue is now bringing the three codes disadvantages. With storage requirements that can not be controlled by the user, the codes may waste storage to produce some undesirable information for the user at those mesh points that are not needed by the user. When one only wants the solution at one point in the domain, for instance, the three codes still have to include many other points in their final mesh and produce solutions at these points. Another clear pattern which emerged from the data we collected is that the storage needed by COLNEW, COLSYS and HAGRON usually increases dramatically as the accuracy of the solution one wants increases. This is evidenced by the fact that as we increase the tolerance during our experiment, often the number of points in the final mesh for these three codes also increases rapidly (see Appendix III section c).

Unlike the other three codes, in order to improve the accuracy of its numerical solution, MUTS usually does not need more storage. It is difficult to say exactly how much storage it really needs in order to solve a certain problem even if the number of output points is supplied by the user. The parameter AMP is often responsible for those output points that are not part of output points supplied by the user. Based on our experience, MUTS is generally more flexible than the other three codes in terms of making use of the storage to serve users' various needs. When we solve L1 with eps=1.D-2, TOL=1.D-4 and IPAR(11)=0, for example, the following is the (minimum) number of points in the final mesh we found

Code	**FMP**
COLNEW	11
COLSYS	11
HAGRON	32
MUTS	2

FMP = the number of points in the final mesh

One might be able to set the array *FIXPNT* to get different numbers of points in the final mesh for COLNEW, COLSYS and HAGRON, but when you want only the output at the two boundary points, it is unlikely that the three codes can match MUTS which can give you exactly the solutions at the two boundary points only. When one only wants the numerical solutions at certain points and the value of AMP is properly set, MUTS can usually do the job very efficiently without producing any undesirable information at any other points. Even when one wants a highly accurate solution at only a few points, with small tolerance, MUTS can usually produce solutions at a higher degree of accuracy without any additional output points and storage. This clearly brings MUTS an advantage over the other three codes, and it means a big saving in storage when one wants a highly accurate solution at only a few output points.

How important this feature is to the portability of a code in terms of the feasibility of using the code on all kinds of machines is beyond the scope of this thesis.

Our speculation is that this feature may make a code like MUTS a very natural candidate for small machine with a small amount of storage available. The other three, on the other hand, are restricted by their basic needs for big storage.

3) Robustness

Robustness, the reliability of the codes or more precisely the 'degree to which they can solve a large class of problems and exit gracefully if not' [17], is a very important issue not only to the software developers, but also to the codes users.

In order to compare the codes on this issue, when running the codes on the test problems, we looked at the following four aspects: 1) At what level of difficulties (when a test problems with a parameter that controls difficulties of the problem is used, the level of difficulties is represented by the value of the parameter) that each one of them starts to fail. 2) For what reason a code fails and is the failure easy to fix. 3) What one can still get when a code fails. 4) the flexibility of the codes in dealing with problems that involve some singularities. By fail or failure above, we simply mean any kind of abnormal exit from the codes or normal exit with a wrong solution. It should be noted that when running a code on a test problem, in case the codes are very expensive to run, only the first or the first and the second failure is recorded, *i.e.* we did not run the code for different values of the parameter(s) that correspond to even higher degree difficulties. The following comparison is based on the testing described in Appendix I.

Our observations concerning the first aspect of robustness are mainly from the runs on the three linear test problems L1, L4, L5 and two nonlinear test problems N2 and N3 where the program failures had occurred. All these five test problems, as it is described in Appendix I, have some parameters that control the difficulties of the problems. When we test the codes on a test problem of this kind, the difficulty of the problem always increases in the order of the tables by which the test results are recorded (see Appendix III, *3*). *e.g.* the second table in table page 1 in Appendix III section three correspond to a problem of higher degree of difficulty than that of the first table. From the data we collected (see Appendix III section two), it appears to us that COLNEW, COLSYS and HAGRON are more capable in terms of their ability to solve difficult problems than MUTS. On the five test problems where most of the codes failures occurred, *i.e.* L1, L4, L5 and N3, MUTS failed the earliest on L1, L4 and N3.

On test problem N2, it is the only code that had failures. COLNEW, COLSYS and HAGRON appear to be quite competitive on this issue since they all failed on L1, L2, and L3 at the same degrees of difficulties. Though we noticed that HAGRON also had failures on N3, it is difficult for us to make any comment beyond what we had above due to the limited number of test problems we had.

In our experiments, the causes for the codes' failures are mainly: 1) the storage needed exceeds the allowed limit. 2) program overflow (*e.g.* a divisor is found to be zero or an exponent is outside the domain of the machine exponential function). 3) the clock time needed exceeds fifteen minutes. 4) Unacceptable solution in terms of both absolute error and relative error. Program failures for COLNEW and COLSYS are mostly due to the first kind cause, *i.e.* storage needed exceeds the allowed limit. The only other cause of failure we experienced for these two codes is the fourth cause. Failures for HAGRON are also mainly due to the first kind of cause. The second major cause for HAGRON is also unacceptable solutions. Apart from these two causes we also experienced occasional program overflow as well as the third kind of cause. Unlike the first three codes, the limited storage brings little trouble to MUTS. Instead, most of the failures for MUTS are caused by unacceptable solutions and program overflow. Occasionally, we also found that the number of nonlinear iterations needed is greater than the limit (50 times) is responsible for the failure. In one case, the third one in the list is found to be the cause.

Among all the causes for failures, in our opinion, only the first cause is easy to fix as long as getting more storage is not a problem. Often, when COLNEW, COLSYS or HAGRON fail because of this reason, we add some more storage to the codes and then the codes work properly. The second kind of cause is harder to deal with but one may try to change the parameter setting of the code and sometimes one can avoid this problem. When solving L1 using MUTS with eps = 1.D-4 and TOL = 1.D-2 , for example, we found that when we change the parameter NRTI (shown in the table below), we can get rid of the overflow.

NRTI	EXIT
20	overflow
50	overflow
100	normal

The third cause, though it appears to be easy to deal with, is a difficult one for people with limited computing resources. As a matter of fact we did not intend to relate it to the robustness of the codes in the beginning. When we started running the codes on the test problems, we simply let a program run until it stopped by itself. However, we soon found that we must impose a elapsing time limit on the runs due to the unaffordable computing expenses. When using HAGRON on L4 with *eps = 1.D-4* and *TOL = 1.D-8*, we tried to let the program stop by itself so that we could bring the program to its limit and see how much time the program may take on that problem. But it did not stop for three hours and it was eventually interrupted by the system manager when the computer account was suspended due to the excessive use of computing fund over the limit that the account was allowed. Another example is when we used MUTS on L1 with eps = 1.D-4, TOL = 1.D-6 and NRTI = 100, the program did not stop for about an hour and had to be interrupted by us due to the expenses. The last cause is more serious in the sense that if the exact solution is not known, the user may have no idea whether or not an abnormal exit due to this cause has happened. When the exact solution of the problem has a very sharp spike or is highly oscillatory in a small region and is relatively very smooth in the rest of the domain, for example, the codes may not be able to detect the spike or oscillatory behaviour of the exact solution as the initial mesh is not fine enough for some mesh points to fall into the rough region. This often is the reason for the fourth type of failure. Provided that the rough region is not too small, this may be avoided by either increasing the number of final mesh points or using a tighter tolerance to produce finer meshes and increase the chance of detecting this kind of rough region. When we solve L4 using COLNEW, COLSYS and HAGRON with eps=1.D-4, for example, we observed the following.

	TOL	NMIP	ERR_{max}
COLNEW	1.D-2	default(=5)	1133.18
COLSYS	1.D-2	default(=5)	1133.18
HAGRON	1.D-2	default(=6)	10000

	TOL	NMIP	ERR_{max}
COLNEW	1.D-2	***20***	0.39D-1
COLSYS	1.D-2	***30***	1.77D-1
HAGRON	1.D-2	***100***	2.95

	TOL	NMIP	ERR_{max}
COLNEW	*1.D-4*	default(=5)	0.24D-3
COLSYS	*1.D-4*	default(=5)	0.55D-1
HAGRON	*1.D-4*	default(=6)	0.23D-2

NMIP = number of points in the initial mesh.

It is clear that the solutions from the three codes in the first table is not acceptable. By increasing the number of points in the initial mesh or tightening the tolerance (or both), one may get a much more accurate solution. Thus using tighter tolerance or a finer initial mesh can reduce the possibility of this kind of failure.

When a code fails due to the second and third causes, we do not have a solution. When a code fails due to the last cause, we have a wrong solution. Thus if a code fails due to one of these three causes, we cannot get anything from the code. On the other hand, one often can get useful information from failures due to the first kind of cause. In particular, COLNEW and COLSYS often can still provide a partially converged solution in the case of this kind of failure, *e.g.* Table 1 and Table 2 in section three of Appendix III. One may make use of these partially converged solutions for continuation as was described in Chapter Three.

There are some problems that involve some harmless singularities. As an example, L7 is a simple second order two point boundary problem, but the coefficient matrix has a singularity point *0*. COLNEW and COLSYS do not have to evaluate the coefficient matrix at the boundary points, thus such a singularity is of no threat to them. The ability of dealing with problems of this type naturally enhanced their robustness in the sense that they solve a broader range of problems. On the other hand, a singularity of this type is somewhat insurmountable to HAGRON and MUTS due to their dependence on the values of coefficient matrix at the boundary points.

Based on the discussion above, COLNEW, COLSYS and HAGRON appear to be clearly more robust than MUTS. Not only they have fewer failures than MUTS but also the causes for their failures are more concentrated and less harmful. Furthermore, during our testing, we had experienced the fewest number and types of abnormal exits from COLNEW and COLSYS. Even when the needed storage exceeded the allowed limit, COLNEW and COLSYS were often able to bring us some reasonable solutions and more importantly they were able to terminate by themselves in a relatively short period of time. Add all these to their ability in dealing with the problems with singularities, COLNEW and COLSYS, in our opinion, are even somewhat stronger than the current version of HAGRON with respect to the robustness of the code.

We conclude the comparison of the robustness of the codes by pointing out that according to the separate study we conducted a year ago and our testing results in Appendix III, COLNEW did not appear to be more robust than COLSYS though it uses a different bases for representing the numerical solution. The robustness of COLNEW and COLSYS is deeply rooted in the spline-collocation method and this robustness did not seem to be improved by the new bases used in COLNEW.

4.b4 Accuracy, tolerance and termination criteria

The accuracy of the numerical solution a code produces, the input tolerances and termination criteria for the code are very closely related to each other. By accuracy of the numerical solution, we mean the absolute or relative error of the numerical solution. The tolerances, on the other hand, are some input parameters that have great influence on the accuracy of the numerical solutions through the role they play in the termination criteria. In theory, when the tolerances decrease, the numerical solution gets more accurate.

As was described in the last chapter, the termination criteria for COLNEW and COLSYS is: if $TOL(j)$ is the tolerance related to the $LTOL(j)^{th}$ component of $z(u)$, the codes will attempts to satisfy on each subinterval

$$|(z(v) - z(u))_{LTOL(j)}| \leq TOL(j)\ (|z(u)_{LTOL(j)}| + 1)$$

where $v(x)$ is the approximate solution vector and $u(x)$ is the exact solution of *3.1a&b*. The termination criterion for HAGRON is: if $TOL(j)$ is the tolerance related to the $LTOL(j)^{th}$ component of $u(x)$, the code will attempt to satisfy at each grid point x

$$|(v(x) - u(x))_{LTOL(j)}| \leq TOL(j)\ max(|v(x)_{LTOL(j)}|,\ 1)$$

where *v(x)* is the approximate solution vector at the grid point *x* and *u(x)* is the exact solution of *3.2a&b*. The norms involved in the two inequalities are maximum norms.

How these termination criteria were approximately carried out by the codes is not our concern here. But the influence of tolerance on the accuracy of the solution is now clear. It is not the real accuracy of the numerical solution yet the numerical solution is expected to be more and more accurate as the tolerance decreases. Users may like the idea of having an input parameter that can be used to specify the accuracy of the numerical solution they want rather than a tolerance that is not really the accuracy of the numerical solution. But this is clearly impossible since the exact solution is not known.

It is important to see that the input tolerance is neither always an upper bound for the absolute errors of numerical solutions nor always an upper bound for the relative errors. For a specific problem, it can be considered as one (and only one) of these two bounds. If the size (maximum norm) of the numerical solution for a certain problem is smaller than ten, the input tolerance can be considered approximately as an upper bound on absolute error of the numerical solution. Otherwise, it can be considered as an upper bound on the relative error.

When one uses these three codes, how should one impose a desired accuracy on the numerical solution by using the input tolerance ? If the size of the exact solution is completely unknown, one does not know whether the input tolerance is going to play a role as an upper bound for the absolute error or an upper bound for the relative error. In this case one has to rely on a test run (when testing the size of the solution, in order to save computing time, set the tolerance to a relatively bigger values, say 0.1) to find out the size of the solution. Once the approximate size of the solution is known, if this size is smaller than ten, user specified tolerance imposes an upper bound on the absolute error of the numerical solution, otherwise the tolerance imposes an upper bound on the relative error.

Unlike COLNEW, COLSYS and HAGRON, MUTS has only two tolerances regardless the dimension of the problem (see parameter ER in Chapter Three—MUTS' documentation). Though we do not have information on how the two tolerances are involved in the termination criteria of the code, in MUSL ER(1) and ER(2) stand for the relative and absolute error tolerances, respectively.

L6 is an example where the domain of the problem and magnitude of the solution (rather than the difficulty of the problem) are controlled by two parameters. One can see that as the magnitude of the solution increases, the absolute error increases but a simple calculation can show that the relative error stays more or less the same. The maximum absolute error of the numerical solution is not always equal to or less than the corresponding tolerance, but the relative error always is. When one sets the tolerance to a certain value, say 1.D-4, one will not be able to know whether this tolerance will result in a maximum absolute error that is less than 1.D-4 or a relative error that is less than 1.D-4 until the numerical solution is calculated, provided that the code terminates exactly as we described above. If the magnitude of the solution is less than one hundred, then the maximum absolute error of the numerical solution is about or less than 1.D-4. Otherwise the relative error is about or less than 1.D-4.

4.b5 User feedback

Each one of the four codes provides many kinds of feedback. It is difficult to say a code is superior to the others in terms of its user feedback since the feedback is often tied with the codes' underlying numerical methods. COLNEW, for example, can provide the user with a complete mesh as well as the approximate numerical solution at the mesh point at every step. But when one uses MUTS, since the mesh points are often chosen by the user and may not vary in the process of solving the problem, there may not be such a varying mesh like that of COLNEW that the user may be interested in.

However, it is possible to comment on the focus of the user feedback of each code. Codes that are based on finite difference methods, *i.e.* COLNEW, COLSYS and HAGRON, provide more information about their process of solving the problem such as the current mesh and have less feedback about the driver setting (input parameters and input subroutines). The biggest problem is that the programs are not able to tell the user whether the allotted storage is enough until the user has a test run. MUTS, on the other hand, provides more feedback about the correctness of the driver. The storage needed by MUTS usually can be estimated very accurately before running the code. However, once the code starts running, little about the process of solving the problem is available.

For all the four codes, modifications concerning their user feedback are needed before they are used to serve people with little knowledge about the literature of ODEBVP. We recommend that the user supplied derivatives be checked by the codes. This may be done by using a standard driver program that calls the routines having this purpose from a software library or by including a subroutine in the codes that does the checking. As incorrect derivatives are the major problem when writing a driver, such a checking is important.

4.b6 Ease of use (II)

In Ease of use (I), we compared the codes on this issue by focusing on the complexity of their drivers. The following are some more observations that can be related to the ease of use for the codes.

Even though we used the whole of Chapter three to describe the meaning of the input parameters and subroutines and explain how to use the codes, there are still a few hidden problems that are important to know about for using the codes properly. From Chapter three, one can hardly see the importance of many input parameters such as IPAR(2) for COLNEW and COLSYS. Can we set this parameter to any legal value described by the program documents? What kind of impact do these parameters have on the solutions? When solving L2 with eps = 1.D-6 and TOL = 1.D-2 using COLSYS and COLNEW, we observed the following change of the maximum absolute error at the mesh points:

IPAR(2)	COLNEW	COLSYS
3	0.29D-5	0.29D-5
4	0.92	0.92
5	0.83	0.83
6	0.84	0.84
7	1.01	1.01

**The last two columns above coincide to the number of digits shown above.

We mentioned how the number of output points can affect the returns from MUTS when we compare the robustness of the code. The number of output points also affects the accuracy of the solution. When using MUTS to solve L2 with the same eps and TOL above, the following table shows how the accuracy of the numerical solution changes with the number of output points:

NRTI	ERROR_{max}
20	0.14
50	0.18D-1
100	0.16
200	0.22D-1

These parameters often significantly affect the solutions, especially when the tolerance is not very small (>1.D-2).

Another interesting observation is about the tolerance. It is clear from *4.b4* that the tolerances are not the desired accuracy. But it may be considered as a good approximation of the upper bound of the absolute accuracy when the norm of the solution is reasonably small and may be considered as that of the relative accuracy when the norm is big. A question that arises naturally here is how well does the accuracy of the numerical solution correspond to the input tolerance?

Based on our testing, all the four codes respond well to the user specified tolerance if one is only concerned that the resulting accuracy is not greater than the tolerance. For all the four codes, when the norm of the solution is small, the maximum absolute error is usually less than or of the same size as the tolerance. When the norm of the solution is big, the maximum relative error is usually less than or of the same size of the tolerance. In the case of the smooth problems, all the four codes usually give accuracies that are far smaller than the input tolerances (see section three in Appendix III for the results on L6 for an example). In particular, HAGRON has fewer input parameters that affect the solutions. Its numerical solutions also appear to have a stronger relationship with the input tolerance than the other three codes. When the it exits normally, the accuracy of the numerical solution is usually about the same size as the tolerance and sometimes less than the input tolerance. For the other three codes, the accuracy of the numerical solution may be bigger than the tolerance if some other parameters, such as IPAR(2) for COLSYS and COLNEW and NRTI for MUTS, are not properly set. This makes HAGRON easier to use in terms of getting the desired accuracy by setting the tolerance, as how to set NRTI for MUTS or IPAR(2) for COLNEW and COLSYS is a very difficult question. As the last remark concerning the ease of use, we point out that this feature of HAGRON is important and very useful to ordinary users for whom the role of the parameters is difficult to understand yet the confidence about the correctness of the solution is critical.

For a brief discussion concerning the relative timing efficiency involving the correspondence between the tolerance and the accuracy, please see part 1.b in Appendix III.

4.c Conclusions

By evaluating the relative efficiency of the codes using a QSO approach, we are actually comparing the potential ability of the codes. The question that how these revealed potentials might be utilized is left behind.

However, it is not difficult to see that QSO approach can be useful. When comparing HAGRON with COLSYS, for example, the traditional approach leads to the conclusion that HAGRON is faster than COLSYS (see [5], [6] and section 1b in Appendix III of in thesis), while our QSO comparison indicates that COLSYS is quite competitive with HAGRON in terms of speed (see 1a in Appendix III). Why do these two approaches lead to different results? When using the traditional approach, a code can be faster than another one even if the two codes converge to the exact solution at the same speed, as long as one code has a *better* error estimator and stops earlier than another one when the required tolerance is satisfied. By taking a QSO approach, the effect of the error estimator is eliminated, and a code can be faster *only* when it converges to the exact solution faster. Thus the results of the two different approaches indicates that the speeds the two codes converge to the exact solution are competitive but HAGRON has a better error estimator. However, we must emphasize that comparing these codes using a QSO approach is strongly biased against HAGRON since the efficiency of the error estimator involved in each code is purposely discounted and having a good error estimator is one of the strongest advantages of HAGRON. Incidentally, the fact that HAGRON has a better error estimator was well known among BVPODE experts. Our study fully agrees with what was known. Furthermore, by including a QSO type of comparison, this study also reveals that HAGRON does not generally converge to the exact solution faster than COLSYS.

The traditional approach has many distinct merits of its own, *e.g.* its results can be easily understood and utilized by the user. However, most of the codes do not yet have the abilities to communicate with the user so well that what the user wants can always be precisely translated into some input parameter settings and then carried out by the codes. When a user wants a solution of an accuracy 1.D-4, for example, there is no code that it can guarantee an accuracy of 1.D-4 by using certain input parameter setting. There are always some random factors involved in the process of solving the objective problems and these random factors make it impossible to predict fairly accurately the goodness of the solution by just look at the input parameters. Thus we felt that the input parameter setting for ODEBVP codes, though it often carries our expectation of the numerical solution and with a certain amount of experience it can be used to predict the goodness of solutions to some degree, does not have a definite relationship with the numerical solutions. The relative efficiency with respect to certain input parameter settings is therefore not the relative efficiency with respect to the quality of numerical solution, and it cannot give us the insight concerning the potential of the codes.

Nevertheless, if all the codes have almost the same sets of input parameters, the parameters in different codes that bear the same names also function similarly in the codes they belong to and the quality of the solutions produced by each code can be very much determined by their input parameters in a common way, then different codes with the same parameter setting will produce numerical solutions that are of the similar qualities. In this case, the quality of numerical solution oriented approach is not of too much difference from the traditional approach due to connection between the numerical solutions and the input parameter setting. Compared with the traditional approach, it is certainly less favourable since the resulting relative efficiency is not of an immediate use to the user.

When making the comparison, from time to time, we felt the desire to use some statistical techniques. For example, for a given value of *TOL*, there may exist some significant statistical relationship between the accuracy of the numerical solution and the setting of *IPAR(2)* in COLNEW and COLSYS. However, we were not able to draw any conclusions on this basis due to the lack of a strategy of selecting test problems and the limited number of test problems we have.

We started out our comparison by taking a different approach, but our finding fully coincides with that of Pereyra and Russell [17]. Our first comment on the general performance of the codes is that all four codes are very sophisticated and are fully capable of dealing with smooth and moderately rough problems. As was evidenced by our comparison above, if one weighs all the issues equally and has no preference to any specific aspects of the codes, the relationships of the codes are clearly that they are complementary to each other rather than competitive. None of them outperforms the others with respect to all the important issues we have considered. Even though their robustness has not been fully explored, considering the degrees of difficulties of those artificial problems they can handle, we have to say that all the four codes are very robust. While still undergoing modifications from time to time, all of them are well written. Taking into consideration that there exists some kind of minimum difficulties one has to overcome when using a ODEBVP code (after all, one cannot expect an ODEBVP code to be as easy to use as a code for solving the linear systems), the codes are reasonably easy to use and quite user friendly in the sense that they provide various types of feedback to the user. Even when they fail, the reasons for failure are often clearly given. Based on our experience, the four codes perform very well regardless the size of the tolerance on smooth problems. On rough problems, the codes seem to have a better performance when the tolerance is set to some value of moderate size (around 1.D-6). However, if timing, storage or efficiency is not a concern compared to the quality of the solution, one may set the tolerance to a much smaller value (around 1.D-10).

As a brief summary to our comparison above, COLNEW and COLSYS are strong in terms of robustness, timing efficiency, the quality of numerical solutions and the classes of problems they can be directly applied to. COLNEW is often even stronger than COLSYS in terms of timing efficiency. Though there are some input parameters that might be difficult to use or not of too much use for ordinary users, our recommendation is to set them to default values. Unless the tolerance is big (say, >1.D-2), these parameters do not have much effect on the numerical solutions. These two codes, based on our experience, can usually solve the smooth problems and moderately rough problems to such a degree that the resulting accuracy of the numerical solution on the entire domain is approximately of the same size as the smallest machine number and they are both very reliable even on rough problems. Though they perform quite well, these two codes, in particular their speeds with respect to a certain input parameter setting, can be further improved. Compared with HAGRON, their potential (the rates of convergence) are not fully utilized because of their inefficient error estimators, in the sense that they often spend more CPU time to produce a solution that satisfies a certain accuracy requirement than they really need. As a result, they often seem to be slower than HAGRON when one makes a comparison using the traditional approach. It is possible to build a more accurate error estimator based on deferred correction method for these two codes. We believe it is important that their error estimators be improved so that their potential ability can be fully released.

HAGRON is strong in terms of timing efficiency, easy of use, robustness and the quality of the numerical solution. Due to the close ties between its underlying numerical method and the method behind COLSYS and COLNEW, the driver for HAGRON bears a lot of resemblance to that of COLNEW and COLSYS, but it has fewer input parameters that may affect its numerical solution, and we experienced much less variation in its solutions when we change the input parameter values as long as the tolerance is fixed. HAGRON can also solve smooth and moderately rough problems to the degree that the numerical solutions have a maximum error about the same size as the smallest machine number at its final mesh points. It is very reliable in general, but the problem that it may take a huge amount of computing time yet fail to solve a certain problem (see L4 for example) is somewhat disturbing. It would be nice if such a problem is resolved.

MUTS is strong in storage efficiency, ease of use and is fairly robust on smooth and moderately rough problems. It is able to deal with non-separated boundary condition directly. Its flexibility on the use of storage brings a unique storage efficiency to the code, but it does not have the ability of detecting the shape of the solution all by itself and it is not as reliable as the other three codes. Like HAGRON, it may take a huge amount of computing time yet fail to solve a certain problem. Its numerical solutions are generally very accurate but the accuracy depend heavily on the number of output points. Compared with the other three codes, it also seems to be slower. In order to be competitive with the other three (especially on rough problems), significant improvement concerning its robustness and speed has to be made. We also notice, however, that MUTS is based upon multiple shooting method which is not the best choice for dealing with problems of singular perturbation type. Users should keep this point in mind when solving problems of singular perturbation type.

Finally, we point out that our comparison is based on the eleven test problems and the conclusions may be changed if one is only concerned with a certain type of problem or a certain aspect of the codes. Our goal has been to provide some useful discussion and raise some questions on the comparison of ODEBVP codes, as well as to provide some useful information to the code user. We hope a reader can benefit from our discussion in these two respects.

REFERENCES

1: U. Ascher, J. Christiansen and R.D. Russell: *Collocation Software for Boundary-Value ODEs*—ACM Trans. on Math. Software. June, 1981.

2: U. Ascher, R.M.M. Mattheij and R.D. Russell: *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*—Prentice Hall, 1988.

3: G. Bader and U. Ascher: *A New Basis Implementation for a Mixed Order Boundary Value ODE Solver*—SIAM J. Stat. Comput. July, 1987.

4: J.R. Cash: *Numerical Integration of Non-Linear Two-Point Boundary-Value Problems Using Iterated Deferred Corrections (I)*—Comput. Math. Appl. 1986.

5: J.R. Cash: *Numerical Integration of Non-Linear Two-Point Boundary-Value Problems Using Iterated Deferred Corrections (II)*—SIAM J. Numer. Anal. 1988.

6: J.R. Cash and Margaret H. Wright: *A Deferred Correction Method for Nonlinear Two-Point Boundary Value Problems: Implementation and Numerical Evaluation*—Manuscript, 1990.

7: B. Ford, G.S. Hodgson and D.K. Sayers: *Evaluation of Numerical Software Intended for Many Machines-Is It Possible?*—in [8], pp. 317-330.

8: L.D. Fosdick: *Performance Evaluation of Numerical Software*—Proc. of IFIP TC 2.5, North Holland, Amsterdam, 1979.

9: E.D. Frind and G.F. Pinder: *A Collocation Finite Element Method for Potential Problems that Arise in irregular Domains*—Int. J. Num. Eng. 14, 1979.

10: W.M. Gentleman: *Discussion of General Aspects of Performance Evaluation* in [8], pp. 89-92.

11: E.N. Houstis, R.E. Lynch, T.S. Papatheodoru and J.R. Rice: *Evaluation of Numerical Methods for Elliptic Partial Differential Equations*—J. Comput. Phys. 27, pp. 323-350, 1978.

12: D. Kahaner: *Comparison of Numerical Quadrature Formulas*—[20], pp. 229-259.

13: F.A. Lootsma: *Performance Evaluation of Non-linear Programming Codes from the Viewpoint of the Decision Maker*—in [8], pp. 285-297.

14: J.N. Lyness: *Performance Profiles and Software Evaluation*—in [8], pp. 51-58.

15: R.M.M. Mattheij and G.W.M. Staarink: *An Efficient Algorithm for Solving General Linear Two-point BVP*—SIAM J. Sci. Stat. Comp. 5, 1984.

16: J.A. Nelder: *Experimental Design and Software Evaluation*—in [8], pp 309-316.

17: V. Pereyra and R.D. Russell: *Difficulties of Comparing Complex Mathematical Software: General Comments and the BVODE Case*—Acta Cient. Venezolana 33 (1982), 15-22.

18: P.M. Prenter and R.D. Russell: *Orthogonal Collocation for Elliptic partial differential Equations*—SIAM J. Numer. Anal. 13, 1976.

19: S. Pruess: *Interpolation Schemes for Collocation Solutions of Two Point Boundary Value Problems*—SIAM J. Sci. Stat. Comput. 7, 1986.

20: J.R. Rice, Editor: *Mathematical Software*—Academic Press 1971

21: R.D. Russell: *Global Codes for BVODES and Their Comparison*—Proc. of Workshop on Numerical Integration of Differential Equations, Springer Lecture Notes, 1980.

22: R.D. Russell and J.M. Varah: *A Comparison of Global Methods for Linear Two-point Boundary Value Problems*—Math. of Comp. 29, pp. 1-13, 1975.

23: L.F. Shampine: *Discussion on the Performance Evaluation in Ordinary Differential Equations*—in [8], pp. 215-217.

24: L.F. Shampine, H.A. Watts and S.M. Davenport: *Solving Nonstiff Ordinary differential Equations - the State of the Art*—SIAM Review 18, 1976.

25: Lloyd N. Trefethen: *A course in Finite Difference and Spectral Methods*—Manuscript, 1988.

26: A. Weiser, S.L. Eisenstat and M.H. Schultz: *On Solving Elliptic Equations to Moderate Accuracy*—SIAM J. Numer. Anal. 17, 99. 908-929, 1980.

Appendix (I) Testing and Test Problems

All testing involved in this thesis was done on Simon Fraser University MTSG (IBM-4300), using the Fortran 77 compiler and double precision. The smallest double precision machine number on SFU MTSG is approximately 2.0×10^{-16}.

Most of the eleven test problems we selected have a parameter that controls the difficulties of the problems. For these problems, we set the parameter to four different values to get four different degrees of difficulties, and then for each degree of difficult we vary the value of tolerance a few times (the values of tolerance we usually choose are 1.D-2, 1.D-4, 1.D-6 and 1.D-8) to collect the information shown in the tables in Appendix III. If we distinguish the same test problems with different degrees of difficulties, then the total number of test problems we have is 35. Among the test problems, L7 is only used in our discussion concerning robustness. No run concerning this test problem is recorded.

As the major reason for abnormal exits from COLNEW, COLSYS and HAGRON is that the user supplied storage is insufficient, it is important that when we speak about an abnormal exit of this kind, the corresponding amount of supplied storage is available for reference. During our testing, we provided all the three codes with a fixed amount of workspace (represented by the sum of IPAR(5) and IPAR(6)) which accounts for more than 99% of the total storage supplied to the codes. The sum of IPAR(5) and IPAR(6) is fixed to be 200,000 throughout our testing. This fixed sum translates to a limit on the maximum number of grid points that each code can use. When setting IPAR(3) in COLNEW and COLSYS to its default value, for the two dimensional problems, the limit on COLNEW is 2856, the limit on COLSYS is 3124. For three dimensional problem, this limit is 1297 for COLNEW and 1427 for COLSYS. For a five dimensional problem, this limit is 583 for COLNEW and 618 for COLSYS. For HAGRON, this limit is 4857 for two dimensional problems, 2766 for three dimensional problems and 1264 for five dimensional problems. Thus when we speak about an abnormal exit due to insufficient storage from one of the three codes, the storage refers to the sum described above.

The CPU time is recorded by using a MTSG system subroutine TIME that can provide a measurement of CPU time in milliseconds. The CPU times in the tables in Appendix III are all in milliseconds. When we repeat runs for a code with a fixed set of input parameters on a test problem, we observed that the CPU time for each run may vary quite a bit. The maximum variation we observed is about 5%.

The maximum errors recorded for all the four codes are the maximum absolute errors between the numerical solutions from the codes and the exact solutions. For HAGRON and MUTS, this maximum error is based on the absolute errors calculated at all the final mesh points. For COLNEW and COLSYS, we recorded both the maximum error at their final mesh points (first type of maximum error) and the maximum error at 30 equidistant points in the domain (second type of maximum error). Due to the fact that the form of the numerical solution is discounted when we evaluate the relative efficiency of the codes, our focus on the maximum error from COLNEW and COLSYS has been the first type of maximum error. In our discussion in Chapter Four, unless specified, the maximum error involved for these two codes is the first type of maximum error.

When we evaluate the relative efficiency of the codes, we have to compare the size of the errors. If the ratio of two errors is between 0.1 and 10, we consider that they are of the same size.

We did not intend to fully explore the potentials of the four codes due to various expenses. One may use a good initial guess to save some CPU time or even to compromise the problem of insufficient storage. One may also consider using continuation to fully utilize the partially converged results from the previous runs. But during our testing, we did not go that far, rather we always set the initial guess to zero if an initial guess is needed or set them to one if zero is obviously not the correct guess. Having noticed that the accuracy for MUTS is often affected by the total number of output points and the accuracy for COLNEW and COLSYS may be affected by the setting of IPAR(2), when we run MUTS we always vary the output points among the following set of values {20,50,100,200,300} and record the run corresponding to the smallest number in the set that results in an accuracy that is of the same size as or smaller than the one of the related tolerance (more specifically, ER(2) for linear problems and ER(1) for nonlinear problems). When we run COLNEW and COLSYS, we usually vary IPAR(2) a bit and record the run with the best accuracy. This is not a big trouble since the legal values for IPAR(2) is very limited. We would like to point out here that the runs we recorded are not necessarily the favourable runs for the codes since as the the accuracy increases, the CPU time and the storage requirement usually increase as well.

When using COLNEW, COLSYS and HAGRON on the ten test problems, for each run the tolerances are all set to the values shown in the first columns of the tables in appendix III. When using MUTS on the linear problems, ER(1) is set to be ER(2)/100 and ER(2) are the values that are recorded in the first columns of the table in Appendix III. When using MUTS on nonlinear problems, ER(1) is recorded in the first column of the tables in Appendix III and the corresponding ER(2) is set to be ER(1)×100. The resulting test problem dependent comparison in Chapter Four is totally based on the testing described above. Should the method of testing be different, the observations may not be the same.

The following is the set of eleven test problems we collected. They are all artificial problems where the exact solutions are known. Except for L5, L6 and the four nonlinear test problems that we added ourselves, the test problems are from [2], [5], [6] and [17].

Linear Test Problems

1: Equations(BL):

$$\varepsilon u_1' = -u_1 + u_2 + q_1(x) \qquad 0<x<1 \qquad (L1.a1)$$

$$u_2^{(4)} = u_1 + u_2 + q_2(x) \qquad (L1.a2)$$

Boundary condition:

$$u_1(0) = 2, \; u_2(0) = u_2(1) = u_2''(0) = u_2''(1) = 0 \qquad (L1.b)$$

Exact solution:

$$y_1 = exp(\frac{-x}{\varepsilon}) + cos(\pi x) \qquad (L1.c1)$$

$$y_2 = \sin(\pi x) \qquad (L1.c2)$$

$q_1(x)$ and $q_2(x)$ are functions such that y_1 and y_2 satisfy L1.a1&a2.

2: *Equation(TPT):*

$$y'' = \frac{-3\varepsilon y}{(\varepsilon + t^2)^2} \qquad -0.1<t<0.1 \qquad (L2.a)$$

Boundary Condition:

$$y(0.1) = y(-0.1) = \frac{0.1}{(\varepsilon + 0.01)^{0.5}} \qquad (L2.b)$$

Exact Solution:

$$y(t) = t(\varepsilon + t^2)^{-0.5} \qquad (L2.c)$$

3: *Equation(OSC):*

$$y'' = a\left(\frac{-\pi^2 y}{4\varepsilon^2}\right) \qquad 0<x<1 \qquad (L3.a)$$

where ε^{-1} is an odd integer.

Boundary Condition:

$$y(0) = 0,\ y(1) = a\sin\left(\frac{\pi}{2\varepsilon}\right) \qquad (L3.b)$$

Exact Solution:

$$y = a\sin\left(\frac{\pi x}{2\varepsilon}\right) \qquad (L3.c)$$

4: *Equation(SPK):*

$$y'' = -\frac{(2\varepsilon + 6(0.5 - t)^2)\,y}{(\varepsilon + (0.5 - t)^2)^2} \qquad 0<t<1 \qquad (L4.a)$$

Boundary Condition:

$$y(0) = y(1) = \frac{1}{\varepsilon + 0.25} \qquad (L4.b)$$

Exact Solution:

$$y = \frac{1}{\varepsilon + (0.5 - t)^2} \qquad (L4.c)$$

5: *Equation(BL):*

$$y'' = \frac{2}{(x+\varepsilon)^3} + \frac{2}{(x - \varepsilon - 1)^3} \qquad 0<x<1 \qquad (L5.a)$$

Boundary Condition:

$$y(0) = \frac{1}{\varepsilon} - \frac{1}{\varepsilon + 1}, \quad y(1) = \frac{1}{\varepsilon + 1} - \frac{1}{\varepsilon} \qquad (L5.b)$$

Exact Solution:

$$y = \frac{1}{x+\varepsilon} + \frac{1}{x - \varepsilon - 1} \qquad (L5.c)$$

6: *Equation(SM):*

$$y_1' = y_2 - y_3 + y_4 \qquad a<t<b \qquad (L6.a1)$$

$$y_2' = -y_2 + y_4 \qquad (L6.a2)$$

$$y_3' = y_3 - y_4 \qquad (L6.a3)$$

$$y_4' = 2\,y_4 \qquad (L6.a4)$$

Boundary Conditon:

$$y_1(b) = e^{-b} + e^{b} + \frac{7}{6}e^{2b} \qquad (L6.b1)$$

$$y_2(a) = -e^{-a} + \frac{e^{2a}}{3} \qquad (L6.b2)$$

$$y_3(a) = -e^{a} - e^{2a} \qquad (L6.b3)$$

$$y_4(b) = e^{2b} \qquad (L6.b4)$$

Exact Solution:

$$y_1 = e^{-t} + e^{t} + \frac{7}{6}e^{2t} \qquad (L6.c1)$$

$$y_2 = -e^{-t} + \frac{e^{2t}}{3} \qquad (L6.c2)$$

$$y_3 = -e^{t} - e^{2t} \qquad (L6.c3)$$

$$y_4 = e^{2t} \qquad (L6.c4)$$

7: *Equation(SM):*

$$u'' = -\frac{1}{x}u' + \left(\frac{8}{8-x^2}\right)^2 \qquad 0<x<1 \qquad (L7.a)$$

Boundary Condition:

$$u'(0) = u(1) = 0 \qquad (L7.b)$$

Exact Solution:

$$u(x) = 2\ln\left(\frac{7}{8-x^2}\right) \qquad (L7.c)$$

Nonlinear Test Problems

1: *Equation(SM):*

$$y_1' = -\frac{2}{y_2^2} \qquad 0<t<1 \qquad (N1.a1)$$

$$y_2' = y_2^2 - \frac{1}{y_1} + e^t \qquad (N1.a2)$$

Boundary Condition:

$$y_1(0) = 1,\ y_2(1) = e^1 \qquad (N1.b)$$

Exact Solution:

$$y_1 = e^{-2t},\ y_2 = e^t \qquad (N1.c)$$

2: *Equation(OSC):*

$$y_1' = 3\, r\, y_2^2 \cos(rx) \qquad 0<x<1 \qquad \text{(N2.a1)}$$

$$y_2' = -\, r \cos(rx) \qquad \text{(N2.a2)}$$

Boundary Condition:

$$y_1(0) = 0,\ \ y_1(1) = -\sin(r) \qquad \text{(N2.b)}$$

Exact Solution:

$$y_1 = \sin^3(rx),\ \ y_2 = -\sin(rx) \qquad \text{(N2.c)}$$

3: *Equation(BL):*

$$y_1 = 2\, x \qquad 0<x<1 \qquad \text{(N3.a1)}$$

$$y_2'' = -\frac{2y_2}{\varepsilon} + \frac{4y_1y_2}{\varepsilon^2} \qquad \text{(N3.a2)}$$

Boundary Condition:

$$y_1(0) = 0,\ \ y_2(0) = 1,\ \ y_2(1) = \exp(-\frac{1}{\varepsilon}) \qquad \text{(N3.b)}$$

Exact Solution:

$$y_1 = x^2,\ \ y_2 = \exp(-\frac{x^2}{\varepsilon}) \qquad \text{(N3.c)}$$

4: *Equation(SM):*

$$y_1' = 2(y_3 - y_2) \qquad 0<x<1 \qquad \text{(N4.a1)}$$

$$y_2' = y_4 + y_5 - x - 2\sin(x) \qquad \text{(N4.a2)}$$

$$y_3' = y_2 + 1 \qquad \text{(N4.a3)}$$

$$y_4' = \cos(\sqrt{y_1}) + 1 \qquad \text{(N4.a4)}$$

$$y_5'' = -\, y_5 + 2y_2 \qquad \text{(N4.a5)}$$

Boundary Condition:

$$y_1(0) = y_4(0) = 0 \tag{N4.b1}$$

$$y_2(1) = e,\ y_3(1) = e + 1,\ y_5(1) = \sin(1) + e \tag{N4.b2}$$

Exact Solution:

$$y_1 = x^2 \tag{N4.c1}$$

$$y_2 = e^x \tag{N4.c2}$$

$$y_3 = e^x + x \tag{N4.c3}$$

$$y_4 = \sin(x) + x \tag{N4.c4}$$

$$y_5 = \sin(x) + e^x \tag{N4.c5}$$

Appendix (II) The Graphs

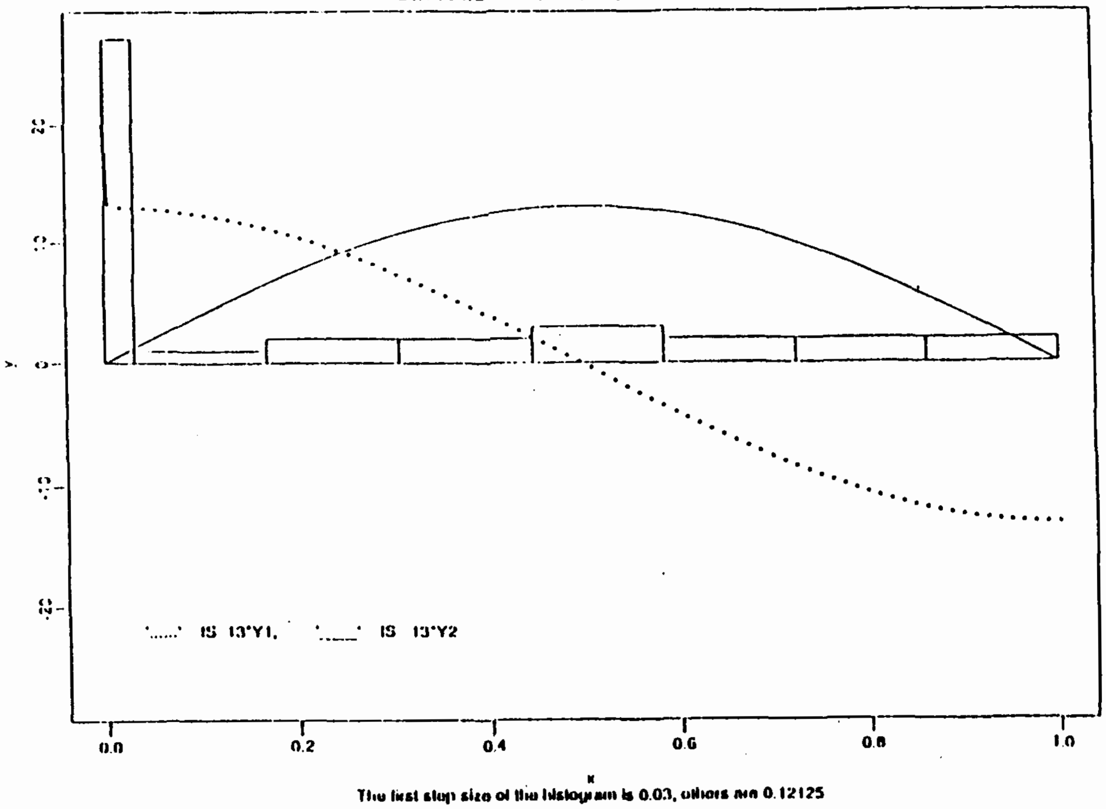
As a reference for our discussion in Chapter Four, we include here the graphs we produced for test problems L1, L4, L5 and some of the graphs for N3.

In order to support our discussion in Chapter Four, we produced two types of graphs. The first type are those used to show the distributions of the final mesh points versus the shapes of the exact solutions. The second type are those that are used to show the graphs of the absolute errors versus the exact solutions. We were hoping that these two types of graphs can guide us to some patterns concerning the behaviours of the distribution of the final mesh points or the locations of the maximum absolute errors at the mesh points. The detailed discussion about our finding is in Chapter Four.

To visualize the distribution of the final mesh points, we used the histogram of these points. A histogram is a bar chart where the height of a bar stands for the number of data (in our case, the number of points in the final mesh) that fall into the interval on which the bar stands. Thus if two bars have the same widths and heights, that means the same number of data points fall into the two intervals on which the two bars stand and therefore the two intervals have the same average density of data points. With a fixed width, the higher the bar, the higher the average density of data points in the interval. To visualize the pattern of the errors at the mesh points, we simply graph the error functions (the difference between the numerical solution and the exact solution at the mesh points) and then compare it with the exact solutions. Often, the magnitude of the heights of the bars and the magnitude of the error functions are too far apart from that of the exact solutions. In these cases, we made some adjustments so that everything in our graph is of the same magnitude. Otherwise, the error functions would look like a straight line.

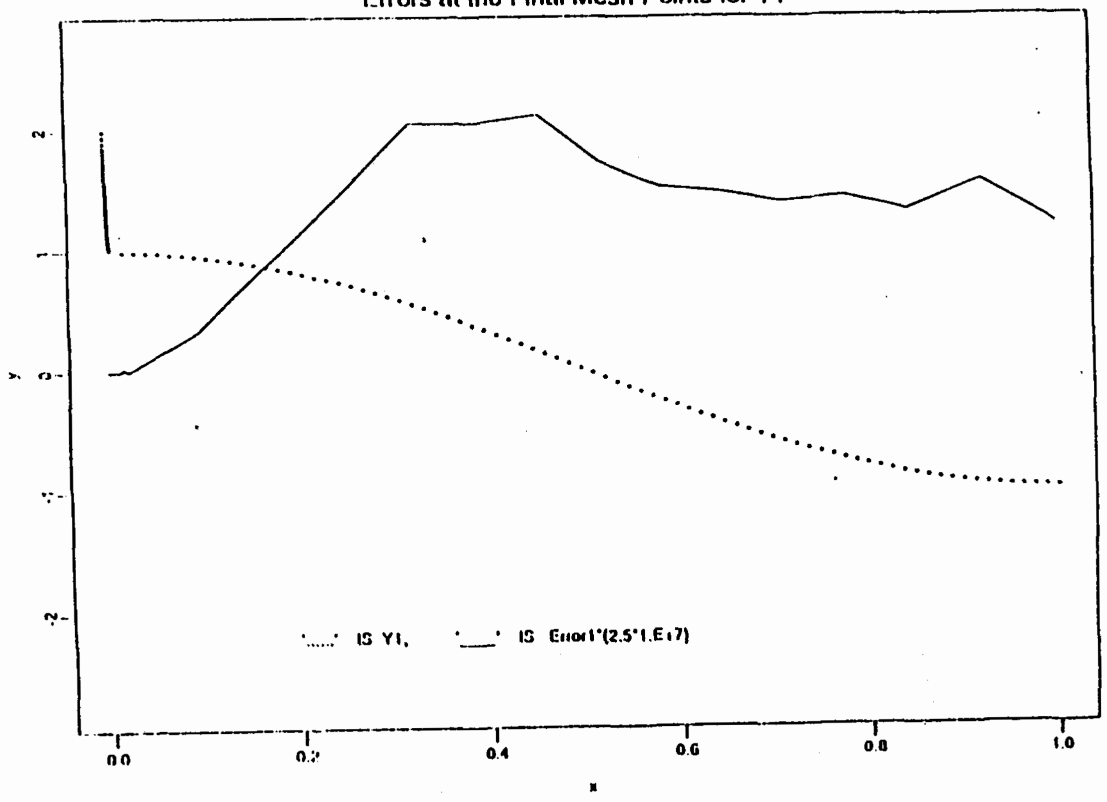
On each graph, most of the related important parameters are indicated. All the graphs are annotated. In the annotations, Y, Y1 or Y2 are used to denote the exact solutions and ERROR, ERROR1, ERROR2 are used to denote the error functions. The width of the bars in the histogram are indicated in the graph. The adjustment we made is also indicated in the graph. For example, 13*Y1 in GP1 means that the first solution for the problem shown on the graph is the first exact solution multiplied by 13. We used 'the first solution' here and hope that this will not cause too much confusion.

COLNEW.L1 (EPS = 1.D-3, TOL = 1.D-6)
Distribution of the Final Mesh Points

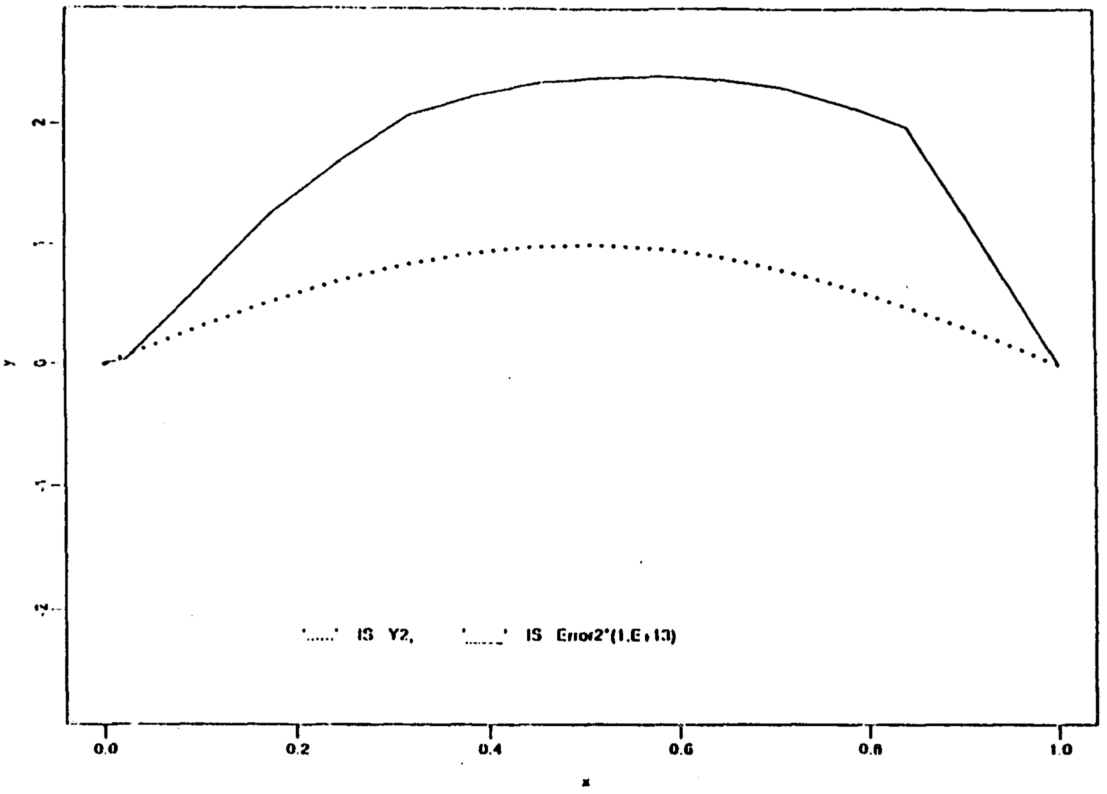


The first step size of the histogram is 0.03, others are 0.12125

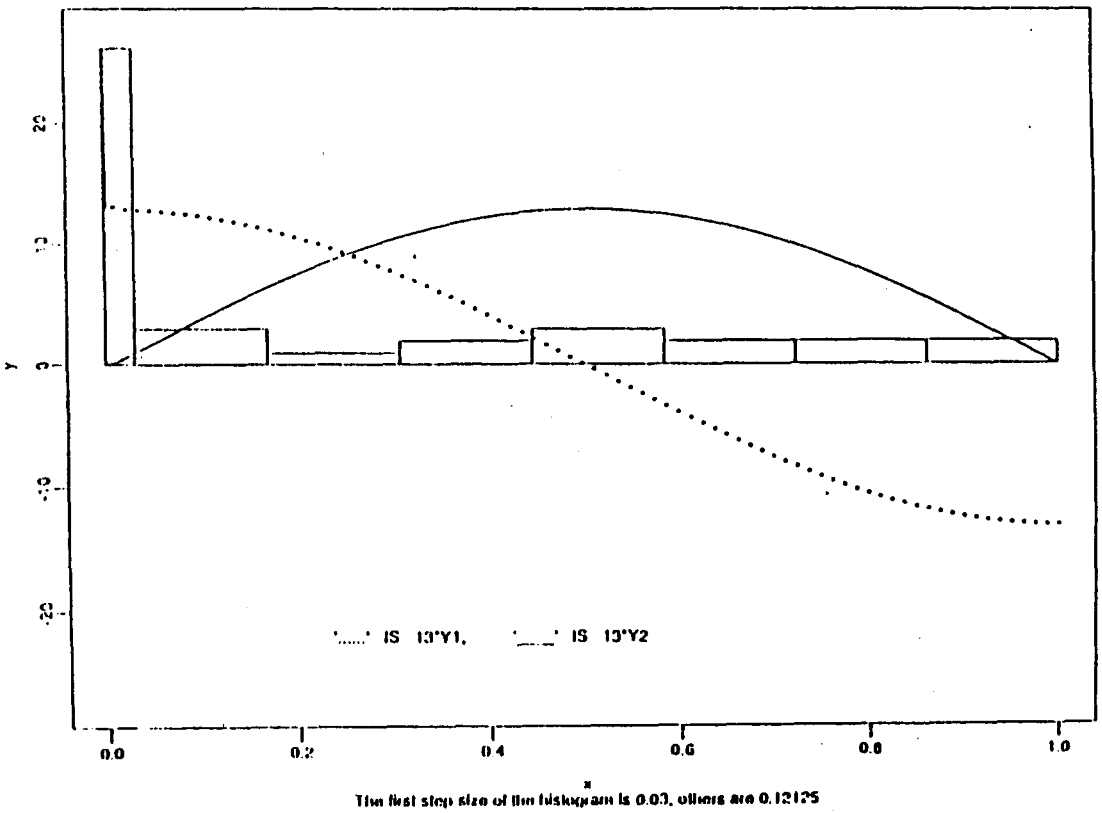
COLNEW.L1 (EPS = 1.D-3, TOL = 1.D-6)
Errors at the Final Mesh Points for Y1



COLNEW.L1 (EPS = 1.D-3, TOL = 1.D-6)
Errors at the Final Mesh Points for Y2

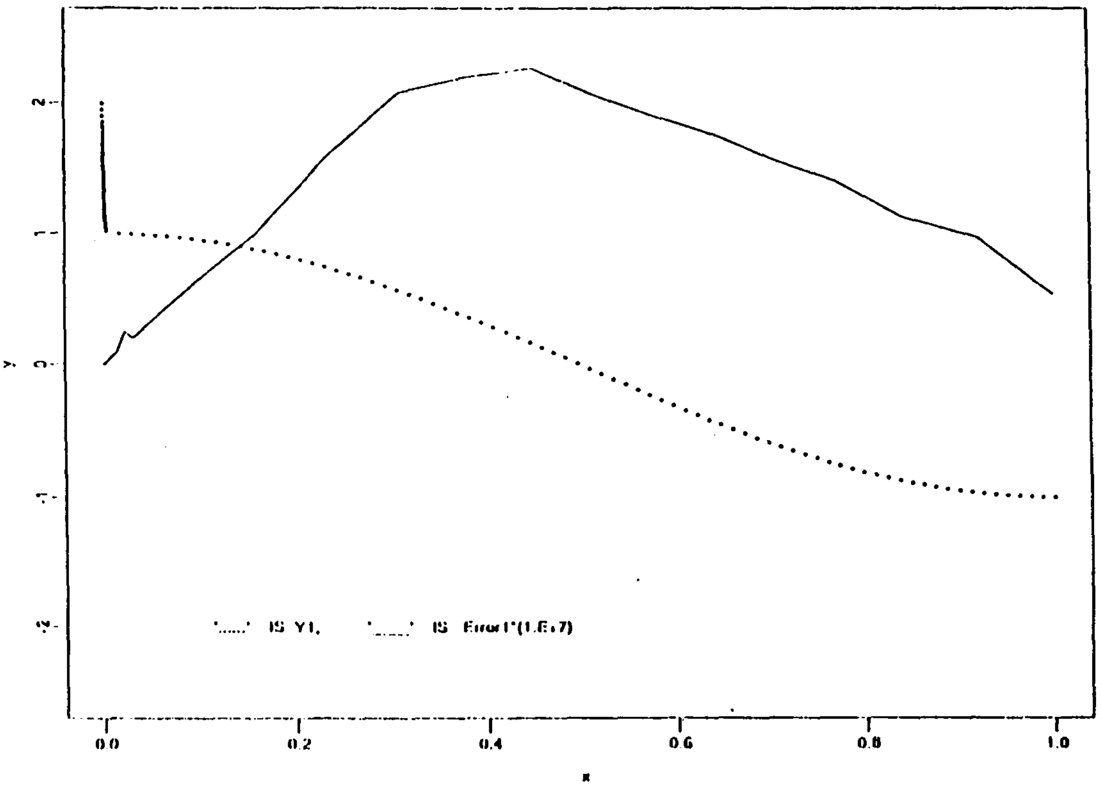


COLSYS.L1 (EPS = 1.D-3, TOL = 1.D-6)
Distribution of the Final Mesh Points

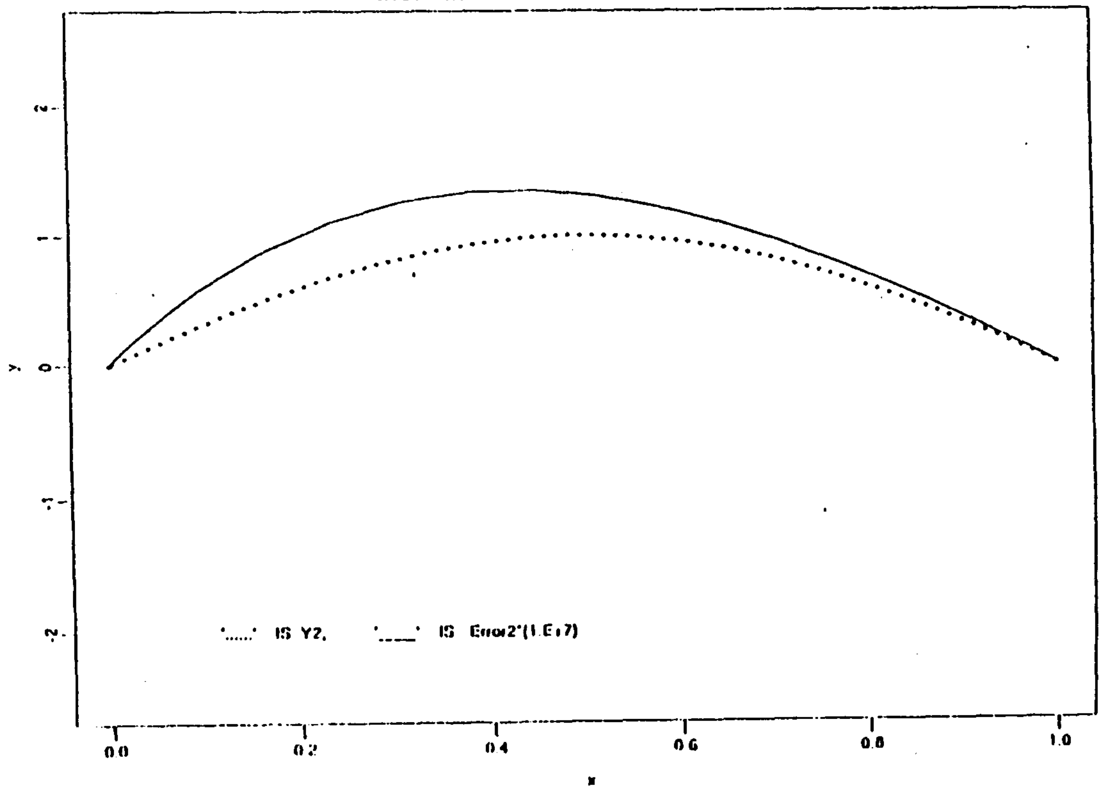


The first step size of the histogram is 0.03, others are 0.12125

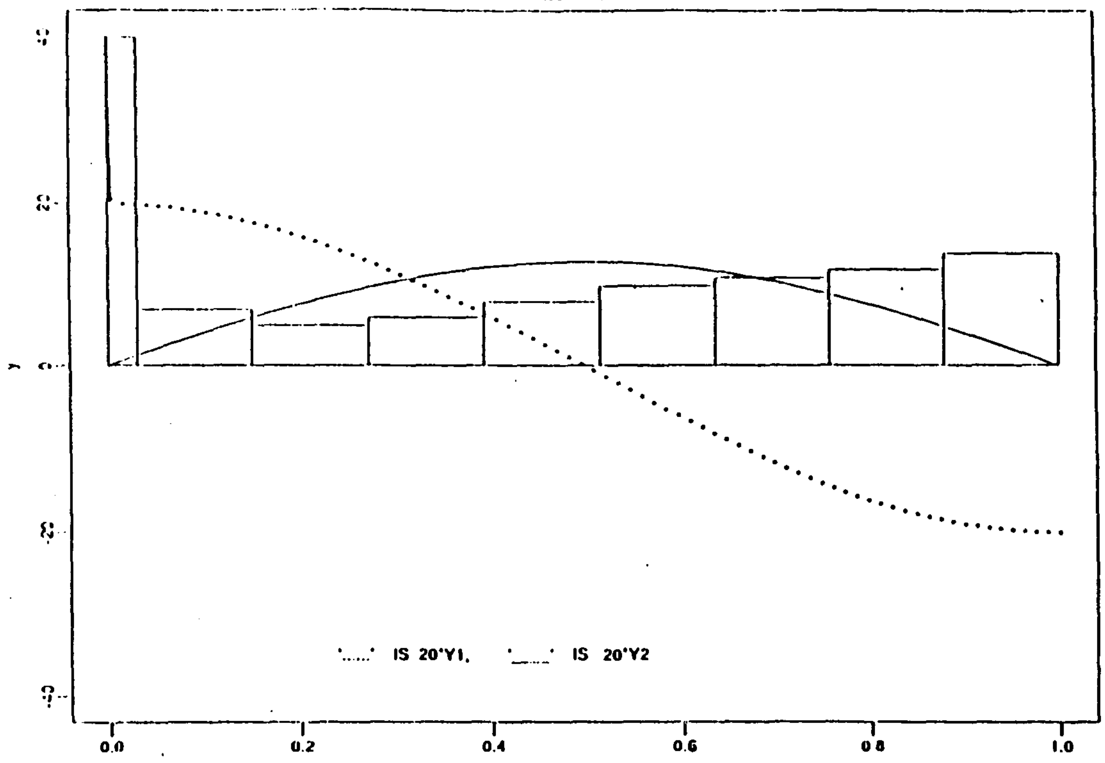
COLSYS.L1 (EPS = 1.D-3, TOL = 1.D-6)
Errors at the Final Mesh Points for Y1



COLSYS.L1 (EPS = 1.D-3, TOL = 1.D-6)
Errors at the Final Mesh Points for Y2

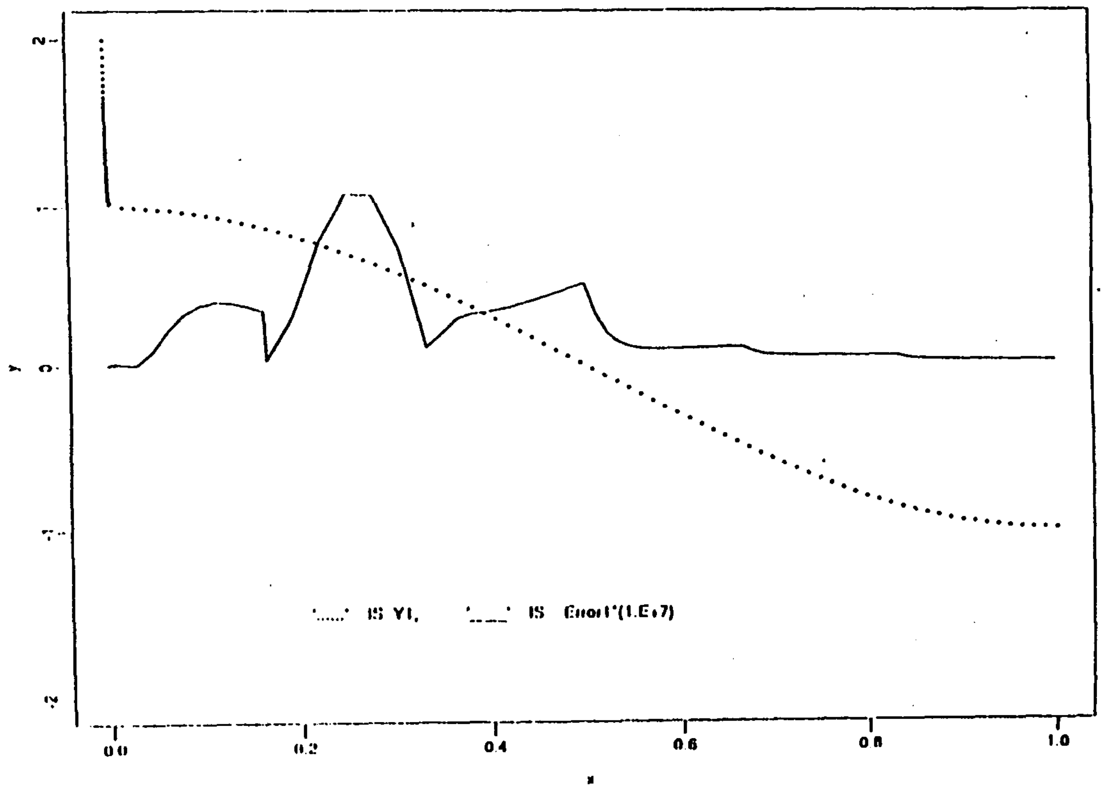


HAGRON.L1 (EPS = 1.D-3, TOL = 1.D-6)
Distribution of the Final Mesh Points

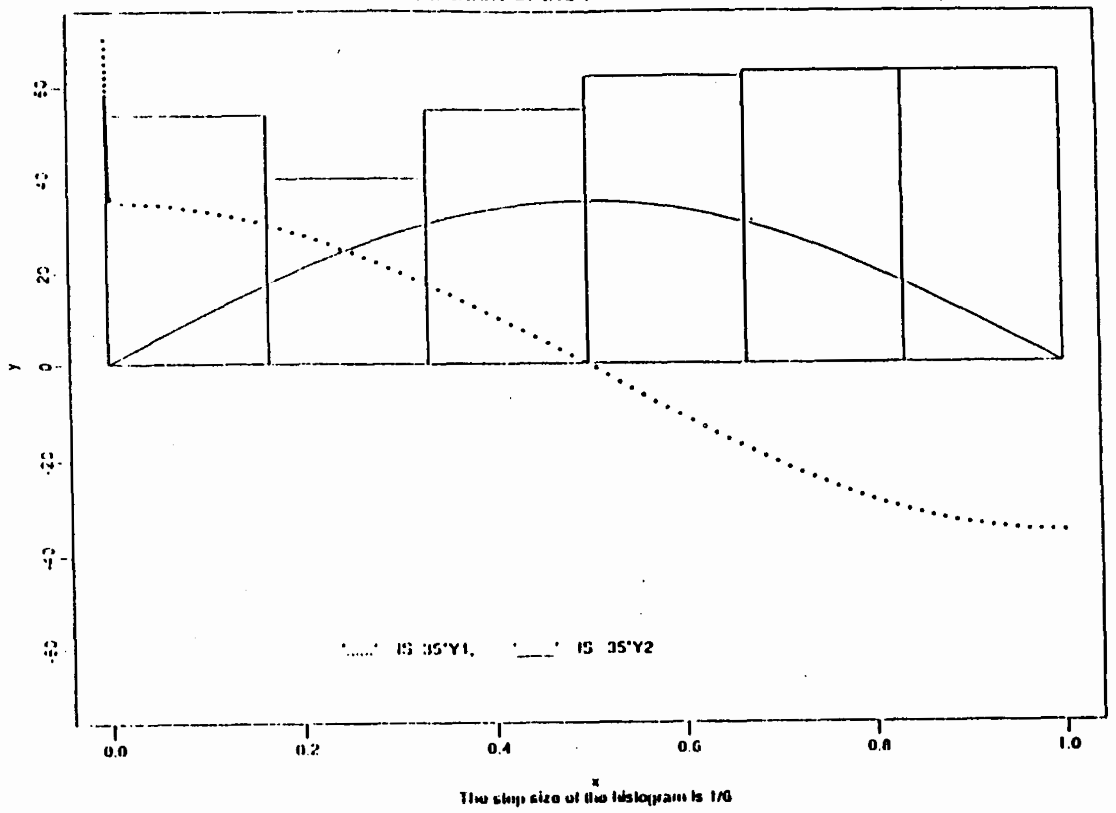


The first step size of the histogram is 0.03, others are 0.12125

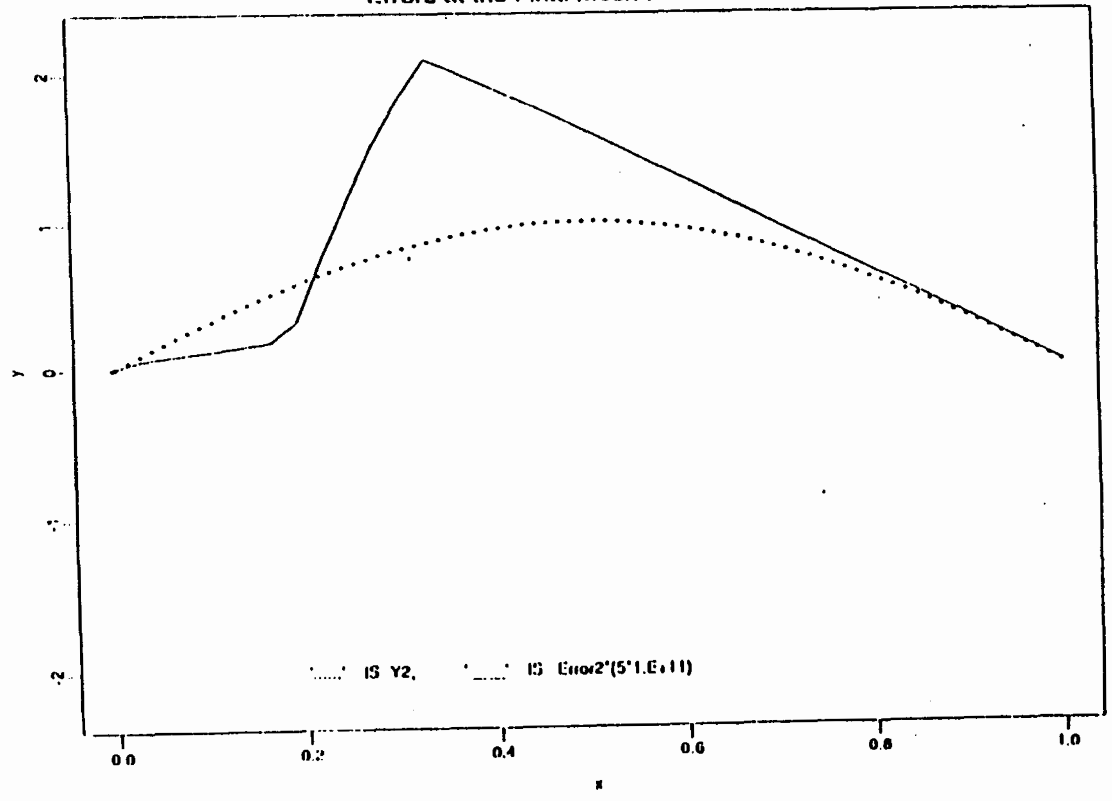
HAGRON.L1 (EPS = 1.D-3, TOL = 1.D-6)
Errors at the Final Mesh Points for Y1



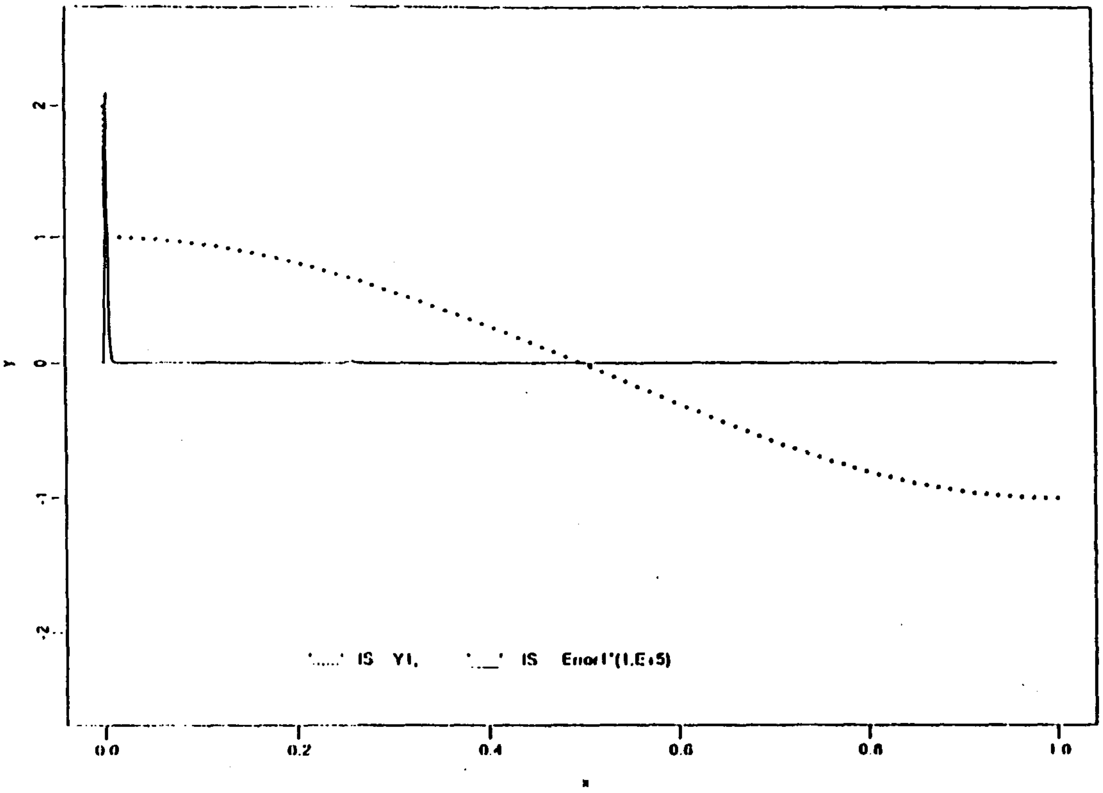
MUTS.L1 (EPS=1.D-3, TOL1=1.D-6, TOL2=1.D-4)
Distribution of the Final Mesh Points



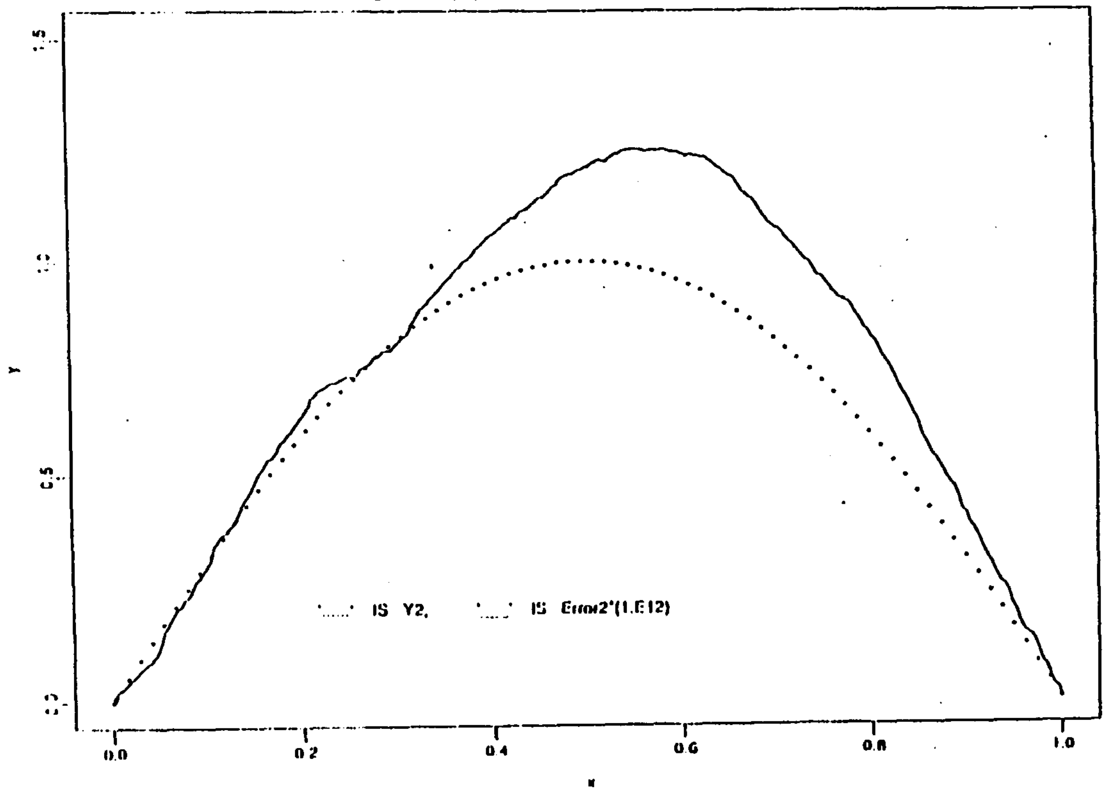
HAGRON.L1 (EPS = 1.D-3, TOL = 1.D-6)
Errors at the Final Mesh Points for Y2



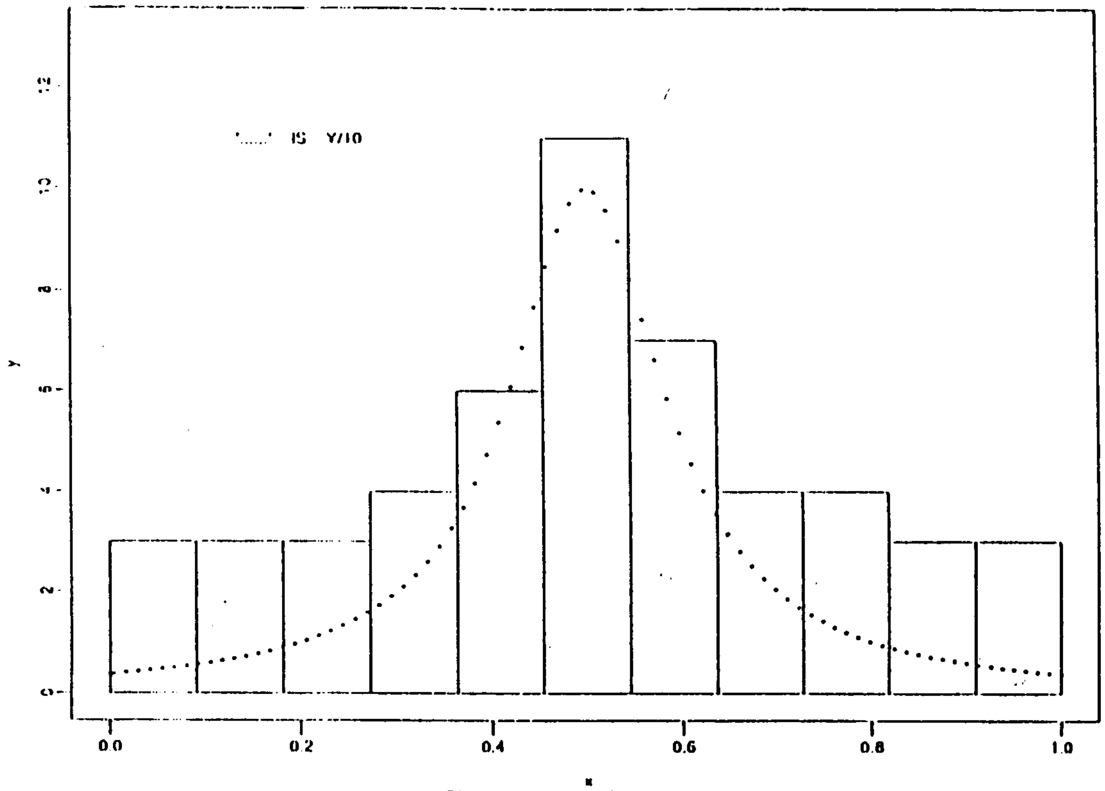
MUTS.L1 (EPS=1.D-3, TOL1=1.D-6, TOL2=1.D-4)
Errors at the Final Mesh Points for Y1



MUTS.L1 (EPS=1.D-3, TOL1=1.D-6, TOL2=1.D-4)
Errors at the Final Mesh Points for Y2

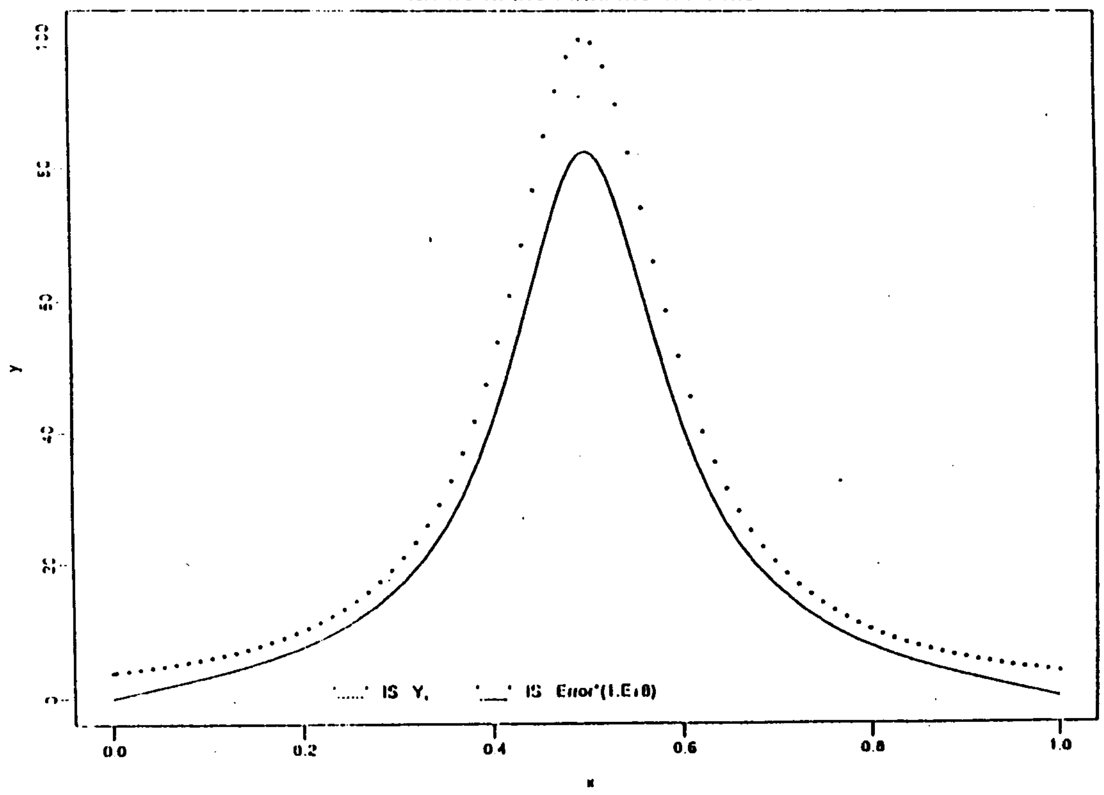


COLNEW.L4 (EPS = 1.D-2, TOL = 1.D-6)
Distribution of the Final Mesh Points

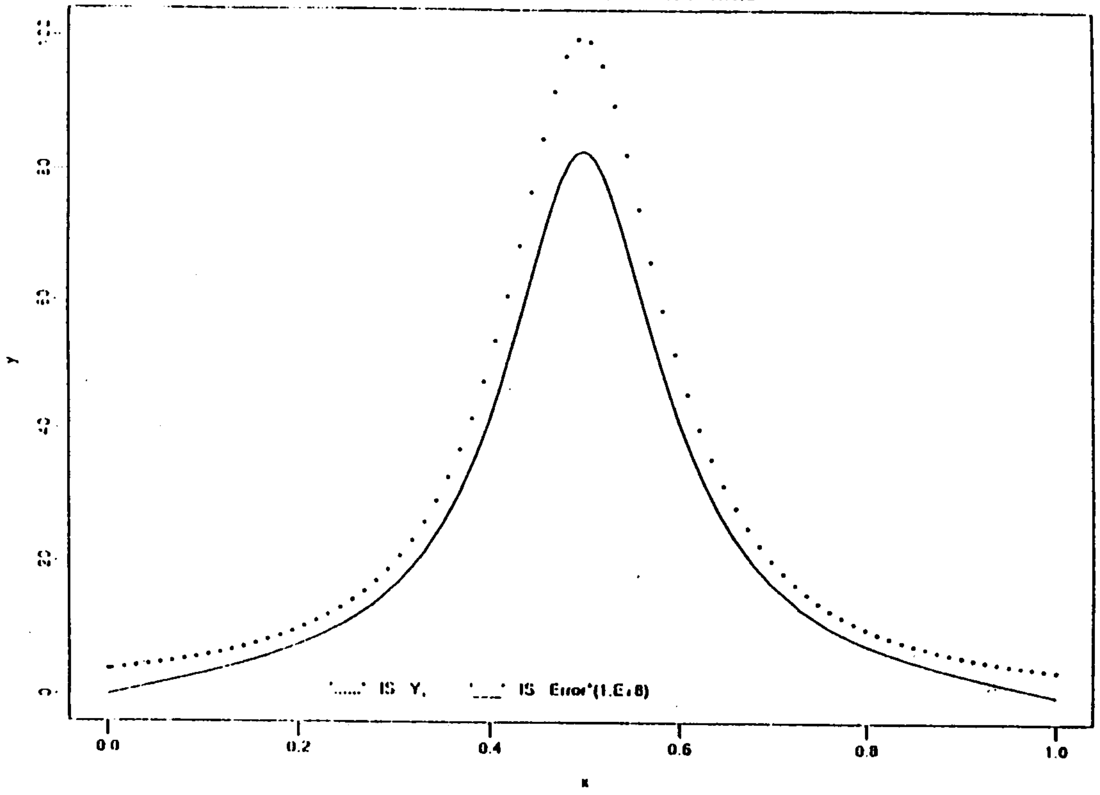


The step size of the histogram is 1/11

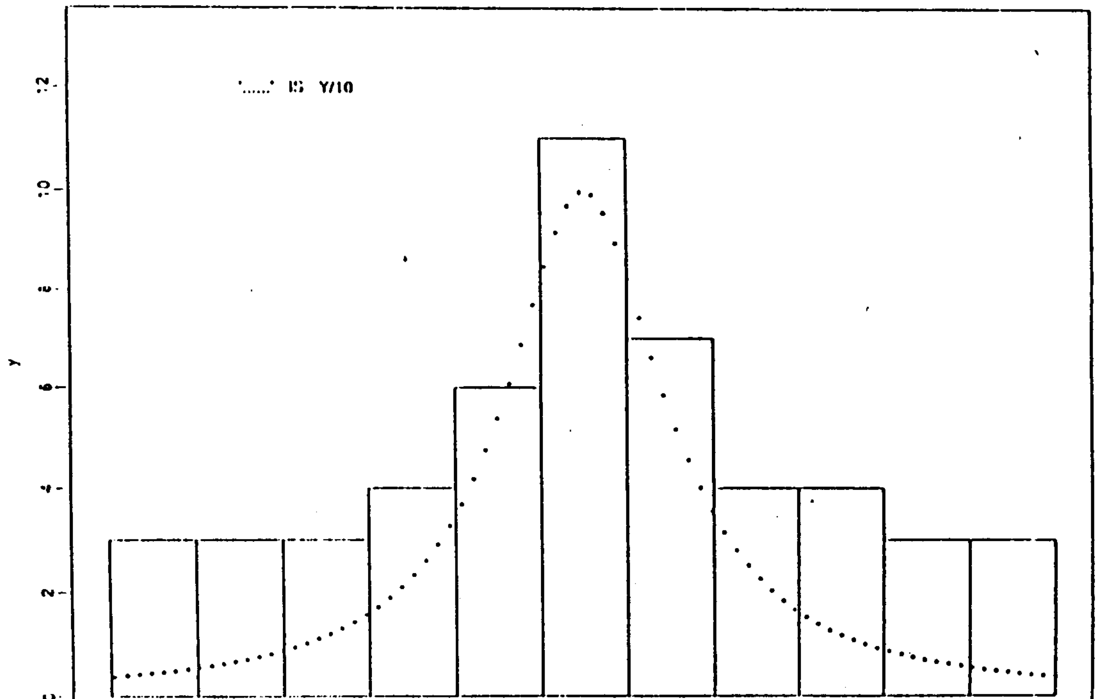
COLNEW.L4 (EPS = 1.D-2, TOL = 1.D-6)
Errors at the Final Mesh Points



COLSYS.L4 (EPS = 1.D-2, TOL = 1.D-6)
Errors at the Final Mesh Points

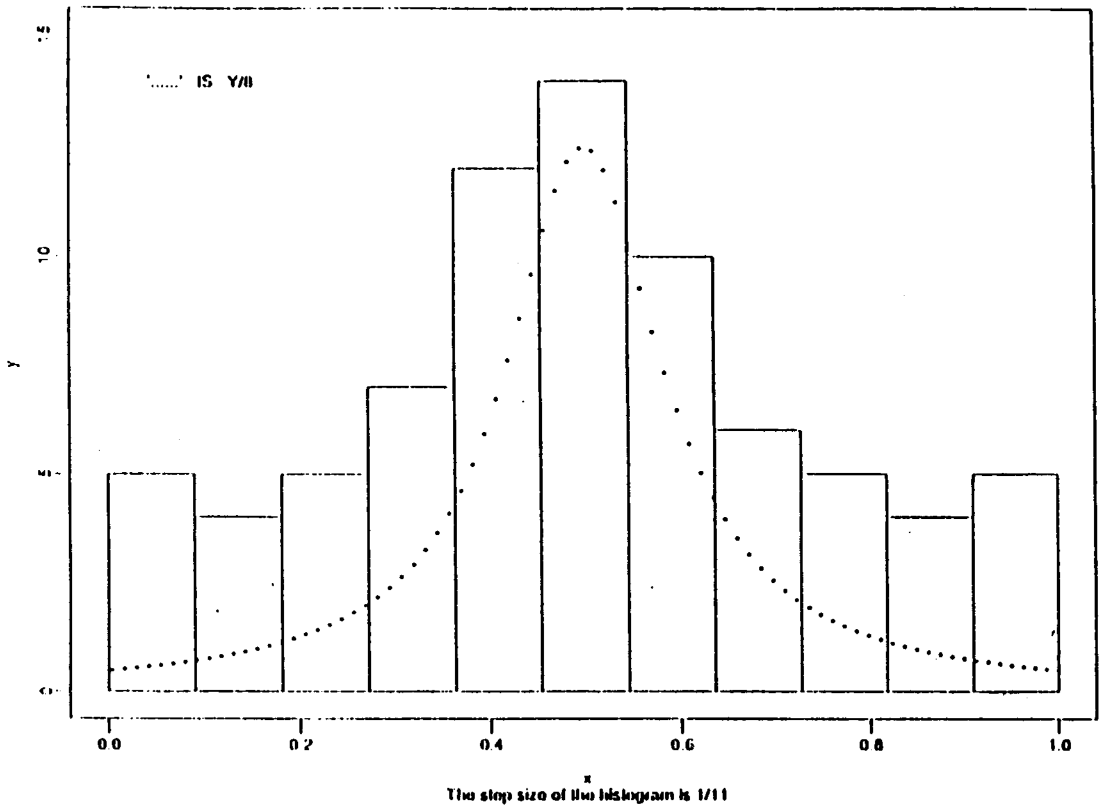


COLSYS.L4 (EPS = 1.D-2, TOL = 1.D-6)
Distribution of the Final Mesh Points

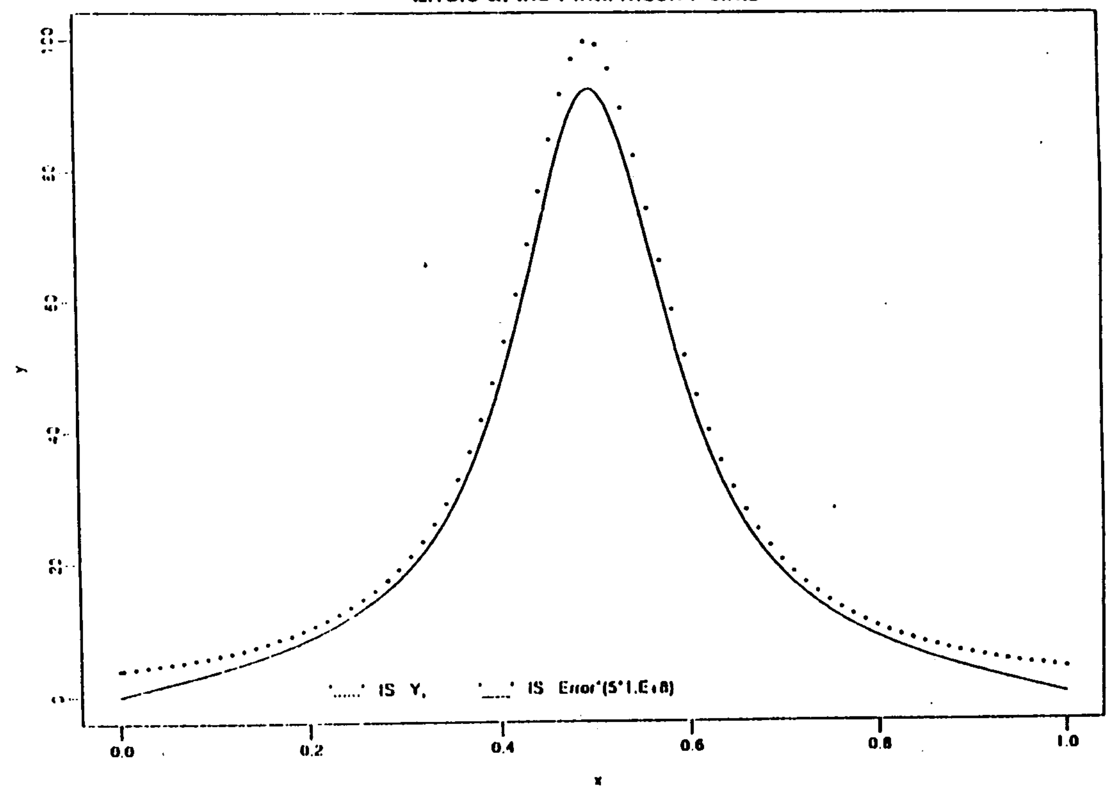


The step size of the histogram is 1/11

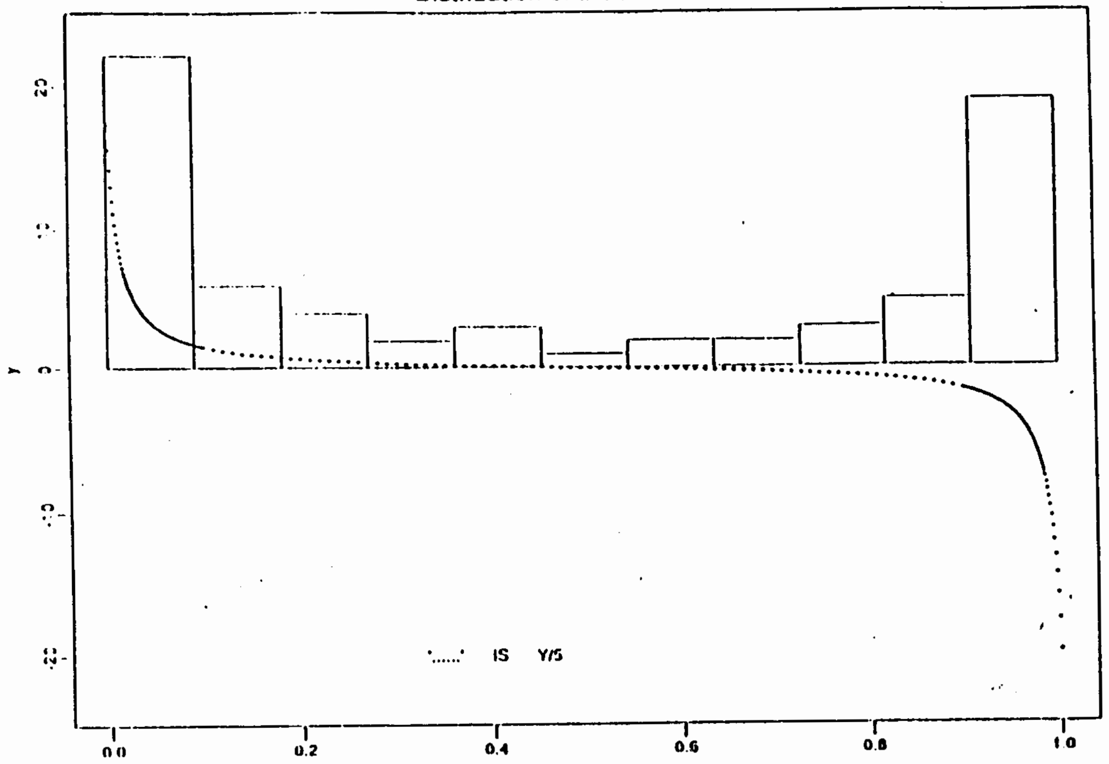
HAGRON.L4 (EPS = 1.D-2, TOL = 1.D-6)
Distribution of the Final Mesh Points



HAGRON.L4 (EPS = 1.D-2, TOL = 1.D-6)
Errors at the Final Mesh Points

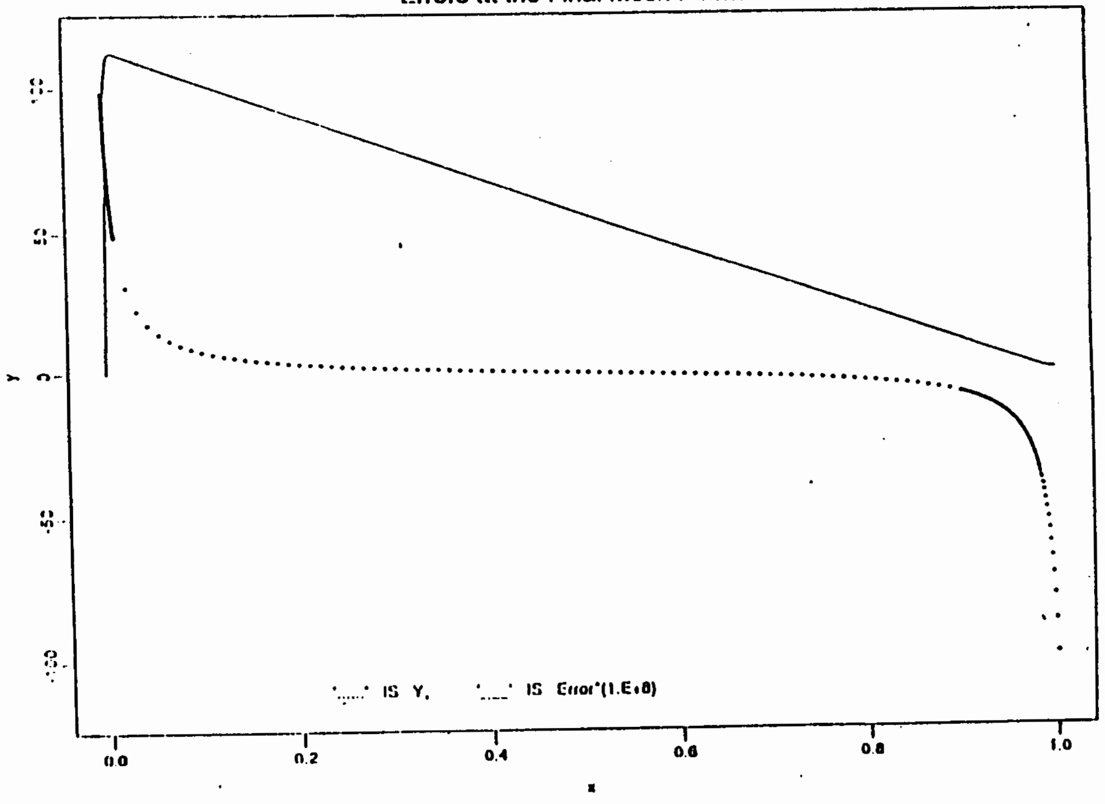


COLNEW.L5 (EPS = 1.D-2, TOL = 1.D-4)
Distribution of the Final Mesh Points

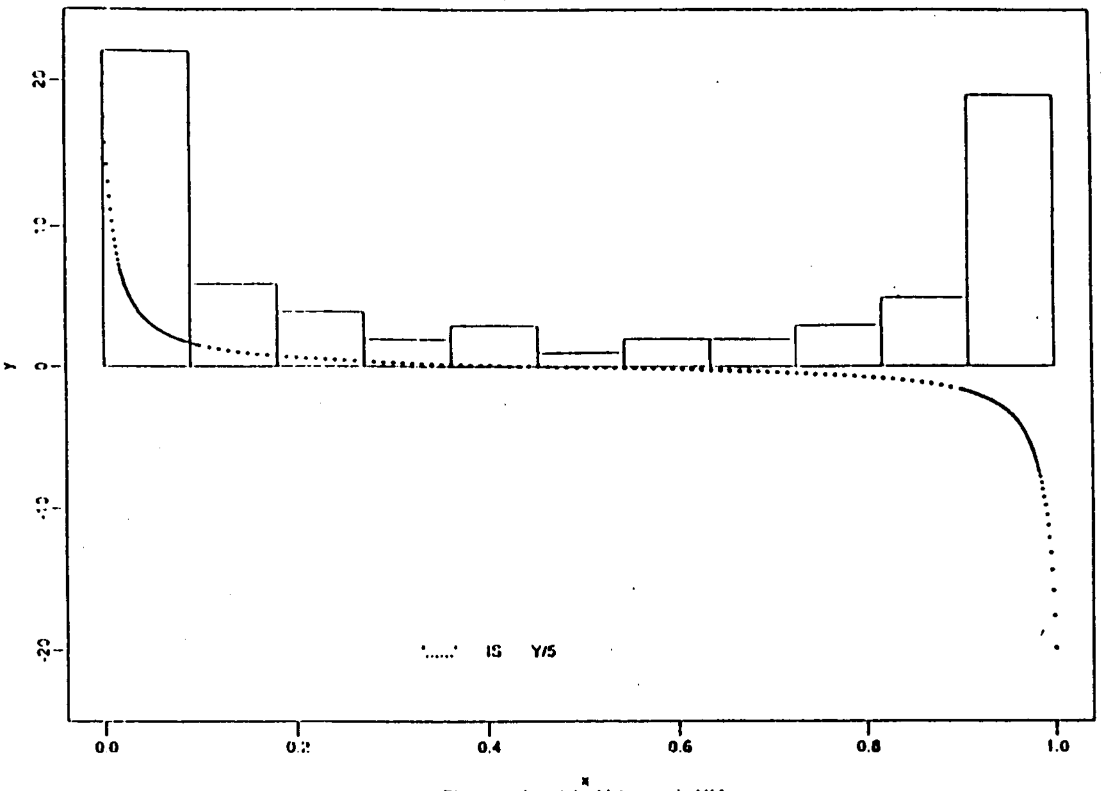


The step size of the histogram is 1/11

COLNEW.L5 (EPS = 1.D-2, TOL = 1.D-4)
Errors at the Final Mesh Points

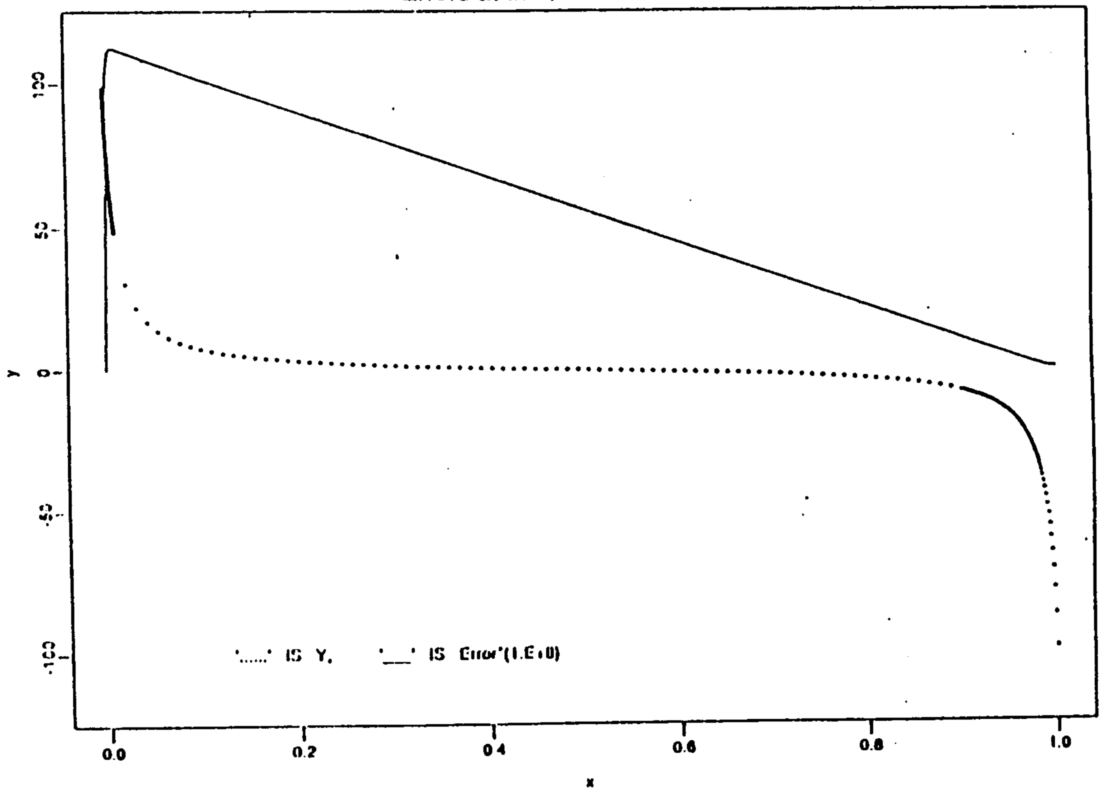


COLNEW.L5 (EPS = 1.D-2, TOL = 1.D-4)
Distribution of the Final Mesh Points

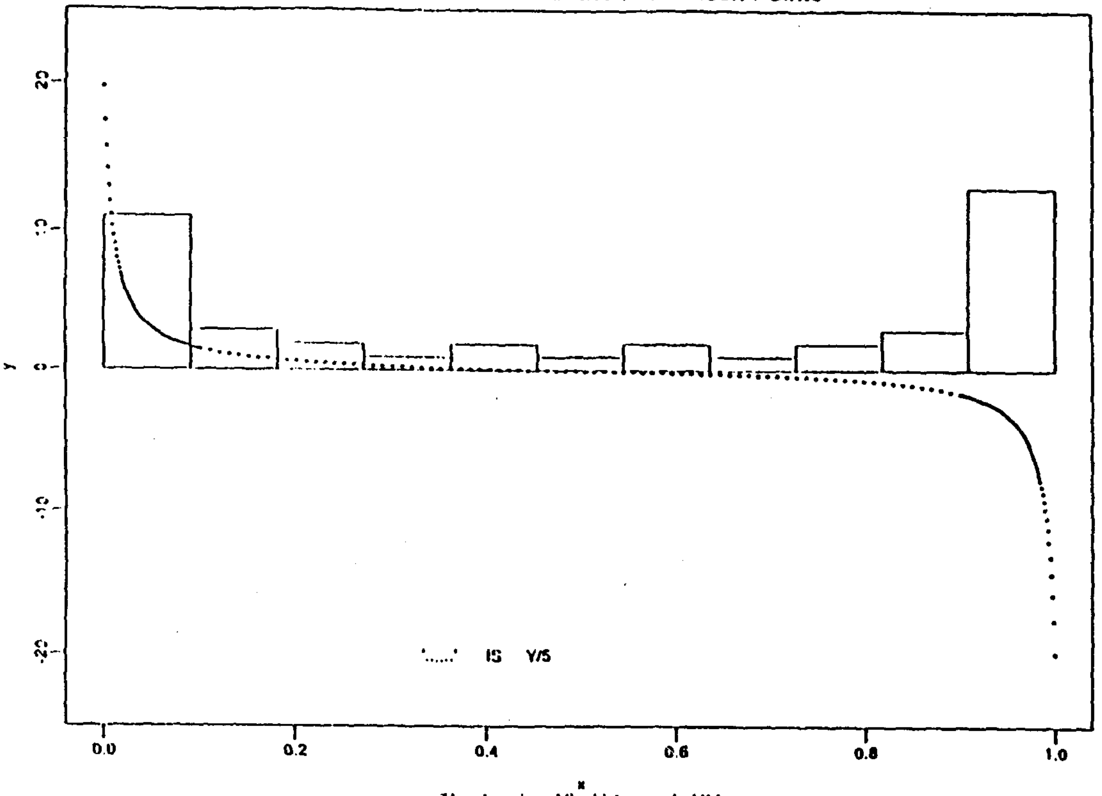


The step size of the histogram is 1/11

COLSYS.L5 (EPS = 1.D-2, TOL = 1.D-4)
Errors at the Final Mesh Points

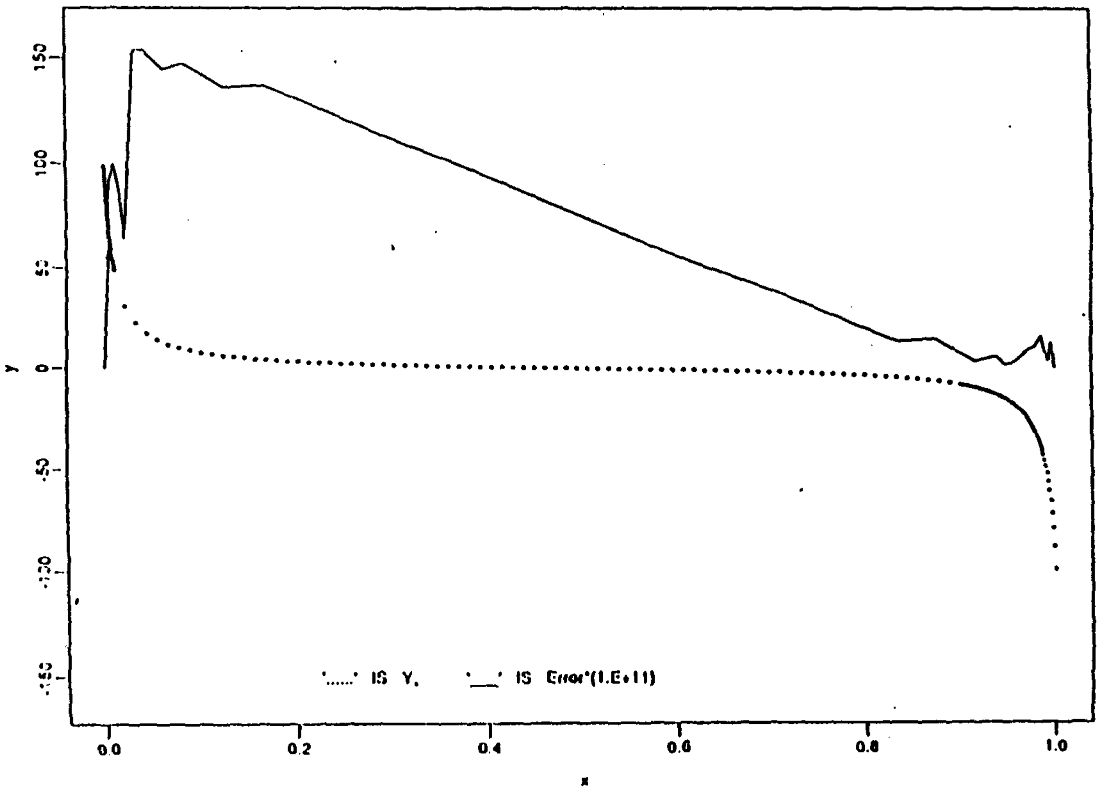


HAGRON.L5 (EPS = 1.D-2, TOL = 1.D-4)
Distribution of the Final Mesh Points

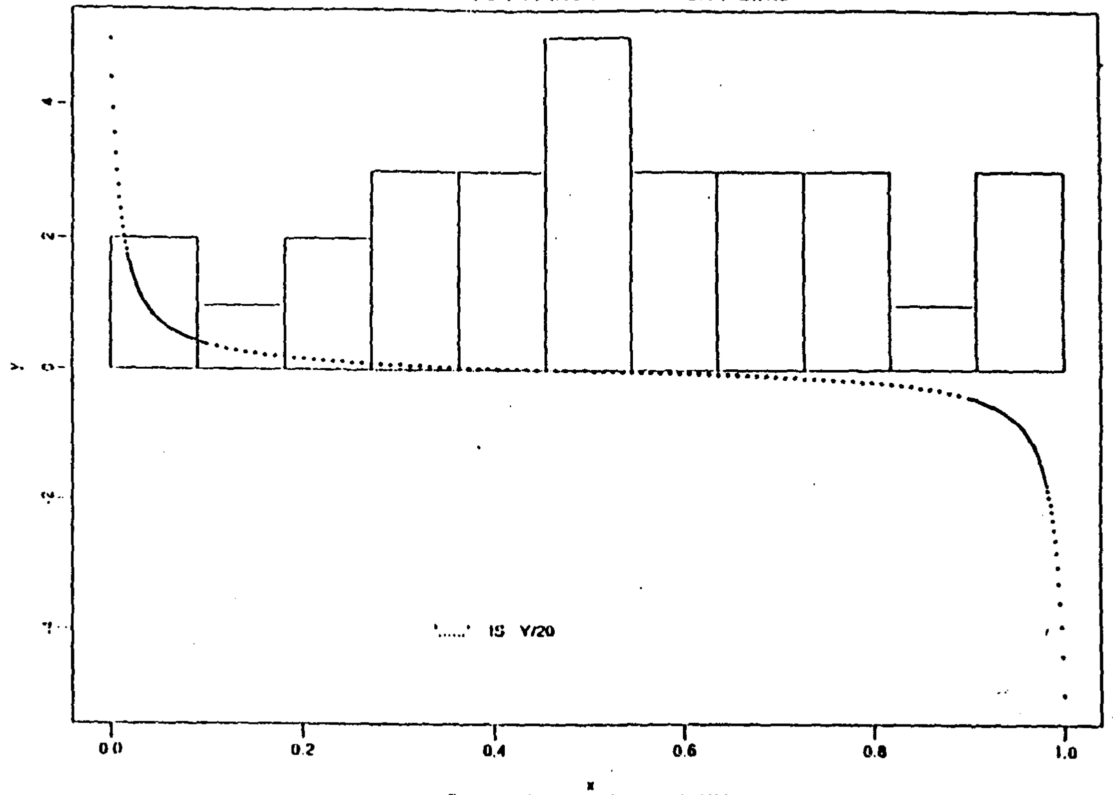


The step size of the histogram is 1/11

HAGRON.L5 (EPS = 1.D-2, TOL = 1.D-4)
Errors at the Final Mesh Points

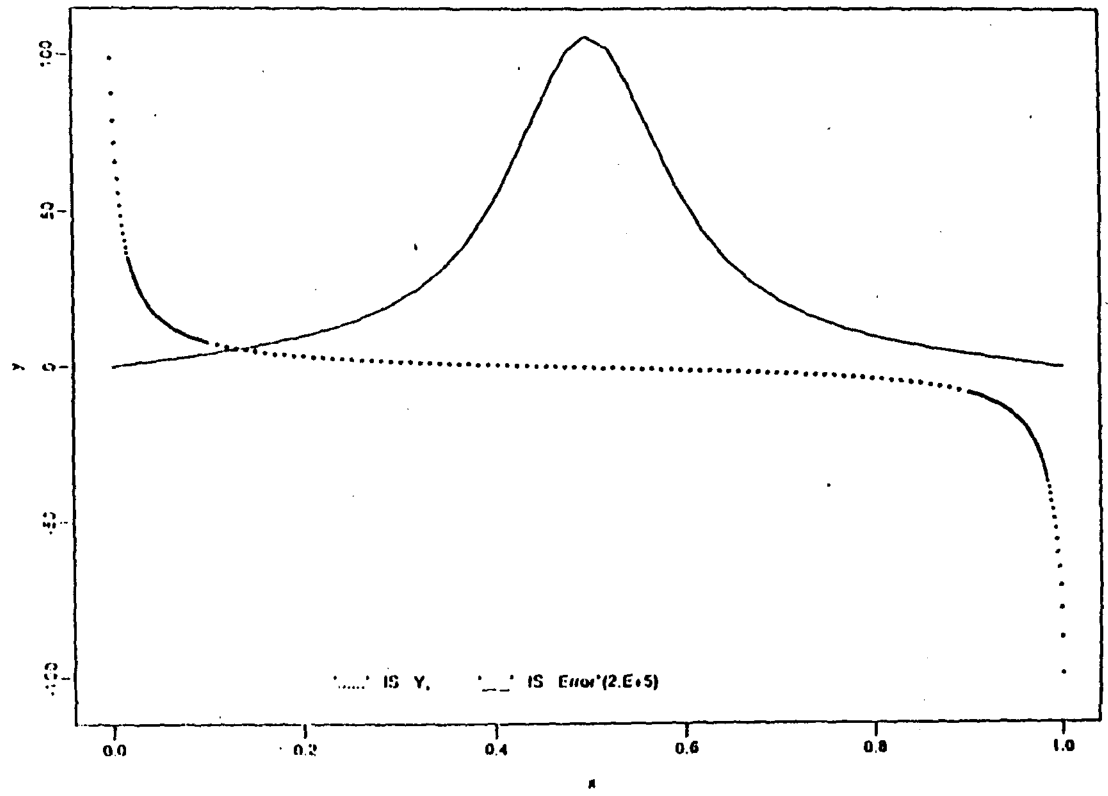


MUTS.L5 (EPS=1.D-2, TOL1=1.D-9, TOL2=1.D-10)
Distribution of the Final Mesh Points

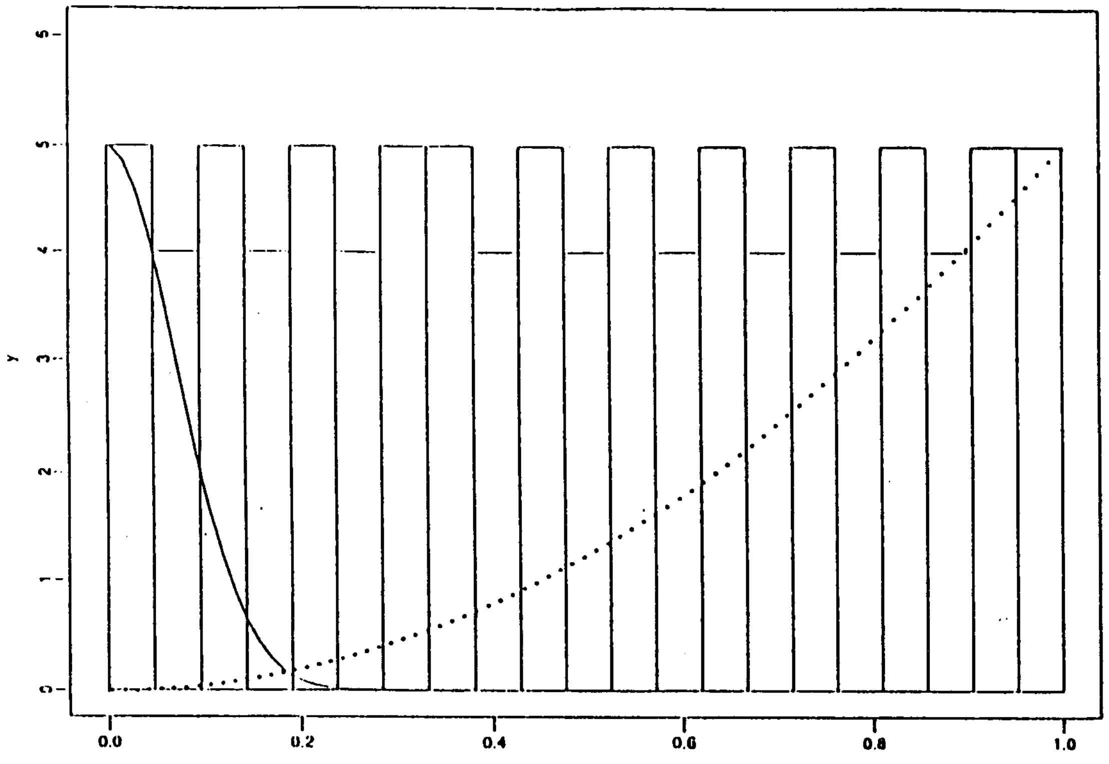


The step size of the histogram is 1/11

MUTS.L5 (EPS=1.D-2, TOL1=1.D-9,TOL2=1.D-10)
Errors at the Final Mesh Points



MUTS.N3 (EPS=1.D-2, TOL1=1.D-2, TOL2=1.D-1)
Distribution of the Final Mesh Points



The step size of the histogram is 1/21

Appendix (III) Testing Results

This Appendix has three parts. The third part contains all the raw data we collected. The first two parts of each contains some condensed information concerning a specific aspect of the data. These two parts are based on the data we have in the third part and may help the reader to understand better some of the conclusions we have in Chapter Four.

1: Timing Efficiency

1.a) Quality of Solution Oriented Efficiency

In the tables shown below, we compare the timing efficiency of the four codes a pair at a time. At the top of each table, the two codes that are compared is stated. The first column of each table contains the values of *eps*, and the first row of the table contains the problem number (see test problems in Appendix I). The rest of the cells are used to indicate our opinion (based on the data we collected) concerning the relative timing efficiency of the codes stated above the table. If '@' is shown at, say the cell corresponds to *L2* and *1.D-2*, it means that the first code shown above the table is more efficient than the second one on test problem *L2* with an *eps 1.D-2* in terms of the quality of solution oriented efficiency. On the other hand, '%' will be used to indicated the opposite. If such a relative efficiencies is not clear from our data, we will leave the cell blank. However, *N1* and *N4* do not have an *eps* and *L6* has two parameters that can vary. In order to make it easy for the reader to visualize the overall comparison concerning all the test problems, we include the comparison concerning these three test problems in our tables, and hereby remind the reader that cells in the columns headed by *N1* and *N3* represent the same test problems. For cells in the column headed by *L6*, *1.D-2* represents *(0,1)*, *1.D-4* represents *(4,5)* and so on. To see how the tables are filled, when running L1 with *eps* equal to *1.D-2* using COLNEW and COLSYS, from data table 1 and 2 one can see that for the four runs we listed, if we match the maximum errors produced by the two codes and compare the corresponding CPU times, COLNEW is faster. *e.g.* when the first type of maximum error is *1.D-4*, *1.D-6* and *1.D-8*, respectively, the corresponding CPU times (in milliseconds) for COLNEW are *140, 226,* and *441;* for COLSYS these CPU times are *185, 324*, and *373*. Thus we put in the cell in the first table below that corresponds *L1* and *1.D-2* a '@'. The comparison conducted here is subjective. As a supplement to this comparison, all the raw data we collected are attached in the raw data tables in the third section of this Appendix.

There are two different types of tables that involve COLNEW or COLSYS. The first type of tables compares COLNEW or COLSYS to other codes by using the first type of errors from the two codes (see Appendix I), and the second type uses the second type of errors. Whether or not a table is in the first or second type is indicated by the heading of the table.

It should be noticed that we do not compare a code to other codes in terms of quality of solution oriented efficiencies on the problems it fails to solve, and we put into the cells that correspond problems on which at least one of the two codes compared completely fail one of the following three symbols: ff, sf and fs. 'ff' stands for both codes compared failed, 'fs' and 'sf' stand for when only the first code and only the second code failed, respectively.

Codes: COLNEW and COLSYS (Type I)

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2	@		@	%	@	@		@	%	@
1.D-4	@	@	%	@	@	@		@	@	
1.D-6	@	@	%	*ff*	@	@		@	@	
1.D-8	*ff*	@	%	*ff*	*ff*	@		@	@	

Codes: COLNEW and HAGRON (Type I)

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2	@	%	@		%	%			%	@
1.D-4	@	%	@	@	%	%		@	@	
1.D-6			@	*ff*				@	*sf*	
1.D-8	*ff*	%	@	*ff*	*ff*			@	*sf*	

Codes: COLNEW and MUTS (Type I)

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2	@	@	@	@	%	@	@	@		@
1.D-4	@	%	@		%	@		@	*sf*	
1.D-6			@	*ff*	%	@		@	*sf*	
1.D-8	*ff*	%	@	*ff*	*ff*	@		@	*sf*	

Codes: COLSYS and HAGRON (Type I)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	%	@	@	%	%			%	
1.D-4	@	%	@	@	%	%		@	@	
1.D-6	@	%	@	ff		%		@	sf	
1.D-8	ff	%	@	ff	ff			@	sf	

Codes: COLSYS and MUTS (Type I)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	@	@	@	%	@	@	@	@	@
1.D-4	@	%	@		%	@		@	sf	
1.D-6		@	@	ff	%	@		@	sf	
1.D-8	ff	%	@	ff	ff	@		@	sf	

Codes: HAGRON and MUTS

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	@	@	@		@	@	@	@	@
1.D-4	@	@			@	@		@	sf	
1.D-6		@		ff	%	@		@	ff	
1.D-8	ff	@		ff	ff			@	ff	

Codes: COLNEW and COLSYS (Type II)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@		@	%	@	@			%	@
1.D-4	@	@	%	@	@	@		@	@	
1.D-6	@	@	%	ff	@	@		@		
1.D-8	ff	@	%	ff	ff	@		@	@	

Codes: COLNEW and HAGRON (Type II)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	%	@		%	%	%	%	%	@
1.D-4	@	%	@	@	%	%		%	@	
1.D-6	@		@	ff		%		%	sf	
1.D-8	ff		@	ff	ff			%	sf	

Codes: COLNEW and MUTS (Type II)

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2	@	@	@	@	%	@	@	@	%	@
1.D-4	@	%	@		%			@		@
1.D-6			@	*ff*	%			@	*sf*	@
1.D-8	*ff*	%	@	*ff*	*ff*			@	*sf*	@

Codes: COLSYS and HAGRON (Type II)

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2	@	%	@	@	%	%	%	%	%	%
1.D-4	@	%	@	@	%	%		%	@	
1.D-6	@	%	@	*ff*		%		%	*sf*	
1.D-8	*ff*	%	@	*ff*	*ff*			%	*sf*	

Codes: COLSYS and MUTS (Type II)

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2	@	@	@	@	%	%	@	@	@	@
1.D-4	@	%	@		%	%		@	*sf*	
1.D-6		@	@	*ff*	%	%		@	*sf*	
1.D-8	*ff*	%	@	*ff*	*ff*			@	*sf*	

1.b) Relative Efficiency Involving the Tolerance Settings

The following tables compare the resulting CPU times for the cases where the codes have the same tolerance setting, ***and*** the maximum ***absolute*** errors from the codes are *of the same size as or less than* the tolerance. By doing so, the exact values of the maximum errors are discounted but the response of the solution to the input tolerance is partially utilized. To users who want to interpret the input tolerance as the upper bound of the maximum absolute errors they prepare to tolerate, and *only* demand that the errors are of the same size as or less than the tolerance, such a relative efficiency answers the question that how efficient the codes can be in terms of the CPU times they need to satisfy the users' demand. Should the maximum relative errors be available, a timing efficiency in terms of the CPU times needed by the codes in order to achieve relative errors that are of the same size as or less than the specified tolerance may also be revealed.

The approach of revealing the relative efficiencies we described above is a compromise between the approach that compares the codes only under the similar input parameter settings and the quality of solution oriented comparison where the input parameter settings are not involved. Compared with the approach that compares the codes only under the similar parameter settings, the above approach takes into consideration some of the responses of the solution to the tolerance, in the sense that the relative efficiencies are evaluated under the condition that the maximum errors are not greater than the tolerance. When compared with the quality of solution oriented comparison, the information concerning the quality of the solution is not fully utilized but it takes into consideration the relationship between the solution and tolerance.

The approaches we newly discussed here may be useful and practical if the role that the tolerance plays in the termination criteria is further explored and the magnitude of the solution is also taken into consideration. We mentioned in Chapter Four that accuracies of the numerical solutions from HAGRON appear to have a stronger relationship with input tolerance than that of the other codes. This is also indicated by the observation that while the accuracies of the solutions from the codes are usually all less than the input tolerance, the solutions produced by HAGRON are often closer to the input tolerance than the solutions of the others. If a user is satisfied as long as the accuracy is close to a certain value of the tolerance (*i.e.* a better accuracy is not needed), then HAGRON has an clear advantage as it usually does not spend much time trying to improve the solution that is already satisfactory. This brings HAGRON a better timing efficiency as one can see from the tables below.

Note: The meaning of '@', '%' and 'ff' etc. are the same as they are in *1.a*

Codes: COLNEW and COLSYS (Type I)

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2	@	%	@	%	@	@		@	%	@
1.D-4	@	@	%		@	@		@	@	
1.D-6	*ff*		%	*ff*	@			@		
1.D-8	*ff*	@	%	*ff*	*ff*			@		

Codes: COLNEW and HAGRON (Type I)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	%	%	@	%	%	%	%	%	%
1.D-4	@	%	@		%	%		%	@	
1.D-6		%	@	ff		%		%	sf	
1.D-8	ff	%	@	ff	ff			%	sf	

Codes: COLNEW and MUTS (Type I)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	@	%		%	%	@	@		@
1.D-4	@	@	%		%	@			sf	
1.D-6		@		ff					sf	
1.D-8	ff			ff	ff				sf	

Codes: COLSYS and HAGRON (Type I)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	%	%	%	%	%	%	%	%	%
1.D-4		%	@		%	%		%	@	
1.D-6		%	@	ff		%		%	sf	
1.D-8	ff	%	@	ff	ff			%	sf	

Codes: COLSYS and MUTS (Type I)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	@	%	@	%	%	@	@	@	@
1.D-4	@	%	@		%				sf	
1.D-6		%	@	ff					sf	
1.D-8	ff		@	ff	ff				sf	

Codes: HAGRON and MUTS

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	@	%	@	%	@	@	@	%	
1.D-4	@	%	%		%	@			@	
1.D-6		%		ff					ff	
1.D-8	ff			ff	ff				ff	

Codes: COLNEW and COLSYS (Type II)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	%	@	@	@	@		@	%	@
1.D-4	@		%		@	@		@	@	
1.D-6		@	%	ff	@			@		
1.D-8	ff	@	%	ff	ff			@		

Codes: COLNEW and HAGRON (Type II)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	%	%	%	%	%	%	%	%	%
1.D-4	@	%	@		%	%		%	@	
1.D-6		%	@	ff		%		%	sf	
1.D-8	ff	%	@	ff	ff			%	sf	

Codes: COLNEW and MUTS (Type II)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	@	%	@	%	%	@	@		@
1.D-4	@	@	%		%	@		@	sf	
1.D-6		@		ff					sf	
1.D-8	ff			ff	ff				sf	

Codes: COLSYS and HAGRON (Type II)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	%	%	%	%	%	%	%	%	%
1.D-4		%	@		%	%		%	@	
1.D-6		%	@	ff		%		%	sf	
1.D-8	ff	%	@	ff	ff			%	sf	

Codes: COLSYS and MUTS (Type II)

ε \ #	L1	L2	L3	L4	L5	L6	N1	N2	N3	N4
1.D-2	@	@	%	@	%	%	@	@	@	@
1.D-4	@	%	@		%				sf	
1.D-6		%	@	ff					sf	
1.D-8	ff		@	ff	ff				sf	

2: The Degrees of Difficulties of the Test Problems Where the Codes Failed

The following tables indicate the degrees of difficulties of the test problems at which the codes failed or program failure started to occur. By 'failed at certain degree of difficulty of a problem', we mean that the four runs that correspond to four different tolerance settings on a test problem with a certain degree of difficult all failed, and in this case we put a '@' at the corresponding cell. *e.g.* COLNEW failed on L1 when *eps* is set to *1.D-8* for all the four runs, thus we put a '@' at the cell that corresponds to L1 and *1.D-8*. If not all runs failed but at least one of the runs did, we will put a '%' in the corresponding cell. Should a code successfully solve a problem at all the four runs, we will leave the corresponding cell blank. For a complete set of code failures on the test problems and the reasons for failures, see the raw data tables in the next section.

Codes: COLNEW

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2										
1.D-4				%						
1.D-6	%			@	%					
1.D-8	@			@	@					

Codes: COLSYS

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2										
1.D-4				%						
1.D-6	%			@	%					
1.D-8	@			@	@					

Codes: HAGRON

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2										
1.D-4				%						
1.D-6	%			@	%				@	
1.D-8	@			@	@				@	

Codes: MUTS

ε \ #	*L1*	*L2*	*L3*	*L4*	*L5*	*L6*	*N1*	*N2*	*N3*	*N4*
1.D-2									%	
1.D-4	%			@				@	@	
1.D-6	@			@	%			@	@	
1.D-8	@			@	@			@	@	

3: Raw Data Tables

The tables below contain the detailed results from our testing. In Appendix I, we have provided the details about our testing. With those details and the parameters shown in the tables below, the testing should be fairly easily reconducted. The meaning for the headings of each column in the tables are explained at the bottom of each table. The following is a list of notation we use to indicate various abnormal exits from the codes:

%	— The supplied storage is not enough (exit with solution);
@	— The supplied storage is not enough (exit with no solution);
*	— Program overflow;
#	— Wrong solution (due to unreasonably big error);
$	— Elapsed time is more than *1/4* hour (runs are stopped manually);
$$	— The number of iterations needed exceed 50.

One may notice that some of the tables are left blank. This is because the code failed to solve the problem with lower difficulties (usually this is indicated by the results in the tables above the blank tables) and the failures are accompanied by big costs. It is difficult for us to quantify everything we say. For example, when we say 'wrong solution due to unreasonably big error', we just want to remind the reader that the error indicated by '#', in our opinion, is too big or at least unusual. One may hold a different view about this if one takes into consideration the magnitude of the maximum value of the solution or the relative error. Often, when a code fails because the supplied storage is not enough, it may be able to provide the user with the partially converged solutions. '%' in the above list is used to indicated the failures of this type. If the current mesh points and the partially converged solution are not available at the time a code fails, we use @ above to indicate the failure.

TABLE 1

CODE: COLNEW TEST PROBLEM: L1

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	140	0.50D-4	0.14D-2
1.D-4	11	226	0.12D-6	0.71D-6
1.D-6	41	441	0.34D-8	0.96D-8
1.D-8	41	555	0.52D-10	0.28D-9

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	81	943	0.62	0.26D-1
1.D-4	81	2886	0.56D-10	0.54D-10
1.D-6	77	3745	0.91D-13	0.82D-13
1.D-8	77	3694	0.82D-13	0.91D-13

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	583	18913	0.20D-1	0.18D-8
1.D-4	583	18879	0.20D-1 %	0.19D-8
1.D-6	583	18899	0.20D-1 %	0.16D-8
1.D-8	583	18917	0.20D-1 %	0.18D-8

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	114	1.00 %	1.00
1.D-4	583	18834	1.00 %	0.82
1.D-6	583	18834	1.00%	0.82
1.D-8	583	18857	1.00 %	0.82

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 2

CODE: COLSYS TEST PROBLEM: L1

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	185	0.50D-4	0.15D-2
1.D-4	11	324	0.12D-6	0.71D-6
1.D-6	11	373	0.46D-8	0.49D-7
1.D-8	19	672	0.26D-8	0.29D-8

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	81	1038	0.84D-1	0.12D-6
1.D-4	81	2892	0.12D-4	0.12D-4
1.D-6	81	3663	0.11D-4	0.11D-4
1.D-8	109	4445	0.52D-5	0.52D-5

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	619	24688	0.74D-1	0.74D-1
1.D-4	495	26222	0.37D-1 %	0.27D-2
1.D-6	495	26404	0.33D-1 %	0.13D-2
1.D-8	495	26079	0.30D-1 %	0.18D-2

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	147	1.00 %	1.00
1.D-4	405	28266	1.00%	0.85
1.D-6	405	28340	1.00 %	0.85
1.D-8	619	24496	1.00 %	1.00

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 3

CODE: HAGRON

TEST PROBLEM: L 1

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	23	240	0.80D-3	
1.D-4	32	483	0.44D-5	
1.D-6	57	645	0.72D-7	
1.D-8	72	1043	0.40D-9	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	52	1751	0.16D-4	
1.D-4	73	2084	0.14D-5	
1.D-6	281	4580	0.13D-4	
1.D-8	475	7155	0.39D-6	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	112	21668	0.11D-3	
1.D-4	247	14790	0.20D-6	
1.D-6	929	24476	0.15D-7	
1.D-8	1265	662234	2.00#	

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	2	16150	@	
1.D-4	2	16121	@	
1.D-6	2	16138	@	
1.D-8	2	16147	@	

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 4

CODE: MUTS

TEST PROBLEM: L1

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	281	0.43D-3	
1.D-4	20	626	0.26D-5	
1.D-6	20	1637	0.16D-7	
1.D-8	20	4123	0.13D-9	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	100	20743	0.69D-7	
1.D-4	100	59968	0.21D-10	
1.D-6	100		$	
1.D-6	100		$	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20		$	
1.D-4	20		$	

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 5

CODE: COLNEW TEST PROBLEM: L2

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	24	0.10D-6	0.53D-7
1.D-4	11	24	0.10D-6	0.53D-7
1.D-6	11	63	0.33D-10	0.33D-10
1.D-8	21	75	0.86D-12	0.86D-12

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	41	151	0.23D-5	0.24D-6
1.D-4	41	165	0.66D-7	0.19D-7
1.D-6	41	331	0.63D-11	0.26D-13
1.D-8	53	363	0.69D-12	0.27D-12

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	89	372	0.29D-5	0.30D-5
1.D-4	29	352	0.42D-8	0.26D-8
1.D-6	73	469	0.66D-10	0.70D-12
1.D-8	321	1331	0.24D-12	0.20D-12

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	28	0.90	0.80
1.D-4	105	787	0.93D-6	0.11D-5
1.D-6	101	950	0.35D-10	0.15D-10
1.D-8	161	1416	0.95D-11	0.82D-12

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 50 EQUALLY SPACED POINTS

TABLE 6

CODE: COLSYS TEST PROBLEM: L2

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	26	0.13D-6	0.84D-7
1.D-4	11	26	0.13D-6	0.84D-7
1.D-6	11	42	0.13D-9	0.57D-10
1.D-8	11	53	0.14D-9	0.81D-10

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	41	171	0.24D-5	0.24D-6
1.D-4	21	174	0.19D-6	0.74D-7
1.D-6	41	463	0.56D-12	0.69D-12
1.D-8	53	498	0.71D-12	0.50D-12

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	89	415	0.29D-5	0.30D-5
1.D-4	81	451	0.48D-7	0.42D-7
1.D-6	73	620	0.16D-9	0.15D-10
1.D-8	81	841	0.13D-10	0.14D-10

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	26	0.90	0.80
1.D-4	105	940	0.10D-5	0.12D-5
1.D-6	105	1295	0.79D-10	0.86D-10
1.D-8	161	1780	0.23D-9	0.55D-9

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 50 EQUALLY SPACED POINTS

TABLE 7

CODE: HAGRON TEST PROBLEM: L2

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	10	0.88D-6	
1.D-4	7	14	0.26D-7	
1.D-6	7	15	0.26D-8	
1.D-8	17	44	0.84D-10	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	16	55	0.76D-4	
1.D-4	19	60	0.25D-6	
1.D-6	44	119	0.90D-9	
1.D-8	78	224	0.51D-10	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	26	159	0.59D-5	
1.D-4	44	153	0.42D-6	
1.D-6	97	414	0.34D-8	
1.D-8	132	430	0.15D-9	

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	9	0.67	
1.D-4	129	632	0.27D-6	
1.D-6	214	577	0.23D-9	
1.D-8	292	955	0.15D-10	

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 8

CODE: MUTS TEST PROBLEM: L2

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	34	0.27D-1	
1.D-4	50	74	0.25D-3	
1.D-6	100	149	0.73D-5	
1.D-8	300	440	0.13D-8	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	35	0.52D-2	
1.D-4	20	40	0.45D-4	
1.D-6	20	67	0.13D-5	
1.D-8	20	134	0.41D-7	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	50	83	0.17D-1	
1.D-4	51	98	0.13D-3	
1.D-6	200	345	0.62D-5	
1.D-8	300	586	0.25D-6	

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	100	162	0.79D-1	
1.D-4	300	464	0.18D-1	
1.D-6	300	547	0.18D-3	
1.D-8	300	700	0.80D-6	

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 9

CODE: COLNEW   TEST PROBLEM: L3

EPS: 1/11

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	46	0.25D-11	0.90D-8
1D-4	11	46	0.25D-11	0.90D-8
1D-6	41	158	0.39D-13	0.22D-9
1D-8	161	546	0.83D-14	0.43D-11

EPS: 1/33

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	81	221	0.44D-5	0.98D-5
1D-4	41	185	0.13D-10	0.18D-7
1D-6	321	1087	0.94D-13	0.55D-10
1D-8	161	644	0.26D-13	0.34D-10

EPS: 1/55

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	41	185	0.87D-8	0.10D-5
1D-4	161	646	0.15D-11	0.12D-8
1D-6	321	1296	0.47D-13	0.98D-11
1D-8	1281	4784	0.74D-13	0.86D-12

EPS: 1/77

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	41	217	0.59D-8	0.15D-5
1D-4	161	646	0.59D-10	0.13D-7
1D-6	321	1295	0.14D-12	0.94D-10
1D-8	621	2593	0.90D-13	0.80D-12

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 10

CODE: COLSYS   TEST PROBLEM: L3

EPS: 1/11

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	62	0.21D-11	0.90D-8
1D-4	11	62	0.21D-11	0.90D-8
1D-6	81	321	0.64D-13	0.29D-9
1D-8	41	245	0.25D-13	0.25D-11

EPS: 1/33

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	62	0.34D-5	0.18D-3
1D-4	21	142	0.44D-9	0.30D-6
1D-6	41	299	0.55D-12	0.70D-9
1D-8	81	611	0.16D-12	0.14D-11

EPS: 1/55

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	41	198	0.12D-5	0.20D-4
1D-4	41	299	0.92D-10	0.73D-7
1D-6	81	617	0.11D-12	0.14D-9
1D-8	81	614	0.11D-12	0.14D-9

EPS: 1/77

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	41	246	0.49D-6	0.17D-4
1D-4	41	300	0.59D-8	0.15D-5
1D-6	161	1019	0.11D-12	0.26D-9
1D-8	161	1240	0.24D-12	0.56D-11

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 11

CODE: HAGRON

TEST PROBLEM: L3

EPS: 1/11

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	25	65	0.12D-5	
1.D-4	35	112	0.65D-6	
1.D-6	49	127	0.41D-8	
1.D-8	97	251	0.16D-10	

EPS: 1/33

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	97	274	0.59D-9	
1.D-4	193	639	0.13D-8	
1.D-6	272	865	0.29D-9	
1.D-8	732	2508	0.80D-13	

EPS: 1/55

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	167	734	0.24D-5	
1.D-4	387	1434	0.52D-9	
1.D-6	520	2504	0.32D-9	
1.D-8	1569	5632	0.29D-13	

EPS: 1/77

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	292	1219	0.17D-2	
1.D-4	443	1502	0.66D-8	
1.D-6	642	4360	0.13D-9	
1.D-8	1977	8756	0.11D-12	

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 12

CODE: MUTS

TEST PROBLEM: L3

EPS: 1/11

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	38	0.48D-2	
1.D-4	50	94	0.54D-4	
1.D-6	50	130	0.45D-5	
1.D-8	50	245	0.59D-7	

EPS: 1/33

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	50	102	0.14D-1	
1.D-4	50	166	0.64D-3	
1.D-6	300	572	0.36D-5	
1.D-8	300	944	0.33D-6	

EPS: 1/55

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	100	0.76D-1	
1.D-4	200	378	0.58D-3	
1.D-6	300	796	0.30D-4	
1.D-8	300	1492	0.42D-6	

EPS: 1/77

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	125	0.81D-1	
1.D-4	300	563	0.57D-3	
1.D-6	300	929	0.74D-4	
1.D-8	300	2049	0.54D-6	

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 13

CODE: COLNEW   TEST PROBLEM: L4

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	41	159	0.29D-6	0.17D-5
1.D-4	41	282	0.37D-7	0.42D-6
1.D-6	41	405	0.46D-8	0.12D-7
1.D-8	59	507	0.68D-10	0.94D-9

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	9	23	1133.18 #	2517.20
1.D-4	225	1087	0.24D-3	0.60D-4
1.D-6	225	1037	0.73D-5	0.18D-5
1.D-8	321	1464	0.13D-3	0.33D-4

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	9	23	1290.43 #	3352.72
1.D-4	11	67	2234.74 #	3352.72
1.D-6	1937	24891	58.87 #	0.20
1.D-8	1937	24519	4656.46 #	15.61

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 14

CODE: COLSYS   TEST PROBLEM: L 4

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	21	143	0.16D-5	0.34D-5
1.D-4	41	200	0.29D-6	0.17D-5
1.D-6	41	281	0.78D-9	0.13D-7
1.D-8	41	656	0.39D-9	0.38D-8

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	9	23	1133.18 #	2517.21
1.D-4	113	1231	0.55D-1	0.14D-1
1.D-6	129	1265	0.82D-3	0.21D-3
1.D-8	1281	16723	3.27 %	0.82

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	27	3352.72 #	21478.85
1.D-4	11	85	3352.72 #	1652.08
1.D-6	11	126	3352.72 #	69576757
1.D-8	1875	29594	3377.03 #	1007251.88

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 15

CODE: HAGRON *TEST PROBLEM: L4*

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	37	127	0.18D-2	
1.D-4	39	148	0.77D-4	
1.D-6	77	308	0.19D-6	
1.D-8	94	244	0.71D-7	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	9	10060	
1.D-4	1395	5435	0.23D-2	
1.D-6	2664	9983	0.36D-3	
1.D-8			$	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	9	1.D+6 #	
1.D-4	7	9	1.D+6 #	
1.D-6			$	
1.D-8			$	

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 16

CODE: MUTS *TEST PROBLEM: L4*

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	100	152	0.95D-2	
1.D-4	200	303	0.15D-3	
1.D-6	300	448	0.13D-4	
1.D-8	300	467	0.14D-4	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	48	9999.99 #	
1.D-4	301	459	10004.97 #	
1.D-6	301	488	12571.10#	
1.D-8	301	628	8806.10 #	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	301	457	1.D+6 #	
1.D-4	302	483	1.D+6 #	
1.D-6	302	600	1.D+6 #	
1.D-8	302	897	1.D+6 #	

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	303	477	1.D+8 #	
1.D-4	303	532	1.D+8 #	
1.D-6	304	735	1.D+8 #	
1.D-8	304	1295	1.D+8 #	

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 17

CODE: COLNEW TEST PROBLEM: L 5

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	29	175	0.48D-2	0.47D-2
1.D-4	45	296	0.83D-6	0.81D-6
1.D-6	61	436	0.13D-10	0.70D-9
1.D-8	41	415	0.52D-11	0.11D-9

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	81	1077	0.27D-5	0.26D-5
1.D-4	57	1141	0.96D-6	0.93D-6
1.D-6	133	1224	0.38D-7	0.36D-7
1.D-8	129	1489	0.24D-7	0.24D-7

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	44	91601555 #	92995926 #
1.D-4	161	2546	0.14D-3	0.13D-3
1.D-6	153	2384	0.11D-3	0.11D-3
1.D-8	1363	2361	0.21D-2	0.20D-2

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	46	88928264 #	93102855
1.D-4	11	44	89896255 #	93097955
1.D-6	1925	26402	33.80%	32.60
1.D-8	1925	26402	33.80%	32.60

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 36 EQUALLY SPACED POINTS

TABLE 18

CODE: COLSYS TEST PROBLEM: L5

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	29	220	0.48D-2	0.47D-2
1.D-4	137	558	0.89D-6	0.92D-6
1.D-6	45	616	0.22D-11	0.39D-9
1.D-8	81	1038	0.60D-11	0.23D-10

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	105	1208	0.37D-3	0.35D-3
1.D-4	153	1501	0.53D-5	0.51D-5
1.D-6	133	1776	0.37D-7	0.35D-7
1.D-8	129	2146	0.43D-8	0.41D-8

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	25	799600 #	93051258
1.D-4	465	3477	0.22D-4	0.21D-4
1.D-6	1281	5331	0.41D-4	0.40D-4
1.D-8	1523	29283	0.62D-4	0.60D-4

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	26	79999600 #	93102926
1.D-4	11	26	79999600 #	93102926
1.D-6	3125	32126	0.64 %	0.62
1.D-8	3125	32862	0.66%	0.64

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 36 EQUALLY SPACED POINTS

TABLE 19

CODE: HAGRON TEST PROBLEM: L5

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	25	100	0.52D-5	
1.D-4	41	187	0.15D-8	
1.D-6	54	202	0.95D-10	
1.D-8	116	402	0.13D-12	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	110	650	0.51D-8	
1.D-4	133	1005	0.11D-8	
1.D-6	216	2168	0.11D-8	
1.D-8	947	4795	0.11D-8	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	169	2463	0.13D-4	
1.D-4	406	5558	0.12D-4	
			*	
			*	

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2			*	
1.D-2			*	
1.D-6			*	
1.D-8			$	

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 20

CODE: MUTS TEST PROBLEM: L5

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	55	0.12D-2	
1.D-4	20	86	0.36D-4	
1.D-6	20	184	0.51D-6	
1.D-8	20	457	0.88D-8	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	99	0.15D-1	
1.D-4	20	194	0.29D-3	
1.D-6	20	508	0.11D-5	
1.D-8	20	1435	0.56D-7	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	300	640	1.66 #	
1.D-4	300	869	0.35D-2	
1.D-6	300	1571	0.64D-3	
1.D-8	300	3532	0.13D-2	

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	300	698	1328.99 #	
1.D-4	300	1072	53.60#	
1.D-6	300	2575	49.63 #	
1.D-8	300	7221	86.35 #	

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 21

CODE: COLNEW TEST PROBLEM: L 6

DOMAIN: (0,1)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	96	0.63D-12	0.61D-7
1.D-9	21	217	0.28D-13	0.19D-8
1.D-10	41	456	0.59D-13	0.55D-10
1.D-12	81	952	0.79D-13	0.15D-11

DOMAIN: (4,5)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	95	0.19D-18	0.18D-3
1.D-9	21	218	0.63D-10	0.58D-5
1.D-10	41	466	0.66D-10	0.16D-6
1.D-12	81	949	0.17D-9	0.38D-8

DOMAIN: (8,9)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	95	0.56D-5	0.53
1.D-9	21	222	0.18D-6	0.18D-1
1.D-10	41	461	0.16D-6	0.48D-3
1.D-12	81	950	0.45D-6	0.13D-4

DOMAIN: (14,15)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	94	0.91	87017
1.D-9	21	222	0.29D-1	2855
1.D-10	41	461	0.29D-1	78.90
1.D-12	81	969	0.25D-1	2.08

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 22

CODE: COLSYS TEST PROBLEM: L6

DOMAIN: (0,1)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	111	0.58D-12	0.61D-7
1.D-9	21	257	0.15D-12	0.20D-8
1.D-10	41	546	0.29D-12	0.15D-10
1.D-12	81	1131	0.45D-12	0.15D-11

DOMAIN: (4,5)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	111	0.18D-8	0.18D-3
1.D-9	21	259	0.17D-9	0.59D-5
1.D-10	41	543	0.54D-9	0.16D-6
1.D-12	81	1114	0.12D-8	0.45D-8

DOMAIN: (8,9)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	111	0.52D-5	0.53
1.D-9	21	258	0.42D-6	0.18D-1
1.D-10	41	549	0.12D-5	0.49D-3
1.D-12	81	1113	0.34D-5	0.13D-4

DOMAIN: (14,15)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	111	0.86	87017
1.D-9	21	256	0.10	2855
1.D-10	41	550	0.24	78.95
1.D-12	81	1125	0.37	2.13

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 23

CODE: HAGRON

TEST PROBLEM: L6

DOMAIN: (0,1)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	22	0.28D-6	
1.D-9	13	80	0.34D-11	
1.D-10	19	102	0.13D-12	
1.D-12	33	211	0.16D-14	

DOMAIN: (4,5)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	22	0.84D-3	
1.D-9	13	78	0.10D-7	
1.D-10	19	101	0.39D-9	
1.D-12	37	211	0.73D-11	

DOMAIN: (8,9)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	22	2.50	
1.D-9	13	78	0.30D-4	
1.D-10	19	102	0.12D-5	
1.D-12	37	213	0.15D-6	

DOMAIN: (14,15)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	22	408641 #	
1.D-9	13	80	4.91	
1.D-10	19	102	0.19	
1.D-12	37	211	0.26D-1	

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 24

CODE: MUTS

TEST PROBLEM: L6

DOMAIN: (0,1)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	75	0.68D-7	
1.D-8	20	101	0.28D-8	
1.D-9	20	156	0.12D-9	
1.D-12	20	337	0.30D-11	

DOMAIN: (4,5)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	74	0.20D-3	
1.D-8	100	362	0.56D-7	
1.D-10	300	1061	0.31D-8	
1.D-12	300	1061	0.31D-8	

DOMAIN: (8,9)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	50	178	0.54D-2	
1.D-8	300	1063	0.92D-5	
1.D-10	300	1063	0.92D-5	
1.D-12	300	1063	0.92D-5	

DOMAIN: (14,15)

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	300	1053	1.50	
1.D-8	300	1053	1.50	
1.D-10	300	1053	1.50	
1.D-12	300	1053	1.50	

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 25

TEST PROBLEM: N1

CODE: COLNEW

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	175	0.87D-13	0.71D-8
1D-4	11	192	0.81D-13	0.71D-8
1D-6	11	194	0.81D-13	0.71D-8
1D-8	11	214	0.81D-13	0.71D-8

CODE: COLSYS

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	176	0.11D-12	0.71D-8
1D-4	11	193	0.81D-13	0.71D-8
1D-6	11	192	0.81D-13	0.71D-8
1D-8	11	211	0.81D-13	0.71D-8

CODE: HAGRON

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	7	63	0.55D-6	
1D-4	7	63	0.55D-6	
1D-6	7	73	0.11D-7	
1D-8	13	121	0.27D-10	

CODE: MUTS

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	20	759	0.20D-5	
1D-4	20	871	0.20D-5	
1D-6	20	976	0.41D-10	
1D-8	20	1001	0.41D-10	

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 26

CODE: COLNEW

TEST PROBLEM: N2

A: 1

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	118	0.23D-15	0.18D-11
1D-4	11	118	0.24D-15	0.18D-9
1D-6	11	120	0.24D-15	0.18D-9
1D-8	11	119	0.24D-15	0.18D-9

A: 10

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	196	0.11D-11	0.16D-5
1D-4	11	196	0.11D-11	0.16D-5
1D-6	21	283	0.19D-13	0.14D-6
1D-8	81	818	0.76D-14	0.62D-9

A: 20

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	199	0.20D-7	0.37D-3
1D-4	21	205	0.36D-10	0.16D-4
1D-6	81	856	0.88D-14	0.41D-7
1D-8	161	1614	0.95D-14	0.69D-11

A:30

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	21	254	0.65D-6	0.15D-2
1D-4	41	585	0.10D-11	0.25D-5
1D-6	81	1058	0.31D-13	0.18D-7
1D-8	161	1680	0.99D-14	0.70D-8

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 27

CODE: COLSYS TEST PROBLEM: N2

A: 1

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	182	1.20D-14	0.18D-11
1D-4	11	227	0.13D-14	0.17D-13
1D-6	11	230	0.13D-14	0.17D-13
1D-8	11	230	0.13D-14	0.17D-13

A: 10

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	242	0.11D-11	0.16D-5
1D-4	11	242	0.11D-11	0.16D-5
1D-6	21	452	0.37D-14	0.64D-5
1D-8	81	1069	0.92D-15	0.62D-9

A: 20

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	11	323	0.20D-7	0.38D-3
1D-4	21	533	0.11D-11	0.16D-5
1D-6	81	1097	0.68D-14	0.41D-7
1D-8	81	1417	0.70D-14	0.11D-8

A: 30

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	21	302	0.65D-6	0.15D-2
1D-4	41	750	0.10D-11	0.24D-5
1D-6	81	1411	0.12D-13	0.18D-7
1D-8	161	2734	0.12D-13	0.15D-9

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 36 EQUALLY SPACED POINTS

TABLE 28

CODE: HAGRON TEST PROBLEM: N2

A:1

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	7	17	0.33D-8	
1D-4	7	17	0.33D-8	
1D-6	7	22	0.95D-13	
1D-8	7	24	0.95D-13	

A:10

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	7	23	0.93D-5	
1D-4	15	114	0.44D-7	
1D-6	42	250	0.49D-10	
1D-8	82	489	0.13D-12	

A:20

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	17	75	0.87D-5	
1D-4	29	148	0.46D-7	
1D-6	72	486	0.17D-9	
1D-8	164	989	0.12D-11	

A:30

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1D-2	25	150	0.11D-5	
1D-4	49	415	0.45D-8	
1D-6	87	551	0.31D-9	
1D-8	237	1717	0.11D-11	

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 36 EQUALLY SPACED POINTS

TABLE 29

CODE: MUTS TEST PROBLEM: N2

A: 1

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1 D-2	20	229	0.58D-2	
1 D-4	21	435	0.44D-11	
1 D-6	21	554	0.44D-11	
1 D-8	21	554	0.44D-11	

A: 10

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1 D-2	300	6526	1.45	
1 D-4	300	6526	1.45	
1 D-6	300	6526	1.45	
1 D-8	300	6526	1.45	

A: 20

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1 D-2	301	6547	1.76	
1 D-4	301	8156	1.76	
1 D-6	301	9687	1.76	
1 D-8	301	11315	1.76	

A: 30

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1 D-2	301	6556	1.84	
1 D-4	301	8165	1.84	
1 D-6	301	9800	1.84	
1 D-8	302	11372	1.84	

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 30

CODE: COLNEW TEST PROBLEM: N3

EPS: 1 D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1 D-2	81	4167	0.14D-10	0.46D-9
1 D-4	81	4401	0.19D-8	0.23D-8
1 D-6	81	4432	0.19D-8	0.23D-8
1 D-8	81	4427	0.19D-8	0.23D-8

EPS: 1 D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1 D-2	11	159	0.28D-1	0.24
1 D-4	15	368	0.15D-4	0.56D-4
1 D-6	21	455	0.18D-5	0.32D-5
1 D-8	113	1467	0.33D-11	0.61D-10

EPS: 1 D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1 D-2	11	101	0.38D-1	0.45
1 D-4	33	528	0.44D-5	0.12D-11
1 D-6	33	747	0.67D-7	0.13D-6
1 D-8	71	1208	0.11D-9	0.74D-12

EPS: 1 D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1 D-2	11	86	0.39D-1	0.45
1 D-4	65	1397	0.62D-10	0.13D-18
1 D-6	39	1027	0.21D-8	0.96D-16
1 D-8	71	1562	0.84D-8	0.14D-7

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 31

CODE: COLSYS

TEST PROBLEM: N3

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	21	1109	0.47D-6	0.58D-6
1.D-4	21	1111	0.47D-6	0.58D-6
1.D-6	21	1111	0.47D-6	0.58D-6
1.D-8	81	1701	0.64D-9	0.45D-9

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	173	0.28D-1	0.24
1.D-4	15	432	0.15D-4	0.56D-4
1.D-6	49	816	0.12D-8	0.15D-7
1.D-8	47	1026	0.51D-7	0.91D-7

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	115	0.39D-1	0.45
1.D-4	33	898	0.11D-7	0.69D-11
1.D-6	39	826	0.22D-8	0.36D-12
1.D-8	75	1473	0.66D-10	0.45D-11

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	97	0.39D-1	0.45
1.D-4	65	1614	0.13D-8	0.22D-8
1.D-6	39	1280	0.55D-7	0.97D-7
1.D-8	53	1195	0.41D-9	0.72D-9

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 32

CODE: HAGRON

TEST PROBLEM: N3

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	25	672	0.21D-5	
1.D-4	35	964	0.11D-6	
1.D-6	52	1063	0.71D-11	
1.D-8	96	1381	0.61D-13	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	68	3255	0.81D-6	
1.D-4	171	3895	0.43D-10	
1.D-6	243	7045	0.17D-10	
1.D-8	335	9347	0.15D-12	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	1713	37230	1.00%	
1.D-4	1713	37230	1.00%	
1.D-6	1713	37230	1.00%	
1.D-8	1713	37230	1.00%	

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2			@	
1.D-4			@	
1.D-6			@	
1.D-8			@	

NOTE: FMP = THE NUMBER OF POINTS IN THE FINAL MESH
CPU = CPU TIME IN MILLISECONDS
MAX ERROR = THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* = THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 33

CODE: MUTS

TEST PROBLEM: N3

EPS: 1.D-2

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	96	3336	0.57D-9	
1.D-4	96	4044	0.57D-9	
1.D-6	229	9668	0.63D-11	
1.D-8	551	73862	0.37 SS	

EPS: 1.D-4

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2			SS	
1.D-4			SS	
1.D-6			SS	
1.D-8			SS	

EPS: 1.D-6

TOL	FMP	CPU	MAX ERROR	MAX ERROR*

EPS: 1.D-8

TOL	FMP	CPU	MAX ERROR	MAX ERROR*

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
* CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS

TABLE 34

TEST PROBLEM: N4

CODE: COLNEW

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	322	0.20D-14	0.62D-9
1.D-4	11	371	0.59D-14	0.63D-9
1.D-6	11	370	0.60D-14	0.62D-9
1.D-8	11	376	0.60D-14	0.62D-9

CODE: COLSYS

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	11	361	0.27D-14	0.62D-9
1.D-4	11	434	0.29D-14	0.62D-9
1.D-6	11	440	0.28D-14	0.62D-9
1.D-8	11	517	0.28D-14	0.62D-9

CODE: HAGRON

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	7	186	0.22D-6	
1.D-4	7	229	0.33D-6	
1.D-6	7	293	0.39D-9	
1.D-8	7	334	0.24D-11	

CODE: MUTS

TOL	FMP	CPU	MAX ERROR	MAX ERROR*
1.D-2	20	1526	0.59D-3	
1.D-4	20	2457	0.26D-8	
1.D-6	20	2453	0.26D-8	
1.D-8	20	2916	0.34D-12	

NOTE: FMP - THE NUMBER OF POINTS IN THE FINAL MESH
CPU - CPU TIME IN MILLISECONDS
MAX ERROR - THE MAXIMUM ERROR AT MESH POINTS
MAX ERROR* - THE MAXIMUM ERROR AT 30 EQUALLY SPACED POINTS