

**The Solution of  
Two Point  
Boundary Value Problems  
in a Parallel Environment**

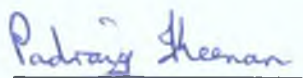
by

Padraig Keenan

A thesis submitted for the degree of Master of Science

I hearby declare that the contents of this thesis, except where otherwise stated, are based entirely on my own work which was carried out in the School of Mathematical Sciences, Dublin City University.

Signed : \_\_\_\_\_  
Dr John Carroll  
(Supervisor)

  
\_\_\_\_\_  
Padraig Keenan

This work is dedicated to Marian, Cormac, Máiréad and Fergus

## Acknowledgements

I wish to express my gratitude for the help and co-operation I received from the following people during the course of this work Colm McGuinness, whose expertise and advice proved invaluable, Joe Crean, Brendan Boulter, Colin Evans and the staff/students at Dublin City University and Regional Technical College, Athlone Special thanks to Dr John Carroll for his assistance, patience, encouragement and friendship during the project

### ABSTRACT

This work focuses on the linear two-point boundary value problem

$$y' = A(x)y + f(x), \quad a \leq x \leq b$$

$$B_a y(a) + B_b y(b) = \gamma$$

with a view to their solution by some form of parallel algorithm. The theory and practice of current sequential solution methods is reviewed to select a method which exhibits concurrent processing potential. The method selected is a version of multiple shooting. The language Ada is chosen to code the algorithm because of the features available in it, particularly the inbuilt tasking facility for concurrent processing. The efficiency of the parallel code as implemented is demonstrated by a series of numerical experiments, the results of which are summarised in tabular form.

## CONTENTS

	Page
<b>Chapter 1 Two Point Boundary Value Problems</b>	<b>1</b>
1 1 The Form of the Problem	2
1 2 Linear and Nonlinear Problems	3
1 3 Physical Examples	4
1 4 Types of Boundary Conditions	5
1 5 Existence and Uniqueness of Solutions	5
1 6 Numerical Solution Methods	8
1 7 Parallelism and Ada	12
1 8 Conclusion	14
 <b>Chapter 2 Numerical Methods for the Solution of Two Point Boundary Value Problems</b>	 <b>15</b>
2 1 Finite Difference Formulae	16
2 2 An Example of the Finite Difference Method	17
2 3 Derivative Boundary Conditions	19
2 4 Increasing the Order of Accuracy	21
2 5 Reduction of the Differential Equation to First Order	24
2 6 Finite Difference and Nonlinear Problems	26
2 7 The Finite Element Method	27
2 8 The Shooting Method	30
2 9 Conclusion	39
 <b>Chapter 3 Multiple Shooting for Two Point Boundary Value Problems</b>	 <b>40</b>
3 1 Motivation for Multiple Shooting	41
3 2 Multiple Shooting - Algorithm 1	43
3 3 Multiple Shooting - Algorithm 2	46
3 4 Adaptive Mesh Selection	49
3 5 Concurrent Processing Possibilities	51
3 6 Concurrent Processing Costs	55
3 7 Nonlinear Problems	56
3 8 Conclusion	58

	Page
<b>Chapter 4    An Ada Implementation of a                  Parallel Boundary Value Solver</b>	<b>59</b>
4 1    An Overview of the Ada Philosophy	60
4 2    Structure of the Ada Programming Language	62
4 3    Development of the Multiple Shooting Algorithm - Matrix Operations	67
4 4    The Interface to the Shooting Method Package	70
4 5    The Integrator Package	76
4 6    Conclusion	78
 <b>Chapter 5    Numerical Experiments</b>	 <b>80</b>
5 1    Criteria for Comparison	81
5 2    General Remarks on the Numerical Experiments	83
5 3    Numerical Examples	84
5 4    Conclusion	87
5 5    Summary Tables	88
 <b>Chapter 6    Conclusions and Future Work</b>	 <b>94</b>
6 1    Aims	95
6 2    Summary	96
6 3    Conclusion	96
 References	 98

## **Chapter 1**

### **Two-Point Boundary-Value Problems**



## 1.1 The Form of the Problem

In this work we consider, in the first instance, the differential equation

$$y' = f(x, y), \quad a \leq x \leq b, \quad 1.1.1a$$

$$g(y(a), y(b)) = 0 \quad 1.1.1b$$

and its solution by means of an algorithm which allows for the use of some form of concurrent processing technique. In the ordinary differential equation, i.e. Eq 1.1.1a, the quantity  $y$  is an  $n$  vector,  $x$  is the independent variable, and  $f$  is a vector function. Eq 1.1.1b represents *boundary conditions*, which are given for two points,  $a$  and  $b$ ,  $a < b$ , in the domain of  $x$ , and hence the name *two-point boundary-value problem* (BVP) is given to such a set of equations.

We shall consider only first order differential equations of the form

$$y' = f(x, y)$$

and we use the fact that higher order equations can be reduced to first order, by substitution, an example of which is now outlined.

Suppose we are given an  $n^{\text{th}}$  order differential equation in the form

$$a_{n+1}(x)y^{(n)} + a_n(x)y^{(n-1)} + \dots + a_1(x)y + a_0(x) = 0 \quad 1.1.2$$

$$a_{n+1}(x) \neq 0, \quad \forall x$$

where  $y^{(n)}$  means "the  $n^{\text{th}}$  derivative of  $y$ ", etc

Let  $y_1 = y$ ,  $y_2 = y'$ , ...,  $y_{n-1} = y^{(n-2)}$ ,  $y_n = y^{(n-1)}$ , which yields the first order system

$$y_1' = y_2$$

$$y_2' = y_3$$

$$y_n' = -(a_n y_{n-1} + \dots + a_1 y_1 + a_0) / a_{n+1},$$

{where  $a_i(x)$  has been written as  $a_i$ ,  $i = 0 \dots n+1$ , for simplicity}

In vector notation, this becomes.

$$y' = f(x, y)$$

and is reduced to the first order form as in Eq 1.1.1a. This well known technique can be applied to any higher order system to effect the same reduction.

## 1.2 Linear and Nonlinear Problems

At this point it is also useful to distinguish between linear and nonlinear problems. A linear problem is such that the elements of the vector  $y$  appear only *linearly* in Eq 1.1.1a. Consider, as an example, Eq 1.1.2, where the coefficients  $a_i$ ,  $i = 0, \dots, n+1$ , depend only on  $x$ , which can be written as

$$y' = A(x)y + f(x) \quad 1.2.1$$

where  $A(x)$  is a matrix depending only on the independent variable,  $x$ , and  $f$  is a vector function of  $x$ . Otherwise the problem is nonlinear. As an example of a nonlinear problem consider the 2nd order problem

$$y'' + \frac{1}{x}y' = \frac{k}{y^2} \quad 0 \leq x \leq 1 \quad 1.2.2$$

$$y'(0) = 0, \quad y(1) = 1$$

In first order form, using vector notation this differential equation may be written as

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} y_2 \\ -\frac{1}{x}y_2 + \frac{k}{y_1^2} \end{bmatrix}$$

i.e. in the general form

$$y' = f(x, y)$$

The solution of a linear problem poses less problems than its nonlinear counterpart, and, as we shall see, some form of linearisation technique is often used when attempting to solve a nonlinear boundary-value problem

### 1.3 Physical Examples

There are a range of physical phenomena for which two-point boundary-value problems provide the model. Examples can be found in many areas of engineering and science ranging from simple beam bending problems in mechanics to the chemical engineering areas of absorption phenomena, chemical reactions, radiation effects and problems connected with heat transfer, fluid flow, dissipation of energy and control theory. The nonlinear boundary-value problem of the previous section, i.e. Eq 1.2.2, for example, describes the equilibrium condition of suspended charged drops [17]

As a second simpler example, consider the mathematical model which describes a load

$$F(x) = 1000 \sin(\pi x/10)$$

applied to a rod that is 10m long. One end of the rod,  $x = 0$ , is clamped. At  $x = 10$  the rod is pinned. If  $\alpha = 0.01$ , where  $\alpha$  is the elastic constant, then the fourth order BVP

$$y^{(4)} = 10 \sin(\pi x/10) \quad 0 \leq x \leq 10$$

$$y(0) = 0, \quad y'(0) = 0, \quad y(10) = 0, \quad y''(10) = 0$$

models the physical problem. The solution to this problem will be used as part of the numerical experiments of Ch. 5.

Two-point boundary-value problems can also result from partial differential equations if, for example, the method of lines is chosen as the solution technique. This leads to high order differential equations. Such large systems are excellent candidates for the consideration of some form of parallelism in their solution.

## 1.4 Types of Boundary Conditions

The boundary conditions, Eq 1.1.1b, associated with the differential equation, Eq. 1.1.1a, may occur in several different forms. For example if the function  $g$ , of Eq 1.1.1b, is given only at values at either  $x = a$  or  $x = b$  then the problem becomes an initial-value problem and Eq 1.1.1b can be written as either  $y(a) = \alpha$  or  $y(b) = \beta$ , where  $\alpha, \beta$  are  $n$  vectors. Initial-value problems will not be considered in this work.

If the conditions can be written in the form

$$B_a y(a) = \alpha, \quad B_b y(b) = \beta,$$

where the vector  $(\alpha, \beta)^T$  is an  $n$ -vector, and  $B_a, B_b$  are appropriate matrices, the conditions are said to be *separated*.

Again, the conditions can be *mixed* between both boundary points and, if linear, may be written as

$$B_0 y(a) + B_1 y(b) = \gamma$$

where  $B_0$  and  $B_1$  are  $n \times n$  matrices and  $\gamma$  is an  $n \times 1$  vector.

Both of the above types of conditions result in computationally more advantageous situations than the general nonlinear case of the form

$$g(y(a), y(b)) = 0$$

For the general case some form of linearisation process is usually done and the solution is found iteratively. In the majority of problems, boundary conditions will be linear, either separated or mixed and examples of each of these types of boundary conditions are contained in the numerical experiments in Ch. 5.

## 1.5 Existence and Uniqueness of Solutions

We now consider the existence and uniqueness of the solution to Eq 1.1.1. Conditions for the existence and uniqueness for the first order scalar initial-value problem (IVP)

$$y' = f(x, y),$$

$$y(x_0) = y_0$$

are established in the understated theorem, accredited to the French mathematician Emile Picard (1856-1941), the proof of which can be found in any standard text, [12], on differential equations

*Theorem 1.1.*

If

- (i)  $f(x,y)$  is continuous and has continuous partial derivatives with respect to  $y$  at each point in the region  $R$  defined by

$$|x-x_0| < a_1, |y-y_0| < a_2,$$

- (ii)  $|f(x,y)| \leq M,$

- (iii)  $h$  is the smaller of the numbers  $a_1$  and  $a_2/M,$

then there exists a unique solution of the IVP on the interval

$$|x-x_0| \leq h$$

The theorem can easily be extended to the vector case. This theorem, then, guarantees that the IVP has a unique solution on an interval

$$x_0-h \leq x \leq x_0+h$$

It should be noted, however, that the problem may have a unique solution even if all the conditions (i)-(iii) are not satisfied.

No such equivalent theorem is available for the general two-point boundary-value problem, although for the linear case conditions for existence and uniqueness and even the form of the solution itself have been established [1]. We know that the linear problem

$$y' = A(x)y + f(x), \quad a \leq x \leq b$$

$$B_a y(a) + B_b y(b) = \gamma$$

will have a unique solution if and only if the matrix

$$Q = B_a Y(a) + B_b Y(b)$$

is nonsingular, where  $Y(x)$  is a matrix of solutions satisfying

$$Y'(x) = A(x)Y(x)$$

found using a set of linearly independent initial condition vectors, i.e. the fundamental solution matrix

As an example of the care which must be taken when examining the existence, or otherwise, of a solution consider the differential equation:

$$y'' + \pi^2 y = 0, \quad 0 \leq x \leq 1,$$

which has the general solution

$$y = A \cos \pi x + B \sin \pi x$$

If the boundary conditions are given as

$$y(0) = 0, \quad y(1) = 1$$

on substitution into the general solution the equations to be solved to find the particular solution are

$$A \cdot 1 + B \cdot 0 = 0$$

$$\Rightarrow A = 0$$

and

$$A \cdot (-1) + B \cdot 0 = 1$$

$$\Rightarrow A = -1$$

which is a contradiction. There is, in fact, no solution to the differential equation on this interval given these boundary conditions. Of course, this can be verified by examining the matrix  $Q$ , as defined above and in [1], which can be shown to be singular. In fact

$$Q = \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix}$$

where the initial conditions are taken to be

$$(i) \ y(0) = 1, \ y'(0) = 0, \text{ and}$$

$$(ii) y(0) = 0, y'(0) = 1$$

In this work, we shall assume that for any particular problem the existence and/or uniqueness of the solution on the given interval is known and that the boundary conditions are also sufficient for the existence of a solution

## 1.6. Numerical Solution Methods

The next task is to classify the various methods used for the numerical solution of two-point boundary-value problems Daniel [4] views a complete method as having three aspects:

- 1 A Transformed Problem
- 2 A Discrete Model of the Transformed Problem, and
- 3 A Solution Technique for the Discrete Model

In the most usual case the Transformed Problem and the original problem coincide. However many BVP's arise in science as the variational or Euler-Lagrange equations for problems in the calculus of variations. This can lead to the problem being transformed to that of minimising an integral. In other cases BVP's can be transformed to the problem of evaluating an integral using an appropriate Green's function. See, for example, [25] for a discussion of these transformations. Such transformations lead to discrete models and solution techniques involving quadrature. It is not proposed to perform such transformation techniques in this work. Rather, methods which attempt to solve the BVP directly will be examined.

It is generally agreed [24] that the numerical solution of two-point BVP's can be divided into three main competitive classes: finite difference methods, shooting methods and finite element methods. Therefore, it seems useful to outline how the discrete model and solution technique as described in [4] is developed for each of these classes of methods.

### The Finite Difference Method

In order to solve Eq 1.1.1 by the method of finite differences, the derivative appearing in the equation, as well as in the boundary conditions, is replaced by an appropriate difference approximation. In its simplest form, the interval  $[a,b]$  is divided into  $N$  equal subintervals, each of width  $h$ , where

$$a = x_0 < x_1 < \dots < x_N = b$$

and

$$x_j = x_0 + jh, \text{ where } h = (b-a)/N \text{ and } j = 0, 1, \dots, N.$$

The solution to the differential equation at these mesh points denoted by  $y_0, y_1, \dots, y_N$  is then sought.

To derive the approximations, Taylor's series for a function of a single variable can be used as follows:

$$y_{j+1} = y_j + hy'_j + O(h^2)$$

$$\Rightarrow y'_j = \frac{y_{j+1} - y_j}{h} + O(h)$$

This yields the simplest finite difference approximation for the (scalar) first derivative, the forward difference approximation. This formula can be extended to cover the vector case quite easily, or, indeed, for higher derivatives if required, so that the complete differential equation and boundary conditions can be replaced by a difference equation. The discrete model is then a difference equation which must be solved to give an approximation to the solution  $y(x)$  at the points  $x_1, x_2, \dots, x_N$ , represented by  $y_1, y_2, \dots, y_N$ . A number of techniques exist for the solution of such equations.

A complete treatment of this method including the introduction of higher order approximations will be given in Ch. 2. Similarly the form of the difference equation, the solution techniques and the possibility for parallelism in any numerical solution technique will be treated at that point.

### The Finite Element Method:

Another approach is to discretise the differential equation using a technique which has several variants known by such names as the finite element method, projection method, Galerkin's method, the Rayleigh-Ritz method etc. The common approach adopted by these methods is to attempt to approximate the solution curve of the differential equation 1.1.1a using



m (finite) linear combinations of known functions. These known functions, called basis or trial functions, are usually low order polynomial or simple trigonometric functions.

In simple form we approximate the solution  $y(x)$  to Eq 1.1.1 by

$$y(x) \approx v(x) = \sum_{i=1}^m c_i \phi_i(x)$$

where the  $\phi_i$  are the selected basis functions which satisfy the boundary conditions 1.1.1b, and the  $c_i$  are the coefficients which must be found.

There are several approaches to finding the coefficients. For example, in the collocation method we require that the approximate solution satisfies Eq. 1.1.1a at  $N$  internal points (grid points). By substituting

$$\sum_{i=1}^m c_i \phi_i(x)$$

for  $y$  in the original differential equation, a system of equations in  $c_i$  is constructed. The solution of the BVP then reduces to the solution of this system in  $c_i$ .

This approach and others, using residual functions, variational methods and splines, will be discussed in Ch. 2. Also the form of the solution technique and the opportunities for parallelism will be treated at this point.

### The Shooting Method .

The general principle underlying the shooting method is the transformation of the boundary value problem to an initial-value problem (IVP). To do this, it is necessary to supply an estimate for any "missing" boundary conditions at, say,  $x = a$ , using a priori information about the problem for the estimates, if possible. The result is an initial-value problem which can then be discretised and solved using any of the standard IVP methods. By comparing the known boundary conditions at  $x = b$  with the solution of the IVP at  $x = b$ , another, hopefully better, estimate of the missing boundary conditions at  $x = a$  can be found.

In the 2nd order case, with separated boundary conditions, say,  $y(a) = \alpha$  and  $y(b) = \beta$ , the procedure can be simply illustrated as follows:

- (i) Solve the differential equation, first with  $y'(a) = \alpha_1$ , to find a solution at  $x = b$ , say  $y_{(1)}(b) = \beta_1$   
Solve again with  $y'(a) = \alpha_2$ , to find another solution at  $x = b$ , say  $y_{(2)}(b) = \beta_2$
- (ii) Interpolate with these solutions  $\beta_1$  and  $\beta_2$  to produce a better estimate for  $y(a)$ , say  $y_{(3)}(a) = \alpha_3$
- (iii) Continue this process until the value  $y_{(i)}(b) = \beta_i$ ,  $i > 2$ , so found is "close" to the correct value  $y(b) = \beta$

The interpolation can be done using any linear interpolation technique For example, the Lagrangian formula would become:

$$\alpha_3 = \frac{\beta - \beta_1}{\beta_2 - \beta_1} \alpha_2 + \frac{\beta - \beta_2}{\beta_1 - \beta_2} \alpha_1$$

The abundance of available initial-value codes is clearly a reason for considering the shooting approach to the solution of a BVP In the case of linear 2nd order boundary-value problems convergence to the true solution can be shown to occur by performing (i) and (ii) above and then interpolating, although for nonlinear problems convergence may take several steps Higher order interpolation may assist the convergence process as more solutions become available

The fact that the equation must be integrated at least twice before any interpolation can be done suggests that some form of concurrent processing could be introduced, even in this simple case In the case of higher order problems, as we shall see, many integrations may be required, even in the linear case, depending on the number of "missing" boundary conditions which must be estimated so concurrent processing capabilities could lead to even greater efficiencies This method, then, suggests itself as a method worth considering in any concurrent processing context

A disadvantage of the method is that for some problems the selection of the initial conditions can be critical. An estimate of the boundary conditions very close to the true value may be required, if the solution is not to "explode" in the given interval Several methods have been proposed to control this phenomenon, including the multiple shooting technique, [13], which divides the interval  $[a,b]$  into smaller parts and the integration of each section is done separately. This approach leads inevitably to the idea of parallel multiple shooting, where the subintervals are integrated simultaneously

In the shooting method, then, the discrete model and solution technique depend on the initial-value solution technique chosen. The method presents inherent possibilities for parallelism which could take advantage of the current generation of parallel computer architecture. A complete treatment of the theory of the shooting method is found in Ch. 2.

## 1.7 Parallelism and Ada

Since this work is primarily concerned with implementing, in parallel, solutions to boundary-value problems, it seems worthwhile at this stage to review the current state of hardware and software in the field of concurrent processing. The classical Von Neuman architecture of sequential computers allows for the execution of only one instruction at any one time. However, attempts have been made to introduce limited parallelism even into this structure.

The idea of virtual memory, whereby only the active part of a large program is stored in main memory, while the rest of the program is kept in backing store was probably the first attempt to improve the efficiency of the basic architecture. Introduced by Kilburn et al in 1962, [11] and used initially in the Atlas machine, it allowed for hitherto overly large programs to be executed. The idea is also used in the time-sharing environment, allowing users to run large programs simultaneously. A similar idea is that of cache memory, whereby a very high speed area of main memory is used to process the active part of the current program, while the rest of the current program is held in lower speed main memory (Wilkes, 1965).

The first attempt at truly concurrent processing is the idea of pipelining. This is a technique to initiate one or more accesses to memory, while executing instructions in the central processor. Thus a series of instructions are held in a pipeline, and executed rapidly as the processor becomes available. Once the pipeline is full, the relatively slow operation of finding the next instruction, decoding it and possibly finding the associated data no longer becomes a bottle neck and the processor is used more efficiently. Many machines now use this idea pioneered on the IBM Stretch, CDC 6600 and now Intel's 8086 processors.

However, any serious attempt at parallel processing must aim at an array, of some kind, of several processors. Only then can there be major advances in the power of the resulting computer. As yet no single architecture has emerged to challenge for supremacy in the field of parallel processing as does the Von Neuman style in the sequential case. The answers to many questions regarding the best arrangement of processors are by no means clear-cut. These include, among others, the cost, in

computer time, of communicating between processors, the amount of memory to be associated with each processor, the amount, if any, of shared memory, the merits of having one powerful master processor, etc. These and other problems are addressed in Ch. 5, when the most efficient hardware design for the parallel algorithm of Ch. 3 is considered.

As well as the hardware problems associated with concurrent processing, we must also consider which language is best suited to the coding of any suitable algorithm. There are several possible choices, including a parallel version of Fortran or C, but one language which was designed with parallel processing in mind is Ada. Ada is a large language which addresses many issues relevant to the programming of practical systems in the real world [26]. Some of the features which contributed to its choice for the coding of the algorithm of Ch. 3 were

(i) Readability and Maintainability

In general, parallel processing is only considered when a large scale or complex problem is to be solved. Thus the program will be large and any language which is used for coding the problem must allow for ease of maintenance. Because Ada is a highly structured language which uses object oriented programming techniques, it encourages the low level details of the implementation of an algorithm to be kept invisible to the user, allowing the problem to be considered at its outermost level. Developments and refinements of the algorithm can thus be implemented at this outer level. Alternatively, more efficient processing techniques can be introduced at the lower level, without any need to change the overall structure of the algorithm.

(ii) Mechanism for Encapsulation

This allows each component of the program to be separately written, compiled and, most importantly, tested. It can then be included in a library and used confidently any time by the main program. Selected components can also be included from other libraries, when available, allowing for improved efficiency.

(iii) Tasking Facility

Since it is proposed that the program be written as a collection of parallel activities, it is essential that the language allows for this idea. In Ada the tasking facility was designed within the language with parallelism in mind, rather than as a feature which is added to certain

implementations Problems associated with the co-ordination of data transfer and the synchronisation of concurrent processes are automatically handled by the language This feature is the most important reason for the selection of Ada as the programming language for coding the algorithm

(v) Generic Units.

To allow a program to be truly general purpose, Ada allows a unit to be written with not only variables as parameters, but also with functions as parameters This is then used as a template by a driver program which supplies the functions and parameters proper to any particular problem. Effectively a copy of the template is produced by the driver program which will include the required user supplied parameters and functions.

A complete review of the relevant Ada facilities used by the algorithm of Ch. 3 is presented with the code in Ch. 4

## 1.8 Conclusion

The problem to be solved, then, is a two-point boundary-value problem whose format is to be fairly general Because of the variety of BVP's, no single code can hope to be used for all such problems However, a selection of problems will be considered, in particular linear problems with general boundary conditions. The solution to this type of problem is useful because it may be used as the core method during each iteration in the solution of a nonlinear problem Improvement in performance will be sought on the basis of speed with no loss of accuracy in the solution The language Ada will be used as the vehicle for coding the algorithm, and the spirit and philosophy of this language will be followed.

The next task is to analyse in detail the competitive numerical techniques for the solution of two-point boundary-value problems and select one which offers the greatest potential for parallelism This is addressed in Ch. 2

**Chapter 2**  
**Numerical Methods for the Solution of Two-Point**  
**Boundary -Value Problems**

## 2.1 Finite Difference Formulae

Since the main algorithm in this work is based in part on the finite difference method, it seems appropriate to present first of all the details of this method. Techniques for improving its efficiency will be mentioned and used as required in the later part of the work.

For the direct numerical solution of a two-point boundary-value problem such as Eq 1.1.1 by the method of finite differences we divide the interval  $[a,b]$  into  $N$  intervals of length  $h$ , and introduce the mesh points

$$x_j = a + jh, \quad j = 0, 1, \dots, N$$

where  $x_0 = a$ ,  $x_N = b$  and  $N$  is an appropriate integer. A scheme is then designed to determine numbers  $y_j$  which will approximate the values  $y(x_j)$  of the true solution at the points  $x_j$ .

One way of doing this is to replace every derivative appearing in the differential equation by an appropriate finite difference approximation as mentioned in Section 1.6. In this case it is not necessary to reduce higher order equations to first order as finite difference formulae can be developed for derivatives of any order. Examples of various approximations are given below, with the order of the error in each case.

$$y'(x_j) = \frac{y_{j+1} - y_j}{h} + O(h)$$

$$y'(x_j) = \frac{y_{j+1} - y_{j-1}}{2h} + O(h^2)$$

$$y''(x_j) = \frac{y_{j+1} - 2y_j + y_{j-1}}{h^2} + O(h^2)$$

$$y'''(x_j) \approx \frac{y_{j+2} - 2y_{j+1} + 2y_{j-1} - y_{j-2}}{2h^3} + o(h^2)$$

$$y^{iv}(x_j) = \frac{y_{j+2} - 4y_{j+1} + 6y_j - 4y_{j-1} + y_{j-2}}{h^4} + O(h^2)$$

All of these formulae, plus any approximations for higher order derivatives, are readily available from combinations of Taylor's series expansions for  $y_{j\pm 1}, \dots, y_{j\pm k}$ ,  $k$  an integer depending on the the order of the derivative being approximated. Other approximations whose error is of higher order are also available, see for example [8]. In the interest of accuracy it is, of course, better to use approximations whose error is of as high an order as possible and a balance is sought between simplicity and accuracy. For many approximation schemes the error is kept to  $O(h^2)$  and techniques are available to achieve a higher order of accuracy based on the lower order solution. Two approaches to this problem, namely Richardson's method and the deferred correction idea will be discussed when appropriate.

In the finite difference method, then, the original differential equation is replaced by a difference equation. The order of the difference equation will depend on the order of the differential equation and the finite difference approximation scheme used. If the differential equation is linear, the difference equation will be linear. It is not required that the step size,  $h$ , be fixed, and for some problems a fixed step size may lead to inefficiencies. However, the main algorithm discussed in Ch. 3 uses a fixed step, which simplifies the problem of load balancing between processors in a parallel environment.

## 2.2 An Example of the Finite Difference Method

To illustrate the procedure for a simple problem, consider the second order linear differential equation given below, for which existence and uniqueness criteria are assumed,

$$y''(x) + p(x)y' + q(x)y = r(x), \quad a \leq x \leq b \quad 2.2.1a$$

$$y(a) = \alpha, \quad y(b) = \beta \quad 2.2.1b$$

Without reduction to first order and using central difference approximations throughout, the approximation, at internal mesh points, to the differential equation becomes





$$b_1 = h^2 r_1 - (1 - \frac{h}{2} p_1) \alpha$$

$$b_j = h^2 r_j, \quad j=2, \dots, N-2$$

$$b_{N-1} = h^2 q_{N-1} - (1 + \frac{h}{2} p_{N-1}) \beta$$

Of course, if higher order equations or higher order difference formulae are used, the structure of the matrix will still be banded, but the number of bands will reflect the order used. The resulting linear system can be shown to always have a unique solution [10]

### 2.3 Derivative Boundary Conditions

The form of the boundary conditions will affect the structure of the linear system as described by Eq 2.2.3. If one or both boundary conditions involve a derivative, then this, too, must be replaced by its finite difference approximation. Forward differencing may be used at  $x = a$ , and backward differencing at  $x = b$ . This will have the effect of reducing the order of accuracy to  $O(h)$ , so the more usual strategy is to use central difference formulae at each boundary, thus retaining  $O(h^2)$  accuracy.

For a second order problem, this will introduce fictitious mesh points  $x_{-1}$  and  $x_{N+1}$ . By introducing two extra difference equations in the linear system i.e. by allowing  $j$  to take values between 0 and  $N$ , the same fictitious values will be introduced. Elimination between these new equations and the boundary-value equations gives  $N+1$  equations in  $N+1$  unknowns which can then be solved.

As an example consider the differential 2.2.1a with boundary conditions given as

$$y'(a) = \alpha, \quad y'(b) = \beta. \tag{2.3.1}$$

The same discretisation process as described in Section 2.2 may be used, but the solution at  $x = a$  and  $x = b$  are now no longer known. To overcome this difficulty we may use, for the left hand side of the interval, the central difference formula

$$\frac{y_1 - y_{-1}}{2h} = \alpha \quad 2.3.2$$

which yields

$$y_{-1} = y_1 - 2h\alpha$$

and, for the right hand side of the interval the formula

$$\frac{y_{N+1} - y_{N-1}}{2h} = \beta \quad 2.3.3$$

which yields

$$y_{N+1} = y_{N-1} + 2h\beta$$

If  $j$  takes values from 0 to  $N$  instead of 1 to  $N-1$ , we gain two extra equations, and the system will now have dimension  $(N+1) \times (N+1)$ . The form of the equation to be solved will be as for Eq 2.2.2. However, the fictitious solution values  $y_{-1}$  and  $y_{N+1}$  will be introduced. These can be eliminated using Eq 2.3.2 and Eq 2.3.3, and the system can be solved at all the mesh points, i.e.

$$a = x_0, x_1, \dots, x_N = b$$

In both these second order examples the structure of the coefficient matrix will be tridiagonal and its structure may be exploited to reduce the amount of computational effort involved in its solution.

This simple example illustrates the basic principle of the finite difference method for the solution of two-point boundary-value problems. However, several important points need to be considered in conjunction with this simple scheme.

Firstly, for problems with solutions which vary rapidly over parts of the interval a large number of mesh points will be needed to accurately trace the solution. For some problems the number of mesh points required may be prohibitively high and the finite difference method may fail in these cases. As an example of such a problem consider the well known boundary layer type problems where the solution varies very rapidly near one or both boundaries. The amount of such variation may depend on a parameter in the differential equation. The solution may be well behaved over other parts of the interval, so that a small number of points (i.e. a

large step size) will achieve the same accuracy. This suggests that a step size which can somehow be allowed to vary would be more efficient in this case. One strategy for varying the mesh size is discussed in the next section (Section 2.4).

The main algorithm of this work, although using a fixed step, can go some way to alleviating the "large size" difficulty by including in the final solution a small or large number of points depending on the behaviour of the solution within a subinterval. Thus boundary layer problems may be successfully treated in parallel by the algorithm.

Again the number of mesh points and hence the size of the system of linear equations gets large as the value of  $h$ , the (constant) step length, decreases. For high order accuracy with low order difference formulae, the step length is necessarily small, so that the computational cost for the solution of the linear system is high. This problem can be partially overcome by using some technique for accelerating convergence towards the true solution, thus achieving greater accuracy without increasing the size of the linear system. This problem is also addressed in Section 2.4.

## 2.4 Increasing the Order of Accuracy.

Probably the best known method used to increase the order of accuracy is Richardson's deferred approach to the limit which operates as follows. Obtain an approximation for the true solution,  $y(x_j)$ , at the selected mesh points  $x_j$ ,  $n = 0 \dots N$ , based on a step length  $h$ , denoted as  $y(x_j, h)$ , with accuracy  $O(h^2)$ . Now use the same scheme to obtain another solution, with step length  $\rho h$ ,  $\rho < 1$ . If the (unknown) error is written as  $e(x_j)$  we may write:

$$y(x_j, h) = y(x_j) + h^2 e(x_j) + O(h^3)$$

and

$$y(x_j, \rho h) = y(x_j) + \rho^2 h^2 e(x_j) + O(h^3)$$

Eliminating  $e(x_j)$  and re-arranging gives

$$y(x) = \frac{y(x, \rho h) - \rho^2 (y(x, h))}{(1 - \rho^2)} + O(h^3)$$

where the subscript has been omitted, for notational convenience

Thus an extra order of accuracy is gained at each mesh point without the need to solve the large system with a higher order difference formula. In certain problems, where only even powers of  $h$  occur in the error term,  $O(h^4)$  can be obtained by one application of Richardson's approach. For a discussion on this theory see [10]. Many numerical schemes use  $\rho = 0.5$ , but Keller [14] suggests that a more suitable value would be an arithmetic reduction of the form

$$h_k = (k+1)^{-1} h_0$$

where  $h_k$  represents the step size after the  $k^{\text{th}}$  refinement

Another approach to the problem of achieving greater accuracy is the method of deferred correction introduced by Fox [8] and outlined in [9]. To illustrate this technique, we again consider Eq. 2.1.1. Assume an approximate solution has been found using the central difference formula above. Making use of Stirling's interpolation formula for  $y(x)$ , where  $x = x_0 + uh$ , and writing  $y_0$  for  $y(x_0)$ , we get,

$$\begin{aligned} y(x) \approx y_0 + \frac{u}{2} (\delta y_{\frac{1}{2}} + \delta y_{\frac{1}{2}}) + \frac{u^2}{2} \delta^2 y_0 + \frac{u^3 - u}{2(3!)} (\delta^3 y_{\frac{1}{2}} + \delta^3 y_{\frac{1}{2}}) \\ + \frac{u^4 - u^2}{4!} \delta^4 y_0 + \frac{u^5 - 5u^3 + 4u}{2(5!)} (\delta^5 y_{\frac{1}{2}} + \delta^5 y_{\frac{1}{2}}) + \end{aligned} \quad 2.4.1$$

where the operator  $\delta$  is the central difference operator, i.e.

$$\delta y_j = y_{j+1/2} - y_{j-1/2}$$

We may differentiate this formula by first finding the derivative at a general point  $x$ , and then using  $x = x_0 + uh$ . The work is as follows.

$$\frac{d(y(x))}{dx} = \frac{d(y(x))}{du} \cdot \frac{du}{dx}$$

but since  $x = x_0 + uh$  this leads to

$$\begin{aligned}
\frac{dx}{du} &= h \\
\Rightarrow \frac{du}{dx} &= \frac{1}{h} \\
\Rightarrow \frac{d(y(x))}{dx} &= y' = \frac{1}{h} \frac{d(y(x))}{du} \\
\Rightarrow hy' &= \frac{d(y(x))}{du}
\end{aligned}$$

Similar expressions for  $y''$  (and higher derivatives if necessary) can be found. Differentiating Eq 2.4.1 and substituting in Eq 2.2.1(a) we get, after some manipulation, the same finite difference equation as before i.e. Eq 2.2.2, with added terms involving  $\delta^3 y$ ,  $\delta^4 y$ ,  $\delta^5 y$  etc. If we call the sum of these correction terms  $C(y)$ , then using a difference table we may numerically estimate the successive differences and hence  $C(y_n)$  for each of the solution values. Adding this estimate to the approximate solution gives a better approximation and the process may be continued until all the values for  $C(y_n)$  are less than some tolerance.

A problem when finding the necessary higher differences arises near the boundary and this can be overcome by extending the solution mesh to include points external to the original mesh [14]. This deferred correction process has been found to be more effective than Richardson's approach [23] and is included as standard in some of the modern codes for the solution of two-point boundary-value problems.

The problems encountered by a fixed step finite difference method for rapidly varying solution within an interval has already been mentioned. A solution to the inefficiencies is to devise some strategy to allow a variable mesh size to be used.

One such approach, proposed by Lentini and Pereyra, [23] is to somehow monitor the local truncation error at each point in the solution. This they do by estimating numerically the first neglected term in the Taylor's series expansion, in the above scheme the term involving  $h^2$ . Of course, other terms could also be taken into account which would lead to a better estimate of the error. The first neglected term in the scheme described by Eq 2.2.2 can be shown to involve the expression

$$\frac{h^2}{12} y_j'''$$

Again having solved the difference equation on some initial mesh, an estimate to accuracy  $O(h^2)$  is formed, and, in an analogous manner to the deferred correction technique, an estimate of the first neglected term is found. An attempt to equidistribute the error by increasing the number of mesh points where the value of  $y_1'''$  is large is then made and the problem is re-solved on this new mesh. This process can be repeated until some stopping condition has been reached, e.g. too many mesh points have been selected, too many iterations have been performed or convergence to the solution has been achieved.

This is at least a two pass operation, but the extra work involved can be used for other useful purposes in the algorithm. For example, if deferred correction is to be used, the first stage of this calculation is already done, i.e. estimating  $y_1'''$ . Another possibility is to use the information about the local error estimate to gain some information about the global error estimate [7].

Another approach to the problem of a rapidly varying solution is to pay particular attention to the initial mesh of points on which the differential equation is to be solved. Quite an amount of work is being done on mesh generation and adaptive mesh techniques for both partial and ordinary differential equations but a complete treatment of such techniques is beyond the scope of this work.

## 2.5 Reduction of the Differential Equation to First Order

In the example discussed earlier, i.e. Eq 2.1.1, no attempt was made to reduce the equation to first order. If this approach were followed, as outlined in Ch 1, Eq 2.1.1 would become:

$$y_1' = y_2$$

$$y_2' = -(p(x)y_2 + q(x)y_1) + r(x)$$

or, in matrix form:

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -q(x) & -p(x) \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ r(x) \end{bmatrix}$$

Using vector notation this can be written in general as:

$$y' = A(x)y + f(x)$$

There are several schemes for discretising the equation in this form. We consider the "box" (or centred Euler or mid-point) scheme. In this method the solution is sought at x-values mid-way between the selected mesh points,  $x_0, x_1, \dots, x_N$ . The usual forward difference approximation is effectively a central difference approximation for this mid-point, and can be shown to yield a truncation error  $O(h^2)$ . The functions of  $x$  are evaluated at the mid-point, usually written as  $x_{j+1/2}$ , and the value of  $y_{j+1/2}$  is taken as the average of  $y_j$  and  $y_{j+1}$ . This gives the equation

$$\frac{y_{j+1} - y_j}{h} = A(x_{j+\frac{1}{2}}) \left( \frac{y_{j+1} + y_j}{2} \right) + f(x_{j+\frac{1}{2}})$$

On multiplication by  $h$ , this becomes

$$y_{j+1} - y_j = \frac{h}{2} A_{j+\frac{1}{2}} (y_{j+1}) + \frac{h}{2} A_{j+\frac{1}{2}} (y_j) + h f_{j+\frac{1}{2}}$$

or

$$(I - \frac{h}{2} A_{j+\frac{1}{2}}) y_{j+1} - (I + \frac{h}{2} A_{j+\frac{1}{2}}) y_j = h f_{j+\frac{1}{2}}$$

The resulting system of linear equations is of block bidiagonal form and, on inclusion of the boundary conditions, can be solved at all required mesh points as before

The dimension of each block in the coefficient matrix will depend on the dimension of the original differential equation. In this example the dimension is 2, and so they are  $2 \times 2$  blocks. On solving the system, the values for each element in the vector  $y$  is found at each mesh point, so in this case as well as the solution for  $y_1$  we also obtain the solution for  $y_2$ . There is potential for using this extra information when monitoring the truncation error where the equations are obtained by reducing the order of the differential equation as outlined in 1.1



## 2.6 Finite Difference and Nonlinear Problems

The preceding section presents the ideas of the finite difference method for the solution of linear two-point boundary-value problems with various refinements and problems inherent in the method. As yet no mention has been made of nonlinear problems. This is because the technique involved in the solution of nonlinear problems results in the solution of a series of linear problems. All the techniques used in the linear case can then be used at each stage of the nonlinear problem solution. Consider again the general boundary-value problem, Eq 1.1.1, i.e.

$$y' = f(x, y), \quad a \leq x \leq b$$

$$y(a) = \alpha, y(b) = \beta$$

When a nonlinear differential equation is approximated by a finite difference formula, the resulting difference equation will be nonlinear and may be written in the form:

$$g(x, y) = 0 \quad 2.6.1$$

This corresponds to the linear equation 2.1.2 and requires some nonlinear technique in order to find its solution. The usual approach adopted is to use Newton's method, or some variation of Newton's method, where the order of the method matches the order of the finite difference approximation. In outline, Newton's method involves writing a linear approximation to the left hand side of Eq 2.6.1, using a Taylor's series expansion, and solving the resulting linear system. So we can write

$$g(x, y) = 0$$

$$\Rightarrow g(x, y) \approx g(x, y^0) + \frac{\partial g(x, y^0)}{\partial y} \Delta y +$$

where  $y^0$  is a vector of values "close" to the true solution,  $\Delta y$  is  $y - y^0$ , and the derivative term is the well known Jacobian matrix, often written as  $J$ . The problem, then, is to solve

$$g(x, y^0) + J(y - y^0) \approx 0$$

$$\Rightarrow J(y-y^0) \approx -g(x,y^0)$$

$$\Rightarrow y \approx y^0 - J^{-1}(g(x,y^0))$$

A recursion may be set up such that a new value of  $y$  is found according to the above formula and this new value used in the next approximation. The recursion can be stopped when the difference between successive approximations for  $y$  is sufficiently small.

One problem with this method is to find a starting vector,  $y^0$ , "close" to the true solution. There are several strategies used for this, for example, solving a simpler (linear) problem close to the actual problem and then using this solution to start the solution to the nonlinear problem. Again continuation techniques can be used. To do this a parameter, say  $\lambda$ , is introduced into the differential equation with the objective of simplifying the problem for  $\lambda = 0$ . This simple problem can then be solved and the value of  $\lambda$  increased by some step, thus forming a new problem. The solution found may be used as the approximate solution to the new problem and the process continued until the original differential equation is recovered for  $\lambda = 1$ .

Because of the linearisation process usually adopted for the numerical solution of nonlinear boundary-value problems, the search in this work for an efficient parallel algorithm concentrates in the first instance, on linear problems. The intention would be to include this parallel routine as the core integrator for the general class of nonlinear problems in the future.

The main computational cost in the pure finite difference method is the linear algebra routine and the potential for parallelism in the method depends on the ability to parallelise this routine. Since the main algorithm in this work involves using a linear algebra routine, some current work in this area will be reviewed in Ch 3.

## **2.7 The Finite Element Method.**

Although the finite element method is not used in the main algorithm in this work, it does not seem appropriate to ignore consideration of this well known approach to the solution of two-point boundary-value problems. Only an outline of the finite element method will be presented here and the potential for parallelism in this area deserves more attention and it will remain as an area for further study.

The idea behind the finite element or projection method is to somehow make a (linear) combination of known functions, satisfying the boundary conditions, which represent the true solution in the given

interval. These known functions, called basis or trial functions, are usually simple polynomial or trigonometric functions. If we regard the true solution as lying in some appropriate (infinite dimensional) space, the solution obtained can be viewed as a finite dimensional approximation to the true solution.

To illustrate the basic idea, consider again Eq. 2.2.1. We attempt to approximate the solution  $y(x)$  by a linear combination of  $m$  functions, i.e.

$$y(x) \approx v(x) = \sum_{i=1}^m c_i \phi_i(x)$$

where the  $\phi_i(x)$  are basis functions, each of which satisfies the given boundary conditions, and the  $c_i$  are coefficients as yet unknown.

We must now decide in what sense the function  $v(x)$  is to approximate the true solution. The method of *collocation* requires that the approximate solution satisfies the differential equation on a set of  $N$  grid points, not necessarily equally spaced, say  $x_j, j=1, \dots, N$ . In the linear second order example already given, i.e.

$$y'' + p(x)y' + q(x)y = r(x), \quad a \leq x \leq b$$

$$y(a) = \alpha, \quad y(b) = \beta,$$

it means that the equation can be written as

$$\sum_{i=1}^m \left[ \frac{d^2}{dx^2} (c_i \phi_i(x_j)) + p(x_j) \frac{d}{dx} (c_i \phi_i(x_j)) + q(x_j) c_i \phi_i(x_j) \right] = r(x_j) \\ j = 1, \dots, N$$

Assuming that the basis functions are twice differentiable, this is a linear equation in the  $c_i$  and can be written simply as

$$Ac = b$$

where  $A$  is the matrix of coefficients of the  $c_i$ , and  $b$  is the known vector of the  $r(x_j)$ . The coefficient matrix  $A$  and the right hand side  $b$  are easily constructed once the values of  $x_j$  are selected and so the system can be solved.

In the *Galerkin* method the approach adopted for evaluating the  $c_i$  is as follows. Define a residual function  $w(x)$  for the approximate solution  $v(x)$  as

$$w(x) = v''(x) + p(x)v'(x) + q(x)v(x) - r(x), \quad a \leq x \leq b$$

If  $v(x)$  were the exact solution then  $w(x)$  would be identically zero. Using the definition of orthogonal functions, i.e. two functions  $f_1$  and  $f_2$  are orthogonal in an interval  $[a, b]$  if

$$\int_a^b f_1(x)f_2(x)dx = 0$$

then the residual function would be orthogonal to every function on the interval. The solution  $v(x)$  is not the true solution but approximates it using a linear combination of the basis functions, so the Galerkin method aims at choosing that  $v(x)$  which makes  $w(x)$  orthogonal to all the basis functions  $\phi_1, \dots, \phi_m$ . For the example above this condition can be written as

$$\int_a^b [v''(x) + p(x)v'(x) + q(x)v(x) - r(x)] \phi_k(x) dx = 0$$

$$k = 1, \dots, m.$$

Writing  $v(x)$  as the linear combination, i.e.

$$v(x) = \sum_{i=1}^m c_i \phi_i(x)$$

the integral equation can be written as

$$\sum_{i=1}^m c_i \int_a^b [\phi_i''(x) + p(x)\phi_i'(x) + q(x)\phi_i(x)] \phi_k(x) dx = \int_a^b r(x) \phi_k(x) dx$$

$$k = 1, \dots, m$$

This again is a linear equation in the  $c_i$  of the form

$$Ac = f$$

where the elements of  $A$  are constructed by evaluating the integral on the left hand side, and the elements of  $f$ , by evaluating the integral on the right hand side

The evaluation of the elements is more complicated than in the collocation method because of the integration involved. Only in the case of simple basis functions can this integration be done explicitly so that some form of numerical integration will usually be required [19]. For nonlinear problems the same approach can be adopted and the resulting system of equations will be nonlinear. Again some iterative scheme would be used in the solution of these equations.

The preceding section outlines briefly the finite element method for the solution of two-point boundary-value problems. As already stated, a full treatment of the method is not relevant to this work. It should be noted, however, that since this method involves the solution of a system of equations in some form, the use of concurrent processing techniques will depend on the development of parallel algorithms for the solution of such systems.

## 2.8 The Shooting Method

When beginning the study of two-point boundary-value problems with a view to their solution using supercomputers, the method which seemed to exhibit inherent concurrency was the shooting method. For this reason the method was chosen as the area for greatest investigation. This section of the work deals with the theory and practice of what is known as the simple shooting method, with an introduction to the idea of multiple or parallel shooting. Ch 3 will examine in detail two possible approaches to implementing multiple shooting on some form of concurrent processing machine.

For a simple way of viewing the shooting method for solving two-point boundary-value problems, consider the 2nd order problem

$$y''(x) = f(x, y, y') \quad a \leq x \leq b$$

$$y(a) = \alpha, \quad y(b) = \beta$$

Suppose we supply the "missing" boundary condition at one end of the interval, say choose

$$y'(a) = \alpha_1$$

and ignore for the moment the given condition at  $x = b$ . This converts the problem to a new initial-value problem and we may solve this to find the solution at  $x = b$ . We may write this as

$$y(b, \alpha_1) = \beta_1$$

where  $\alpha_1$  is included to show that the solution is dependent on the assumed boundary condition, i.e.  $y'(a) = \alpha_1$ .

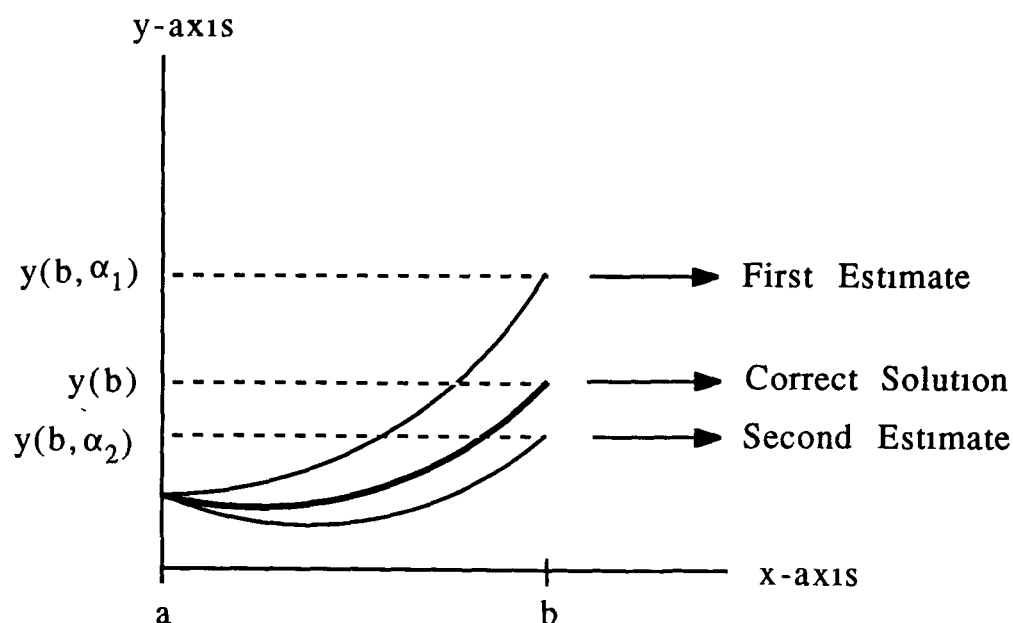
In other words, we "shoot" across the interval  $[a, b]$  to find a value for  $y(b, \alpha_1)$ . If we repeat this process for another assumed value of  $y'(a)$ , say

$$y'(a) = \alpha_2$$

we can find

$$y(b, \alpha_2) = \beta_2$$

We may now compare our two solutions,  $\beta_1$  and  $\beta_2$ , to the given condition at  $x = b$ , i.e.  $y(b) = \beta$ . Using interpolation we can hopefully find a better estimate for  $y'(a)$ , say  $\alpha_3$ , which will give a solution at  $x = b$  which better matches the given condition for  $y(b)$  (see diagram).



By repeating this process we can, in theory, find the correct value of  $y'(a)$  as accurately as required

As an actual example consider the analytic solution to the simple problem

$$y'' = 6x \quad 0 \leq x \leq 1$$

$$y(0) = 0, \quad y(1) = 1$$

As a first estimate take  $y'(0) = 1$ , which, on integrating twice and substituting the initial conditions, yields the solution

$$y = x^3 + x$$

This solution gives a value for  $y$  at  $x = 1$  of  $y(1) = 2$

If we then take  $y'(0) = 0.5$ , the solution is

$$y = x^3 + 0.5x$$

which gives, at  $x = 1$ , the result  $y(1) = 1.5$ . Applying the simple interpolation formula from Section 1.6, a better estimate for  $y'(0)$  will be

$$y'(0) = 0$$

which is, in fact, the correct value for this problem. Thus the correct solution is obtainable using two "shots". This is not surprising, as it can be shown that for linear second order problems the correct value can always be found in this way.

Such an approach would seem to be appropriate to solve linear or nonlinear problems and the availability of accurate numerical methods for the solution of initial-value problems provides the motivation for further investigation of the method.

If the problem is of a higher order there may be several conditions "missing" at either end of the interval  $[a, b]$ . This means that the simple interpolation formula outlined in Section 1.6 will no longer be sufficient and the problem will have to be solved more than twice to find a better estimate at  $x = a$ .

In order to illustrate this idea and to put the theory on a sounder footing, it is worth investigating in more detail the continuous or analytic shooting technique in the first instance [14]

The general superposition procedure for solving a linear non-homogeneous initial-value differential equation is:

- (a) find a particular solution to the problem,
- (b) solve the corresponding homogeneous problem with linearly independent initial conditions

The required solution will then be the sum of the particular solution and a combination of the solutions to the homogeneous problem

If this approach is pursued in solving a  $n^{\text{th}}$  order linear boundary-value problem, with linear boundary conditions i.e.,

$$y' = Ay + f, \quad a \leq x \leq b$$

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

we must find a particular solution to the differential equation and then find  $n$  solutions to the corresponding homogeneous equation with  $n$  linearly independent initial conditions. The solution to the boundary value problem can then be found from the requirement that the solution must satisfy the given boundary conditions. Mathematically, using the usual notation, this consists of solving the  $n+1$  initial-value problems

$$y_0' = Ay_0 + f, \quad y_0(a) = \gamma$$

$$\text{and} \quad y_v' = Ay_v, \quad y_v(a) = e_v, \quad v = 1, \dots, n$$

where  $\gamma$  is used to represent any vector of initial conditions, and the vectors  $e_v$  are the usual unit vectors in  $n$ -dimensional space. The required solution can then be written as

$$y = y_0 + \sum_{v=1}^n c_v y_v$$

where  $c_v$  represents the coefficients in the linear combination of solutions. In simple form this becomes

$$y = y_0 + Yc \quad 2.8.1$$



where  $Y$  is an  $n \times n$  matrix of the solutions  $y_n$ , as mentioned earlier, called the fundamental solution matrix and  $c$  is a vector of coefficients. Eq 2.8.1 must satisfy the given boundary conditions

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

so, on substitution, this leads to the system of equations

$$B_a[y_0(a) + Y(a)c] + B_b[y_0(b) + Y(b)c] = \beta$$

Re-arranging and substituting the boundary conditions as appropriate we get

$$Qc = [B_a + B_b Y(b)]c = \beta - B_a y_0 - B_b y_0(b)$$

This, of course, is a linear system of equations which can be solved to find the coefficients  $c_1, \dots, c_n$ , and the solution  $y(x)$  can then be constructed using these coefficients

In order for a solution to the linear system to exist, the coefficient matrix  $Q$  must be non-singular, and indeed this is another statement of the existence theorem for linear two-point boundary-value problems [14], [1].

Thus, by solving  $n+1$  initial-value problems, the original differential equation has now been reduced to an algebraic system, which can more easily be solved (If  $Q$  is ill-conditioned, i.e. nearly singular, the solution of the linear system by numerical methods may be difficult and examples where this occurs will be presented later)

The above is the essence of analytic shooting and the numerical shooting method consists of following the same procedure, except that the solutions to the initial-value problems are found numerically. It can be shown that if a stable, order  $m$  initial-value method is used to solve the initial-value problems, then  $O(h^m)$  accuracy can be achieved using the shooting method [14] [10]

To carry on the general second order problem presented earlier, i.e. Eq 2.2.1, the shooting method solution would be found as follows

- 1) Re-write the equation in first order form

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -q(x) & -p(x) \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ r(x) \end{bmatrix}$$

$$\Rightarrow \mathbf{y}' = \mathbf{A}\mathbf{y} + \mathbf{f}$$

- 2) Find the solution,  $\mathbf{y}_0$ , to the problem with

$$\mathbf{y}(a) = \boldsymbol{\gamma}$$

where  $\boldsymbol{\gamma}$  is any vector of initial conditions

- 3) Find the solution  $\mathbf{y}_1$  and  $\mathbf{y}_2$  to the initial-value problem

$$\mathbf{y}' = \mathbf{A}\mathbf{y}$$

with

$$(i) \mathbf{y}(a) = \mathbf{e}_1, \text{ and}$$

$$(ii) \mathbf{y}(a) = \mathbf{e}_2$$

The actual solution,  $\mathbf{y}$ , could then be written as

$$\mathbf{y} = \mathbf{y}_0 + c_1\mathbf{y}_1 + c_2\mathbf{y}_2$$

Introducing the boundary conditions as given in Eq 2.2.1(b) the system of equations to be solved will be

$$(\mathbf{B}_a + \mathbf{B}_b[\mathbf{y}_1(b), \mathbf{y}_2(b)])(c_1, c_2)^T = [\alpha, \beta]^T - \mathbf{B}_a\boldsymbol{\gamma} - \mathbf{B}_b\mathbf{y}_0(b)$$

So, the original BVP has been reduced to a linear system of equations by solving 3 initial value problems, possibly numerically. In the context of parallel processing there is no reason why these 3 IVPs should not be solved simultaneously.

However for nonlinear problems we cannot superpose solutions so that this theory would not seem to be available for the nonlinear case. It can be shown, [9] however, that the shooting method will work for such problems and in order to illustrate this, we will first consider the general second order problem

$$y_1' = f_1(x, y_1, y_2)$$

$$y_2' = f_2(x, y_1, y_2) \quad a \leq x \leq b,$$

$$g_m(y_1(a), y_1(b), y_2(a), y_2(b)) = 0, \quad m = 1, 2$$

If we choose  $y_1(a) = \lambda$  and  $y_2(a) = \mu$ , we again have an initial-value problem which can be solved numerically by forward integration from  $x = a$  to  $x = b$ , to find  $y_1(b; \lambda, \mu)$  and  $y_2(b; \lambda, \mu)$  as before. The correct values of  $\lambda$  and  $\mu$  are those values which satisfy the boundary conditions, i.e. which satisfy the equations

$$g_m(\lambda, \mu, y_1(b, \lambda, \mu), y_2(b, \lambda, \mu)) = 0, \quad m = 1, 2$$

These two equations will be, in the general case, nonlinear and can be solved using Newton's method. This implies that initial estimates  $\lambda_0$  and  $\mu_0$ , for  $\lambda$  and  $\mu$ , are known and better estimates  $\lambda_1$  and  $\mu_1$  can be found according to the equations:

$$\Delta \lambda_0 \frac{\partial g_1}{\partial \lambda} + \Delta \mu_0 \frac{\partial g_1}{\partial \mu} + g_1(\lambda_0, \mu_0) = 0$$

$$\Delta \lambda_0 \frac{\partial g_2}{\partial \lambda} + \Delta \mu_0 \frac{\partial g_2}{\partial \mu} + g_2(\lambda_0, \mu_0) = 0$$

where the derivatives of  $g_1$  and  $g_2$  are evaluated at  $\lambda_0$  and  $\mu_0$ . Since  $g_1$  and  $g_2$  are functions of  $y_1(b, \lambda, \mu)$  and  $y_2(b, \lambda, \mu)$ , the calculation of the partial derivatives in the above equations will involve the evaluation of

$$\frac{\partial y_1}{\partial \lambda}, \frac{\partial y_1}{\partial \mu}, \frac{\partial y_2}{\partial \lambda}, \frac{\partial y_2}{\partial \mu}$$

all evaluated at  $x = b$ . If we call these quantities  $k, l, m$  and  $n$ , respectively, we can most easily evaluate them by first differentiating the differential equation and boundary conditions with respect to  $\lambda$  and  $\mu$  as follows

$$\frac{\partial(y_1')}{\partial \lambda} = \frac{\partial f_1}{\partial \lambda} = \frac{\partial f_1}{\partial y_1} \frac{\partial y_1}{\partial \lambda} + \frac{\partial f_1}{\partial y_2} \frac{\partial y_2}{\partial \lambda}$$

$$\Rightarrow k = \frac{\partial f_1}{\partial y_1} + l \frac{\partial f_1}{\partial y_2}$$

Similarly other linear differential equations, called the variational equations, can be developed for  $l'$ ,  $m'$ , and  $n'$ , and they are stated hereunder

$$l' = k \frac{\partial f_2}{\partial y_1} + l \frac{\partial f_2}{\partial y_2}$$

$$m' = m \frac{\partial f_1}{\partial y_1} + n \frac{\partial f_1}{\partial y_2}$$

$$n' = m \frac{\partial f_2}{\partial y_1} + n \frac{\partial f_2}{\partial y_2}$$

Initial conditions for this system of equations can easily be shown to be  $k(a) = 1$ ,  $l(a) = 0$ ,  $m(a) = 0$  and  $n(a) = 1$ . By forward integration to  $x = b$ , the required values of  $k(b, \lambda_0, \mu_0)$ , etc, can be found.

The nonlinear system can now be solved to find  $\Delta\lambda_0$ , and  $\Delta\mu_0$ , and the usual iterative scheme with  $\lambda_1 = \lambda_0 + \Delta\lambda_0$  can be set up. It can be seen that two second order initial-value equations must be integrated to find the coefficients for each step of the Newton scheme. For simplicity of coding, approximations to these coefficients may be found by solving the original equations with small changes in  $\lambda$  and  $\mu$  and approximating the derivative by a difference formula. Some of the power of the Newton scheme will, however, be lost if this approach is adopted. On the other hand, the advantages include relieving the user of the chore of having to supply analytic expressions for the derivatives and having to fully understand the entire solution process.

A closer examination of the treatment of nonlinear equations reveals that this approach reduces to superposition for the linear case. The variational equations are equivalent to the equations for the fundamental matrix of solutions in the linear method [9]. The theory, briefly outlined above, shows that the shooting method can be applied to both linear and nonlinear BVPs, which justifies its numerical application for each type of problem. For further discussion of this theory the reader is referred to [9].

There are many problems which need to be understood when considering the shooting method as a means of solving two-point boundary-value problems. Even with relatively simple looking linear problems numerical simple shooting may not be a realistic option. The main reason for this is the fact that the initial estimate for the missing conditions at  $x = a$ , say, may be critical. In some problems, if the estimate is not sufficiently "close", integration from  $a$  to  $b$  may fail because, while the

boundary-value problem may be well behaved, the artificially manufactured initial-value problem may not be, at least in the desired range. The solution may introduce a true singularity, or explode and cause overflow on the machine before the end of the range is reached.

Many solutions to this problem have been suggested including shooting-splitting, continuation, and multiple shooting. The first of these, proposed by Firnett and Troesch [6], involves monitoring the distance across the range a particular initial-value technique will solve and whether the overflow condition is positive or negative, for different initial guesses. When upward and downward diverging solutions are found for a particular guess at some internal point, say  $x_1$ , the initial guess is refined using simple bisection and the process continued until the solutions differ by less than some tolerance. These values are then used to start the integration along the next interval,  $[x_1, x_2]$  and the process continued until  $x = b$  is reached. The technique is simple to use and has been shown to work for a range of sensitive problems.

The method of continuation is a well known technique for solving many difficult mathematical problems. Basically it removes the difficulty and attempts to solve a simpler problem. For boundary-value problems this may mean shortening the interval  $[a, b]$  to  $[a, x_1]$  and solving the problem on the shorter interval, i.e. matching the boundary conditions at  $x_1$  rather than  $b$ . The solution obtained can then be used as a starting guess for the solution to the problem over a longer range. The amount by which the range is increased will depend on the sensitivity of the problem. This technique is "continued" until the solution has been found over the full range. It may be used with success in cases where one of the boundaries is at  $\infty$ .

Another approach to continuation for boundary-value problems is to eliminate difficult terms as outlined earlier by multiplication of the differential equation by a suitable parameter. The parameter is then varied from 0 to 1, using suitable steps, and the solution at each step is used as a starting value for the next step. The amount by which the parameter is varied, and indeed if the step should be uniform, depends on the problem being solved and requires many considerations outside the scope of this work.

Perhaps the most successful method for the solution of problems too difficult for simple shooting is the multiple shooting technique. The multiple or parallel shooting technique for the solution of two-point boundary-value problems was developed by Keller [13] in 1968 as a method for overcoming the problem of "exploding" solutions. Briefly it involves dividing the interval into a number of subintervals and simultaneously solving the boundary-value problem on each subinterval. Although the

technique was introduced before the advent of parallel computing power, because of the inherent parallel nature of the algorithm it is possible that problems requiring a major investment of computing time could be solved more efficiently by this method in parallel using an array of processors.

In this work the multiple shooting technique forms the basis of the algorithm for the solution of two-point boundary-value problems in a parallel environment and the background of two approaches to the method, their advantages and disadvantages, will be examined in Ch 3 and an appropriate algorithm for implementation is selected in Ch 4.

## **2.9 Conclusion**

The above sections outline the standard techniques available for the solution of two-point boundary-value problems. The three main methods are included and these can be classified as finite difference, finite element and shooting methods. Each of these areas is well developed in a sequential environment, with various subclasses within each of them. Their development into a parallel environment is ongoing and as a contribution to that process the potential for parallelism using some form of multiple shooting will be examined in Ch 3.

## **Chapter 3**

### **Multiple Shooting for Two-Point Boundary-Value Problems**

### 3.1 Motivation for Multiple Shooting

The multiple shooting method for the solution of two-point boundary-value problems, as stated earlier, owes its development to Keller in the 1960's. The motivation for his work was to alleviate the difficulties which the "ordinary" (simple) shooting method often encounters when the solution of Eq 1.1.1 grows rapidly in the interval  $[a,b]$ . To illustrate this difficulty, consider the linear 2nd order problem [19]

$$\begin{aligned}y'' - 100y &= 0, & 0 \leq x \leq 1, \\ y(0) &= 1, & y(1) = 0\end{aligned}$$

The solution is of the form

$$y = Ae^{-10x} + Be^{10x}$$

where A and B may be found from the boundary conditions and can be shown to be

$$A = \frac{1}{1 - e^{-20}} \quad \text{and} \quad B = \frac{1}{1 - e^{20}}$$

The correct value for  $y'(0)$  is then

$$y'(0) = \frac{-10}{1 - e^{-20}} + \frac{10e^{-20}}{e^{-20} - 1} \approx -10$$

If this problem is solved as an initial-value problem, taking

$$y'(0) = s,$$

the solution will depend on the choice of value for  $s$ . In fact the solution of the initial-value problem will depend on  $s$  as follows.



$$y = \frac{10 - s}{20} e^{-10x} + \frac{10 + s}{20} e^{10x}$$

This expression is very sensitive to the choice of  $s$ , for if

$$y'(0) = s = -10.1$$

is used, the solution at  $x = 1$  turns out to be

$$y(1) \approx -110$$

whereas if

$$y'(0) = s = -9.9$$

is used the solution at  $x = 1$  is

$$y(1) \approx +110$$

The dependence of  $y'(1)$  on the chosen value of  $s$  is even more dramatic. Of course any errors introduced by an (approximate) numerical method will be similarly magnified and may lead to numerical instability.

This simple linear example demonstrates the care which needs to be taken when supplying "missing" conditions to convert a boundary-value problem to an initial-value problem. Very often the solution to the boundary-value problem can be much more "well-behaved" than its initial-value counterpart. In the above example the solution to the initial-value problem grows as a factor of  $e^{10x}$  and, in order that this fast-growing component be suppressed, it is necessary to obtain a very accurate estimate of the missing initial condition. This behaviour is typical of differential equations with very large positive and/or negative eigenvalues.

Another view of this sensitivity problem can be obtained by returning to the analytic shooting procedure of section 2.3. An examination of the coefficient matrix  $Q$  for the above problem reveals that it is ill-conditioned. In fact  $Q^{-1}$  has non-zero elements from order  $10^1$  down to order  $10^{-11}$ . When constructing the various approximations for the numerical shooting method, any numerical errors which may occur will be quickly magnified. This may lead to a situation where overflow can occur before the integration has been done over the complete interval. Thus matching of the boundary conditions cannot be achieved and the shooting method will fail for this problem.

These two views of the problem of numerical instability are equivalent because exponentially growing solutions lead to ill-conditioning of the Q matrix

As a means of overcoming this problem the idea of integrating over shorter intervals and somehow combining these to cover the full interval seemed appealing. This is the idea behind multiple shooting. The traditional multiple shooting technique as applied to the general n-dimensional problem with linear boundary conditions, i.e.,

$$y' = f(x, y), \quad a \leq x \leq b$$

$$B_a y(a) + B_b y(b) = \gamma$$

is considered in the following paragraphs

### 3.2 Multiple Shooting - Algorithm 1

Multiple shooting [18] proceeds by dividing the interval of integration  $[a, b]$  into  $N$ , not necessarily equal, subintervals by the points  $x_j$ , called break points, such that

$$a = x_0 < x_1 < x_2 < \dots < x_{N-1} < x_N = b$$

Let

$$\Delta_j = x_j - x_{j-1}, \quad j = 1, \dots, N$$

be the width of the  $j$ th subinterval. A change of variables, using the transformations

$$t = \frac{x - x_{j-1}}{\Delta_j}, \quad x_{j-1} < x < x_j$$

$$y_j(t) = y(x_{j-1} + t\Delta_j)$$

$$f_j(t, z) = \Delta_j f(x_{j-1} + t\Delta_j, z)$$

allows us to rewrite a separate differential equation in each of the subintervals  $(x_{j-1}, x_j)$  as follows:

$$\frac{dy_j}{dt} = f_j(t, y_j(t)), \quad 0 \leq t \leq 1, \quad j = 1, \dots, N$$

This is a system of differential equations with dimension  $N$  times that of the original system, and, given adequate boundary conditions, any of the standard solution methods may be used when attempting to find its solution

How sufficient boundary conditions can be included will now be examined. The original boundary conditions will transfer to the first and last differential equation in the new system. These can be rewritten as:

$$B_a y_1(0) + B_b y_N(1) = \gamma$$

Now if the solution to the original differential equation is assumed to be continuous, with continuous derivatives at the break points, we can add in extra boundary conditions which describe this continuity, i.e.

$$y_{j+1}(0) - y_j(1) = 0 \quad j = 1, \dots, N-1$$

This yields exactly enough conditions to guarantee a solution to the new system of differential equations

Combining these results together, the new problem can be written in condensed form as

$$\frac{d}{dt} Y = F(t, Y) \quad 0 \leq t \leq 1,$$

$$P Y(0) + Q Y(1) = \alpha$$

In the new problem  $Y$ ,  $F$  and  $\alpha$  are the vectors

$$Y = y_j(t), \quad j = 1, \dots, N$$

$$F = f_j(t, y_j), \quad j = 1, \dots, N$$

$$\alpha = (\gamma, 0, \dots, 0)^T$$

$P$  is an  $(nN \times nN)$ -matrix whose structure is:

$$\begin{array}{cccc}
B_a & 0 & & 0 \\
0 & I & 0 & \\
\cdot & & & \\
0 & & & I
\end{array}$$

and  $Q$  is an  $(nN \times nN)$ -matrix whose structure is

$$\begin{array}{cccc}
0 & 0 & & B_b \\
-I & 0 & & \\
0 & -I & & \\
0 & & -I & 0
\end{array}$$

If the shooting method is to be used on this new equation the problem will involve solving a differential equation of the form

$$\frac{dU}{dt} = F(t, U) \quad 3.2.1$$

with a number of different initial condition  $U(0) = s$ . The usual system of algebraic equations will result, which can be solved using an appropriate method.

The parallel operation is made possible by virtue of the structure of the  $nJ$ -system of differential equations (Eq 3.2.1). Successive blocks of the system are uncoupled and so may be solved simultaneously. It is only when the boundary conditions are being matched that coupling occurs, i.e. during the solution of the algebraic system.

It is worth observing that if the same numerical scheme and the same step size is used, this multiple shooting algorithm produces an error proportional to  $e^{K\Delta^2}$  rather than  $e^K$  for the simple shooting method, where  $K$  is the Lipschitz constant for the original differential equation. In simple terms, since the error is proportional to the length of the interval of integration, a reduction in interval length will reduce the error also.

This suggests that the parallel method is inherently more accurate than its sequential counterpart and gives an extra reason for pursuing this method as a possible concurrent algorithm. It is not the version of the algorithm implemented in this work, so interested readers are referred to [13] and [18] where the theory is treated in detail.

To summarise, then, because of the availability of high performance initial-value numerical code for the solution of systems such as Eq 3.2.1,

the possibility of subdividing the problem into as many subintervals as is optimal for a particular machine architecture and the improved error conditions, this algorithm would seem to be suitable for development as a robust code for parallel or multiple shooting. However, a major disadvantage is the problem of properly selecting the break or shooting points. The algorithm requires *a priori* selection and there is no mechanism for including extra shooting points automatically, if the problem so requires. A problem with special features (e.g. boundary or interior layers), may require a large number of shooting points over parts of the interval and very few or no shooting points where the solution is well behaved. Again it may be desirable to use different initial-value codes over different subintervals. If this information is available to the user he may be able to fine tune the algorithm to efficiently solve the problem but otherwise bad mesh selection can lead to at best an inefficient solution, at worst the failure of the algorithm to solve the problem at all.

Although the method outlined in this section can usefully be applied to a wide range of problems, a method which adaptively selects shooting points can be regarded as necessary for some problems where *a priori* knowledge of the behaviour of the solution is not available. Such an algorithm will be examined in the next section.

### 3.3 Multiple Shooting - Algorithm 2

In [15] Keller and Nelson propose a variation of the multiple shooting method for linear boundary-value problems with separated boundary conditions. The form of the problem which they consider is

$$y' = A(x)y + f(x), \quad a \leq x \leq b \quad 3.3.1(a)$$

$$B_a y(a) = b_a, \quad B_b y(b) = b_b \quad 3.3.1(b)$$

whose dimension is assumed to be  $n$ . This, of course, is a linear two-point boundary-value problem, with separated boundary conditions.

The interval is divided into a mesh having  $N+1$  equally spaced mesh points as usual, i.e.

$$x_j = a + jh, \quad j = 0, \dots, N$$

where  $h = (b-a)/N$ .

A single step finite difference approximation is then chosen on this mesh. The particular approximation is not important, but the centered Euler or "box" method proposed by Keller, is chosen because of its

simplicity and the error is  $O(h^2)$  The difference equation for 3.3.1(a) becomes

$$y_{j+1} - y_j = h[A_{j+1/2}(y_{j+1} + y_j)/2 + f_{j+1/2}] \quad 3.3.2$$

and for 3.3.1(b)

$$B_a y_0 = \beta_a, \quad B_b y_N = \beta_b$$

The notation  $A_{j+1/2}$  and  $f_{j+1/2}$  means  $A(a+(j+1/2)h)$  and  $f(a+(j+1/2)h)$  respectively and  $y_j$  is the approximation to  $y(x_j)$  thus defined

Eq 3.3.2 may be transposed to give a recursive formula as follows:

$$(I - hA_{j+1/2})y_{j+1} = (I + hA_{j+1/2})y_j + hf_{j+1/2}$$

or

$$y_{j+1} = (I - hA_{j+1/2})^{-1}(I + hA_{j+1/2})y_j + h(I - hA_{j+1/2})^{-1}f_{j+1/2}$$

3.3.3

It is assumed that  $h$  is sufficiently small so that the required inverse matrix exists

The parallel method proposed by Keller and Nelson can be summarised briefly as follows. A subset of the original mesh points,  $x_j, j = 0$

$N$ , is defined as shooting points and these special mesh points may be written as  $x_{j_i}$ , where  $j_i$  belong to the strictly increasing sequence

$$\{j_i, i=1, \dots, s\},$$

with  $j_1 > 0$  and  $j_s = N$ . In effect this means that some of the  $N$  original mesh points are selected and reclassified as

$$x_{j_1}, x_{j_2}, \dots, x_{j_s} = x_N$$

An attempt is now made to use Eq 3.3.3 to integrate between shooting points in the interval. The right hand side of Eq 3.3.3 depends on the (unknown) solution value,  $y_j$ , at each of the mesh points. However, because the problem is linear with linear boundary conditions, the matrix  $A$  and the vector  $f$  depend only on the (known) mesh points. The coefficient of the  $y$  vector and the independent term can therefore be

evaluated without knowledge of the solution. The approach adopted is to build up these terms from one shooting point to the next and then to solve a (linear) system of equations to find the approximate solution at these points.

More rigorously we may define the  $n \times n$  matrices  $\Phi_i$  as

$$\Phi_i = \prod_{j=j_{i-1}}^{j_i-1} (I - hA_{j+1/2}/2)^{-1} (I + hA_{j+1/2}/2)$$

where  $x_{j_0} = a, i \in j_0 = 0$ .

Similarly we define

$$\phi_i = (I - hA_{j_i-1/2}/2)^{-1} (I + hA_{j_i-1/2}/2) \bar{y}_{j_i-1-j_{i-1}}, i = 1, \dots, s$$

where  $\bar{y}_0 = 0$ , and

$$\begin{aligned} \bar{y}_{k+1} = & (I - hA_{j_i+k+1/2}/2)^{-1} (I + hA_{j_i+k+1/2}/2) \bar{y}_k \\ & + h(I - hA_{j_i+k+1/2}/2)^{-1} f_{j_i+k+1/2} \end{aligned}$$

It can be easily shown that

$$y_{j_i} = \Phi_i y_{j_{i-1}} + \phi_i, \quad i = 1, \dots, s \quad 3.3.4$$

Eq 3.3.4 comprises a system of  $n(s)$  linear equations in the  $n(s+1)$  unknowns  $y_{j_0}, \dots, y_{j_s}$ . The boundary conditions may be included to give the required number of equations so that the system can be solved. (It may be noted in passing that although separated boundary conditions are assumed it is trivial to include mixed boundary conditions without affecting the general solution technique). The stated boundary conditions thus become

$$B_a y_{j_0} = \beta_a, \quad B_b y_{j_s} = \beta_b$$





points, if necessary. The calculation of  $\Phi_k$  is effectively the integration of the system of equations over each subinterval. The "size" of the product matrix  $\Phi_k$  may be monitored using some matrix norm. When this norm grows large it indicates that the solution is beginning to change rapidly. Therefore when the norm grows beyond some acceptable value the calculation may be stopped, the current values of  $\Phi$  and  $\phi$  can be stored and a new set of calculations begun using the same starting and stopping criteria. The "acceptable value" will depend on the dimension of the original system and in his work Keller adopts a heuristic approach to the selection of a maximum value and, indeed, to the type of norm chosen. Speed of calculation is the criteria used for the selection of the type of norm.

This monitoring of the norm means that a new shooting point is introduced as is required by the behaviour of the solution. In fact, each shooting interval may be allowed to become as large as a requirement of the form

$$||\Phi_1|| \leq M, \quad 1 = 1, \dots, s$$

will allow. Here  $M$  is some user selected parameter and  $||\cdot||$  is some easily computed norm. Other conditions on the norm might produce a more rigorous control on the error but the experience of Keller and Nelson has shown that the extra computational effort does not justify the additional assurances of stability. (See also [16])

It should be pointed out that in the extreme case, where  $M$  is set too low, an additional shooting point will be introduced at every mesh point, and the algorithm becomes the ordinary finite difference method with the solution obtained at all the original mesh points. On the other hand if no shooting points are chosen, either by the user or by the algorithm, then the method reduces to simple shooting. These are the two extreme cases for this method, so that the method may be regarded as a combination of the traditional finite difference method and the original simple shooting method.

The problem of adaptively selecting break points, then, is overcome and this was a most important consideration in selecting this method as the basis for producing a working code. This allows us to address another problem associated with any all purpose code for the solution of two-point boundary-value problems, the range of problems reasonably soluble by the code. As outlined in Ch 1, no code can be regarded as the definitive one for such problems. The aim should be to attack as wide a range of problems as possible, and the possibility of adaptively selecting the shooting points broadens the range of problems potentially soluble by the algorithm.

Further discussion of the variety of problems presented to the code will be left until Ch 5, where an analysis of its performance will be presented. Problems which require several automatically selected shooting points, as well as those requiring none, will be examined at that stage.

### 3.5 Concurrent Processing Possibilities

To analyse the possibilities for concurrent processing, consider firstly the integration phase. The computation involved in each subinterval is totally independent of each other subinterval. This is because the problem being considered is linear, so that the  $\Phi$  matrices and the  $\phi$  vectors involve only the independent variable. This may be considered as an ideal structure for any kind of parallel architecture. No interaction between tasks is needed so the integration phase may be divided into as many tasks as there are processors to do these tasks. However as we shall see, because of the knock-on effect at the solution phase this may not be the most efficient choice overall.

The solution phase involves the solution of a linear system of equations. The structure of the coefficient matrix is predictable, being block bidiagonal. However the parallelisation of this phase is dependent on parallel techniques for solving such linear systems. Unfortunately no single efficient parallel algorithm has emerged in the literature which can be regarded as the "best" method.

As stated earlier, any attempt to "segment" the coefficient matrix may cause local instability. Therefore it is appropriate to consider first methods which treat the matrix as a whole. Keller suggests two possible strategies for the mapping of the problem onto the processors. He suggests some form of alternate row and column elimination method. In the first, which he calls a domain decomposition, a given processor is assigned the task of all computations within a particular subinterval. This would allow at most two processors to be involved in elimination at any one time because of the structure of the matrix. The method would be essentially serial in nature.

In simple terms his second method may be thought of as requiring that processor-1 would be involved in the elimination calculations associated with the first column in each  $\Phi$  matrix, processor-2 with the second column in each, etc. This method he calls a column decomposition. It may be noted that this method could be employed in the integration phase also. While more efficient than the domain decomposition, the communications cost are higher and the organisation of the code is more difficult. Because of the difficulty of parallelising the solution phase while treating the large block diagonal matrix it seemed more appropriate to

examine the possibility of treating each block as a separate entity and achieve some form of concurrent processing in this way

Paprzycki and Gladwell [21] have proposed a particular segmentation process which takes account of the instability problems inherent in some decomposition methods. Their general approach is to divide the system into smaller segments of similar structure which can be factorised independently. The stability is maintained by careful selection of the segments and a particular combination of elimination and row interchanges. No proof is offered for this approach, but for all the examples with structurally singular segments considered by them, the decomposition was successful.

They assume that the starting number of blocks is large and the number of processors is small, and so use only one processor to solve the reduced system. However, where the number of processors is large, recursive "tearing", as they call their approach, may be considered. The decomposition of the matrix and the back substitution phases are separated as, for nonlinear problems, the solution may have to be found sequentially for several right hand sides.

The best theoretical speed-up over the sequential version of the algorithm is a factor of 4 on small systems using up to 40 processors. For larger systems this reduces to a factor of 3. Because of these poor results they conclude that their "tearing" algorithm will not be competitive for the type of large problems which require the power of expensive parallel architecture.

Another possible approach to the problem of parallelising of the solution phase is the technique known as cyclic reduction [20]. To illustrate the technique, consider the general tridiagonal system of  $M$  equations in  $M$  unknowns,  $x_1, x_2, \dots, x_M$

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 & & = c_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 & & = c_2 \\ & a_{32}x_2 + a_{33}x_3 + a_{34}x_4 & = c_3 \\ & \vdots & \vdots \\ & a_{mm-1}x_{m-1} + a_{mm}x_m & = c_m \end{array}$$

Eliminate  $x_1$  from the second equation using the first equation and eliminate  $x_3$  from the second equation using the third equation, to get a new equation in  $x_2$  and  $x_4$ . Similarly use equations 3, 4 and 5 to produce a new equation in  $x_2$ ,  $x_4$  and  $x_6$ . Continuing in this way, a new set of equations involving only the even subscripts can be developed. These new

equations can be reduced again and the process continued until no further reduction is possible (If, for example,  $M = 2^P - 1$ , this will be a single equation) The final small equation may then be solved, and the other values found by back substitution

For the special form of block diagonal matrix which appears in the solution phase of Keller's algorithm, cyclic reduction can be done using blocks rather than individual coefficients This approach may be successfully adopted to produce a parallel code for the pure finite difference method for two-point boundary-value problems However, because of the instability problems already mentioned, the cyclic reduction process must be stopped if the norm of a particular block grows above some acceptable limit. Since this is exactly the criteria which may lead to the inclusion of a block in the original system, the process of cyclic reduction does not seem to represent a general solution method for the parallelisation of the solution phase of this algorithm

As can be seen from the last few paragraphs, the problem of producing a concurrent code for the solution phase of the algorithm is not trivial However, when considering the extension of the algorithm to cover problems with non-separated boundary conditions, the form of the coefficient matrix seemed to offer an simple and efficient approach to the decomposition, with obvious concurrent possibilities In the case of non-separated boundary conditions the coefficient matrix for the solution phase of the algorithm will be of the form

$$\begin{array}{cc} -\Phi & I \\ & -\Phi \quad I \\ & & & -\Phi & I \\ B_a & & & & B_b \end{array}$$

or in more detail for a 4th order problem

xxxx1000				
xxxx0100	0	0	0	
xxxx0010				
xxxx0001				
	xxxx1000			
	xxxx0100	0	0	
0	xxxx0010			
	xxxx0001			
				1000
				0100
				0010
				0001
xxxx				xxxx
xxxx	0	0	0	xxxx
xxxx				xxxx
xxxx				xxxx

where "x" means some unknown, and usually nonzero, element

An examination of the matrix reveals that if the  $\Phi$  blocks in the matrix were upper triangular, the only decomposition required would be the elimination of the  $B_a$  block in the bottom left hand corner and the decomposition of the final right hand corner block. The cost involved in achieving this structure and the final decomposition would be  $O(n^3G)$ , where  $G$  is the number of  $\Phi$  blocks in the system. If, however, the decomposition were done sequentially the cost would be  $O(n^3G^3)$ . Thus, a considerable saving can be made by including this technique in the solution phase.

In the spirit of attempting to perform as much calculation as possible in the (parallel) integration phase, the  $\Phi$  blocks in the matrix may be made upper triangular during this phase. This will reduce the cost marginally, although the overall order remains  $O(n^3G)$ . The upper triangular blocks can then be stored and are available for use in the (sequential) solution phase. In large systems this reduction in work during the solution phase may be significant and the inclusion of this simple decomposition method can be justified.

The major difficulty with any method of decomposition for these problems is the fact that there may be singular segments in the coefficient matrix. However, because of the simplicity of the method and the fact that a wide range of problems require no special decomposition technique, this enhancement has been included in the parallel implementation of Ch 4. If a local singularity occurs during computation, the algorithm can detect this

and the problem may be solved using a normal decomposition. This will add to the cost of computation for some problems but, on balance, the inclusion of the technique can be justified.

The method of decomposition has implications for load balancing between processors, which, because a fixed step was used while integrating, was easy to achieve during this phase. If a large number of shooting points is selected in one subinterval, a large number of  $\Phi$  matrices must be stored with consequently a large number of reductions to be done by that processor.

For a fairly general range of problems, this method significantly reduces the amount of work done in solving the linear system of equations. Numerical examples covering the potential problems outlined in the previous paragraph will be presented in Ch. 5 and any improvement in performance analysed at this point.

### **3.6 Concurrent Processing Costs**

Traditionally concurrent processing introduces overheads not associated with sequential processing. For example, in many applications, information must be interchanged between tasks while the tasks are active. This involves communications difficulties/costs as well as synchronisation problems. As will be seen when presenting the numerical results in Ch 5, the integration phase is the main computational cost in this algorithm. The tasks in this phase are completely independent so that problems associated with communication and information interchange is not a consideration during this phase.

Another problem which must be confronted when using several processors simultaneously is the effective work load distribution between processors. In other words each processor should carry out approximately the same number of calculations. Because a fixed step is used for the discretisation of the differential equation, the number of calculations in each subinterval is predictable. For this reason it is important that break points are selected at roughly equal intervals by the user for integration by each processor.

The introduction of the "triangularisation" process described above during the integration phase will affect the equidistribution of the work load for some problems. If a large number of shooting points is selected in one interval, a large number of operations will be required to prepare the data for storage for the later solution phase. This will mean that a processor dealing with less shooting points will be idle during part of this time. However, if a sequential or slightly parallel method was used for the solution phase, this would require that almost all processors would be idle.

during this time. Even with a highly parallel linear algebra routine the problem of subdividing the problem and communicating between processors would introduce other overheads. On balance, the simplicity of the method and the speed up possibilities warrant its addition to the overall algorithm.

The number of processors (subintervals) used, the restriction imposed on the size of  $||\Phi||$  or the step size chosen does not cause any problem to the effective parallelisation of the integration phase. The tasks are completely independent and any type of parallel architecture can efficiently subdivide and process this stage of the solution. By choosing to use as few break points as the problem will allow, the size of the linear system can be kept small and the solution phase can be made more efficient. It should be noted, then, that for well behaved problems or small problems, it may be inefficient to use a large number of processors.

This introduces the question as to when expensive parallel machines should be used in general. For a discussion on this see [15] which concludes that "the basic justification for the use of computers with novel architecture must be regarded as the solution to problems not otherwise (reasonably) soluble, not merely faster solution of problems presently soluble." The problems presented to this algorithm were not in this category but the results achieved will show that there is potential for presenting much larger problems with the prospect of similar success.

### 3.7 Nonlinear Problems

By their very nature, nonlinear problems are more difficult to solve than their linear counterparts. In general some form of linearisation procedure is carried out when solving nonlinear problems and the solution found using some form of iterative method. For example, if the finite difference method is applied directly to a nonlinear two-point boundary-value problem of dimension  $n$ , the result is a set of  $nN$  nonlinear algebraic equations, where  $N$  is the number of mesh points. This system may be solved using some standard technique, for example Newton's method, as was outlined earlier.

In general, multiple shooting may be applied directly to the solution of nonlinear two point boundary value problems, while maintaining the parallel possibilities inherent in the method. Indeed, multiple shooting has traditionally been called "parallel shooting" for this very reason. Since this variant of the method exploits the linear nature of the problem, the parallelism would be lost if the algorithm of section 3.3 were applied directly to a nonlinear BVP. This occurs because the solution  $y(x_{j+1})$  at every point  $x_{j+1}$  depends on the solution  $y(x_j)$ . This means that the matrix type

products required at each mesh point cannot be explicitly found so that, if this technique is applied, each mesh point effectively becomes a break point and the method reduces to pure finite differences

A method of allowing the algorithm of section 3.3 to be directly to solve nonlinear problems is now outlined. Consider applying the following linearisation technique to the general second order differential equation

$$y'' = f(x, y, y')$$

On applying the usual reduction, the equation becomes

$$y_1' = y_2 = f_1(x, y_1, y_2)$$

$$y_2' = f(x, y, y') = f_2(x, y_1, y_2)$$

If the right hand sides are approximated by Taylor's series expansions we get

$$y_1' = f_1(x, y_1^0, y_2^0) + (y - y_1^0) \frac{\partial f_1}{\partial y_1} + (y - y_2^0) \frac{\partial f_1}{\partial y_2} + \dots$$

$$y_2' = f_2(x, y_1^0, y_2^0) + (y - y_1^0) \frac{\partial f_2}{\partial y_1} + (y - y_2^0) \frac{\partial f_2}{\partial y_2} + \dots$$

where  $f_1(x, y_1, y_2)$  is written as  $f_1$ ,  $i = 1, 2$ , in the derivative terms, and the superscript is used to denote approximate values for  $y_1$  and  $y_2$ . In matrix form this becomes

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} \approx \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} - \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} \end{bmatrix} \begin{bmatrix} y_1^0 \\ y_2^0 \end{bmatrix}$$

or, more simply,



$$y' \approx Jy + \{f - Jy^0\}$$

where  $J$  is the Jacobian matrix. This is a linear differential equation which can be solved using an initial estimate for  $y^0$ , and then continually resolved using the solution thus found. It can be shown [5] that, under certain fairly general conditions, the sequence of solutions to this equation converges to the true solution. Thus the linear equation technique outlined earlier can be used as a core integrator for the more general nonlinear problems. Similarly nonlinear boundary conditions can be linearised, if necessary.

Because the solution is only available at the shooting points, some sort of interpolation procedure is required to estimate the solution at the other mesh points. This may be to use the break point as a starting point for an initial value problem and integrate the ODE across the shooting interval. However, because break points are introduced whenever the solution varies rapidly, a low order interpolation formula can be justified when approximating the solution at the other mesh points. No attempt is made in this work to extend this algorithm to cover nonlinear problems. It remains an area for further research.

### 3.8 Conclusion

Having introduced the theory of multiple shooting and outlined its capabilities, the next part of the work is to attempt to implement the selected algorithm using a code which takes advantage of the parallelism inherent in the method. Ch 4 introduces the computer language Ada, with special emphasis on the features used by this algorithm, as well as providing a detailed analysis of the procedures adopted to make the code as efficient as possible. Selected problems and the performance of the code on these problems will be discussed in Ch 5.

## **Chapter 4**

### **An Ada Implementation of a Parallel Boundary-Value Solver**

## 4.1 An overview of the Ada Philosophy

The history of the development of the Ada programming language and the Ada programming environment is well documented in various publications [26] [3]. Briefly, the language was developed over the period 1974 to 1980 by the United States Department of Defence in response to the proliferation of languages being used at that time. It was designed by a team lead by Jean Ichbiah from France working with Honeywell/Honeywell Bull, with significant input from computer scientists from several other countries.

The philosophy behind its development was to create a language which maps more easily onto the problem spaces in which computer problems lie. Theoretically any programming language can solve any computational problem but, because of the complexity of the problems on the one hand and the modern computer architecture on the other, the language should be a help rather than hinder the development of a solution.

When searching for an improved computer language, several existing languages were examined to assess their suitability for the kind of problems now being confronted. Again the results are well documented and, for instance, FORTRAN was considered too imperative with not enough emphasis on data structures, while COBOL was seen to rely too heavily on data structures. The goals in the design of Ada can be summarised as

### 1 Modifiability

Modifiability, without increasing the complexity of the program, is difficult to achieve and measure. Essentially it implies that some parts of the system can be changed or replaced, without altering the overall structure. The approach adopted by the Ada design team was to rely on object oriented ideas. This means collecting objects and the operations associated with them into modules which can be hidden from the user. The user is offered an interface to these modules but may be unable to change the internal structure and operations of the module. This "information hiding" approach is not a new approach and it involves considering each object at the level most relevant at any particular time.

Each module (and interface) can be separately compiled and tested. If the efficiency of the operations are somehow improved or updated the interface can be left unchanged and the "user" may still reuse the module without change to his own program.

As a simple example, consider square matrices and the well known operations associated with them. Ada allows these "objects" (matrices) to be defined and the operations (addition, inverse, etc.) to be coded in a module (package). The operations available to users of the module can be listed in the interface (package specification) and the method used to implement each operation is invisible to the user. If a more efficient inverting routine, say, is introduced at a later stage or other operations are to be made available, the use of the interface need not change. We have, then, the idea of a software component, analogous to a hardware component, which may be removed, improved and then replaced, without affecting the rest of the machine (program).

## 2 Reliability

Reliability means the prevention of failure in design and operation. To aid this process Ada allows separate compilation and testing of the various modules. Thus their reliability can be assured before they are included in the overall solution.

Ada also allows for error recovery during the operation of the program by the inclusion of user defined exceptions. To continue the example of the matrix package, all the operations in the package can be thoroughly tested before inclusion in any program. Even so, depending on the matrix, during inversion, division by zero may occur. This need not cause failure of the component because an exception may be included which can take some preventative action e.g. pivoting, or simply return control to the user with a suitable error message.

## 3 Understandability

Because much of the human resource invested in software projects involves maintenance of existing software rather than the creation of new software, the goal of understandability is of major importance. The amount of external documentation and internal comments should be kept to a minimum. For this reason Ada uses naming conventions, syntax and control structures which ensure ease of reading rather than ease of writing, because, although only written once, a program may be read many times.

## 4 Concurrency

With the development of modern supercomputers any new language should include in its design the capabilities of concurrent processing. Ada does this by its "task" structure which allows for

subprograms which automatically run in parallel, if this hardware facility is available. The inherent problem of synchronisation and message passing between tasks is tackled by the language so that the programmer may design his algorithm independently of the target machine. It should be noted, at this stage, that this was the single most important consideration in the selection of Ada as the programming language in this work.

The above is a brief overview of the philosophy behind the Ada programming language. It is now worth considering the structure of the language and how the above philosophy is encapsulated in the design of the language. It is, of course, not possible, or necessary, to introduce the full Ada language in this work, but, as required the various elements will be introduced and explained.

## **4.2 Structure of the Ada Programming Language**

We may begin the examination of Ada by considering type definitions and object declarations. The types available are scalar (integer, real, enumeration), composite (array, record), access, private, subtype and derived data types. While not giving a complete explanation of all possible types it is, perhaps, worth examining some of them.

Scalar types are the standard types with no structure, i.e. number types. The enumeration type allows the user to define his own special types for a particular application. The Boolean type which contains two elements "true" and "false" is an enumeration type.

The composite type array is a collection of the same type of elements, while a record type is a collection of the same or different elements. Access types are used for objects whose structure may not be static (known at compilation time). For example, the total number of shooting points in the multiple shooting algorithm of the previous chapter is not known until run time. Therefore an access type must be used to store details of these points as they become available. Access types are similar to pointers in Pascal. The above types are used as required in the following code.

In Ada a class of objects must be defined as a particular type before variables of these types can be declared. Ada does not allow objects having different types to be directly combined, i.e. it is a strongly typed language. This is to allow possible errors to be identified at compile time rather than run time, as far as possible. It also enforces the object oriented approach to design, because each type can be regarded as a class of objects. As an example of a type used in the Ada implementation of the algorithm of Ch 3, we may think of a square matrix as a "type" of data structure. At this stage there are no operations considered for this type.

Names, expressions and operations used by Ada follow the usual rules for modern high level languages. Names can be made up of a group of characters, with various restrictions involving blanks, special characters, etc. It is recommended that meaningful names should be used when identifying objects, to conform with the readability and ease of maintenance criteria.

There is one point worthy of note, however, and that is the ability to overload names and operators. As an example, consider the operation of adding (i) integers, (ii) complex numbers, (iii) matrices. In each case, the mathematical symbol is the same and Ada allows the symbol "+" to be overloaded to represent any mathematical operation of "addition". The relevant "addition" operation for each type must, of course, be defined, and the arguments on each side of the operator control which addition is selected at run time. Care should be taken when using this facility as overuse may cause confusion.

The next structure to consider is the program unit. The first types to consider are the subprograms, i.e. procedures and functions. These subprograms contain two parts, the specification (user interface), and body (a sequence of statements). As already stated the statements in the body of the subprogram may be invisible to its user. The modularity inherent in the subprogram approach to designing is therefore enhanced by the idea of information hiding. The user of the subprogram need not consider the low level implementation of the statements, and, indeed, may not be aware of them. So the subprogram can be considered as an object by its user.

The general format and structure of the subprogram body mirrors that of other high level languages. Note that a main program must be a procedure.

Further up the scale in the structure of Ada is the package. This is a collection of objects, i.e. types, functions, procedures, etc. which are somehow logically related. Again it consists of a specification and body. In the following code, an example of a package is the package which defines a square matrix and all the matrix functions and procedures which apply in this algorithm. (The overloading of the algebraic operators, (+, -, \*), is contained in this package.) Again the implementation details of this package are unknown to the main program (user). This means that, for example, more efficient matrix routines can be introduced and tested at any time without interfering with the main program. At this level a package may be thought of as an object.

An important class of program unit which must also be considered is the task. It is Ada's means of concurrent processing. The Ada tasking model is based on the concept of communicating processes. We can, therefore, view two tasks as independent processes which operate

concurrently, and may communicate with each other by passing messages. Synchronisation problems can be controlled within the language, either automatically or via wait statements

Because of its unique approach to concurrent processing it is perhaps important to give a simple example of the Ada task. This example is based on the code which will be outlined later in this chapter. Suppose we wish to integrate, using some numerical technique, over both halves of the interval  $[a,b]$ , in parallel, using two Ada tasks.

We first define two tasks, called `FIRST_HALF` and `SECOND_HALF` which will have contained in them the same routine for integrating over any interval from some start point to some finish point. The operation of each task is identical, except for the start and finish point. We therefore define two parameters, which can be given values when the task is activated. In practice, when any task is activated it immediately begins operation, suspending operation when an "entry" (parameter) is required.

If another task is called before the first task is completed it begins operation and the user has then no control over the order in which the tasks are serviced by the machine or which one will finish first. When the "entry" statement is encountered in either task the task waits until an appropriate parameter is sent to it before continuing. Thus if tasks need to communicate before finishing, an entry statement will cause suspension of the task until the required information is available, perhaps from another task. This is Ada's mechanism for synchronisation within tasks.

The tasks to perform the integration will then be

\*\*\*\*\*

```
task type FIRST_HALF is
  entry LOCAL_LEFT(L1  FLOAT),
  entry LOCAL_RIGHT(R1  FLOAT),
end FIRST_HALF;
```

```
task type SECOND_HALF is
  entry LOCAL_LEFT(L2  FLOAT),
  entry LOCAL_RIGHT(R2  FLOAT),
end SECOND_HALF,
```

\*\*\*\*\*

Two task bodies can now be written which will carry out the process of integration from  $L1$  to  $R1$  and  $L2$  to  $R2$  respectively. Within the body of the task will be an "accept" statement which mirrors the above, i.e.

\*\*\*\*\*

task body FIRST\_HALF is

-- types, procedures, functions etc needed by the task

begin

accept LOCAL\_LEFT(L1 FLOAT) do

-- Use the parameter as required by the task

end LOCAL\_LEFT,

accept LOCAL\_RIGHT(R1 FLOAT) do

-- Use the parameter as required by the task

end LOCAL\_RIGHT,

-- Statements to be executed by the task

end FIRST\_HALF,

\*\*\*\*\*

SECOND\_HALF will have a similar structure. In general there may, of course, be several times when the tasks must wait to receive or to give information (rendezvous), but our simple example requires only one, at the beginning, to give the start and finish points for the range of integration.

The tasks are activated from a main program very much like a procedure. In our example, if we wish to integrate from 0.0 to 0.5 and 0.5 to 1.0 the call would be done as in the following program extract.

\*\*\*\*\*

begin

FIRST\_HALF LOCAL\_LEFT(0.0),

FIRST\_HALF LOCAL\_RIGHT(0.5),

SECOND\_HALF LOCAL\_LEFT(0.5),

SECOND\_HALF LOCAL\_RIGHT(1.0),

end,

\*\*\*\*\*

On a parallel machine, the sequence of events for the first line of the program segment would be



- 1     Activate FIRST\_HALF
- 2     Suspend operation of the task because an "entry" statement  
      is required
- 3     Put the value 0 0 into L1
4.    Begin processing the statements in FIRST\_HALF

The next statement in the main program segment would then be processed

Similarly for the remaining 3 lines. At this stage the tasks would be operating independently, with no user control. When the tasks are finished the information may be passed to a central memory area for co-ordination. Note that the order of the arrival of the data in the central area is unpredictable and the algorithm must take account of this. In the code outlined later, a separate list is used for data from each task, and only when all tasks are finished is the information used. Also, when a task is terminated the entities within it are no longer accessible to the program.

In structure, then, a task is similar to a procedure but it has the powerful message passing capability outlined above built into its design. It is, of course, an object and as such may be included as a building block in other structures. In the subsequent code, a one dimensional array of identical tasks is created as a means of assigning tasks to available processors. The number of tasks in the array is obviously dependent on the number of processors available. The task structure is the means by which Ada allows concurrent operations, which attempts to take advantage of the new generation of supercomputers.

In many operations it is critical that a system be able to recover from error without user intervention. Ada's exception handling is an attempt to design such a feature. It allows a block structured approach to error handling. If an exception occurs, i.e. division by zero, normal processing is suspended and control is passed to the exception handler. Control will continue to pass through different block levels until a handler is encountered or the operating system is reached. As mentioned earlier, this design is an attempt to increase the reliability of a software system, particularly one which requires minimum human intervention, e.g. an embedded system.

Finally it is worth considering generic program units. This is Ada's attempt to make its software components "re-usable". What the designer does is to write a template for a particular sequence of actions. These actions may be performed on different items, but essentially the same operations are required. As a simple example consider the operation of printing numbers on screen. The procedure PUT is used, but the type of the argument (integer, float, user defined subtype, etc) will vary. It is necessary

}

to use the language defined template for the procedure PUT to create an instance of the required PUT procedure

As a more important example of a generic package consider the operation of solving a linear two-point boundary-value problem using the multiple shooting algorithm. The algorithm will be the same from one problem to the next, but the matrices and functions will be problem dependent. The strategy employed is to write a generic program unit which can be instantiated, with the relevant functions, and used to solve a particular problem. The functions are used as parameters to create an instance of the package when required.

The generic program unit can, of course, be compiled just like any other program unit. Its form is similar to an ordinary program unit, preceded by the word "generic", after which is listed the parameters.

### **4.3 Development of the Multiple Shooting Algorithm** **- Matrix Operations**

The first step in the development of the the algorithm was the creation of a package to encapsulate all the types and operations peculiar to this algorithm. The name chosen for the package was `GENERIC_REAL_TYPES`, that is to say a base level package of useful operations on real types needed by the main integrator package.

To allow these operations to be carried out on as wide a variety of types as possible, the package is made generic with respect to a floating point type, called `FLOAT_TYPE`. Two array types, `REAL_VECTOR` and `REAL_MATRIX` are then defined in terms of the floating type and the required operations (procedures and functions) then follow. Whenever the package is needed the user supplies the required floating type to be used in the calculations and the package creates the relevant array types and operations while creating a new version of this package.

The operations required are best summarised by using the specification of the package.

\*\*\*\*\*

with ARRAY\_EXCEPTIONS, use ARRAY\_EXCEPTIONS,

generic

type FLOAT\_TYPE is digits <>,

package GENERIC\_REAL\_TYPES is

-- Types

type REAL\_VECTOR is array(INTEGER range <>) of FLOAT\_TYPE;

type REAL\_MATRIX is array(INTEGER range <>) of FLOAT\_TYPE;

-- Scalar Subprograms

procedure SWAP(X,Y in out FLOAT\_TYPE),

-- Vector Arithmetic Operations

function "+"(V,W REAL\_VECTOR) return REAL\_VECTOR,

-- Vector Scaling Operations

function "\*" (X : FLOAT\_TYPE,  
              V REAL\_VECTOR) return REAL\_VECTOR,

-- Matrix Arithmetic Operations

function "+"(A,B REAL\_MATRIX) return REAL\_MATRIX,

function "\*" (A,B REAL\_MATRIX) return REAL\_MATRIX,

function "\*" (A : REAL\_MATRIX,  
              V : REAL\_VECTOR) return REAL\_VECTOR,

-- Other Matrix Operations

function INVERT(A REAL\_MATRIX) return REAL\_MATRIX;

function NORM(A : REAL\_MATRIX) return FLOAT\_TYPE,

function UNIT\_MATRIX(N INTEGER) return REAL\_TYPE;

procedure FORM\_ITERATION\_MATRICES

```

      (A          : in REAL_MATRIX,
       H          : in FLOAT_TYPE;
       I_MINUS    : out REAL_MATRIX,
       I_PLUS     : out REAL_MATRIX );

```

```

end GENERIC_REAL_TYPES,

```

\*\*\*\*\*

The "with" and "use" clauses at the beginning make a package of exception handlers, called ARRAY\_EXCEPTIONS, available for recovering from non fatal errors during operation of this package. The specification of this package is

\*\*\*\*\*

```

package ARRAY_EXCEPTIONS is

```

```

    ARRAY_INDEX_ERROR      exception,
    NEARLY_SINGULAR        exception,

```

```

end ARRAY_EXCEPTIONS,

```

\*\*\*\*\*

The specification of the package GENERIC\_REAL\_TYPES, then, tells the user what objects are available (floating point, vector and matrix types) as well as the operations defined for these types. These operations are implemented in the body of the package GENERIC\_REAL\_TYPES. The details of how this is done is not relevant at this level of analysis.

The "+" and "\*" functions are overloaded to include addition and multiplications involving matrices and vectors. Again, the final FORM\_ITERATION\_MATRICES procedure is specific to this algorithm.

This completes the specification of the first package which the shooting algorithm requires. In keeping with Ada's idea of "software components", the required matrix operations are engineered into a block and this block will be one of the components in the overall software system. Only matrix operations specifically required by this algorithm are included in the package. In an ideal world of well stocked libraries of such operations, the most efficient of the available components could be "bought" and fashioned into this package. In fact, each of these components

has been coded specifically for this algorithm but replacements may be "bought" and included, without alteration to the main program

In order that this generic package can be used, an instance of the package must be created, which includes the floating point type required by the user. As an example suppose the implementation defined LONG\_FLOAT type is to be used, then a package called, say, LONG\_FLOAT\_REAL\_TYPES must be instantiated This is done as follows

\*\*\*\*\*

```
with GENERIC_REAL_TYPES
package LONG_FLOAT_REAL_TYPES is new
    GENERIC_REAL_TYPES(FLOAT_TYPE => LONG_FLOAT);
```

\*\*\*\*\*

This is the package that is used throughout the numerical experiments of Ch. 5

#### 4.4 The Interface to the Shooting Method Package

Continuing the idea of modularisation and information hiding, it is not necessary that the user should be aware of the coding involved in carrying out the multiple shooting algorithm described in Ch 3 Instead he is offered an interface to a package containing all the operations required to successfully solve his particular linear boundary-value problem

Recall the form of the problem

$$\mathbf{y}' = \mathbf{A}(x)\mathbf{y} + \mathbf{f}(x), \quad a \leq x \leq b$$

$$\mathbf{B}_a(x)\mathbf{y}(a) + \mathbf{B}_b(x)\mathbf{y}(b) = \gamma$$

The matrix  $\mathbf{A}$ , in linear problems, may depend on the independent variable,  $x$ . In other words this matrix will contain functions of  $x$  as elements, and these functions will need to be evaluated at each step of the algorithm Similarly the vector function  $\mathbf{f}$  may have to be evaluated for each problem and at each step of the algorithm These functions are problem dependent, so a package is written using general functions and the actual functions are used to create a specific instance of the package when required. All this is to say that the package is made "generic" with respect to

the problem dependent functions, and the boundary condition matrices  $B_a$  and  $B_b$

The special functions and procedures used by the algorithm must also be made available to this package. This is done by including them in the generic part of the package specification. Certain packages are required by this package, in particular the implementation supplied package TEXT\_IO, used for input/output, the previously defined ARRAY\_EXCEPTIONS package and a package used for calculating CPU time, called ADA\_TIMER. This is a machine specific routine which calls an existing timing routine written in FORTRAN, using the pragma facility available in Ada. The specification then becomes .

\*\*\*\*\*

```
with ARRAY_EXCEPTIONS, use ARRAY_EXCEPTIONS;
with TEXT_IO,          use TEXT_IO,
with ADA_TIMER,        use ADA_TIMER,
```

generic

```
type FLOAT_TYPE is digits <>,
type VECTOR_TYPE is array(INTEGER range <>) of FLOAT_TYPE;
type MATRIX_TYPE is array(INTEGER range <>,
                           INTEGER range <>) of FLOAT_TYPE;
with function "+" (V,W : VECTOR_TYPE)
    return VECTOR_TYPE is <>,
with function "*" (X : FLOAT_TYPE,
                  V : VECTOR_TYPE)
    return VECTOR_TYPE is <>;
with function "*" (A : MATRIX_TYPE,
                  V : VECTOR_TYPE) return VECTOR_TYPE is <>;
with function "*" (A : MATRIX_TYPE,
                  B : MATRIX_TYPE) return MATRIX_TYPE is <>;
with function INVERT(A : MATRIX_TYPE)
    return MATRIX_TYPE is <>;
with function UNIT_MATRIX(N : INTEGER)
    return MATRIX_TYPE is <>,
with function NORM(A : MATRIX_TYPE)
    return FLOAT_TYPE is <>,
with procedure SWAP(X,Y : in out FLOAT_TYPE) is <>;
```

```

with procedure FORM_ITERATION_MATRICES(
    A in MATRIX_TYPE,
    X in FLOAT_TYPE,
    B out MATRIX_TYPE,
    C out MATRIX_TYPE) is <>,
with procedure PUT(N in INTEGER;
    M in INTEGER = 0,
    P in INTEGER := 10) is <>;
with procedure PUT(X in FLOAT_TYPE,
    N in INTEGER = 0,
    M in INTEGER = FLOAT_TYPE'digits,
    P in INTEGER = 2) is <>,
with procedure GET(Q out CHARACTER) is <>,

BA_MATRIX : MATRIX_TYPE,
BB_MATRIX : MATRIX_TYPE,
with function FORM_A(T FLOAT_TYPE) return MATRIX_TYPE;
with function FORM_F(T FLOAT_TYPE) return VECTOR_TYPE;

package GENERIC_INTEGRATOR is

    procedure SOLVE_BVP(NO_OF_EQUATIONS in INTEGER;
        LEFT_X_VALUE in FLOAT_TYPE;
        RIGHT_X_VALUE in FLOAT_TYPE
        NO_OF_DECIMALS in INTEGER;
        GAMMA in VECTOR_TYPE,
        MAX_NORM in FLOAT_TYPE;
        NO_OF_PROCESSORS in INTEGER
        SPECIAL in out BOOLEAN := TRUE);

end GENERIC_INTEGRATOR,

```

\*\*\*\*\*

The parameters required by the algorithm are as follows.

- 1 The number of equations or the dimension of the differential equations,
- 2 The value of a and b (the value of x at each boundary);
- 3 The number of decimal places of accuracy required in the solution, which controls the initial step size chosen,

- 4 The value of  $\gamma$ , the value of the right hand side of the boundary conditions (a constant vector);
- 5 The maximum size of the vector norm to be used,
- 6 The number of processors to be used in the solution
- 7 A signal to indicate whether the special method is used when decomposing the coefficient matrix The default value, TRUE, means that by default the special triangularisation technique is used

Before considering the implementation details of the procedure SOLVE\_BVP, it is appropriate at this stage to first of all consider the other packages that the algorithm needs and the form of the driver program for a simple second order problem

The package needs certain input, output and mathematical facilities which, in Ada are generally supplied as generic packages/procedures Since the floating point type LONG\_FLOAT has been chosen as the base type for all numerical experiments appropriate packages are now defined

```

*****

with MATH_LIB;
  package LONG_FLOAT_MATH_LIB is
    new MATH_LIB(LONG_FLOAT),

*****

with TEXT_IO,
  package LONG_FLOAT_IO is
    new TEXT_IO FLOAT_IO(LONG_FLOAT);

*****

with TEXT_IO,
  package INTEGER_IO is
    new TEXT_IO INTEGER_IO(INTEGER),

*****

```

At this stage all the required packages are available and the driver program for a specific problem can be outlined. Consider the second order problem:



$$y'' = (1 - \frac{t}{5})y + t, \quad 1 \leq t \leq 3$$

$$y(1) = 2, \quad y(3) = -1$$

In matrix form, the differential equation becomes.

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ (1 - \frac{t}{5}) & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} 0 \\ t \end{bmatrix}$$

or, in general

$$y' = Ay + f$$

and the boundary conditions become

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_1(1) \\ y_2(1) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_1(3) \\ y_2(3) \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

$$B_a y(a) + B_b y(b) = \gamma.$$

So the driver program, in its simplest form it could look like.

```

*****
with TEXT_IO;                               use TEXT_IO,
with LONG_FLOAT_REAL_TYPES,                 use LONG_FLOAT_REAL_TYPES;

with LONG_FLOAT_MATH_LIB,                   use LONG_FLOAT_MATH_LIB;
with INTEGER_IO,                             use INTEGER_IO;
with LONG_FLOAT_IO,                         use LONG_FLOAT_IO;
with ADA_TIMER,                             use ADA_TIMER,
with GENERIC_INTEGRATOR,
```

```
procedure SOLVER_1 is
```

```

  B1 : REAL_MATRIX := ((1 0, 0 0), (0 0, 0 0)),
  B2 : REAL_MATRIX := ((0 0, 0 0), (1 0, 0 0));
```

```

function A(T LONG_FLOAT) return REAL_MATRIX,
function F(T LONG_FLOAT) return REAL_VECTOR;

package LONG_FLOAT_INTEGRATOR is
    new GENERIC_INTEGRATOR(
        FLOAT_TYPE => LONG_FLOAT,
        VECTOR_TYPE => REAL_VECTOR,
        MATRIX_TYPE => REAL_MATRIX,
        BA_MATRIX => B1,
        BB_MATRIX => B2,
        FORM_A => A,
        FORM_F => F ),

    use LONG_FLOAT_INTEGRATOR,

    function A(T LONG_FLOAT) return REAL_MATRIX is
        M MATRIX(1 2,1 2) = (others => (others => 0 0));
    begin
        M(1,2) = 1 0,
        M(2,1) = 1 0 - T/5 0,
    end A;
    function F(T LONG_FLOAT) return REAL_VECTOR is
        V VECTOR(1 2) = (others => 0 0),
    begin
        V(2) := T,
    end F,
begin
    SOLVE_BVP(NO_OF_EQUATIONS => 2,
        LEFT_X_VALUE => 1.0,
        RIGHT_X_VALUE => 3 0,
        NO_OF_DECIMALS => 6,
        GAMMA => (2 0,-1 0),
        MAX_NORM => 3 0,
        NO_OF_PROCESSORS => 4
        SPECIAL => TRUE ),
end SOLVER_1,

```

\*\*\*\*\*

The call to SOLVE\_BVP passes the actual values of the parameters used for this problem. This call uses a tolerance of  $10^{-6}$ , with a

maximum norm of 3.0 when forming the  $\Phi$  matrix and assumes 4 processors are available

The above section of code is a simple driver program which uses the mam package `GENERIC_INTEGRATOR` to solve the boundary-value problem as given. In the following section an outline of the steps involved in solving the problem as well as of the integration program is presented

## 4.5 The Integrator Package

The package `GENERIC_INTEGRATOR` is the main component of the total software package required for the solution of a boundary-value problem using the algorithm due to Keller and Nelson which was presented in Ch 3. This section is not intended as a complete explanation of all the programming decisions taken during the construction of the package. However the various building blocks required together with the interfaces between them are presented. The low level implementation details are not relevant at this level.

The first requirement of the package is to declare the variables required throughout the package (global variables). Recall that there are two phases in the solution, the integration and the solution phase. Both of these phases must be carried out a sufficient number of times to achieve the required accuracy. The global variables are used to store information which must be carried between phases and between runs. Without listing all of them, they include a variable to store the total number of shooting points, the largest residual (difference between solution values at the same point) in the solution. Again a Boolean variable is declared which is set to "TRUE" when convergence is achieved and another to distinguish between the first run of the algorithm, when shooting points may be added by the algorithm, and subsequent runs which are needed for convergence.

The mam procedure which must be defined is the procedure `SOLVE_BVP`, which was referred to in the previous section. Within the procedure `SOLVE_BVP` the main entities declared are

1. A task which is used as the integrator over a subinterval. Ada allows for the creation of an array of such tasks, where each element of the array integrates over a different subinterval. The dimension of the array depends on the number of processors to be used, which has been passed as a parameter from the driver program.

2. A series of access types which are used to store the values of the independent variable, the elements of the  $\Phi$  matrix and  $\phi$  vector at each of

the "break" or shooting points Recall that any number of these may be selected within each subinterval by the algorithm so that the number of values stored may be different for each of the subintervals. As already outlined, this has implications for load balancing between processors when developing a strategy for later integration runs The policy adopted is to keep the subintervals of equal length thus guaranteeing an equal number of calls to the integration step The fact that the algorithm may begin the decomposition process at this stage means that for equations with large dimensions and boundary layers this "storage" time factor could be unpredictable However, the extra time required for storage is insignificant when compared with the time required for the actual process of carrying out one step of the integration

3 A procedure to write to and read from the access types, which use record types to link the values together As an example the procedure for writing to the list is

```

procedure STORE(WHERE      in NAME_OF_LIST;
                 F          in FLOAT_TYPE;
                 LIST_NO    in INTEGER)

```

There will be a different list for each subinterval and the value of LIST\_NO will depend on the subinterval number

4 A procedure which divides the original interval [a,b] into equally spaced subintervals whose size depends on the number of processors to be used, and passes the appropriate start and finish x-values to the elements of the array of tasks.

5. A procedure which carries out the integration from left to right over any subinterval This has defined within it a procedure called ONE\_STEP which, during the first run, is called until the right hand boundary is reached or until the norm of the matrix  $\Phi$  exceeds the limit set by the user In subsequent runs the same break points are always used eliminating the need for checking the norm after each step of the integration. The reason for this is to allow the solution to be always found at the same points so that Richardson's extrapolation procedure can be used to accelerate convergence at these points

6 A declaration block which sets up the entities required during the solution phase These include a (large) matrix whose size depends on the dimension of the original differential equation and the total number of

shooting points used for the problem and several solution vectors which hold the current, previous and "improved" solution values for the vector  $y$  over all the shooting points. The procedures for solving, extrapolating and checking convergence are also defined within this block.

These procedures are called as many times as is required to secure convergence. The step size is continually halved and, Richardson's technique, as outlined in Ch 2, is applied until the difference between any two solution values at the same point (the largest residual) is less than the user defined number of digits.

#### **4.6 Conclusion**

This brief outline is meant to give an overall picture of how the Ada implementation of the algorithm of Ch 3 is constructed. A full discussion of the avenues explored during the writing of the program is not possible. However, an idea of the type of philosophy adopted in the early stages of its development, and followed throughout, is now given. As well as using efficient programming practice, the design decisions were made having regard to the following criteria, not necessarily in the order of importance :

- 1 The implementation should be highly parallel

As previously outlined, the integration phase can be carried out completely concurrently while it also begins the solution phase in parallel. The remainder of the solution phase is carried out sequentially. Reference to the tables of results will demonstrate the success which was achieved in this aim.

- 2 The solution should be machine independent

The choice of Ada as the programming language ensures that this is so, and standard Ada features are used throughout. The timing package is machine dependent, but is included only as an aid to testing the efficiency of the implementation. The synchronisation and assignment problems associated with parallel implementations are also handled at the language level. There is minimum communication between tasks during their operation so that the particular structure of the processor array is not of critical importance. There should be, however, a certain quantity of internal memory assigned to each processor.

- 3 The spirit and philosophy of the Ada programming language should be followed

Ada features employed include an object oriented approach, modularity, information hiding, readability, data abstraction, tasking, access types, arrays of various objects, record types, operator overloading, generic packages, subprograms and exception handling

4. The class of problem solved should be as large as possible

As outlined in **Ch. 1** boundary-value problems come in a variety of forms and complexity No algorithm can hope to solve all BVP's efficiently so one must aim for a subset of the wider class **Ch 5** contains a selection of the problems submitted to this code In general the code is designed for linear boundary-value problems with linear boundary conditions, either separated or non separated

- 5 The user interface should be as friendly as possible

As can be seen the driver program requires the minimum of programming skills The problem is converted to standard mathematical form, coded and the generic package `GENERIC_INTEGRATOR` then sets up all the types, arrays etc , needed by the algorithm, without reference to the user

**Chapter 5**

**Numerical Experiments**

△

↶

,

## 5.1 Criteria for Comparison

When attempting to set up numerical experiments to test the performance of any code, the criteria which are being tested should be clearly stated. The approach adopted is to select a variety of problems to test, so that the range of problems that the code is capable of solving can be identified. As already stated this work concentrates on linear two-point boundary-value problems, with linear boundary conditions, either separated or non separated. Within this range an attempt is made to examine problems with "smooth" solutions or rapidly changing solutions, i.e. problems which require the automatic selection of extra shooting points and problems which do not. The examples used are taken from standard text books, except where otherwise stated, and are designed to illustrate various features of the performance of the code.

Given that the code is capable of solving a reasonable selection of problems, the next task is to examine on what basis improvements are identified. Since the reason for developing parallel algorithms is to solve problems not otherwise reasonably soluble, a reduction in CPU time, without loss of accuracy, must be regarded as the most important aim in the following tests. As the number of processors is increased, the expectation is that the total CPU time will be reduced. The approach chosen for the analysis of the results in this chapter, then, is as follows:

1. Submit a problem to existing sequential solvers
2. Submit the problem to this solver run in sequential mode, i.e. using only 1 processor
3. Submit the problem to this solver using 4, 8 and 16 processors.

Improvements in terms of the CPU time taken would certainly be expected between the sequential and parallel versions of this algorithm. However, because of the difference in language used (FORTRAN as against Ada), comparisons between existing solvers and this algorithm were less predictable. The existing codes used in the experiments were all taken from the FORTRAN NAG library, and they may be regarded as the most efficient of these routines. Thus the first part of the experiment, although useful, cannot be regarded as an absolute judgement of the potential of the code. Again, by careful choice of parameters, relying on prior knowledge of the problem, dramatic improvements can be found in the solution time for the same problem.

The Ada code, at this first level of refinement, makes no claim to be really efficient in sequential mode. The aim at all times was to examine its



contribution to the parallel solver environment. Despite the above limitations, a set of tables detailing comparative times, using existing FORTRAN code with default parameters, is given as a part of each of the tables in this chapter.

Of much more significance is the possibility of improved performance which may accrue by running the parallel version of the Ada code rather than its sequential version. The problems submitted to the code in this chapter can be regarded as "small", so that we seek to analyse the areas of the solution where improvement in speed is observed. We may then have some idea of the type of improvement which may be expected for larger problems.

We shall be seeking a speed up which will reflect the number of processors used. The ideal speed up for  $p$  processors would result in the solution time being  $1/p$  times the solution time for the sequential version of the code.

As an extra examination, each problem is solved initially using Keller's algorithm with the solution phase being done using a sequential linear solver which treats the co-efficient matrix as a full matrix. The problem is then re-solved using the technique outlined in Ch. 3, where sections of the co-efficient matrix are decomposed in the integration phase and the resulting co-efficient matrix requires elimination of the bottom block, plus back substitution. Significant improvements were predicted where this method could be applied and the results verify this prediction.

As a final consideration in the production of the results tables, we examine the type of machine on which the experiments were done. No parallel machine or parallel Ada compiler was available, so the target machine is a sequential machine, in fact a VAX 6230 running DEC Ada compiler version 5.1 was used. This machine is a time-sharing machine with 3 processors. This raises the question of whether, if 2 or more processors are available at the same time, the machine may, in fact, run 2 or more Ada tasks concurrently.

An experiment to test the validity of the conclusions drawn when analysing the results was set up to verify that the machine is, in fact, a purely sequential machine. This involved running a job which required 4 million assignments using 1, 2, 4 and 8 Ada tasks. If 1 task is taken as the reference time (100%), the time for 2 tasks to perform the same job is represented by 100.26%, 4 tasks by 96.64% and 8 tasks by 96.90%. This shows that no significant improvement is obtained on this sequential machine by using Ada's tasking facility. It is worth noting that if an Ada procedure is used instead of a task the time is represented by 91.2%, indicated the overheads associated with the setting up of Ada tasks. On the other hand, if a procedure called by a main procedure is used, the time becomes 116.28%,

indicating the even greater overheads inherent in this type of program structure. The above conclusions were also verified by direct contact with the suppliers of the machine and compiler.

## 5.2 General Remarks on the Numerical Experiments

In the following problems, a constant set of parameters are used throughout. The user supplied parameters, with the values used, where appropriate, are

- 1 The order of the differential equation
- 2 The boundaries of the interval
- 3 The number of decimal places of accuracy required (in all examples 6 decimal places of accuracy)
- 4 The vector of boundary values
- 5 The maximum norm (after some experimentation the values 3.0 for second order, 15.0 for fourth order were used)
- 6 The number of processors to be used (1, 4, 8, 16).

The first table for each problem represents the results obtained using no special decomposition technique, while the second table represents the results using the partial decomposition technique during the integration phase outlined in Ch. 3. Finally, the appropriate times are presented for the solution of the problem using a finite difference code (D02RAF), and a shooting code (D02SAF), from the FORTRAN NAG library of routines. Once again note that default parameters are used in these routines. This may lead to failure of the shooting code for some problems. The choice of more appropriate parameters would allow these problems to be solved by the routine D02SAF.

The various columns in the tables represent

- 1 The number of processors used (P)
- 2 The number of shooting points selected by the algorithm.
- 3 The largest difference between solution values over the final two runs of the algorithm  
(For convenience, this value is labelled "Approx. Error").
- 4 The CPU time, in seconds, for the integration phase of the algorithm (I)
- 5 The CPU time, in seconds, for the solution phase of the algorithm (S)
- 6 The total time taken for the solution, run sequentially (I + S)

7. The actual time taken (A), if a parallel machine were available ( $A = I/P + S$ )
8. The speed up observed ( $SU = \text{Time using one processor} / A$ ).
9. The efficiency using P processors ( $SU./P$ )
10. The percentage of the actual time used in the solution phase
11. The solution time, in seconds, using NAG routine D02RAF
12. The solution time, in seconds, using NAG routine D02SAF

The tables are designed to illustrate the effectiveness of the algorithm in a variety of ways. Firstly, the speed up and efficiency are a measure of how close a particular version of the algorithm is to the theoretical ideal. An efficiency value of 1 indicates "perfect" speed up. The calculation of the actual time can be justified by remembering that the integration phase is purely parallel and the algorithm pays particular attention to load balancing. The only extra work which may occur in an interval depends on the number of shooting points in that interval. The positions of any such break points are listed underneath the tables where appropriate.

The significance of the solution time as a percentage of the actual time comes from the fact that the solution phase is a purely sequential operation and will therefore involve all processors, except one, being idle. This time should be kept as small as possible. In the examples presented, dramatic reductions in this time are achieved by the use of the integration phase to begin the solution phase.

### 5.3 Numerical Examples

#### Example 1

The first problem to be examined is a simple second order problem with separated boundary conditions, previously mentioned to illustrate the driver program of Ch. 4. It is

$$y'' = (1-t/5)y + t, \quad 1 \leq t \leq 3,$$

$$y(1) = 2, \quad y(3) = -1$$

The problem would not be expected to create difficulties for any of the codes to which it is submitted. This, in fact, is the case, although a small number of shooting points is selected by the Ada code, reflecting area of the

solution where changes are taking place. Because the problem is "small" major improvements in efficiency would not be expected by the use of a large number of processors. The results are summarised in Table 1a and 1b.

### Example 2

The next problem presented to each code is a second order problem involving non separated boundary conditions.

$$y'' + y' + y = x^3 - 5, \quad 0 \leq x \leq 3,$$

$$y(0) + y(3) = 2, \quad y'(0) + y'(3) = 9$$

The reason for the inclusion of this small problem is to illustrate the ability of the Ada code to solve boundary-value problems with different kinds of boundary conditions. Again major efficiencies with a large number of processors would not be expected. The results are summarised in Table 2a and 2b.

### Example 3

$$y'' = 0.09y - 1.8, \quad 0 \leq x \leq 10,$$

$$y(0) = 100, \quad y(10) = 20$$

This example is a second order problem, the interval of integration being from  $x = 0$  to  $x = 10$ . Because of the relatively large range of integration, most of the work is being done in the integration phase and therefore in parallel. Because of this, the efficiency of the Ada code would be expected to show a major improvement, and this indeed is reflected in the results as presented in Table 3a and 3b. Efficiencies in excess of 0.74 can be achieved using 16 processors or less.

### Example 4

$$y^{iv} = 10 \sin(\pi x/10), \quad 0 \leq x \leq 10,$$

$$y(0) = 0, \quad y'(0) = 0, \quad y(10) = 0, \quad y''(10) = 0$$

This example involves the fourth order problem already mentioned, and a high proportion of the work will occur in the integration phase. To use a large number of processors should be more efficient for this example rather than the earlier smaller problems. This example is used to test the ability and efficiency of the Ada code on higher order problems. The results, which again reflect excellent efficiency values, are summarised in Tables 4a and 4b.

#### Example 5

$$y_1' = \lambda y_2,$$

$$y_2' = \lambda y_1 + \lambda \cos^2(\pi x) + 2/\lambda \pi^2 \cos(2\pi x), \quad 0 \leq x \leq 1,$$

$$\lambda = 20.0, \quad y_1(0) = 0 \quad y_1(1) = 0$$

This is an example of a "difficult" problem, taken from [1], reflected by the fact that a large number of shooting points is selected by the algorithm. This, in fact, reduces the algorithm to pure finite difference (The shooting points selected are equispaced over the interval and so are not listed). However, efficiency values above 0.50 using 16 processors or less can be achieved. Larger values of  $\lambda$  reduce the efficiency of the code and introduce singular blocks in the solution matrix, but this may be alleviated somewhat by adopting a different approach to the selection of an initial step length. Tables 5a and 5b summarise the results achieved with  $\lambda = 20.0$ .

#### Example 6

$$\varepsilon y'' + xy' = \varepsilon \pi^2 \cos \pi x - \pi x \sin \pi x, \quad -1 \leq x \leq 1,$$

$$\varepsilon = 0.1, \quad y(-1) = -2, \quad y(1) = 0$$

This problem, again taken from [1], is an example of a problem with a boundary layer at the left hand side of the interval, reflected by the code's selection of shooting points. These points are listed for the one processor case. For smaller values of  $\varepsilon$  similar automatic selection occurs. Results for  $\varepsilon = 0.1$  are outlined in Tables 6a and 6b. For smaller values of  $\varepsilon$ , some tuning would be desirable, especially in relation to the choice of initial step size.

## 5.4 Conclusions

The preceding section outlines the results achieved on a selection of two-point boundary-value problems using a variation of the algorithm due to [15] as outlined in Ch 3, and coded in Ch. 4. As already stated, no single code can claim efficiency for all types of problems, so it is useful to outline the type of problem which best suits the code.

Since the main parallel section of the code is during the integration phase, problems which require a large amount of work in this phase achieve greatest efficiency. If the problem requires a small amount of work in the solution phase, this will increase efficiency even more. Thus, a large problem, with a large interval of integration, will allow all processors to be busy during the purely parallel integration phase. If the solution is reasonably smooth, i.e. a small number of shooting points is selected by the algorithm, this will minimise the size of the solution matrix. The amount of work in the (sequential) solution phase will be similarly minimised.

The inclusion of the "special" reduction process for blocks of the solution matrix increases the efficiency dramatically and warrants inclusion, even though it may cause the algorithm to fail for certain problems.

Ch. 6 contains a brief summary of the work done during the research of this thesis and an indication of the direction which future work with this algorithm might take.

$$y'' = (1 - x/5)y + x, \quad 1 \leq x \leq 3,$$

$$y(1) = 2, \quad y(3) = -1$$

**Table 1a**

No of Processors	No of Shooting Points	Approx Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	3	1.95e-9	0.83	0.07	0.90	0.90	----	----	7.8	0.38	0.23
4	1	1.95e-9	0.88	0.09	0.97	0.31	2.90	0.73	29.0	0.38	0.23
8	0	1.95e-9	1.03	0.30	1.33	0.43	2.09	0.26	69.8	0.38	0.23
16	0	1.95e-9	1.23	1.62	2.85	1.70	----	----	95.3	0.38	0.23

\*  $x = 1.46785, 1.96875, 2.5.$

\*\*  $x = 1.46875.$

**Table 1b (Special Decomposition)**

No of Processors	No of Shooting Points	Approx Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	3	1.95e-9	0.83	0.02	0.85	0.85	----	----	2.4	0.38	0.23
4	1	1.93e-9	0.87	0.03	0.90	0.25	3.40	0.85	12.0	0.38	0.23
8	0	1.93e-9	1.02	0.02	1.04	0.15	5.76	0.72	13.3	0.38	0.23
16	0	1.98e-9	1.26	0.05	1.30	0.13	6.63	0.41	38.5	0.38	0.23

$$y'' + y' + y = x^3 - 5, \quad 0 \leq x \leq 3$$

$$y(0) + y(3) = 2, \quad y'(0) + y'(3) = 9.$$

**Table 2a**

No. of Processors	No. of Shooting Points	Approx. Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	0	4.75e-8	1.23	0.02	1.25	1.25	----	----	1.6	1.40	0.38
4	0	5.97e-8	1.29	0.06	1.35	0.38	3.28	0.82	15.8	1.40	0.38
8	0	5.97e-8	1.42	0.28	1.70	0.46	2.72	0.34	60.9	1.40	0.38
16	0	6.01e-8	1.63	1.62	3.25	1.72	----	----	94.2	1.40	0.38

**Table 2b (Special Decomposition)**

No. of Processors	No. of Shooting Points	Approx. Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	0	4.75e-8	1.19	0.01	1.20	1.20	----	----	0.8	1.40	0.38
4	0	5.97e-8	1.25	0.01	1.26	0.32	3.75	0.86	3.13	1.40	0.38
8	0	5.97e-8	1.39	0.02	1.41	0.19	6.32	0.79	10.53	1.40	0.38
16	0	6.01e-8	1.58	0.05	1.63	0.15	8.00	0.50	33.33	1.40	0.38



$$y'' = 0.09y - 1.8, \quad 0 \leq x \leq 10,$$

$$y(0) = 0, \quad y(10) = 0$$

**Table 3a**

No of Processors	No of Shooting Points	Approx Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	11*	5 17e-10	4 00	0 79	4 79	4 79	----	----	16 5	0 38	0 16
4	8**	5 17e-10	4 05	0 79	4 84	1 80	2 66	0 66	43 9	0 38	0 16
8	8***	5 17e-10	4 14	1 68	5 82	2 20	2 18	0 27	76 4	0 38	0 16
16	0	5 17e-10	4 32	1 63	5 95	1 90	2 52	0 16	85 6	0 38	0 16

\* x = 0.875, 1.75, 2.625, 3.5, 4.375, 5.25, 6.125, 7.0, 7.875, 8.75, 9.625

\*\* x = 0.875, 1.75, 3.325, 4.25, 5.875, 6.75, 8.375, 9.25

\*\*\* x = 0.875, 2.125, 3.375, 4.625, 5.875, 7.125, 8.375, 9.625

**Table 3b (Special Decomposition)**

No of Processors	No of Shooting Points	Approx Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	11	5 17e-10	4 02	0 03	4 05	4 05	----	----	0 7	0 38	0 16
4	8	5 17e-10	4 05	0 04	4 09	1 05	3 85	0 96	3 8	0 38	0 16
8	8	5 17e-10	4 14	0 06	4 20	0 58	7 01	0 88	10 3	0 38	0 16
16	0	5 17e-10	4 33	0 07	4 40	0 34	11 91	0 74	20 6	0 38	0 16

$$y^{iv} = 10 \sin (\pi x/10), \quad 0 \leq x \leq 10,$$

$$y(0) = 0, \quad y'(0) = 0, \quad y(10) = 0, \quad y''(10) = 10.$$

**Table 4a**

No. of Processors	No. of Shooting Points	Approx. Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	5 *	2.12e-8	11.80	0.99	12.79	12.79	----	----	7.7	1.10	1.57
4	4 **	2.21e-8	11.84	1.96	13.80	4.92	2.60	0.65	39.8	1.10	1.57
8	0	2.21e-8	12.03	1.95	13.98	3.45	3.70	0.46	56.5	1.10	1.57
16	0	2.21e-8	12.45	12.63	25.08	13.41	----	----	94.2	1.10	1.57

\*  $x = 1.96875, 3.9375, 5.90625, 7.825, 9.84375$ .

\*\*  $x = 1.96875, 4.46875, 6.96875, 9.46875$ .

**Table 4b (Special Decomposition)**

No. of Processors	No. of Shooting Points	Approx. Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	5	2.21e-8	11.78	0.06	11.84	11.84	----	----	0.5	1.10	1.57
4	4	2.21e-8	11.78	0.07	11.90	3.00	3.94	0.99	2.3	1.10	1.57
8	0	2.21e-8	12.00	0.08	12.08	1.58	7.49	0.94	5.1	1.10	1.57
16	0	2.21e-8	12.42	0.20	12.62	0.98	12.13	0.76	20.4	1.10	1.57

$$y_1' = \lambda y_2,$$

$$y_2' = \lambda y_1 + \lambda \cos^2(\pi x) + 2/\lambda \pi^2 \cos(2\pi x), \quad 0 \leq x \leq 1$$

$$\lambda = 20, \quad y_1(0) = 0, \quad y_1(1) = 0.$$

**Table 5a**

No of Processors	No of Shooting Points	Approx Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	31	3 17e-8	4 07	23 64	27 71	27 71	----	----	82 9	0 62	----
4	28	3 17e-8	4 29	22 97	27 26	24 04	1 15	0 29	95 6	0 62	----
8	24	3 17e-8	4 52	23 24	27 76	23 81	1 16	0 15	97 6	0 62	----
16	16	3 17e-8	5 31	23 32	28 63	23 65	1 17	0 07	98 6	0 62	----

**Table 5b (Special Decomposition)**

No of Processors	No of Shooting Points	Approx Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	31	3 47e-8	4 16	0 23	4 39	4 39	----	----	5 5	0 62	----
4	28	3 47e-8	4 34	0 26	4 60	1 35	2 97	0 74	19 3	0 62	----
8	24	3 47e-8	4 55	0 25	4 80	0 82	5 36	0 67	30 5	0 62	----
16	16	3 47e-8	4 90	0 21	5 11	0 52	8 50	0 53	41 2	0 62	----

$$\epsilon y'' + xy' = \epsilon \pi^2 \cos(\pi x) - \pi x \sin(\pi x), \quad 0 \leq x \leq 1,$$

$$\epsilon = 0.1, \quad y(-1) = -2, \quad y(1) = 0.$$

**Table 6a**

No of Processors	No of Shooting Points	Approx Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	7*	1 01e-8	3 87	0 49	4 36	4 36	----	----	11 2	0 61	----
4	6	1 01e-8	3 89	0 0 77	4 66	1 74	2 51	0 63	44 3	0 61	----
8	5	1 01e-8	4 07	1 56	5 63	2 07	2 11	0 26	75 4	0 61	----
16	3	1 15e-8	4 41	4 31	8 72	4 59	----	----	93 9	0 61	----

x = -0.90625, -0.8125, -0.71875, -0 625, -0.5, -0 34375, -0 09375

**Table 6b (Special Decomposition)**

No of Processors	No of Shooting Points	Approx Error	Integration Time (secs)	Solution Time (secs)	Total Time (secs)	Actual Time (secs)	Speed Up	Efficiency	Solution Time (%)	D02RAF CPU Time	D02SAF CPU Time
1	7	1 01e-8	3 96	0 03	3 99	3 99	----	----	0 8	0 61	----
4	6	1 01e-8	4 12	0 06	4 18	1 09	3 7	0 92	5 5	0 61	----
8	5	1 01e-8	4 29	0 07	4 36	0 61	6 6	0 82	11 5	0 61	----
16	3	1 15e-8	4 40	0 09	4 49	0 37	10 9	0 68	24 3	0 61	----

## **Chapter 6**

### **Conclusions and Further Work**

## 6.1 Aims

The original aim of the work contained in this thesis was to examine how the solution methods for the general class of two-point boundary-value problems could take advantage of the current generation of parallel processing machines. To do this, the first task was to examine the traditional methods being used for the solution of such problems. A variety of methods are available, some for a narrow range of problems, but only three methods could offer the ability to solve a wide range of problems, namely finite element, finite difference and shooting methods.

The first two methods mentioned involve the solution of linear or nonlinear systems of equations and the availability of concurrent packages for the solution of such systems will govern the efficiency or otherwise of these methods. As mentioned earlier, work is ongoing in this field. However because of the nature of the shooting algorithm, more particularly the multiple shooting method, this seemed an obvious path to choose when considering parallel processing capabilities.

The "original" multiple shooting method was designed to solve problems which simple shooting failed to solve because of the nature of the differential equation. Recall the instability involved in the simple shooting method when initial guesses were not close to the true missing boundary conditions. Even multiple shooting could fail for some problems if the selected break point were too far apart. Effectively simple shooting is used within a subinterval and the same instability problems may arise. Therefore any multiple shooting code which can truly be called general purpose should have the capability of automatically selecting break points to control these instabilities. With this in mind the second algorithm outlined in Ch. 3 was selected as the most general purpose algorithm for coding in a parallel environment.

The next problem involved the selection of a suitable language for the coding of the algorithm. The language Ada seemed to offer the various features which were required for this task. In particular, the inbuilt parallel processing capabilities were a definite attraction. The fact that Ada contains many of the best features of a modern programming language as outlined in Ch. 4 confirmed its suitability as a vehicle for the programming work.

The aim of the work, then, can be summarised as the production of a working package for the solution of two-point boundary-value problems, using a multiple shooting algorithm, with a high degree of concurrency using the Ada programming language, philosophy and environment.

## 6.2 Summary

The survey of solution methods for two-point boundary-value problems undertaken in Ch. 1 and Ch. 2 served as a platform for identifying the difficulties and pitfalls involved in these solution methods. In conjunction with this, it was necessary to investigate parallel architectures and parallel programming languages. Having selected Ada as the language, the type of computer model became less critical because Ada attempts to be machine independent. For this reason only a brief summary of parallel architectures is included.

An algorithm was then sought which would minimise overheads associated with different arrays of processors. This led to the selection of the algorithm due to Keller and Nelson as communication costs during the integration phase are trivial and the solution phase time can be reduced to allow a sequential solver to achieve efficiencies. Of course, the added advantages of this algorithm as outlined earlier played a significant role in its adoption for coding.

Some preliminary work in the Ada language involved working as part of a group writing matrix routines and some work using Newton's method for the solution of nonlinear algebraic equations. These small projects served to provide a sound base for the coding of the larger packages as outlined in Ch 4.

The major achievement in the work was the construction, implementation and testing of the parallel Ada code. Of particular importance was the inclusion of the special decomposition technique in the integration phase. As the tables of results indicate, this allowed for increased efficiencies over a purely sequential method in the solution phase. The failure of this technique for some problems indicates that further development is required to make the code more general purpose. In particular, closer monitoring of the norm may be required during the integration phase. Despite this restriction, a representative subset of linear problems was successfully solved using the revised algorithm.

## 6.3 Conclusion

In conclusion, it seems appropriate to indicate the direction which future research in this area might usefully be directed. As indicated earlier, closer examination of the behaviour of the norm of the matrix blocks in the integration phase might increase the range of problems soluble efficiently by the algorithm. Problems whose boundaries include  $\pm\infty$  would seem possible candidates for efficient solution because of the size of the

interval of integration. Multi-point boundary-value problems might also be approached in a similar manner

Perhaps the single most important area where more research needs to be done is the parallel solution of nonlinear problems. Generally a sequence of linear problems must be solved when attempting to solve nonlinear problems. There is no reason why this code, with some adaptation, could not be used as the core integrator, thus extending its capability into the field of nonlinear two-point boundary-value problems. Although no code can be efficient on every problem, work in the indicated areas could provide further additions to the range of problems which could be efficiently solved in a parallel environment



## References

- [1] Ascher, U M , Matheij, R M M , Russell, R D , "Numerical solutions of boundary value problems for ordinary differential equations", Prentice Hall, 1988.
- [2] Bailey, Paul B , Shampine, Lawrence F , and Waltman Paul E., "Nonlinear two-point boundary-value problems", Academic Press, 1968.
- [3] Booch, Grady, "Software engineering with Ada", Second Edition, Benjamin/Cummings Publishing Company Inc , 1987
- [4] Daniel, James W , "A road map of methods for approximating two-point boundary-value problems in ordinary differential equations", Ed Childs, B., Scott, M., Daniel, J.W., Denman, E , Nelson, P , from "Lecture notes in computer science" Ed. Goos, G and Hartmanis, J , No 76, Springer-Verlag, 1979
- [5] Doolan, E P , Miller, J J H and Schilders, W H. A , "Uniform numerical methods for problems with initial and boundary layers", Boole Press, 1980
- [6] Firnett, P J , Troesch, B. A , "Shooting-splitting method for sensitive two-point boundary-value problems", from "Lecture Notes in Computer Science", Ed Dold, A., Eckmann, B., No. 362, Springer-Verlag, 1972
- [7] Fox, L., "Some improvements in the use of relaxation methods for the solution of ordinary and partial differential equations", Proc. Roy Soc , No. 190, 1947
- [8] Fox, L , "The numerical solution of two-point boundary-value problems in ordinary differential equation", Oxford University Press, 1957.

- [9] Fox, L., "Numerical methods for boundary-value problems", from "Computational techniques for ordinary differential equations", Ed. Gladwell, I and Sayers, D. K , Academic Press, 1980.
- [10] Henrici, Peter, "Discreet variable methods in ordinary differential equations", Wiley 1962
- [11] Hwang, Kai and Briggs, Fayé A , "Computer Architecture and Parallel Processing", McGraw Hill Book Co , 1985
- [12] Ince, E L., "Ordinary differential equations", Dover Publications, Inc , 1926
- [13] Keller, H. B , "Numerical methods for two-point boundary-value problems", Blaisdell, 1968
- [14] Keller, H B , "Numerical solutions of two-point boundary-value problems", Society for Industrial and Applied Mathematics, No 24, 1976
- [15] Keller, H. B. and Nelson, Paul, "A hypercube implementation of parallel shooting", Caltec 1986.
- [16] Krogh, F. T., "Workshop selection of shooting points", Ed. Childs, B., Scott, M , Daniel, J W , Denman, E., Nelson, P., from "Lecture Notes in Computer Science" Ed Goos, G and Hartmanis, J , No 76, Springer-Verlag, 1979
- [17] Kubíček, Milan and Hlaváček, Vladimír, "Numerical solution of nonlinear boundary-value problems with applications", Prentice Hall International Series in the Physical and Chemical Sciences, 1983
- [18] Miranker, W. L., "A survey of parallelism in numerical analysis", SIAM Rev. Vol 13, Oct 1979.

- [19] Ortega, James M and Poole, William G Jr, "An introduction to numerical methods for differential equations", Pitman Publishing Inc , 1981
- [20] Ortega, James M , "Introduction to parallel and vector solutions of linear systems", Frontiers of Computer Science, Series Ed Rosenberg, Arnold L , 1988.
- [21] Paprzycki, Marcin and Gladwell, Ian, "Solving almost block diagonal systems on parallel computers", SMU Math Rept, 89-18, 1989
- [22] Pereyra, V , "The difference correction method for non-linear two-point boundary-problems of class M", Rev Union Mat, Argentina, No 22, 1965
- [23] Pereyra, V , "PASVA3 an adaptive finite difference Fortran program for first order nonlinear, ordinary boundary problems", Ed Childs, B ,Scott, M., Daniel, J W., Denman, E., Nelson, P , from "Lecture Notes in Computer Science" Ed. Goos, G and Hartmanis, J , No. 76, Springer-Verlag, 1979.
- [24] Walsh, J , "Boundary-value problems in ordinary differential equations", from "The state of the art in numerical analysis", Academic Press, 1977.
- [25] Wylie, C. Ray and Barrett, Louis C , "Advanced engineering mathematics", 5th edition, McGraw Hill International Book Co , 1982
- [26] Young, S J , "An introduction to Ada", Ellis Horwood Publishers, 1983