LBL-13287
(STAN-CS-82-899)

AN ADAPTIVE FINITE DIFFERENCE METHOD FOR HYPERBOLIC SYSTEMS IN ONE SPACE DIMENSION[1]

John H. Bolstad

Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Ph.D. Dissertation,
Computer Science Department,
Stanford University

June, 1982

[1] Supported in part by the Office of Naval Research under contract N00014-75-C-1132, and by the Director, Office of Basic Energy Sciences, Engineering, Mathematical, and Geosciences Division of the U. S. Department of Energy under Contract DE-AC03-76SF00098.

Table of Contents

1 Introduction 1
  1.1 Statement of the Problem 1
  1.2 Brief History of Adaptive Methods 4
  1.3 Other Adaptive Methods for Time-Dependent Problems 7
  1.4 Summary of Results 12

2 Mesh Structure and Solution Algorithm 16
  2.1 The Continuous Problem 16
  2.2 Mesh Structure - First Description 19
  2.3 Mesh Structure--Second Description 23
  2.4 Operations on Refinements 31
  2.5 Difference Approximation 34
  2.6 Solution Algorithm 40

3 Stability 50
  3.1 Preliminaries 50
  3.2 Need for a New Stability Definition 55
  3.3 Stability of Refinement Algorithm 62

4 Error Analysis 64
  4.1 Modes of Convergence 66
  4.2 Interpolation Error 67
  4.3 Rate of Convergence, I 70
  4.4 Rate of Convergence, II 73

5 Estimation of the Local Truncation Error 81
  5.1 Methods not Using Local Truncation Error 82
  5.2 Four Methods 83
  5.2.1 Differences 84
  5.2.2 Estimating an Interpolant with 'Small' Derivative 85
  5.2.3 Two-Step Richardson Extrapolation 87
  5.2.4 Three-Step Richardson Extrapolation 87
  5.3 Coarse/Fine Interfaces 94
  5.4 Boundaries 95
  5.5 Systems 99

6 Data Structures 102
  6.1 Deques 102
  6.2 Trees 105

6.3 Memory Repacking 108
6.4 Alternative Data Structures 109

7 Choice of Programming Language 114

8 Computational Results 120
8.1 Model Problems 121
8.2 Qualitative Results 123
8.3 Choosing Refinement Ratios and Maximum Levels 156
8.4 Efficiency of the Method 161
8.5 Behavior as $h \to 0$ 165
8.6 Estimating the Local Truncation Error in the Interior 167
8.7 Estimating the Local Truncation Error at Boundaries 167
8.8 How Often Should the Local Truncation Error Be Checked? 169
8.9 Linear vs. Quadratic Interpolation 171

9 Conclusions and Extensions 174
References 178

A Appendix: Program Listing 184

Abstract

Many problems of physical interest have solutions which are generally quite smooth in a large portion of the region of interest, but have local phenomena such as shocks, discontinuities or large gradients which require much more accurate approximations or finer grids for reasonable accuracy. Examples are atmospheric fronts, ocean currents, and geological discontinuities.

In this thesis we develop and partially analyze an adaptive finite difference mesh refinement algorithm for the initial boundary value problem for hyperbolic systems in one space dimension. The method uses clusters of uniform grids which can "move" along with pulses or steep gradients appearing in the calculation, and which are superimposed over a uniform coarse grid. Such refinements are created, destroyed, merged, separated, recursively nested or moved based on estimates of the local truncation error. We use a four-way linked tree and sequentially allocated deques (double-ended queues) to perform these operations efficiently. The local truncation error in the interior of the region is estimated using a three-step Richardson extrapolation procedure, which can also be considered a deferred correction method. At the boundaries we employ differences to estimate the error. Our algorithm was implemented using a portable, extensible Fortran preprocessor, to which we added records and pointers.

The method is applied to three model problems: the first order wave equation, the second order wave equation, and the inviscid Burgers' equation. For the first two model problems our algorithm is shown to be three to five times more efficient (in computing time) than the use of a uniform coarse mesh, for the same accuracy. Furthermore, to our knowledge, our algorithm is the only one which adaptively treats time-dependent boundary conditions for hyperbolic systems.

Acknowledgments

I am grateful to my research advisor, Professor Joseph Oliger, for suggesting the topic of this thesis, and for his encouragement and advice during its preparation. I would also like to thank Professor Gene Golub for his hospitality and for bringing many numerical analysts from around the world to the numerical analysis group at Stanford. I also wish to thank the other members of my reading and oral committees, Professors John Herriot, Joseph Steger and George Homsy.

I wish to thank Dr. Paul Concus for encouragement and support. Also my thanks to Marsha Berger, Phil Colella and Tony Chan for suggestions and technical discussions. I also owe a debt of gratitude to fellow students in the numerical analysis group at Stanford, from whom I learned a great deal, but who are too numerous to mention.

I acknowledge computing facilities used: at the Stanford Linear Accelerator Center, operated for the U. S. Department of Energy by Stanford University; and at the Computer Center of the Lawrence Berkeley Laboratory, operated for the Department of Energy by the University of California.

Finally, I gratefully acknowledge financial support provided by Stanford University as a teaching assistant, by the Office of Naval Research under contract N00014-75-C-1132, and by the Director, Office of Energy Research, Office of Basic Energy Sciences, Engineering, Mathematical and Geosciences Division of the U.S. Department of Energy under contract DE-AC03-76SF00098.

CHAPTER 1

Introduction

In this chapter we will give a justification and motivation for developing a finite difference mesh refinement algorithm, and then give a brief history of adaptive methods for numerical computations. Next, we review some other adaptive algorithms for time-dependent partial differential equations. Finally, we will summarize what is contained in the rest of the thesis.

1.1. Statement of the Problem

Many problems of physical interest have solutions which are smooth in a large portion of the region of interest, but have local phenomena such as shocks, discontinuities or large gradients which require much more accurate approximation or finer meshes for reasonable accuracy. Examples of this are atmospheric fronts, ocean currents, geological discontinuities, and storm surges.

When the positions of the gradients are known *a priori*, and are independent of time, one can use coordinate transformations, a technique used extensively in aerodynamic computations, *e.g.*, Steger and Chaussee [1980].

In more detail, the coordinate transformation technique is as follows. Suppose one wishes to study the two-dimensional flow around an airfoil, viewed in a coordinate system fixed to the airfoil. It is known that steep gradients exist near the surface. Hence a mesh is designed which follows the contours of the airfoil, and in which the mesh size grows exponentially smaller as the surface of the airfoil is approached. This irregular mesh is then mapped (sometimes conformally) onto a rectangular region with uniform mesh. The differential equations governing the flow are similarly transformed. The transformed differential

equations are then solved on the uniform mesh. Finally, the results are transformed back to the original coordinate system.

For a problem in which the position of the gradients is known, and fixed for all time, this method is obviously advantageous. And, if the position of the gradients changes as an *a priori* function of time, the mapping function can change with time (*e.g.*, flow past a helicopter blade). However, when the manner in which the gradients move is not known in advance, this technique cannot be used.

In such a case, one procedure is to use a fine mesh throughout the entire calculation region. But such an approach usually requires too much computer time and/or storage. An alternative method is to use an underlying coarse mesh for the entire region, and to superimpose a fine grid, or grids, on the region(s) where the solution is varying rapidly. The crucial difficulty is that the refined region(s) must then move along with the rapidly varying portion of the solution, at all times enclosing this portion.

The necessity for this is illustrated in Figure 1.1, taken from Browning, Kreiss and Oliger [1973]. The figure illustrates the numerical solution of the initial boundary value problem

$$
\begin{aligned}
u_t &= u_x, && -1 \le x \le 1,\ 0 \le t, \\
u(x,0) &= 0, && -1 \le x \le 1, \\
u(1,t) &= g(t), && 0 \le t,
\end{aligned}
$$

where $g(t)$ is a rapidly oscillating sine wave. In Figure 1.1 we see the result of the computation on a mesh which is divided into a coarse region on the interval $-1 \le x \le 0.495$ and a fine region in the interval $0.495 \le x \le 1$. The mesh width (0.01) in the coarse region is five times the width in the fine region. The figure is plotted after a time in which the influence of the initial condition has almost completely "washed downstream".

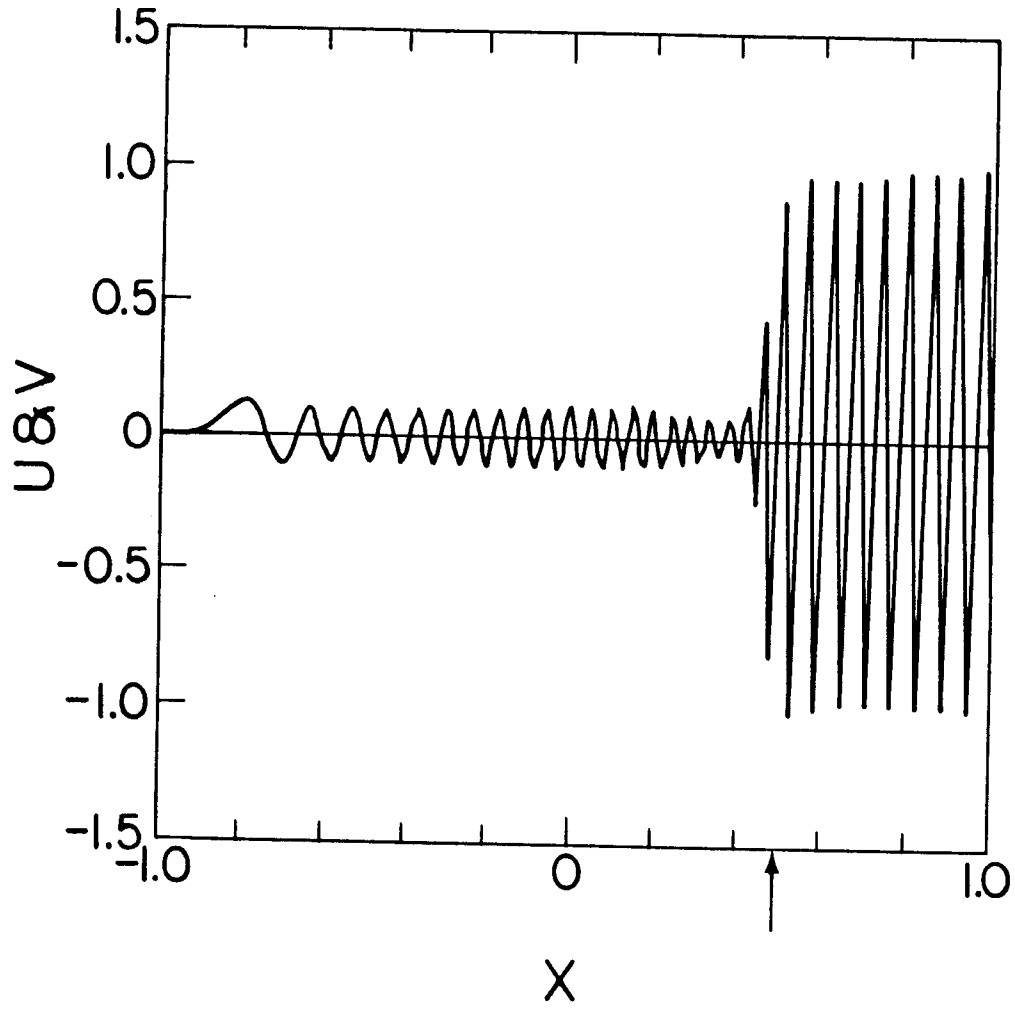


Figure 1.1 Necessity for Moving Refinements

It can be seen that the wave is accurately represented in the fine region (30 points per wave length) but has been mutilated in the coarse region (6 points per wave length). From this it is clear that the rapidly varying part of the solution must not be allowed to escape the refinement region.

1.2. Brief History of Adaptive Methods

We see, then, that the mesh must "adapt" itself to the character of the solution. This is very much in the spirit of recent trends in numerical analysis. A brief, but necessarily incomplete, history of adaptive methods for the solution of numerical problems is therefore in order.

An early and widely used type of algorithm employing adaptive principles was the o.d.e. solver, which solves the initial value problem for a first order system of ordinary differential equations. This was perhaps due to the needs of the U. S. space program in the late 1950's and early 1960's. The first such algorithms used a fixed step size throughout the interval of integration. It was later realized that such a technique was wasteful, and the step size should be finer in regions where the solution varies more rapidly, and conversely. The next step in these programs was the incorporation of a mechanism to halve and double the step size. Such a mechanism required a careful estimation of the local truncation error, together with certain heuristics. Then, to gain further efficiency, integration formulas of varying orders were incorporated, since, for smooth solutions, higher order methods are more efficient than lower order ones. A widely known program of this type is Gear's [1971] DIFSUB.

Still further refinements were the programs of Krogh (DVDQ) [1969] and ODE/DE/STEP of Shampine and Gordon [1975], which allowed truly variable step size, and allowed still higher order integration formulas.

Another area of numerical analysis to use the adaptive approach was quadrature. The first published program of this type, which was written in Algol 60 and recursively subdivided the integration interval, was given by McKeeman [1962]. A more sophisticated analysis by Lyness [1970] led to the program SQUANK (using an adaptive Simpson algorithm) which was in turn superseded by de Boor's CADRE [1971a, 1971b] (using "cautious" adaptive Romberg integration). These methods obtain a sequence of approximations on finer and finer meshes (which need not occupy the entire interval of integration) and use some form of linear extrapolation to determine which parts of the interval to refine further. Also of interest in de Boor's routine is the use of a nontraditional (for numerical analysis) data structure--a stack. This has been carried one step further in QUADPACK [de Doncker, 1978], as typified by routine DQAGS. The Gauss-Kronrod scheme is used, but with *nonlinear*, rather than linear extrapolation, using Wynn's epsilon algorithm. This enables the routine to handle some singularities in the integrand.

A related area using the adaptive approach is multi-dimensional quadrature. Although other workers have proposed adaptive algorithms in this area, the one of greatest interest for our purposes was given by Kahaner and Wells [1979]. The region of integration is divided into finer and finer simplices (not rectangles) and as usual a linear extrapolation is performed to estimate the error. The new ideas that enter here are the use of sophisticated data structures (*i.e.*, heaps, queues, hashing) and programming language. Similar ideas are used in our own algorithm.

An area with a more recent beginning is the adaptive approach to the numerical solution of two-point boundary value problems in ordinary differential equations, as typified by Ascher, Christiansen and Russell's COLSYS [1979] and Lentini and Pereyra's PASVA3 [1977]. Here again, an attempt is made to place more mesh points where the solution varies more rapidly. In order to

implement PASVA3, Pereyra and Sewell [1975] introduced the concept of an *equidistributing mesh*, and this idea, or a slight alteration of it, underlies many of the adaptive algorithms for two-point boundary value problems and elliptic and parabolic partial differential equations. In Section 4.4 we use this idea to justify our own algorithm. The method of deferred corrections was used in PASVA3 to estimate the local error, and in Section 5.2.3 we also use it (in a somewhat disguised form) for the same purpose.

A fifth area using the adaptive approach is elliptic partial differential equations. One adaptive algorithm for this problem was developed by Bank and Sherman [1979] and incorporated in the package PLTMG. This package uses (two-dimensional) triangular finite elements. To decide where to refine, it uses a criterion of Babushka and Rheinboldt [1978]. This criterion requires that a (computable) estimate of the (global) error in the energy norm is approximately equal for all elements. A tree is used to represent the refinement structure.

Another adaptive algorithm for elliptic equations is the multigrid (finite difference) method of Brandt [1977a, 1977b]. Here the differential equation is discretized in the usual manner, but on a sequence of ever-finer grids. Instead of solving the approximations by relaxation independently on each grid, the computation can proceed from one member of the sequence to another in a complicated manner, and the results of one "level" are used to help in the solution at other levels. Although this scheme in its original form did not *automatically* insert more points in regions where the solution changes rapidly, Brandt has indicated how to modify the method (using estimates of the local truncation error) to make it adaptive.

The last area in which adaptive methods are being used is our own of time-dependent partial differential equations.

1.3. Other Adaptive Methods for Time-Dependent Problems

Let us now comment on some other adaptive methods for the initial boundary value problem for time-dependent problems. As in the previous section, this survey is certainly incomplete.

Two other finite difference methods similar to our own are those of W. Gropp [1980] and M. Berger [to appear]. Both of these are for hyperbolic equations in two space dimensions.

Gropp calculated a shock satisfying the two-dimensional inviscid Burgers equation. He used a uniform coarse mesh, with one uniform refinement (which could "move" with time) superposed on it. All meshes were parallel to the coordinate axes. The time step in the refinement was allowed to be smaller than in the coarse mesh. He estimated the error by approximating spatial gradients. Unfortunately, it is difficult to generalize this criterion to other situations (*e.g.*, smooth solutions). (See Section 5.1.) But his results showed the feasibility of this approach. We previously noted that the local truncation error criterion that we use can also locate shocks (although it is probably not the best way to do this).

Berger's algorithm is similar to ours, and is proceeding parallel to our own. (However, she is not handling boundary conditions.) Like us, she uses a tree structure, but her data structure is of necessity more complicated. Her refinements need not be aligned along coordinate axes, but are free to rotate. Her error estimation is the same as ours.

Another finite difference method was developed by Dwyer, Kee and Sanders [1980] for parabolic problems in one and two space dimensions. In two dimensions, they perform a coordinate transformation in one of the spatial variables, and adapt in that variable only. As a criterion for mesh placement, they use an integral of one plus a constant times the absolute value of the first derivative of

the solution. In one space dimension they successfully compute solutions with severe boundary layers.

Brackbill and Saltzman [to appear] have proposed another finite difference method for parabolic problems in two and three dimensions. It also uses coordinate transformations. The criterion for mesh placement is based on a constrained variational formulation. The constraints enforce orthogonality and smoothness of the coordinate transformation. The variational form contains the squared lengths of the gradients of the transformation variables and of the gradient of the (desired) solution. Forming the Euler equations (from the variational formulation) then produces a coupled, time-dependent set of difference equations for the solution and the coordinate transformation, which are then solved. The authors have actually computed (*e.g.*, the convective-transport-reaction-diffusion equation) with this method in two and three space dimensions.

The method of Rai and Anderson [1982] is for one and two dimensional parabolic and hyperbolic equations. Once again, coordinate transformations are used. Mesh points are considered to attract or repel each other; the former where the local truncation error is high, and conversely. The authors then determine the coordinate transformation by laws resembling Newton's laws, with particles replaced by mesh points, charges (or masses) replaced by truncation errors, and the resulting force replaced by the coordinate in the computational plane. The number of mesh points is fixed during the duration of the computation. The authors compute the laminar boundary layer over a flat plate, and flow past a cylinder in a supersonic free stream with associated bow shock (the Euler 2-D gas dynamic equations). They use low order difference methods, and estimate the truncation error by higher order differences. They state that they would like more accurate error estimates.

A striking feature of these adaptive difference methods, and many others, is the wide variety of criteria used to determine the placement of mesh points. We discuss this further in Section 5.1.

Other adaptive approaches use finite elements. Some of these are Gannon's [1980], in two space dimensions for parabolic problems, Davis and Flaherty's [1982] method in one space dimension, also for parabolic problems, and Miller's "moving finite element" method [Gelinas, Doss and Miller, 1981], [Miller and Miller, 1982], in one space dimension for parabolic and hyperbolic problems.

Davis and Flaherty's method is different from the other two. Instead of expanding the solution in spatial basis functions with time-dependent coefficients

$$v(x, t) = \sum_{j=1}^{n} a_j(t)\varphi_j(x) \tag{1.1}$$

as the other methods do, they use a finite element Galerkin method on trapezoidal space-time elements. The time step is uniform and constant. The number of trapezoids is fixed for all time. Given two partitions of the interval $[a, b]$ at times t_m and t_{m+1}, respectively, each of which contains the endpoints a and b and consists of n points, the trapezoids have as their vertices adjacent members of the two partitions. The main problem, then, is choosing a partition at time t_{m+1} when one is known at time t_m. The authors do this by approximately equidistributing the projection error, and hence the global error. Since the partition can be nonuniform, this involves at each step solving a nonlinear system of equations whose Jacobian is block tridiagonal. This method strikes us as quite expensive. However, no computer times are given.

Gannon's Galerkin method uses the standard expansion (3.15), and is based on the results of Babushka and Rheinboldt for elliptic problems. He uses time steps which are variable but uniform in space, and elements which are piecewise

uniform and always parallel to the coordinate axes. The number of elements is not fixed. Unlike ours and Berger's approach, the finer elements are not considered to "overlay" the coarse elements. He proceeds as we do in that an element structure is kept until error estimation is performed; then the elements are adjusted. To estimate error, elements are chosen as in Babushka-Rheinboldt so that an approximation to the (global) error (in energy norm) is approximately equidistributed on elements. At the next time the error is checked, the elements are adjusted if the estimates deviate "too much" from equality. As is usual for Galerkin methods, one uses a stiffly stable o.d.e. solver to step forward in time. One needs to solve a system of equations at each time step, but unlike the Davis-Flaherty method, the system is linear when the differential equation is linear. A tree-type data structure is used to keep track of refinements.

In Miller's moving finite element method, again the standard expansion (1.1) is used. Instead of choosing the coefficients a_j and fixing the basis functions φ_j, the φ_j (which are, for example, piecewise linear "hat" or "chapeau" basis functions) are allowed to have their centers vary (in space) as well. This leads to a modified Galerkin method. Again, a stiffly stable o.d.e. solver is used to step forward in time. Thus, a system of equations is solved at each time step, which is (in general) nonlinear even when the differential equation is linear. The matrix of the system may become singular, so further parameters ("spring constants") are introduced to regularize it. Furthermore, the minimization problem that the Galerkin method solves is replaced by a weighted minimization problem, so the weights must be chosen. The time step is variable but uniform in space.

Gelinas, Doss and Miller illustrate their algorithm with a number of interesting problems. One of them is similar to our P2 (the second order wave equation with counter-streaming Gaussian pulses). Instead, they use square waves. Their squares are resolved almost perfectly. The boundary conditions of this problem

are zero (and in fact disagree with the exact solution at $t = 0$), and we believe their method cannot handle time-dependent boundary conditions (2.3), (2.4) for hyperbolic problems. In contrast, our ("open") boundary conditions for this problem are time-dependent, and neither purely inflow or outflow. They allow the pulses to pass in or out of the region.

Let us summarize a few features common to all these finite element methods. All use a time step which is the same at every spatial point at a given time. (Recent work of T. Dupont [to appear] is an exception.) This is largely due to the use of an o.d.e. solver to advance in time (in two of the methods). All use implicit time-stepping methods. Since we use explicit methods, it is essential to allow finer time steps in refinements than in the coarse mesh. (Otherwise, stability would force us to use tiny time steps throughout the region.) So for hyperbolic systems, the time step in our method is limited by accuracy, not stability.

A significant difference between our method and Miller's, Davis and Flaherty's, Dwyer, Kee and Sander's, and Rai and Anderson's (but not Gannon's) is that we allow a variable number of refinement points as needed. Refinements can be created or destroyed. In contrast, in the other methods, the number of basis functions or mesh points is fixed for all time. In Miller's method, the basis functions do indeed "bunch up" around steep gradients or shocks, as desired. But suppose one started with one steep gradient, and then two or three others developed. In this case, the fixed number of basis functions would either be insufficient or else excessive (and hence wasteful), at some times, but not others.

Thus it is clear, even from this incomplete survey, that adaptive algorithms are playing an increasingly important role in numerical computations, and will continue to do so. Somewhat less clear but still discernible is a trend toward more complicated data structures than vectors and matrices (such as deques,

heaps, stacks and trees), and the need for more flexible programming languages to implement them.

1.4. Summary of Results

We now summarize what is contained in the rest of this study, and point out what we believe to be new and significant.

In Chapter 2 we describe our adaptive mesh-refinement algorithm in detail. The general philosophy and methodology was given in Oliger [1978] and Budnik and Oliger [1977], but we contributed some ideas not in these papers, such as the necessity for recursive refinements and the choice of data structure. We believe this is the first detailed description of the algorithm, and that ours was the first implementation. We first describe the continuous problem, the usual first order hyperbolic system on a strip in one space dimension. Next we describe our mesh structure. We give two descriptions of this; the first is used in the theoretical work in Chapters 3 and 4, and the second is used in describing the algorithm in Chapters 2 and 5. In this second description we define a refinement, and introduce the idea of recursive refinements. Next we state the difference approximations we use. The fundamental restriction is to *explicit* difference methods. For convenience in implementation and error estimation, we also insist on two-(time) level schemes. A detailed description of the algorithm is then provided, including techniques at boundaries and interfaces between coarse and fine meshes. One part of the algorithm description--the estimation of the local truncation error--is deferred until Chapter 5.

In Chapter 3 we give a brief discussion of stability. We show why the stability definition of Gustafsson, Kreiss and Sundström [1972] cannot be used, and state the stability definition of Berger, Gropp and Oliger [to appear]. We then prove (Proposition 3.10) that, if a difference scheme is stable on one horizontal

strip in the $x-t$ plane, then it is stable for any number of strips, under a few weak assumptions (such strips are described in Section 2.2). We did not prove that our algorithm was stable according to this definition, but there is good reason to believe that it is.

Chapter 4 treats convergence of the difference scheme, and relates bounds on the global truncation error to bounds on the local truncation error. We first state (but do not prove) a proposition on the rate of convergence of difference approximations to the solution of the differential equation. This Proposition 4.1 is the analogue of one given by Gustafsson [1975], but for a difference scheme which is stable according to the new stability definition mentioned above. Using this proposition, and the theory of Pereyra and Sewell on equidistribution of meshes arising in approximations to two-point boundary value problems, we prove a relation between the global truncation error and the local truncation errors (Proposition 4.2), which can be said to provide a theoretical justification for our algorithm. This proposition is new, but it is an analogue of a similar theorem for the Cauchy problem given by Oliger [1978].

Chapter 5 discusses the estimation of the local truncation error, which is crucial to the success of the algorithm. We first discuss alternatives to the local truncation error in placing mesh refinements. Then four methods of local truncation error estimation are described. One is totally impractical; another (differences) is marginally successful, and the other two are successful. Only the last one (three-step Richardson extrapolation), however, is general, and convenient to implement for interior approximations. (It was suggested by Oliger.) For this method we prove a theorem (Theorem 5.1) which indicates that this method is valid under quite general circumstances. This theorem is new. The proof of this theorem shows that this algorithm is simultaneously a deferred correction method. We then give a very simple method for error estimation at coarse/fine interfaces which do not abut boundaries. For boundary

approximations, we can sometimes use a modification of the Richardson method. But the most convenient procedure is to rewrite the time derivatives appearing in the local truncation error as spatial derivatives (using the differential equation), and approximate the result by differences.

We believe that our work on adaptive boundary conditions (which can be time-dependent) is not only new but unique. That is, no other algorithm of which we are aware gives a systematic method for adaptively treating time-dependent boundary conditions in hyperbolic systems.

Chapter 6 describes the data structure we used to implement the algorithm. The data structure has two parts--a four-way linked tree of records to hold structural information about refinements, and an array of sequentially allocated deques to hold solution values for the hyperbolic system. We describe our repacking strategies for the deques. The deque structure is a modification of a similar structure for stacks in Knuth [1973]. M. Berger [Ph.D. thesis, to appear] has earlier devised a similar tree structure. In a certain sense a tree structure is "obvious" when recursive refinements are used, and other adaptive methods also use them (*e.g.*, Gannon [1980], Rheinboldt and Mesztenyi [1980]). In each case the tree is modified to suit the application at hand. However, our choice and implementation of the sequentially allocated deques is new, and cannot be generalized to more space dimensions.

Chapter 7 discusses the language used to implement our algorithm. Because Fortran lacks both control structures and data structures, we rejected it. But because of the portability and wide use of Fortran in scientific computation, we had to reject other languages as well. The compromise we chose was Mortran, a macro preprocessor for Fortran [Cook and Shustek, 1975]. Because Mortran is extensible (unlike many other Fortran preprocessors), we were easily able to add records and pointers to it, which made implementation of the data

structures quite convenient. We believe we were the first to use a macro preprocessor to develop adaptive mesh refinement algorithms; recently Gropp [to appear] has done a much more systematic development of a language for these algorithms.

Chapter 8 provides computational results of our algorithm. We first describe three model problems: the first order wave equation (color equation) with traveling pulse; the second order wave equation (rewritten as a first order system) with two oppositely-traveling and interacting pulses, and the inviscid Burgers equation with a shock. In particular, the refinements do properly enclose the pulses or shock at all times. These calculations also show that refinements properly merge, separate, move, and are created and destroyed. (We believe that the only other adaptive algorithm that can track crossing pulses is that of M. Berger, who uses an approach similar to ours in two dimensions.)

Section 8.4 contains the most important result of this thesis, namely, the efficiency of our algorithm. Our model problems show decreases in execution times of factors of 3 to 5 for smooth solutions (compared with using a uniform mesh which achieves the same level of accuracy). Storage savings are achieved as well, but the gains are not so dramatic.

Section 8.5 experimentally shows the rate of convergence of the method as the step size approaches zero, and thus confirms Proposition 4.1. Section 8.6 compares three methods for estimating the interior local truncation error. Section 8.7 compares different boundary approximations and methods for estimating the error of these approximations.

Chapter 9 gives our conclusions and suggestions for further research, and the appendix gives a program listing for one of our model problems.

CHAPTER 2

Mesh Structure and Solution Algorithm

In this chapter we will state the class of partial differential equations to be considered, together with assumptions about the behavior of the solution of the equations. Next we describe, in two different ways, the mesh structure on which we will compute the difference approximation. We then introduce a scalar model problem and describe our algorithm for advancing the solution in time. Finally, we discuss the modifications necessary for systems of equations. The underlying approach throughout is that of Budnik and Oliger [1977] and Oliger [1978].

2.1. The Continuous Problem

Let Ω denote the spatial interval $a \le x \le b$. We will assume given a linear first order, one (space)-dimensional, $n \times n$ hyperbolic system

$$Lu \equiv u_t - A(x,t)u_x - B(x,t)u = F(x,t), \tag{2.1}$$

on a "vertical" strip $\Omega \times \{t \ge 0\}$, with initial condition

$$u(x,0) = f(x), \qquad x \in \Omega, \tag{2.2}$$

and boundary conditions

$$u^{\mathrm{I}}(a,t) = S_{\mathrm{II}}(t)u^{\mathrm{II}}(a,t) + g_1(t), \qquad t \ge 0, \tag{2.3}$$

$$u^{\mathrm{II}}(b,t) = S_{\mathrm{I}}(t)u^{\mathrm{I}}(b,t) + g_2(t), \qquad t \ge 0. \tag{2.4}$$

Here A and B are $n \times n$ matrices and F is an n-vector. We have, as usual, assumed that A has already been transformed into diagonal form by a nonsingular uniformly bounded similarity transformation $T(x,t)$, so that

$$A = \begin{bmatrix} A^{\mathrm{I}} & 0 \\ 0 & A^{\mathrm{II}} \end{bmatrix}$$

with $T(x,t)$ and $T(x,t)^{-1}$ uniformly bounded, and

$$A^{\mathrm{I}} = \mathrm{diag}(\kappa_1, \kappa_2, \ldots, \kappa_J) < 0,$$

$$A^{\mathrm{II}} = \mathrm{diag}(\kappa_{J+1}, \kappa_{J+2}, \ldots, \kappa_n) > 0,$$

$$u^{\mathrm{I}} = (u_1(x,t), u_2(x,t), \ldots, u_J(x,t))^T,$$

$$u^{\mathrm{II}} = (u_{J+1}(x,t), u_{J+2}(x,t), \ldots, u_n(x,t))^T,$$

and

$$S_{\mathrm{I}} \in C^{(n-J)\times J}, \; S_{\mathrm{II}} \in C^{J\times(n-J)}.$$

By far the most important restriction is that our problem has only one space dimension. The problem even for two space dimensions has severe additional difficulties, such as irregular geometries, orientation of refinements, pattern recognition, the need for more complicated data structures, and boundaries. (M. Berger's thesis [to appear] is treating this problem.) The restriction to hyperbolic behavior insures that we can use explicit time steps. This assumption greatly simplifies both the error estimation and the manipulation of moving meshes. However, many computational problems in fluid dynamics and elsewhere are of this type.

The assumption that the matrix A is in diagonal form is not necessary in practice, as shown by computations on problem P2 in Chapter 8. This assumption makes it easier to develop the theory, and to write down boundary conditions (2.3)-(2.4) which yield a well-posed problem.

For the theory in Chapters 3, 4 and 5, we will assume that (2.1), (2.3)-(2.4) have constant coefficients. But in practice, the type of problem can be considerably more general than (2.1)-(2.4). For example, the system of equations can be nonlinear. In Chapter 8 we will show computations for the inviscid Burgers' equation

$$u_t + uu_x = 0,$$

which even has shocks. Furthermore, the problem need not necessarily be hyperbolic. For example, we can treat the Korteweg-de Vries equation

$$u_t + uu_x + \nu u_{xxx} = 0.$$

The important restriction is to equations which allow explicit difference approximations for their efficient solution. Thus the heat equation is excluded. (Our algorithm can accurately approximate the heat equation, but we doubt that it would be more efficient than using an implicit method on a uniform grid.)

We next assume that the overall phenomena being studied are such that, except for relatively small regions, a coarse uniform mesh is sufficient to resolve them. We further assume these small regions change with time in a way which cannot conveniently be determined *a priori*.

We also assume that these small regions are the same for all solution components. In other words, if the differential equations describe velocity and pressure, then large pressure gradients occur in approximately the same regions where large velocity gradients occur. (The assumptions in this paragraph are necessary only for efficiency. The method will work without them, but it might refine too large a portion of the region.)

We assume that we have smooth solutions. This means, first of all, that there are no corner discontinuities, *i.e.*,

$$u^{\mathrm{I}}(a,0) = f(a) = S_{\mathrm{II}}(0)u^{\mathrm{II}}(a,0) + g_1(0)$$

and

$$u^{\mathrm{II}}(b,0) = f(b) = S_{\mathrm{I}}(0)u^{\mathrm{I}}(b,0) + g_2(0).$$

Furthermore, it means there are no shocks present. These assumptions enable us to estimate the local truncation error using higher derivatives of the solution of the differential equation. In practice, our algorithm will work even for shocks

(as mentioned above for Burgers' equation), but then the error estimation is not theoretically justified, and should be done in a different manner for efficiency.

Before giving the difference approximations to our problem (2.1)-(2.4), we must first describe the mesh system on which such a solution is computed.

We will compute on a basic rectangle $R = \Omega \times [0, T]$, We will think of this rectangle as graphed in the $x-t$ plane, with x horizontal and t vertical.

2.2. Mesh Structure - First Description

We will now formally describe our mesh structure in a way suitable for theoretical purposes. We will then describe it a second time in a manner more suited to implementation.

Following Oliger [1978], we divide the rectangle R into "horizontal" strips using time division points

$$0 = t^0 \leq t^1 \leq \cdots \leq t^{s-1} \leq t^s = T \tag{2.5}$$

and define a grid on each strip

$$S_i = \Omega \times [t^{i-1}, t^i], \; i = 1, 2, \ldots, s,$$

as the set of points

$$\Delta_i = \{(x_j^i, t_{j,m}^i) : j = 0,1, \ldots, N_i, \; m = 0,1, \ldots, m_j^i\},$$

where $h_{j+1}^i = x_{j+1}^i - x_j^i > 0$, $t_{j,m}^i = t^{i-1} + mk_j^i$, $k_j^i > 0$, $(t^i - t^{i-1})/k_j^i = m_j^i$, an integer, $x_0^i = a$, and $x_{N_i}^i = b$ for $i = 1,2, \ldots, s$. When $(x_j^i, t_{j,m}^i)$ is a point of Δ_i we will say that k_j^i corresponds to h_j^i. See Figure 2.1. In each strip S_i we will take time steps of equal length at each point x_j^i (although these steps can be different at different x_j^i). We will now restrict our grid to be locally uniform in x: we assume there are a finite set of intervals

$$I_\nu^i \subset [a, b], \; \nu = 1,2, \ldots, \nu_i$$

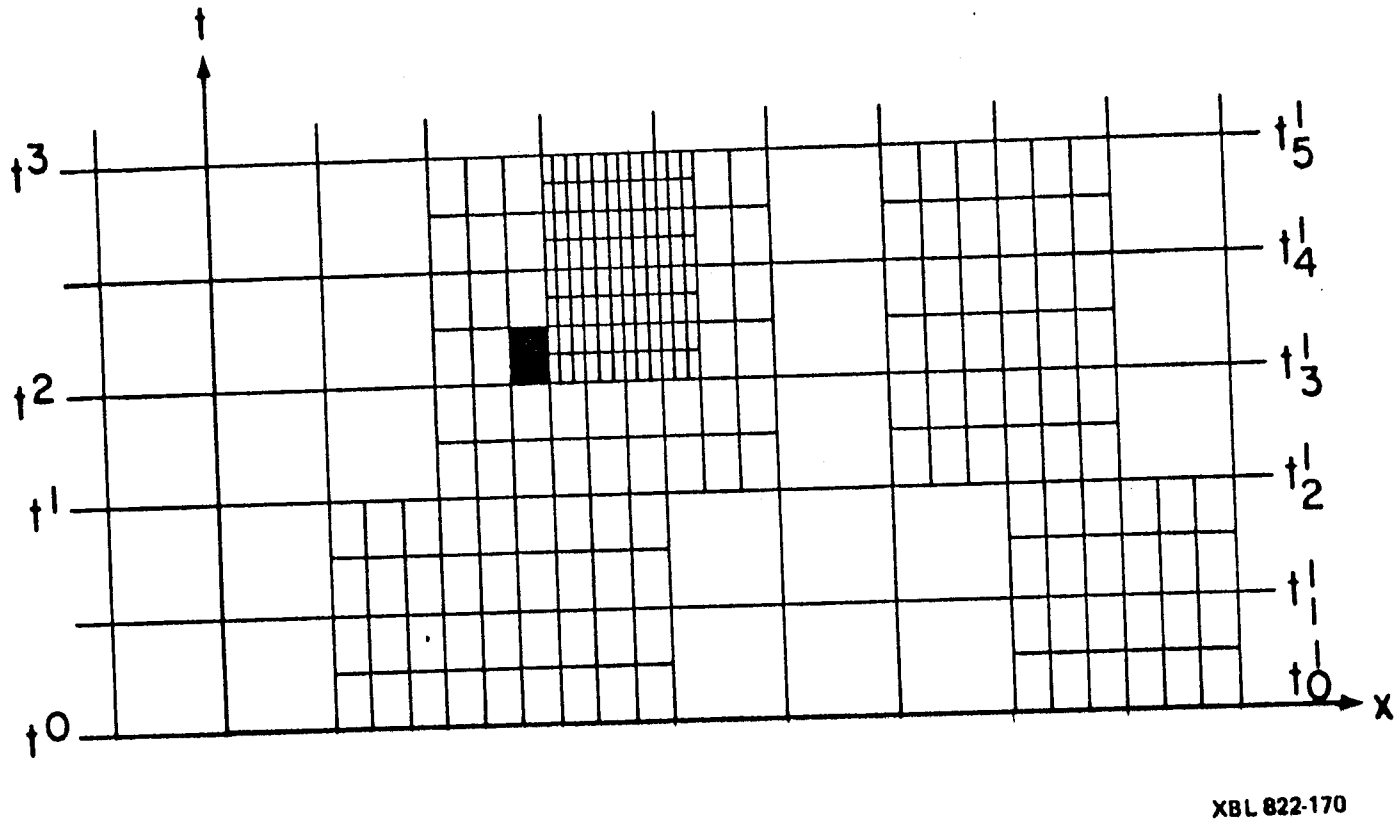


Figure 2.1 Mesh Structure

whose endpoints are among the x_j^i,

$$\bigcup_{\nu=1}^{\nu_i} I_\nu^i = [a, b], \tag{2.6}$$

any two I_ν^i intersect in at most one point, and the x_j^i contained in any I_ν^i are equally spaced. Furthermore, all the k_j^i occurring in $I_\nu^i \times [t^{i-1}, t^i]$ must be equal. Such a grid is uniform over rectangles in the $x-t$ plane. For convenience, we shall assume that for each i the I_ν^i's constitute a minimal set of intervals with these properties. (That is, any other such set describing the same mesh has more than ν_i members.)

To make this grid structure even easier to implement, we will make still further restrictions. Since there are only a finite number of rectangles in the region R, let $h_1, h_2, \ldots, h_\Lambda$ be a list of all the distinct space steps h_j^i, listed in descending order. Corresponding to each h_j^i is a k_j^i. We will then obtain a corresponding list $k_1, k_2, \ldots, k_\Lambda$ which we will also assume has distinct members in descending order. We then require

$$h_{l+1} = Nk_l\ ,\ \ k_{l+1} = Mk_l,\ \ l = 1,2, \ldots, \Lambda-1,$$

where N and M are integers greater than one. This restriction is not crucial, but makes implementation easier.

Let us examine one of the decompositions (2.6). With each I_ν^i is now associated an h_j^i. If I_μ^i is adjacent to I_ν^i, we shall call their intersection a *coarse/fine interface*, and will require that $h_\mu^i = Nh_\nu^i$ or $h_\nu^i = Nh_\mu^i$. That is, the transition in spatial mesh size should be "smooth". No such restriction is made in time. In Figure 2.1 we have illustrated this for $N = 3$, $M = 2$.

We now modify the time steps at the coarse/fine interfaces. In the i-th strip S_i, $i = 1, 2, \ldots, s$, we add grid points to any coarse/fine interface where a fine mesh lies to the right of a coarser mesh. That is, let I_μ be any interval adjacent to, and to the left of, interval I_ν, such that $h_\mu^i = Nh_\nu^i$. According to our

definition, the time steps k_j^i on the interface are the same as those in I_μ^i, not I_ν^i (*i.e.*, they are "coarse" rather than "fine"). We add more points to this interface so that the time steps k_j^i on the interface are the same as those in I_μ^i.

At the time division points t^i, $i = 1,2, \ldots, s-1$, there are two sets of spatial mesh points: those belonging to Δ_i and those belonging to Δ_{i+1}. This is because we readjust the (spatial) mesh at these times. We will introduce two sets of spatial grid points at $t^0 = 0$ also, by letting Δ_0 denote the initial uniform coarse mesh

$$\{(x_j^0, 0),\ x_j^0 = a + jh_1,\ j = 0,1, \ldots, N_0\},$$

where $h_1 = (b-a)/N_0$. We immediately readjust this mesh, getting the mesh in Δ_1. Therefore, we will ignore Δ_0 in our theory, and measure quantities (such as the initial error) with respect to the mesh points in Δ_1 with $t = 0$.

An alternative implementation would allow fully variable h_j^i, as is done in programs for two-point boundary value problems for ordinary differential equations (*e.g.*, PASVA3 and COLSYS, mentioned in Chapter 1.) However, there are several compelling reasons for using our approach. The first is ease of implementation. The second is storage, although this is not as serious. One would need to store a vector of x_j^i's. In our method only a few indices are used. The third reason is that with such a general mesh, the only difference schemes that can be used in two-point boundary-value problems are Keller's box scheme and the trapezoidal rule (if second-order accuracy is desired). Although there do exist second order approximations for the initial boundary value problem (particularly for conservation laws) on a nonuniform spatial mesh, this would severely restrict our choice of available difference schemes. (As we shall see, our method already imposes some other restrictions on the difference scheme.) The final reason is that a general mesh would make analysis and estimation of the local truncation error much more difficult.

2.3. Mesh Structure—Second Description

We will now provide an alternate description of our grid structure. The above description is more suited to the theory; the following one is more suited to implementation and allows slightly more generality than the first description.

We will proceed recursively by "levels of refinement". The word *level* in this context refers not to the time level, but to how fine a grid spacing is. Finer grids will have higher levels. We will use different notation for gridpoints (x, t) than in the previous description. We regret the necessity for this, but certain formulas (*e.g.*, norms) which are natural in one notation become extremely cumbersome in the other.

On each level $l = 0, 1, \ldots, \Lambda-1$ we will introduce a finite number of space-time *refinement rectangles* or boxes B_ν^l contained in rectangle R. (All such rectangles will be *solid*, that is, they include both interior and boundary.) Each such rectangle will have sides parallel to the coordinate axes, and for $l \geq 1$ each l-th level rectangle must lie entirely in an $l-1$-st level rectangle. Furthermore, no two l-th level rectangles can overlap. The boundary of each l-th level rectangle will be the boundary of a uniform $l+1$-st level (space-time) grid. All $l+1$-st level grids will have the same space and time steps. Loosely speaking, an $l+1$-st level refinement is one of these grids viewed at a fixed time.

To prime the recursive pump, we will define the *zero-th level spatial division points* of the interval $[a, b]$ as the sequence of points $<x_0^0 = a, x_1^0 = b>$. Similarly, the *zero-th level time division points* of the interval $[0, T]$ comprise the sequence $<t_0^0 = 0, t_1^0 = T>$. Let $h_0 = b - a$ and $k_0 = T$ be the *zero-th level space* and *time steps*, respectively. We define U^0 as the set of four corner points of the rectangle R.

For $l = 0, 1, \ldots, \Lambda-1$ we now proceed recursively by levels of refinement. We form the *l-th level partition* P_l of $[0, T]$, which is a subsequence of the time

division points $<t_m^l>$:

$$0 = t_0^l < t_{m_1}^l < t_{m_2}^l < \cdots < t_{m_{s_l-1}}^l < t_{m_{s_l}}^l = T. \tag{2.7}$$

Notice that the subsequence $<m_i>$ depends on l; this dependence is omitted from the notation. For $l = 0$, P_0 is identical to the sequence $<t_m^0>$ of time division points. Thus $s_0 = 1$ and $m_1(0) = 1$. For $l \geq 1$, P_l must contain as a subsequence the points in the partition P_{l-1}.

This partition divides the region R into *l-th level* horizontal *strips* S_i^l, $i = 1, 2, \ldots, s_l$. For $l = 0$ the only such strip S_1^0 is identical to the rectangle R. For $l \geq 1$ each of these strips is contained in an $l-1$-st level strip, since P_{l-1} is a subsequence of P_l. The partition points are the times when we adjust the mesh. (The partitions and strips for $l > 1$ could be dispensed with if we never adjust the mesh *between* coarse time steps.)

We will now introduce a set of zero or more nonoverlapping *l-th level* (solid) *refinement rectangles*

$$\{B_\nu^l,\ \nu = 1, 2, \ldots, g_l\}.$$

(If $g_l = 0$ the recursion ends.) There is only one zero-th level rectangle B_1^0, and it is identical to the rectangle R. For $l \geq 1$, each rectangle B_ν^l is required to lie entirely in some $l-1$-st level refinement rectangle B_π^{l-1}. The latter will be called a *parent* of the former. Each such rectangle B_ν^l will have horizontal sides whose t-coordinates are required to be *adjacent* members of the partition P_l (2.7). That is, the horizontal sides of the rectangle are the same as the the horizontal sides of the l-th level strip in which it is contained. Since P_{l-1} is a subsequence of P_l for $l \geq 1$, we are guaranteed that B_ν^l is "vertically contained" in its parent.

For $l \geq 1$, the x-coordinates of the vertical sides of rectangle B_ν^l can be any l-th level spatial division point, so long as B_ν^l is "horizontally contained" in its parent. In other words, let its parent B_π^{l-1} have left and right vertical sides with

coordinates

$$x = x^{l-1}_{\alpha(\pi)} = x^{l}_{N^{(l-1)}\alpha(\pi)} = a + h_{l-1}\alpha(\pi)$$

and

$$x = x^{l-1}_{\omega(\pi)} = x^{l}_{N^{(l-1)}\omega(\pi)} = a + h_{l-1}\omega(\pi),$$

respectively. (Here $\alpha(\pi)$ and $\omega(\pi)$ are nonnegative integers.) Then for the coordinates $x^{l}_{\alpha(\nu)}$ and $x^{l}_{\omega(\nu)}$ of the left and right vertical sides of rectangle B^{l}_{ν}, we require

$$x^{l-1}_{\alpha(\pi)} < x^{l}_{\alpha(\nu)} < x^{l}_{\omega(\nu)} < x^{l-1}_{\omega(\pi)}, \tag{2.8a}$$

i.e.,

$$N^{(l-1)}\alpha(\pi) < \alpha(\nu) < \omega(\nu) < N^{(l-1)}\omega(\pi). \tag{2.8b}$$

For $l \geq 0$, let $N^{(l)}$ and $M^{(l)}$ (the l-th level spatial and time *refinement ratios*) be integers greater than one. (For $N^{(0)}$ we shall take the N_0 of the last section.) Let $h_{l+1} = h_l / N^{(l)}$ and $k_{l+1} = k_l / M^{(l)}$ be the *l+1-st level space* and *time steps*, respectively. We now define the sequences of (uniform) *l+ 1-st level spatial* and *time division points*

$$<x^{l+1}_j = a + jh_{l+1}: j = 0,1, \ldots, \prod_{\mu=0}^{l} N^{(\mu)}>,$$

and

$$T^{l+1} = <t^{l+1}_m = mk_{l+1}: m = 0,1, \ldots, \prod_{\mu=0}^{l} M^{(\mu)}>,$$

of the intervals Ω and $[0, T]$, respectively. They are respectively $N^{(l)}$ and $M^{(l)}$ times as fine as the l-th level ones. The set of all points

$$U^{l+1} = \{(x^{l+1}_j, t^{l+1}_m)\}$$

occupies the entire rectangle $R = \Omega \times [0, T]$. The subset of these points contained in the (solid) refinement rectangle B^{l}_{ν} is defined to be the $(l+1)$-st *level* (space-time) *grid* G^{l+1}_{ν} occupying B^{l}_{ν}. More specifically, if B^{l}_{ν} occupies the l-th

level horizontal strip S_i^l, then G_ν^{l+1} consists of that subset of U^{l+1} whose x components have subscripts

$$j = \alpha(\nu)N^{(l)}, \alpha(\nu)N^{(l)}+1, \ldots, \omega(\nu)N^{(l)},$$

and whose t components have subscripts

$$m = m_{i-1}(l)M^{(l)}, m_{i-1}(l)M^{(l)}+1, \ldots, m_i(l)M^{(l)}.$$

(Recall that the subsequence $<m_i>$ depended on the level l.) This completes our recursive definition.

Now we come to the most important definition of this thesis.

Definition. Let G_ν^{l+1}, $l = 0, 1, \ldots, \Lambda-1$, be an $l+1$-st level grid, occupying an l-th level rectangle B_ν^l, whose mesh points are as given above. Let t be any time such that

$$t^l_{m_{i-1}(l)} \le t \le t^l_{m_i(l)}, \tag{2.9}$$

and let t_m^{l+1} be the greatest $l+1$-st level time division point not exceeding t. An *$l+1$-st level refinement* at time t, corresponding to B_ν^l or G_ν^{l+1}, is a sequence of ordered pairs

$$R_\nu^{l+1}(t) = <(x_j^{l+1}, v_j^{l+1}(t_m^{l+1})): j = \alpha(\nu)N^{(l)}, \alpha(\nu)N^{(l)}+1, \ldots, \omega(\nu)N^{(l)}>;$$

the first components comprise the sequence of $l+1$-st level spatial division points contained in the horizontal sides of the refinement rectangle B_ν^l (equivalently, the sequence of x components of the grid points in G_ν^{l+1}); the second components are the approximate solution values (if any) evaluated at these spatial points, but at time t_m^{l+1}. Here $v_j^{l+1}(t)$ is an approximation to the vector $u(x_j^{l+1}, t)$.

An important property of our definition is that an $l+1$-st level refinement exists not only at $l+1$-st level time division points T^{l+1}, but also at "finer" time division points $T^{l+2}, \ldots, T^\Lambda$ satisfying (2.9). (Alternatively, we could have

defined refinements only for times T^A satisfying (2.9).) However, solution values for an $l+1$-st level refinement are only updated at $l+1$-st level time division points. In the next section we will see why we defined a refinement as a *sequence* rather than a set.

For $l \geq 1$ let B_ν^l be any refinement rectangle, and R_ν^{l+1} its corresponding refinement. A vertical side of B_ν^l which does not lie on the boundary of the region R will be called a *coarse/fine interface*. Similarly, the left or right end-point of R_ν^{l+1} will also be called a coarse/fine interface if it does not lie on the left or right boundary of the region R.

The first level (or *coarse*) space-time grid occupies the whole rectangle $B_1^0 = R$. Hence, the first level, or coarse, refinement is present at all times, and higher level refinements are considered to be superimposed on it. (Strictly speaking, we should not call this a refinement, since it doesn't refine anything. We use this terminology to avoid special cases.) We will assume as given the largest wave propagation speed. This is usually known by the problem originator, and determines the spacing of the coarse refinement.

Another factor which must determine the spacing of the coarsest refinement is the wavelength of any "background disturbances" to the phenomenon of interest (see our model problem P1 later in this chapter for an example). This too is assumed known; for guides to the number of mesh points needed per wave length, see Kreiss and Oliger [1972].

We will now discuss some further restrictions imposed on our refinement rectangles. We will require that no two l-th level refinement rectangles in the same l-th level horizontal strip can intersect or abut. (But l-th level rectangles in adjacent strips may abut.) Assume an l-th level strip contains two l-th level rectangles B_μ^l and B_ν^l, having left and right vertical sides with x coordinates

$$x^l_{\alpha(\mu)},\ x^l_{\omega(\mu)} \text{ and } x^l_{\alpha(\nu)},\ x^l_{\omega(\nu)},$$

respectively. Without loss of generality, assume that the left side of the former is to the left of the left side of the latter, $\alpha(\mu) < \alpha(\nu)$. Then

$$\omega(\mu) < \alpha(\nu).$$

This is no restriction in practice; if two such rectangles overlap or abut, we simply consider them to be one rectangle.

In the last section we mentioned that the mesh should vary "smoothly" in space -- *i.e.*, an l-th level refinement can abut a $l+1$-st or $l-1$-st level refinement, but not a refinement of any other level. This restriction is enforced by inequality (2.8a or b), which says that the rectangle B^l_ν is ***properly*** "horizontally contained" in the parent rectangle B^{l-1}_π. Actually, this restriction is too severe because of boundaries. In (2.8) we allow the leftmost inequality to become "$\leq$" when the parent rectangle abuts the left boundary, *i.e.*, $\alpha(\pi) = 0$. Similarly, when the parent B^{l-1}_π abuts the right boundary ($\omega(\pi) = \prod_{\mu=0}^{l-2} N^{(\mu)}$), we allow the rightmost inequality to become "$\leq$". In particular, if an $l-1$-st level refinement occupies the whole spatial region and is too coarse over the whole region (according to our error estimates), then the l-th level refinement will occupy the entire region.

Let $\lambda_l = k_l / h_l$. Then our construction ensures that $\lambda_l = \text{constant}_l$. For simplicity, our implementation restricts the refinement ratios for $l \geq 1$ to be the same, *i.e.*, $N^{(l)} = N$ and $M^{(l)} = M$, for $l = 1, 2, \ldots, \Lambda-1$. This condition is not essential, but it poses no real restriction, as we will see in Section 8.3. For convergence studies, we shall in addition assume that $M = N$, so that λ_l = constant, independent of l.

Suppose we want both descriptions to characterize the same mesh points. How must we modify these descriptions to achieve this? Let us consider the

situation in time first. We claim that the second description is more general than the first. To see why this is so, consider the blackened rectangle in Figure 2.1. If this rectangle contains no interior mesh point, the mesh of Figure 2.1 satisfies the first description. The time division points t^0, t^1, t^2, t^3 are shown. These points also comprise the partition P_1 of the second description.

However, if the blackened rectangle contains six subrectangles, then this mesh satisfies the second description but not the first. This illustrates the criterion for the first description to coincide with the second:

Proposition 2.1. In the second description of the grid structure, choose a first-level partition P_1 of the interval $[0, T]$:

$$0 = t_0^1 < t_{m_1}^1 < t_{m_2}^1 < \cdots < t_{m_{s_1}}^1 = T.$$

If all succeeding partitions P_l, $l = 2,3, \ldots, \Lambda-1$ (2.7) consist of exactly the same points as P_1, then the partition P_1 of the second description coincides with the partition (2.5) of the first description.

For, the first description requires the t coordinates of the horizontal sides of all refinement rectangles to be adjacent members of the partition (2.5) of the first description; this will be the case for the second description only if no new points are introduced when constructing partition P_l from P_{l-1}, $l = 2, 3, \ldots, \Lambda-1$. By the assumption in the first description that there are only a finite number of step sizes $k_1, k_2, \ldots, k_\Lambda$ which are multiples of each other, the partition points (2.5) are a subsequence of a set of equally spaced points, just as P_1 is.

Our theory will assume the two descriptions coincide, and thus will use only the first level partition, in the notation of (2.5). In this case, all horizontal strips S_i^l of the second description coincide with the first level strips S_i^1. Henceforth, we shall drop the superscript 1 for strips, which is the notation used in the first description. Also, s_1, the number of strips, is shortened to s.

In practice, when we choose partitions as in this equivalence proposition, we usually check the local error every ϑ coarse time steps, where ϑ is a small positive integer. Thus every partition P_j, $j \geq 1$ is of the form

$$0 = t_0^1 < t_\vartheta^1 < t_{2\vartheta}^1 < \cdots < t_{(s-1)\vartheta}^1 < t_{s\vartheta}^1 = T.$$

In Section 8.8 we use the capability to check the error, and adjust refinements, between coarse time steps. For the model problem studied there, we find that this is no more efficient than using the partitions as above with $\vartheta = 1$. (But this conclusion may not be generally true; see Section 8.8.)

The partitions P_j must be chosen *a priori*, before the solution of the problem.

Now consider the situation in space. We modify the second description as follows. Let $(x,t) \in S_i$ be any point which is a grid point of more than one grid B_ν^{l+1}, each such grid (and its associated refinement rectangle B_ν^l) lying entirely in strip S_i. We shall say that the point (x,t) is covered by more than one mesh point. Then all such grids (resp. refinement rectangles) must be at different levels of refinement. At such a point, delete all but the grid point on the finest level. Then, except possibly for times $t = t^i$, each point $(x,t) \in R$ is covered by at most one grid point.

Since the first definition allows no overlapping mesh points except possibly for times $t = t^i$, the grid points of the first and second descriptions now coincide. But how do the rectangles of the two descriptions relate? In strip S_i, $i = 1, 2, \ldots, s$, let the highest level refinement rectangle be B_μ^γ. (Note that γ depends on i.) Then B_μ^γ is identical to one of the rectangles $I_\nu^i \times [t^{i-1}, t^i]$ in the first description. If $B_\pi^{\gamma-1}$ is a refinement rectangle in this strip with the next highest level, it will correspond to the union of three, two or one adjacent rectangles $I_\nu^i \times [t^{i-1}, t^i]$ of the first description (three if B_μ^γ abuts no boundary, two if it abuts one boundary, and one if it occupies the whole strip S_i).

In general, if no rectangle in the i-th strip abuts a boundary, the l-th level rectangle (of the second description) corresponds to the union of $2(\gamma-l)+1$ adjacent rectangles of the first description. If some rectangle (of the second description) abuts a boundary, then the number of rectangles in the union will be fewer. Thus, it is quite inconvenient to define refinements in the first description.

2.4. Operations on Refinements

In the last section, we observed that l-th level refinement rectangles in the strip S_i may not intersect or abut, but those in adjacent strips S_i and S_{i+1} may abut. This leads to interesting consequences for refinements. For simplicity, we shall assume that all partitions P_l are the same for $l \geq 1$, and use the notation (2.5) for P_1.

We shall say that two refinements are *equivalent* when their first components (x coordinates) are the same, regardless of the time or the solution values. Thus, for all times (2.9) encompassed by the refinement rectangle B_ν^l, the refinements $R_\nu^{l+1}(t)$ corresponding to B_ν^l are equivalent. In this sense, we can say that to each rectangle B_ν^l or grid G_ν^{l+1} there corresponds *one* refinement. This equivalence concept is useful for describing refinement manipulations which do not depend on the differential equation calculations. Clearly, only refinements with the same level can be equivalent.

Suppose first that there is an l-th level refinement rectangle B_μ^l ($l > 0$) contained in the strip $S_i = \Omega \times [t^{i-1}, t^i]$. Assume that the horizontal sides of B_μ^l occupy the interval

$$x_{\alpha(\mu)}^l \leq x \leq x_{\omega(\mu)}^l.$$

Also assume that no part of any l-th level refinement rectangle in strip S_{i-1} lies in this interval. Then we will say that the refinement R_μ^{l+1} corresponding to B_μ^l

has been *created* at time $t = t^{i-1}$. Similarly, if we replace S_{i-1} by S_{i+1} and t^{i-1} by t^i, we say that R_μ^{l+1} has been *deleted* at time t^i.

Now suppose there are two l-th level refinement rectangles $B_\mu^l \subset S_i$ and $B_\nu^l \subset S_{i+1}$. According to our definition, the refinement R_μ^{l+1} corresponding to B_μ^l only exists for $t^{i-1} \le t \le t^i$, and the refinement R_ν^{l+1} corresponding to B_ν^l only exists for $t^i \le t \le t^{i+1}$. We will now examine the possible relationships between these refinements.

Suppose the rectangles B_μ^l and B_ν^l have the same left and right sides, $\alpha(\mu) = \alpha(\nu)$ and $\omega(\mu) = \omega(\nu)$. Then the first components of the refinements corresponding to these rectangles are the same. By our definition, the refinements corresponding to B_μ^l and B_ν^l are equivalent. In this sense we may say that a single refinement now exists for times $t^{i-1} \le t \le t^{i+1}$.

Now suppose that the refinement rectangles are situated as before, but

$$\alpha(\mu) \le \alpha(\nu) < \omega(\nu) \le \omega(\mu),$$

(with at most one equality), and no part of any other l-th level refinement rectangle in strip S_{i+1} lies in the interval

$$x_{\alpha(\mu)}^l \le x \le x_{\omega(\mu)}^l.$$

Then we will say that the refinement R_μ^{l+1} has *contracted* at $t = t^i$ to form the refinement R_ν^{l+1}. By interchanging refinement rectangles and strips, respectively, an analogous definition can be given for an *expanding* refinement.

If B_μ^l and B_ν^l are situated as before, but

$$\alpha(\mu) < \alpha(\nu) < \omega(\mu) < \omega(\nu),$$

and no part of any other l-th level refinement rectangle in strips S_i or S_{i+1} occupies the interval

$$x_{\alpha(\mu)}^l \le x \le x_{\omega(\nu)}^l,$$

then refinement R_μ^{l+1} has *moved right* at $t = t^i$ to become the refinement R_ν^{l+1}. Analogously, we can define what it means for a refinement to *move left*.

Finally, suppose rectangle B_μ^l is in strip S_i as before, but strip S_{i+1} contains two (disjoint) l-th level refinement rectangles B_ν^l and B_π^l, with the former to the left of the latter. Assume that

$$\alpha(\nu) \leq \alpha(\mu) < \omega(\nu) < \alpha(\pi) < \omega(\mu) \leq \omega(\pi),$$

and that no part of any other l-th level refinement rectangle in strips S_i or S_{i+1} lies in the interval

$$x_{\alpha(\nu)}^l \leq x \leq x_{\omega(\pi)}^l.$$

Then the refinement R_μ^{l+1} is said to *separate* or *split* into refinements R_ν^{l+1} and R_π^{l+1}. Analogous definitions can be given for two refinements to *merge* into a third.

The above are typical operations on refinements, but they do not exhaust the possibilities (for example, a refinement could split into three refinements, although this is quite rare). Fortunately, however, an exhaustive listing is not needed. All that is required is an algorithm which takes a set of l-th level refinements ($l > 1$) at time $t = t^i$, $i = 0, 1, \ldots, s-1$ and produces a new set of such refinements. For each l, once the left and right edges of the new refinements are determined (by local error estimates), this readjustment can be done in a single left-to-right scan of the existing l-th level refinements.

One might ask why all this is necessary. The answer was given in Section 1.1, where we noted that we must not allow the information in "fine" refinements to escape into "coarse" ones. Thus, we cannot throw away any "information" (the second components of refinements) from "fine" refinements (unless the error estimates allow it).

In Section 2.6 we will see how these operations fit into our overall algorithm. In Chapter 6 we will explain how these operations are implemented.

2.5. Difference Approximation

Having described the grid structure, we can now define our difference approximations. We will first describe the general form of difference schemes allowed, then give our first model problem, and finally specify the particular difference schemes used on this problem. We will use the notation of Section 2.3 throughout.

In general, we will compute with explicit two (time)-level difference approximations to (2.1) in the interior of refinements,

$$v_\nu^l(t+k_l) = Q_0 v_\nu^l(t) + k_l F_\nu^l(t), \tag{2.10}$$

where $t = t_m^l$,

$$Q_0 = Q_0(l) = \sum_{j=-r}^{q} A_j(x_\nu^l + jh_l, t, h_l)E^j ,$$

$E = E(l)$ is the shift operator

$$E_j v_\nu^l(t) = v_{\nu+j}^l(t),$$

q and r are nonnegative integers, $v_\nu^l(t)$ is an approximation to $u(x_\nu^l, t)$, and $F_\nu^l(t) = F(x_\nu^l, t)$. (By the *interior* of a refinement, we mean all its points except the r leftmost and q rightmost ones.) As initial condition we use

$$v_\nu^1(0) = f(a + \nu h_1), \quad \nu = 0, 1, \ldots, N_0. \tag{2.11}$$

The coefficients A_j are assumed to depend smoothly on their arguments.

The restriction to two-level schemes is necessary to simplify manipulations with refinements. (When the spatial mesh is adjusted at time t^i, $i = 0, 1, \ldots, s-1$, it would be awkward, and require more storage, to adjust the mesh at previous time levels too.) This also simplifies error estimation. If a

three-level scheme were used with Richardson extrapolation-type error estimation (to be discussed in Chapter 5), then several additional past time levels would have to be saved. This would be highly impractical in multidimensional problems. Other than this storage limitation, there is no difficulty with error estimation for multi-level explicit schemes. (This restriction does not exclude two-level schemes with fractional time steps, such as two-step Lax-Wendroff.)

The restriction to explicit schemes is more fundamental. As we have observed, this is no restriction for purely hyperbolic problems, but can be a restriction for more complicated problems (*e.g.*, coupled heat and sound). As we will see, our algorithm calculates solutions at a given time level piecewise in various parts of the interval $a \le x \le b$. Obviously, then, the restriction to explicit schemes is not merely for convenience.

In order to use the most convenient form of error estimation given in Chapter 5 (three-level Richardson extrapolation), we shall make an additional restriction on the interior approximation: *The local truncation error (per unit time step) must have the same order in both space and time.* If this restriction does not hold, or the approximation is implicit, we must use difference approximations to high-level derivatives, which is less convenient. Since we will most often use interior approximations which are second order in space and time, this restriction is not too severe.

At coarse-fine interfaces (between level $l-1$ and level l refinements) we use the same scheme as above on $l-1$-st level spatial mesh points, but with an $l-1$-st level spatial step and (an integer multiple of) an l-th level time step. This will be explained in more detail in the next section.

Finally, boundaries are treated the same as with a uniform mesh; at the left boundary,

$$v_\mu^l(t+k_l) = \sum_{\sigma=-1}^{0} S_\sigma^{(\mu)} v_r^l(t-\sigma k_l) + g_\mu^l(t), \qquad \mu = 0,1,\ldots,r-1, \tag{2.12}$$

where

$$S_\sigma^{(\mu)}(l) = \sum_{j=-\tilde{r}}^{\tilde{q}} C_{j,\sigma}^{(\mu)}(x_r^l + jh_l,\ t-\sigma k_l,\ h_l)E^j, \qquad \sigma = 0,-1$$

$t = t_m^l$, $\tilde{r} \le r$, and $C_{-\tilde{r}+\mu,-1}^{(\mu)} = 0$. The approximation at the right boundary is analogous.

Once again we have restricted ourselves to two time levels, for the same reasons as before. We allow ourselves implicit boundary conditions here ($S_{-1}^{(\mu)} \neq 0$) since we can first solve for the points on the right hand side of (2.12) using the explicit interior approximation.

Notice that the boundary formulas apply on any refinement level. If a level l refinement abuts the left or right boundary, all the subscripts and operators refer to the l-th refinement level, not the first level.

If we assume that the boundary approximation is explicit, and that its local truncation error (per unit time step) has the same order in space and time, then once again we can estimate the error using 3-level Richardson extrapolation. But we do not do make this assumption, not only because it excludes too many boundary approximations, but also because differences are less inconvenient here.

We will now introduce our first model problem. It will be used both in our computations in Chapter 8, and to help describe our algorithm in the next section. It is the first order wave equation ("color equation")

$$\begin{aligned} u_t &= -cu_x, & a \le x \le b,\ 0 \le t,\ 0 < c, \\ u(x,0) &= g(x), & a \le x \le b, \\ u(0,t) &= g(-ct), & 0 \le t, \end{aligned} \tag{P1}$$

with exact solution $u(x,t) = g(x-ct)$. We take $a = 0$, $b = 4$, and $c = 1$. The

function g is taken to be a Gaussian pulse, traveling to the right with speed c, superimposed on a sinusoidal background,

$$g(x) = \exp(-\alpha(x+\tfrac{1}{2})^2) + 0.1\sin 2\pi(x+\tfrac{1}{2}),$$

with $\alpha = 200$. The parameter α control the steepness and thickness of the pulse. For $\alpha = 200$, the pulse occupies about 8 percent of the interval $[0, 4]$. Figure 2.2 gives an illustration of the trajectory of the pulse. This models more realistic problems such as an atmospheric front or storm surge.

We will consider two different finite difference approximations to this problem. In the first method we use a second order method (Lax-Wendroff) on all refinements. In the second method, we use a fourth-order approximation (Oliger, [1974]) on the coarsest refinement, and a second-order method (Lax-Wendroff) on all other refinements. This is to better resolve the sinusoidal background.

We will need to define the forward, backward, and centered difference operators D_+^l, D_-^l, and D_0^l, operating on l-th level refinements:

$$D_+^l v_\nu(t) = h_l^{-1}(E-1)v_\nu(t) = (v_{\nu+1}(t) - v_\nu(t))/h_l,$$
$$D_-^l v_\nu(t) = h_l^{-1}(1-E^{-1})v_\nu(t) = (v_\nu(t) - v_{\nu-1}(t))/h_l,$$
$$D_0^l v_\nu(t) = (2h_l)^{-1}(E-E^{-1})v_\nu(t) = (v_{\nu+1}(t) - v_{\nu-1}(t))/2h_l,$$

where we have omitted the superscript l on v. More generally,

$$D_0^l(jh_l)v_\nu(t) = (2jh_l)^{-1}(E^j - E^{-j})v_\nu(t) = (v_{\nu+j}(t) - v_{\nu-j}(t))/2jh_l,$$

for $j = 1,2,\cdots$ Also, for $l = 1, 2, \ldots, \Lambda$ let $\lambda_l = k_l/h_l$.

The Lax-Wendroff approximation to our model problem in the interior of a refinement (with $t = t_m^l \equiv mk_l$) is

$$v_j^l(t+k_l) = (I - ck_l D_0^l + \tfrac{1}{2}c^2k_l^2 D_+^l D_-^l)v_j^l(t). \tag{2.13}$$

We use the prescribed values

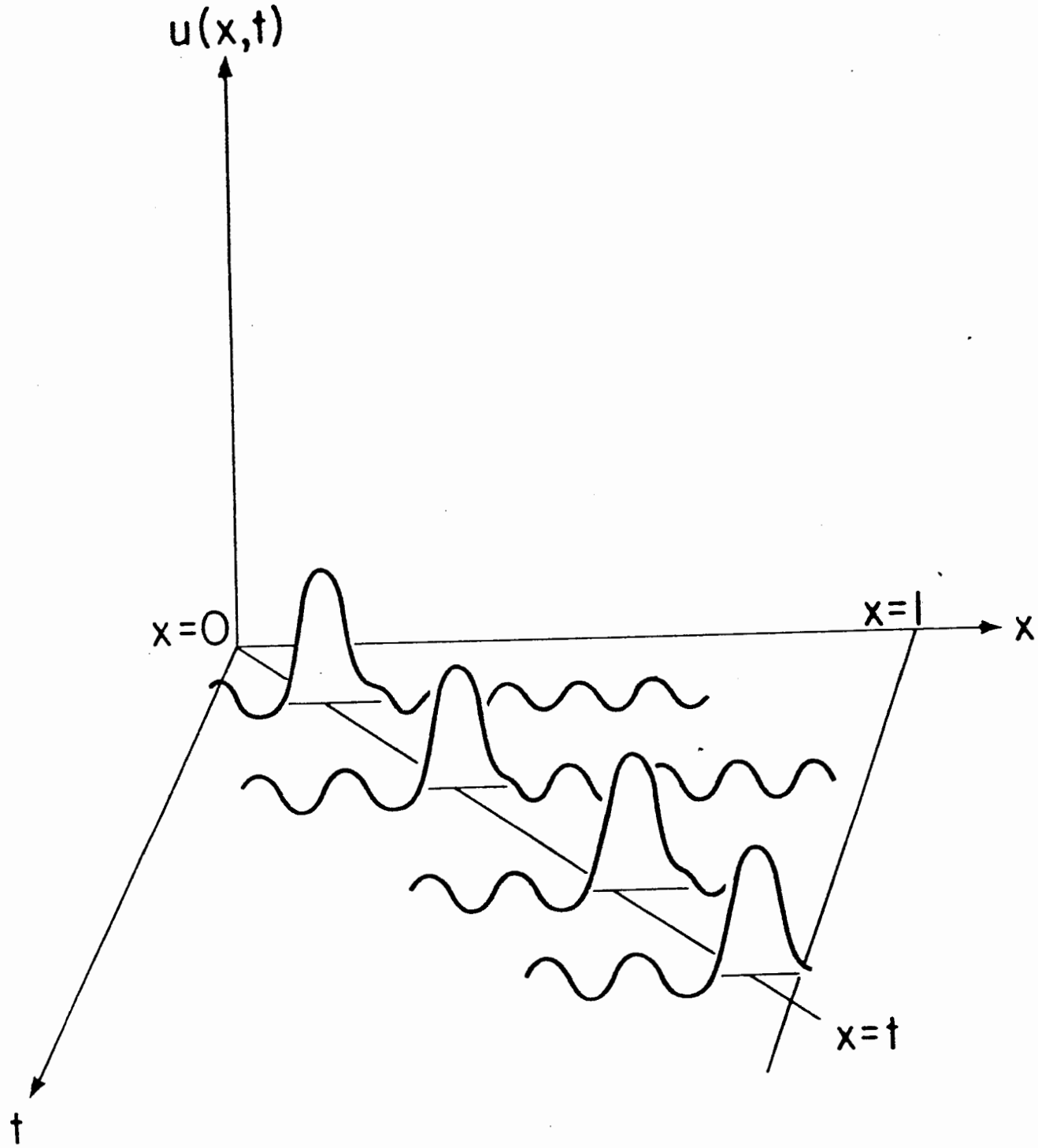


Figure 2.2 Gaussian Pulse Solution of First Order Wave Equation

$$v_j^l(t+k_l) = g(-c(t+k_l)) \tag{2.14}$$

at the left boundary; and upwind differencing

$$v_j^l(t+k_l) = (I - ck_l D_-^l)v_j^l(t) \tag{2.15}$$

at the right boundary.

The formulas (2.14) and (2.15) are only used if a refinement abuts a left or right boundary, respectively. We will explain later what to do at interfaces between refinements.

The fourth order approximation (with $k_1 = k$, and omitting the superscript 1 on v) is

$$\begin{aligned} v_j(t+k) &= v_j(t-k) - 2ck\left(\frac{4}{3}D_0^1(h_1) - \frac{1}{3}D_0^1(2h_1)\right)v_j(t) \\ &= v_j(t-k) - c\lambda_1(v_{j-2}(t) - 8v_{j-1}(t) + 8v_{j+1}(t) - v_{j+2}(t))/6 \end{aligned}$$

in the interior of the coarse refinement. By using a four-point one-sided difference approximation to u_x, we obtain the (third-order in space, second order in time) approximation at the right boundary (with $j = N_0$)

$$\begin{aligned} v_j(t+k) = v_j(t-k) &- c\lambda_1(-2v_{j-3}(t) + 9v_{j-2}(t) + 18v_{j-1}(t) \\ &+ 5.5(v_j(t+k)+v_j(t-k)))/3. \end{aligned}$$

We use (2.14) at the left boundary, with $l = 1$. In addition, we have to use special approximations for points which are a distance h_1 from the left and right boundaries. Again, these approximations result from using four-point uncentered difference approximations to u_x. They are

$$\begin{aligned} v_j(t+k) = v_j(t-k) &- c\lambda_1(-2v_{j-1}(t) - 1.5(v_j(t+k) + v_j(t-k)) \\ &+ 6v_{j+1}(t) - v_{j+2}(t))/3 \end{aligned}$$

for points a distance of h_1 from the left boundary (*i.e.*, $j = 1$); and

$$v_j(t+k) = v_j(t-k) - c\lambda_1(v_{j-2}(t) - 6v_{j-1}(t) + 1.5(v_j(t+k) + v_j(t-k))$$

$$+ 2v_{j+1}(t))/3$$

for points at distance h_1 from the right boundary (*i.e.*, $j = N_0-1$).

2.6. Solution Algorithm

We now describe our algorithm on the model problem. We will explain the method which uses the Lax-Wendroff approximation on all refinements. For concreteness, assume we have the underlying coarse refinement 1, on which is superimposed one finer refinement 2. (Usually the spatial region covered by refinement 2 is a proper subset of the region covered by the coarse mesh.) Superimposed on refinement 2 (but covering only a part of the region occupied by it) is a still finer refinement 3. This is an example of a recursive refinement. The general case, in which there can be several refinements superimposed on refinement 1, and even further recursive refinements, will then be clear.

As in the initial value problem for ordinary differential equations, we will need to give a tolerance δ on the local truncation error, which will be used to decide where to refine the mesh. (We have only used absolute error since all of our example problems vary between 0 and 1. In general, one should use a combination of relative and absolute error, as in Shampine and Gordon [1975].)

For the initial value problem for o.d.e.'s, Stetter [1979], and others, have shown how to estimate (but not control) the global error while the solution is being computed. This requires only a small amount of additional computation and storage for that case. Further investigation would be needed to apply this to the initial boundary value problem. But even if were done, one would still need to prescribe a local error tolerance.

We have implemented the algorithm, and estimated the error for this model problem in a way which applies to more complicated problems. For example, in our model problem P1 one boundary is an inflow boundary and the other is an

outflow boundary. Our difference schemes and error estimation do not take advantage of this fact. The difference schemes on refinements use the same treatment at the left and right coarse/fine mesh interfaces (except when a refinement abuts the left or right boundary of the region). A later model problem (P2 in Chapter 8) will show that our algorithm is indeed insensitive to the direction of characteristics.

Assuming we have a solution on all mesh points at time $t = nk_1$, we proceed to time $t = (n+1)k_1$ by advancing on the highest level refinements first, then the next highest, etc. This can be described as working "inside out". (One can also proceed from the coarsest level to the finest, and this may be advantageous for some types of problems.)

1. If the time t is a member of the partition P_1, we first estimate the local truncation error that would be made *if* we took one forward time step in the level l refinement for $l = 3, 2, 1$ (this estimation is discussed more fully in Chapter 5), but we do not actually take the step. Mesh points whose advancement would exceed the (absolute value of the) local error tolerance are marked as needing refinement. These points are grouped into intervals. Several extra "buffer" mesh points are added to both ends of each such interval. This will be explained later.

For our discussion here, we will assume that the level 3 refinement produces *no* level 4 intervals, and levels 2 and 1 produce exactly *one* level 3 and level 2 interval, respectively.

In general, there may be more than one refinement at each level (except the first). In that case, the operations are done for all refinements on a given level, starting with the leftmost refinement.

2. For $l = 3, 2$ compare the intervals produced in step 1 with the existing refinements. If these are not identical, refinements may have to be "moved",

created, deleted, merged or separated. If refinement l moves into a region formerly occupied only by refinement $l-1$, we may need solution values that do not yet exist at mesh points in refinement l. These are obtained by linear or quadratic interpolation in space from solution values on the next coarser (parent) $l-1$-st level refinement.

Creation of a new refinement is done the same way, by spatial interpolation from its parent refinement. At any mesh adjustment time, an $l-1$-st level parent refinement can give birth to any number of l-th level refinements, but no higher level ones. An exception is made at $t = 0$. If refinement(s) of the coarse mesh are needed at that time, we obtain the new solution values directly from the initial function f rather than from interpolation. This allows us to add as many levels of refinement as are necessary. Thus, the method performs properly even when the initial mesh is "too coarse".

If a refinement occupies a spatial interval I, it can be deleted when it has no children, and the local error estimate of its parent in interval I is below the tolerance.

3. Advance the solution at interior points of the finest refinement 3 from $t = nk_1$ to the next level 3 time level $t = nk_1 + k_3$, using (2.13) with $l = 3$.

4. At the interfaces between refinements 2 and 3 use a hybrid method, the *coarse/fine approximation* corresponding to (2.13):

$$v_j^{l-1}(t+k_l) = (I - ck_l D_0^{l-1} + \tfrac{1}{2}c^2 k_l^2 D_+^{l-1} D_-^{l-1}) v_j^{l-1}(t), \tag{2.16}$$

with $l = 3$, where the spatial operators act on the $l-1$-st refinement. We are using the Lax-Wendroff formula with space step h_{l-1} and time step k_l. This amounts to using the Lax-Wendroff method on mesh $l-1$ but replacing λ_{l-1} by k_l / h_{l-1}. This method was used by Ciment [1971]. Figure 2.3 shows the stencil in the case when $L = M = 3$ so that the third level space step $h_3 = h_2/3$ and the third level time step $k_3 = k_2/3$. Points A, B, and C are used to advance to point

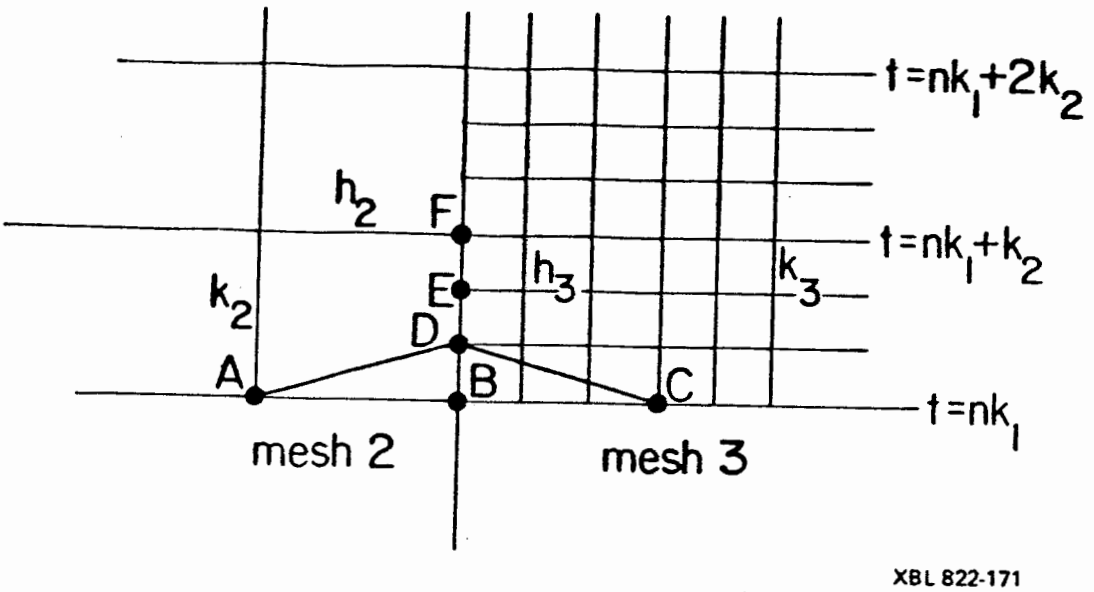


Figure 2.3 Coarse/Fine Interface

D.

For a difference scheme (2.10) whose stencil is more than three mesh points wide, we will need to use the coarse/fine approximation r times in Figure 2.3 (and correspondingly q times at the right end of a refinement). This is done, *e.g.*, for $r = 2$, by using the stencil illustrated in Figure 2.3 to get point D, then shifting the stencil one finer mesh point to the right to get the point to the right of D. (This involves a spatial interpolation in the coarser mesh, which is done as in Step 2).

5. Repeat steps 3 and 4 until the next time level in refinement 2 is reached. (In Figure 2.3, this would be $M = 3$ times.) In formula (2.16), the quantity k_l, $l = 3$, must be replaced successively by $2k_l$ (A, B, and C in Figure 2.3 are used to obtain the value at E), $3k_l$ (A, B, and C produce the value at F) $, \ldots, Mk_l = k_{l-1}$.

6. At level $t = nk_1 + Mk_3 = nk_1 + k_2$, certain points (x, t) are covered by both a second and third level mesh point. We already observed in our second description of the mesh structure that we allow this; it is done for simplicity. For these points, copy the solution values from refinement 3 to the appropriate positions in refinement 2.

7. For all points which are in the interior of refinement 2, but not in refinement 3, advance the solution one time step k_2 from $t = nk_1$ to $t = nk_1 + k_2$ using (2.13) with $l = 2$. (We are proceeding "outward" by starting to advance on coarser refinements.)

8. Now advance the solution one k_2 time step at the interface(s) between refinement 1 and refinement 2 using the coarse/fine approximation (2.16) with $l = 2$. This takes us from $t = nk_1$ to $t = nk_1 + k_2$.

9. We now have all solution values on refinements 2 and 3 for $t = nk_1 + k_2$. If the partition P_2 of $[0, T]$ contained the time level $t = nk_1 + k_2$, it is time to

check refinements 2 and 3 for possible adjustment. We repeat steps 1 and 2, but only for refinements with level greater than or equal 2. (Usually partition P_2 contains the same time division points as P_1, so this step is omitted.)

10. Repeat steps 3 and 4, advancing successively M (= 3 in Figure 2.3) k_3 time levels in the interior of refinement 3, from $t = nk_1 + k_2$ to $t = nk_1 + k_2 + k_3$, then to $nk_1 + k_2 + 2k_3, \ldots,$ finally to $t = nk_1 + k_2 + Mk_3 = nk_1 + 2k_2$. Next we repeat step 6 at level $t = nk_1 + 2k_2$ by copying solution values from refinement 3 to refinement 2 here. Then we repeat step 7 to advance at points which are in the interior of refinement 2 but not in refinement 3 from $t = nk_1 + k_2$ to $t = nk_1 + 2k_2$. Finally, we modify step 8 on the interface between refinements 1 and 2 to advance one step from $t = nk_1$ to $nk_1 + 2k_2$. This uses formula (2.16) with $l = 2$, but with k_2 replaced by $2k_2$.

11. We now have all solution values on refinements 2 and 3 at time $t = nk_1 + 2k_2$. If it is time to adjust the spatial mesh (*i.e.*, if partition P_2 contains this time level), repeat step 9.

12. Apply steps 10 and 11 $M-2$ more times. At the end of the first application of step 10, we will have reached $t = nk_1 + 3k_2$ (from $t = nk_1 + 2k_2$). We then successively reach $t = nk_1 + 4k_2, \ldots, nk_1 + Mk_2 = (n+1)k_1$.

13. At level $t = (n+1)k_1$ certain points (x, t) are covered by both a point of refinement 2 and a point of refinement 1. Copy the solution values at such points from refinement 2 to refinement 1.

14. For points which are in the interior of refinement 1, but are not in refinements 2 or 3, advance the solution one coarse (k_1) time step from $t = nk_1$ to $(n+1)k_1$.

15. Finally, if refinement 2 does not abut the left boundary, advance the solution at the left boundary using (2.14) with $l = 1$. If refinement 2 does not abut the right boundary, advance the solution at the right boundary using (2.15)

with $l = 1$. If some refinement (with level greater than one) abuts a boundary, we treat this at the same time an interface is treated in the above steps, but instead of the coarse/fine approximation, we use (2.14) or (2.15) as appropriate, with the appropriate level l.

An extremely important feature of this method is the use of a buffer on either end of any refinement (except the coarsest one), as mentioned in Step 1. If we are estimating the truncation error for the refinement $\{x_j^l\}$ and the error tolerance is exceeded between $j = \alpha$ and $j = \omega$, then we instead refine from $j = \alpha - b_l$ to $j = \omega + b_l$, where b_l is the buffer length for refinements of level $l+1$. That is, both ends of the $l+1$-st level refinement are padded with b_l extra cells of width h_l. In general, if our l-th level refinement requires several intervals of $l+1$-st level refinement (according to the error estimate), then each such interval is padded as above. (This may cause some $l+1$-st level refinements to merge.)

How do we choose b_l? From Figure 2.3 on the use of the coarse/fine approximation, it is clear that b_l should be at least one. For safety we make it two. Another consideration is, How often do we check the local truncation error (how fine is the partition P_{l-1}) and what is the largest wave speed? (As we said, we are assuming the largest wave propagation speed is known. In our model problem it is c.) For simplicity we shall assume that all partitions P_l, $l \geq 1$, are the same, and that we check the error every ϑ coarse time steps. Therefore, in time k_1 a wave could travel left or right a distance of $c\vartheta k_1 = c\vartheta\lambda_1 h_1 = c\vartheta\lambda_1 N^{l-1} h_l$, or $c\vartheta\lambda_1 N^{l-1}$ cells of width h_l. (Here the $l-1$ is an exponent, not a superscript.) So we take $b_l = 2 + \lceil c\vartheta\lambda_1 N^{l-1} \rceil$, where $\lceil x \rceil$ is the ceiling function (the least integer greater than or equal to x). (For difference approximation (2.10), b_l must be modified by replacing 2 by $q+1$ at the left end of a refinement, and by $r+1$ at the right end.) Obviously, higher level refinements have larger buffers.

The buffer mechanism has several beneficial consequences. First, and most important, it insures that the rapidly varying part of the solution does not escape into the coarser region. As we saw in Chapter 1, this is absolutely essential to the success of the algorithm. Secondly, this policy allows us to use difference approximations at coarse/fine interfaces which would otherwise not be accurate enough in the fine mesh. We can also estimate the local truncation error at coarse/fine interfaces in a very simple manner (see Section 5.3). Third, it allows "smooth" transitions in mesh width, as mentioned in Section 2.3. That is, a level l refinement can abut a level $l+1$ or $l-1$ refinement, but not others. This is important when using recursive refinements. Fourth, it keeps the refinements from splitting into tiny pieces, because level $l+1$ level refinements which are closer than $2b_l$ level l cells apart (before buffering) are joined together. (If the local truncation error were large in absolute value but suddenly changed sign, this might cause splitting into pieces.) We make this condition even more stringent by joining together any level $l+1$ refinements which are less than $2b_l+2$ level l cells (of length h_l) apart (before buffering). Fifth, buffering allows us to specify *a priori* the times to check the local error (and adjust the mesh). In particular, we need not check the error at *every* time step (Chapter 8 shows that this is very expensive). We can instead check at every *coarse* time step, or even every ϑ coarse time steps, where ϑ is a small positive integer. Sixth, buffering contributes greatly to the robustness of the algorithm. Buffers make the algorithm relatively insensitive to small inaccuracies in the local error estimation.

Let us comment on the storage required for this algorithm. If we use a two-level method and perform the operations in the order given, then we need two levels of solution values, just as for a uniform mesh. As soon as all the solution values in a refinement at a new time level are known, we can overwrite them on the old solution values. (This would not have been quite the case if we had

advanced the coarsest mesh first, because of the use of the coarse/fine approximation. But even in this case, only a slight amount of additional storage would be required.) Thus the storage requirement for solution values is no greater than for a uniform mesh with a similar number of solution values. A slight amount of additional storage is needed for pointers and indices; this is minuscule, compared to space for solution values. Next, free space is needed to separate the solution values on refinements. The amount is variable, but can be chosen quite small. (This will result in more memory repacking; see Chapter 6.) Finally, storage is needed for error estimates; but these can be done a refinement at a time, so we only need two vectors (when solving a scalar equation), each the size of the largest refinement. We should note that we did not implement our algorithm in a way which minimizes the amount of storage.

So far we have described the algorithm for a single equation. What are the modifications necessary for an $n \times n$ (coupled) system in one space variable?

For many problems that occur in applications, sharp gradients of different components of the vector u tend to occur in approximately the same place, and travel together. For such problems, a simple modification of the above scheme will suffice. A refinement, instead of consisting of a scalar set of solution values evaluated at l-th level mesh points $\{v_j^l(t_m^l)\}$ is now a *vector* set of solution values evaluated at these mesh points. So we simply store n times as many solution values. Importantly, the refinements are the same for each component, so the manipulation of refinements (creating, destroying, merging, separating, moving) is unchanged. Furthermore, evaluating the difference equations for any mesh point at spatial position x poses no difficulty, since all components of the approximate solution will also be available at x. To decide where to refine, we estimate the error at a position x for each component, and then compare the maximum (absolute value) of these estimates to our tolerance.

When the components of the solution u have steep gradients at different positions, we can use our algorithm, but it may refine regions which are not necessary for some components, and this may affect the efficiency of the algorithm. To ameliorate this, two modifications would be needed in the algorithm. The most important is that at a position x where v needs to be evaluated, not all components of v will be available, since x may be in a refinement for one component, but not in a refinement for another component. One must then interpolate (in space) to find v at x for the missing components. *By assumption, this is* justified, because the missing components do not have large gradients at x.

The other modification is the need to account for n sets of refinements. The amount of extra storage required (beyond space for solution values) would be very small. But considerable additional complexity would be introduced into the mesh manipulations. So we did not implement this extension.

CHAPTER 3

Stability

In this chapter we will examine the stability of our scheme. We will need the stability of our scheme to justify computing with it, and also to use in convergence results (stability plus consistency implies convergence).

The usual stability definition is Definition 3.3 of Gustafsson, Kreiss and Sundström [1972] (hereafter referred to as the GKS definition). We show that this definition does not lend itself to proving convergence on our refined mesh system, and following Berger, Gropp, and Oliger [to appear] propose a new stability definition which does lend itself to proving convergence. We then show that under mild assumptions, a method which is stable for a mesh consisting of one strip (in the sense of Section 2.2), is also stable for any number of strips. We then state a stability proposition for our mesh refinement scheme.

3.1. Preliminaries

This section introduces some notation and definitions in what follows. We will assume the coefficients A, B are constant. We will also use the notation of Section 2.2 instead of Section 2.3 for mesh points and difference approximations.

We assume that there is an upper bound K on the ratio of spatial step sizes: Let $h^i = \max_j h_j^i$, and assume $h^i / \min_j h_j^i \leq K$. (This is automatically ensured by our scheme because we select a maximum refinement level in advance. In fact, $K = N^{\Lambda-1}$.) This will ensure that all h_i have the same asymptotic order as $h \to 0$. Let $h = h_1 = \max_i h^i$, and $k = k_1 = \max_{i,j} k_j^i$. We denote an approximation to $u(x_j^i, t_{j,m}^i)$ by $v_j^i(t_{j,m}^i)$.

We can specify a uniform mesh by specializing our notations. Specifically, there is only one horizontal strip, $S_1 = [a, b] \times [0, T]$, all h_j^i are equal, and denoted by h, all $m_j^i = 1$, and all k_j^i are equal, and denoted by k. The mesh point $(x_j^i, t_{j,m}^i)$ is abbreviated to (x_j, t_m). The approximation $v_j^i(t_{j,m}^i)$ is abbreviated to $v_j(t_m)$.

We will now rewrite our difference approximations in the new notation. Our left boundary approximation (2.12), with $t = t_{\mu,m}^i = t_{r,m}^i$, $k = k_\mu^i = k_r^i$, $i = 1, 2, \ldots, s$, $m = 0, 1, \ldots, m_j^i$, is

$$v_\mu^i(t+k) = \sum_{\sigma=-1}^{0} S_\sigma^{(\mu)} v_r^i(t-\sigma k) + g_\mu^i(t), \qquad \mu = 0,1, \ldots, r-1, \tag{3.1}$$

where

$$S_\sigma^{(\mu)} = \sum_{j=-\tilde{r}}^{\tilde{q}} C_{j\sigma}^{(\mu)}(h_{r+j}^i, k_{r+j}^i) E^j, \qquad \sigma = 0,-1.$$

(The shift operator E tacitly depends on i and r also. Furthermore, $S_\sigma^{(\mu)}$ depends on i as well.) We assume $C_{-r+\mu,-1}^{(\mu)} = 0$, $\mu = 0,1, \ldots, r-1$. Note that all the coefficients are the same as before (except we have assumed they are constant); only the numbering of the solution values has changed.

Similarly, the right boundary approximation, with i, m as before, $t = t_{N_i-\mu,m}^i$, $k = k_{N_i-\mu}^i$, is

$$v_{N_i-\mu}^i(t+k) = \sum_{\sigma=-1}^{0} S_\sigma^{(-\mu-1)} v_{N_i-q}^i(t-\sigma k) + g_{-\mu-1}^i(t), \qquad \mu = 0,1, \ldots, q-1, \tag{3.2}$$

where

$$S_\sigma^{(-\mu-1)} = \sum_{j=-\hat{r}}^{\hat{q}} C_{j,\sigma}^{(-\mu-1)}(h_{N_i-q+j}^i, k_{N_i-q+j}^i) E^j, \qquad \sigma = 0, -1.$$

Here q and r are nonnegative integers and $\hat{q} \le q$. If we had introduced "fictitious" boundary points as Gustafsson, Kreiss, and Sundström did, we could have written our boundary conditions in their form, or in the form of Gustafsson

[1981]. We did not do so because it simplifies our analysis in the next chapter. It is important, both theoretically and practically, to confine the dependence between strips S_i and S_{i+1} to the points with $t = t^i$.

At the r leftmost fine points at the left end of a refinement (which does not abut the left boundary), and at the q rightmost fine mesh points at the right end of a refinement, we use the coarse/fine approximation, as described in Section 2.5. At all other points our interior approximation (2.10), (with $t = t^i_{\nu,m}$), becomes

$$v^i_\nu(t+k^i_\nu) = Q_0 v^i_\nu(t) + k^i_\nu F^i_\nu(t)\ , \tag{3.3}$$

$$Q_0 = Q_0(i,\nu) = \sum_{j=-r}^{q} A_j(h^i_{\nu+j},\, k^i_{\nu+j})E^j.$$

As initial conditions we will prescribe

$$v^0_j(0) = f_j,\ j = 0,\ 1,\ \ldots,\ N_0, \tag{3.4}$$

where the values f_j are arbitrary.

We next need to define discrete l_2 norms. This cannot be done exactly as for a uniform mesh. The discrete $l_2(x)$ norm can only be defined at coarse grid points (more generally, if we did not use an underlying coarse mesh, only at the time division points t^i of the strips S_i). For, as is evident in Figure 2.3, if a point in a level l refinement has coordinates $(x,\, t)$, and is not a coarse mesh point, there may exist no other grid points in other refinements with the same coordinate t.

A similar difficulty occurs with the $l_2(t)$ norm. This can be defined for all strips $0 \le t \le T$ only on coarse grid points. If we did not use an underlying coarse grid, the $l_2(t)$ norm could be defined for $0 \le t \le T$ only at the boundaries. However, the $l_2(x,\, t)$ norm can be defined in a natural manner, by adding $l_2(x,\, t)$ norms on each strip.

Definition 3.1. For $i = 1, 2, \ldots, s$ the discrete $l_2(x)$ inner product of two vector grid functions v and w defined on our grid at a time division point $t = t^i$ in strip S_i is

$$(v^i(t^i), w^i(t^i))_x = \sum_{j=0}^{N_i} h_j^i v_j^i(t^i)^* w_j^i(t^i) , \tag{3.5}$$

where we have defined $h_0^i = h_1^i$. The discrete $l_2(x)$ norm is given by

$$||v^i(t^i)||_x^2 = (v^i(t^i), v^i(t^i))_x .$$

(There are two spatial meshes at the time division points $t = t^i$; the definition above was for the mesh obtained *before* adjustment. The norm for the mesh obtained *after* adjustment is $||v^{i+1}(t^i)||_x$.) As usual, * denotes the conjugate transpose of a vector.

For certain purposes we will need an alternative definition of the $l_2(x)$ norm. If $u(x_j^i,t)$ is an approximation to $v_j^i(t)$, the above definition is the rectangle rule $O(h)$ approximation to $\int_a^b |u(x, t^i)|^2 dx$. We could also have approximated this integral by the trapezoid rule, which provides an $O(h^2)$ approximation.

Definition 3.1a. In the formula (3.5) for the $l_2(x)$ inner product of two gridfunctions, replace h_j^i by $(h_j^i)'$, where

$$(h_j^i)' = \begin{cases} \frac{1}{2}h_j^i & \text{for } j = 0 \text{ or } j = N_i, \\ \frac{1}{2}(h_j^i + h_{j+1}^i) & \text{at coarse/fine interfaces } x_j^i, \\ h_j^i & \text{otherwise.} \end{cases}$$

The norm corresponding to this inner product, called the *trapezoidal* $l_2(x)$ *norm*, will be denoted by $|||\cdot|||_x$.

It is well-known that, for uniform grids, both norms are equivalent, that is, there exist constants c_1 and c_2, such that

$$c_1 \| \cdot \|_x \le |||\cdot|||_x \le c_2 \| \cdot \|_x .$$

Definition 3.2. The discrete $l_2(t)$ inner product of two vector functions v and w on the strip $S_i = [a, b] \times [t^{i-1}, t^i]$ is given by

$$(v_j^i(\cdot), w_j^i(\cdot))_{t,i} = \sum_{m=0}^{m_j^i-1} k_j^i v_j^i(t_{j,m}^i)^* w_j^i(t_{j,m}^i).$$

The discrete $l_2(t)$ norm on the i-th strip is given by

$$\|v_j^i(\cdot)\|_{t,i}^2 \equiv \|v_j(\cdot)\|_{[t^{i-1},t^i]}^2 = (v_j^i(\cdot), v_j^i(\cdot))_{t,i} .$$

Definition 3.3. The discrete $l_2(x,t)$ inner product of two vector functions v and w on the strip S_i is

$$(v(\cdot), w(\cdot))_{x,t,i} = \sum_{j=0}^{N_i} \sum_{m=0}^{m_j^i-1} h_j^i k_j^i v_j^i(t_{j,m}^i)^* w_j^i(t_{j,m}^i) .$$

The discrete $l_2(x,t)$ norm on the i-th strip is

$$\|v(\cdot)\|_{x,[t^{i-1},t^i]}^2 = (v, v)_{x,t,i} .$$

The discrete $l_2(x,t)$ inner product for the entire region $R = [a, b] \times [0, T]$ is then given by

$$(v, w)_{x,[0,T]} = \sum_{i=1}^{s} (v, w)_{x,t,i},$$

and the discrete $l_2(x,t)$ norm for the region is

$$\|v\|_{x,[0,T]}^2 = \sum_{i=1}^{s} \|v\|_{x,[t^{i-1},t^i]}^2 .$$

For certain purposes we will need a grid defined for $-\infty < x < \infty$, so on the i-th strip S_i, we extend our grid uniformly to the left of $x = a$, with space and time steps h_1^i and k_1^i, respectively. On S_i we similarly extend the mesh to the right of $x = b$ using space steps $h_{N_i}^i$ and $k_{N_i}^i$. These extensions produce no new coarse/fine interfaces.

3.2. Need for a New Stability Definition

As we mentioned at the beginning of this chapter, to relate the local truncation error to the global truncation error, we need to use a variety of other results. One such result is Gustafsson's [1975], work on the convergence rate for approximations to the initial boundary value problem.

This work was based on the Gustafsson-Kreiss-Sundström (GKS) [1972] definition of stability. As usual, Gustafsson showed stability plus consistency implied convergence on a uniform mesh. In this section, we show why the GKS stability definition cannot be generalized for our refined grids, and present the alternative stability definition of Berger, Gropp and Oliger [to appear].

In order to give our generalization of the GKS definition, we will need to extend our integration to $t = \infty$. To do this, we can add additional strips S_i beyond $t = T$. We require that the "width" $t^{i+1} - t^i$ of these strips for $t^i \geq T$ be the same. Then the following is a direct generalization of the GKS definition 3.3 for the right quarter plane $[0, \infty) \times [0, \infty)$, rather than our vertical strip $[a, b] \times [0, \infty)$. (We set $a = 0$, remove the right boundary condition, use the extended mesh for $0 \leq x < \infty$, and define the $l_2(x,t)$ norm on the extended mesh in this definition only.)

Definition 3.4. Assume that the initial data f_j (3.4) in the difference approximation are zero. Let $\lambda = k_l / h_l$ = constant, independent of l. The approximation is stable if there are constants $K_0 > 0$, $\alpha_0 \geq 0$ such that, for all $\alpha > \alpha_0$, for all k, all mesh spacings of the type described, all left boundary functions g_μ, and all inhomogeneous terms F,

$$\left(\frac{\alpha-\alpha_0}{\alpha k+1}\right) \sum_{i=1}^{\infty} \left[\sum_{\mu=0}^{r-1} \|e^{-\alpha t} v_\mu\|^2_{[t^{i-1},t^i]} + \left(\frac{\alpha-\alpha_0}{\alpha k+1}\right) \|e^{-\alpha t} v\|^2_{x,[t^{i-1},t^i]}\right]$$

$$\leq K_0^2 \sum_{i=1}^{\infty} \left[\left(\frac{\alpha-\alpha_0}{\alpha k+1}\right) \sum_{\mu=0}^{r-1} \|e^{-\alpha(t+k)} g_\mu\|^2_{[t^{i-1},t^i]} + \|e^{-\alpha(t+k)} F\|^2_{x,[t^{i-1},t^i]}\right]$$

This definition seems plausible on the surface. We have merely applied the GKS definition on each horizontal strip, and added. (We have tacitly assumed that on each strip, the GKS definition applies even though our mesh is nonuniform. We will discuss this later.) But there are several problems with this approach, if we wish to use this definition to prove convergence.

The first is the assumption that the initial data f are zero. This may be acceptable for $t = 0$, but after we integrate over the strip S_1, we in effect start a new initial boundary value problem at $t = t^1$, and now the "initial" data is not zero. It is difficult to incorporate a nonzero f into the GKS definition, since it was derived using Laplace transform. However, it is necessary if we wish to use it to prove convergence. For, in an interval $0 \le t \le T$ the number of strips S_i becomes unbounded as $h \to 0$. The solution at $t^s = T$ depends on the values of the solution ("initial data") at all previous strips, but this dependence on values at times t^{s-1}, to $t^{s-2}, \ldots, t^1$, has to be removed if we want to prove convergence. This can only be done if f appears explicitly.

A second difficulty is related to the first. The GKS definition assures us that $\exp(-\alpha t)$ times the solution is in $l_2(x,t)$, but for any fixed t does not assure us that the solution is unconditionally in $l_2(x)$. (Compare Theorem 3.1, GKS). In order to integrate over a new strip S_i, we wish to treat the solution values at $t = t^{i-1}$ as "initial" values, and this requires that they be in $l_2(x)$.

A third (less important) difficulty is the Laplace transform parameter. (Recall that α is the real part of the Laplace transform parameter $s = \alpha + i\omega$, and the right half-plane in which the transform and its inverse converge absolutely is $\{s : \mathrm{Re}\, s > \alpha_0\}$. Call α_0 the "abscissa of convergence".) If we apply the GKS definition "stripwise" and add, we would need to be assured that the abscissas of convergence α_0 for each strip were uniformly bounded. (If we were dealing with a quarter plane problem with constant coefficients and no

undifferentiated terms, we could take $\alpha_0 = 0$. Otherwise, α_0 may not be explicitly computable.)

The above remarks should not be taken as a criticism of the GKS definition; only as pointing out that their definition is unsuitable for our purposes.

Motivated by these considerations, Berger, Gropp and Oliger [to appear] have given a new stability definition. It applies to approximations in any number of space dimensions, and is the discrete analog of the following well-posedness condition for differential equations.

Definition 3.5. Let Ω be a region in real Euclidean n-space R^n. Let R be the space-time region $\Omega \times [0, T]$. In R, consider the differential equation

$$Lu = F, \qquad x \in R,$$

where $L = \partial_t + P(x,t,\partial_x)$, together with initial condition

$$u(x, 0) = f(x), \qquad x \in \Omega,$$

and boundary conditions

$$Bu(x, t) = g(t), \qquad x \in \partial\Omega,$$

where $\partial\Omega$ denotes the boundary of Ω. Let $\|\cdot\|_\Omega$, $\|\cdot\|_{\partial\Omega\times[0,T]}$, and $\|\cdot\|_{\Omega\times[0,T]}$ denote the usual L_2 norms in space, time (evaluated at the boundary), and space-time, respectively. This problem is said to be *well-posed* if for any $T \geq 0$ there exists a constant $K_T^* > 0$ such that, for all f, g and F, the estimate

$$\|u(\cdot,T)\|_\Omega + \|u(x,\cdot)\|_{\partial\Omega\times[0,T]} \leq K_T^*\left(\|f\|_\Omega + \|g\|_{\partial\Omega\times[0,T]} + \|F\|_{\Omega\times[0,T]}\right)$$

holds.

The analog for the discrete approximation (3.1)-(3.4) with for a uniform mesh is then obvious.

Definition 3.6. Let $\lambda = k_1/h_1 =$ constant. The difference approximation (3.1)-(3.4) on a vertical strip $[a, b] \times [0, T]$ is *stable* for a uniform mesh if for any

$T > 0$ there exists a constant $K_T > 0$ such that, for all F, g_μ, and f, and for all $k_1 > 0$ such that $T = mk_1$, m integer, an estimate

$$||v(T)||_x + \sum_{\mu=0}^{r-1} ||v_\mu||_{[0,T]} + \sum_{\mu=0}^{q-1} ||v_{N_0-\mu}||_{[0,T]} \le K_T \left(||f||_x + \sum_{\mu=-q}^{r-1} ||g_\mu||_{[0,T]} + ||F||_{x,[0,T]} \right)$$

holds. (It is well-known that K_T can be replaced by $K_1 \exp(\alpha T)$ for some constant $K_1 > 0$ and some α.)

It is not too difficult to extend this definition to our refined mesh scheme so that it can be used to prove convergence. We proceed in two steps: (1) Extend this definition to our refined mesh on one strip; (2) extend it to several strips.

To extend the definition to one strip, it is only necessary to examine the coarse/fine interface between two refinements. The question is whether such an interface introduces any additional terms into the stability definition given above. The answer is no. The reasoning follows the work of Ciment [1971] and Oliger [1976]. To consider the stability near this interface, one extends the mesh on the left side to $-\infty$ (in space) and on the right side to ∞. Thus, one has two quarter-plane problems. The left quarter plane is folded along the coarse/fine interface, resulting in a right quarter plane problem for a coupled 2 × 2 system. The interface conditions become *homogeneous* (coupled) boundary conditions. Since only *inhomogeneous* boundary conditions enter the stability definition, the latter stays the same when a uniform mesh is replaced by one strip of our refinement scheme.

(This discussion has assumed that the time steps in the two quarter planes are equal. If they are not, then there is no analysis to support our discussion. However, computations by Oliger, Ciment, ourselves, and others seem to indicate the truth of the assertion even in this case.)

Next, we need to extend this definition to several strips. Before doing this, we need to introduce an additional complicating factor for our refined grids.

Recall that at times t^i, $i = 0, 1, \ldots, s-1$ we adjust the spatial mesh by interpolation. This produces an *interpolation error*, which we will need to account for in our analysis. (Note that this error arises even if we ignore the differential equation we are approximating.) We will examine its magnitude in the next chapter. For $i = 0, 1, \ldots, s-1$ we will define this error $I(t^i)$ by

$$\|v^{i+1}(t^i)\|_x = \|v^i(t^i)\|_x + I(t^i). \tag{3.6}$$

We will now extend the stability definition to two horizontal strips; the general case then follows by induction. For $i = 1,2, \ldots, s$ and any grid function w we first define the *boundary sum* of w at the left and right ends of the strip S_i as

$$\sum_{\mu=\partial} w^i_{\mu,m} \equiv \sum_{\mu=0}^{r-1} w^i_{\mu,m} + \sum_{\mu=0}^{q-1} w^i_{N_i-\mu,m}.$$

(A similar definition holds for the $l_2(t)$ norm of w on S_i.) Then for the strip S_1

$$\|v^1(t^1)\|_x + \sum_{\mu=\partial} \|v^1_\mu\|_{[0,t^1]} \le K_1 e^{\alpha t^1}\left(\|v^1(0)\|_x + \|F\|_{x,[0,t^1]} + \sum_{\mu=-q}^{r-1} \|g_\mu\|_{[0,t^1]}\right); \tag{3.7}$$

and similarly, on the strip $t^1 \le t \le t^2 = T$, using $v^2(t^1)$ as initial data,

$$\|v^2(t^2)\|_x + \sum_{\mu=\partial} \|v_\mu\|_{[t^1,t^2]} \le K_1 e^{\alpha(t^2-t^1)}\left(\|v^2(t^1)\|_x + \|F\|_{x,[t^1,t^2]} + \sum_{\mu=-q}^{r-1} \|g_\mu\|_{[t^1,t^2]}\right).$$

Adding these, using (3.6) for $i = 1$ and then subtracting $\|v^1(t^1)\|_x$ from both sides gives

$$\begin{aligned} &\|v^2(t^2)\|_x + \sum_{i=1}^{2}\sum_{\mu=\partial} \|v_\mu\|_{[t^{i-1},t^i]} \\ &\le K_1 e^{\alpha(t^2-t^1)}\left(I(t^1) + \|F\|_{x,[t^1,t^2]} + \sum_{\mu=-q}^{r-1} \|g_\mu\|_{[t^1,t^2]}\right) \\ &\quad + (K_1 e^{\alpha(t^2-t^1)} - 1)\|v^1(t^1)\|_x + K_1 e^{\alpha t^1}\left(\|v^1(0)\|_x + \|F\|_{x,[0,t^1]} + \sum_{\mu=-q}^{r-1} \|g_\mu\|_{[0,t^1]}\right). \end{aligned} \tag{3.8}$$

We wish to eliminate the dependence on $v^1(t^1)$, so we use inequality (3.7) to replace that term on the right of (3.8), use (3.6) again with $i = 0$, and use (3.4)

to obtain

$$\|v^2(t^2)\|_x + \sum_{i=1}^{2}\sum_{\mu=\partial}\|v_\mu\|_{[t^{i-1},t^i]} \le K_1 e^{\alpha(t^2-t^1)}\Bigg(\|F\|_{x,[t^1,t^2]} + \sum_{\mu=-q}^{r-1}\|g_\mu\|_{[t^1,t^2]}$$
$$+ I(t^1)\Bigg) + K_1^2 e^{\alpha t^1} e^{\alpha(t^2-t^1)}\Bigg(\|v^1(0)\|_x + \|F\|_{x,[0,t^1]} + \sum_{\mu=-q}^{r-1}\|g_\mu\|_{[0,t^1]}\Bigg)$$
$$\le K_T\Bigg(\|f\|_x + \|F\|_{x,[0,t^2]} + \sum_{i=1}^{2}\sum_{\mu=-q}^{r-1}\|g_\mu\|_{[t^{i-1},t^i]} + \sum_{i=1}^{2} I(t^{i-1})\Bigg),$$

where

$$K_T = \max(K_1^2 e^{\alpha t^2},\ K_1 e^{\alpha(t^2-t^1)}) .$$

It is now clear what the stability definition should be for any number s of strips:

Definition 3.7. Let $\lambda = k_l / h_l =$ constant, independent of l. The difference approximation (3.1)-(3.4) on a vertical strip $[a, b] \times [0, T]$ is ***stable*** for a refined mesh (as described in Section 3.1) if for any $T > 0$, there exists a constant $K_T > 0$ such that, for all positive integers s, all sets of time division points

$$0 = t^0 < t^1 < \cdots < t^{s-1} < t^s = T, \tag{3.9}$$

all $k_l > 0$, $l = 1, 2, \ldots, \Lambda$, satisfying our restrictions for refined meshes, and all F, g_μ, I, and f, an estimate

$$\|v^s(T)\|_x + \sum_{i=1}^{s}\sum_{\mu=\partial}\|v_\mu\|_{[t^{i-1},t^i]}$$
$$\le K_T\Bigg(\|f\|_x + \|F\|_{x,[0,T]} + \sum_{i=1}^{s}\sum_{\mu=-q}^{r-1}\|g_\mu\|_{[t^{i-1},t^i]} + \sum_{i=1}^{s} I(t^{i-1})\Bigg).$$

holds.

Our extension to several strips, then, will be complete if we show that K_T is uniformly bounded, independent of the number of strips s. For the general case of s strips (3.9), the corresponding K_T will be a maximum of s expressions of the form

$$K_1^i \exp(\alpha(t^s - t^{s-i})),$$

for $i = 1, 2, \ldots, s$. If all (positive) powers of K_1 are bounded by a uniform bound M_3, then this maximum will be bounded uniformly for any number of strips s in the interval $0 \le t \le T$ by $M_3 e^{\alpha T}$ if $\alpha \ge 0$, and M_3 if $\alpha < 0$.

The following assumption will ensure the uniform boundedness of the powers of K_1:

Assumption 3.1. There exists a $k_1^* > 0$ so that, for $0 \le k_1 \le k_1^*$ and $0 \le t \le T$,

(a) $K_1 = 1 + O(k_1)$, that is, there exists $M_1 > 0$ so that $K_1 \le 1 + M_1 k_1$;

(b) $sk_1 = \text{constant} = C_1$.

Assumption (a) is natural if one defines the solution operator $E(t_2, t_1)$, which takes a solution at time t_1 and produces a solution at time $t_2 > t_1$. When $t_2 = t_1$, E is the identity operator. Assumption (b) is only a very slight restriction. It says that as the largest time step k_1 becomes small, the number of strips s (in the same fixed time interval $0 \le t \le T$) becomes large. In practice, when we halve h_1 (and hence k_1), we can either keep the division points t^i the same, or use twice as many division points. To control the local truncation error, we do the latter, and this fulfills the assumption. (This assumption is an analog of our restriction on spatial step sizes in Section 3.1, but is a milder restriction.) If we check the local truncation error every ϑ coarse time steps (with ϑ fixed), this will automatically fulfill the assumption.

If assumptions (a) and (b) are satisfied, then the product of any number of factors K_1 is bounded. For,

$$K_1^s \le (1 + M_1 k_1)^s = (1 + M_1 k_1)^{C_1/k_1} \le e^{C_1 M_1}, \text{ for } 0 \le k \le k_1^*.$$

We have shown

Proposition 3.1. If the difference scheme (3.1)-(3.4) is stable in the sense of Definition 3.7 for one horizontal strip of a refined mesh, and if Assumption 3.1 holds, then it is also stable in the sense of Definition 3.7 for any number of strips.

We believe the GKS stability definition does not lend itself to a proposition of this kind.

3.3. Stability of Refinement Algorithm

In this section we shall outline results which we believe are true for our mesh refinement algorithm.

For a uniform mesh, a scheme is stable in the sense of Gustafsson, Kreiss, and Sundström if it is stable in the sense of Definition 3.6, either for a quarter-plane or strip problem. We believe that the converse is not true in general; that is, the new definition is stronger.

This means that each individual difference scheme must be proved stable *ab initio*. Certainly, however, a *dissipative* difference scheme such as we have been using will be stable under almost any (reasonable) definition, for a uniform mesh. In order to prove this for the new stability definition, one cannot use the normal mode analysis as in the GKS approach. Instead, the energy method is appropriate.

For a refined mesh, we showed in the last section that we need only consider one horizontal strip. Then a question which arises naturally is stability along a coarse/fine interface. This question already has been examined (using the GKS definition) in Ciment [1971] and Oliger [1976] for the case of equal time steps on both sides of the interface. Oliger found that if leap-frog was used on both sides of the interface, certain restrictions on the refinement ratio needed to be made. But if the difference scheme was dissipative on one side of the interface, all stability problems vanished. Since we are using refinements throughout the region, and possibly recursive ones, this suggests using a dissipative scheme throughout the region.

Still to be examined is the stability (in the sense of either GKS or Definition 3.6) along a coarse/fine interface for unequal time steps.

Even though our analysis is far from complete, we believe that our scheme is indeed stable in the sense of Definition 3.7.

CHAPTER 4

Error Analysis

It is clear from Chapter 2 that the success or failure of our algorithm will hinge on the reliability and efficiency of the local error estimation process, because this is what decides where to locate refinements. And in the next chapter we will see that estimating the local error in turn demands a knowledge of the behavior of the global error. Thus, this chapter will answer the following questions, which are of interest not only in their own right, but also for the success of the algorithm:

1. How does the order of accuracy of the interior, boundary and coarse/fine interface approximations, and the interpolation affect the global error?
2. Does mesh refinement increase the (global) order of accuracy of a difference approximation (compared to using a similar approximation on a uniform mesh)?
3. If not, can some theoretical arguments be given to justify mesh refinement?

Since our algorithm has two basic convergence-inducing parameters (the maximum step size h_1 and the local truncation error tolerance δ) instead of one, we first discuss different modes of convergence. Next we prove a theorem relating the pointwise interpolation error to its $l_2(x)$ norm.

The chief result giving the rate of convergence for difference approximations to the initial boundary value problem is due to Gustafsson [1975]. It bounds a weighted $l_2(x,t)$ norm of the global error in terms of the local errors. We restate (but do not prove) his result for a scheme which is stable with respect to the new stability definition introduced in the last chapter (Proposition 4.1). This proposition bounds the $l_2(x)$ norm of the global error in terms of

the local errors.

Based on this proposition, we prove another proposition, which assures us that the same rate of convergence obtains even when we economize on mesh points by placing fewer in regions where the solution is not changing rapidly. This result (Proposition 4.2) is based on the approach of de Boor [1973] which was applied by Pereyra and Sewell [1975] to solve boundary value problems for ordinary differential equations: at each time t^i choose a mesh which (approximately) equidistributes the local truncation error. This result provides the required theoretical justification for our method and also suggests where to place refinements.

In order to obtain a practical algorithm, still further compromises must be made. Although some numerical algorithms for boundary value problems in o.d.e's actually do use the equidistribution criterion more or less directly (Lentini-Pereyra [1977], White [1979]), this process is much too expensive to implement at every time step of a time-dependent calculation. Furthermore, for the reasons given in Section 2.2, we want to use piecewise uniform meshes. By doing so, and by using recursive refinements, we can achieve the effect of equidistribution.

A final compromise involves getting bounds for the local truncation error. de Boor [1975a, b] has given bounds for derivatives in terms of differences, and in principle we could use these for our local error estimation. However, we shall show in Chapter 5 that these bounds are hopelessly conservative. We are forced to *estimate*, rather than bound, the local truncation error, and even then, we estimate only leading terms of the asymptotic expansion of the local error.

Having made all these compromises, we then implemented four methods for estimating the local truncation error. These are explained in Chapter 5. We apply three of these methods to our model problems in Chapter 8.

4.1. Modes of Convergence

Let us first discuss what we mean by convergence. In all cases we let $\lambda = k_l / h_l =$ constant, independent of l. Throughout this chapter we shall assume the exact solution is sufficiently smooth. We shall also make Assumption 3.1 of the last chapter. There are several possibilities.

(a) We could hold our refined mesh and the local error tolerance δ (Section 2.6) fixed, and let the maximum possible refinement level Λ increase without bound. This will not produce convergence, since (for smooth solutions) increasing Λ beyond a certain point will introduce no further actual refinements (as we will see in Section 8.3). We recommend computing with a large enough level of Λ (say 10) so that the algorithm can refine as much as it pleases. In the rest of this chapter we assume this has been done.

(b) Keep the local error tolerance δ and the maximum refinement level Λ fixed, and let the largest spatial step h_1 approach zero. If we take a sufficiently large value for Λ as above, then the algorithm will refine as much as it needs to. Furthermore, (for smooth solutions) our method then has a property which leads to simplified analysis: *For sufficiently small h_1, our refined mesh becomes a uniform mesh.* This is because of our local error tolerance. If it is held fixed and h_1 approaches zero, then so do all h_l. Hence, our local error estimates (to be discussed in the next chapter) will ultimately become less than the tolerance δ at every mesh point on every refinement level. Thus, no refinements will ultimately be introduced in the first level (coarse) mesh. This type of convergence is not desirable, since the advantages of refinement are ultimately lost.

(c) One might object to the above procedure, on the grounds that the local error tolerance δ should not be held constant. After all, to study convergence in o.d.e. initial value solvers, one gives a decreasing sequence of local error tolerances, and (until the round-off level of the machine is reached) this produces a

decreasing sequence of maximum mesh sizes. But our algorithm is different. Decreasing the tolerance does not directly decrease the largest mesh spacing h_1. However, for any refinement level l and any fixed time, we can choose a sufficiently small tolerance δ so that the entire spatial region will be covered at that time by one l-th level refinement. Thus, the effect is as though the the largest spatial mesh size has been reduced from h_1 to h_l at that time. However, the same δ may not work for all times; this makes the method difficult to analyze. In addition, this method does not satisfy Assumption 3.1, since the number of strips s remains constant (instead of increasing) as δ is decreased, if we adjust the mesh every ϑ coarse time steps. To overcome these problems we need to let h_1 depend on δ.

(d) A fourth method would let $h_1 \to 0$ and choose δ as a function of h_1, so that $\delta \to 0$ also. (Alternatively, we could let $\delta \to 0$ and choose h_1 as a function of δ.) If one knows the order of the global error, one could choose $\delta = C(h_1)^p$ for some constant C. This is certainly the theoretically most appealing method, and we shall use it in our analysis of this chapter, and in our numerical experiments in Section 8.5. If we use this method, then the grid does *not* approach a uniform coarse grid as $h_1 \to 0$, as in method (b). Rather, the ratio of the width of any refinement to the width of its parent should approach a constant as $h_1 \to 0$. However, for checking the asymptotic behavior of the program in Section 8.5 we shall also use method (b), since it does not beg the question by assuming the behavior of the global error.

4.2. Interpolation Error

As we mentioned in the last chapter, in addition to the usual truncation errors, for a refined mesh we have another source of error—the interpolation error $I(t^i)$ at the time division points t^i, $i = 0, 1, \ldots, s-1$. How does the trapezoidal norm $|||v^i(t^i)|||_x$ change when we readjust the mesh to produce

$|||v^{i+1}(t^i)|||_x$?

Theorem 4.1. Let our refined mesh be as in Section 2.2, with horizontal strips S_i, $i = 1, 2, \ldots, s$. At time division points t^i, $i = 0, 1, \ldots, s-1$, obtain new approximate solution values $v^{i+1}(t^i)$ from the old ones $v^i(t^i)$ by (a) linear or (b) quadratic interpolation in space. Then for the method (d) of convergence,

$$|||v^{i+1}(t^i)|||_x^2 = |||v^i(t^i)|||_x^2 + O(h^2) \qquad \text{as } h \to 0,$$

regardless of the (two-level) difference scheme used. Here h is the maximum mesh size, and $\lambda = h_l / k_l$ = constant independent of l.

Proof. It is sufficient to assume we are dealing with one level l spatial interval at time $t = t^i$ for case (a), or two for case (b), in (each of) which are interpolated $N-1$ (resp. $2N-2$) new level $l+1$ approximate solution values. (N is the spatial refinement ratio.) For, as described in Chapter 2, we are allowed to insert only one level of refinement at each t^i, except at $t = 0$. (At $t = 0$ we can use the initial condition directly, producing no interpolation error.) We are allowed to delete more than one level at a time, but in that case we can prove our theorem recursively a level at a time. Without loss of generality, we shall assume that new points are introduced but not removed.

Let us now examine linear interpolation. We will use a simplified notation. On one level l interval $[x_0, x_N]$ the contribution to the trapezoidal sum $|||v|||_x^2$ is

$$\Sigma_1 = \tfrac{1}{2}h_l[v_0^2 + v_N^2] = \tfrac{1}{2}Nh_{l+1}[v_0^2 + v_N^2].$$

(The 2's are now exponents, not superscripts.) For $N \geq 2$, we use linear interpolation to obtain approximate solution values $v_1, v_2, \ldots, v_{N-1}$, and we form the new contribution

$$\Sigma_2 = h_{l+1}[\tfrac{1}{2}v_0^2 + \sum_{j=1}^{N-1} v_j^2 + \tfrac{1}{2}v_N^2]$$

to the trapezoidal rule sum. We wish to compare this to the previous sum Σ_1. We

use the formula for linear interpolation

$$v_j = v_0 + \frac{j}{N}(v_N - v_0), \quad j = 1, 2, \ldots, N-1,$$

and substitute in Σ_2. Let u be the exact solution. Since the method is convergent, we can assume that the global error $e = u - v$ is $O(h^\beta)$ as $h \to 0$, for $\beta \geq 1$. Therefore,

$$(v_N - v_0)^2 = (v_N - u_N + u_N - u_0 + u_0 - v_0)^2 = 0(h^2).$$

After some elementary manipulations we obtain

$$h_{l+1}[\tfrac{1}{2}(v_0^2 + v_N^2) + (N-1)v_0 v_N + O(h^2)].$$

Since $v_0 v_N = \tfrac{1}{2}(v_0^2 + v_N^2) + 0(h^2)$, the last sum becomes $\Sigma_1 + O(h^3)$.

Now let ν be the number of level l intervals in which we interpolated. Then the $O(h^3)$ term is multiplied by ν. If we use method (d) of convergence, then ultimately ν is a fixed fraction of the region $a \leq x \leq b$ (*i.e.*, ν / N_0 = constant as $h \to 0$). So $\nu h = (\nu / N_0)(b - a)$ and one power of h is lost. This proves the first assertion. (If we had instead used method (b) of convergence, then $\nu \to 0$ as $h \to 0$, and one power of h is not lost.)

For quadratic interpolation on two intervals $[x_0, x_N]$, $[x_N, x_{2N}]$ of the level l mesh, the contribution to the trapezoid rule sum is

$$\Sigma_3 = \tfrac{1}{2} N h_{l+1}[v_0^2 + 2v_N^2 + v_{2N}^2].$$

We wish to compare it to the sum

$$\Sigma_4 = h_{l+1}[\tfrac{1}{2}v_0^2 + \sum_1^{2N-1} v_j^2 + \tfrac{1}{2}v_{2N}^2],$$

where $v_1, v_2, \ldots, v_{N-1}, v_{N+1}, \ldots, v_{2N-1}$ result from quadratic interpolation

$$v_j = v_0 + \frac{j}{N}(v_N - v_0) + \frac{j(j-N)}{2N^2}(v_0 - 2v_N + v_{2N}).$$

As before, $v_N - v_0 = O(h^2)$, and $v_0 - 2v_N + v_{2N} = O(h^2)$, so our sum Σ_4 becomes

$$h_{l+1}[\tfrac{1}{2}v_0^2 + (2N-1)v_0^2 + \frac{2}{N}v_0(v_n - v_0)\sum_1^{2N-1} j\,] + O(h^3).$$

After simple manipulations, this becomes

$$h_{l+1}[\tfrac{1}{2}v_0^2 + (2N-1)v_N^2 + \tfrac{1}{2}v_{2N}^2] + O(h^3).$$

Using

$$\frac{(N-1)}{2}(v_0^2 - 2v_N^2 + v_{2N}^2) = 0(h^2),$$

we obtain $\Sigma_3 + O(h^3)$. By the same arguments as before, we lose one power of h.

If we use the rectangle rule for the $l_2(x)$ norm ($\|\cdot\|_x$), a similar proof shows that we obtain $O(h)$ instead of $O(h^2)$ for either type of interpolation. This latter norm is more suited for our analysis to follow.

4.3. Rate of Convergence, I

Before deriving our main convergence result (Proposition 4.2) we will state, but not prove, the analogue of Theorem 2.1 of Gustafsson [1975], on the rate of convergence of difference approximations to the initial boundary value problem for hyperbolic systems in one space dimension. Our analogue uses the stability definition given in Chapter 3 instead of the GKS definition. Thus the result is in terms of the $l_2(x)$ norm of the solution instead of the weighted $l_2(x,t)$ norm. Aside from this, the only differences in our proposition are the change from the quarter plane to strip, the inclusion of interpolation error, and the compatibility assumption (4.1), which is a weakening and generalization of a similar assumption of Gustafsson's.

As an application of this proposition, suppose that one uses an $O(h^2)$ interior approximation and $O(h)$ boundary approximations. Also suppose that we use linear interpolation ($O(h^2)$) to obtain solution values on $l+1$-st level

refinements from l-th level refinements, and an $O(h^2)$ approximation at coarse/fine interfaces. Finally, suppose that this scheme is stable in the sense of Definition 3.7. Then, subject to certain compatibility and smoothness assumptions, this proposition says that the global error is $O(h^2)$.

For simplicity, we shall eliminate the initial error at $t = 0$ by absorbing it into the interpolation error there, since we are using a one-step (two time level) method.

We now state the analog of Gustafsson's convergence result.

Proposition 4.1. Consider the differential equation (2.1)-(2.3) on the strip $[a, b] \times [0, T]$. Approximate it by the difference scheme (3.1)-(3.4) on a grid as described in Section 3.1. Suppose that Assumptions 3.1(a) and (b) hold, and that the approximation is stable with respect to Definition 3.7. Assume that the boundary conditions (3.1), (3.2) can be solved boundedly for the left-hand side.

We assume the local truncation error per unit time step is $O(h^p)$ in the interior and at interfaces, the local truncation error is $O(h^{p-1})$ for the initial function and at the boundary, and the (pointwise) interpolation error is $O(h^p)$, where $p \geq 1$. Thus, on the i-th strip, $i = 1,2, \ldots, s$, with the global truncation error $e = u - v$, and $t = t^i_{\nu,m}$,

$$e^i_\nu(t+k^i_\nu) = Q_0 e^i_\nu(t) + k^i_\nu (h^i_\nu)^p d_1(x^i_\nu, t)$$

in the interior ($\nu = r, r+1, \ldots, N_i - q$). We also assume the error at coarse/fine interfaces is $O(h^p)$; if we use the coarse/fine approximation mentioned in Section 2.6, this can be subsumed in d_1. At the boundaries

$$e^i_\nu(t+k^i_\nu) = \sum_{\sigma=-1}^{0} S_\sigma^{(\nu)} e^i_r(t-\sigma k^i_\nu) + (h^i_\nu)^p d_2(x^i_\nu, t), \quad \nu = 0,1, \ldots, r-1,$$

$$e^i_\nu(t+k^i_\nu) = \sum_{\sigma=-1}^{0} S_\sigma^{(\nu-N_i-1)} e_{N_i-q}(t-\sigma k^i_\nu) + (h^i_\nu)^p d_2(x^i_\nu, t), \qquad \nu = N_i - q + 1, \ldots, N_i.$$

For the interpolation error, let $u = v$ at the "old" mesh points (x_j^{i-1}, t^{i-1}), and let $v_I(x, t^{i-1})$ be the (continuous) function obtained from interpolation at these points. Given any "new" point x_ν^i, find its nearest surrounding "old" points $x_j^{i-1} \le x_\nu^i \le x_{j+1}^{i-1}$. Then we assume the Lagrange interpolation error

$$u(x_\nu^i, t^{i-1}) - v_I(x_\nu^i, t^{i-1}) = (h_\nu^i)^p d_4(\xi, t^{i-1}) = (h_\nu^i)^p d_3(x_\nu^i, t^{i-1}),$$

where $x_j^{i-1} \le \xi \le x_{j+1}^{i-1}$. Since ξ is a function of x_ν^i, we have rewritten d_4 in the alternate form d_3. We assume d_1, d_2, d_3 are uniformly bounded, and that $k_\nu^i / h_\nu^i = \lambda =$ constant, independent of i and ν.

On the extended mesh described in Section 3.1, we assume that the difference approximation is stable for the Cauchy problem, *i.e.*, for $F_\nu^i(t) = 0$, there exist constants $K_2 > 0$, $\alpha_1 \ge 0$ such that for $i = 1, 2, \ldots, s$

$$\sum_{\nu=-\infty}^{\infty} h_\nu^i |u_\nu^i(t^i)|^2 \le K_2 e^{2\alpha_1 t} \sum_{\nu=-\infty}^{\infty} h_\nu^1 |u_\nu^1(0)|^2.$$

Finally, for compatibility between interpolated values at $t = t^i$ and boundary approximations, we require for $\nu = 0, 1, \ldots, r-1$, $i = 1, 2, \ldots, s$,

$$|d_2(x_\nu^i, t^{i-1}) - (d_3(x_\nu^i, t^{i-1}+k_\nu^i) - \sum_{\sigma=-1}^{0} S_\sigma^{(\nu)} d_3(x_\nu^i, t^{i-1}-\sigma k_\nu^i))| = O(1), \tag{4.1}$$

(with a similar condition at the right boundary) where $d_3(x_\nu^i, t^{i-1}+k_\nu^i)$ is defined on the extended mesh (for the Cauchy problem) by

$$d_3(x_\nu^i, t^{i-1}+k_\nu^i) = Q_0 d_3(x_\nu^i, t^{i-1}) + k_\nu^i d_1(x_\nu^i, t^{i-1}).$$

(At times $t = t^i$ we extend the function d_3 smoothly to the left of $x = a$ so that $d_3 = 0$ for $x \le a - 1/h_1$. Similarly, we extend d_3 smoothly to the right of $x = b$ so that $d_3 = 0$ for $x \ge b + 1/h_1$.) Then for any $T > 0$, there exists a constant $K_T' > 0$ such that, for all positive integers s, all sets of time division points

$$0 = t^0 < t^1 < \cdots < t^{s-1} < t^s = T,$$

all $k_l > 0$, $l = 1, 2, \ldots, \Lambda$, satisfying our restrictions on refined meshes, and all

d_1, d_2 and d_3,

$$\|e(T)\|_x^2 + \sum_{i=1}^{s} \sum_{\mu=\partial} \|e_\mu(\cdot)\|^2_{[t^{i-1},t^i]} \tag{4.2}$$

$$\leq K_T' h_1^{2p} \left\{ \|d_1(\cdot,\cdot)\|^2_{x,[0,T]} + \sum_{i=1}^{s} \left\{ \|d_3(\cdot,t^{i-1})\|_x^2 + \sum_{\mu=\partial} \|d_2(x_\mu,\cdot)\|^2_{[t^{i-1},t^i]} \right\} \right\}.$$

Hence the convergence rate is $O(h^p)$ as $h \to 0$.

4.4. Rate of Convergence, II

In this section we shall derive the result of Section 4.1 under somewhat different assumptions. Proposition 4.1 told us the rate of convergence achieved when the local truncation errors were uniformly bounded. However, this result did not tell us how many mesh points are required, or where they should be distributed, in order to obtain a given bound on the local truncation error.

Proposition 4.2 asserts the same rate of convergence as in Proposition 4.1 when we economize on the number of mesh points. More specifically, we shall assume that at each time division point t^i the mesh is approximately equidistributed. We shall follow the approach of Oliger [1978] for the Cauchy problem, and use the results of Pereyra and Sewell [1975] on equidistribution of the local truncation error. Proposition 4.2 in a sense gives a theoretical justification for our algorithm.

For our local truncation errors d_1 and d_2 we shall assume

$$(h_j^i)^p d_\nu(x,t) = (h_j^i)^{\alpha_\nu} T_\nu(x,t) + (k_j^i)^{\beta_\nu} U_\nu(x,t) + O(h^{p+1}),$$

where $x = x_j^i$, $t = t_{j,m}^i$, $\nu = 1,2$, α_ν and β_ν are positive integers, and T_ν and U_ν are uniformly bounded. (If $U_2 \equiv 0$ then $\beta_2 = \infty$.) We then assume $p = \min(\alpha_1, \beta_1, \alpha_2, \beta_2)$. As before, we assume d_3 is also uniformly bounded.

We also assume that the time steps k_j^i are chosen small enough so that the spatial truncation error dominates the time error, that is,

$$|(k_j^i)^{\beta_\nu} U_\nu(x,t)| \le |(h_j^i)^{\alpha_\nu} T_\nu(x,t)|, \qquad \nu = 1,2. \tag{4.3}$$

We will now define the hybrid local truncation error for $x = x_j^i$, $t = t_{j,m}^i$ as

$$\hat{\tau}(x,t) = (h_j^i)^p \cdot \begin{cases} d_1(x,t), \ j = r, r+1, \ldots, N_i - q \\ d_2(x,t), \ j = 0, 1, \ldots, r-1, N_i - q + 1, N_i - q + 2, \ldots, N_i. \end{cases}$$

(It is a hybrid of the boundary local truncation error and the interior local truncation error per unit time step.) We define the interior error $d_1(x_j^i, t)$ as zero for $j = 0, 1, \ldots, r-1, N_i - q + 1, \ldots, N_i$, and similarly define other functions as zero outside their range of definition. We will then attempt to choose the mesh points x_j^i to minimize spatial errors, and assume that the time steps k_j^i are chosen so that temporal errors are no larger.

It has been suggested by de Boor [1973] that the (spatial) mesh be chosen in such a way that at the i-th time division point, $i = 1, \ldots, s$,

$$h_j^i |\hat{\tau}(x_j^i, t^{i-1})|^2 = \text{constant} = E_i, \quad j = 0,1,2, \cdots$$

and such a mesh is called *equidistributing*. As we noted in Section 1.3, this approach has been applied to boundary value problems for ordinary differential equations by Pereyra and Sewell [1975]. Since this expression depends on the mesh, Pereyra and Sewell introduced the idea of an approximately equidistributing mesh.

Let

$$\sup_{x,t} (|T_1(x,t)|, |T_2(x,t)|) \le M_1$$

and $\varepsilon = M_1 / K^{1/\sigma}$ where $\sigma = 2/(2p+1)$. (K relates the largest and smallest values of h_j^i.) Let

$$z(x,t) = \max(|T_1(x,t)|, |T_2(x,t)|, \varepsilon) \tag{4.4}$$

and $\gamma_j^i(t^{i-1}) = (h_j^i)^p z(x_j^i, t^{i-1})$. A mesh is said to be *approximately equidistributing* over $a \le x \le b$ at $t = t^{i-1}$ if

$$h_j^i(\gamma_j^i)^2(t^{i-1}) = E_i(1 + O(h_1))$$

for $a \le x_j^i \le b$. The ε has the effect of disallowing excessively large step sizes h_j^i in the Pereyra-Sewell development. Our use of a uniform coarse mesh precludes the (spatial) step size from exceeding h_1.

Pereyra and Sewell show that if a mesh is equidistributed with respect to

$$E_i = \int_a^b |z(x,t^{i-1})|^\sigma dx,$$

then such a mesh is also approximately equidistributing with

$$E_i = \left[N_{i-1}^{-1}\int_a^b |z(x,t^{i-1})|^\sigma dx\right]^{2p+1},$$

where $N_{i-1}+1$ is the number of mesh points at time t^{i-1} in the interval $a \le x \le b$.

Our theoretical strategy, then, would be to assume the spatial mesh is "approximately equidistributed" at time $t = t^{i-1}$, $i = 1,2,\ldots,s$, compute forward in time until this is "nearly violated" at time $t = t^i$, and then approximately equidistribute again. In practice, it is probably more work to discover when this is "nearly violated" than it is to simply approximately equidistribute again, so we usually choose our "equidistribution" times *a priori* as some integer multiple of the coarsest time step k_1. A somewhat similar strategy has been used by Gannon [1980] for parabolic problems with finite elements in two space dimensions.

Let us now make precise what we mean by "nearly violated". That is, the mesh shouldn't deviate too far from (approximate) equidistribution between

times t^i when we check the local truncation error and adjust the mesh. We shall assume that, for $i = 1,2,\ldots,s$, $m = 1,2,\ldots,m_j^i$, and $x = x_j^i$,

$$|d_\nu(x,t_{j,m}^i)|^2 \le (1 + C_2 k_j^i)|d_\nu(x,t_{j,m-1}^i)|^2 ; \tag{4.5}$$

here $\nu = 1,2$; $j = 0,1,\ldots,r-1,N_i-q+1,\ldots,N_i$ for $\nu = 2$; $j = r,r+1,r+2,\ldots,N_i-q$ for $\nu = 1$; and C_2 is a nonnegative constant independent of the mesh spacing and the d_ν.

Next, we define sets where the truncation error is excessive. For $i = 0,1,\ldots,s-1$ let $M_{L_i}(t^i) = \{x : |z(x,t^i)| \ge L_i,\ a \le x \le b\}$, where L_i is chosen so that

$$\int_{M_{L_i}(t^i)} |z(x,t^i)|^\sigma dx = \frac{1}{2}\int_a^b |z(x,t^i)|^\sigma dx .$$

We let $\mu(M_{L_i}(t^i))$ be the measure of the set $M_{L_i}(t^i)$ and $\mu_{\max} = \max_i \mu(M_{L_i}(t^i))$.

According to Pereyra and Sewell's Lemma 3.1, since the mesh is approximately equidistributed at time $t = t^{i-1}$, $i = 1,2,\ldots,s$,

$$||d_1(\cdot,t^{i-1})||_x^2 + \sum_{j=0} h_j^i |d_2(x_j^i,t^{i-1})|^2 \le C_3 \mu_{\max}^{2p} ||z(\cdot,t^{i-1})||_\Omega^2 + O(h_1),$$

where C_3 is a constant. Note that the norm for z is the *continuous* $L_2(x)$ norm on the interval $\Omega = [a, b]$.

We need to eliminate the h_j^i factor on the left. We can do this by modifying the Pereyra-Sewell argument, treating boundary and interior terms separately. We use our assumption that the smallest step size is a bounded fraction of the largest step size. We obtain

$$||d_1(\cdot,t^{i-1})||_x^2 + \sum_{j=0} |d_2(x_j^i,t^{i-1})|^2 \le C_3(1 + \frac{K(q+r)}{b-a})\mu_{\max}^{2p} ||z(\cdot,t^{i-1})||_\Omega^2 + O(h_1). \tag{4.6}$$

This gives us an estimate at each time division point t^{i-1} between adjacent horizontal strips. Then on the i-th strip S_i we obtain

$$\|d_1(\cdot,\cdot)\|^2_{x,[t^{i-1},t^i]} + \sum_{j=\partial} \|d_2(x^i_j,\cdot)\|^2_{[t^{i-1},t^i]} \tag{4.7}$$

$$= \sum_{j=r}^{N_i-q} h^i_j \sum_{m=0}^{m^i_j-1} k^i_j |d_1(x^i_j,t^i_{j,m})|^2 + \left[\sum_{j=1}^{r-1} + \sum_{j=N_i-q+1}^{N_i}\right] \sum_{m=0}^{m^i_j-1} k^i_j |d_2(x^i_j,t^i_{j,m})|^2$$

$$\le G(C_2,(t^i-t^{i-1}))\left\{\|d_1(\cdot,t^{i-1})\|^2_x + \sum_{j=\partial} |d_2(x^i_j,t^{i-1})|^2\right\},$$

where G is the Lipschitz function: $G(\alpha,t) = (exp(\alpha t) - 1)/\alpha$, $\alpha > 0$, and $G(\alpha,t) = t$ when $\alpha = 0$.

We now apply the equidistribution inequality (4.6) to (4.7) for each $i = 1,2,\ldots,s$ and add together the results. Then we assume the hypotheses of Proposition 4.1, in particular, that the number of strips is $O(1/k_1)$ as $k_1 \to 0$. We shall also assume that the interpolation error is $O(h^{p+1})$ rather than $O(h^p)$. Applying Proposition 4.1, we obtain the principal result of this chapter:

Proposition 4.2. Under the assumptions of Proposition 4.1, together with the assumptions that (a) the spatial truncation error dominates the time error; (b) the mesh is approximately equidistributed at times t^i, $i = 0,1,\ldots,s-1$; (c) assumption (4.5) on the growth of the error inside a strip S_i; and (d) the interpolation error is one order higher than the interior error; we obtain a convergence rate of order p, and the following estimate holds for the global truncation error e at $t = t^s \equiv T$:

$$\|e(T)\|^2_x \le K''_T h^{2p} \sum_{i=0}^{s-1} \left(\mu^{2p}_{\max}(1 + \frac{K(q+r)}{b-a})\|z(\cdot,t^i)\|^{2^i}_\Omega)\|^2_x\right) + O(h^{2p+1}),$$

where K''_T is a constant independent of the number of strips s, the mesh spacing, and the local truncation error functions d_1, d_2 and d_3; it may depend on T, however. Here z is defined in (4.4).

A comparison of the above result with Gustafsson's [1975] theorem for a uniform mesh shows that our algorithm does *not* provide any increase in the order p of convergence, and our results in Section 8.5 confirm this. Instead, our

algorithm introduces the factor

$$\mu_{\max}^{2p}(1 + \frac{K(q+r)}{(b-a)})$$

into the estimate. Loosely speaking, our method does not increase p but instead multiplies the coefficient of h^p in the global error by this factor. When the solution has rapid variations only in a small part of the (spatial) region, then the local truncation error is small over most of the region, and $\mu_{\max}$ is therefore small. Thus our algorithm can use fewer mesh points in regions where the local truncation is small (compared to using the same difference scheme on a uniform mesh which achieves the same level of accuracy) and this produces significant economies, as shown in Section 8.4.

Proposition 4.2 depends on Proposition 4.1, which we have not proved. We believe it can be proved at least when the interior approximation is $0(h^p)$, the initial and boundary approximations are $O(h^p)$, and the interpolation error is pointwise $0(h^{p+1})$ (hence, $O(h^p)$ in $l_2(x)$), using the energy method. Our results of Sections 8.5 and 8.9 show that the claimed rate of convergence (even with the less accurate initial and boundary approximations) is indeed achieved. This experimentally confirms Propositions 4.1 and 4.2.

Let us now comment on some of the assumptions used in deriving these results. Many of these, such as the assumption that the number of strips s is $0(1/k_1)$ as $k_1 \to 0$, are quite natural. The assumption (4.5) on the growth of the truncation error between adjustments of the spatial mesh is natural but not *a priori* verifiable. We try to enforce this assumption by allowing enough "buffer" at either end of a refinement (see Section 2.5). We believe that the assumption on the order of the interpolation error is unnecessary and can be relaxed.

The most tenuous assumption is (4.3) that the local truncation errors are such that the spatial error dominates the time error, for both interior and boun-

dary approximations. For our interior scheme, Lax Wendroff, it is well-known that the truncation error decreases (in general) as $\lambda = k_1/h_1$ increases to the upper stability limit. So this assumption is probably not satisfied in our model problem P1 (the first order wave equation) for $\lambda = 0.8$. Naturally, for fixed h_1 we could in theory choose a λ small enough so that the assumption was satisfied, but this might entail wastefully small time steps.

It is clear that the reason for this assumption is technical convenience. Without it, we would have to consider equidistributing in both space and time simultaneously. If we used uniform time steps throughout a horizontal strip S_i, we might be able to do this analysis by considering space-time rectangles with horizontal sides lying on the lines $t = t^{i-1}$ and $t = t^i$, the division points between strips. As we mentioned earlier, however, it is necessary to use different time steps in different parts of the spatial region.

Even when the assumption is not true, the qualitative result that follows from it still is. We are looking for sets M_L where the local truncation error is much larger than that of the surrounding region. Even if the assumption is not satisfied, it is likely that both time and spatial truncation errors are much larger in this set than outside it. At isolated points of M_L the spatial and time errors may cancel, but this is dealt with by the buffer mechanism.

We should emphasize that we do not approximately equidistribute in practice, as it would be too expensive. Our use of recursive refinements achieves a primitive form of approximate equidistribution at much less cost.

It may be easier to prove Proposition 4.1 if we do not write our boundary approximations as in (3.1)-(3.2), but instead use fictitious points outside the region. Then we can assume the boundary approximations are all on one (time) level and of extrapolation type. This form was used by Gustafsson [1981]. But this form would have complicated our equidistribution analysis because we

would have had to introduce constraints on the mesh points, so that two of them were at $x = a$ and $x = b$.

Having related the global to the local error, we next examine ways to estimate the local error.

CHAPTER 5

Estimation of the Local Truncation Error

In Chapters 2 and 4 we saw that we needed a way to obtain estimates for the local truncation error made in advancing the solution of the difference equation one time step. In this chapter we study several methods for obtaining these estimates in the interior of the region, at coarse/fine interfaces which do not abut boundaries, and at boundaries. In Chapter 8 we will use numerical experiments to illustrate the accuracy and efficiency of these methods.

We first examine methods for the placement of mesh points which do not rely on the local truncation error. Next we describe our first three methods on model problem P1 (the first order wave equation; see section 2.4), using Lax-Wendroff in the interior of the region. These methods are differences, two-step Richardson extrapolation, and bounding derivatives in terms of differences. (The last is not practical.) We then explain the three step Richardson method, and prove Theorem 5.1, which states that this procedure is justified under quite general conditions. The fundamental restrictions are that the difference approximation (interior or boundary) must have the same order of accuracy in both space and time, and be explicit. The proof also shows that this method is simultaneously a deferred correction method. Then we propose a simple scheme for estimating the error at coarse/fine interfaces which do not abut boundaries. Next we examine our methods at the boundaries. The three-step Richardson algorithm applies here only with modifications, but its range of applicability is more limited, and it is considerably less convenient to use than differences. Finally, we explain the modifications necessary for $n \times n$ systems of equations.

The first important conclusion of this Chapter (and of the numerical results in Chapter 8) is that several methods can be used to estimate the error, both in the interior and at boundaries. Thus our algorithm is quite general as well as robust. The second important finding is that for both interior and boundaries, one of the methods is more convenient and general than the others. In the former case this is three-step Richardson extrapolation, and in the latter it is differences.

For simplicity, we shall write all approximations as occurring on a uniform mesh, then consider modifications at coarse/fine interfaces and boundaries. We will always assume that $\lambda = k/h =$ constant, and that the solution of the differential equation has sufficiently many derivatives. When we speak of asymptotic estimates and leading terms, we shall always mean "as $h \to 0$".

5.1. Methods not Using Local Truncation Error

Many other methods have been suggested for adaptive placement of mesh points. A good survey of such methods for two-point boundary value problems is given in Russell and Christiansen [1978]. This paper gives many references, most of which will be omitted here.

Most of the alternative methods for two-point boundary value problems in effect try to approximately equidistribute a lower derivative of the solution than that occurring in the expression for the local truncation error. For example, if a second-order finite difference method is used, then the local truncation error for a first order system usually depends on the third derivative. Alternative methods attempt to equidistribute the first or second derivative. A variation of the first derivative method is to use arc length. An early method, used among others by Pearson [1968], attempted to equidistribute the variation in the solution (for monotone solutions), and this can be considered as attempting to

equidistribute an approximation to the first derivative.

A method which does not fall into any of these categories is that of White [1979]. He introduces a new variable and considers the mesh distribution as a function of it. The equidistributing mesh and the solution are computed simultaneously by solving the nonlinear equations using a finite difference method with equal mesh spacing. We believe this approach is too expensive to use in our context, where we deal with explicit methods.

Russell and Christiansen state that, in their context, when one frequently has a very crude first approximation to the solution, equidistribution using a lower order derivative frequently produced a better mesh. But they conclude (if one excludes methods such as White's) "If high accuracy is required, the [mesh selection] strategy should ... incorporate the asymptotic form of the error." In our situation, where we (in principle) must find a spatial mesh which approximately equidistributes the error that would be made in taking the next time step, we have a very good approximate solution at the current time. We agree with Russell and Christiansen, because Proposition 4.2 shows that significant economies can be achieved by controlling the local truncation error. This analysis also shows that we should control different order derivatives in the interior and at the boundary (if, as usual, we use first order boundary approximations and second order interior approximations). Additionally, the use of the truncation error (third derivatives) in a sense contains the information of the lower derivatives, but not conversely. Furthermore, the existing programs for ordinary differential equations (both initial value problems and boundary value problems), as well as our own results in Chapter 8, suggest the efficacy of this approach in practice.

5.2. Four Methods

To motivate what will be entailed in producing an error estimate for the initial boundary value problem, let us examine a procedure for the initial value problem for a first order system of ordinary differential equations. For a p-th order linear ρ-step method on a uniform mesh with step size h, the local truncation error is

$$C_{p+1}h^{p+1}y^{(p+1)}(x_m) + O(h^{p+2}), \tag{5.1}$$

where C_{p+1} is a known constant, y is the exact solution, and $x_{m-\rho}, x_{m-\rho+1}, \ldots, x_m$ are the gridpoints involved in the method [Henrici, 1962]. Direct use of divided differences to approximate the first term of (5.1) generally produces a poor estimate.

Instead one performs an *extra computation* to estimate the local truncation error. We assume that the local truncation error (5.1) is for an implicit multistep method called the corrector. Then we use another (explicit) linear multistep method called the predictor, which has the same form of local truncation error, but with a different constant C^*_{p+1}. (We assume the order p is the same, for simplicity.)

If one makes reasonable smoothness assumptions about the solution (and even if the corrector is not iterated to convergence), one obtains an estimate for the first term in the asymptotic expansion of the local truncation error (5.1) by subtracting the predicted and corrected values y^*_m and y_m at x_m:

$$C_{p+1}h^{p+1}y^{(p+1)}(x_m) = \frac{C_{p+1}}{C^*_{p+1} - C_{p+1}}(y_m - y^*_m) + O(h^{p+2}),$$

[Henrici, 1962, p. 257]. This is called *Milne's device*. It obviates the need to approximate the high order derivative in (5.1).

For the initial boundary value problem, we will also need extra computations to estimate the leading term of the asymptotic expansion of the local

truncation error. Consider the interior of a refinement, where the mesh is uniform. Later we will consider boundaries and coarse/fine interfaces. The local truncation error for the Lax-Wendroff approximation to P1 on a uniform mesh with stencil centered at (x,t) is

$$\frac{k}{6}(-k^2 u_{ttt}(x,t) - h^2 u_{xxx}(x,t)) + O(h^4), \tag{5.2}$$

where $k = \Delta t$ is the time step and $h = \Delta x$ is the space step.

We will now consider four methods for estimating the dominant terms of this error.

5.2.1. Differences

We use the differential equation to replace t derivatives by x derivatives in the expression (5.2) for the local truncation error. We then obtain

$$ck\frac{h^2}{6}u_{xxx}(x,t)(c^2\lambda^2 - 1) + O(h^4). \tag{5.3}$$

We may now approximate u_{xxx} by a five-point divided difference at points in the interior of a refinement. Specifically, with t dependence omitted,

$$\begin{aligned} u_{xxx}(x) = {} & (-2u(x-2h) + 4u(x-h) - 4u(x+h) + 2u(x+2h))/4h^3 \\ & - h^2 u_{xxxxx}(\xi)/4, \end{aligned}$$

where $x-2h < \xi < x+2h$.

5.2.2. Estimating an Interpolant with 'Small' Derivative

The second method is based on a theorem of Favard [de Boor, 1975a, b]. Given a function defined on mesh points, how large can the k-th derivative of a "smooth" interpolant to $k+1$ of these function values be? Favard gives this bound in terms of certain divided differences. de Boor's contribution was to greatly reduce the constant appearing in this inequality.

More specifically, let $\{x_i\}_1^{n+k}$ be a sequence of strictly increasing mesh points in the interval $[a, b]$, and let $\{g_i\}$ be the given grid function values at these points. We can certainly find an interpolant f to the gridfunction which has $k-1$ continuous derivatives on the interval $[a, b]$, whose $k-1$-st derivative is absolutely continuous, and whose k-th derivative is in $L_p[a, b]$. Favard's theorem states that among such interpolants there exists one for which

$$||f^{(k)}||_{\infty,[x_i,x_{i+1}]} \leq K_0(k) \max_{i-k\leq j\leq i} k!\,|[x_j, \ldots, x_{j+k}]g|,$$

where $[\ldots]g$ denotes the k-th divided difference of g at the indicated points. (The norm indicates that the supremum is taken for the interval $x_i \leq x \leq x_{i+1}$.) This provides us with a method for bounding the k-th derivative of *some* interpolant to the gridfunction in terms of (computable) divided differences.

We can use this to estimate the local error by again replacing t derivatives by x derivatives to obtain (5.3). We assume that the third derivative of the interpolant approximates the third derivative of the exact solution u. Using the theorem, we then estimate u at a point x_i by taking the maximum of the three third-order divided differences which "include" the point x_i, and multiplying by $K_0(3)$, which is 6.854.

Table 5.1 gives a typical result of this procedure. The computation was performed on P1 (the first order wave equation), with parameters as in Table 8.1 (Chapter 8), except that the maximum number of refinement levels was 5, and the refinement ratios $M = N = 4$. The values given were at time $t = 3.6$, at locations $x = 3.10, 3.15, 3.20, 3.25$. Clearly, the bounds so obtained are hopelessly conservative. (Bounds at other x were also typically off by factors of eight to ten.) Therefore, this procedure was abandoned.

	Third Derivative of u			
Estimated (Bound)	38911	38848	22928	12842
Actual	-24.8	9680.9	-2185.4	-1614.3

Table 5.1 Estimating Interpolant with 'Small' Derivative

5.2.3. Two-Step Richardson Extrapolation

In this method we again replace t derivatives in the expression for the local truncation error with x derivatives. Next we take a step forward in time, using a stencil centered at (x,t) with spacing k and h, and obtain expression (5.3) for the local error.

We then perform a separate step forward in time, using a stencil centered at (x,t) with spacing $2h$ and k, which has local error

$$c\frac{k}{6}(2h)^2 u_{xxx}(x,t)(c^2(\frac{\lambda}{2})^2 - 1) + O(h^4).$$

By subtracting these two estimates and multiplying by $(c^2\lambda^2 - 1)/3$, we obtain an estimate for the local truncation error of the first calculation. This method, illustrated in Figure 5.1 (with stencils not overlayed for clarity), uses values at only the time levels t and $t + k$. It should be noted that the difference scheme used for this method (and the next method to be discussed) must have the same order of accuracy in both space and time. Excluded are schemes such as Oliger's [1974] $O(h^4 + k^2)$ method. This is a mild restriction, as we usually take second order methods anyway.

5.2.4. Three-Step Richardson Extrapolation

This error estimation method is much more general than the preceding ones. Here we do *not* rewrite t derivatives in the local truncation error in terms of x derivatives. We shall apply it not only to our model problem, but to our

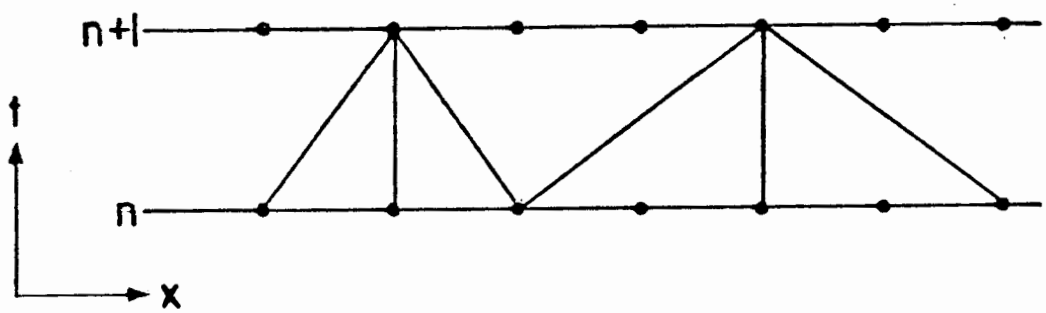


Figure 5.1 2- step Richardson

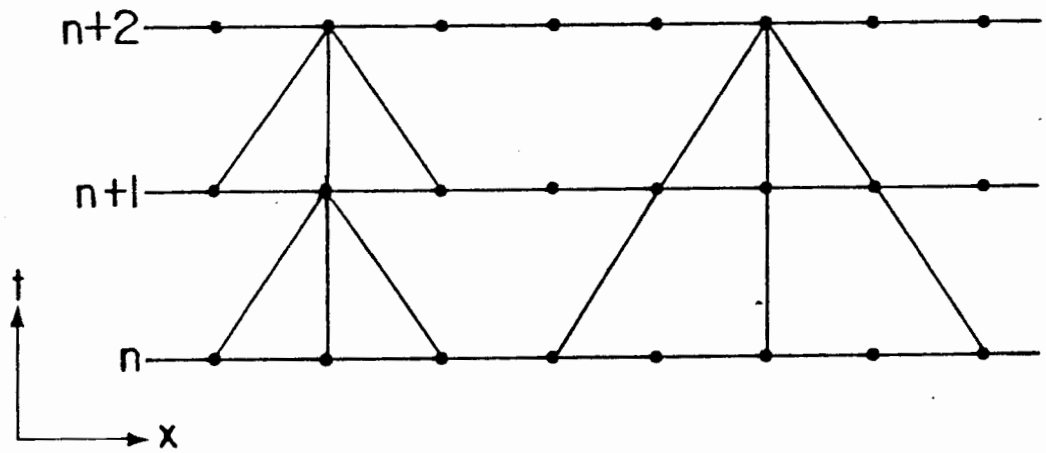


Figure 5.2 3- step Richardson

XBL 822-165

linear hyperbolic operator

$$Lu \equiv u_t - A(x,t)u_x - B(x,t)u = F(x,t). \tag{2.1}$$

In the interior of a refinement we approximate this system by any linear multi-(time) level explicit difference scheme whose local truncation error per unit time step has the same order p in space and time:

$$v_\nu(t+k) = \sum_{\sigma=0}^{\rho} Q_\sigma v_\nu(t-\sigma k) + kF_\nu(t), \qquad \nu = r, r+1, \ldots, N_0-q. \tag{5.4}$$

Here $x = x_\nu \equiv a + \nu h$, $t = t_m \equiv mk$,

$$Q_\sigma = \sum_{j=-r}^{q} A_{j\sigma}(x_\nu + jh,\ t-\sigma k,\ h)E^j \qquad \sigma = 0, 1, \ldots, \rho, \tag{5.5}$$

the $A_{j\sigma}$ are matrix coefficients, and E is the shift operator. We shall write (5.4) symbolically as $L_h v(x,t) = kF(x,t)$. If both the differential and difference equations have constant coefficients, we shall prove the validity of this method. In practice, our interior difference approximation will always be two-level ($\rho = 0$).

The local truncation error is given by

$$\begin{aligned} u(x,t+k) = \sum_{\sigma=0}^{\rho} Q_\sigma u(x,t-\sigma k) + kF(x,t) + k(h^p T_1(x,t) + k^p U_1(x,t)) \\ + O(h^{p+2}), \end{aligned} \tag{5.6a}$$

where T_1 and U_1 are sufficiently smooth functions of x and t. Symbolically,

$$L_h u(x,t) = kF(x,t) + k\tau(u,x,t). \tag{5.6b}$$

In fact, we will define the local truncation error $k\tau$ for *any* sufficiently smooth function w by

$$L_h w(x,t) - kLw(x,t) = k\tau(w,x,t). \tag{5.7}$$

(For this purpose we assume that the difference approximation is defined for *all* (x,t), not only at mesh points.) For the global truncation error $e(x,t) = u(x,t) - v_\nu(t)$ we obtain the expansion

$$e(x,t+k) = \sum_{\sigma=0}^{\rho} Q_\sigma e(x,t-\sigma k) + k(h^p T_1(x,t) + k^p U_1(x,t)) + O(h^{p+2})$$

by subtracting (5.4) from (5.6).

In the three-step Richardson method, we take one (time) step of the approximation (5.4) with $x = x_\nu$, $t = t_m$, using mesh spacing h in space and k in time. (See Figure 5.2 for a two-level scheme, with stencils shifted horizontally for clarity.) We then repeat the step using (5.4) with t_m replaced by t_{m+1} and the same mesh spacing. (Before performing this second step, we will need to generate more points on level $t = (m+1)k$ by applying (5.4) with (x_ν, t_m) replaced by $(x_{\nu+j}, t_m)$, $j = -r, -r+1, \ldots, 0, 1, \ldots, q$. Thus in practice this estimation is done for all interior points of a refinement at once.) We obtain the approximation $v_\nu(t+2k)$ with error

$$\begin{aligned} e(x,t+2k) = \sum_{\sigma=0}^{\rho} Q_\sigma e(x,t+k-\sigma k) &+ k(h^p T_1(x,t+k) + k^p U_1(x,t+k)) \\ &+ O(h^{p+2}). \end{aligned} \tag{5.8}$$

Next, we take a step using (5.4) with $(x,t) = (x_\nu, t_m)$, but with spacing $2h$ and $2k$ (*i.e.*, replace jh by $2jh$ and k by $2k$ in (5.5)). This difference approximation will be called $v_\nu^{(2)}(t+2k)$. The error $e^{(2)}(x,t+2k) = u(x,t+2k) - v_\nu^{(2)}(t+2k)$ is

$$\begin{aligned} e^{(2)}(x,t+2k) = \sum_{\sigma=0}^{\rho} Q_\sigma^{(2)} e(x,t-2\sigma k) &+ 2k((2h)^p T_1(x,t) + (2k)^p U_1(x,t)) \\ &+ O(h^{p+2}). \end{aligned} \tag{5.9}$$

(The superscript 2 denotes a double stencil.) Here we see that using a multi-level scheme will entail storing many previous time levels of the solution. We also see why the order of the method must be the same in both space and time.

In (5.8), we can change the arguments of T_1 and U_1 to (x,t) (by Taylor expansion) at the cost of an $O(h^{p+2})$ error. We subtract the result from (5.9) to

obtain the computable quantity

$$\begin{aligned}\Delta &= v_\nu(t+2k) - v_\nu^{(2)}(t+2k) = e^{(2)}(x,t+2k) - e(x,t+2k)\\ &= \sum_{\sigma=0}^{\rho}[Q_\sigma^{(2)}e(x,t-2\sigma k) - Q_\sigma e(x,t+k-\sigma k)] + (2^{p+1}-1)k(h^p T_1(x,t)\\ &+ k^p U_1(x,t)) + O(h^{p+2}) \equiv \Theta + (2^{p+1}-1)k\tau(u,x,t).\end{aligned} \tag{5.10}$$

We now need an expression for Θ. (It is tempting to suppose that $\Theta = O(h^{p+2})$, but this is not quite true.) If the undifferentiated term Bu in the differential equation is zero, we can assume that the coefficients $A_{j\sigma}$ are independent of h. But if $B \neq 0$, we shall assume that the $A_{j\sigma}(h)$ are sufficiently smooth functions of h. Substituting the definition (5.5) of Q_σ in the expression for Θ yields

$$\sum_{\sigma,j}[A_{j\sigma}(2h)e(x+2jh,t-2\sigma k) - A_{j\sigma}(h)e(x+jh,t+k-\sigma k)].$$

We now expand the e's in a Taylor series in h and k about the point (x,t), and the coefficients about $h = 0$. In the product we keep only terms up to (and including) the first order in h and k. Since e is $0(h^p)$, the neglected terms h^2e_{xx}, hke_{xt}, and k^2e_{tt}, h^2e, etc., are $O(h^{p+2})$. After an elementary computation we obtain

$$\sum_{\sigma,j}A_{j\sigma}(0)[jhe_x - (k+\sigma k)e_t + kA'_{j\sigma}(0)e] + O(h^{p+2}).$$

By consistency, $\sum_{\sigma,j}A_{j\sigma}(0) = I$,

$$(I + \sum_{\sigma,j}\sigma A_{j\sigma}(0))B = \sum_{\sigma,j}A'_{j\sigma}(0),$$

and

$$\lambda(I + \sum_{j,\sigma}\sigma A_{j\sigma}(0))A = \sum_{\sigma,j}jA_{j\sigma}(0),$$

where I is the identity matrix. (These relations are obtained by expanding both sides of (5.6) in the same manner as above, replacing u_t by using the differential equation, equating coefficients of u and u_x, and finally equating coefficients of

h^0 and h^1 in the result.) Substituting these in Θ yields

$$(I + \sum_{j,\sigma}\sigma A_{j\sigma}(0))[\lambda h A e_x - k e_t + k B e] + O(h^{p+2}) = -k(I + \sum_{j,\sigma}\sigma A_{j\sigma}(0))Le + O(h^{p+2}).$$

Using (5.7), we can write $-kLe = k\tau(e) - L_h e$. But

$$\begin{aligned} L_h e &= L_h u - L_h v \\ &= (L_h u - kLu) + kLu - L_h v \\ &= k\tau(u). \end{aligned}$$

(These are the equations for the method of deferred corrections. Thus our three-step Richardson method could also be considered as a deferred correction method. See Pereyra [1973] or Keller [1968].) Therefore, Θ becomes

$$k(I + \sum_{j,\sigma}\sigma A_{j\sigma}(0))(\tau(e,x,t) - \tau(u,x,t)) + 0(h^{p+2}). \tag{5.11}$$

Now e is $O(h^p)$, and, since the difference method is convergent, $\tau(e)$ is $O(h^{p+1})$. We can therefore substitute Θ into (5.10) and obtain

$$\Delta = k[(2^{p+1}-2)I - \sum_{\sigma,j}\sigma A_{j\sigma}(0)]\tau(u,x,t) + O(h^{p+2}).$$

We have shown

Theorem 5.1 Approximate the hyperbolic operator (2.1) by the consistent multilevel explicit interior difference scheme (5.4). Assume that both operators have constant coefficients. If the undifferentiated term Bu in (2.1) is nonzero, assume that the coefficients (5.5) in the difference operator are smooth functions of h. Assume that the local truncation error per unit time step τ (5.6) has the same order p in space and time, that the solution u of the differential equation and the global error $e = u - v$ are sufficiently smooth functions of x and t, and that $\lambda = k/h$ = constant. Also assume e is of order p. (For the initial boundary value problem (2.1)-(2.4) Gustafsson [1975] or Proposition 4.1 gives sufficient conditions for the latter to hold.) Then we can estimate the (lowest terms of the asymptotic expansion of the) local truncation error $k\tau(u,x,t)$ at

the point $(x,t) = (x_\nu, t_m)$ in the interior of a refinement using the three-step Richardson method, and

$$\begin{aligned} k\tau(u,x,t) &\equiv kh^p d_1(x,t) = k(h^p T_1(x,t) + k^p U_1(x,t)) + O(h^{p+2}) \\ &= [(2^{p+1}-2)I - \sum_{\sigma,j} \sigma A_{j\sigma}(0)]^{-1}(v_\nu(t+2k) - v_\nu^{(2)}(t+2k)) + O(h^{p+2}), \end{aligned}$$

where T_1 and U_1 are sufficiently smooth functions of x and t, $v_\nu(t+2k)$ is the approximation obtained by applying one step of (5.4) with t replaced by t_{m+1} and mesh spacing h and k, and $v_\nu^{(2)}(t+2k)$ is the approximation obtained by applying one step of (5.4) with $(x,t) = (x_\nu, t_m)$ and mesh spacing $2h$ and $2k$.

In Chapter 9 we suggest possible generalizations of this theorem.

We have called these methods "Richardson extrapolation" and have mentioned that three-step Richardson is also a deferred correction method. However, we are using these methods in a non-traditional way. Both our method and the traditional approaches (for o.d.e.'s and elliptic p.d.e.'s) improve the accuracy of the approximate solution by estimating the local truncation error. But we use the estimate to decide where to refine; the traditional approaches add the estimate to the approximate solution. (Doing the latter would not be useful to us, since our estimation is not being done at every time step.) As a consequence, the traditional approaches improve the order of accuracy of the basic difference scheme; as we pointed out in Section 4.4, our method does not.

In all three methods, the quantity we control in the interior is not the local truncation error, but the local truncation error per unit time step

$$|\tau(u,x,t)| \le \delta,$$

where δ is the local error tolerance supplied to the program. This was shown in the last chapter; one power of h in accuracy is lost in going from the local truncation error of the interior approximation to the global error [Gustafsson, 1975].

Our numerical results in Chapter 8 show that for our model problem P1, any of the three methods produces approximately equally accurate estimates of the local truncation error in the interior. Thus it is clear that three-step Richardson is more expensive to compute than two-step Richardson. However, we recommend the exclusive use of three-step Richardson in the interior of refinements because of its greater generality and convenience. The other two methods required us to write the t derivatives as x derivatives. This frequently can be done, but it may be extremely cumbersome. For example, the Lax-Wendroff method applied to the inviscid Burgers' equation

$$u_t + uu_x = 0$$

has (after replacing t derivatives by x derivatives) local truncation error

$$\frac{k}{12}[k^2(-9uu_x^3 - 15u^2u_xu_{xx} - 2u^3u_{xxx}) + h^2(3u_xu_{xx} + 2uu_{xxx})] + O(h^4).$$

This effectively excludes the use of differences; the situation could be much worse for a system of equations. Even with the use of the symbol-manipulation program MACSYMA, the coding of the expressions for the local truncation error could be very tedious.

The great advantage of the three-step Richardson method is that we need not rearrange or even calculate (by hand or by MACSYMA) the local truncation error of the difference scheme; one need only know the order p of the method and the factor $2^{p+1}-2$ used to divide the difference $v_\nu - v_\nu^{(2)}$ of the two approximations at time $t+2k$.

5.3. Coarse/Fine Interfaces

Let us now discuss the modifications needed for coarse/fine interfaces which do not abut boundaries. For concreteness, assume that an l-th level refinement R_l has a descendant $l+1$-st level refinement R_{l+1} which does not abut

the left or right boundaries $x = a$ or $x = b$. This introduces two coarse/fine interfaces, namely, the ends of R_{l+1}. (See Figure 2.3 for the left end of R_{l+1}.) Recall that, the last time we estimated the error, we added enough padding or buffering (see Section 2.6) to both ends of R_{l+1} to ensure that waves could not escape it, plus two extra level l (spatial) cells. This guarantees that we will not need to refine the ends of refinement R_{l+1} (unless they abut boundaries) and assures "smooth" mesh transitions. Since our local truncation error estimates are used only to decide where to refine, we can safely set our estimate at the ends of R_{l+1} to zero.

As a less attractive alternative, we could set the estimate at the ends of R_{l+1} to the corresponding estimate at the same spatial position in R_l. This would require us to estimate the errors from the coarsest mesh to the finest, which is somewhat inconvenient. But we implemented this and found it produces the same results as the easier method given above.

The next question is the choice of estimator at mesh points which are one (spatial) mesh point on the "fine" side of a coarse/fine interface, or one mesh point away from a boundary. (This is for the case of a stencil with three adjacent spatial points, *i.e.*, $q = r = 1$ in (2.10) or (5.4). In the case where q or r is greater than one, similar considerations apply to the q or r points on the "fine" side of the interface.) Figures 5.1 and 5.2 show that neither of the Richardson methods yields an estimate here. We also set the estimate to zero here, for the same reasons as before.

5.4. Boundaries

Let us consider local error estimation at boundaries. On the left boundary, there are J boundary conditions specified for the differential equations (2.1)-(2.4). We can approximate these in the obvious way with no local truncation

error. We will call these "exact" boundary approximations.

When $r \geq 1$ in the interior difference approximation (2.10) or (5.4), then we need $n-J$ "extra" boundary conditions at the left boundary,

$$v_\mu(t+k) = \sum_{\sigma=-1}^{\rho} S_\sigma^{(\mu)} v_r(t-\sigma k) + g_\mu(t), \qquad \mu = 0, \tag{5.12}$$

where $S_\sigma^{(\mu)}$ is as given in (2.12), but with the appropriate time level. If $r > 1$, we also need $n(r-1)$ additional boundary conditions of type (5.12) for $\mu = 1,\ldots,r-1$. (Similar statements hold at the right boundary, with J replaced by $n-J$ and r replaced by q. We will only discuss the left boundary; the right is similar.) We will first consider the extra conditions; at the end of the next section we shall examine the "exact" boundary approximations.

The local truncation error $k\tilde{\tau}$ of (5.12) is

$$u(x_\mu, t+k) = \sum_{\sigma=-1}^{\rho} S_\sigma^{(\mu)} u(x_r, t-\sigma k) + k\tilde{\tau}(u,x,t), \qquad \mu = 0, 1, \ldots, r-1.$$

For a restricted class of boundary approximations, it is tempting to recycle Theorem 5.1 to produce the following false proposition.

Proposition 5.1. Assume that the hypotheses of Theorem 5.1 apply instead to the boundary approximation (5.12). That is, the centered difference operators Q_σ are replaced by the uncentered difference operators $S_\sigma^{(\mu)}$; and T_1, U_1, kd_1, p and τ are replaced by T_2, U_2, d_2, $\tilde{p}$ and $\tilde{\tau}$, respectively. Assume that the boundary approximation is consistent with the differential equation, explicit ($S_{-1}^{(\mu)} = 0$ for all μ), and its local truncation error per unit time step has the same order $\tilde{p}$ in space and time

$$\tilde{\tau}(u,x,t) = (h^{\tilde{p}} T_2(x,t) + k^{\tilde{p}} U_2(x,t)) + O(h^{\tilde{p}+1}).$$

In addition, assume the global error e is smooth and of order p, where $p = \tilde{p}$ or $p = \tilde{p}+1$. Then the local truncation error of the boundary approximation may be estimated by three-step Richardson as in Theorem 5.1, using $\tilde{p}$ in place of p.

For the "proof" we first note that Theorem 5.1 did not require that the stencil be centered. Next we must examine the magnitudes of the terms $\tilde{\tau}(u)$ and $\tilde{\tau}(e)$ in (5.11). If the order of the boundary approximation is greater than p, then the term $\tilde{\tau}(e)$ has the same order as $\tilde{\tau}(u)$ and cannot be neglected. If the order of the boundary approximation is less than $p-1$, then it is known that Gustafsson's theorem [1975] does not hold, and the global error e is not $O(h^p)$. So even though this corollary holds here, this is of no interest.

Unfortunately, the proposition fails because the boundary operator was not the only operator used to produce solution values at previous time levels. For example, if we use the first order upwind boundary approximation and the Lax-Wendroff interior approximation on our problem P1 (the first order wave equation), then, to obtain a boundary estimate, we apply upwind three times, and Lax-Wendroff once. Nevertheless, the proposition can sometimes be used in practice. A detailed calculation shows that, for problem P1, with Lax-Wendroff and upwind differencing, we should divide the difference of the two estimates at the boundary not by $2^{\tilde{p}+1}-2 = 2$, but by $2 + \lambda_l$ instead. Since the use of buffers makes our method robust, this small change produces almost no difference in practice. A similar change from 2 to $2 + \lambda_l$ is needed at both boundaries in Problem P2, (the second order wave equation), to be introduced in Chapter 8.

Thus, this Richardson method is not very useful at boundaries for several reasons. First, we must do a tedious calculation of the local truncation error for each different problem. Second, there are relatively few boundary approximations which have the same order spatial and time error. Third, we doubt that this method works for implicit approximations.

Since two-step Richardson suffers from the same restrictions, differences must be used for all other boundary conditions. In contrast to the interior approximation, it is usually practicable to write down the local truncation error

for the boundary approximation. (Furthermore, the boundary approximation is usually of lower order accuracy than the interior one, so we can use lower order differences.) Then we must rewrite x derivatives in terms of t derivatives. For example, in our model first order wave equation, if we use upwind differencing at the right boundary

$$v_\nu(t+k) = v_\nu(t) - c\lambda(v_\nu(t) - v_{\nu-1}(t)),$$

the local truncation error is

$$\tfrac{1}{2}\cdot(k^2 u_{tt} - c\lambda h^2 u_{xx}) + 0(h^3);$$

replacing t derivatives by x derivatives yields

$$\tfrac{1}{2}ckh(c\lambda-1)u_{xx} + O(h^3). \tag{5.13}$$

If we are using differences, we simply replace the u_{xx} term by a one-sided finite difference. If we are using the two-step Richardson method, we obtain (5.13) for the truncation error when using the stencil with spacing h and k. We obtain

$$\tfrac{1}{2}ck(2h)(c\lambda/2-1)u_{xx} + O(h^3)$$

when we use the stencil centered at the same point but with spacing $2h$ and k. We can then subtract these and multiply the result by $(c\lambda-1)$ to obtain an estimate of the local truncation error at the boundary.

The local truncation error for extrapolation boundary conditions

$$(hD_+)^j v_0(t+k) = 0, \qquad \text{for fixed } j \geq 1$$

can only be estimated by differences. For $j = 2$, we simply estimate the truncation error

$$h^2 u_{xx} + O(h^3)$$

by using four-point one-sided differences (since the three-point estimate yields zero.)

In Section 8.7 we numerically compare different methods of error estimation at boundaries.

Gustafsson's [1975] analysis and our own Proposition 4.1 show that the order of accuracy of the boundary approximation may be one order lower (but not less) than that of the interior approximation, in order to preserve the global order of accuracy, which is then the same as the order of the local error per unit time step for the interior approximation. This means that when we are examining the truncation error at boundary points to see if refinement is necessary, we should *not* control the local error per unit time step, but instead the local error,

$$|k\tilde{\tau}(u,x,t)| \leq \delta,$$

where δ is the local error tolerance input to the algorithm. Our computations in Section 8.5 confirm that we can indeed use one less order of accuracy at the boundaries, and still obtain the desired order for the global error.

5.5. Systems

So far our discussion has proceeded as though we were estimating the error for a single differential equation. For a system, we simply estimate the error in each component and take the maximum absolute value at each spatial mesh point. Then we base refinement decisions *for all solution components* on this maximum. Thus, the refinements are the same for all solution components.

This procedure is quite conservative, and will result in refinements not being inserted in unnecessary regions if the assumption of Section 2.1 is fulfilled: that steep gradients occur in approximately the same positions for all solution components. This procedure also results in a simplification for our data structures, as described in Section 6.3.

One final detail concerns the difference approximations corresponding to boundary conditions in the differential equation (the so-called "exact" boundary approximations). Using the obvious difference approximation for these conditions yields zero local truncation error. But there is a difficulty with using zero in our error estimation procedure. This can be seen in our problem P1 (Section 2.5), the first order wave equation. The left boundary condition contains a forcing (inhomogeneous) term g which results in the wave entering the region.

Clearly, we want to put refinements around any "large" wave entering the left as soon as possible. If we set the truncation error estimate to zero at the boundary, we will not detect the entering wave until it has already entered the region. This can be remedied by treating the forcing term g_1 or g_2 of (2.3) or (2.4) as generating a local truncation error. For our first order wave equation, where $u(0,t) = g(t)$, we write down the local truncation error (5.2) for the interior approximation, then replace x derivatives by t derivatives, using the differential equation. Since $u_{ttt}(0,t) = g_{ttt}$, we can analytically differentiate g to arrive at the result. This procedure was actually used in our computations with P1 in Chapter 8. (Notice that we used an interior error for a boundary approximation, and then controlled the local error per unit time step. We could equally well have used the boundary error, which depends on u_{tt} and controlled the local error. In either case we would control the same power of h.) For a system of equations, the procedure is very simple if we use our assumption that gradients occur at approximately the same positions in different solution components. Suppose that at no boundary do we use *only* "exact" boundary approximations. (This is not true for our first order wave equation, nor would it be true when all components are prescribed by inflow boundary conditions on the same boundary, as in supersonic inflow.) Then some of the components of the difference approximation have nonzero local truncation error, and our usual error estimates for these component(s) will detect any incoming wave. This

technique was used in our problem P2 (the second order wave equation) in Chapter 8.

If the boundary conditions are of the supersonic inflow type in *all* components, then we may assume our problem (2.1)-(2.4) is in diagonal (characteristic) form, and the left boundary condition

$$u^{\mathrm{I}} = Su^{\mathrm{II}} + g_1 \tag{2.3}$$

has $S = 0$ (since u^{II} has no components, *i.e.*, $u = u^{\mathrm{I}}$). In this case, we can proceed as in our first order wave equation. For a local truncation error of the form

$$\alpha u_{xxx} + \beta u_{ttt} \tag{5.14}$$

we analytically differentiate g_1 to obtain the second term. The first term can be approximated by one-sided spatial differences or else rewritten in terms of t derivatives using the differential equation,

$$u_x = A^{-1}(u_t - Bu - f).$$

In practice, this rewriting may be cumbersome. (The right boundary is treated similarly.)

The final case is for systems where gradients do not occur in the same positions in different solution components. We again need to estimate (5.14), with u replace by u^{I}. We approximate the first term by one-sided spatial differences. To get the second term, we differentiate the boundary condition, obtaining

$$u^{\mathrm{I}}_{ttt} = Su^{\mathrm{II}}_{ttt} + (g_1)_{ttt},$$

and use the differential equation to replace u^{II}_{ttt} by u_{xxx} . Then we use analytic differentiation for $(g_1)_{ttt}$, and one-sided differences for u_{xxx}. (We have tacitly assumed that S was constant, but a nonconstant S introduces no additional difficulties.)

CHAPTER 6

Data Structures

In this chapter, we discuss the choice of data structures appropriate for our mesh refinement algorithms. The data structures used are not nearly so important to our algorithm in one space dimension as are the other details, such as estimating the local truncation error. However, this choice becomes much more important in two space dimensions.

We will see that the data structure has two parts: a structure to show the relationships between refinements (a four-way linked tree of records), and a mechanism for storing solution values (second components) of refinements (sequential allocation of deques). Then we will discuss alternative implementations. We will describe the deques first.

6.1. Deques

Let us examine the operations to be performed on the solution values. First, it is convenient to keep all the points in a single refinement together. We also need to be able to "move" a refinement to the left or right. This is accomplished for a wave moving right by deleting points from the left of the mesh and adding points on the right. Similarly, we need to be able to delete points from the right and add them on the left. We also need to merge two refinements, and to split them apart. (These operations are needed when two pulses "cross" each other, that is, they are waves traveling in opposite directions.) Finally, we may need to create a refinement or delete it.

The key operations are adding or deleting points from the left or right end of a refinement. Points are never inserted in or deleted from the middle. A data structure having these properties is called a "deque", or a double-ended queue [Knuth, 1973]. We are then faced with the problem of storing a collection of deques.

A natural way to store the solution values for refinements is sequentially, as shown in Figure 6.1. Here we see a region with two refinements, and one of the refinements itself contains a refinement. The solution values for the coarsest mesh occupy a fixed region at the lower end of a vector which we will call v. The refinements occupy contiguous sections of the remaining available memory, with variable-width gaps of free space separating the memory occupied by refinements. The gaps allow us to expand or contract refinements (to a limited degree) without moving the function values in memory. The solution values corresponding to refinements are ordered as follows.

The coarse mesh is labelled refinement 1, and its solution values always occupy the lowest end of the vector v. It is followed in v by the "second level" refinements (labelled 2 and 3 in Figure 6.1), which are ordered in v in the same order as the refinements are encountered in proceeding from left to right in the computational region. Following these are the "third level" refinements (as is refinement 4 in Figure 6.1), again in the same order as they occur in a left-to-right scan of the computational region, and ignoring the positions of any coarser (second level) refinements encountered in the computational region. Then would appear all fourth level refinements, and so forth. This scheme duplicates certain solution values in the vector v, namely the ones which correspond to mesh points which lie on different level refinements. However, doing this makes the program much simpler.

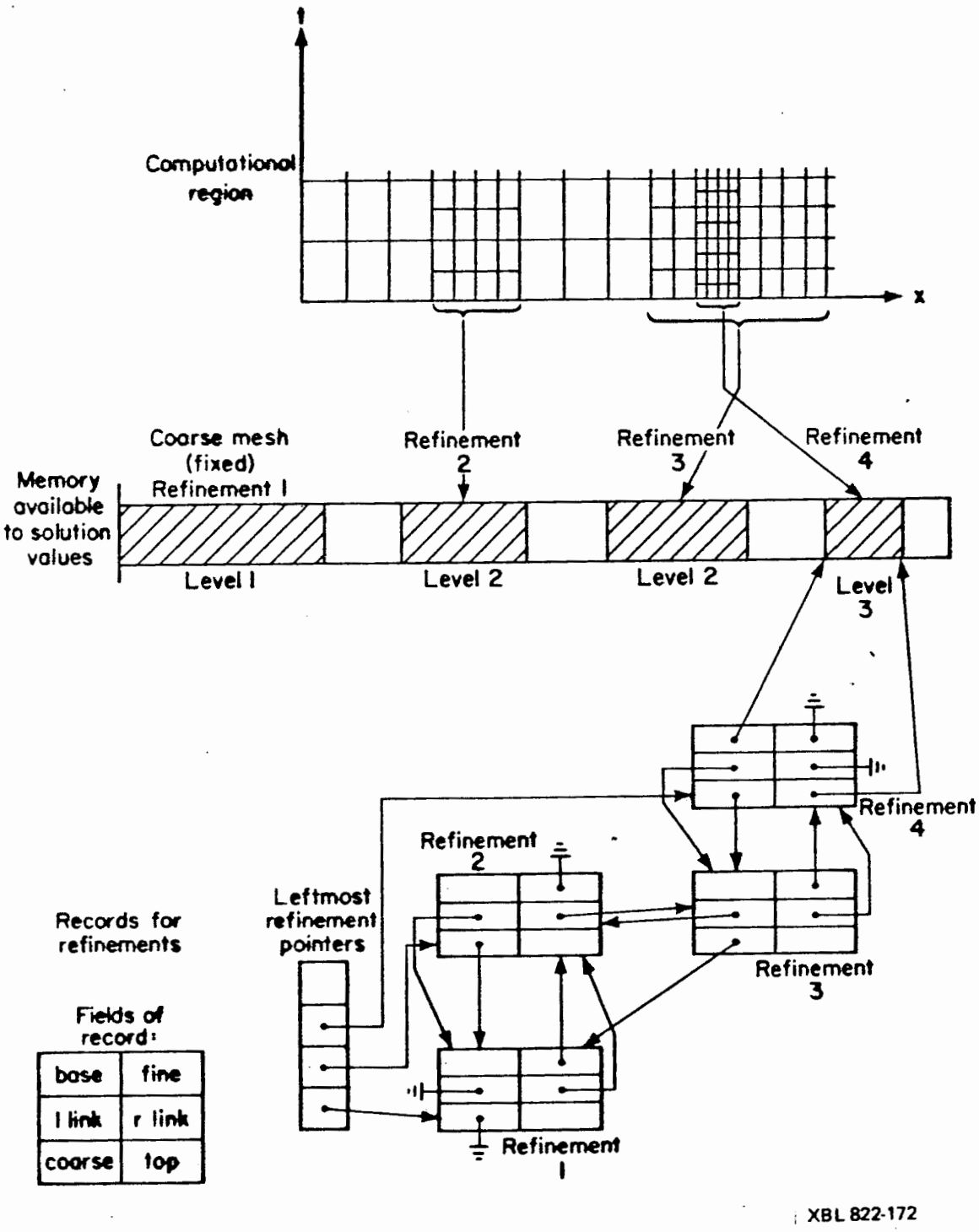


Figure 6.1 Data Structure: Tree and Vector of Deques

6.2. Trees

Next we will describe the (four-way linked) tree of records, which is necessary to show relationships between refinements. Each node (record) corresponds to a refinement. Trees are natural in this context, since we use recursive refinements, and are used in all adaptive solvers for elliptic equations described in Chapter One. In the following, we will identify a refinement with its node (record), and use the term "refinement" to mean "the node corresponding to a refinement". We will sometimes call the coarsest mesh a "refinement" for uniformity.

Obviously, a node has as many branches (descendants) as it has refinements. The coarsest mesh corresponds to the root of the tree. The root has level 1, its immediate successors are at level 2, their successors have level 3, and so forth. Each node contains all the information about a refinement, except its solution values. We will now describe some of this information. We first need to know where in the vector v the solution values for a refinement are located. This is done by using two indices *base* and *top*. This is shown in Figure 6.1 for the fourth (level 3) refinement, but omitted for the other refinements to avoid clutter. Also needed is a pointer to the parent of a refinement (shown in Figure 6.1 as the field *coarse)*. Furthermore, we need pointers to all refinements of a refinement. We can avoid using a variable number of pointer fields for this by using the usual device. We use one pointer to the leftmost descendant (called *fine* in Figure 6.1) and then chain together all immediate descendants (siblings) using the "right" pointers, called *rlink* in the figure. A refinement other than the root also needs indices to denote its endpoints within its parent, that is, which part of its parent it refines. These are not shown in the figure.

Since we will often be adding or deleting nodes, we decided to implement the record structure as a linked list. Up to this point our records form a *triply linked tree*, exactly as in Knuth [1973, p. 352]. However, additional links are needed.

The solution is advanced in time, and the error is estimated a level at a time. Because we already have the *rlink* pointers, we can chain together *all* refinements on the same level (not just those with a common parent) using *rlink*. Then we introduce an array of pointers pointing to the leftmost refinement on each level. (These are shown in Figure 6.1.) This is related to the *level-order representation* of a tree [Knuth, 1973, p. 350].

The last operation needed on our data structure is a repacking of the v array, to be discussed shortly. This requires us to sweep through the v array in both directions, as will be seen. Thus we also require our *rlink* pointers to point from the rightmost refinement (node) on level l to the leftmost refinement on level $l+1$. To enable a leftward sweep, we introduce "left" pointers *llink*, which are inverse to the *rlink* pointers. That is, if node p has right pointer *rlink* pointing to node q, then q has left pointer *llink* pointing to p.

The result of using all these pointers is a four-way linked tree: a triply linked tree with the additional property that all the leaves (nodes) are linked together in a doubly linked list. The linked list starts at the root and proceeds to the leftmost refinement of level 2, then through all the refinements of level 2 (in left-to-right order), next to the leftmost node of level 3, and so forth. This structure is similar to one that Knuth [1973, p. 356] suggests for manipulating multi-variable polynomials. The difference is that in his scheme, the direct descendants of any node are doubly linked together; this means that if level l has k direct descendants, each descendant in turn having descendants, then level $l+2$ has exactly k such doubly linked lists. In our scheme, all the nodes are linked in

one doubly linked list in level-order.

Because the space devoted to records is small relative to the space consumed by solution values, the space for all the pointers in our scheme is inconsequential.

We now examine how the operations on refinements are effected using this data structure. Advancing the difference approximation (in time) can be done a level at a time, starting with the highest (most refined) level, using the *llink* and *rlink* pointers and the leftmost pointers on each level. Here we also use the "ancestor" or *coarse* pointers to copy solution values from finer meshes to coarser meshes for points x which lie on more than one refinement. The error estimation is done in the same manner.

Similarly, we adjust the refinements level by level, starting with the highest (finest) level. The mesh adjustment operations can be effected using four elementary operations, which are natural for a deque. They are shorten left, shorten right, extend left, and extend right. Shortening either end of a refinement is a trivial operation, accomplished by moving a *base* or *top* index. Deleting a refinement is the same, but also involves removing a record from the tree. If there is enough space available, extending either end of a refinement involves changing an index, copying solution values from the parent refinement, and filling in new solution values using linear or quadratic interpolation in space. Creation is the same, plus the operation of inserting a new node in the tree. Separation of a refinement into two refinements involves changing indices and inserting a new node. Finally, merging two refinements is easy because we insisted on the left-to-right ordering of refinements in the solution value vector. We move left the solution values of the right refinement, if necessary, then extend the left refinement to the right, change some indices, and delete the right node. Complicating the last two operations is the need to adjust pointers

to descendant refinements.

6.3. Memory Repacking

A problem occurs during an "extend" operation when there is insufficient expansion room between refinements. This calls for a repacking of memory, and two algorithms for doing this are given by Knuth [1973, pp. 245-6] for the case of a sequence of stacks (rather than deques). We will therefore describe the modifications to these algorithms for our data structure.

When a refinement runs out of room in the v vector, moving only the adjacent refinement will probably cause *another* repacking to occur soon, so it is better to reallocate *all* available memory when a refinement runs out of room. Knuth breaks this into two parts: Algorithm G, which decides how to allocate the free memory to the refinements, and Algorithm R, which actually moves the refinements into the positions dictated by Algorithm G. It is Algorithm R which requires the forward and backward sweep of the v vector in order to avoid overwriting any information.

Our Algorithm R differs from Knuth's only in that our refinements are indexed from zero rather than one. So only Algorithm G is of interest. Knuth's main idea is to share ten percent of the free memory equally among the refinements, and the other ninety percent is divided proportionately to the amount of increase in refinement size since the previous repacking. This idea is not useful in our case. For a traveling wave, all refinements stay about the same size, but "move". However, we can modify this rule by awarding the ninety percent of available memory proportionately to the amount each refinement has *moved* since the last repacking. We discover whether a refinement has moved primarily left or right (in memory) since the last repacking, and award its share of the ninety percent to its left or right, respectively.

This change to Knuth's algorithm greatly reduces the number of repackings compared to more naive allocation methods. Since the coarse mesh doesn't move it receives none of the ninety percent allocation. Furthermore, the higher level refinements move further (measured in number of mesh points, not physical distance) than the lower level ones, so they are awarded more free space by this scheme.

So far we have discussed the case of a single scalar equation. If we are instead solving a system of n equations in one space dimension, only slight modifications are needed, given our crucial assumption that the refinements for all solution components are the same. We simply use the same tree-like record structure, and store the solution values in a matrix with n rows. Each row has exactly the same structure of solution values for refinements separated by gaps, as illustrated in Figure 6.1. Now we repack memory whenever one (hence all) of the components needs repacking.

An advantage of this organization is that the mesh-adjusting mechanism is separated from the differential-equation-advancing and error-estimating mechanism; to solve a different system of equations requires changes only in the differential equation advancement and error-estimation calculations.

6.4. Alternative Data Structures

Having described the data structures we actually used, we will attempt to justify our choice by examining some alternatives that we rejected.

One alternative that suggests itself is to store solution values in a matrix (for the case of one differential equation) of dimensions μ x ν (where μ is the maximum number of refinements, and ν is the maximum number of mesh points in a refinement), instead of using deques. This is inconvenient, wasteful of storage space, and inefficient in execution speed. It is inconvenient because the

size of refinements is unpredictable, and if one refinement exceeds ν in size, the computation stops, even though there may be much memory available for other refinements. It is wasteful of storage space, for the same reason. In addition, the maximum number of refinements is also unpredictable, and thus a great deal of space is wasted for nonexistent refinements. (In our scheme, one specifies the maximum number of refinements to be much larger than necessary. The sequentially allocated deques and the repacking algorithm then assure that all the memory allocated to solution values is used efficiently. The only wasted memory is occupied by records corresponding to nonexistent refinements, and this is small compared to the space occupied by solution values.)

Finally, use of this scheme probably implies that a wave that moves left or right is being implemented (in at least one direction) by moving (in memory) *all* the solution values in a refinement. This is inefficient, because on many machines (e.g., the CDC 7600) it costs almost as much to move an item as to add two items. Furthermore, we then need a three-dimensional array for a system of differential equations, and this, too, will lead to inefficiencies. If one decides to avoid excessive memory moving by adding points to the left or right of a refinement, one might as well abandon the array entirely, and use our scheme.

An alternative to our tree readily suggests itself -- a threaded tree [Knuth, 1973, pp. 332 ff.]. A threaded tree has links pointing back to ancestors, similar to our *coarse* links, but only from rightmost descendants. But as Knuth [1973, p. 352] points out, finding ancestors of nodes is not as convenient with such a scheme; a triply-linked tree is more convenient. Furthermore, we use the links that are used for "threads" instead to chain together refinements on the same level.

Another alternative involves the method of storing solution values. Our method is the one Knuth suggests for storing a sequence of stacks. But Knuth suggests a different scheme for storing a sequence of deques [Knuth, 1973, p. 249]. As before, *base* and *top* point to the bottom and top, respectively, of the memory *available* to the refinement in question. But this time the *actual* left of the refinement is pointed to by *front*, which may be between *base* and *top* (Figure 6.2a). Similarly, the actual right of the refinement is not at *top*, but at *rear*, which is initially between *front* and *top*. Suppose now this refinement "moves" right, but does not expand in area (this is the case for a traveling pulse). Then if there is not enough room to the right of *rear* (between *rear* and *top)*, the refinement is stored in its available memory using circular wraparound, as illustrated in Figure 6.2b. Now the refinement has been split into two parts in its available memory, with the free space intervening. A refinement always has one of the two forms shown.

This scheme has the virtue of reducing the number of memory repackings, since we repack memory only when the space available to an individual refinement has run out, and not when a refinement "moves" too far left or right. Thus, with a traveling pulse we need never repack memory. Despite this, however, we decided not to use the wraparound scheme. It greatly complicates the differential equation calculation, in effect introducing still another interface, in addition to all the coarse/fine interfaces. We instead were willing to allow extra memory repackings, which occur infrequently anyway.

In sum, our data structures seem well-suited to manipulating refinements in one space dimension. For two or more space dimensions, the situation becomes much more complicated. A tree structure to exhibit the relationship between refinements is usually used; and additional structures may be necessary. The storage of solution values cannot be generalized from our scheme. See, e.g., Rheinboldt and Mesztenyi [1980], Gannon [1980], Berger, Gropp, and

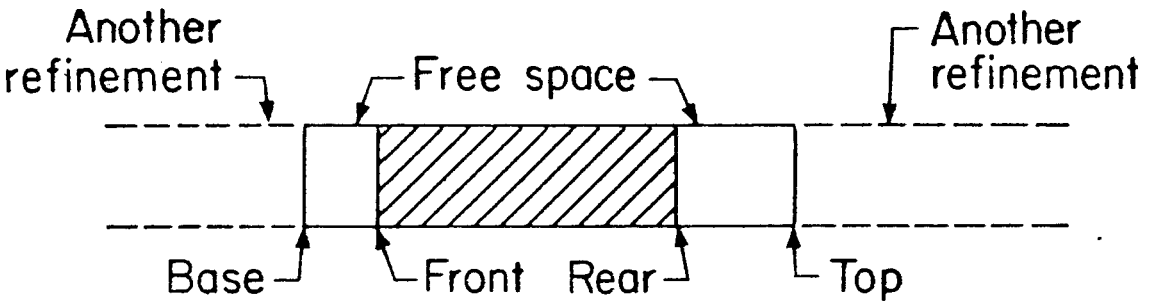


Figure 6.2a

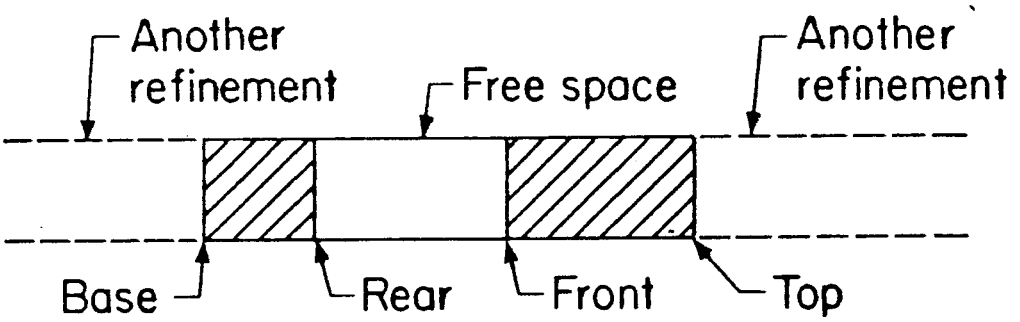


Figure 6.2b

XBL 822-167

Alternative Storage for Deques

Oliger [1980].

CHAPTER 7

Choice of Programming Language

In this short chapter, we explain and justify our choice of implementation method for the programs used in our computations.

Possible alternatives include Algol W, PL/I, Algol 60, Algol 68, Pascal, Fortran, and Fortran with preprocessor. The arguments against the first four are lack of availability of a compiler and/or lack of portability. Raw Fortran (even Fortran 77) is cumbersome to use because of the lack of control and data structures, both of which are crucial for our task. However, most numerical software is written in Fortran, and if one uses another language, there must be an interface to Fortran. It was with some regret that we were unable to use Pascal, despite its excellent data and control structuring facilities. It does have a Fortran interface, but it can be very awkward to use [Mohilner, 1977]. Furthermore, earlier versions of Pascal compilers required array bounds to be known at compile time -- a restriction even more severe than Fortran imposes. Finally, we are interested in portability, and using a Pascal/Fortran interface does not lend itself to this.

Due to the desire for portability, two other approaches were rejected. The first is Feldman's [1979] EFL, which is a Fortran preprocessor specifically designed for numeric computations. It requires the writing of a sizable two-pass translator. This translator is written in the language C, which is achieving wider use, but cannot be said to be portable. The other approach is Grosse's [1978] language T. T is implemented as a preprocessor which generates PL/I output; unfortunately PL/I is far from portable. Both of these languages did merit consideration, though, since their respective authors have devoted considerable

thought to the problem of appropriate language constructs with which to express numerical algorithms.

The two portable Fortran preprocessors we examined were Kernighan's [1975] Ratfor and Cook and Shustek's [1975] Mortran. Although we believe the former is more widely used, we chose the latter because it is far more general and flexible. Brandt [1977] has also decided to use a macro preprocessor for Fortran to implement his software for the multigrid method.

The term Mortran, like Fortran, has several meanings. It can mean a structured source language, a translator for that language, or a macro-processor. The structured language is implemented as a set of macros which are used by the Mortran macro processor to translate the language into Fortran. The resulting Fortran program is then run like any other Fortran program. Figure 7.1 shows the mechanics of running a Mortran program.

In contrast to most other Fortran preprocessors, the Mortran preprocessor is written in a portable subset of ANSI (standard) Fortran. Hence the Mortran preprocessor, and, more importantly, Mortran source programs, are portable between different machines. Furthermore, Mortran source and Fortran source can be intermixed, so the Mortran user has access to all existing Fortran software.

We felt that there was one property of Mortran which made it especially desirable for this project: *extensibility*. This means that new data structures, operations on data structures, and control structures can be added to the language (at rather small cost in implementation time) by adding additional macros to the language.

Let us contrast this situation with other languages such as Fortran or Pascal. Fortran is completely inextensible. Pascal is extensible only in the sense that one can add additional data types. For example, Pascal has no **complex**

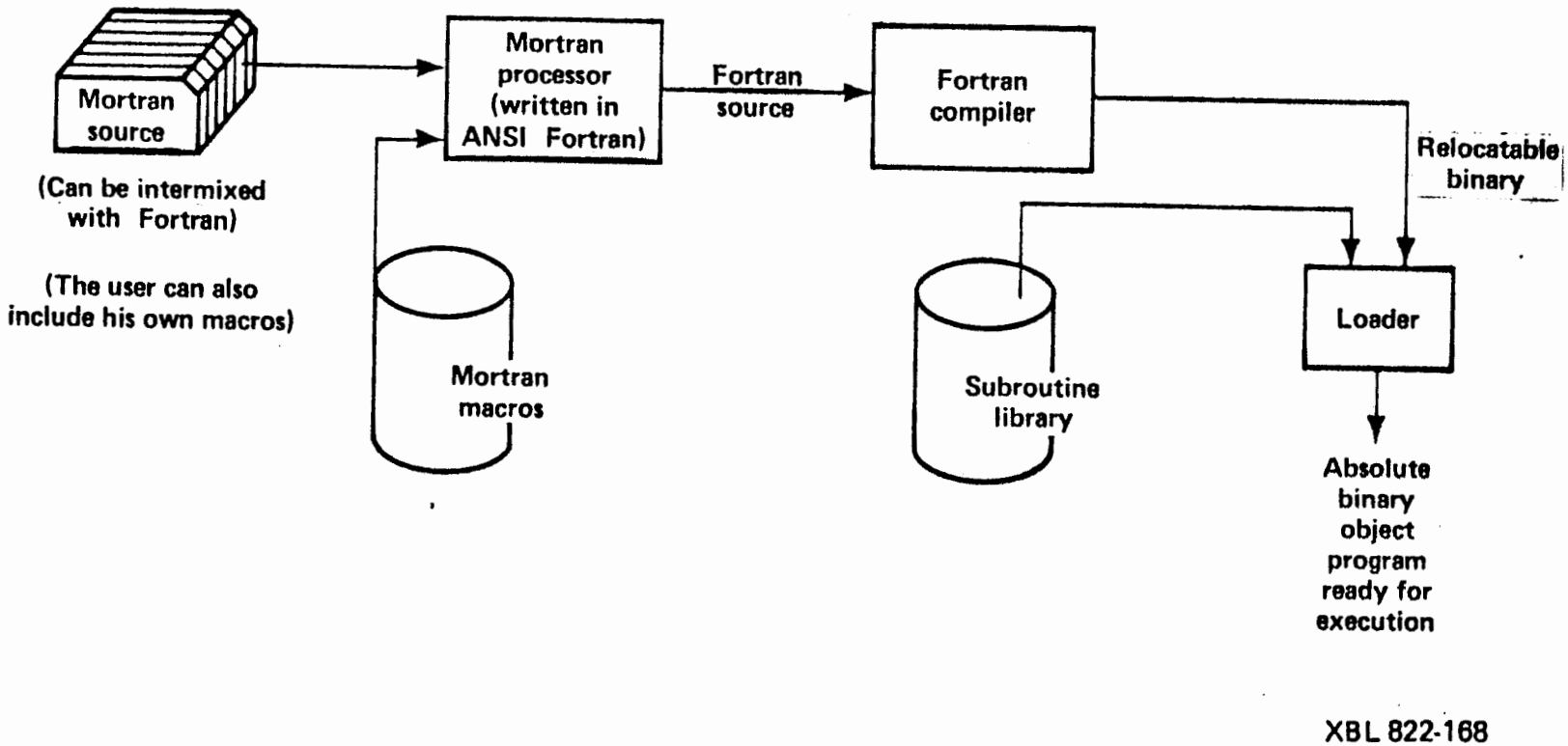


Figure 7.1 Running Mortran Programs

data type. But one can be added to the language by defining records consisting of two real fields. However, one is unable to extend Pascal by defining new *operations* for data types, e.g., complex addition or multiplication. One is forced to use subroutine calls instead. Furthermore, in Pascal one is unable to define new control structures, such as a loop with premature exit. By contrast, Mortran (but not T or EFL or Ratfor) permits all of these things.

Two applications of this extensibility are important to our algorithm. From our description of the data structures (four-way linked trees and deques) it is obvious that we need records and pointers, in order to operate conveniently and efficiently on refinements. Mortran allows us to define these new data types and to define operations (such as following the pointer) on these data types. We did this using a modification of the method in Zahn [1975]. Pascal-like notation is used [Jensen and Wirth, 1974].

A second application of extensibility is to defining new control structures. In contrast to many of the traditional numerical algorithms (such as linear equation solvers or eigenvalue routines) where operations with loops predominate, in our programs (especially the sections involving merging, separating, creating, destroying, and moving refinements) the use of decisions is extensive. Mortran has adequate decision (or "conditional") statements (like **if...then, if...then...else),** indefinite looping constructs **(while...do, repeat...until,** etc.) and (nested) block structure within subroutines. The lack of these features in Fortran would have resulted in code whose correctness would be exceedingly difficult to verify by visual inspection.

Additional features of Mortran which are helpful but not essential are additional control structures (such as **next** and **exit** to prematurely exit a loop), alphanumeric statement labels (however, most labels disappear because of the rich supply of control structures), free field format (column and card boun-

daries ignored), comments inserted anywhere in the text, conditional (alternative) compilation, and variable names of arbitrary length.

Of course, some of the restrictions of Fortran remain. Among these are lack of dynamic storage allocation, lack of arrays with arbitrary subscripts, and recursion. Although our refinements are nested recursively, we almost always operate on them in "level order" (one level at a time). The only exception is when we are graphing solutions. Then we need to search the tree in preorder, so we need to simulate recursion. The subscript problem is harder to solve, and requires modifying the Mortran preprocessor to accept macro-time expressions. We did not do this. Instead, since we needed only zero array subscripts, we used an extra dummy element preceding the array in **common** to achieve this effect. The dynamic storage limitation is impossible to overcome, but is not crucial.

Another aspect of the macro preprocessor is the kind of (Fortran) code it produces. For the **while...do, repeat...until, if...then,** and **if...then...else** constructs, the Fortran code produced is as efficient as possible without using global flow analysis. The only problem is with **for** loops. The macro processor has two stacks to allow nesting of loops, but there is no stack to remember symbols, such as loop indices. Therefore, all testing and incrementing for **for** loops are done at the top of the loop, and this is not quite as efficient as testing and incrementing at the bottom. Mortran does, however, (correctly) test the loop condition before initially entering (unlike the Fortran **do**). Mortran allows the Fortran **do** (with all its restrictions and generation of efficient code) so we compromised. We kept the existing Mortran **for** statement, but redefined the **do** statement to check the loop limits once before entering the loop (to test for a null loop) and then generate the usual (efficient) Fortran **do**.

In sum, we felt that the use of Mortran greatly aided the development of the programs for this project. For further details on Mortran, refer to Cook and

Shustek [1975] and Zahn [1975]. Reading Mortran programs is quite easy for those familiar with modern block-structured languages. The only things to note are that left angle bracket means **begin** and right angle bracket means **end,** as in Algol or PL/I. Other notation (especially that dealing with records and pointers) is similar to that of Pascal [Jensen and Wirth, 1974]. In particular, the notation ($p\uparrow.field$) or ($p\frown.field$) denotes the $field$ of the record pointed to by the pointer p.

We finally remark that portability was indeed achieved by this method, since our programs were run on an IBM 370/168, a CDC 7600, and a DEC VAX with minimal conversion problems between them.

CHAPTER 8

Computational Results

In this chapter we answer the following questions about the method described in the preceding chapters:

1. Does our method "work", *i.e.*, does it "follow" or "track" steep gradients? Is it fooled by background effects?
2. Is the algorithm sufficiently general to allow refinements to be created, destroyed, merged, separated, moved, and to abut boundaries?
3. Is the algorithm sensitive to the direction of characteristics, or dependent on knowing that certain boundary conditions are inflow or outflow?
4. Will the method handle nonlinear problems?
5. How well will the method follow discontinuities or shocks?
6. Are recursive refinements worthwhile?
7. How should one choose the refinement ratios N and M?
8. How efficient is the method, both in execution time and memory?
9. Does the global error behave according to the theory of Chapter 4 as $h, k \to 0$?
10. How do the three methods of interior local error estimation of Chapter 5 compare in accuracy and efficiency?
11. How do different boundary approximations and methods of estimating their error affect the solution?
12. How often should one monitor the local truncation error (and adjust refinements)?

8.1. Model Problems

Since we believe it is impossible to answer these questions analytically, we resort to numerical experiments on model problems. We now introduce several such problems. Problem P1, the first-order wave equation, was introduced in Section 2.5. (We again use $\alpha = 200$.)

P2 is the second order wave equation, written as a 2×2 first order system, with "open" boundary conditions (*i.e.*, the boundaries are "transparent" to traveling waves). As exact solution we use two counter-streaming Gaussian pulses, superimposed on a sinusoidal background. The differential equation is

$$u_t = Au_x, \qquad a \le x \le b,\ 0 \le t,\ 0 < c, \tag{P2}$$

where

$$A = \begin{pmatrix} 0 & c \\ c & 0 \end{pmatrix},$$

with initial conditions

$$\left.\begin{aligned} u_1(x,0) &= f(x) + g(x) \\ u_2(x,0) &= -f(x) + g(x) \end{aligned}\right\}, \qquad a \le x \le b,$$

and open boundary conditions

$$\left.\begin{aligned} u_1(a,t) &= u_2(a,t) + 2f(a - ct) \\ u_1(b,t) &= -u_2(b,t) + 2g(b + ct) \end{aligned}\right\}, \qquad t \ge 0.$$

As in P1, we choose $a = 0$, $b = 4$, $c = 1$. The exact solution is

$$u_1(x,t) = f(x - ct) + g(x + ct),$$
$$u_2(x,t) = -f(x - ct) + g(x + ct).$$

To produce our interacting pulses, we take $f(x)$ exactly as in P1, and

$$g(x) = -\exp(-\alpha(x-4.5)^2),$$

where $\alpha = 200$ as before. Once again, each pulse occupies about 8 percent of the region $a \le x \le b$. This problem decisively answers questions 2 and 3 above, since the pulses start out outside the computational region (only the sinusoid being present). They then enter the region, and two sets of recursive refinements are set up. The pulses eventually cross, so the sets of refinements must merge and then separate. Finally, the pulses exit the region, so all refinements (except the coarse mesh) are destroyed. Note that both boundaries act as inflow boundaries at some times and as outflow boundaries later on. This problem thus shows that nothing we have done depends on the direction of wave motion. It also shows that the method works for a system.

The difference approximation in the interior is again Lax-Wendroff

$$v_j(t+k_l) = (I + Ak_l D_0^l + \tfrac{1}{2}A^2 k_l^2 D_+^l D_-^l)v_j(t),$$

with coarse/fine approximation

$$v_j^{l-1}(t+k_l) = (I + Ak_l D_0^{l-1} + \tfrac{1}{2}A^2 k_l^2 D_+^{l-1} D_-^{l-1})v_j^{l-1}(t)$$

at interfaces, the obvious initial condition, and obvious boundary approximations for v_1. For v_2, we need extra boundary conditions at both $x = a$ and $x = b$, and we use either

(a) upwind/downwind differencing:

$$v_{j2}(t+k_l) = (I + ck_l D_+^l)v_{j2}(t) \quad \text{at } x = a,$$

$$v_{j2}(t+k_l) = (I + ck_l D_-^l)v_{j2}(t) \quad \text{at } x = b,$$

where $v_{j2}(t)$ denotes an approximation to $u_2(x,t)$ at $x = a + jh_l$ on an l-th level refinement; or

(b) first-order extrapolation

$$(D_+^l)^2 v_{j2}(t+k_l) = 0 \quad \text{at } x = a,$$

$$(D_-^l)^2 v_{j2}(t+k_l) = 0 \quad \text{at } x = b.$$

(The first 2 in each line is an exponent, not a superscript.) Gustafsson, Kreiss and Sundström [1972] showed that both approximations (a) and (b) are stable with Lax-Wendroff, according to their stability definition.

Let $H(x)$ denote the Heaviside unit function, which is 0 for $x < 0$, and 1 for $x \geq 1$. Our third problem is again a scalar equation, the inviscid Burgers equation

$$u_t + \tfrac{1}{2}(u^2)_x = 0, \qquad 0 \leq x \leq 4,\ t \geq 0,$$

with boundary condition

$$u(0,t) = 1, \qquad t \geq 0,$$

and initial condition

$$u(x,0) = 1 - H(x - 0.1), \qquad 0 \leq x \leq 4.$$

The exact solution is

$$u(x,t) = 1 - H(x - \tfrac{1}{2}t - 0.1),$$

a shock traveling to the right with speed $\tfrac{1}{2}$. We approximate this using the usual two-step Lax-Wendroff method, as given in Richtmyer and Morton [1967, p. 302] with $F(u) = \tfrac{1}{2}u^2$. Naturally we modify this by adding subscripts and superscripts l in the appropriate places. The only dissipation in our scheme is that inherent in the method itself. Since our previous calculations used time-dependent boundary conditions, we use constant ones here. (Of course, at coarse/fine interfaces, we make modifications as in P1 and P2.)

8.2. Qualitative Results

Figures 8.1(a) to 8.1(i) illustrate our algorithm when applied to problem P1, the first order wave equation. We used a coarse mesh of 81 points, with

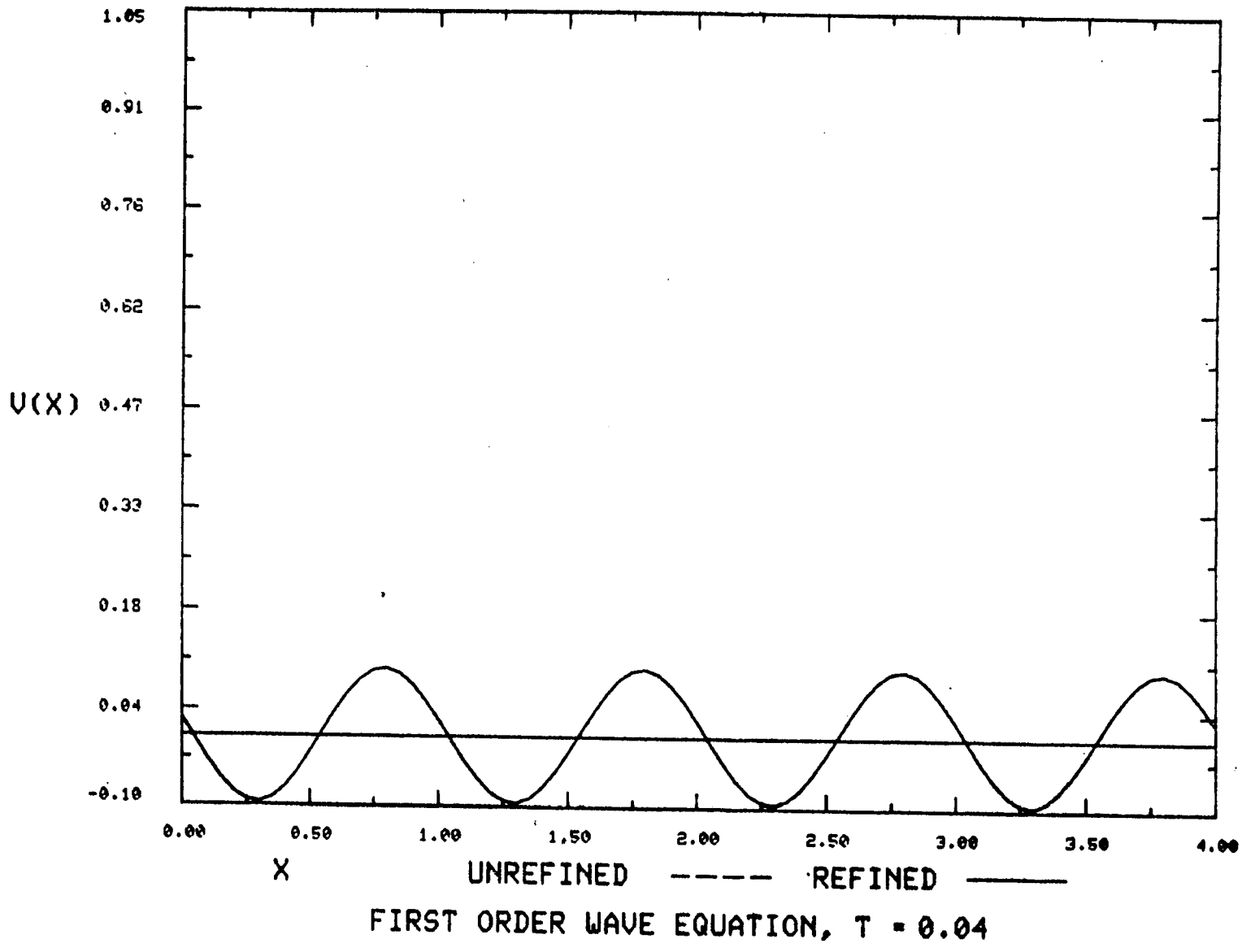
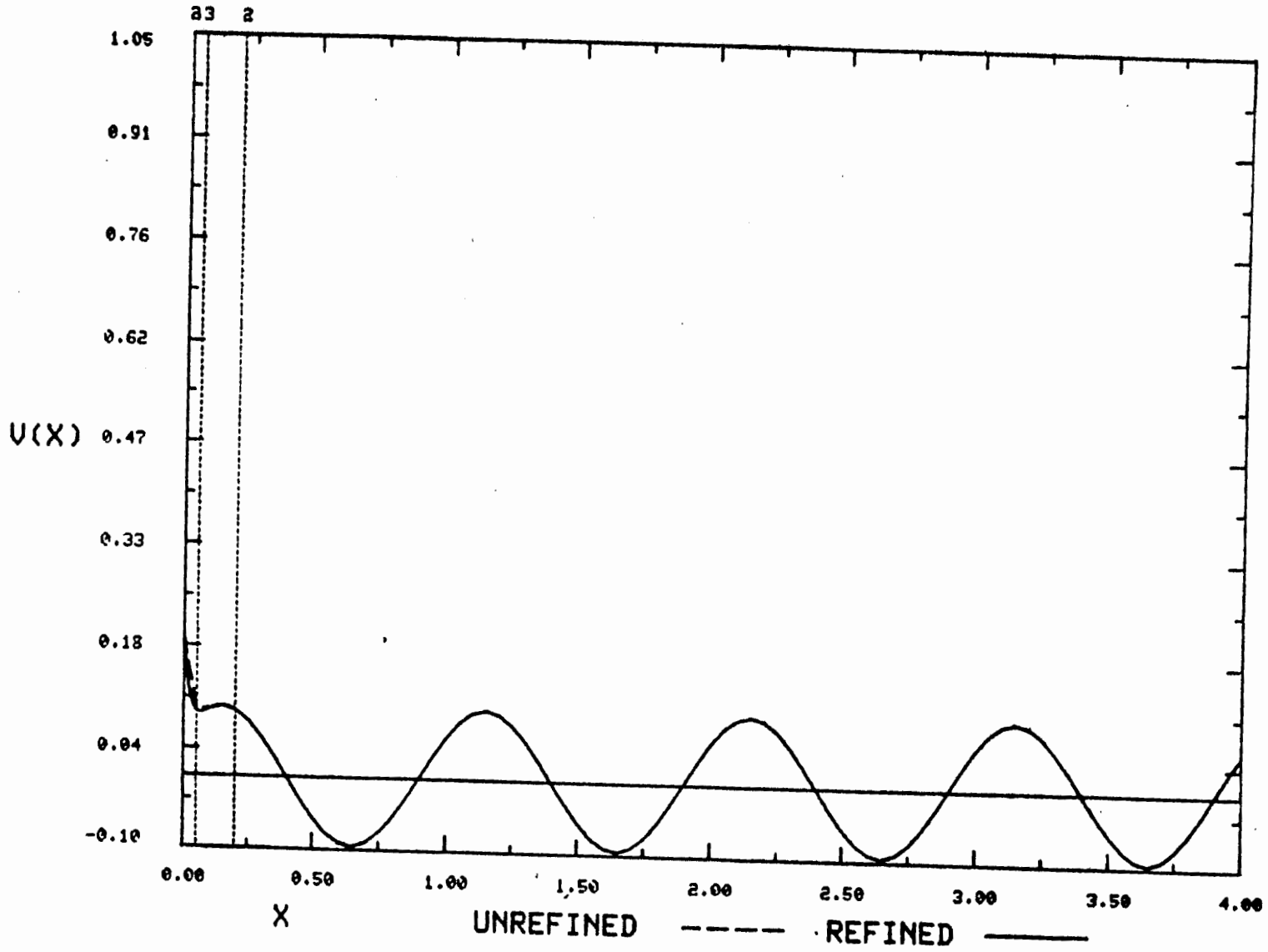

1.05
0.91
0.76
0.62
V(X) 0.47
0.33
0.18
0.04
-0.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
FIRST ORDER WAVE EQUATION, T = 0.04

Figure 8.1(a)



FIRST ORDER WAVE EQUATION, T = 0.40

Figure 8.1(b)

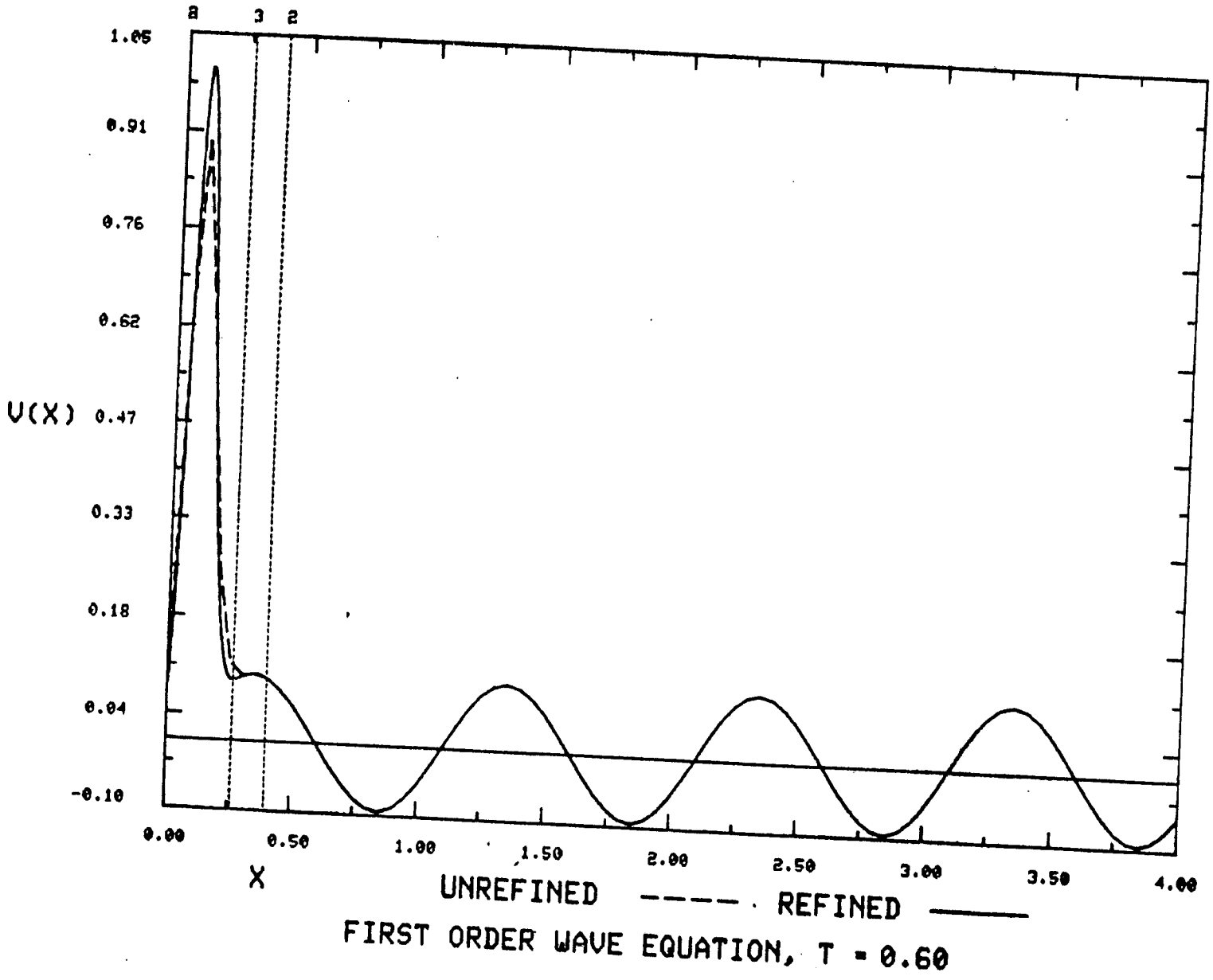

8
3
2
1.05
0.91
0.76
0.62
V(X)
0.47
0.33
0.18
0.04
-0.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
FIRST ORDER WAVE EQUATION, T = 0.60

Figure 8.1(c)

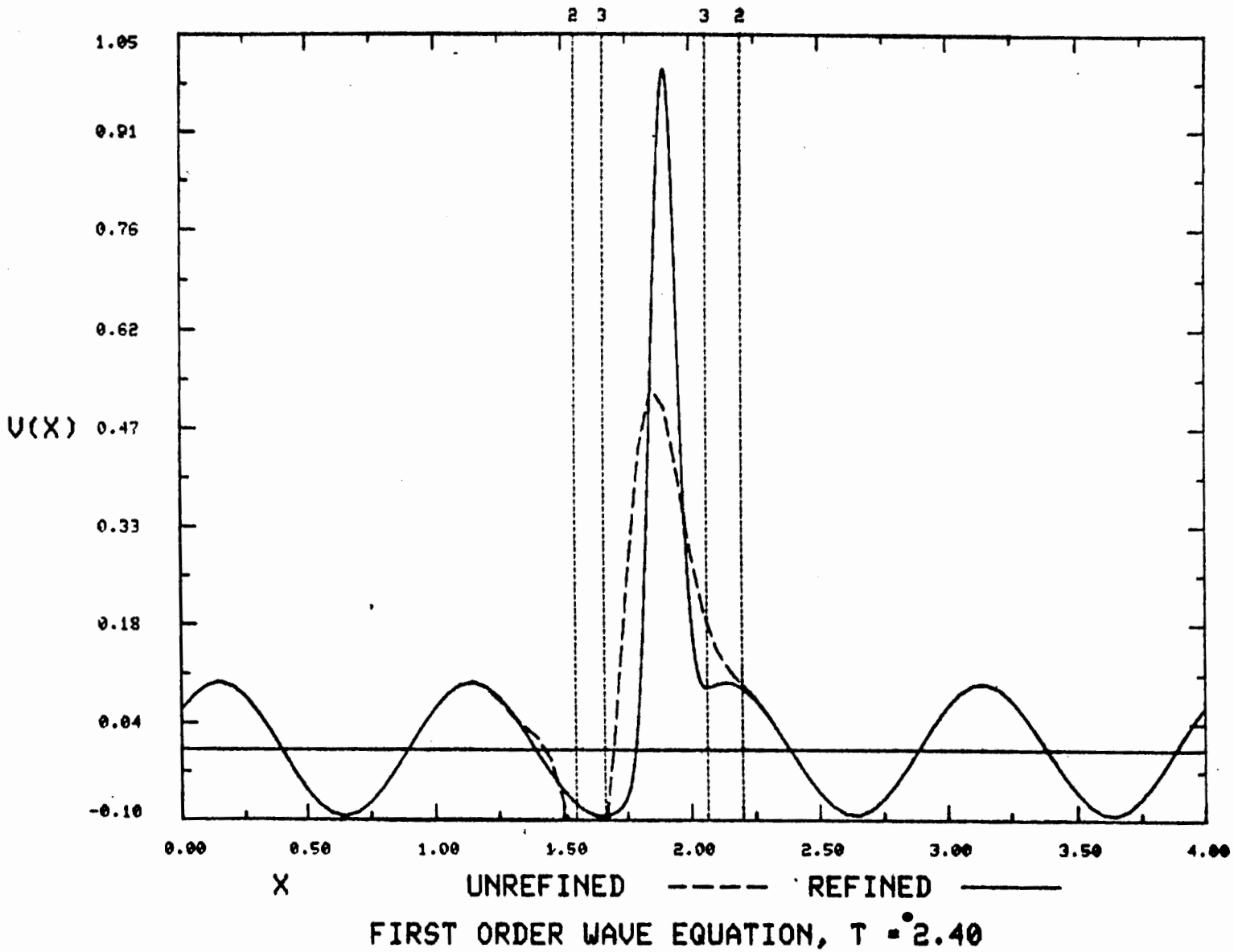

2 3
3 2
1.05
0.91
0.76
0.62
V(X) 0.47
0.33
0.18
0.04
-0.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
FIRST ORDER WAVE EQUATION, T = 2.40

Figure 8.1(d)

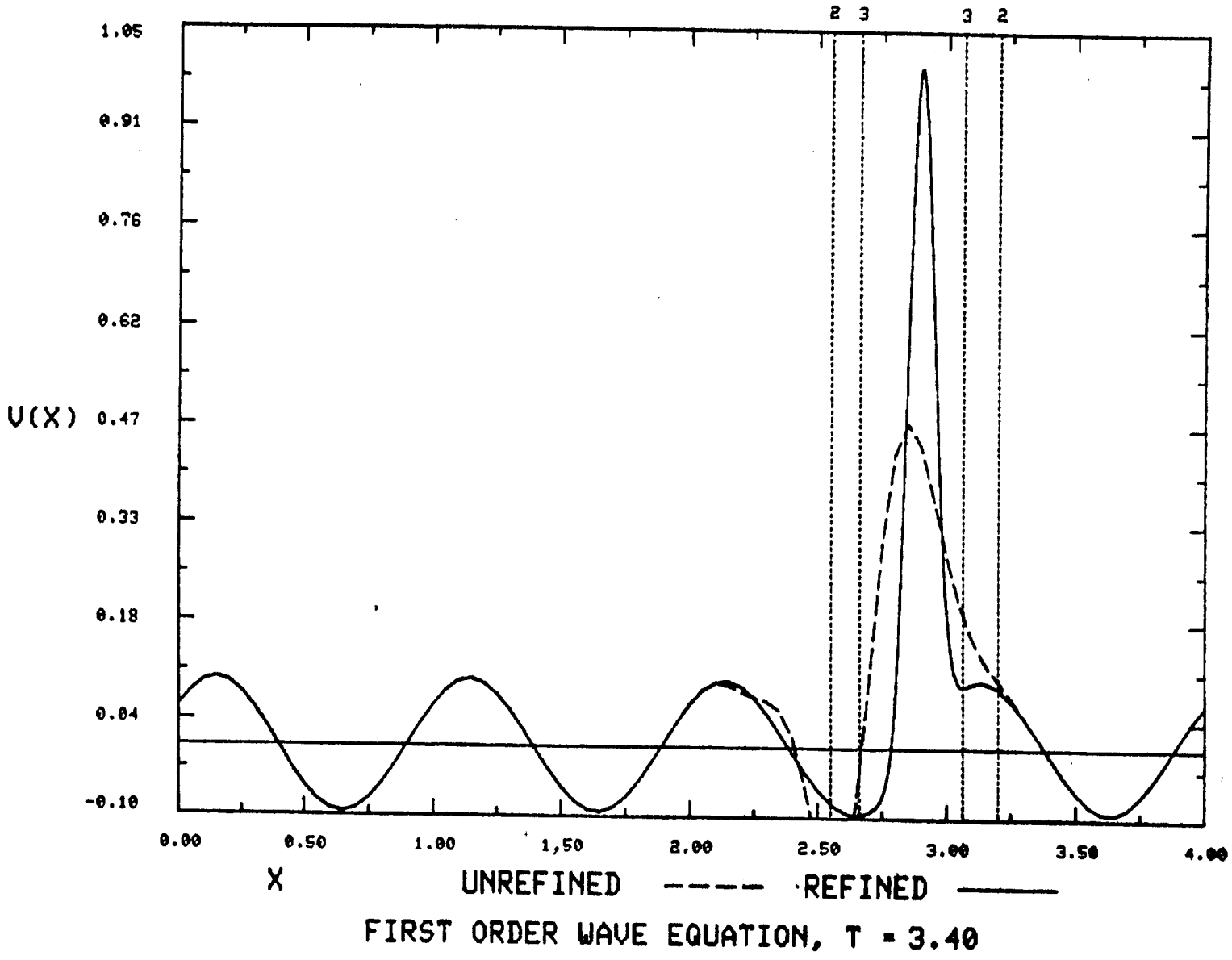
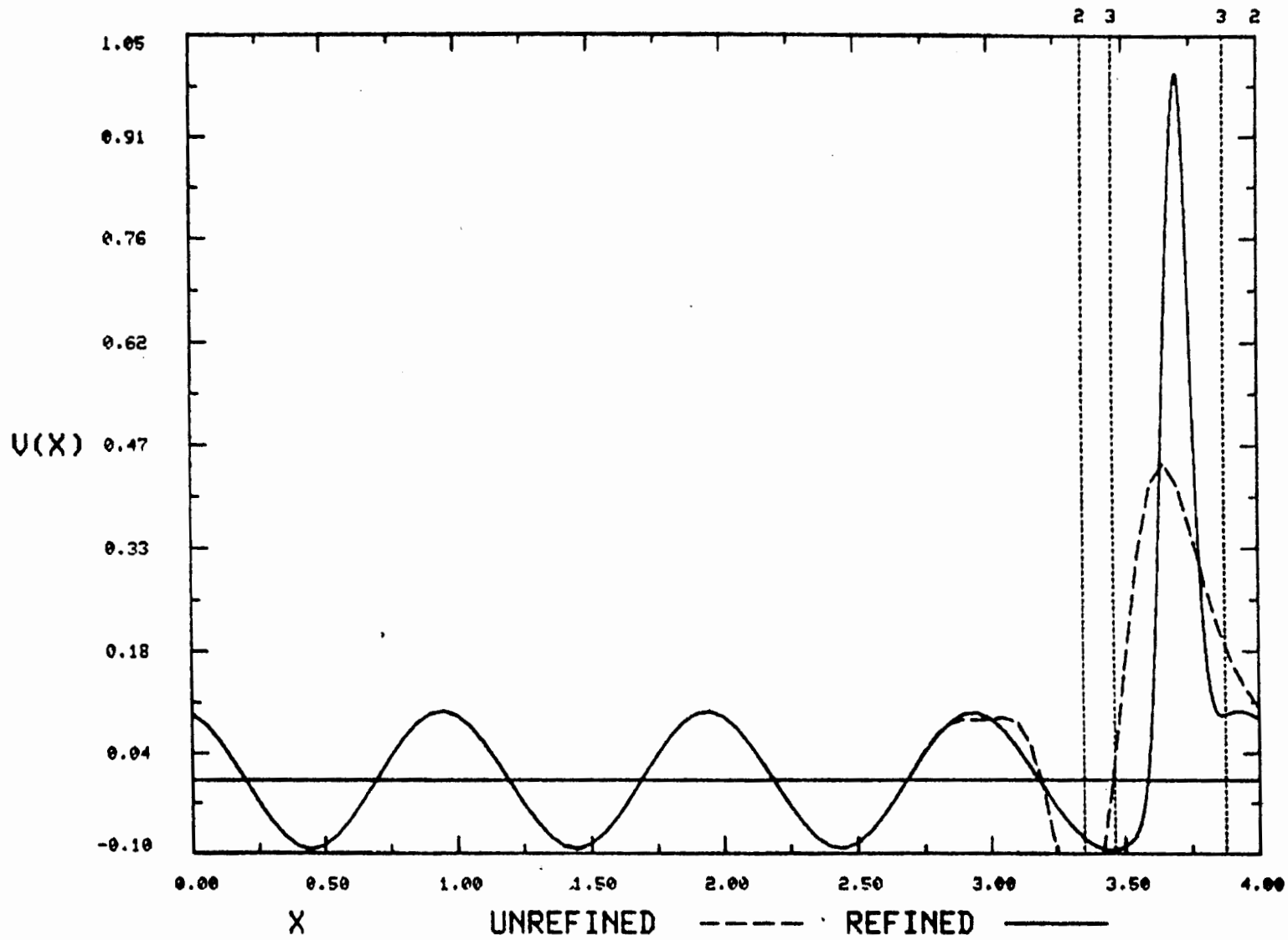

2 3
3 2
1.05
0.91
0.76
0.62
V(X) 0.47
0.33
0.18
0.04
-0.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
FIRST ORDER WAVE EQUATION, T = 3.40

Figure 8.1(e)



FIRST ORDER WAVE EQUATION, T = 4.20

Figure 8.1(f)

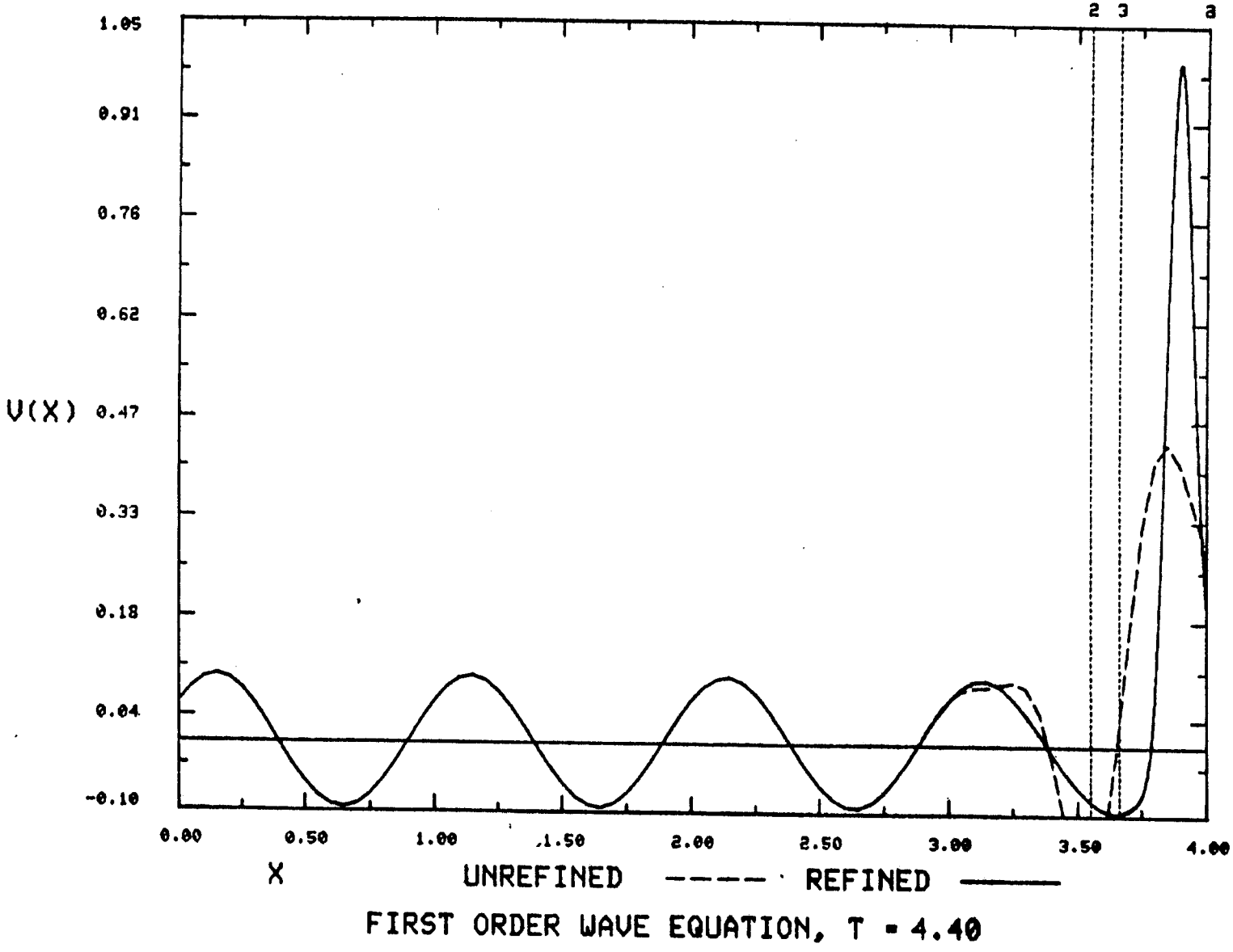

1.05
0.91
0.76
0.62
0.47
0.33
0.18
0.04
-0.10
V(X)
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
2
3
a
UNREFINED
REFINED
FIRST ORDER WAVE EQUATION, T = 4.40

Figure 8.1(g)

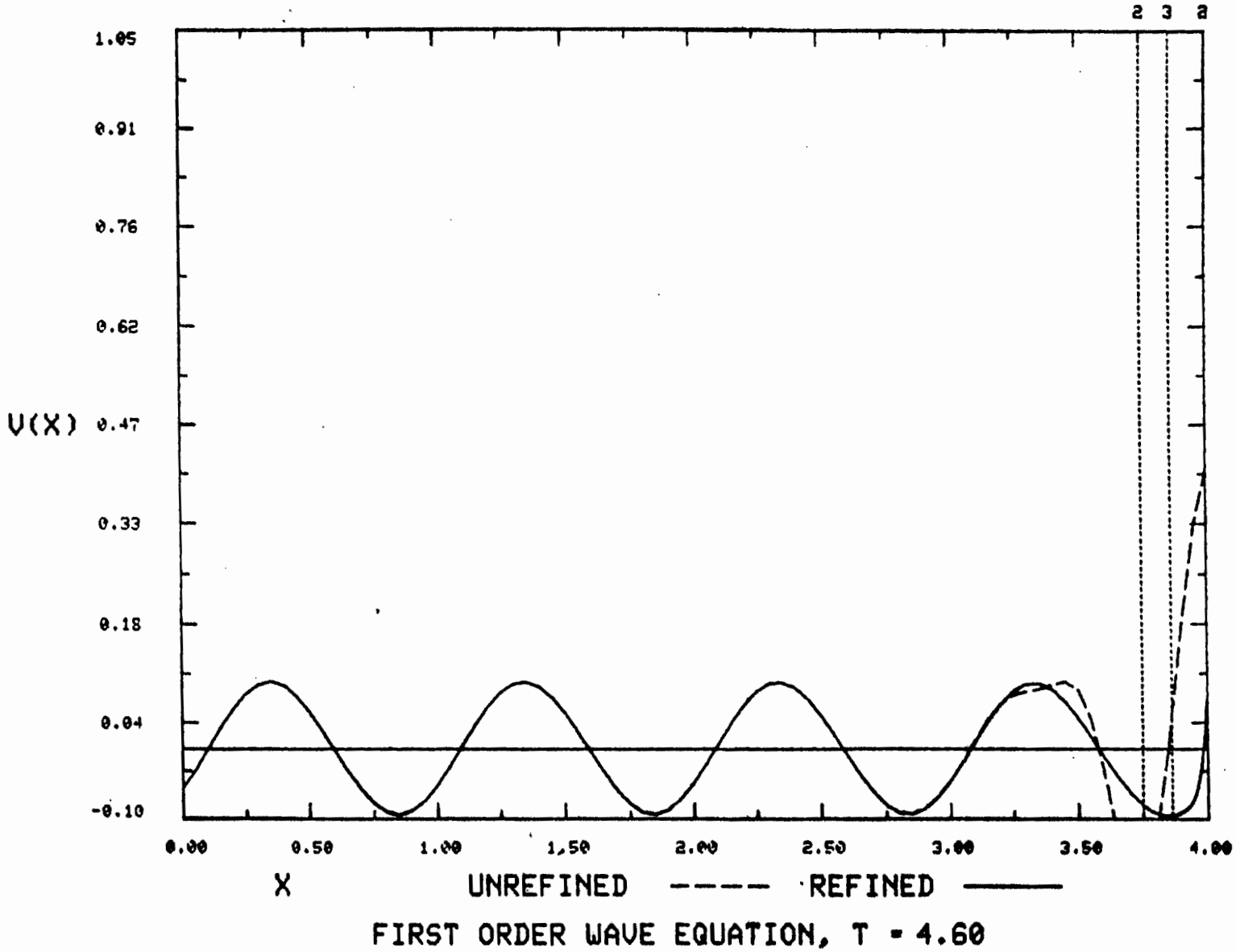

2 3 2
1.05
0.91
0.76
0.62
V(X) 0.47
0.33
0.18
0.04
-0.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
FIRST ORDER WAVE EQUATION, T = 4.60

Figure 8.1(h)

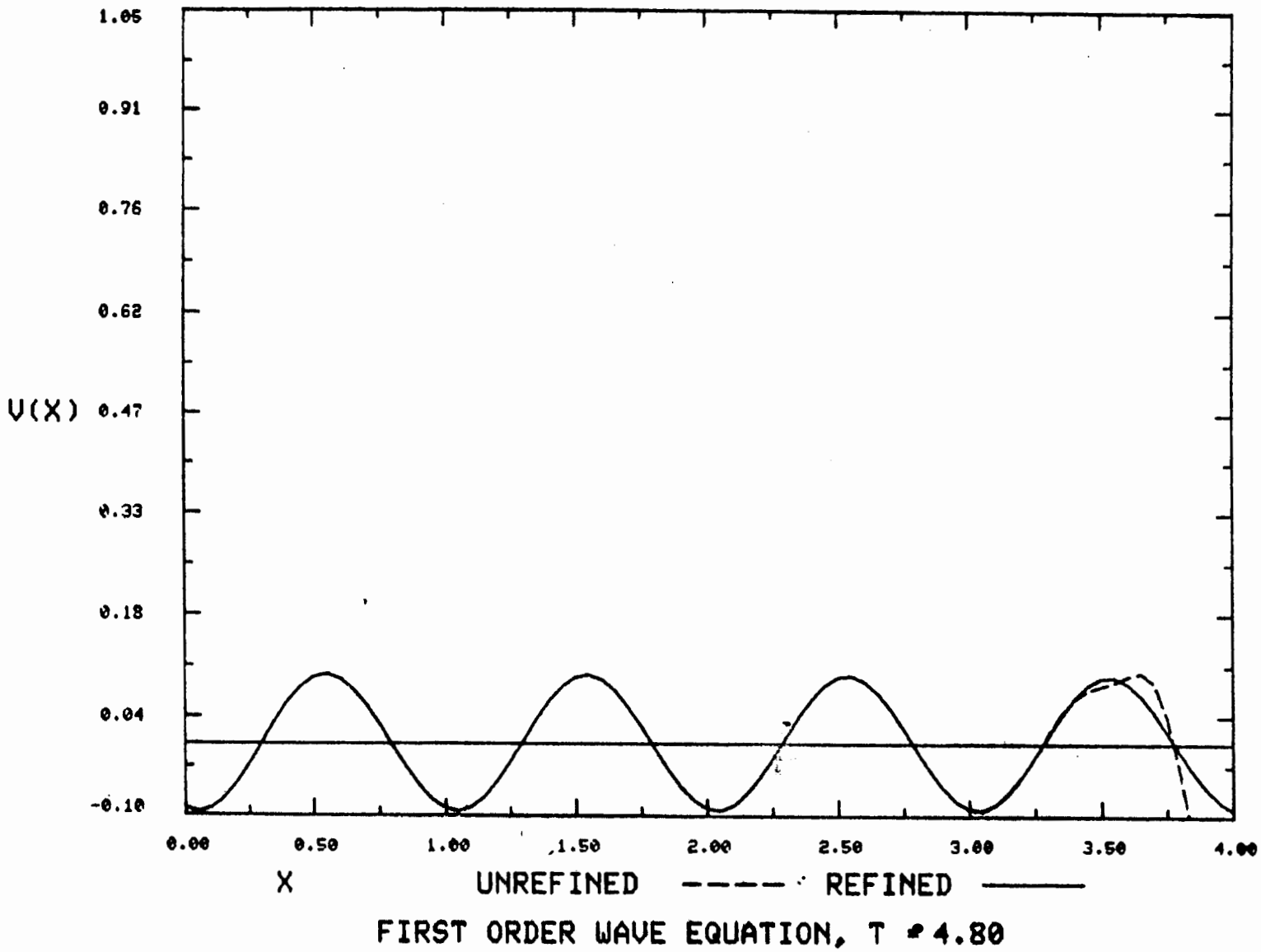

1.05
0.91
0.76
0.62
V(X) 0.47
0.33
0.18
0.04
-0.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
FIRST ORDER WAVE EQUATION, T = 4.80

Figure 8.1(i)

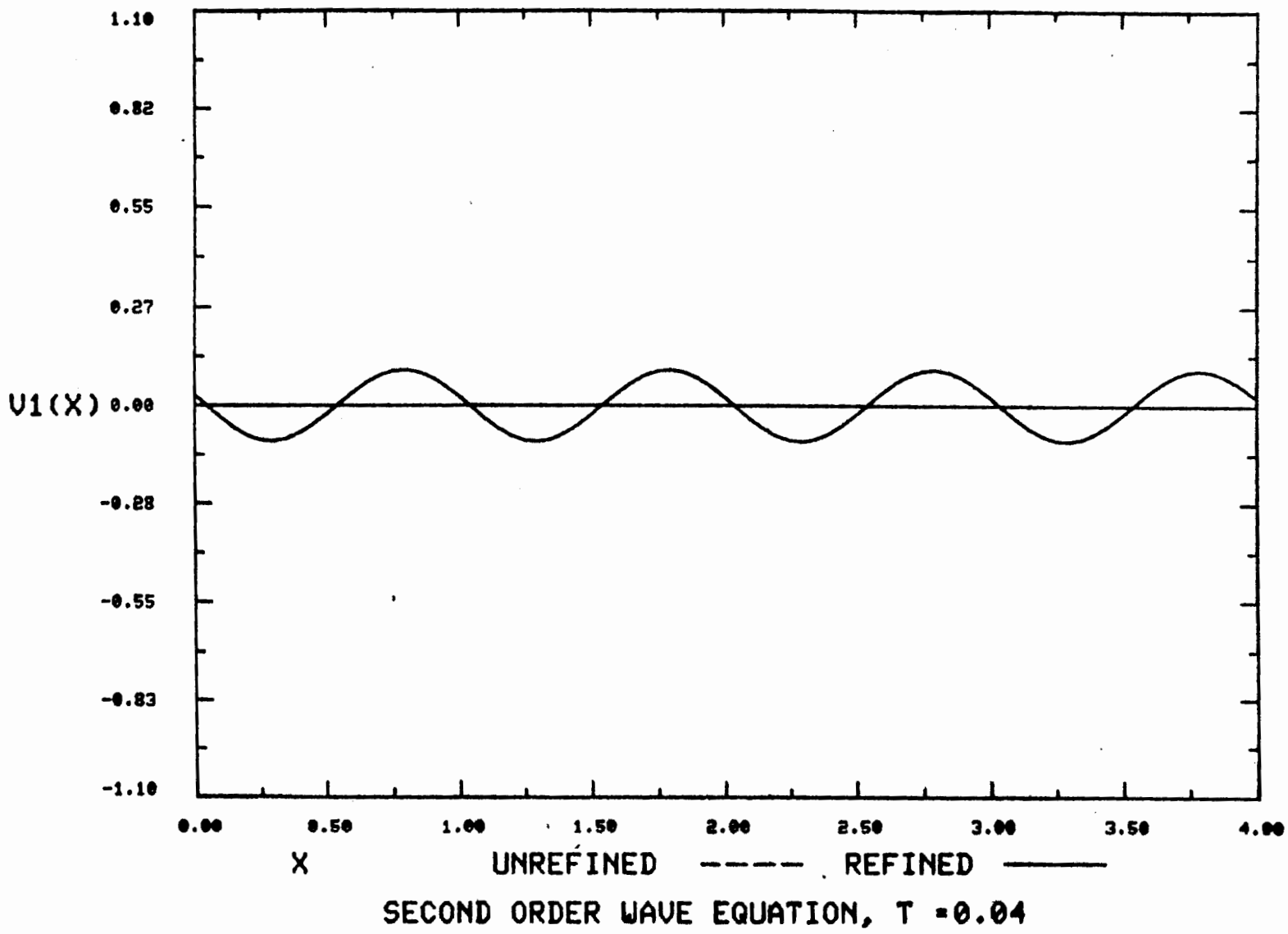

1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =0.04

Figure 8.2(a)

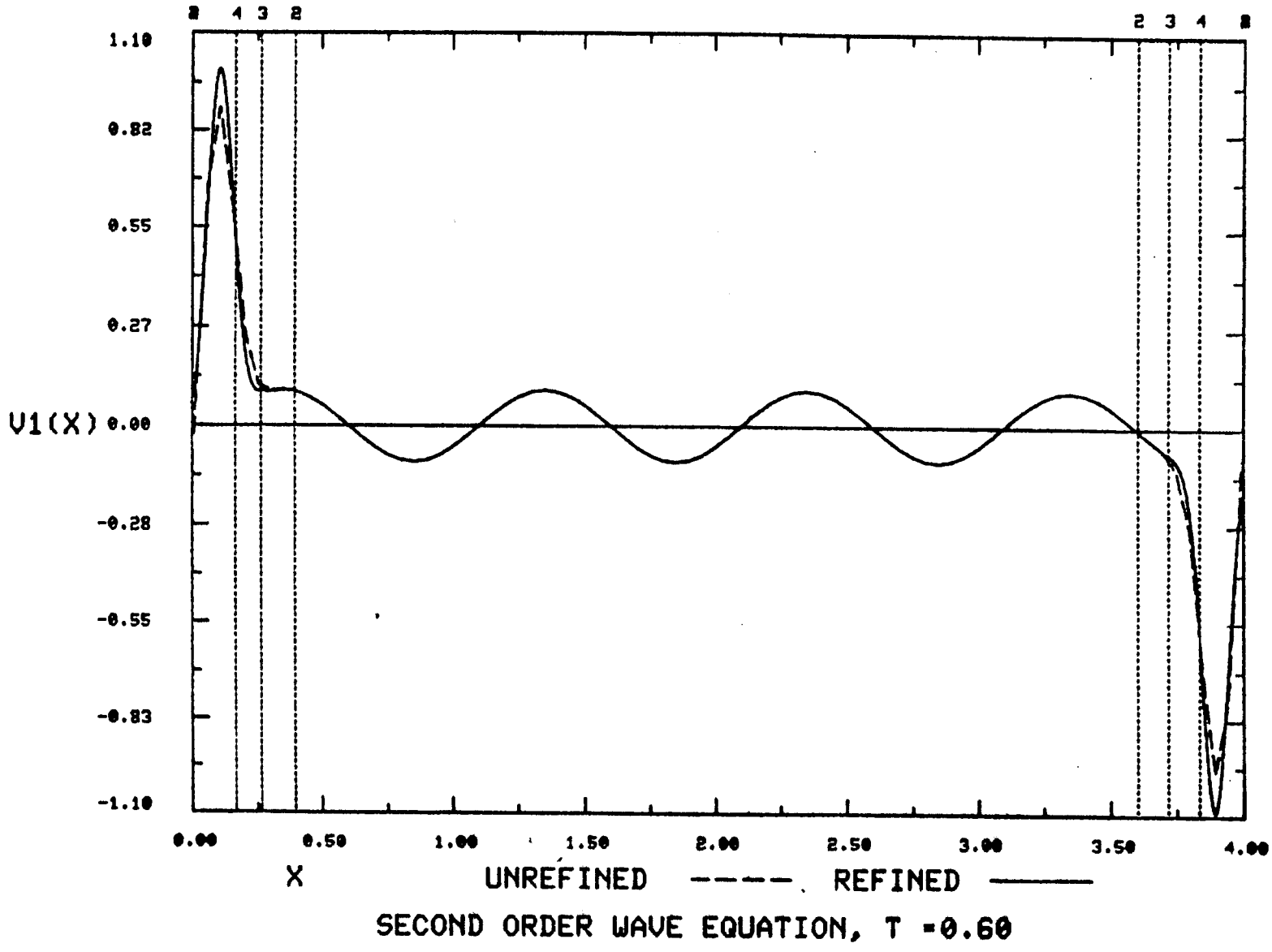

2 4 3 2
2 3 4 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =0.60

Figure 8.2(b)

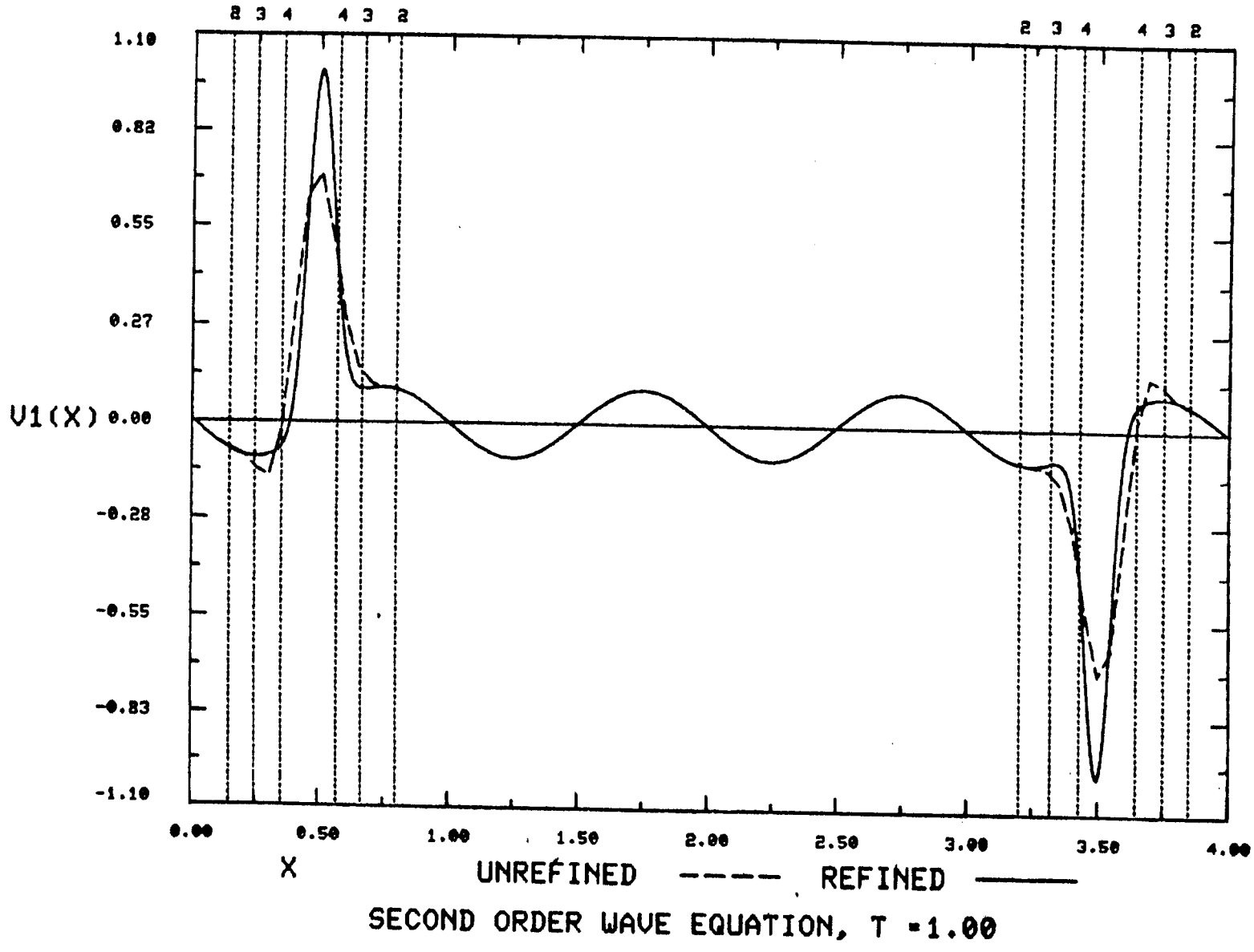

2 3 4 4 3 2
2 3 4 4 3 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =1.00

Figure 8.2(c)

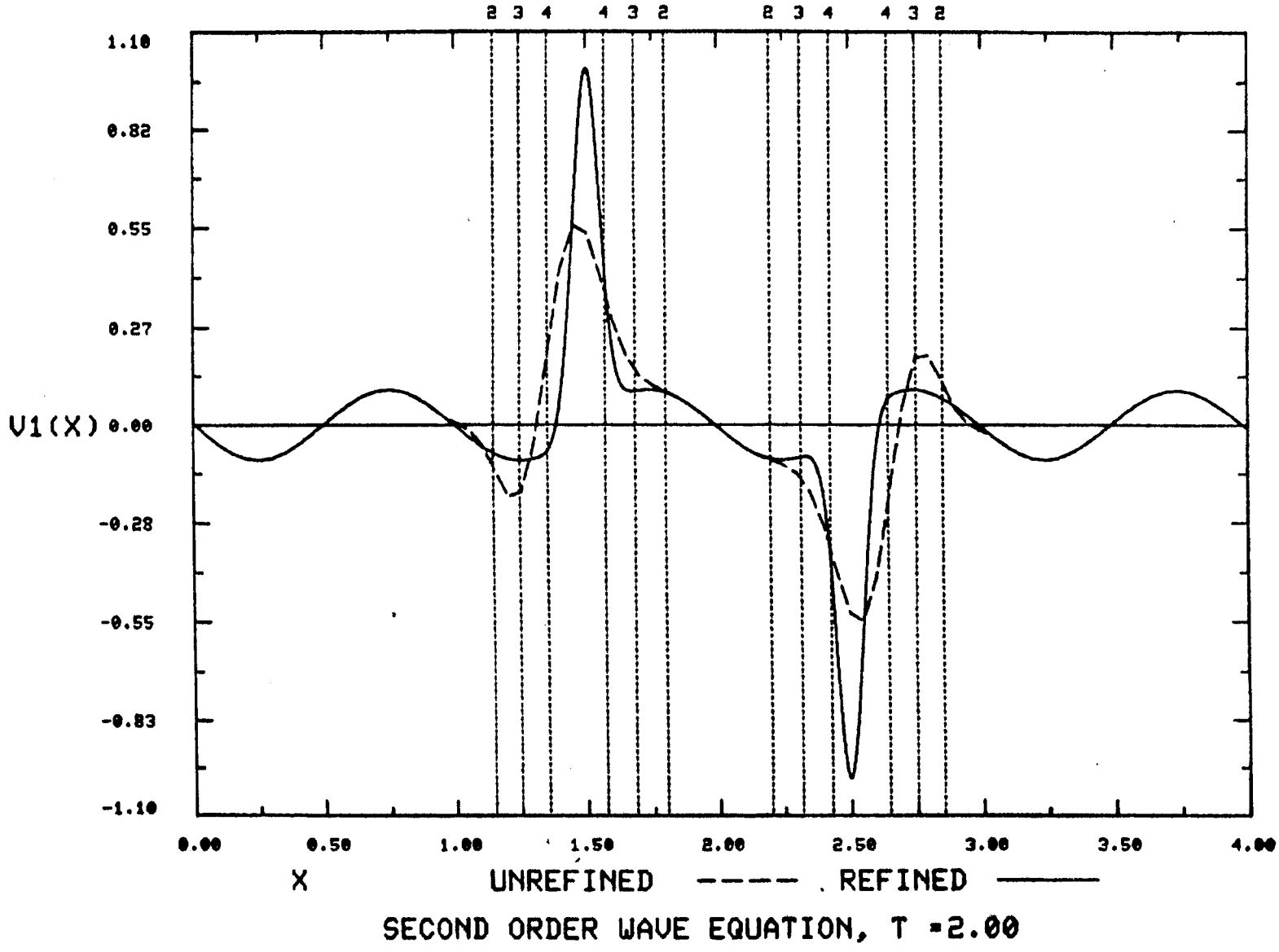

2 3 4 4 3 2 2 3 4 4 3 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00 0.50 1.00 1.50 2.00 2.50 3.00 3.50 4.00
X
UNREFINED ---- REFINED ——
SECOND ORDER WAVE EQUATION, T =2.00

Figure 8.2(d)

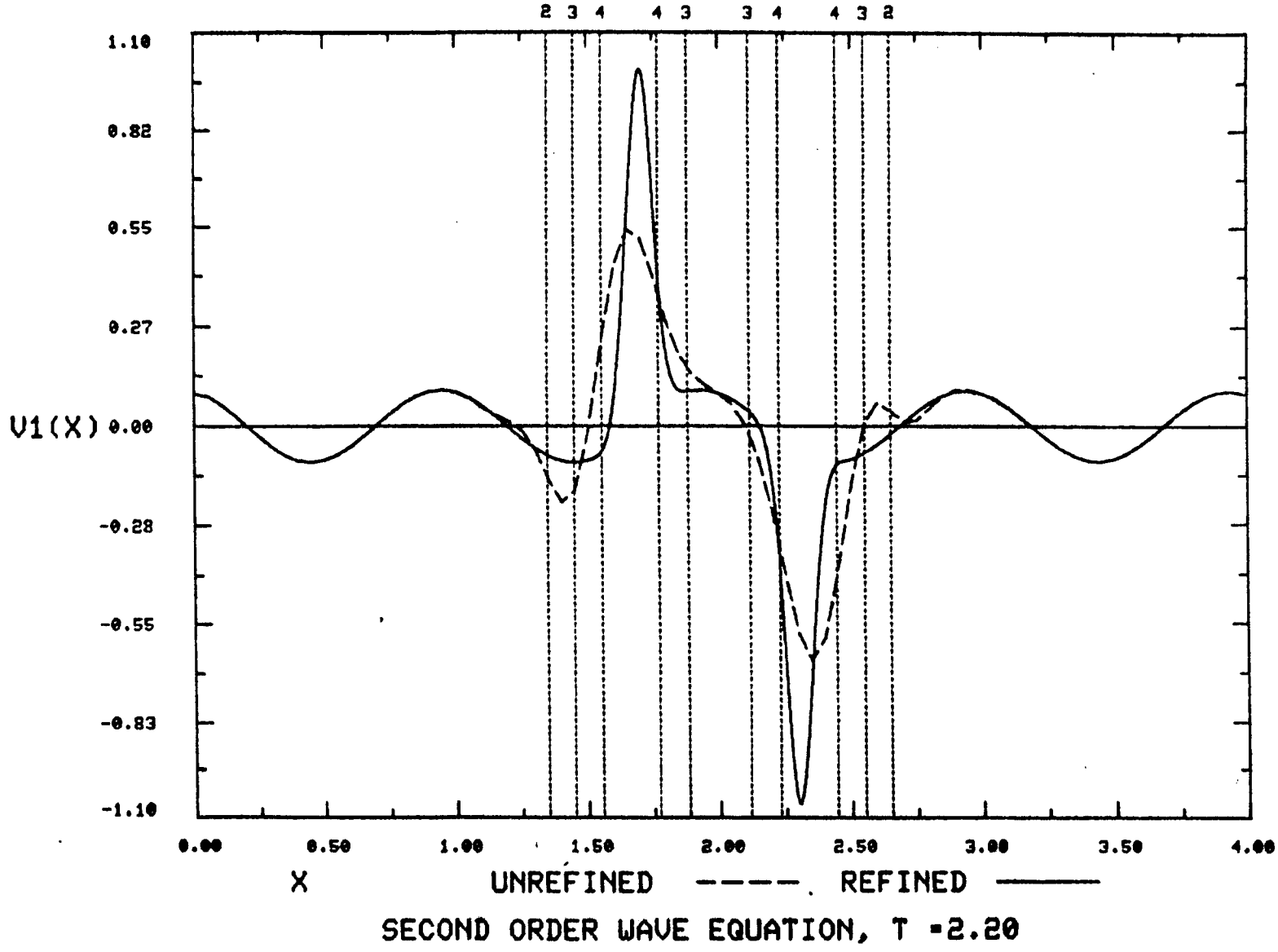

2 3 4 4 3 3 4 4 3 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =2.20

Figure 8.2(e)

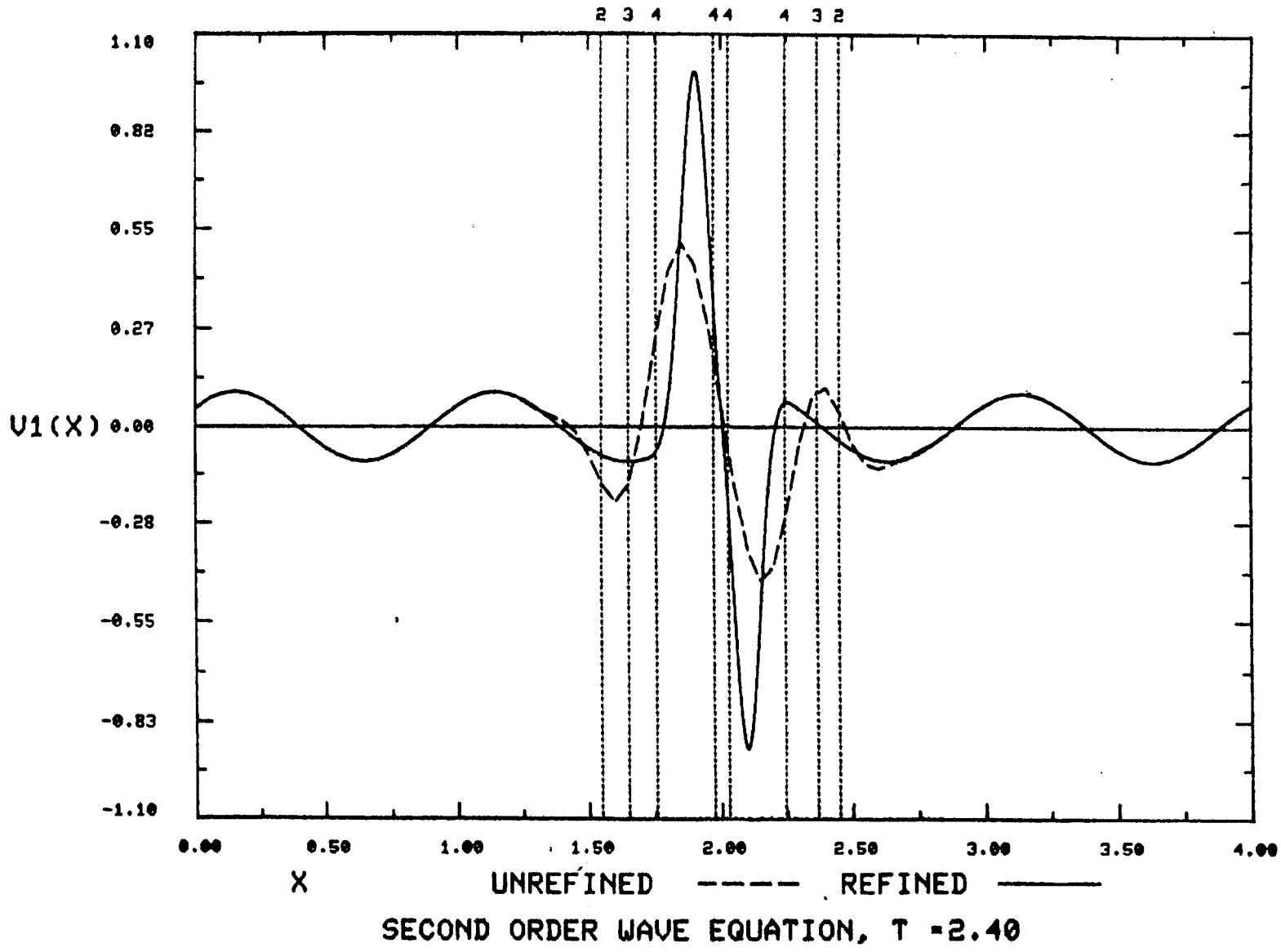

2 3 4 4 4 4 3 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =2.40

Figure 8.2(f)

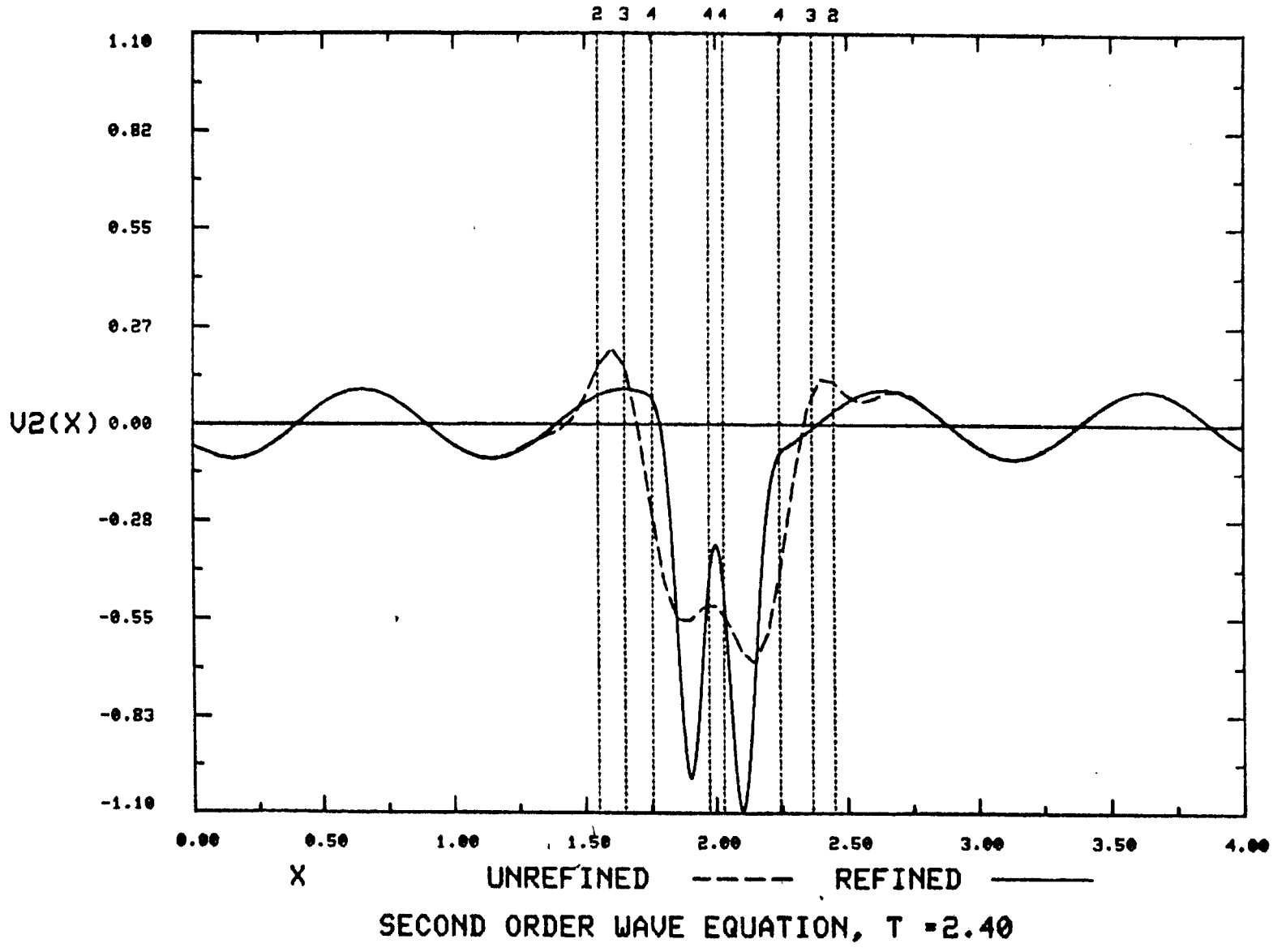

2 3 4 4 4 4 3 2
1.10
0.82
0.55
0.27
V2(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =2.40

Figure 8.2(g)

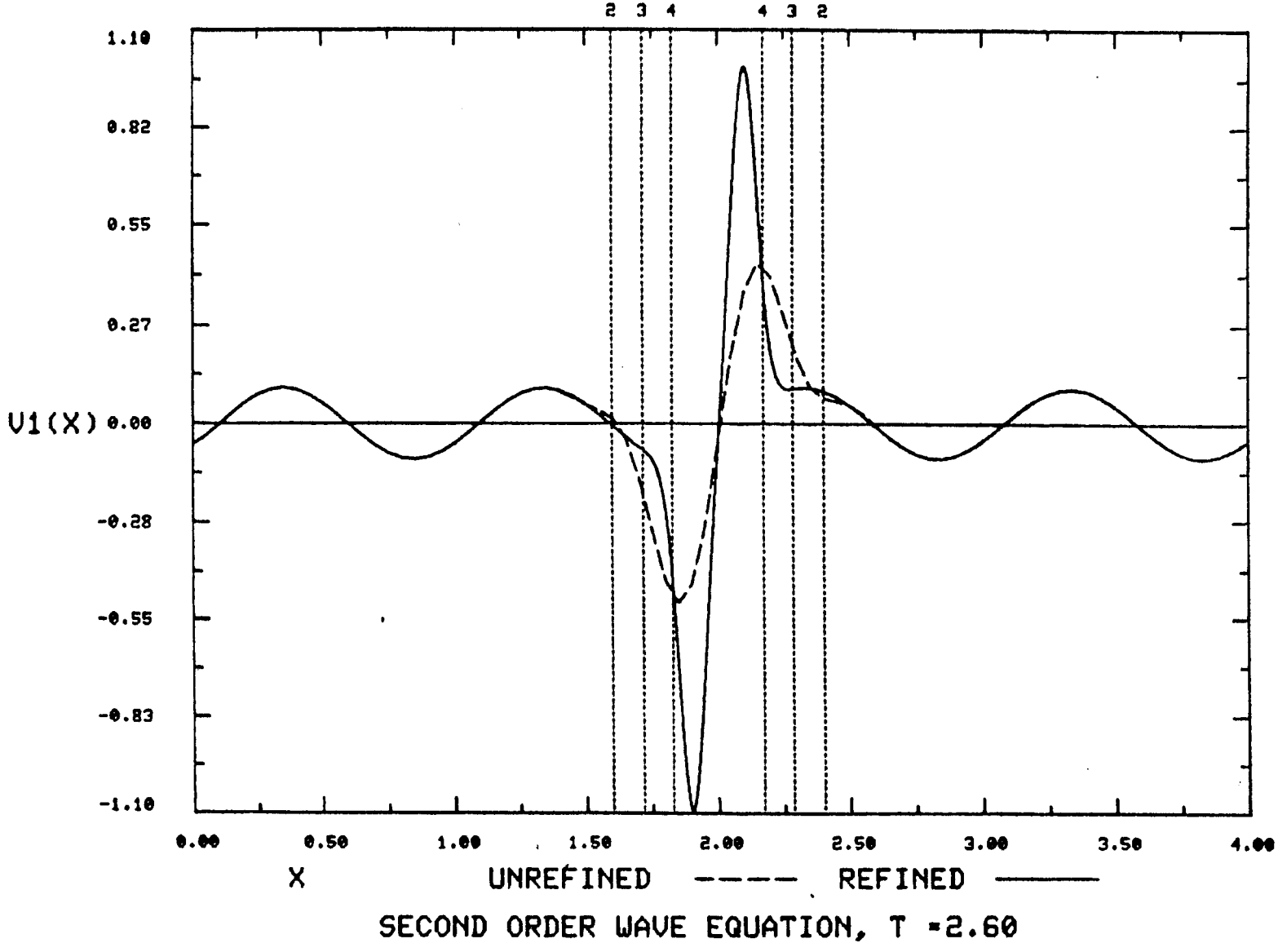

2 3 4 4 3 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00 0.50 1.00 1.50 2.00 2.50 3.00 3.50 4.00
X
UNREFINED ---- REFINED ——
SECOND ORDER WAVE EQUATION, T =2.60

Figure 8.2(h)

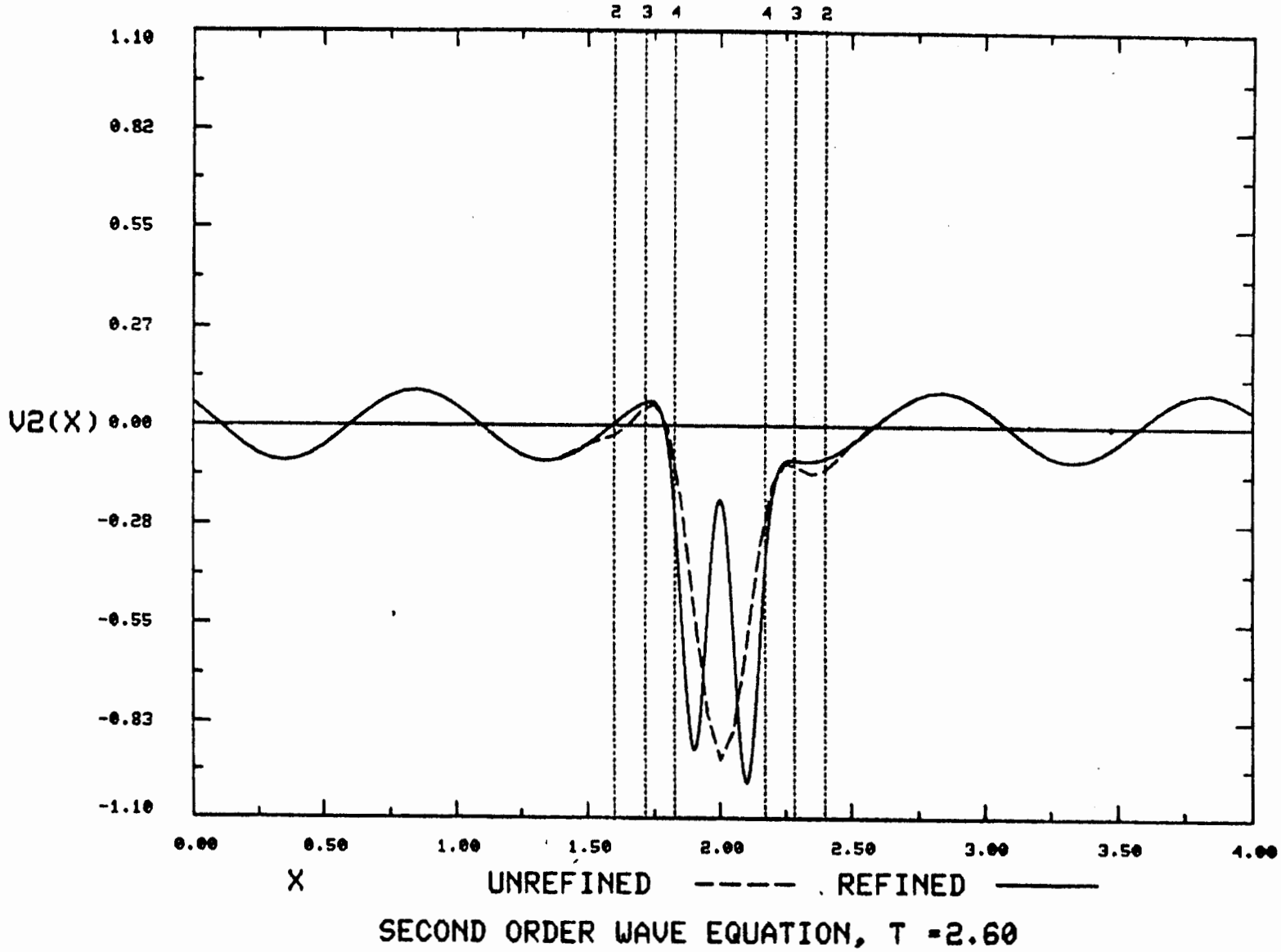

2 3 4 4 3 2
1.10
0.82
0.55
0.27
V2(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =2.60

Figure 8.2(i)

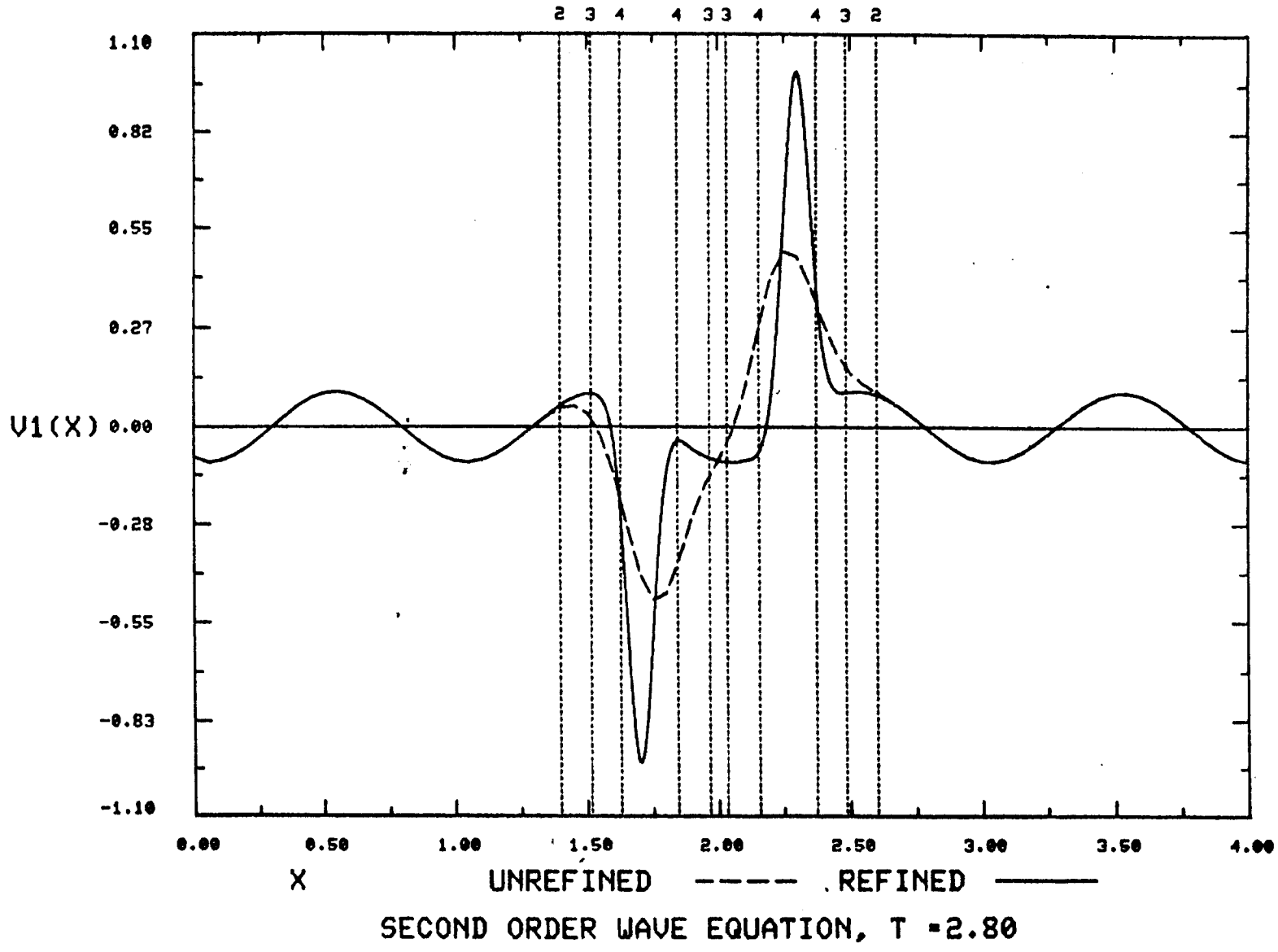

2 3 4 4 3 3 4 4 3 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00 0.50 1.00 1.50 2.00 2.50 3.00 3.50 4.00
X
UNREFINED ---- REFINED ——
SECOND ORDER WAVE EQUATION, T =2.80

Figure 8.2(j)

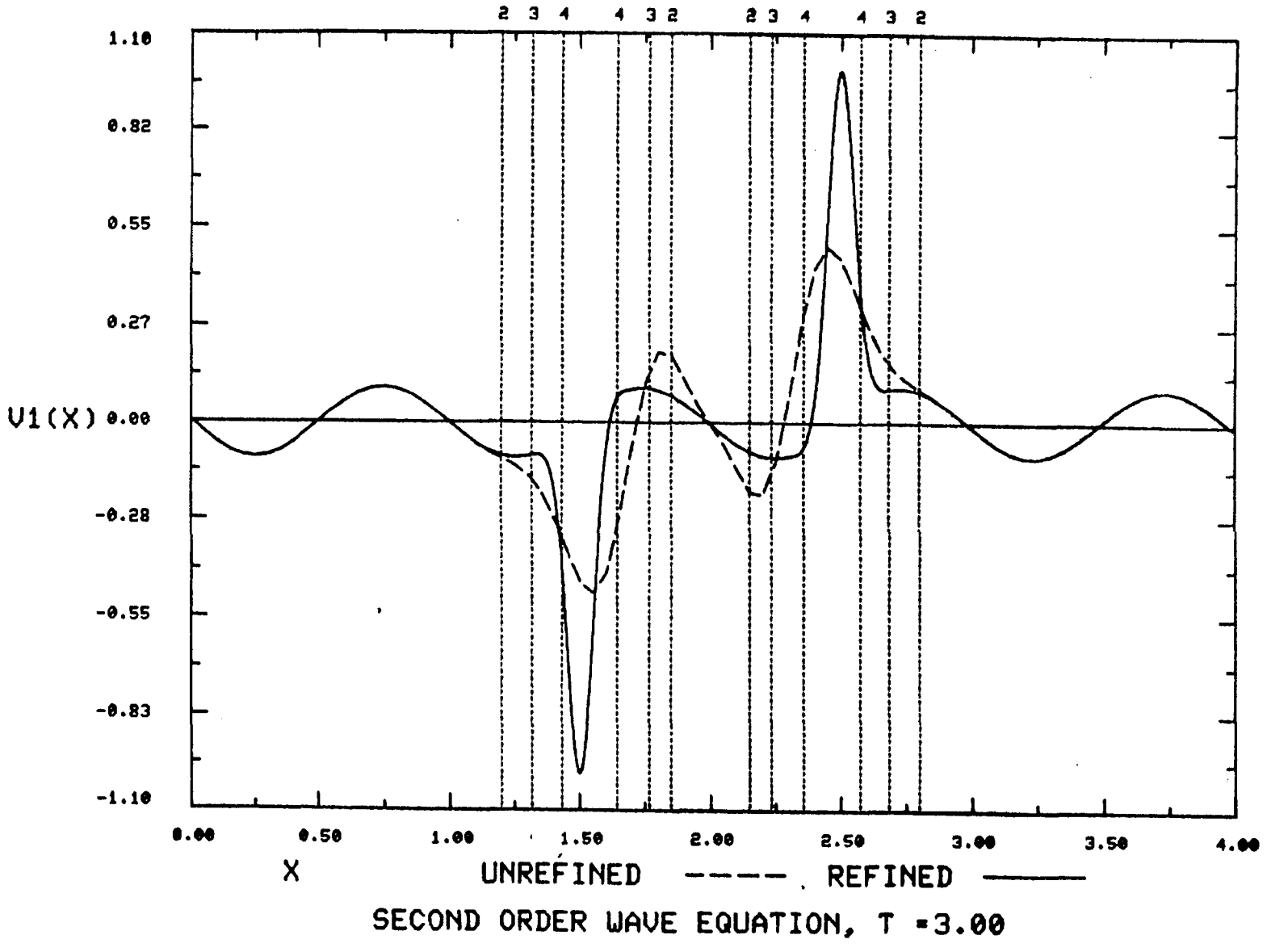

2 3 4 4 3 2 2 3 4 4 3 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =3.00

Figure 8.2(k)

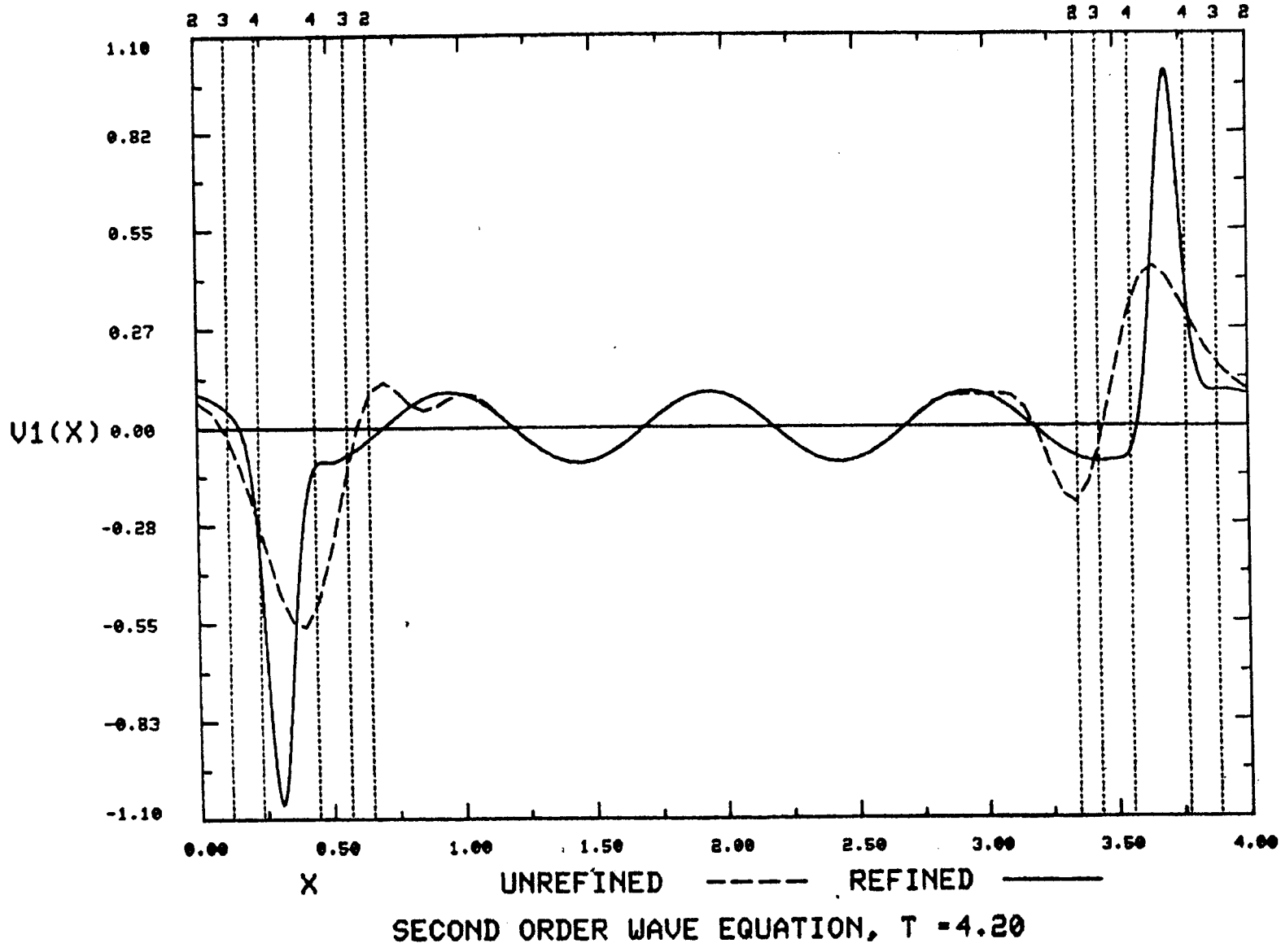

2 3 4 4 3 2
2 3 4 4 3 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =4.20

Figure 8.2(1)

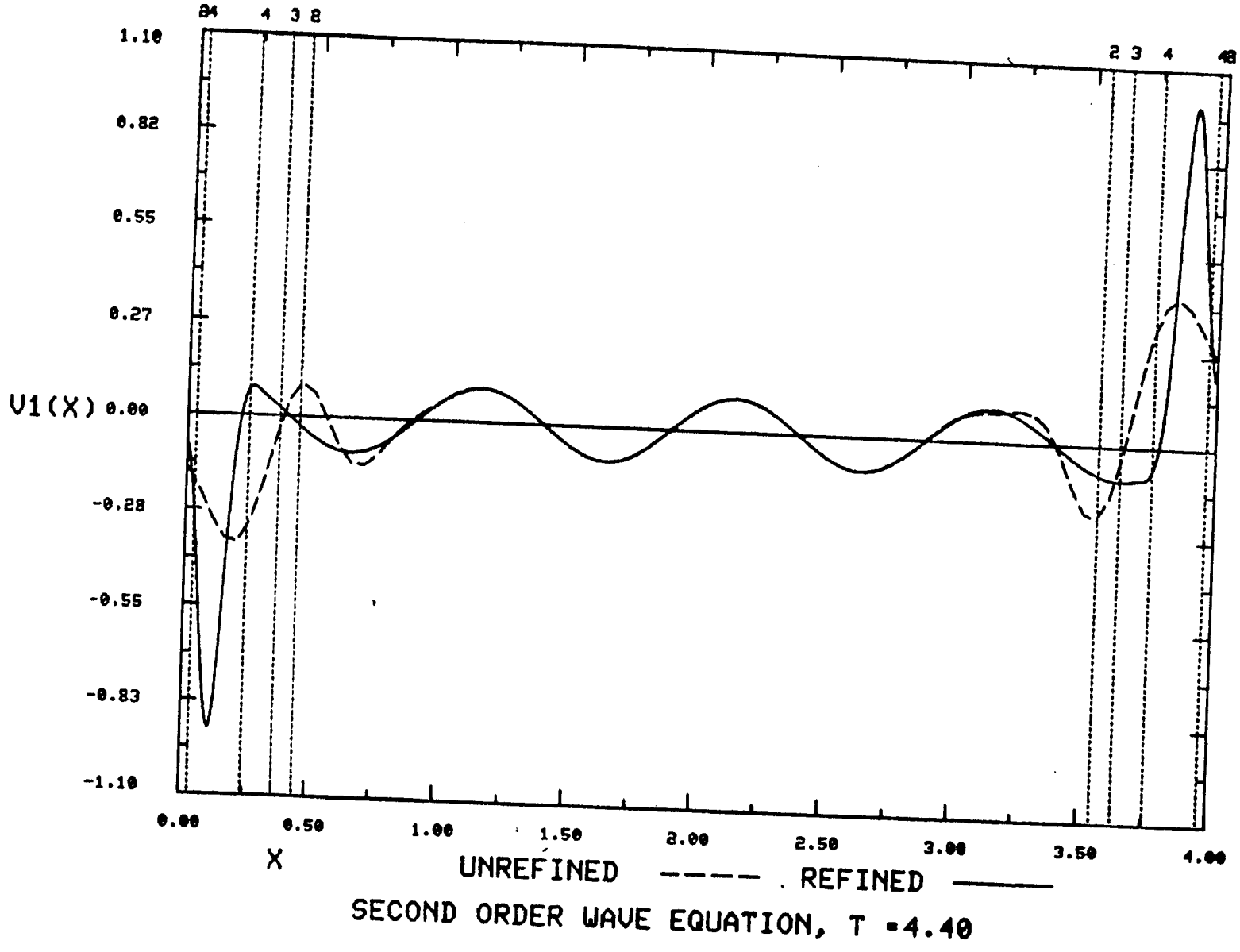

84 4 3 8
2 3 4 48
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED ----- REFINED ———
SECOND ORDER WAVE EQUATION, T =4.40

Figure 8.2(m)

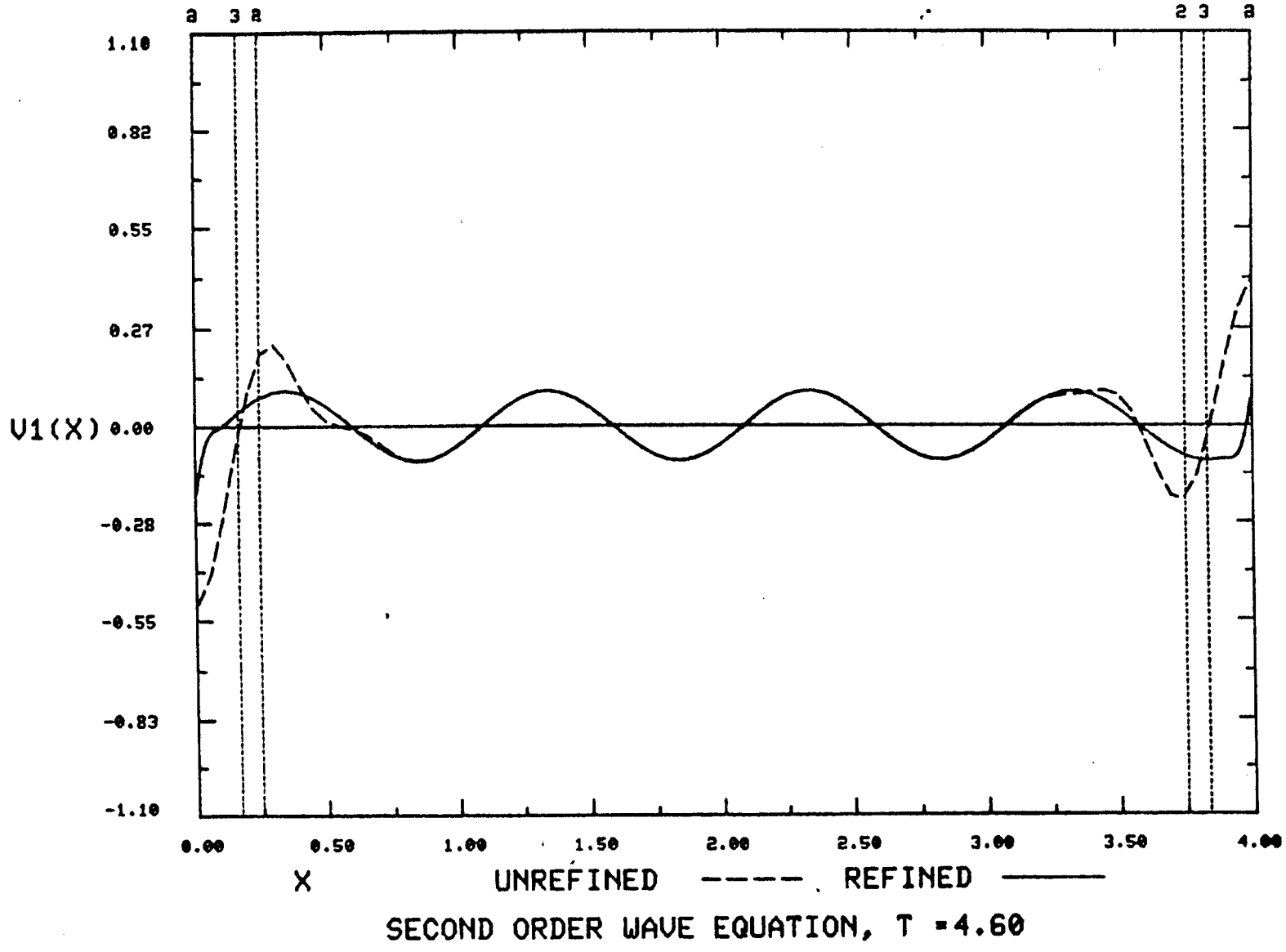

2 3 2
2 3 2
1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =4.60

Figure 8.2(n)

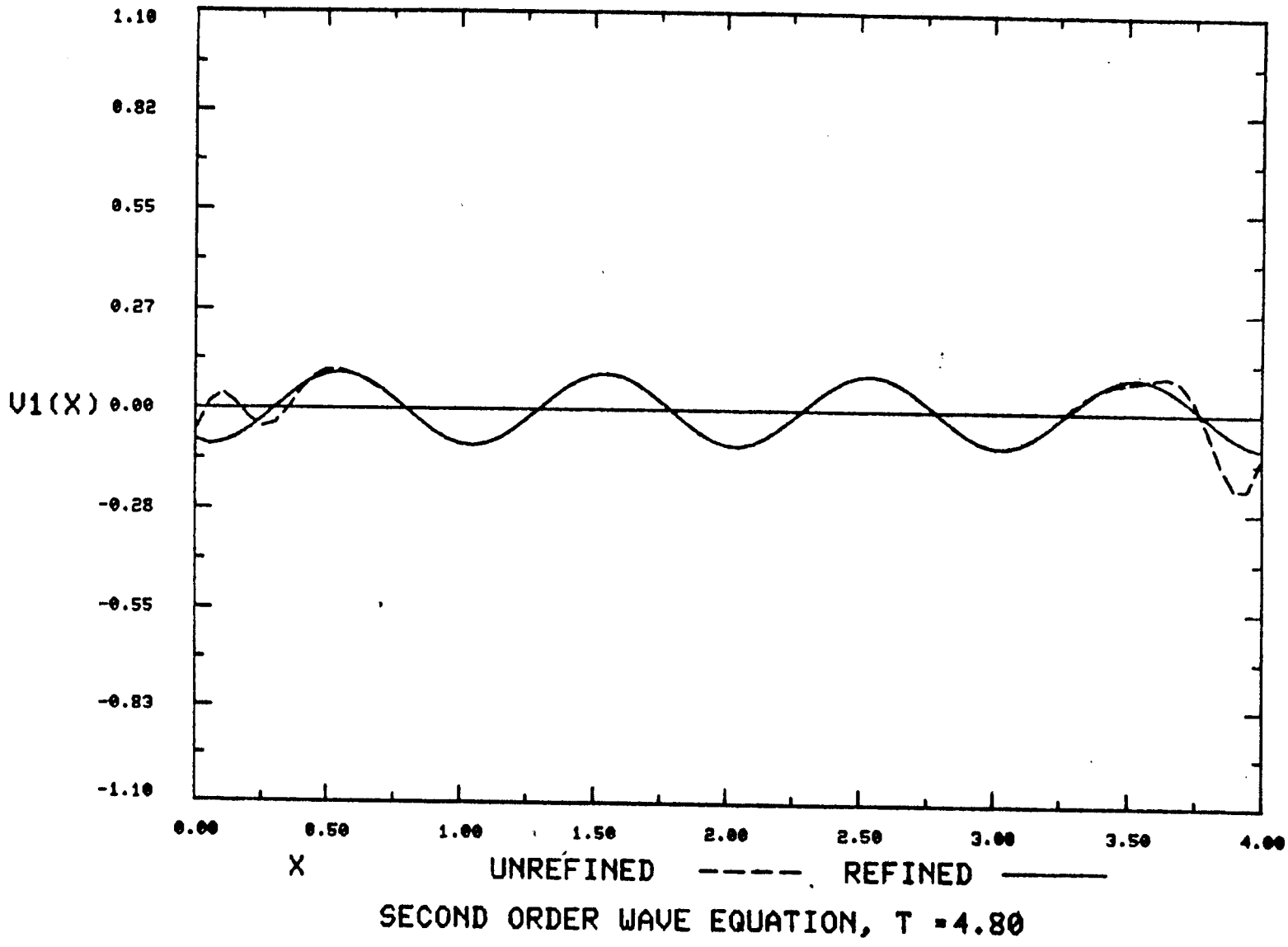

1.10
0.82
0.55
0.27
V1(X) 0.00
-0.28
-0.55
-0.83
-1.10
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
SECOND ORDER WAVE EQUATION, T =4.80

Figure 8.2(o)

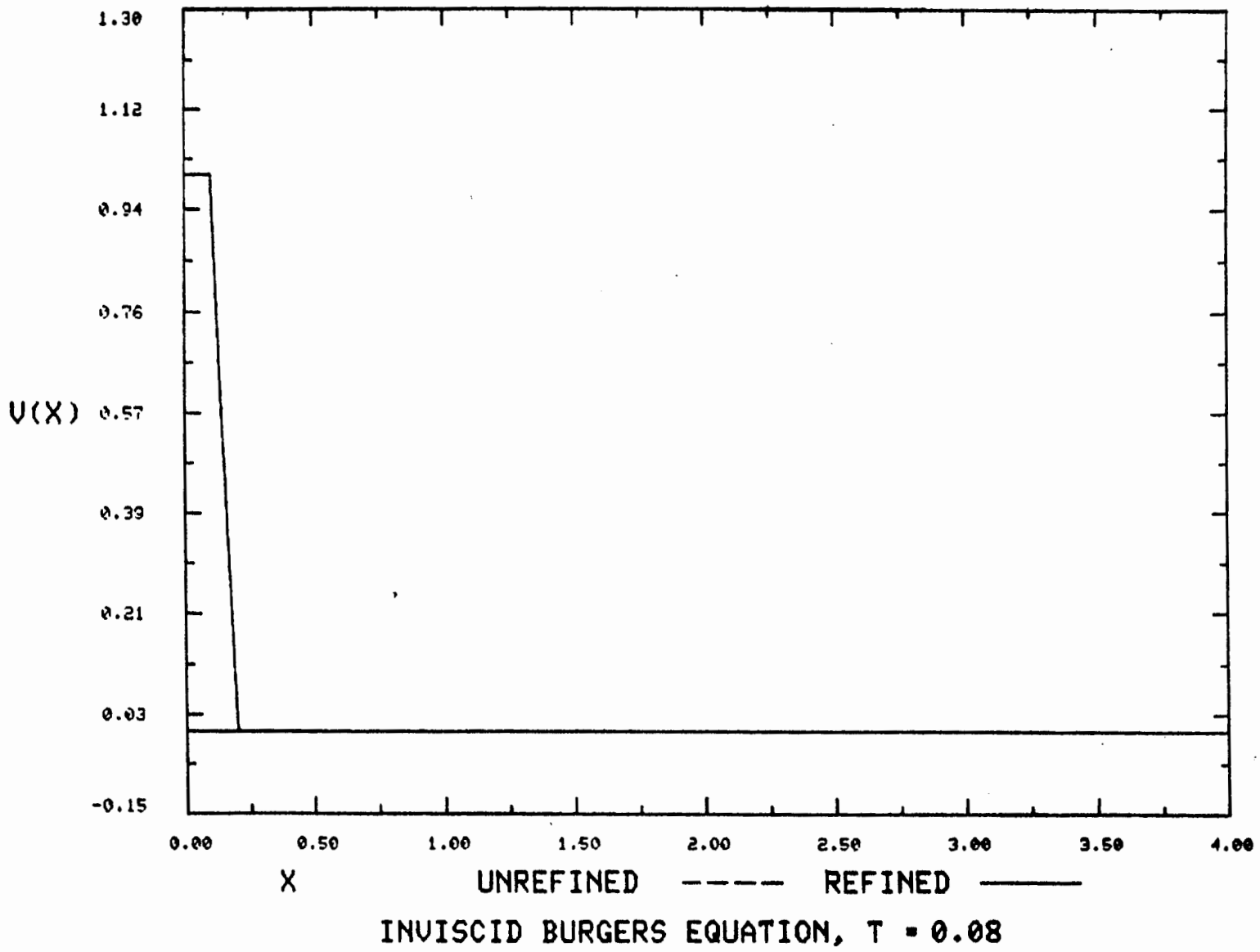

1.30
1.12
0.94
0.76
V(X) 0.57
0.39
0.21
0.03
-0.15
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
INVISCID BURGERS EQUATION, T = 0.08

Figure 8.3(a)

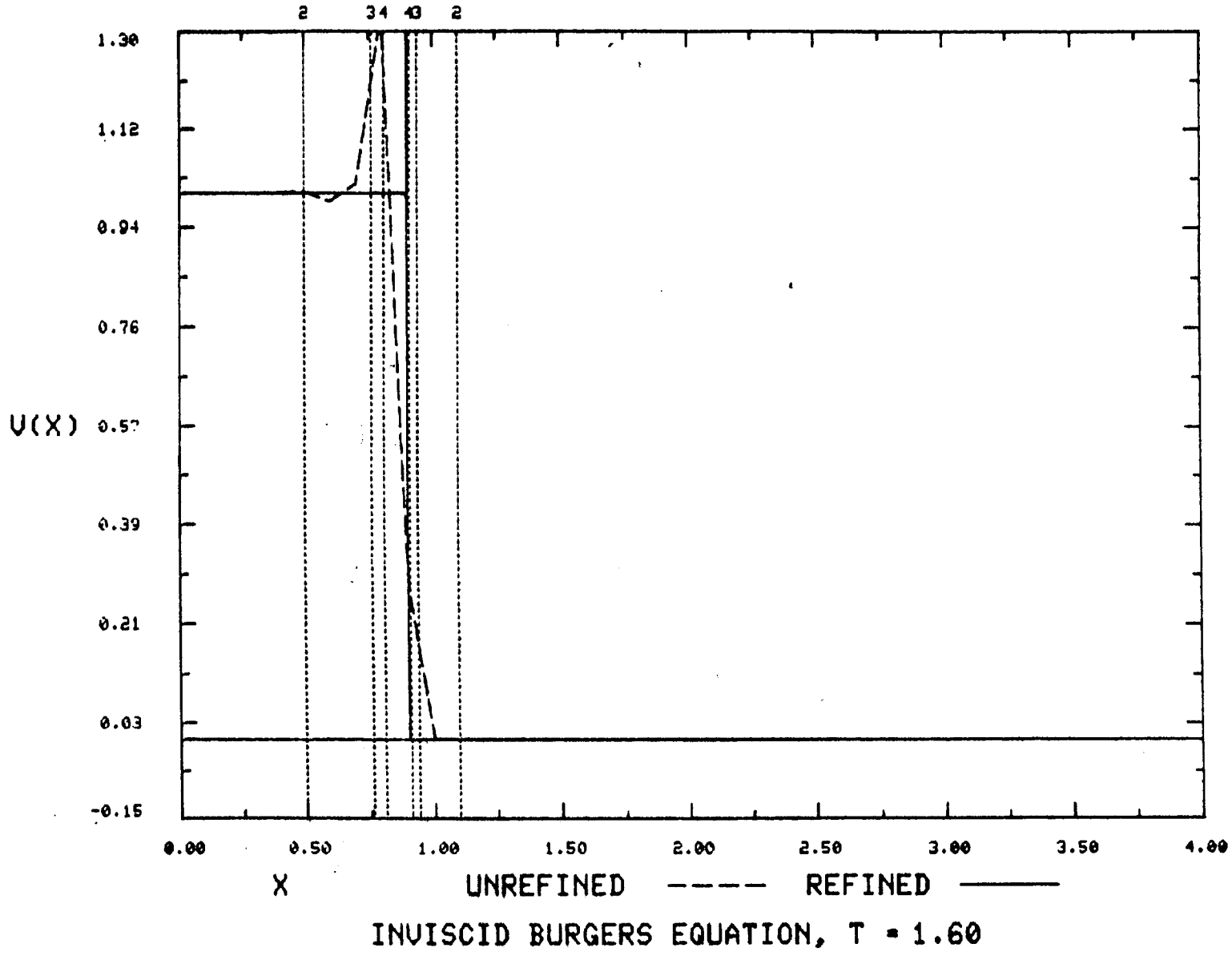

2
34
43
2
1.30
1.12
0.94
0.76
U(X)
0.57
0.39
0.21
0.03
-0.15
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
INVISCID BURGERS EQUATION, T = 1.60

Figure 8.3(b)

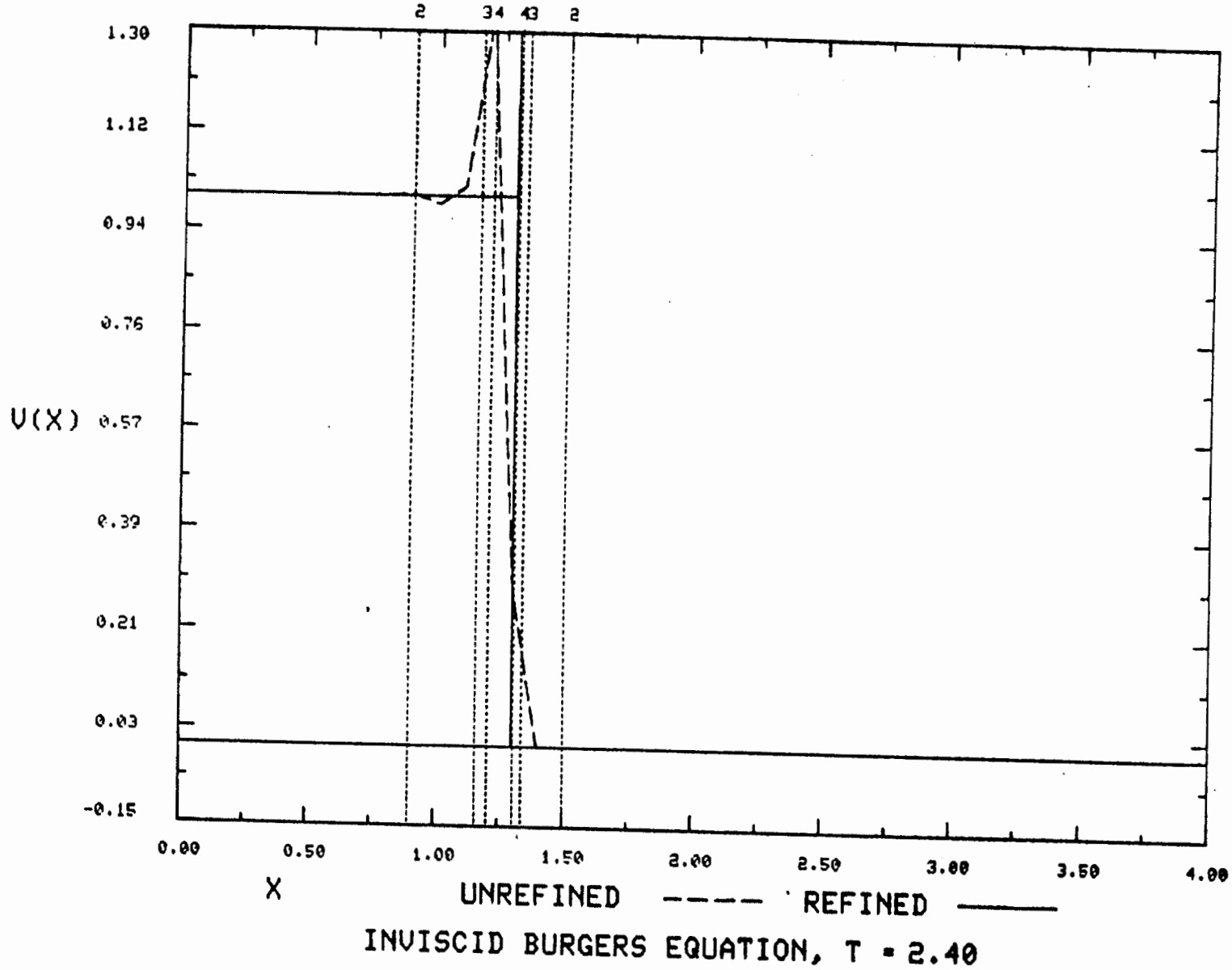

2
34
43
2
1.30
1.12
0.94
0.76
0.57
0.39
0.21
0.03
-0.15
V(X)
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
INVISCID BURGERS EQUATION, T = 2.40

Figure 8.3(c)

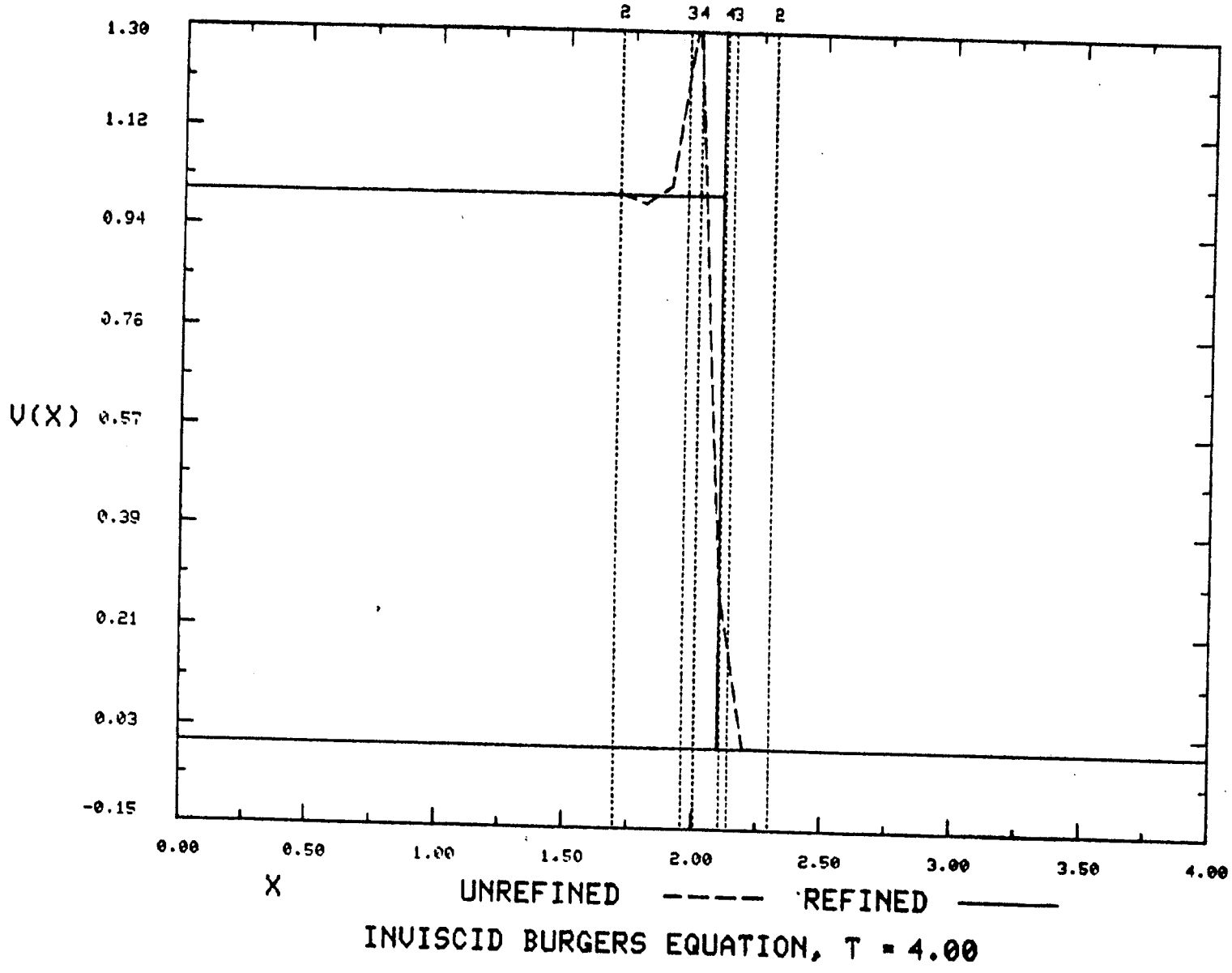

2
34
43
2
1.30
1.12
0.94
0.76
V(X)
0.57
0.39
0.21
0.03
-0.15
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
UNREFINED
REFINED
INVISCID BURGERS EQUATION, T = 4.00

Figure 8.3(d)

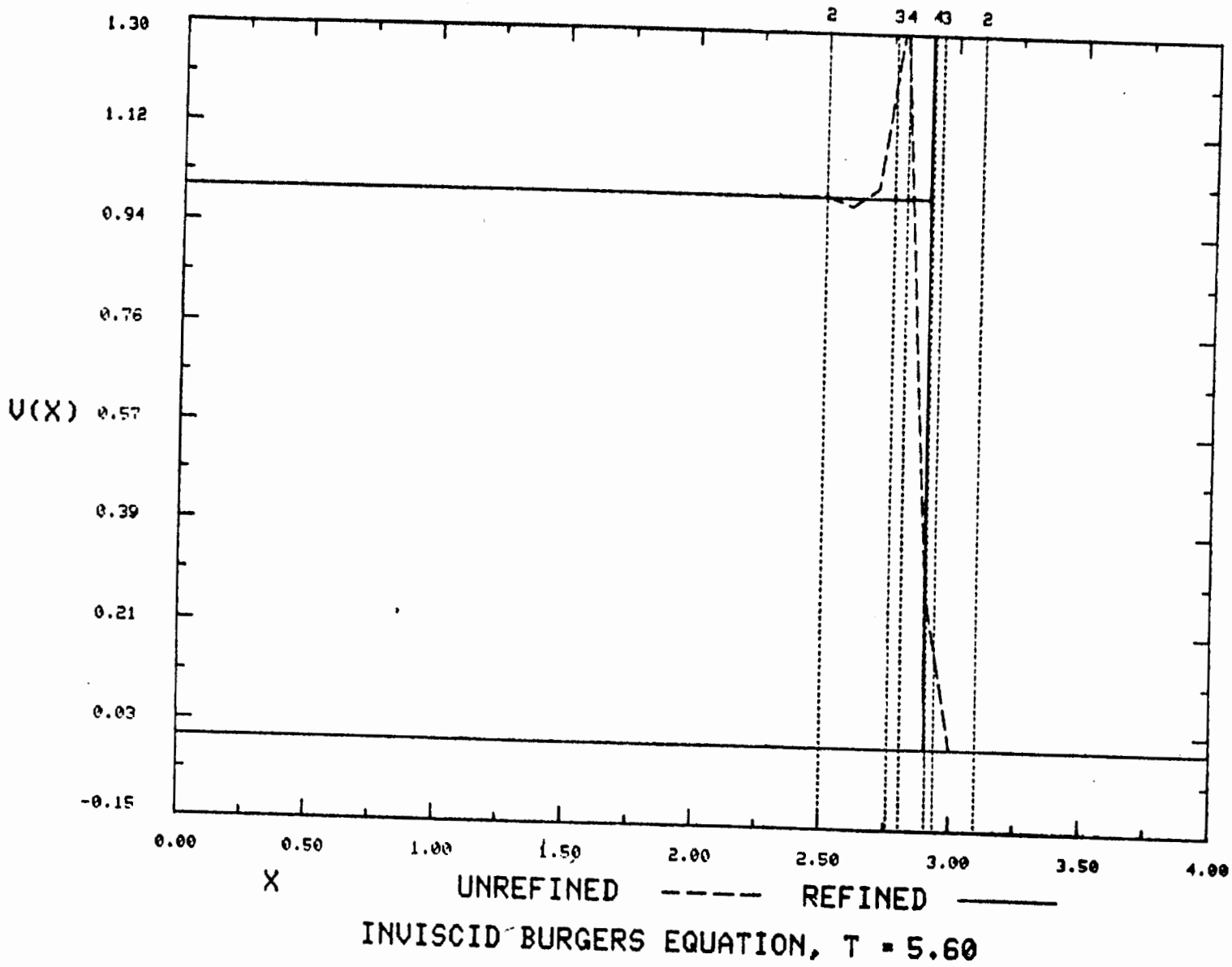

1.30
1.12
0.94
0.76
V(X) 0.57
0.39
0.21
0.03
-0.15
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
X
2
34
43
2
UNREFINED ---- REFINED ——
INVISCID BURGERS EQUATION, T = 5.60

Figure 8.3(e)

refinement ratios $N = M = 4$ and maximum number of refinement levels = 4. (Only three levels were actually used in the calculation.) The local error tolerance was 0.01, $\lambda_1 = 0.8$, and the wave speed $c = 1$. Each graph plots the approximate solution $v(x)$ versus x (at a fixed time) as a solid line, together with a separate calculation done with no refinement (maximum refinement level = 1), shown as a dotted line. Since the maximum error in this calculation was about 2 percent, the refined solution may be taken as the exact solution on the graph.

In Figure 8.1(a) we see that the solution consists solely of the sinusoidal background--the pulse has not yet entered. (This graph is taken at $t = 0.04$ rather than $t = 0$ since we use the exact solution at $t = \Delta t = 0.04$ so we can compare with the hybrid method later. This is true in all calculations.) In Figure 8.1(b) the pulse has started to enter the left boundary, and refinements have been created at the second and third levels. (The small numbers at the top are the level numbers of the refinements. Both refinements abut the left boundary.) In Figure 8.1(c) the pulse has fully entered and the refinements have moved right.

Figures 8.1(d) and (e) show the pulse moving across the region, and the refinements following it. Note that the unrefined solution has become a very poor approximation to the pulse--the unrefined peak has only half the height it should. Note also that behind the pulse, the unrefined solution has a large undershoot.

In Figure 8.1(f) the pulse has neared the right boundary, and the refinements now abut this boundary. In Figure 8.1(g) the pulse is leaving the region--this again shows the time-dependent boundaries at work. In Figure 8.1(h) the pulse has left the region, but the unrefined solution has developed a phase error. Finally, Figure 8.1(i) shows that the second and third level refinements have been deleted. Only the background sinusoid is present, but

there is still some "backwash" left in the unrefined solution.

These calculations show that the method is not fooled by background oscillations, and that the refinements do follow the steep gradients. Furthermore, this problem and the next one show that our method is able to adapt to time-dependent boundary conditions.

Figures 8.2(a) through 8.2(o) show our algorithm applied to the second-order wave equation with counter-streaming pulses. This conclusively answers questions 1 to 3, and shows that our method works on a system of differential equations. It also shows the necessity for good data structures, since there are from one to seven refinements, and, as we shall see, they interact in complicated ways. In this calculation we again used 81 mesh points in the coarse mesh, with refinement ratios $N = M = 3$ and maximum number of levels = 5. (Only four levels were actually used.) The local error tolerance was again 0.01, $\lambda_1 = 0.8$, and the wave speed $c = 1$. The same conventions about unrefined and refined solutions, and the labeling of refinements are used. We usually show the first component $v_1(x,t)$ versus x, but in a few instances we show $v_2(x,t)$ versus x (all at a fixed time).

Note that the graphs are plotted at the end of a time step, before the refinements are adjusted in preparation for the next time step. Hence the pulses will appear at one side of the fourth level refinements.

In Figure 8.2(a) the pulses have not yet entered the region and we see only the sinusoidal background. In Figure 8.2(b) both pulses have entered and refinements on levels 2, 3 and 4 abut both boundaries. In Figure 8.2(c) both pulses have left the boundaries and are moving towards each other. Already we can see that the peak of the unrefined solution has decayed substantially.

In Figure 8.2(d) the second-level refinements are about to merge, and in Figure 8.2(e) they have merged; however, the third and fourth level refinements

have not yet merged. In Figure 8.2(f) the third level refinements have merged. Figure 8.2(g) shows the second component of the solution at the same time. In Figure 8.2(h) the fourth level refinements have merged. Figure 8.2(i) shows the second component of the solution at the same time.

In Figure 8.2(j) the pulses have crossed, and the third and fourth level refinements have separated. (Note that the pulses cross, but the refinements do not.) In Figure 8.2(k) the second level refinements have separated as well. Note the degradation in the unrefined solution at this point. Figure 8.2(l) shows the pulses approaching the boundaries. Now the unrefined solution has phase errors as well as amplitude errors. In Figure 8.2(m) the pulses are leaving the region, and the refinements "bunch up" against the boundary. Figure 8.2(n) shows that the fourth level refinements have been deleted. Finally, Figure 8.2(o) shows that all refinements have been deleted. Only the sinusoidal background remains.

Figures 8.3(a) to 8.3(e) show the algorithm applied to Burgers' equation (P3). The purpose of this calculation is to show that our method can "follow" shocks. This also shows that our method can handle a nonlinear problem. One would not necessarily expect the former, since we monitor the local truncation error, which requires a continuous third derivative. We do not suggest this as a practical method for computing with shocks. (But by modifying the error estimation, we believe we can develop a viable method for doing this, which will be described in a later paper.) We used 41 points on the coarsest mesh, with ratios $N = M = 3$, and maximum number of levels = 4. (In contrast to the smooth solution case, the estimate of the local truncation error does not decrease on finer meshes; hence our method will always use the maximum number of levels in this case.) The local error tolerance was 0.01, and the wave speed input to the program was 0.5 (which is the shock speed). Again, $\lambda_1 = 0.8$, and the dotted line shows the unrefined calculation.

Figure 8.3(a) shows the shock at $t = \Delta t = 0.08$, given as the exact solution. Figure 8.3(b) shows the shock later, at time $t = 1.6$. Notice that our method has eliminated all the wiggles, leaving a very sharp shock, except for the Gibb's phenomenon (whose height is 1.33). This can be eliminated by adding a small amount of dissipation, but only on the highest level of refinement.

Later graphs (Figures 8.3(c), (d), (e)) show nothing new, except that the refinements are following the shock, as desired.

We now proceed to more quantitative questions.

8.3. Choosing Refinement Ratios and Maximum Levels

Having answered the first five questions posed at the beginning of this chapter, we now proceed to the questions of how to choose the refinement ratios N and M (described in Chapter 2), and whether to use recursive refinements.

We used problem P1 (the first order wave equation) for this study. We chose 81 points for the coarse mesh, so $h_1 = 0.05$. We took $\lambda_1 = h_1/k_1 = 0.8$, $c = 1$, and recorded all errors at time $t = 3.6$. This is the time just before the pulse leaves the computational region $0 \le x \le 4$. As usual, the Lax-Wendroff method was used.

If the exact solution is denoted by u and the approximation by v, then the global truncation error is $e = u - v$. The $l_2(x)$ norm of v is given in Definition 3.1 of Chapter 3, in the notation of Section 2.2. The $l_2(x)$ norm of e is analogous. The maximum norm of e is (in the same notation)

$$\| e^i(t^i) \|_\infty = \max_{0 \le j \le N_i} |v_j^i(t^i)|_\infty, \; i = 0, 1, \ldots, s,$$

where $|\cdot|_\infty$ denotes the maximum absolute value of all the components of the vector (for an $n \times n$ system).

In our tables, a maximum mesh level of 1 signifies that only the coarse mesh is present (no refinement), 2 signifies one additional level of refinement, and so forth. The times reported are CPU seconds on a CDC 7600. (Since this machine runs only in batch, these times are highly reproducible between runs, and we did not need to take average times.) The storage used is for solution values only, and is the maximum storage used at the (refinement) level listed, for all times. Since the coarse mesh is static, it always uses 81 locations. The total listed is the sum over all levels.

Tables 8.1 and 8.2 show the results using three-level Richardson extrapolation. Table 8.1 uses a local truncation error bound of $\delta = 0.01$ and Table 8.2 uses $\delta = 0.001$. Before examining the results, let us state our expectations regarding the global truncation error. Let δ be the pointwise local truncation error bound, and $E(t) = \|e(t)\|_x$ be the $l_2(x)$ norm of the error at time t.

Propositions 4.1 and 4.2 contain constants K_T', K_T'' which depend only on the (time) interval of integration $0 \le t \le T$. It is well-known that these can be replaced by $K_2\exp(\alpha_2 t)$, $K_3\exp(\alpha_3 t)$, for some positive constants K_2, K_3, and some constants α_2, α_3, respectively. When the interior local truncation error per unit step $h^p d_1$, the boundary error $h^p d_2$, and the interpolation error $h^p d_3$ are all bounded (pointwise) by δ, then these propositions assert that the $E(T)$ is bounded by $C_4\exp(\alpha_i T)\delta[(b-a)T]^{1/2}$, where C_4 is constant and $i = 2$ or 3. Computational evidence strongly suggests that for our problems there is no exponential growth, $\alpha_i = 0$. (For the corresponding differential equation there is no growth either. It is also well-known that the interior approximation alone for the Cauchy problem produces no growth.) By examination of our results, we determined that $C_4 \approx 0.5$. Therefore, for an input tolerance of δ, we would expect $E(T)$ at $T = 3.6$ not to exceed $0.5 \cdot (4 \cdot 3.6)^{1/2}\delta \approx 2\delta$. Hence we expect the $l_2(x)$ norm of the error not to exceed 0.02 in Table 8.1 and 0.002 in Table 8.2, if enough levels of refinement are used.

Maximum Refinement Level	L	M	ℓ_2 Error	Maximum Error	ℓ_2-norm of Solution	Time (sec.)	Maximum Storage by Levels 2	3	4	5	Total
1	-	-	.198	.562	.268	.040					81
2	3	3	.102	.311	.317	.197	58				139
2	4	4	.0722	.230	.323	.233	73				154
2	5	5	.0524	.175	.325	.275	81				162
2	6	6	.0393	.131	.327	.324	97				178
2	7	7	.0306	.101	.327	.400	113				194
2	8	8	.0247	.0790	.328	.456	129				210
2	10	10	.0178	.0514	.328	.622	161				242
2	16	16	.0115	.0201	.329	1.35	257				338
3	3	3	.0207	.0628	.328	.505	49	94			224
3	4	4	.0115	.0201	.329	1.03	65	149			295
3	5	5	.0102	.0126	.329	2.04	81	216			378
3	6	6	.0100	.0126	.329	3.32	97	271			449
3	7	7	.0104	.0142	.329	4.74	113	330			524
3	8	8	.0105	.0143	.329	6.95	129	353			563
3	10	10	.0103	.0125	.329	15.0	161	471			713
4	3	3	.0107	.0139	.329	1.66	49	94	130		354
4	4	4	.0115	.0201	.329	1.17	65	149	0		295
4	5	5	.0102	.0126	.329	2.31	81	216	0		378
4	6	6	.0100	.0126	.329	3.57	97	271	0		449
5	3	3	.0107	.0139	.329	1.86	49	94	130	0	354
2	9	8	.0138	.0333	.328	0.49	145				226
2	18	16	.0101	.0125	.329	1.47	289				370

Table 8.1 Global Errors and Memory Used for Problem P1 Using 3-Level Richardson Extrapolation and Local Error Tolerance = 0.01

Maximum Refinement Level	L	M	ℓ_2 Error	Maximum Error	ℓ_2-norm of Solution	Time (sec.)	Maximum Storage by levels 2	3	4	5	Total
1	-	-	.198	.562	.268	.040					81
2	3	3	.101	.311	.318	.402	241				322
2	4	4	.0715	.230	.323	.601	321				402
2	5	5	.0514	.175	.326	.860	401				482
2	6	6	.0381	.131	.327	1.21	481				562
2	7	7	.0290	.101	.328	1.55	561				642
2	8	8	.0227	.0790	.329	1.97	641				722
2	10	10	.0148	.0514	.329	3.01	801				882
2	15	15	.00669	.0229	.329	6.72	1201				1282
2	20	20	.00378	.0128	.330	11.8	1601				1682
3	3	3	.0182	.0628	.329	.971	241	109			431
3	4	4	.00592	.0201	.329	1.83	321	177			579
3	5	5	.00246	.00820	.329	3.31	401	266			748
3	6	6	.00120	.00398	.330	5.62	481	355			917
3	7	7	.000665	.00215	.330	9.45	561	463			1105
4	3	3	.00237	.00705	.329	3.02	241	109	247		678
4	4	4	.000746	.00126	.329	11.3	321	177	497		1076
5	3	3	.00124	.00192	.329	11.6	241	109	247	364	1042

Table 8.2 Same as Table 8.1, but with Local Error Tolerance = 0.001

Table 8.1 illustrates one of the important features of our algorithm. It shows what happens when the refinement process is (recursively) carried as far as possible on smooth solutions. Compare $N = M = 4$, maximum refinement levels 3 and 4, or $N = M = 3$, maximum refinement levels 4 and 5. In these cases, allowing an additional level of refinement has resulted in no increase in memory used, and no decrease in error. This illustrates that, for smooth solutions, when the algorithm has satisfied the local error tolerance, it refuses to refine further, as it should.

In many of the cases shown, our expected l_2 error of 0.02 was not attained. But in all such instances (and even in some instances where this error was attained-- *i.e.*, the program is slightly over-conservative) the program gave a warning that the local error tolerance was not attained at some point. Thus, for smooth solutions, we should always choose a large value for the maximum level number (say 10) and let the program refine as much as possible. So choosing the maximum number of levels is not a problem.

Choosing the refinement ratios N and M is more problematic. It appears from the table that the most efficient combination of ratios which attains our desired l_2 error of 0.02 is $N = M = 4$, maximum level = 3 (or 4). However, a very close competitor is $N = M = 16$, maximum level = 2. Either one would do in this case. However, there is a reason to prefer the former. This can be seen by looking at the case $N = M = 10$, maximum level = 3. We see that the time goes up drastically for "relatively large" (8 or more) refinement ratios as the number of levels actually used goes up. Since we recommend using as many levels as necessary, if we ran the case $N = M = 16$, we couldn't be sure (in advance) that it would use only two levels. (Indeed, the case $N = M = 12$ wants to use 3 levels, but $N = M = 13$ uses only two.) Therefore, as Table 8.2 will also show, we recommend $N = M = 4$, 5 or 6 in general, with lower factors for coarser local error tolerances, and conversely. Using $N = M = 3$ is generally inefficient, because it

requires too many levels, and the cost goes up quickly with the number of levels, until no further levels are used.

At the end of Table 8.1 we showed that we are not confined to taking ratios N and M equal. However, there is no advantage to choosing $N \neq M$. For example, one of our entries has $N = 9$, $M = 8$. Since $\lambda_1 = k_1/h_1 = 0.8$, this implies $\lambda_2 = k_2/h_2 = 0.9$. If we took another level of refinement, we would have $\lambda_3 = 1.0125$ and our difference scheme would be unstable on the third level. Thus we will in the future take $N = M$.

Table 8.2 shows results similar to Table 8.1, but with a smaller tolerance. Note that in Table 8.2 the second refinement level is the whole region. (This was not the case in Table 8.1.) Taking only two refinement levels with large N and M ($N = M = 20$) does not produce the desired l_2 error 0.002. And the maximum error for this case is greater than 0.01, about ten times higher than desired. Comparing the cost with case $N = M = 6$, maximum level = 3 shows conclusively the necessity for recursive refinements. It furthermore affirms our policy of choosing N and M between 4 and 6.

We next come to the most important question of this chapter (and perhaps this thesis).

8.4. Efficiency of the Method

In Section 8.2 we showed that our method was able to resolve steep gradients, and even shocks, in the solution. We showed this by comparing the solution obtained by our method with one obtained on a uniform coarse mesh. This clearly showed the qualitative superiority of our "refined" solution over the "unrefined" one.

However, the "unrefined" solution cost far less to compute than the "refined" one. For example, in Figures 8.1(a) to (i), it cost 1.17 seconds to

compute the "refined" solution up to time $t = 3.6$ (without graphic output, etc.) *vs.* 0.04 seconds to compute the "unrefined" solution up to the same time, as shown in Table 8.1. This is a factor of 29 more expensive. But the unrefined solution is worthless.

Thus, to study the efficiency of our method, we need to compare the computing time taken by our method with the computing time taken to produce (approximately) the same error on a uniform (fine) mesh. As a by-product, we will also be able to compare the memory taken in the two approaches.

Because this is probably the most important result in this thesis, we made this study on two of our model problems: P1, the first-order wave equation, and P2, the second-order wave equation. The result is the same for both, and gives us confidence in extrapolating this to larger systems.

Our method is simple. Instead of comparing a "refined" solution Σ with a solution computed on only its coarsest mesh, we compare Σ with a solution computed on a uniform fine mesh whose spacing is the same as the spacing of the Σ's finest mesh. If these two produce approximately the same error, then we have a valid comparison.

Table 8.3 shows the results. In this table we have used two different difference schemes: the first (LW) is Lax-Wendroff on all refinement levels; the second (4th) is a hybrid method which uses the fourth order method (see Section 2.5) on the coarsest mesh, and Lax-Wendroff on all others. For the LW method, we used $\lambda_1 = 0.8$, but because of stability considerations, we had to use $\lambda_1 = 0.6$ for the hybrid method. We used three-level Richardson extrapolation for the LW method, but had to use differences for the hybrid method. As usual, the errors are at $t = 3.6$ and all other parameters for the refined examples are as in Table 8.2. For P2, the maximum error is the maximum over both components of the solution, and the l_2 error is the maximum of the l_2 errors of

Problem No.	Method	λ	No. Intervals on Coarsest Mesh	Maximum Refinement Level	L	M	ℓ_2 Error	Maximum Error	Time (sec.)	Memory Used	Work per Point
P1	LW	.8	1280	1	-	-	5.89-3	2.01-2	7.67	1281	5.99-3
P1	LW	.8	80	3	4	4	5.92-3	2.01-2	1.83	579	3.16-3
P1	LW	.8	2000	1	-	-	2.42-3	8.20-3	18.4	2001	9.20-3
P1	LW	.8	80	3	5	5	2.46-3	8.20-3	3.31	748	4.43-3
P1	4th	.6	1280	1	-	-	5.89-3	2.02-2	11.1	1281	8.67-3
P1	4th	.6	80	3	4	4	1.05-2	3.60-2	2.04	579	3.52-3
P1	4th	.6	2000	1	-	-	2.42-3	8.22-3	27.1	2001	1.35-2
P1	4th	.6	80	3	5	5	4.37-3	1.46-2	4.04	748	5.40-3
P2	LW	.8	1280	1	-	-	8.33-3	2.01-2	13.0	1281	1.02-2
P2	LW	.8	80	3	4	4	8.35-3	2.01-2	4.33	756	5.72-3
P2	LW	.8	2000	1	-	-	3.43-3	8.22-3	30.0	2001	1.50-2
P2	LW	.8	80	3	5	5	3.45-3	8.24-3	8.50	1009	8.42-3
P2	LW	.8	2880	1	-	-	1.65-3	3.99-3	63.16	2881	2.19-2
P2	LW	.8	80	3	6	6	1.67-3	4.00-3	15.19	1278	1.19-2

Table 8.3 Efficiency of the Method

either component. The memory given is the memory per solution component. We used upwind/downwind boundary conditions (see Section 8.1).

We see that in terms of computer time our method using LW is 3 to 5.5 times as efficient as using a uniform fine mesh which produces the same error. In terms of memory, a factor of 1.7 to 2.2 is gained. At first it might seem surprising that our method could be more efficient, since it requires much greater overhead than the uniform mesh method. The overhead is needed to estimate the local truncation error and adjust the refinements. This is compensated for, however, by being able to take large time steps in unrefined regions. Using a uniform mesh, we must take fine time steps everywhere.

An additional aspect of efficiency that should be mentioned involves the work per mesh point. Our mesh refinement algorithm reduces the (maximum) number of mesh points needed to achieve a given accuracy, and this naturally reduces the amount of work, but does the amount of effort *per mesh point* decrease?

Table 8.3 also gives this figure, obtained by dividing the computer time by the maximum number of mesh points used. It is clear that in all cases the work per mesh point is decreased by a factor of two (the notation $n-m$ means $n \cdot 10^m$).

Our results also show that the hybrid LW/4th order method is not competitive in efficiency with pure Lax-Wendroff. We also tried using the hybrid method with $\lambda_1 = 0.2$, all other $\lambda_l = 0.8$. Though improved somewhat, the results still were not competitive with pure Lax-Wendroff. In addition, the hybrid method was quite cumbersome to implement.

In general, of course, one will not get a factor of three or any other specific efficiency factor. This depends primarily on the fraction of the region needing refinement, and other factors such as the local error tolerance, when (for which

t) we are doing the comparison, the wave speed, and so forth.

It might be argued that we obtained our results only by adjusting or tuning the parameters N, M. To refute this charge, we have shown several different values of N and M. This shows that although we cannot easily determine the optimal N, M, even suboptimal choices still yield a significant savings in execution time.

8.5. Behavior as $h \to 0$

As we pointed out in Chapter 3, we can study two types of convergence. (In all cases we hold the refinement ratios N and M fixed.) In the first, we hold the local error tolerance δ fixed and let $h_1 \to 0$. In the second, we let $h_1 \to 0$ and let $\delta = C(h_1)^2$.

We first keep the maximum number of refinement levels constant, and less than necessary (for the method to refine as much as possible), and study the first type of convergence. Table 8.4 shows these results on P1 using pure Lax Wendroff, $\lambda_1 = 0.8$, $N = M = 4$, three-level Richardson extrapolation, and local error tolerance = 0.001. For the smaller values of h, the $O(h^2)$ behavior of the errors (both maximum and l_2) is apparent.

Next we do the same test, but choose the maximum number of levels large enough so that the method refines as much as possible. The maximum level is 5 here. The convergence is $O(h^2)$ for the maximum error with the coarse mesh size going from $N_0 = 80$ to 160. But as we saw in Chapter 3, the grid is approaching a uniform mesh as $h \to 0$ and this slows down the convergence. As $h \to 0$ the number of levels used approaches 1.

Finally, we let $h \to 0$ and let $\delta = 0(h^2)$. Here we see the convergence is faster, and the maximum error is finally $O(h^2)$. The l_2 error does not behave as well.

No. of Intervals on Coarse Mesh	Maximum Refinement Level	L	M	Local Error Tol.	ℓ_2 Error	Maximum Error	ℓ_2-norm of Solution	Time (sec.)	Maximum Storage by Levels 1	2	3	4	Total
40	2	4	4	.001	.144	.391	.304	.178	41	161			202
80	2	4	4	.001	.0715	.230	.323	.647	81	321			402
160	2	4	4	.001	.0228	.0790	.328	.782	161	122			283
320	2	4	4	.001	.00592	.0201	.329	2.74	321	157			478
40	3	4	4	.001	.0228	.0790	.328	.592	41	161	158		360
80	3	4	4	.001	.00592	.0201	.329	1.89	81	321	181		583
160	3	4	4	.001	.00296	.00503	.329	3.77	161	118	281		560
320	3	4	4	.001	.000756	.00126	.329	13.5	321	161	445		927
40	5	4	4	.001	.00240	.00325		33.8	41	161	121	349	672
80	5	4	4	.001	.000743	.00126		12.35	81	321	177	497	1076
160	5	4	4	.001	.00260	.00330		22.5	161	101	277	433	972
320	5	4	4	.001	.000756	.00126		13.46	321	157	433	0	911
640	5	4	4	.001	.000454	.00140		28.8	641	257	374	0	1272
1280	5	4	4	.001	.000418	.00126		43.9	1281	433	0		1714
40	5	4	4	.01	.00388	.0109		2.78	41	161	109	213	524
80	5	4	4	.0025	.00139	.00498		8.20	81	321	169	305	876
160	5	4	4	.000625	.000438	.00139		26.7	161	641	289	493	1584
320	5	4	4	.000156	.000134	.000354		93.6	321	1076	545	869	2811

Table 8.4 Behavior of Global Error as $h_1 \to 0$

8.6. Estimating the Local Truncation Error in the Interior

In this section we will consider the other ways of estimating the interior local truncation error that we examined in Chapter 5: differences and two-level Richardson extrapolation. In all other tables in this chapter we used three-level Richardson estimation (except on the first level of the hybrid method) in the interior of the region.

Table 8.5 shows these results for problem P1. As usual, the parameters not listed in the table are the same as in the computations for Table 8.1 or 8.2. R3 signifies 3-level Richardson extrapolation, R2 is two-level Richardson extrapolation, and D signifies differences. Omitted entries are the same as the entries above them.

We see that there is very little difference in efficiency between these methods for problem P1. The use of differences seems to be slightly more efficient. But in our opinion, the greater convenience of three-level Richardson for interior approximations far outweighs any small efficiency differences.

8.7. Estimating the Local Truncation Error at Boundaries

In this section we will fix our problem (P2), and our interior approximation and error estimation methods (Lax Wendroff and three-step Richardson, respectively). Then we will vary our boundary approximation and our method of error estimation at the boundary.

We will use upwind/downwind differencing and first-order extrapolation (see Section 8.1). For the former we will estimate the error both by using the modified Richardson 3-step method (Section 5.4) and by replacing t derivatives by x derivatives in the truncation error and using differences. For extrapolation we can only use differences. The results are shown in Table 8.6. In all cases, the number of intervals on the coarsest mesh is 80, the maximum number of

Maximum Refinement Level	L	M	Local Error Tolerance	Method	ℓ_2 Error	Maximum Error	Time (sec.)
3	4	4	.01	R3	.0115	.0201	1.03
				R2	.0115	.0201	1.05
				D	.0115	.0201	.957
3	6	6	.01	R3	.0100	.0126	3.32
				R2	.0099	.0126	3.65
				D	.00992	.0126	3.35
3	4	4	.001	R3	.00592	.0201	1.83
				R2	.00592	.0201	1.74
				D	.00592	.0201	1.52
3	6	6	.001	R3	.00120	.00398	5.62
				R2	.00120	.00398	5.59
				D	.00120	.00398	5.28

Table 8.5 Using Different Methods to Estimate the Local Truncation Error

refinement levels is 5, and the refinement ratios $L = M = 4$. In all cases the fifth refinement level was not used. U/D signifies upwind/downwind differencing, and Rich. signifies the modified 3-step Richardson method. The memory occupied by solution values is the maximum total over all refinement levels for one component of the solution. As usual, all other parameters not shown are the same as in the computation for Table 8.1, except that we use Problem P2 rather than P1.

Clearly, the different boundary approximations and error estimation methods produce approximately the same results. This supports our claim that our method of adaptively handling boundaries is quite general.

8.8. How Often Should the Local Truncation Error Be Checked?

In Chapter 2 we used subsequences to describe the times at which we estimate the local truncation error (and possibly alter refinements). In this section we shall show that for Problem P1 it is unwise to monitor the local truncation error more often than every coarse time step.

Table 8.7 shows the results of these computations for Problem P1. All parameters not mentioned are the same as in the computations for Table 8.1. The meaning of (a) under "tolerance frequency" in Table 8.7 is how many coarse

Boundary Approx.	Error Estimation	Local Error Tolerance	l_2 Error	Maximum Error	Time (sec.)	Memory Used
U/D	Rich.	.01	1.20-2	2.18-2	3.84	630
U/D	diff.	.01	1.21-2	2.18-2	3.87	630
extrap.	diff.	.01	1.18-2	2.18-2	3.93	642
U/D	Rich.	.001	8.02-4	1.34-3	42.3	1774
U/D	diff.	.001	8.02-4	1.34-3	41.9	1774
extrap.	diff.	.001	8.00-4	1.34-3	42.6	1814

Table 8.6. Error Estimation at Boundaries

Maximum Refinement Level	L	M	Tol.	Tol. (a)	Freq. (b)	ℓ_2 Error	Maximum Error	Time (sec.)	Maximum Storage by Levels 2	3	4	Total
4	4	4	.01	1	1	.0115	.0201	1.17	65	149	0	295
				1	2	.0115	.0201	1.64	65	125	0	271
				1	3	.0115	.0201	3.10	65	125	0	271
				2	1	.0112	.0202	1.16	69	173	0	323
				2	2	.0119	.0201	1.15	69	149	0	299
				2	3	.0112	.0202	1.24	69	173	0	323
				3	1	.0119	.0238	1.26	81	205	0	367
				3	2	.0114	.0219	1.21	81	161	0	323
				3	3	.0119	.0238	1.38	81	205	0	367
				4	1	.0107	.0241	1.31	85	221	0	387
				5	1	.0129	.0380	1.41	89	249	0	419
				6	1	.0235	.0857	1.48	93	281	0	455
3	6	6	.001	1	1	.00120	.00398	5.62	481	355		917
				1	2	.00120	.00398	7.86	481	313		875
				1	3	.00120	.00398	7.90	481	313		875
				2	1	.00120	.00398	6.10	481	415		977
				2	2	.00120	.00398	5.68	481	355		917
				2	3	.00120	.00398	6.06	481	415		977
				3	1	.00121	.00398	6.52	481	475		1037

Table 8.7 How Often Should the Local Truncation Error Be Checked?

time steps occur between checks of the local error. The column (b) has two different meanings, depending on column (a). If column (a) is 1 then we check the truncation error at any time a refinement whose level is less than or equal to (b) is about to be advanced. Thus, in these cases we check *more often* then every coarse time step. Table 8.7 shows that this is very costly and produces no benefits whatever.

If (a) in Table 8.7 is greater than one, a one in column (b) signifies that we check all refinements every (a) coarse time steps. If column (a) is greater than one and (b) is greater than one, we check refinements with levels greater than or equal to (b) every coarse time step, and all others every (a) coarse time steps. Of course, in all cases in this table, the buffers mentioned in Section 2.6 have to be modified, in a way analogous to the argument given there.

Our results for these cases show very little difference from checking every coarse time step, until the checking frequency becomes too seldom (as in case (a) = 6, (b) = 1). Then the accuracy starts to deteriorate, because a pulse may enter the boundary before it is enclosed in refinement(s). (The algorithm could easily be modified to check the boundaries at every coarse time step, but we did not do this.)

We conclude that for this problem we may as well check the local error every coarse time step, although this may depend on factors such as the spacing of the coarse mesh, the wave speed, and the presence of forcing functions (terms kF in (2.1)). Also, the results may be radically different in more than one space dimension (M. Berger, Ph.D. thesis [to appear]).

8.9. Linear vs. Quadratic Interpolation

One final implementation detail we considered is whether to use linear or quadratic interpolation when a level l refinement moves into a region formerly

occupied only by a level $l-1$ refinement. (This is relevant to the statement of Propositions 4.1 and 4.2.) Table 8.8 shows that there is practically no difference. (All parameters not mentioned are as in the computation for Table 8.1. As usual, omitted values are the same as the ones above.) We used quadratic interpolation elsewhere in this chapter, but linear interpolation would be preferred because it is easier to program.

Maximum Refinement Level	L	M	Local Error Tolerance	Linear or Quadratic Interp.	ℓ_2 Error	Maximum Error	Time (sec.)
3	4	4	.01	Q	.0115	.0201	1.03
				L	.0116	.0201	1.06
3	6	6	.01	Q	.0100	.0126	3.32
				L	.0100	.0127	3.60
3	4	4	.001	Q	.00592	.0201	1.83
				L	.00593	.0201	1.87
3	6	6	.001	Q	.00120	.00398	5.62
				L	.00121	.00398	5.76
4	3	3	.001	Q	.00237	.00705	3.02
				L	.00240	.00705	3.13

Table 8.8 Linear vs. Quadratic Interpolation

CHAPTER 9

Conclusions and Extensions

In this thesis we have developed and partially analyzed an adaptive finite difference method for hyperbolic systems in one space dimension. It is intended for problems which are smooth in most parts of the spatial region, but which have large gradients which require "moving" refinement(s) for accurate approximation. The algorithm was described in Chapter 2.

Although our method was originally developed for problems with smooth solutions, and the analyses hold only for that situation, we found in Section 8.2 that our method also works for problems with shocks, in the sense that the refinement(s) follow the shock. However, the method is not yet efficient for that case.

The most important result of this thesis is that our method can be much more efficient (for a given level of accuracy) than using a uniform grid. Specifically, in Section 8.4 we found that our method was 3-5 times more efficient (in computing time). Work by W. Gropp [1980] and M. Berger [to appear] in two spatial dimensions confirms this. We believe the efficiency of the shock calculation mentioned above can be greatly improved by changing the method of error estimation, and we will do this in the future.

Our method also provides efficiencies in storage, but these are not as dramatic as the execution time savings. We expect the savings to be greater for more space dimensions.

Our algorithm is the only one of which we are aware which adaptively treats time-dependent boundary conditions for hyperbolic systems (as was shown by

problem P2 in Section 8.7) in a systematic and general way. This is obviously important for limited area weather forecasting, among other problems.

We saw that our algorithm does indeed accurately "track" moving pulses, even when they merge, separate, or pass through boundaries, without being distracted by background "noise" (as appeared in our problems P1 and P2).

As explained in Chapter 3, our method of mesh refinement requires the use of a stability definition different from the usual Gustafsson-Kreiss-Sundström [1972] definition. In Chapter 4 we stated but did not prove a convergence proposition analogous to Gustafsson's [1975] result, but using the new stability definition. Using this, and the results of Pereyra and Sewell, we proved a result (Proposition 4.2) which gives insight into why our algorithm can be expected to produce economies. Our algorithm does not increase the order of convergence, but, loosely speaking, it can greatly decrease the constant multiplying h^p in the global error. Our computations in Section 8.5 confirm the rate of convergence given in Proposition 4.2.

In Chapter 5 we examined methods for estimating the local truncation error. For interior approximations we found that the three-step Richardson extrapolation method was the most versatile and easy to use. We proved that this procedure was valid for a large class of explicit difference schemes (namely those whose local truncation error has the same order in both space and time). We found that this scheme can sometimes be applied (with modifications) at the boundaries, but that differences provide the most versatile method here.

In Chapter 6 we discussed the data structures necessary for an efficient implementation of the algorithm. The nested structure of recursive refinements was indicated by a four-way linked tree of records, and the solution values were contained in sequentially allocated deques. We used a macro preprocessor for Fortran to implement this, since Fortran lacks convenient facilities for data and

control structures.

Many additional areas for research suggest themselves. The first is the completion of the theoretical results using the new definition of stability in Chapter 3. Although it is well-known that our difference scheme when applied to problems P1 and P2 on a uniform mesh is stable according to the Gustafsson-Kreiss-Sundström definition, we have not proved that it is stable according to the new definition. For refined meshes with nonuniform time steps, the only known stability result has recently been given in M. Berger's Ph.D. thesis [to appear] for the GKS stability definition. Still needed is a similar result for the new stability definition. Proposition 4.1 of Chapter 4 on the rate of convergence also needs to be proved.

We believe Theorem 5.1 is true in more general circumstances although whether it can be proved then is an open question. The first is variable coefficients in one space dimension. The second is linear hyperbolic systems in more than one space dimension (but here a theorem like Gustafsson's is lacking to guarantee the order of the global error). Another generalization is to difference schemes that depend nonlinearly on approximate solution values. We believe that Theorem 5.1 holds for some of these cases, although we are unable to state which ones. Evidence for this was provided by our shock calculation for the inviscid Burgers' equation in Chapter 8. On the other hand, we doubt that this theorem generalizes to implicit methods, or to some other types of equations (such as parabolic equations).

In proving Theorem 5.1, we assumed that not only the solution, but also the global error was sufficiently smooth. That is, we assumed the existence of an asymptotic expansion for the global truncation error. To our knowledge, the best result for the initial boundary value problem is Gustafsson's [1975] theorem, which only gives the size of the global error, but says nothing about its

smoothness. We believe this theoretical gap will be very difficult to overcome.

In the realm of implementation in one space dimension, the method of local error estimation needs to be altered (for efficiency reasons) in shock calculations. It also needs to be extended to implicit difference schemes (this is more difficult) and possibly to conservative difference schemes. On the other hand, we believe our algorithm applies without change to moving boundary layer problems.

In two space dimensions there are many more problems, but these are being considered in M. Berger's Ph.D. thesis [to appear]. Some of the new problems are: more complicated data structures, orientation of refinements, and clustering analysis.

References

Ascher, U., Christiansen, J., and Russell, R. D., "A Collocation Solver for Mixed Order Systems of Boundary Value Problems," *Math. Comp. 33* (1979), 659-679.

Babushka, I., and Rheinboldt, W., "Error Estimates for Adaptive Finite Element Computations," *SIAM J. Numer. Anal. 15* (1978), 736-754.

Bank, Randolph, and Sherman, Andrew, "PLTMG Users' Guide," July 1979 Version, Center for Numerical Analysis CNA 152, University of Texas at Austin (September 1979).

Berger, M., Stanford University Ph.D. thesis, to appear.

Berger, M., Gropp, W., and Oliger, J., "Mesh Generation for Time-Dependent Problems: Criteria and Methods," in Proc. Workshop on Numerical Grid Generation Techniques for Partial Differential Equations, NASA Langley Research Center (October, 1980).

Berger, M., Gropp., W., and Oliger, J., Stability Analysis, to appear.

Brackbill, J. U. and Saltzman, J., "Adaptive Zoning for Singular Problems in Two Dimensions," to appear in *J. Comp. Phys.*

Brandt, Achi, "Multi-Level Adaptive Solutions to Boundary Value Problems," *Math. Comp. 31* (1977a), 333-390.

Brandt, Achi, "Multi-Level Adaptive Techniques (MLAT) for Partial Differential Equations: Ideas and Software," in Rice, John, ed., *Mathematical Software III*, Academic Press, New York (1977b) 277-318.

Browning, G., Kreiss, H., and Oliger, J., "Mesh Refinement," *Math. Comp. 27* (1973), 29-39.

Budnik, P., and Oliger, J., "Algorithms and Architecture," in Kuck, D. J., Lawrie, D. H. and Sameh, A. H., *High Speed Computer and Algorithm Organization*,

Academic Press, New York (1977), 355-370.

Ciment, Melvyn, "Stable Difference Schemes with Uneven Mesh Spacings," *Math. Comp. 25* (1971), 219-227.

Cook, A. James, and Shustek, L. J., "A User's Guide to MORTRAN2," Computation Research Group, Stanford Linear Accelerator Center, Stanford, Calif. (June 1975).

Davis, Stephen, and Flaherty, Joseph, "An Adaptive Finite Element Method for Initial Boundary-Value Problems for Partial Differential Equations," *SIAM J. Sci. Stat. Computing 3* (1982), 6-27.

de Boor, Carl, "On Writing an Automatic Integration Algorithm," in Rice, John, ed., *Mathematical Software*, Academic Press, New York (1971a), 201-209.

de Boor, Carl, "CADRE: Cautious Adaptive Romberg Extrapolation," in Rice, John, ed., *Mathematical Software*, Academic Press, New York (1971b), 417-449.

de Boor, Carl, "Good Approximation by Splines with Variable Knots. II," in *Conference on the Numerical Solution of Differential Equations*, Lecture Notes in Mathematics 363, Springer Verlag, New York (1973), 12-20.

de Boor, Carl, "How Small Can One Make the Derivatives of an Interpolating Function?" *J. Approx. Theory 13* (1975a), 105-116.

de Boor, Carl, "A Smooth and Local Interpolant with 'Small' k-th Derivative," in *Numerical Solutions of Boundary Value Problems for Ordinary Differential Equations*, Academic Press, New York (1975b), 177-197.

de Doncker, Elise, "An Adaptive Extrapolation Algorithm for Automatic Integration," *ACM SIGNUM Newsletter 13* (1978), 12-17.

Dupont, Todd, "Mesh Modification for Evolution Equations," to appear.

Dwyer, H. A., Kee, R. J. and Sanders, B. R., "Adaptive Grid Method for Problems in Fluid Mechanics and Heat Transfer," *AIAA J. 18* (1980), 1205-1212.

Feldman, Stuart, "The Programming Language EFL," Bell Laboratories Comp. Sci. Tech. Rep. No. 78 (1979).

Gannon, Dennis, "Self Adaptive Methods for Parabolic Partial Differential Equations," Dept. of Computer Science, Univ. of Illinois (Aug. 1980).

Gear, C. William, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, N. J. (1971).

Gelinas, R. J., Doss, S. K., and Miller, K., "The Moving Finite Element Method: Applications to General Partial Differential Equations with Multiple Large Gradients," *J. Comp. Phys. 40* (1981), 202-249.

Gropp, William D., "A Test of Moving Mesh Refinement for 2-D Hyperbolic Problems," *SIAM J. Sci. Stat. Computing 1* (1980), 191-197.

Gropp, William D., Preprocessor Language, to appear.

Grosse, Eric, "Software Restyling in Graphics and Programming Languages," Stanford Univ. Comp. Sci. Report STAN-CS-78-663 (1978).

Gustafsson, Bertil, "The Convergence Rate for Difference Approximations to Mixed Initial Value Problems," *Math. Comp. 29* (1975), 396-406.

Gustafsson, Bertil, "The Convergence Rate for Difference Approximations to General Mixed Initial Boundary Values Problems," *SIAM J. Numer. Anal. 18*, (1981), 179-190.

Gustafsson, B., Kreiss, H., and Sundstrom, A., "Stability Theory of Difference Approximations for Mixed Initial Boundary Value Problems, II," *Math. Comp. 26* (1972), 649-686.

Henrici, Peter, *Discrete Variable Methods in Ordinary Differential Equations*, Wiley, New York (1962).

Jensen, Kathleen, and Wirth, Niklaus, *Pascal User Manual and Report*, 2nd ed., Springer Verlag, New York (1974).

Kahaner, D. K., and Wells, M. B., "An Experimental Algorithm for N-Dimensional Adaptive Quadrature," *ACM TOMS 5* (1979), 86-96.

Keller, H. B., *Numerical Methods for Two-Point Boundary-Value Problems*, Blaisdell, Waltham, Mass. (1968), 78-80.

Kernighan, Brian, "RATFOR - a Preprocessor for a Rational Fortran," *Software -- Practice and Experience 5* (1975), 395-406.

Knuth, Donald E., *The Art of Computer Programming*, vol. 1, 2nd ed., Addison-Wesley, Reading, Mass. (1973).

Kreiss, H., and Oliger, J., "Comparison of Accurate Methods for the Integration of Hyperbolic Equations," *Tellus XXIV* (1972), 199-215.

Krogh, F. T., "VODQ/SVDQ/DVDQ - Variable Order Integrators for the Numerical Solution of Ordinary Differential Equations," TU Doc. No. CP-2308, NPO-11643, Jet Propulsion Laboratory, Pasadena, Calif. (1969).

Lentini, M., and Pereyra, V., "An Adaptive Finite Difference Solver for Nonlinear Two-Point Boundary Value Problems with Mild Boundary Layers," *SIAM J. Numer. Anal. 14* (1977), 91-111.

Lentini, M., and Pereyra, V., "Boundary Problem Solvers for First Order Systems Based on Deferred Corrections," in *Numerical Solutions of Boundary Value Problems for Ordinary Differential Equations*, Academic Press, New York (1975).

Lindberg, Bengt, "Error Estimation and Iterative Improvement for Discretization Algorithms," *BIT 20* (1980), 486-500.

Lyness, James, "Algorithm 379: SQUANK (Simpson Quadrature Used Adaptively - Noise Killed)," *CACM 13* (1970), 260-263.

McKeeman, William, "Algorithm 145: Adaptive Numerical Integration by Simpson's Rule," *CACM 5* (1962), 604.

Miller, K., and Miller, R., "Moving Finite Elements," *SIAM J. Numer. Anal.*, to appear.

Mohilner, Patricia, "Using Pascal in a Fortran Environment," *Software - Practice and Experience 7* (1977), 357-362.

Oliger, Joseph, "Approximate Methods for Atmospheric and Oceanographic Circulation Problems," Proc. Third International Symposium on Computing

Methods in Applied Sciences and Engineering, Springer Verlag, New York (1978).

Oliger, Joseph, "Fourth Order Difference Methods for the Initial Boundary-Value Problem for Hyberbolic Equations," *Math. Comp. 28* (1974), 15-25.

Oliger, Joseph, "Hybrid Difference Methods for the Initial Boundary-Value Problem for Hyperbolic Equations," *Math. Comp. 30* (1976), 724-738.

Oliger, Joseph, "Constructing Stable Difference Methods on Piecewise Uniform Grids," to appear.

Pearson, Carl E., "On a Differential Equation of Boundary Layer Type," *J. Mathematical Phys. 47* (1968), 134-154.

Pereyra, V., "Higher Order Finite Difference Solution of Differential Equations," Stanford Univ. Comp. Sci. Report STAN-CS-73-348 (1973).

Pereyra, V., and Sewell, E. G., "Mesh Selection for Discrete Solution of Boundary Problems in Ordinary Differential Equations," *Numer. Math. 23* (1975), 261-268.

Rai, M. M. and Anderson, D. A., "Application of Adaptive Grids to Fluid-Flow Problems with Asymptotic Solutions," *AIAA J. 20* (1982), 496-502.

Rheinboldt, Werner C. and Mesztenyi, Charles, "On a Data Structure for Adaptive Finite Element Mesh Refinements," *ACM TOMS 6* (1980), 166-187.

Richtmyer, Robert and Morton, K. W., *Difference Methods for Initial Value Problems*, 2nd. ed., Wiley, New York (1967).

Russell, R. D. and Christiansen, J., "Adaptive Mesh Selection Strategies for Solving Boundary Value Problems," *SIAM J. Numer. Anal. 15* (1978), 59-80.

Shampine, L. M. and Gordon, M. K., *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, Freeman, San Francisco (1975).

Steger, Joseph and Chaussee, Denny, "Generation of Body-Fitted Coordinates Using Hyperbolic Partial Differential Equations," *SIAM J. Sci. Stat. Computing 1* (1980), 431-437.

Stetter, Hans, "Global Error Estimation in Adams PC-Codes," *ACM TOMS 5* (1979), 415-430.

White, Andrew B., "On Selection of Equidistributing Meshes for Two-Point Boundary Value Problems," *SIAM J. Numer. Anal. 16* (1979), 472-502.

Zahn, Charles T., Jr., "A User Manual for the MORTRAN2 Macro-Translator," Computation Research Group, Stanford Linear Accelerator Center, Stanford, Calif. (August 1975).

APPENDIX A

Appendix: Program Listing

The following is the listing of the mesh refinement program for problem P2 (the second order wave equation with counter-streaming pulses). We include it both to resolve any small details which had to be omitted from the text, and to show the advantages of using a preprocessor language for this type of algorithm. As described in Chapter 7, the language used is an extension of Mortran [Cook and Shustek, 1975].

```
  $U5
0
0  " PROGRAM TO SOLVE THE INITIAL BOUNDARY-VALUE PROBLEM FOR THE ONE-
0    DIMENSIONAL SECOND ORDER WAVE EQUATION
0
0                        2
0                   U   - C U   = 0,  C .GT. 0,
0                    TT     XX
0
0
0    REWRITTEN AS THE 2 BY 2 FIRST ORDER SYSTEM, WITH V = C*DU/DX, AND
0    W = DU/DT,
0
0                   V  = C*W
0                    T     X
0
0                   W  = C*V
0                    T     X
0
0    IN THE STRIP A .LE. X .LE. B, T .GE. 0, WITH AUTOMATIC INSERTION
0    OF MESH REFINEMENTS IN REGIONS WHERE THE SOLUTION IS CHANGING
0    RAPIDLY.  INITIAL CONDITION  V(X, 0) = F(X) + G(X),  W(X, 0) =
0    -F(X) + G(X), WHERE F AND G ARE GIVEN BY SUBROUTINE EXACT.SOLUTION.
0    BOUNDARY CONDITIONS  V(A, T) = W(A, T) + 2F(A - C*T),
0                    V(B, T) = -W(B, T) + 2G(B + C*T).
0    EXACT SOLUTION  V(X, T) = F(X - C*T) + G(X + C*T),
0                W(X, T) = -F(X - C*T) + G(X + C*T).
0
0    REFERENCE:
0      BOLSTAD, JOHN H., 'AN ADAPTIVE FINITE DIFFERENCE METHOD FOR
0      HYBPERBOLIC SYSTEMS IN ONE SPACE DIMENSION', LBL-13287 AND
0      STAN-CS-82-899, JUNE, 1982.
0
0  "
0          " SHORTEN IDENTIFIERS"
  $ 'INITIALIZE' = 'INITIA'
  $ 'ADVANCE.SOLUTION' = 'ADVSOL'
0
0    REAL TIME;
0    INTEGER HANDLE, MILSEC;
0    LOGICAL RUNNING;
0
0    HANDLE = 0;
0    CALL LIB$INIT_TIMER(HANDLE);
0    RUNNING = .TRUE.;
0    WHILE RUNNING <
1      CALL INITIALIZE(RUNNING);
1      IF (.NOT. RUNNING) EXIT;
1      CALL ADVANCE.SOLUTION;
1      CALL LIB$STAT_TIMER(2, MILSEC, HANDLE);
1      TIME = 0.01*MILSEC;
1      OUTPUT TIME; (' ELASED TIME = ', F9.3, ' SECONDS');
1      CALL LIB$INIT_TIMER(HANDLE)>
0    STOP;
0    END;
0
0             "MORTRAN MACROS"
0
  $ 'ONE.TIME.STEP' = 'ONETIM'
  $ 'EXACT.SOLUTION' = 'EXTSOL'
  $ 'INITIAL.CONDITIONS' = 'INITCN'
  $ 'EXTEND.LEFT' = 'EXTLFT'
  $ 'EXTEND.RIGHT' = 'EXTRGT'
  $ 'DETERMINE.REFINEMENTS' = 'DETREF'
  $ 'ADJUST.MESH' = 'ADJMES'
```

```
  $ 'FILL.IN' = 'FILLIN'
  $ 'SEPARATE.REF' = 'SEPRAT'
  $ 'INTERPOLATE' = 'INTRPO'
  $ 'ESTIMATE.ERROR' = 'ESTERR'
  $ 'TOLERANCE' = 'TOLRNC'
  $ 'TOL.ACHIEVED' = 'TOLACV'
  $ 'TOL.FREQ' = 'TOLFRQ'
  $ 'DIFFERENCES' = 'DIFRNC'
  $ 'DIFFERENCES2' = 'DIFRN2'
  $ 'RICHARDSON2' = 'RICHA2'
  $ 'RICHARDSON3' = 'RICHA3'
  $ 'LEFTMOST' = 'LFTMST'
  $ 'ESTERROR' = 'ESTRRR'
  $ 'MAXLEVEL' = '5'        "MAXIMUM NUMBER OF LEVELS OF REFINEMENT"
  $ 'MAXLEVELP1' = '6'      "MAXLEVEL + 1"
  $ 'MAXRFINE' = '15'       "MAXIMUM NUMBER OF REFINEMENTS.  THIS NUMBER
0                    MUST BE .GE. MAXLEVEL + 1."
  $ 'MEMAVAIL' = '3000'     "MEMORY AVAILABLE FOR V AND REFINEMENTS"
  $ 'HEIGHT' = '16'         "VERTICAL HEIGHT OF CRUDE PLOTS"
  $ 'HEIGHTP1' = '17'       "HEIGHT + 1"
  $ 'PAGEWIDTH' = '120'     "NO. OF CHARS. ON LINE OF OUTPUT"
  $ 'EMPTY' = '-1'          "SIGNAL FOR RIGHT END OF EMPTY MESH"
  $ 'NCOMP' = '2'           "NUMBER OF EQUATIONS AND COMPONENTS IN
0                    SOLUTION VECTOR"
0
  $ 'USE SOLN;' = 'COMMON /SOLN/ ZVNEW, VNEW, ZV, V;
0       REAL ZVNEW(NCOMP), VNEW(NCOMP, MEMAVAIL), ZV(NCOMP), V(NCOMP,
0          MEMAVAIL);'
0          "ZVNEW = VNEW(1,0), ZV = V(1,0).
0          THIS SIMULATES V(1:NCOMP, 0:MEMAVAIL) IN ALGOL NOTATION."
0    ARRAY RFIN(MAXRFINE) OF RECORD <
1       POINTER TO RFIN: LLINK, RLINK, COARSE, FINE;
1       INTEGER:  BASE, TOP, OLDBASE, OLDTOP, LEFT, RIGHT; >
  $ 'USE LFTMST;' = 'COMMON /LFTMST/ LEFTMOST;
0              POINTER TO RFIN: LEFTMOST(MAXLEVELP1);'
  $ 'USE COM3;' = 'COMMON /COM3/ A, B, N;
0              REAL A, B;    INTEGER N;'
  $ 'USE COM4;' = 'COMMON /COM4/ C, FACTOR, SCALE, TWOPI;
0              REAL C, FACTOR, SCALE, TWOPI;'
  $ 'USE COM5;' = '
0       COMMON /COM5/ NLEVEL, NPTSM, POWER, RIGHTB, BUFFER;
0       INTEGER NLEVEL, NPTSM(MAXLEVELP1), POWER(MAXLEVELP1),
0       RIGHTB(MAXLEVEL), BUFFER(MAXLEVEL);'
  $ 'USE STEPSZ;' = '
0       COMMON/STEPSZ/ H0, H, K, LAMBDA;
0       REAL H0, H(MAXLEVEL), K(MAXLEVEL), LAMBDA(MAXLEVEL);'
  $ 'USE COM7;' =
0       'COMMON /COM7/ C1, C2, C3, L2, L3, L6, NM1, NM2, NM3;
0       REAL C1, C2, C3, L2, L3, L6;
0       INTEGER NM1, NM2, NM3;'
  $ 'USE ZERO;' = 'COMMON /ZERO/ ZERO;       INTEGER ZERO;'
  $ 'USE XRATIO;' = 'COMMON /XRATIO/ XRATIO;      INTEGER XRATIO;'
  $ 'USE COUNT;' = 'COMMON /COUNT/ NSHRT, NSHL;
0              INTEGER NSHRT, NSHL;'
  $ 'USE DEBUG;' = 'COMMON /DEBUG/ DEBUG;      INTEGER DEBUG;'
  $ 'USE COM12;' = 'COMMON /COM12/ TOLERANCE, NTIME, SKIPPR, TRATIO,
0              QUADRAT, PRINT, RICHSN, TOLCHK, TOL.FREQ,
0              TOL.ACHIEVED, BDRY;
0              REAL TOLERANCE;
0       INTEGER BDRY, NTIME, RICHSN, SKIPPR, TOL.FREQ(2), TRATIO;
0       LOGICAL QUADRAT, PRINT, TOLCHK, TOL.ACHIEVED;'
  $ 'USE METHOD;' = 'COMMON /METHOD/ METHOD;    INTEGER METHOD;'
  $ 'USE COM14;' = 'COMMON /COM14/ C6, C7, C8, C10;
0       REAL C6(MAXLEVEL), C7(MAXLEVEL), C8(MAXLEVEL), C10(MAXLEVEL);'
  $ 'USE ERROR;' = '
0       COMMON /ERROR/ ZESTER, ESTERROR;
0       REAL ZESTER(NCOMP), ESTERROR(NCOMP, MEMAVAIL);'
```

```
0       "THIS SIMULATES ESTERROR(1:NCOMP, 0:MEMAVAIL)"
0
0   SUBROUTINE INITIALIZE(RUNNING);
0
0     USE SOLN; USE LFTMST; USE COM3; USE COM4; USE COM5; USE STEPSZ;
0     USE COM7; USE ZERO; USE XRATIO; USE COUNT; USE DEBUG; USE COM12;
0     USE METHOD; USE COM14; USE ERROR; DEFINE RFIN;
0     REAL S;
0     INTEGER I, J, L, NLEVP1;
0     POINTER TO RFIN: P;
0     LOGICAL RUNNING;
0     CHARACTER*9 DAY;
0     DATA ZERO /0/;
0
0     INPUT NLEVEL, XRATIO, TRATIO, DEBUG, RICHSN, BDRY, TOL.FREQ, METHOD,
0     N, SKIPPR, NTIME, QUADRAT, TOLCHK, LAMBDA(1), SCALE, TOLERANCE, C;
0     (9I2, 3I4, 2L1, 4F7.3);
0     IF N .LE. 0 <
1       RUNNING = .FALSE.;
1       GO TO :EXIT:>
0     IF NLEVEL .LE. 0 .OR. NLEVEL .GT. MAXLEVEL <
1       OUTPUT NLEVEL; (' INCORRECT INPUT, NLEVEL = ', I5);
1       RUNNING = .FALSE.;
1       GO TO :EXIT:>
0     IF MAXRFINE .LT. MAXLEVELP1 <
1       OUTPUT; (' *** MAXRFINE TOO SMALL');
1       RUNNING = .FALSE.;
1       GO TO :EXIT:>
0     IF TRATIO .LT. 2 .OR. XRATIO .LT. 2 <
1       OUTPUT XRATIO, TRATIO; (' XRATIO, TRATIO =', 2I4, ' TOO SMALL');
1       RUNNING = .FALSE.;
1       GO TO :EXIT:>
0     TOL.ACHIEVED = .TRUE.;
0     NM1 = N - 1;
0     NM2 = N - 2;
0     NM3 = N - 3;
0     A = 0.;
0     B = 4.;
0     H(ZERO) = 1.;
0     H(1) = (B - A)/N;
0     TWOPI = 6.2831853071796;
0     L2 = C*LAMBDA(1) / 2.;
0     L3 = C*LAMBDA(1) / 3.;
0     L6 = C*LAMBDA(1) / 6.;
0     K(1) = H(1)*LAMBDA(1) / C;
0     C1 = 1. - L2;
0     C2 = 1. + L2;
0     C3 = 1. + 11.*L6;
0     FACTOR = 1.;
0     NSHL = 0;
0     NSHRT = 0;
0
0     "SET UP A PERMANENT EMPTY (NIL) REFINEMENT ON LEVEL NLEVEL+1."
0     MAKEAVAIL RFIN;
0     NLEVP1 = NLEVEL + 1;
0     NEW(P);
0     LEFTMOST(NLEVP1) = P;
0     WITH P <
1     ~BASE = MEMAVAIL + 1;
1     ~TOP = ~BASE + (EMPTY);
1     ~LEFT = 0;
1     ~RIGHT = EMPTY;
1     ~RLINK = NIL;
1     ~COARSE = NIL;
1     ~FINE = NIL;
1
1     "SET UP COARSE (1ST LEVEL) REFINEMENT."
```

```
1     NEW(P);
1     LEFTMOST(1) = P;
1     ^LEFT = 0;
1     ^RIGHT = 1;
1     ^TOP = N;
1     ^OLDTOP = ^TOP;
1     ^BASE = 0;
1     ^OLDBASE = ^BASE;
1     ^LLINK = NIL;
1     ^RLINK = NIL;
1     ^COARSE = NIL;
1     ^FINE = NIL;
1     (LEFTMOST(NLEVP1)^.LLINK) = P>
0
0     RIGHTB(1) = 1;
0     POWER(1) = 1;
0     S = C*LAMBDA(1);
0     BUFFER(1) = METHOD + 2 + TOL.FREQ(1)*S;
0     DO J = ZERO, MEMAVAIL <
1       DO I = 1, NCOMP <
2         VNEW(I, J) = 0.;
2         ESTERROR(I, J) = 0.>>
0
0     DO L = 2, NLEVEL <
1       IF L .EQ. 2 <RIGHTB(2) = N>
1       ELSE <RIGHTB(L) = XRATIO*RIGHTB(L-1)>
1       H(L) = H(L-1)/XRATIO;
1       K(L) = K(L-1)/TRATIO;
1       LAMBDA(L) = (LAMBDA(L-1)*XRATIO)/TRATIO;
1       LEFTMOST(L) = NIL;
1       S = XRATIO*S;
1       IF (TOL.FREQ(1) .EQ. 1 .AND. L .LE. TOL.FREQ(2))
1         S = S/TRATIO;
1       IF TOL.FREQ(1) .GT. 1 .AND. L .LT. TOL.FREQ(2) .OR.
1       TOL.FREQ(2) .EQ. 1 <
2         BUFFER(L) = 3 + S*TOL.FREQ(1)>
1       ELSE <
2         BUFFER(L) = 3 + S>
1       >
0     DO L = 1, NLEVEL <
1       C6(L) = C*H(L)**2*(1. - C*C*LAMBDA(L)**2)/6.;
1       C7(L) = (1. - C*C*LAMBDA(L)**2)/(3.*K(L));
1       C8(L) = 1./(6.*K(L));
1       C10(L) = 0.5*C*LAMBDA(L)*H(L)**2;
1       NPTSM(L+1) = 0;
1       POWER(L+1) = POWER(L)*TRATIO>
0
0     OUTPUT;('1SOLUTION OF SECOND-ORDER WAVE EQUATION WITH OPEN ',
0     'BOUNDARY CONDITIONS USING LAX-WENDROFF MESH REFINEMENT');
0     IF METHOD .EQ. 2 < OUTPUT;(' FOURTH ORDER (SPACE) ON COARSE MESH')>
0               ELSE <OUTPUT;(' LAX-WENDROFF ON COARSE MESH')>
0     IF RICHSN .EQ. 1 <
1       OUTPUT; (' ERROR ESTIMATION USING DIFFERENCES')>
0     ELSE IF RICHSN .EQ. 2 <
1       OUTPUT; (' ERROR ESTIMATION USING 2-LEVEL RICHARDSON '
1       'EXTRAPOLATION')>
0     ELSE <
1       OUTPUT; (' ERROR ESTIMATION USING 3-LEVEL RICHARDSON '
1       'EXTRAPOLATION')>
0     IF BDRY .EQ. 1 < OUTPUT; (' EXTRAPOLATION BOUNDARY CONDITIONS')>
0     ELSE IF BDRY .EQ. 2 <
1       OUTPUT; (' UPWIND B.C. WITH ESTIMATION BY DIFFERENCES')>
0     ELSE <
1       OUTPUT; (' UPWIND B.C. WITH ESTIMATION BY RICHARDSON EXTRAP.')>
0     OUTPUT N, H(1), K(1), LAMBDA(1), C; (/'   NO. OF INTERVALS ON ',
0     'COARSE MESH', I6, ' HCOARSE ', F8.5, ' KCOARSE ', F8.5, ' LAMBDA ',
0     F8.5, ' C', F10.2);
```

```
0     OUTPUT NLEVEL, XRATIO, TRATIO, TOLERANCE; (' HIGHEST LEVEL ',
0     'REFINEMENT', I4, ' H RATIO ', I3, ' K RATIO ', I3,
0     ' LOCAL TRUNCATION ERROR BOUND', 1PE15.7);
0     OUTPUT SKIPPR, DEBUG, NTIME, SCALE, QUADRAT, TOLCHK, TOL.FREQ;
0     (' SKIPPR =', I4, ' DEBUG =', I4, ' NTIME = ', I5,
0     ' SCALE =', E15.7,' QUADRATIC INTERPOLATION =', L2,
0     ' CHECK ERROR =', L2 / ' TOL FREQUENCY =', 2I3);
0     CALL DATE(DAY);
0     OUTPUT DAY; (' DATE = ', A9);
0     OUTPUT (BUFFER(J), J = 1, NLEVEL); (' BUFFER ', 20I5);
0     DO L = 1, NLEVEL <
1       LAMBDA(L) = C*LAMBDA(L)>
0   :EXIT:
0     RETURN;
0   END; "INITIALIZE"
0
0
0   SUBROUTINE INITIAL.CONDITIONS(BASE, V, VNEW);
0
0     "FIND EXACT SOLUTION AT FIRST TWO LEVELS"
0
0     USE COM3; USE STEPSZ; USE ZERO;
0     REAL TEMP(NCOMP), V(NCOMP, 1), VNEW(NCOMP, 1), X;
0     INTEGER BASE, I;
0
0     DO J = ZERO, N <
1       X = A + J*H(1);
1       CALL EXACT.SOLUTION(X, 0., TEMP);
1       DO I = 1, NCOMP <
2         VNEW(I, BASE+J) = TEMP(I)>
1       CALL EXACT.SOLUTION(X, K(1), TEMP);
1       DO I = 1, NCOMP <
2         V(I, BASE+J) = TEMP(I)>>
0     RETURN;
0   END; "INITIAL.CONDITIONS"
0
0
0   SUBROUTINE EXACT.SOLUTION(X, T, TEMP);
0   USE COM4;
0   REAL PHASE, T, TEMP(NCOMP), X;
0     PHASE = X - C*T + 0.5;
0     F = FACTOR*EXP(-SCALE*PHASE**2)+0.1*SIN(TWOPI*PHASE);
0     PHASE = X + C*T - 4.5;
0     G = -FACTOR*EXP(-SCALE*PHASE**2);
0     TEMP(1) = F + G;
0     TEMP(2) = G - F;
0     RETURN;
0   END;
0
0
0   SUBROUTINE UXX(X, T, TEMP);
0   USE COM4;
0   REAL F, G, PHASE, SP2, T, TEMP(NCOMP), X;
0     PHASE = X - C*T + 0.5;
0     SP2 = SCALE*PHASE*PHASE;
0     F = 2.*SCALE*FACTOR*EXP(-SP2)*(2.*SP2 - 1.)
0     - 0.1*(TWOPI**2) * SIN(TWOPI*PHASE);
0     PHASE = X + C*T - 4.5;
0     SP2 = SCALE*PHASE*PHASE;
0     G = -2.*SCALE*FACTOR*EXP(-SP2)*(2.*SP2 - 1.);
0     TEMP(1) = F + G;
0     TEMP(2) = G - F;
0     RETURN;
0   END;
0
0
0   SUBROUTINE UXXX(X, T, TEMP);
```

```
0  USE COM4;
0  REAL PHASE, SP2, T, TEMP(NCOMP), X;
0    PHASE = X - C*T + 0.5;
0    SP2 = SCALE*PHASE*PHASE;
0    F=FACTOR*EXP(-SP2)*4.*SCALE*SCALE*PHASE*(3.-2.*SP2)
0    - 0.1*(TWOPI**3) * COS(TWOPI*PHASE);
0    PHASE = X + C*T - 4.5;
0    SP2 = SCALE*PHASE*PHASE;
0    G = -FACTOR*EXP(-SP2)*4.*SCALE*SCALE*PHASE*(3.-2.*SP2);
0    TEMP(1) = F + G;
0    TEMP(2) = G - F;
0    RETURN;
0  END;
0
0
0  SUBROUTINE ADVANCE.SOLUTION;
0
0    "ADVANCE THE SOLUTION IN TIME.  USE THE EXACT SOLUTION AT
0    THE FIRST TIME LEVEL (NECESSARY FOR THE FOURTH ORDER METHOD, BUT
0    NOT NECESSARY FOR LAX WENDROFF, EXCEPT FOR COMPARISON PURPOSES)"
0
0    USE SOLN; USE LFTMST; USE COM5; USE STEPSZ; USE COUNT; USE DEBUG;
0    USE COM12; DEFINE RFIN;
0    REAL T, TIME;
0    INTEGER BOTTOM, HLEVEL, L, LM, LT, M, NTIME1, NEWHIL, SUM, TLEVEL,
0    TOP, TOTPTS, TOUT;
0    POINTER TO RFIN: P;
0
0    CALL INITIAL.CONDITIONS(0, V, VNEW);
0    T = K(1);
0    PRINT = SKIPPR .EQ. 1;
0    IF (DEBUG .GT. 0) CALL PLOT(1, T);
0    IF TOLCHK <TOP = NLEVEL> ELSE <TOP = NLEVEL - 1>
0    L = 1;
0    WHILE L .LE. TOP .AND. LEFTMOST(L) .NE. NIL <
1      P = LEFTMOST(L);
1      TOTPTS = 0;
1      REPEAT <
2        CALL ESTIMATE.ERROR(P, L, 1, T);
2        CALL DETERMINE.REFINEMENTS(P, L, 1, TOTPTS);
2        P = (P~.RLINK)>
1      UNTIL P .EQ. LEFTMOST(L+1);
1      NPTSM(L+1) = MAX0(NPTSM(L+1), TOTPTS);
1      L = L + 1>
0    "FIND LEVEL OF FINEST MESH"
0    HLEVEL = L - 1;
0    IF (LEFTMOST(L) .NE. NIL) HLEVEL = L;
0    NTIME1 = NTIME - 1;
0    DO TLEVEL = 1, NTIME1 <
1      "ADVANCE THE SOLUTION FROM T = K(1)*TLEVEL TO K(1)*(TLEVEL+1),
1      STARTING WITH THE HIGHEST LEVEL (FINEST) MESH."
1      PRINT = MOD(TLEVEL+1, SKIPPR) .EQ. 0 .OR. TLEVEL .EQ. NTIME1;
1      M = 1;
1      REPEAT <
2        L = HLEVEL;
2        REPEAT <
3          LT = M/POWER(HLEVEL+1-L);
3          LM = MOD(LT-1, TRATIO) + 1;
3          TIME = T + LT*K(L);
3          P = LEFTMOST(L);
3          REPEAT <
4            CALL ONE.TIME.STEP(P, L, LM, LAMBDA(L), TIME, 1, .FALSE.,
4            V, VNEW);
4            P = (P~.RLINK)>
3          UNTIL P .EQ. LEFTMOST(L+1);
3          CALL MOVE(L);
3          L = L - 1>
```

```
2         UNTIL MOD(M, POWER(HLEVEL+1-L)) .NE. 0;
2         L = L + 1;
2         IF L .EQ. 1 <
3           TOUT = TLEVEL + 1;
3           IF (PRINT) CALL PLOT(TOUT, TIME);
3           IF (TLEVEL .EQ. NTIME1) GO TO :FINI:>
2         ELSE <TOUT = TLEVEL>
2         IF TOL.FREQ(1) .EQ. 1 .AND. L .LE. TOL.FREQ(2) .OR.
2         TOL.FREQ(1) .GT. 1 .AND. L .EQ. 1 .AND. (MOD(TLEVEL,
2         TOL.FREQ(1)) .EQ. 0 .OR. TOL.FREQ(2) .NE. 1) <
3           "CHECK TRUNCATION ERROR INSIDE OR AT COARSE TIME STEP"
3           IF TOLCHK .AND. TOL.ACHIEVED <TOP = NLEVEL>
3                           ELSE <TOP = NLEVEL-1>
3           TOP = MIN0(TOP, HLEVEL);
3           IF (TOL.FREQ(1) .GT. 1 .AND. MOD(TLEVEL, TOL.FREQ(1))
3           .NE. 0) L = TOL.FREQ(2);
3           IF L .LE. TOP <
4             BOTTOM = L;
4             L = TOP;
4             WHILE L .GE. BOTTOM <
5               P = LEFTMOST(L);
5               TOTPTS = 0;
5               REPEAT <
6                 CALL ESTIMATE.ERROR(P, L, TOUT, TIME);
6                 CALL DETERMINE.REFINEMENTS(P, L, TOUT, TOTPTS);
6                 P = (P~.RLINK)>
5               UNTIL P .EQ. LEFTMOST(L+1);
5               NPTSM(L+1) = MAX0(NPTSM(L+1), TOTPTS);
5               L = L - 1>
4             "FIND LEVEL OF FINEST MESH"
4             NEWHIL = BOTTOM;
4             UNTIL LEFTMOST(NEWHIL+1) .EQ. NIL <
5               NEWHIL = NEWHIL + 1>
4             IF NEWHIL .GT. HLEVEL <
5               M = M * TRATIO**(NEWHIL - HLEVEL)>
4             ELSE IF NEWHIL .LT. HLEVEL <
5               M = M / TRATIO**(HLEVEL - NEWHIL)>
4             HLEVEL = NEWHIL>
3           >
2         M = M + 1>
1       WHILE M .LE. POWER(HLEVEL);
1
1       T = K(1) * (TLEVEL+1);
1       > "END OF ONE COARSE TIME STEP"
0
0   :FINI:
0     OUTPUT; (/' MAXIMUM STORAGE FOR SOLUTION VALUES (PER COMPONENT) ');
0     SUM = (LEFTMOST(1)~.TOP) + 1;
0     DO L = 2, NLEVEL <
1       TOP = NPTSM(L);
1       SUM = SUM + TOP;
1       OUTPUT L, TOP; (' LEVEL', 2I6)>
0     OUTPUT SUM; (' TOTAL', I5);
0     OUTPUT NSHL, NSHRT; (' MESH SHIFTED LEFT', I4, ' TIMES, RIGHT',
0     I4, ' TIMES');
0     RETURN;
0   END; "ADVANCE.SOLUTION"
0
0
0   SUBROUTINE ONE.TIME.STEP(P, L, LM, LAMBDAL, T, STENCIL, EXTRAP, V,
0   VNEW);
0     "ADVANCE THE SOLUTION ONE LEVEL L TIME STEP ON A LEVEL L REFINE-
0     MENT.  THESE REFINEMENTS ARE CHAINED TOGETHER USING THE RLINK
0     POINTERS.  WHILE ADVANCING REFINEMENT P, WE SKIP OVER ANY
0     REFINEMENTS OF P.  IF METHOD .EQ. 1, USE LAX WENDROFF, WITH FIRST
0     ORDER ACCURATE BOUNDARY APPROXIMATION, WHILE IF METHOD .EQ. 2,
0     USE OLIGER'S FOURTH ORDER METHOD IN SPACE (LEAP FROG IN TIME) WITH
```

```
0     THIRD ORDER ACCURATE BOUNDARY CONDITIONS.  AT INTERFACES, USE THE
0     'COARSE-FINE' LAX-WENDROFF APPROXIMATION.  THIS ROUTINE IS ALSO
0     USED TO ESTIMATE THE LOCAL TRUNCATION ERROR USING RICHARDSON
0     EXTRAPOLATION."
0
0     USE COM3; USE COM4; USE COM5; USE STEPSZ; USE COM7; USE ZERO;
0     USE XRATIO; USE COM12; USE METHOD; DEFINE RFIN;
0     REAL LAMBDAL, LAMBCF, LCF2, LF2, MULT, PHASE, T, V(NCOMP, 1),
0     VNEW(NCOMP, 1);
0     INTEGER BASE, BJ, I, J, L, LAST, LEFTR, LM, NPTSP2, OFFSET, PBASE,
0     RIGHTR, S, STENCIL, TOP, TS;
0     POINTER TO RFIN: P, PARENT, UP;
0     LOGICAL EXTRAP;
0
0     IF STENCIL .EQ. 2 <MULT = 1.> ELSE <MULT = 0.>
0     S = STENCIL;
0     TS = 2*STENCIL;
0     LF2 = LAMBDAL*0.5;
0     LAMBCF = LM*LAMBDAL/XRATIO;
0     LCF2 = LAMBCF*0.5;
0     OFFSET = XRATIO*(P~.LEFT);
0     BASE = (P~.BASE);
0     TOP = (P~.TOP);
0     NPTSP2 = TOP - BASE + 2;
0     LAST = TOP - BASE - S;
0     UP = (P~.FINE);
0     IF UP .EQ. NIL .OR. EXTRAP <
1       LEFTR = NPTSP2>
0     ELSE <
1       LEFTR = (UP~.LEFT) - 1 - OFFSET;
1       RIGHTR = (UP~.RIGHT) + 1 - OFFSET>
0     PARENT = (P~.COARSE);
0     PBASE = (PARENT~.BASE) - XRATIO*(PARENT~.LEFT);
0
0     "FOURTH ORDER METHOD - COARSEST MESH ONLY"
0     "NOT IMPLEMENTED FOR THIS PROBLEM"
0
0     "USE A SECOND ORDER LAX-WENDROFF METHOD IN THE INTERIOR
0     OF REFINEMENT P."
0     J = MIN0(S, LEFTR+1);
0     GO TO :L2:;
0     REPEAT <
1       BJ = BASE + J;
1       VNEW(1,BJ) = V(1,BJ) + LF2 * ((V(2,BJ+S) - V(2,BJ-S))
1       + LAMBDAL*(V(1,BJ+S) - 2.*V(1,BJ) + V(1,BJ-S))) - MULT*
1       VNEW(1,BJ);
1       VNEW(2,BJ) = V(2,BJ) + LF2 * ((V(1,BJ+S) - V(1,BJ-S))
1       + LAMBDAL*(V(2,BJ+S) - 2.*V(2,BJ) + V(2,BJ-S))) - MULT*
1       VNEW(2,BJ);
1       J = J + 1;
1   :L2:  IF J .EQ. LEFTR+1 <
2         "THE FOLLOWING SKIPS OVER ALL REFINEMENTS OF THIS
2         REFINEMENT."
2         J = RIGHTR;
2         UP = (UP~.RLINK);
2         IF (UP~.COARSE) .EQ. P <
3           LEFTR = (UP~.LEFT) - 1 - OFFSET;
3           RIGHTR = (UP~.RIGHT) + 1 - OFFSET>
2         ELSE <LEFTR = NPTSP2>>>
0     WHILE J .LE. LAST;
0
0     "IF THE LEFT EDGE OF REFINEMENT P TOUCHES THE LEFT BOUNDARY OF
0     THE REGION, USE REFLECTION FOR V AND UPWIND DIFFERENCING FOR W.
0     OTHERWISE, USE LAX-WENDROFF WITH COARSE SPACE STEP AND FINE
0     TIME STEP."
0     IF (P~.LEFT) .EQ. 0 <
1       IF BDRY .EQ. 1 <
```

```
2         VNEW(2,BASE) = 2.*VNEW(2,BASE+1) - VNEW(2,BASE+2)>
1       ELSE <
2         VNEW(2,BASE) = V(2,BASE) + LAMBDAL*(V(1,BASE+S) - V(1,BASE))
2         - MULT*VNEW(2,BASE)>
1       PHASE = A - C*T + 0.5;
1       VNEW(1,BASE) = VNEW(2,BASE) + 2.*FACTOR*EXP(-SCALE*PHASE**2) +
1       0.2*SIN(TWOPI*PHASE)>
0     ELSE <
1       J = PBASE + (P~.LEFT);
1       VNEW(1,BASE) = V(1,J) + LCF2*((V(2,J+1) - V(2,J-1)) +
1       LAMBCF*(V(1,J+1) - 2.*V(1,J) + V(1,J-1)));
1       VNEW(2,BASE) = V(2,J) + LCF2*((V(1,J+1) - V(1,J-1)) +
1       LAMBCF*(V(2,J+1) - 2.*V(2,J) + V(2,J-1)))>
0
0     "IF THE RIGHT END OF REFINEMENT P TOUCHES THE RIGHT EDGE
0     OF THE REGION, USE REFLECTION FOR V AND DOWNWIND DIFFERENCING
0     FOR W.  OTHERWISE USE COARSE-FINE LAX-WENDROFF."
0       IF (P~.RIGHT) .EQ. RIGHTB(L) <
1       IF BDRY .EQ. 1 <
2         VNEW(2,TOP) = 2.*VNEW(2,TOP-1) - VNEW(2,TOP-2)>
1       ELSE <
2         VNEW(2,TOP) = V(2,TOP) + LAMBDAL*(V(1,TOP)-V(1,TOP-S))
2         - MULT*VNEW(2,TOP)>
1       PHASE = B + C*T - 4.5;
1       VNEW(1,TOP) = -VNEW(2,TOP) - 2.*FACTOR*EXP(-SCALE*PHASE**2)>
0     ELSE <
1       J = PBASE + (P~.RIGHT);
1       VNEW(1,TOP) = V(1,J) + LCF2*((V(2,J+1) - V(2,J-1)) +
1       LAMBCF*(V(1,J+1) - 2.*V(1,J) + V(1,J-1)));
1       VNEW(2,TOP) = V(2,J) + LCF2*((V(1,J+1) - V(1,J-1)) +
1       LAMBCF*(V(2,J+1) - 2.*V(2,J) + V(2,J-1)))>
0     IF S .EQ. 2 <
1       DO I = 1, NCOMP <
2         VNEW(I,BASE+1) = 0.;
2         VNEW(I,TOP-1) = 0.>>
0
0     RETURN;
0   END; "ONE.TIME.STEP"
0
0
0   SUBROUTINE MOVE(L);
0     "MOVE SOLUTION VALUES FROM REFINEMENT(S) ON LEVEL L+1 (IF ANY) TO
0     THE CORRESPONDING POSITIONS ON LEVEL L.  ALSO MOVE SOLUTION VALUES
0     ON LEVEL L FROM VNEW TO V IN PREPARATION FOR NEXT TIME STEP."
0
0     USE SOLN; USE LFTMST; USE XRATIO; DEFINE RFIN;
0     REAL TEMP;
0     INTEGER BC, BF, I, J, L, LEFT, PBASE, RIGHT, TOP;
0     POINTER TO RFIN: P, PARENT;
0
0     PARENT = LEFTMOST(L);
0     REPEAT <
1       PBASE = (PARENT~.BASE);
1       TOP = (PARENT~.TOP);
1       P = (PARENT~.FINE);
1       WHILE (P~.COARSE) .EQ. PARENT <
2         LEFT = (P~.LEFT);
2         RIGHT = (P~.RIGHT);
2         BC = PBASE - XRATIO*(PARENT~.LEFT) + LEFT;
2         BF = (P~.BASE);
2         DO J = LEFT, RIGHT <
3           DO I = 1, NCOMP <
4             VNEW(I, BC) = V(I, BF)>
3           BC = BC + 1;
3           BF = BF + XRATIO>
2         P = (P~.RLINK)>
1
```

```
1       "SHIFT MESH VALUES FOR NEXT TIME STEP"
1       DO J = PBASE, TOP <
2         DO I = 1, NCOMP <
3           V(I,J) = VNEW(I,J)>>
1
1       PARENT = (PARENT~.RLINK)>
0     UNTIL PARENT .EQ. LEFTMOST(L+1);
0
0     RETURN;
0   END; "MOVE"
0
0
0   SUBROUTINE ESTIMATE.ERROR(P, L, TLEVEL, T);
0     "ESTIMATE LOCAL TRUNCATION ERROR."
0
0     USE COM12;  REAL T;
0     INTEGER L, TLEVEL;
0     POINTER TO RFIN: P;
0
0     GO TO (:DIFF:, :RICH2:, :RICH3:), RICHSN;
0   :DIFF:  CALL DIFFERENCES(P, L, TLEVEL, T);
0         GO TO :OUT:;
0
0   :RICH2: CALL RICHARDSON2(P, L, TLEVEL, T);
0         GO TO :OUT:;
0
0   :RICH3: CALL RICHARDSON3(P, L, TLEVEL, T);
0
0   :OUT: RETURN;
0     END;
0
0
0   SUBROUTINE DIFFERENCES(P, L, TLEVEL, T);
0     "ESTIMATE LOCAL TRUNCATION ERROR USING DIFFERENCES"
0
0     USE COM3; USE COM5; USE STEPSZ; USE ZERO; USE DEBUG; USE COM12;
0     USE COM14; USE ERROR; DEFINE RFIN;
0     REAL EXACT(10), T, TEMP(NCOMP), XLEFT;
0     INTEGER BASE, I, J, L, M, NM1, NPTS, TLEVEL, TOP;
0     POINTER TO RFIN: P, PARENT;
0     LOGICAL XPRINT;
0
0     WITH P <
1     XPRINT = PRINT .AND. MOD(DEBUG, 4)/2 .EQ. 1;
1     CALL DIFFERENCES2(P, L);
1     PARENT = ~COARSE;
1     BASE = ~BASE;
1     TOP = ~TOP;
1     NPTS = TOP - BASE;
1     NM1 = NPTS - 1;
1     XLEFT = A + H(L-1)*~LEFT;
1     "COMPONENTS OF ESTIMATED ERROR ARE INTERCHANGED."
1     DO J = 1, NM1 <
2       DO I = 1, NCOMP <
3         ESTERROR(I,BASE+J) = C6(L)*ESTERROR(I,BASE+J)>>
1     IF ~LEFT .EQ. 0 < "LEFT BOUNDARY"
2       ESTERROR(1,BASE) = C10(L)*(LAMBDA(L)*ESTERROR(2,BASE)
2       - ESTERROR(1,BASE));
2       ESTERROR(2,BASE) = 0.>
1     ELSE <
2       DO I = 1, NCOMP <
3         ESTERROR(I,BASE) = 0.>>
1     IF ~RIGHT .EQ. RIGHTB(L) < "RIGHT BOUNDARY"
2       ESTERROR(1,TOP) = C10(L)*(LAMBDA(L)*ESTERROR(2,TOP)
2       + ESTERROR(1,TOP));
2       ESTERROR(2,TOP) = 0.>
1     ELSE <
```

```
2       DO I = 1, NCOMP <
3         ESTERROR(I,TOP) = 0.>>
1     IF XPRINT <
2       DO I = 1, NCOMP <
3       OUTPUT I, TLEVEL, P, L; (/' ESTIMATED LOCAL TRUNCATION ',
3       'ERROR OF V(', I1, ') AT T =', I5, ' DELTA T, REFINEMENT',
3       I3, ' LEVEL', I4);
3       OUTPUT (ESTERROR(I,J), J = BASE, TOP); (1X, 1P10E12.4);
3       OUTPUT; (' LOCAL TRUNCATION ERROR USING EXACT DERIVATIVES');
3       DO J = ZERO, NPTS <
4         M = MOD(J, 10) + 1;
4         CALL UXXX(XLEFT + J*H(L), T, TEMP);
4         EXACT(M) = C6(L)*TEMP(3-I);
4         IF J .EQ. 0 <
5           IF I .EQ. 1 <EXACT(M) = 0.>
5           ELSE <
6             CALL UXX(A, T, TEMP);
6             EXACT(M) = C10(L)*(LAMBDA(L)*TEMP(2) -
6             TEMP(1))>>
4         IF M .EQ. 10 .OR. J .EQ. NPTS <
5           IF J .EQ. NPTS .AND. ~RIGHT .EQ. RIGHTB(L) <
6             IF I .EQ. 1 <EXACT(M) = 0.>
6             ELSE <
7               CALL UXX(B, T, TEMP);
7               EXACT(M) = C10(L)*(LAMBDA(L)*TEMP(2) +
7               TEMP(1))>>
5           OUTPUT (EXACT(M2), M2 = 1, M); (1X, 1P10E12.4)>>>>>
0
0     RETURN;
0   END; "DIFFERENCES"
0
0
0   SUBROUTINE DIFFERENCES2(P, L);
0     "COMPUTE A DIFFERENCE APPROXIMATION TO UXXX AT EACH INTERIOR POINT
0     ON THE LEVEL L MESH, AND TO UXX AT BOUNDARIES."
0
0     USE SOLN; USE COM5; USE STEPSZ; USE ERROR; DEFINE RFIN;
0     REAL D3VDX(NCOMP, 1), HCUBE4, HSQ;
0     INTEGER BASE, BASEP2, BJ, I, L, TOP, TOPM2;
0     POINTER TO RFIN: P;
0     EQUIVALENCE (ESTERROR, D3VDX);
0
0     HSQ = H(L)**2;
0     HCUBE4 = 4.*H(L)**3;
0     BASE = (P~.BASE);
0     BASEP2 = BASE + 2;
0     TOP = (P~.TOP);
0     TOPM2 = TOP - 2;
0     DO I = 1, NCOMP <
1       DO BJ = BASEP2, TOPM2 <
2         D3VDX(I,BJ) = 2.*(-V(I,BJ-2) + 2.*(V(I,BJ-1) - V(I,BJ+1))
2         + V(I,BJ+2))/HCUBE4>
1
1       IF (P~.LEFT) .EQ. 0 <
2         D3VDX(I,BASE) = (2.*V(I,BASE) - 5.*V(I,BASE+1) + 4.*
2         V(I,BASEP2) - V(I,BASE+3))/HSQ;
2         IF (L .GT. 1) D3VDX(I,BASEP2) = 0.>
1       D3VDX(I,BASE+1) = 0.;
1       D3VDX(I,TOP-1) = 0.;
1       IF (P~.RIGHT) .EQ. RIGHTB(L) <
2         D3VDX(I,TOP) = (-V(I,TOP-3) + 4.*V(I,TOPM2) -5.*V(I,TOP-1)
2         + 2.*V(I,TOP))/HSQ;
2         IF (L .GT. 1) D3VDX(I,TOPM2) = 0.>>
0
0     RETURN;
0   END; "DIFFERENCES2"
0
```

```
0
0  SUBROUTINE RICHARDSON2(P, L, TLEVEL, T);
0    "ESTIMATE LOCAL TRUNCATION ERROR USING RICHARDSON EXTRAPOLATION.
0    THIS METHOD USES TWO TIME STEPS OF LENGTH DELTA T AND IS SUITABLE
0    ONLY WHEN T DERIVATIVES IN THE DIFFERENTIAL EQUATION CAN BE
0    REWRITTEN IN TERMS OF X DERIVATIVES."
0
0    USE SOLN; USE COM3; USE COM5; USE STEPSZ; USE ZERO; USE DEBUG;
0    USE COM12; USE COM14; USE ERROR; DEFINE RFIN;
0    REAL EXACT(10), NEXTT, T, TEMP(NCOMP), XLEFT;
0    INTEGER BASE, I, J, L, M, M2, NM1, NPTS, TLEVEL, TOP;
0    POINTER TO RFIN: P, PARENT;
0    LOGICAL XPRINT;
0
0    WITH P <
1    NEXTT = T + K(L);
1    CALL ONE.TIME.STEP(P, L, 1, LAMBDA(L), NEXTT, 1, .TRUE., V,
1    ESTERROR);
1    CALL ONE.TIME.STEP(P, L, 1, 0.5*LAMBDA(L), NEXTT, 2, .TRUE., V,
1    ESTERROR);
1
1    XPRINT = PRINT .AND. MOD(DEBUG, 4)/2 .EQ. 1;
1    PARENT = ~COARSE;
1    BASE = ~BASE;
1    TOP = ~TOP;
1    NPTS = TOP - BASE;
1    NM1 = NPTS - 1;
1    XLEFT = A + H(L-1)*~LEFT;
1    DO I = 1, NCOMP <
2      DO J = 1, NM1 <
3        ESTERROR(I,BASE+J) = C7(L)*ESTERROR(I,BASE+J)>>
1    IF ~LEFT .EQ. 0 < "LEFT BOUNDARY"
2      ESTERROR(1,BASE) = 0.;
2      ESTERROR(2,BASE) = ESTERROR(2,BASE)/3.>
1    ELSE <
2      DO I = 1, NCOMP <
3        ESTERROR(I,BASE) = 0.>>
1    IF ~RIGHT .NE. RIGHTB(L) <
2      DO I = 1, NCOMP <
3        ESTERROR(I,TOP) = 0.>>
1    ELSE <  "RIGHT BOUNDARY"
2      ESTERROR(1,TOP) = 0.;
2      ESTERROR(2,TOP) = ESTERROR(2,TOP)/3.>
1    IF XPRINT <
2      DO I = 1, NCOMP <
3      OUTPUT I, TLEVEL, P, L; (/' ESTIMATED LOCAL TRUNCATION ',
3      'ERROR OF V(', I1, ') AT T =', I5, ' DELTA T, REFINEMENT',
3      I3, ' LEVEL', I4);
3      OUTPUT (ESTERROR(I,J), J = BASE, TOP); (1X, 1P10E12.4);
3      OUTPUT; (' LOCAL TRUNCATION ERROR USING EXACT DERIVATIVES');
3      DO J = ZERO, NPTS <
4        M = MOD(J, 10) + 1;
4        CALL UXXX(XLEFT + J*H(L), T, TEMP);
4        EXACT(M) = C6(L)*TEMP(3-I);
4        IF J .EQ. 0 <
5          IF I .EQ. 1 <EXACT(M) = 0.>
5          ELSE <
6            CALL UXX(A, T, TEMP);
6            EXACT(M) = C10(L)*(LAMBDA(L)*TEMP(2) -
6            TEMP(1))>>
4        IF M .EQ. 10 .OR. J .EQ. NPTS <
5          IF J .EQ. NPTS .AND. ~RIGHT .EQ. RIGHTB(L) <
6            IF I .EQ. 1 <EXACT(M) = 0.>
6            ELSE <
7              CALL UXX(B, T, TEMP);
7              EXACT(M) = C10(L)*(LAMBDA(L)*TEMP(2)
7              + TEMP(1))>>
```

```
5          OUTPUT (EXACT(M2), M2 = 1, M); (1X, 1P10E12.4)>>>>>
0
0    RETURN;
0  END; "RICHARDSON2"
0
0
0  SUBROUTINE RICHARDSON3(P, L, TLEVEL, T);
0    "ESTIMATE LOCAL TRUNCATION ERROR USING RICHARDSON EXTRAPOLATION.
0    THIS METHOD USES TWO TIME STEPS OF LENGTH DELTA T (ON THE
0    APPROPRIATE MESH) AND ONE OF LENGTH 2 DELTA T.  IT IS THUS SUITABLE
0    FOR ANY SYSTEM OF EQUATIONS."
0
0    USE SOLN; USE COM3; USE COM5; USE STEPSZ; USE ZERO; USE DEBUG;
0    USE COM12; USE COM14; USE ERROR; DEFINE RFIN;
0    REAL EXACT(10), NEXTT, T, TEMP(NCOMP), XLEFT;
0    INTEGER BASE, I, J, L, M, M2, NM1, NPTS, TLEVEL, TOP;
0    POINTER TO RFIN: P, PARENT;
0    LOGICAL XPRINT;
0
0    WITH P <
1    NEXTT = T + 2.*K(L);
1    CALL ONE.TIME.STEP(P, L, 1, LAMBDA(L), T+K(L), 1, .TRUE., V, VNEW);
1    CALL ONE.TIME.STEP(P, L, 1, LAMBDA(L), NEXTT, 1, .TRUE., VNEW,
1    ESTERROR);
1    CALL ONE.TIME.STEP(P, L, 1, LAMBDA(L), NEXTT, 2, .TRUE., V,
1    ESTERROR);
1
1    XPRINT = PRINT .AND. MOD(DEBUG, 4)/2 .EQ. 1;
1    PARENT = ~COARSE;
1    BASE = ~BASE;
1    TOP = ~TOP;
1    NPTS = TOP - BASE;
1    NM1 = NPTS - 1;
1    XLEFT = A + H(L-1)*~LEFT;
1    DO J = 1, NM1 <
2      DO I = 1, NCOMP <
3        ESTERROR(I,BASE+J) = C8(L)*ESTERROR(I,BASE+J)>>
1    IF ~LEFT .EQ. 0 < "LEFT BOUNDARY"
2      ESTERROR(1,BASE) = 0.;
2      IF BDRY .NE. 3 <
3        ESTERROR(2,BASE) = 2.*V(2,BASE) - 5.*V(2,BASE+1) + 4.*
3        V(2,BASE+2) - V(2,BASE+3)>
2      IF BDRY .EQ. 2 <
3        ESTERROR(2,BASE) = 0.5*LAMBDA(L)*(LAMBDA(L)*ESTERROR(2,BASE)-
3        2.*V(1,BASE) + 5.*V(1,BASE+1) - 4.*V(1,BASE+2) + V(1,BASE+3))>
2      ELSE IF BDRY .EQ. 3 <
3        ESTERROR(2,BASE) = ESTERROR(2,BASE)/(2.+LAMBDA(L))>
2      IF L .GT. 1 <
3        ESTERROR(1,BASE+2) = 0.;
3        ESTERROR(2,BASE+2) = 0.>>
1    ELSE <
2      DO I = 1, NCOMP <
3        ESTERROR(I,BASE) = 0.>>
1    IF ~RIGHT .NE. RIGHTB(L) <
2      DO I = 1, NCOMP <
3        ESTERROR(I,TOP) = 0.>>
1    ELSE <  "RIGHT BOUNDARY"
2      IF L .GT. 1 <
3        ESTERROR(1,TOP-2) = 0.;
3        ESTERROR(2,TOP-2) = 0.>
2      ESTERROR(1,TOP) = 0.;
2      IF BDRY .NE. 3 <
3        ESTERROR(2,TOP) = 2.*V(2,TOP) - 5.*V(2,TOP-1) + 4.*
3        V(2,TOP-2) - V(2,TOP-3)>
2      IF BDRY .EQ. 2 <
3        ESTERROR(2,TOP) = 0.5*LAMBDA(L)*(LAMBDA(L)*ESTERROR(2,TOP) +
3        2.*V(1,TOP) - 5.*V(1,TOP-1) + 4.*V(1,TOP-2) - V(1,TOP-3))>
```

```
2       ELSE IF BDRY .EQ. 3 <
3         ESTERROR(2,TOP) = ESTERROR(2,TOP)/(2.+LAMBDA(L))>>
1     IF XPRINT <
2       DO I = 1, NCOMP <
3       OUTPUT I, TLEVEL, P, L; (/' ESTIMATED LOCAL TRUNCATION ',
3       'ERROR OF V(', I1, ') AT T =', I5, ' DELTA T, REFINEMENT',
3       I3, ' LEVEL', I4);
3       OUTPUT (ESTERROR(I,J), J = BASE, TOP); (1X, 1P10E12.4);
3       OUTPUT; (' LOCAL TRUNCATION ERROR USING EXACT DERIVATIVES');
3       DO J = ZERO, NPTS <
4         M = MOD(J, 10) + 1;
4         CALL UXXX(XLEFT + J*H(L), T, TEMP);
4         EXACT(M) = C6(L)*TEMP(3-I);
4         IF J .EQ. 0 <
5           IF I .EQ. 1 <EXACT(M) = 0.>
5           ELSE <
6             CALL UXX(A, T, TEMP);
6             EXACT(M) = 2.*C10(L)*TEMP(2)>>
4         IF M .EQ. 10 .OR. J .EQ. NPTS <
5           IF J .EQ. NPTS .AND. ~RIGHT .EQ. RIGHTB(L) <
6             IF I .EQ. 1 <EXACT(M) = 0.>
6             ELSE <
7               CALL UXX(B, T, TEMP);
7               EXACT(M) = 2.*C10(L)*TEMP(2)>>
5           OUTPUT (EXACT(M2), M2 = 1, M); (1X, 1P10E12.4)>>>>>
0
0     RETURN;
0   END; "RICHARDSON3"
0
0
0   SUBROUTINE DETERMINE.REFINEMENTS(PARENT, L, TLEVEL, TOTPTS);
0
0       "DETERMINE WHERE TO REFINE THE LEVEL L REFINEMENTS.
0     REFINE THEM WHENEVER THE INTERIOR LOCAL TRUNCATION ERROR
0     PER UNIT TIME STEP, OR THE BOUNDARY LOCAL TRUNCATION ERROR
0     (BOTH PROPORTIONAL TO H*H) IS GREATER THAN TOLERANCE.
0     THE DESIRED INTERVAL(S) OF REFINEMENT WILL EXTEND FROM THE
0     LEFTN(J)-TH MESH POINT TO THE RIGHTN(J)-TH POINT (RELATIVE TO
0     THE LEFT SIDE OF THE REGION) ON THE PARENT REFINEMENT,
0     J = 1, ..., NRFIN."
0
0     $ 'MAX.INTERVALS' = '10'
0     USE COM5; USE XRATIO; USE DEBUG; USE COM12; USE ERROR; DEFINE RFIN;
0     INTEGER BASE, GAP, I, J, L, LEFT, LEFTN(MAX.INTERVALS), LP1, N,
0     NRFIN, OFFSET, RIGHTN(MAX.INTERVALS), RT, TLEVEL, TOTPTS;
0     POINTER TO RFIN: P, PARENT;
0     LOGICAL COND;
0
0     GAP = BUFFER(L) + 2;
0     LP1 = L + 1;
0     OFFSET = XRATIO*(PARENT~.LEFT);
0     BASE = (PARENT~.BASE) - OFFSET;
0     N = (PARENT~.TOP) - BASE;
0     NRFIN = 0;
0     LEFT = OFFSET;
0     WHILE LEFT .LE. N <
1       REPEAT <
2         COND = .FALSE.;
2         DO I = 1, NCOMP <
3           COND = COND .OR. ABS(ESTERROR(I,BASE+LEFT)) .GT.
3           TOLERANCE>
2         IF (COND) GO TO :BEGIN INTERVAL:;
2         LEFT = LEFT + 1;>
1       WHILE LEFT .LE. N;
1       EXIT;
1   :BEGIN INTERVAL:
1       RT = LEFT;
```

```
1       IF NRFIN .GE. 1 <IF RIGHTN(NRFIN) + GAP .GE. LEFT<
3         LEFT = LEFTN(NRFIN) + BUFFER(L);
3         NRFIN = NRFIN - 1>>
1       REPEAT <
2         COND = .FALSE.;
2         DO I = 1, NCOMP <
3           COND = COND .OR. ABS(ESTERROR(I,BASE+RT)) .GT.
3           TOLERANCE>
2         IF (.NOT. COND) EXIT;
2         RT = RT + 1;>
1       UNTIL RT .GT. N;
1       RT = RT - 1;
1       NRFIN = NRFIN + 1;
1       IF NRFIN .GT. MAX.INTERVALS <
2         OUTPUT TLEVEL, LP1; (' TOO MANY REFINEMENTS', 2I7);
2         STOP>
1       LEFTN(NRFIN) = MAX0(OFFSET, LEFT - BUFFER(L));
1       IF (LEFTN(NRFIN) .EQ. OFFSET + 1) LEFTN(NRFIN) = OFFSET;
1       RIGHTN(NRFIN) = MIN0(N, RT + BUFFER(L));
1       IF (RIGHTN(NRFIN) .EQ. N-1) RIGHTN(NRFIN) = N;
1       LEFT = RT + 2>
0
0     IF DEBUG .GT. 0 <
1       IF NRFIN .EQ. 0 <
2         OUTPUT TLEVEL, L, PARENT;
2         (' TLEVEL, LEVEL, REF, NO REFINEMENTS  ', I5, 2I3);>
1       ELSE <
2       OUTPUT TLEVEL, L, PARENT, (LEFTN(J), RIGHTN(J), J = 1, NRFIN);
2       (' TLEVEL, LEVEL, REF, N(LEFTN, RIGHTN)', I5, 2I3, 12I5);>>
0
0     IF NRFIN .EQ. 0 <
1       IF ((PARENT~.FINE) .NE. NIL) GO TO :ALPHA:>
0     ELSE <
1       IF ((LEFTN(1) .EQ. OFFSET .AND. (PARENT~.LEFT) .NE. 0) .OR.
1       (RIGHTN(NRFIN) .EQ. N .AND. (PARENT~.RIGHT) .NE. RIGHTB(L)))
1       OUTPUT TLEVEL, LP1, PARENT;
1       (' TLEVEL', I5, ' DANGER, LEVEL', I4, ' REFINEMENT ABUTS '
1       'LEFT OR RIGHT SIDE OF REFINEMENT', I4);
1       IF L .EQ. NLEVEL <
2         OUTPUT TLEVEL, PARENT; (' TLEVEL', I5, ' REFINEMENT',
2         I3, ' DANGER, LOCAL ERROR TOLERANCE NOT ACHIEVED.');
2         TOL.ACHIEVED = .FALSE.>
1       ELSE <
2   :ALPHA:    CALL ADJUST.MESH(PARENT, NRFIN, LP1, TLEVEL, LEFTN,
2         RIGHTN, TOTPTS)>>
0
0     RETURN;
0   END; "DETERMINE.REFINEMENTS"
0
0
0   SUBROUTINE ADJUST.MESH(PARENT, NRFIN, L, TLEVEL, LEFTN, RIGHTN,
0   TOTPTS);
0       "IF THE DESIRED L-TH LEVEL INTERVALS PRODUCED BY DETERMINE.
0     REFINEMENTS DIFFER FROM THE EXISTING REFINEMENTS OF THE
0     (L-1)-ST LEVEL PARENT, WE MUST ADJUST THE REFINEMENTS.
0     THIS IS DONE IN A SINGLE LEFT-TO-RIGHT SCAN OF THESE
0     REFINEMENTS.  THIS MAY INVOLVE CREATION, DELETION, SEPARATING
0     OR MERGING OF REFINEMENTS.  USUALLY, HOWEVER, IT
0     INVOLVES ONLY THE MANIPULATION OF THE INDICES LEFT, RIGHT,
0     BASE, AND TOP BELONGING TO A REFINEMENT.  ONLY SELDOM ARE
0     SOLUTION VALUES ACTUALLY MOVED IN MEMORY."
0
0     USE SOLN; USE COM5; USE XRATIO; USE DEBUG; USE COM12; DEFINE RFIN;
0     INTEGER BASE, I, J, L, LEFTN(1), M, NPTS, NRFIN, RIGHTN(1), TLEVEL,
0     TOP, TOTPTS;
0     POINTER TO RFIN: P, PARENT, Q, UP;
0     LOGICAL COND;
```

```
0
0     WITH P <
1     P = (PARENT~.FINE);
1     FOR J = 1 TO NRFIN <
2       UNTIL LEFTN(J) .LE. ~RIGHT .OR. ~COARSE .NE. PARENT <
3         CALL DELETE(P, L);
3         P = ~RLINK;>
2       IF RIGHTN(J) .LE. ~LEFT .OR. ~COARSE .NE. PARENT <
3         CALL CREATE(PARENT, P, L, LEFTN(J), RIGHTN(J), 0, P)>
2       IF RIGHTN(J) .LT. ~RIGHT <
3         Q = ~RLINK;
3         IF (Q~.COARSE) .EQ. PARENT .AND. J .NE. NRFIN <
4         IF LEFTN(J+1) .LT. ~RIGHT .AND. RIGHTN(J+1) .GT. (Q~.LEFT) <
5           "THE REFINEMENT Q TO THE RIGHT HAS MOVED LEFT."
5           CALL FILL.IN(P, Q, L, TLEVEL);
5           CALL SEPARATE.REF(P, Q, RIGHTN(J));
5           (Q~.BASE) = (Q~.BASE) - XRATIO*((Q~.LEFT) - LEFTN(J+1));
5           (Q~.LEFT) = LEFTN(J+1)>>
3         IF J .NE. NRFIN <
4         IF (RIGHTN(J+1) .LT. (Q~.LEFT) .OR. (Q~.COARSE) .NE. PARENT)
4         .AND. LEFTN(J+1) .LT. ~RIGHT <
5           "SEPARATE REFINEMENTS"
5           BASE = ~BASE + XRATIO*(LEFTN(J+1) - ~LEFT);
5           CALL CREATE(PARENT, Q, L, LEFTN(J+1), ~RIGHT, BASE, Q);
5           (Q~.RIGHT) = ~RIGHT;
5           (Q~.TOP) = ~TOP;
5           CALL SEPARATE.REF(P, Q, RIGHTN(J))>>
3
3         "DELETE RIGHT END OF REFINEMENT P"
3         ~RIGHT = RIGHTN(J);
3         ~TOP = ~BASE + XRATIO*(~RIGHT - ~LEFT)>
2
2       IF LEFTN(J) .NE. ~LEFT <
3         IF LEFTN(J) .GT. ~LEFT <
4           "DELETE LEFT END OF REFINEMENT P."
4           ~BASE = ~BASE + XRATIO*(LEFTN(J) - ~LEFT);
4           ~LEFT = LEFTN(J)>
3         ELSE
3           <CALL EXTEND.LEFT(P, LEFTN(J), L, TLEVEL)>>
2
2       IF RIGHTN(J) .GT. ~RIGHT <
3         Q = ~RLINK;
3         WHILE (Q~.COARSE) .EQ. PARENT .AND. RIGHTN(J) .GT.
3         (Q~.LEFT) <
4           CALL FILL.IN(P, Q, L, TLEVEL);
4           ~RIGHT = MIN0(RIGHTN(J), (Q~.RIGHT));
4           ~TOP = ~BASE + XRATIO*(~RIGHT - ~LEFT);
4           IF ~RIGHT .EQ. (Q~.RIGHT) <
5             CALL MERGE(P, Q);
5             Q = (Q~.RLINK)>
4           ELSE <
5             IF J .NE. NRFIN <COND = LEFTN(J+1) .GT. (Q~.RIGHT)>
5                     ELSE <COND = .TRUE.>
5             IF COND <CALL MERGE(P, Q)>
5             ELSE <  "REFINEMENTS P AND Q HAVE MOVED RIGHT.  CHECK
6                   IF SOME OF Q'S REFINEMENTS NOW BELONG TO P."
6               UP = (Q~.FINE);
6               IF UP .NE. NIL <
7                 WHILE (UP~.COARSE) .EQ. Q .AND. (UP~.RIGHT) .LE.
7                 XRATIO* ~RIGHT <
8                   (UP~.COARSE) = P;
8                   UP = (UP~.RLINK)>
7                 IF (UP .NE. (Q~.FINE) .AND. ~FINE .EQ. NIL)
7                 ~FINE = (Q~.FINE);
7                 IF (UP~.COARSE) .EQ. Q <(Q~.FINE) = UP>
7                           ELSE <(Q~.FINE) = NIL>>>
5             GO TO :EXIT:>>
```

```
3
3         CALL EXTEND.RIGHT(P, RIGHTN(J), L, TLEVEL, .TRUE.)>
2   :EXIT:
2       NPTS = XRATIO*(~RIGHT - ~LEFT);
2       ~TOP = ~BASE + NPTS;
2       TOTPTS = TOTPTS + NPTS + 1;
2
2       IF PRINT .AND. DEBUG/8 .GE. 1 <
3         OUTPUT TLEVEL, P, L; (/' TLEVEL', I5, ' REFINEMENT', I4,
3         ' LEVEL', I4);
3         BASE = ~BASE;
3         TOP = ~TOP;
3         OUTPUT ~RLINK, ~LLINK, ~FINE, ~COARSE, BASE, TOP, ~LEFT,
3         ~RIGHT; (' RLINK, LLINK, FINE, COARSE, BASE, TOP, LEFT, ',
3         'RIGHT' / 8I5);
3         DO I = 1, NCOMP <
4           OUTPUT I; (' V(', I1, ')');
4           OUTPUT (V(I, M), M = BASE, TOP); (1P8E15.7)>>
2       P = ~RLINK;>
1
1     UNTIL ~COARSE .NE. PARENT <
2       CALL DELETE(P, L);
2       P = ~RLINK;>
1     >
0     RETURN;
0   END; "ADJUST.MESH"
0
0
0   SUBROUTINE SEPARATE.REF(P, Q, PRIGHT);
0     "GIVEN TWO REFINEMENTS P AND Q, WITH P TO THE LEFT OF Q, WHICH MAY
0     HAVE MOVED OR BEEN SEPARATED.  CHECK IF SOME OF P'S REFINEMENTS NOW
0     BELONG TO Q."
0
0     USE XRATIO; DEFINE RFIN;
0     INTEGER PRIGHT;
0     POINTER TO RFIN: P, Q, UP;
0
0     UP = (P~.FINE);
0     IF UP .NE. NIL <
1       WHILE (UP~.COARSE) .EQ. P .AND. (UP~.RIGHT) .LE. XRATIO*PRIGHT <
2         UP = (UP~.RLINK)>
1       IF (UP .EQ. (P~.FINE)) (P~.FINE) = NIL;
1       IF ((UP~.COARSE) .EQ. P) (Q~.FINE) = UP;
1       WHILE (UP~.COARSE) .EQ. P <
2         (UP~.COARSE) = Q;
2         UP = (UP~.RLINK)>>
0
0     RETURN;
0   END; "SEPARATE.REF"
0
0
0   SUBROUTINE MERGE(P, Q);
0     "MERGE TWO ADJACENT REFINEMENTS POINTED TO BY P AND Q.  THE P
0     REFINEMENT IS ASSUMED TO BE SPATIALLY TO THE LEFT OF Q."
0
0     USE DEBUG; DEFINE RFIN;
0     POINTER TO RFIN: P, Q, R, UP;
0
0     "CHAIN TOGETHER THE REFINEMENTS OF P AND THE REFINEMENTS OF Q."
0     UP = (Q~.FINE);
0     IF UP .NE. NIL <
1       IF ((P~.FINE) .EQ. NIL) (P~.FINE) = (Q~.FINE);
1       WHILE (UP~.COARSE) .EQ. Q <
2         (UP~.COARSE) = P;
2         UP = (UP~.RLINK)>>
0
0     "UNCHAIN THE RECORD FOR THE Q REFINEMENT."
```

```
0     R = (Q~.RLINK);
0     (P~.RLINK) = R;
0     (R~.LLINK) = P;
0     DISPOSE(Q);
0
0     WITH P <
1     IF (DEBUG .GT. 0) OUTPUT Q, P, ~LEFT, ~RIGHT, ~FINE, ~BASE, ~TOP;
1     (' DELETE', I4, ' MERGE', I4, ' LEFT, RIGHT, FINE, BASE, TOP',
1     5I5);>
0     RETURN;
0   END; "MERGE"
0
0
0   SUBROUTINE FILL.IN(P, Q, L, TLEVEL);
0     "FILL IN THE AREA BETWEEN TWO REFINEMENTS P AND Q, BY INTERPOLATION."
0
0     USE SOLN; DEFINE RFIN;
0     INTEGER BASE, DELTA, I, J, L, TLEVEL, TOP;
0     POINTER TO RFIN: P, Q;
0
0     CALL EXTEND.RIGHT(P, (Q~.LEFT), L, TLEVEL, .FALSE.);
0     BASE = (Q~.BASE);
0     IF BASE .NE. (P~.TOP) <
1       "SHIFT Q MESH VALUES DOWN."
1       TOP = (Q~.TOP);
1       DELTA = BASE - (P~.TOP);
1       DO I = 1, NCOMP <
2         DO J = BASE, TOP <
3           V(I, J-DELTA) = V(I, J)>>
1       (Q~.BASE) = (P~.TOP);
1       (Q~.TOP) = (Q~.TOP) - DELTA>
0
0      RETURN;
0   END;  "FILL.IN"
0
0
0   SUBROUTINE CREATE(PARENT, Q, LEVEL, LEFTN, RIGHTN, BASE, NEW);
0     "CREATE A NEW REFINEMENT AT LEVEL LEVEL, WITH PARENT POINTED TO BY
0     PARENT.  INSERT IT TO THE LEFT OF REFINEMENT Q.  RETURN THE POINTER
0     'NEW' TO IT."
0
0     USE LFTMST; USE XRATIO; USE DEBUG; DEFINE RFIN;
0     INTEGER BASE, LEVEL, LEFTN, RIGHTN, TOP;
0     POINTER TO RFIN: L, NEW, P, PARENT, Q;
0
0     P = Q;
0     IF P .EQ. NIL <
1       "PARENT HAS NO DESCENDANTS; FIND THE FIRST REFINEMENT TO THE
1       RIGHT OF PARENT WHICH HAS A DESCENDANT."
1       P = (PARENT~.RLINK);
1       WHILE (P~.FINE) .EQ. NIL .AND. P .NE. LEFTMOST(LEVEL) <
2         P = (P~.RLINK)>
1       IF P .NE. LEFTMOST(LEVEL) <P = (P~.FINE)>
1       ELSE <P = LEFTMOST(LEVEL+1)>>
0     L = (P~.LLINK);
0     NEW(NEW);
0     WITH NEW <
1     IF BASE .EQ. 0 <
2       "BASE NOT SPECIFIED, FIND IT."
2       TOP = (L~.TOP) + 1;
2       ~BASE = TOP + MAX0(0, ((P~.BASE) - TOP - XRATIO*(RIGHTN - LEFTN))
2       /2);>
1     ELSE <
2       ~BASE = BASE>
1
1     IF (P .EQ. LEFTMOST(LEVEL)) LEFTMOST(LEVEL) = NEW;
1     IF ((PARENT~.FINE) .EQ. P .OR. (PARENT~.FINE) .EQ. NIL)
```

```
1       (PARENT^.FINE) = NEW;
1     "INSERT IN HORIZONTAL DOUBLY LINKED LIST."
1     (L^.RLINK) = NEW;
1     (P^.LLINK) = NEW;
1     ^RLINK = P;
1     ^LLINK = L;
1     ^LEFT = LEFTN;
1     ^RIGHT = LEFTN;
1     ^TOP = ^BASE;
1     ^OLDBASE = ^BASE;
1     ^OLDTOP = ^BASE;
1     ^COARSE = PARENT;
1     ^FINE = NIL;
1     IF (DEBUG .GT. 0) OUTPUT NEW, ^BASE, ^OLDTOP; (' CREATE', I4,
1     ' BASE, TOP', 2I5);>
0
0     RETURN;
0   END; "CREATE"
0
0
0   SUBROUTINE DELETE(P, LEVEL);
0     "CHECK TO SEE IF REFINEMENT P HAS ANY CHILDREN.  IF NOT, DELETE IT."
0
0     USE LFTMST; USE DEBUG; DEFINE RFIN;
0     INTEGER LEVEL;
0     POINTER TO RFIN: L, P, PARENT, R;
0     EQUIVALENCE (PARENT, L);
0
0     WITH P <
1     IF ^FINE .EQ. NIL <
2       PARENT = ^COARSE;
2       R = ^RLINK;
2       IF (PARENT^.FINE) .EQ. P <
3         "SEE IF P HAS ANY SIBLINGS, OR IF THERE ARE ANY OTHER
3         REFINEMENTS TO THE RIGHT OF P ON THIS LEVEL."
3         IF (R^.COARSE) .EQ. ^COARSE <
4           (PARENT^.FINE) = R>
3         ELSE <
4           (PARENT^.FINE) = NIL>
3         IF (P .EQ. LEFTMOST(LEVEL)) LEFTMOST(LEVEL) = R>
2
2       "UNCHAIN REFINEMENT P."
2       L = ^LLINK;
2       (L^.RLINK) = R;
2       (R^.LLINK) = L;
2       DISPOSE(P);
2       IF (DEBUG .GT. 0) OUTPUT P; (' DELETE', I4)>>
0
0     RETURN;
0   END; "DELETE"
0
0
0   SUBROUTINE EXTEND.RIGHT(P, NEWEND, L, TLEVEL, LASTPT);
0     "EXTEND THE REFINEMENT POINTED TO BY P TO THE RIGHT.  IF LASTPT IS
0     TRUE, DO NOT FILL IN THE EXTREME RIGHTMOST POINT.  THIS IS
0     IMPORTANT TO AVOID MEMORY REPACKINGS DURING A MERGE."
0
0     USE XRATIO; USE COUNT; DEFINE RFIN;
0     INTEGER EXTENT, L, NEWEND, RIGHT, TLEVEL;
0     LOGICAL LASTPT;
0     POINTER TO RFIN: P, PARENT, Q;
0
0     WITH P <
1     Q = ^RLINK;
1     RIGHT = ^RIGHT;
1     EXTENT = XRATIO*(NEWEND - RIGHT);
1     IF (.NOT. LASTPT) EXTENT = EXTENT - 1;
```

```
1     IF ~TOP + EXTENT .GE. (Q~.BASE) <
2       IF ~BASE .EQ. ~TOP <
3         "A PREVIOUSLY EMPTY REFINEMENT IS A SPECIAL CASE."
3         EXTENT = EXTENT + 1;
3         ~TOP = ~TOP - 1>
2       CALL REALLOC(P, EXTENT);
2       IF (~BASE .GT. ~TOP) ~TOP = ~TOP + 1;
2       NSHL = NSHL + 1>
1     PARENT = ~COARSE;
1     CALL INTERPOLATE(PARENT, ~TOP, RIGHT, NEWEND, L, TLEVEL, LASTPT);
1     ~RIGHT = NEWEND;
1     ~TOP = ~BASE + XRATIO*(~RIGHT - ~LEFT)>
0
0     RETURN;
0   END; "EXTEND.RIGHT"
0
0
0   SUBROUTINE EXTEND.LEFT(P, LEFTN, L, TLEVEL);
0     "EXTEND THE REFINEMENT POINTED TO BY P TO THE LEFT"
0
0     USE XRATIO; USE COUNT; DEFINE RFIN;
0     INTEGER EXTENT, L, LEFT, LEFTN, TLEVEL;
0     POINTER TO RFIN: P, Q;
0
0     Q = (P~.LLINK);
0     LEFT = (P~.LEFT);
0     EXTENT = XRATIO*(LEFT - LEFTN);
0     IF (P~.BASE) - EXTENT .LE. (Q~.TOP) <
1       CALL REALLOC(P, -EXTENT);
1       NSHRT = NSHRT + 1>
0     (P~.BASE) = (P~.BASE) - EXTENT;
0     CALL INTERPOLATE((P~.COARSE), (P~.BASE), LEFTN, LEFT, L, TLEVEL,
0     .TRUE.);
0     (P~.LEFT) = LEFTN;
0
0     RETURN;
0   END; "EXTEND.LEFT"
0
0
0   SUBROUTINE INTERPOLATE(PARENT, BFINE, LEFT, RIGHT, L, TLEVEL, LASTPT);
0     "COPY SOLUTION VALUES IN LOCATIONS LEFT TO RIGHT-1, INCLUSIVE, OF THE
0     REFINEMENT POINTED TO BY PARENT, TO ITS (DESCENDANT) REFINEMENT.
0     IF LASTPT = TRUE, DO THE SAME FOR THE RIGHT POINT.  THEN
0     INTERPOLATE SOLUTION VALUES BETWEEN THE COPIED VALUES IN THE
0     DESCENDANT."
0
0     USE SOLN; USE COM3; USE STEPSZ; USE XRATIO; USE COM12; DEFINE RFIN;
0     REAL FRAC, TEMP(NCOMP);
0     INTEGER BC, BF, BFINE, I, J, L, LEFT, M, RIGHT, RM1, TLEVEL, XRM1;
0     POINTER TO RFIN: PARENT;
0     LOGICAL LASTPT;
0
0     RM1 = RIGHT - 1;
0     XRM1 = XRATIO - 1;
0     BC = (PARENT~.BASE) - XRATIO*(PARENT~.LEFT) + LEFT;
0     BF = BFINE;
0     DO J = LEFT, RM1 <
1       DO I = 1, NCOMP <
2         V(I, BF) = V(I, BC)>
1       BF = BF + 1;
1       DO M = 1, XRM1 <
2         FRAC = FLOAT(M)/XRATIO;
2         IF TLEVEL .EQ. 1 <
3           CALL EXACT.SOLUTION(A + (XRATIO*J + M)*H(L), K(1), TEMP);
3           DO I = 1, NCOMP <
4             V(I, BF) = TEMP(I)>>
2         ELSE IF QUADRAT <
```

```
3          "QUADRATIC INTERPOLATION, USING TWO CLOSEST COARSE MESH
3          POINTS AND ONE TO THE LEFT, EXCEPT AT LEFT BOUNDARY"
3          DO I = 1, NCOMP <
4          IF J .NE. 0 <
5            V(I,BF) = V(I,BC) + FRAC*(V(I,BC) - V(I,BC-1) + 0.5*
5            (FRAC + 1.)*(V(I,BC+1) - 2.*V(I,BC) + V(I,BC-1)))>
4          ELSE <
5            V(I,BF) = V(I,BC) + FRAC*(V(I,BC+1) - V(I,BC) + 0.5*
5            (FRAC - 1.)*(V(I,BC+2) - 2.*V(I,BC+1) + V(I,BC)))>>>
2        ELSE < "LINEAR INTERPOLATION"
3          DO I = 1, NCOMP <
4            V(I,BF) = V(I,BC) + FRAC*(V(I,BC+1) - V(I,BC))>>
2        BF = BF + 1>
1      BC = BC + 1>
0
0    IF LASTPT <
1      DO I = 1, NCOMP <
2        V(I,BF) = V(I,BC)>>
0
0    RETURN;
0  END; "INTERPOLATE"
0
0
0  SUBROUTINE REALLOC(Q, EXTENT);
0      "THE REFINEMENT POINTED TO BY POINTER Q HAS RUN OUT OF ROOM ON
0    ITS RIGHT (IF EXTENT .GT. 0) OR ITS LEFT (IF EXTENT .LT. 0).
0    REALLOCATE MEMORY.  SEE D. E. KNUTH, THE ART OF COMPUTER PROGRAM-
0    MING, VOL. 1, PP. 240-249.  THIS IS A MODIFICATION OF ALGORITHM G,
0    P. 245.  WE AWARD THE AVAILABLE FREE SPACE TO THE REFINEMENTS AS
0    FOLLOWS.  APPROXIMATELY 10 PERCENT OF THE AVAILABLE MEMORY WILL BE
0    SHARED EQUALLY AMONG THE REFINEMENTS.  THE OTHER 90 PERCENT IS
0    AWARDED PROPORTIONATELY TO THE AMOUNT OF MOVEMENT SINCE THE LAST
0    REPACKING.  IF THE RIGHT END OF A REFINEMENT HAS MOVED RIGHT SINCE
0    THEN (COMPARE TOP AND OLDTOP), THE AWARD IS TO THE RIGHT OF THAT
0    REFINEMENT.  IF THE LEFT END HAS MOVED LEFT SINCE THE LAST
0    REPACKING (COMPARE BASE AND OLDBASE), THE AWARD IS TO THE LEFT."
0
0    USE LFTMST; USE COM5; USE DEBUG; DEFINE RFIN;
0    REAL ALPHA, BETA, SIGMA, TAU;
0    INTEGER D(MAXRFINE), EXTENT, FREESP, INC, LEFT, NEWBASE(MAXRFINE),
0    NRFINE, RT;
0    POINTER TO RFIN: LAST, P, Q;
0    EQUIVALENCE (NEWBASE, D);
0
0    WITH P <
1    LAST = LEFTMOST(NLEVEL+1);
1    IF DEBUG .GT. 0 <
2      OUTPUT Q, EXTENT; (' REPACK; REF. NO', I5, ' EXTENT ', I5
2      / ' REF. NO, BASE, TOP ');
2      P = LEFTMOST(1);
2      REPEAT <
3        OUTPUT P, ~BASE, ~TOP; (3I6);
3        P = ~RLINK;>
2      UNTIL P .EQ. LAST;>
1
1    IF EXTENT .GT. 0 <(Q~.TOP) = (Q~.TOP) + EXTENT>
1            ELSE <(Q~.BASE) = (Q~.BASE) + EXTENT>
1    FREESP = MEMAVAIL + 1;
1    INC = 0;
1    NRFINE = 0;
1    P = LAST;
1    D(LAST) = 0;
1    REPEAT <
2      NRFINE = NRFINE + 1;
2      P = ~LLINK;
2      D(P) = 0;
2      FREESP = FREESP - (~TOP - ~BASE + 1);
```

```
2       RT = MAX0(0, ~TOP - ~OLDTOP);
2       LEFT = MAX0(0, ~OLDBASE - ~BASE);
2       INC = INC + MAX0(RT, LEFT);
2       IF RT .GT. LEFT <D(~RLINK) = D(~RLINK) + RT>
2               ELSE <D(P) = LEFT>>
1     UNTIL P .EQ. LEFTMOST(1);
1
1     IF FREESP .LT. 0 <
2       OUTPUT; (' *** MEMORY OVERFLOW.  PROGRAM ENDED. ');
2       STOP>
1     IF INC .GT. 0 <
2       ALPHA = (0.1*FREESP)/NRFINE;
2       BETA = (0.9*FREESP)/INC>
1     ELSE <
2       ALPHA = FLOAT(FREESP)/NRFINE;
2       BETA = 0.>
1     P = LEFTMOST(1);
1     NEWBASE(P) = ~BASE;
1     SIGMA = 0.;
1     UNTIL ~RLINK .EQ. LAST <
2       TAU = SIGMA + ALPHA + D(~RLINK)*BETA;
2       NEWBASE(~RLINK) = NEWBASE(P) + ~TOP - ~BASE + 1 + INT(TAU)
2       - INT(SIGMA);
2       SIGMA = TAU;
2       P = ~RLINK;>
1
1     IF EXTENT .GT. 0 <(Q~.TOP) = (Q~.TOP) - EXTENT>
1             ELSE <(Q~.BASE) = (Q~.BASE) - EXTENT;
2                 NEWBASE(Q) = NEWBASE(Q) - EXTENT>
1     CALL REPACK(NEWBASE);
1     P = LEFTMOST(1);
1     REPEAT <
2       ~OLDBASE = ~BASE;
2       ~OLDTOP = ~TOP;
2       P = ~RLINK;>
1     UNTIL P .EQ. LAST;
1
1     IF EXTENT .GT. 0 <(Q~.OLDTOP) = (Q~.OLDTOP) + EXTENT>
1             ELSE <(Q~.OLDBASE) = (Q~.OLDBASE) + EXTENT>
1     IF DEBUG .GT. 0 <
2       OUTPUT; (' REF. NO, BASE, TOP =');
2       P = LEFTMOST(1);
2       REPEAT <
3         OUTPUT P, ~BASE, ~TOP; (3I6);
3         P = ~RLINK;>
2       UNTIL P .EQ. LAST;>
1     >
0
0     RETURN;
0   END; "REALLOC"
0
0
0   SUBROUTINE REPACK(NEWBASE);
0     "RELOCATE SEQUENTIAL TABLES.  THIS IS ALGORITHM R OF KNUTH, VOL.
0     1, P. 246.  THE ONLY CHANGE IS BECAUSE OUR ARRAY V STARTS FROM 0
0     INSTEAD OF 1."
0
0     USE SOLN; USE LFTMST; USE COM5; DEFINE RFIN;
0     INTEGER BASE, DELTA, I, J, NEWBASE(MAXRFINE), TOP;
0     POINTER TO RFIN: LAST, P, SECOND;
0
0     WITH P <
1     SECOND = LEFTMOST(2);
1     LAST = LEFTMOST(NLEVEL+1);
1     P = SECOND;
1     UNTIL P .EQ. LAST <
2       IF NEWBASE(P) .LT. ~BASE <
```

```
3         "SHIFT DOWN."
3         BASE = ^BASE;
3         TOP = ^TOP;
3         DELTA = BASE - NEWBASE(P);
3         DO J = BASE, TOP <
4           DO I = 1, NCOMP <
5             V(I, J-DELTA) = V(I, J)>>
3         ^BASE = NEWBASE(P);
3         ^TOP = ^TOP - DELTA>
2       P = ^RLINK;>
1
1     "FIND START OF SHIFT."
1     P = LAST;
1     UNTIL P .EQ. SECOND <
2       P = ^LLINK;
2       IF NEWBASE(P) .GT. ^BASE <
3         "SHIFT UP."
3         DELTA = NEWBASE(P) - ^BASE;
3         FOR J = ^TOP BY -1 TO ^BASE <
4           DO I = 1, NCOMP <
5             V(I, J+DELTA) = V(I, J)>>
3         ^BASE = NEWBASE(P);
3         ^TOP = ^TOP + DELTA>>>
0
0     RETURN;
0   END; "REPACK"
0
0
  $ 'USE COM16;' = 'COMMON /COM16/ L2NORM, MAX, MIN;
0             REAL L2NORM(NCOMP,2), MAX(NCOMP,2), MIN(NCOMP);'
0
0   SUBROUTINE NORM(T);
0
0     "COMPUTE MAXIMUM AND MEAN SQUARE ERROR AT ONE
0     TIME LEVEL.  COMPUTE L2 NORM OF SOLUTION"
0
0     USE SOLN; USE LFTMST; USE COM3; USE STEPSZ; USE XRATIO; USE COM16;
0     USE ERROR; DEFINE RFIN;
0     REAL DIFF(NCOMP, 1), EXACT(NCOMP), MAXREAL, T, TEMP(NCOMP,2),
0     XLEFT;
0     INTEGER BASE, I, J, L, LEFTR, M, NPTS, NPTSP2, OFFSET, RIGHTR;
0     POINTER TO RFIN: P, UP;
0     EQUIVALENCE (ESTERROR, DIFF);
0
0     MAXREAL = 1.E30;
0     DO I = 1, NCOMP <
1       DO M = 1, 2 <
2         L2NORM(I, M) = 0.;
2         MAX(I, M) = - MAXREAL>
1       MIN(I) = MAXREAL>
0     L = 1;
0     P = LEFTMOST(1);
0     REPEAT <
1       REPEAT <
2         BASE = (P^.BASE);
2         NPTS = (P^.TOP) - BASE;
2         NPTSP2 = NPTS + 2;
2         OFFSET = XRATIO*(P^.LEFT);
2         XLEFT = A + H(L-1)*(P^.LEFT);
2         UP = (P^.FINE);
2         IF UP .NE. NIL <
3           LEFTR = (UP^.LEFT) - OFFSET;
3           RIGHTR = (UP^.RIGHT) - OFFSET>
2         ELSE <
3           LEFTR = NPTSP2>
2         J = 0;
2         GO TO :L3:;
```

```
2         REPEAT <
3           CALL EXACT.SOLUTION(XLEFT + J*H(L), T, EXACT);
3           DO I = 1, NCOMP <
4             TEMP(I,1) = V(I, BASE+J);
4             EXACT(I) = EXACT(I) - TEMP(I,1);
4             IF L .EQ. 1 <
5               DIFF(I,J) = EXACT(I)>
4             TEMP(I,2) = ABS(EXACT(I));
4             MIN(I) = AMIN1(MIN(I), TEMP(I,1));
4             DO M = 1, 2 <
5               L2NORM(I,M) = L2NORM(I,M) + H(L)*TEMP(I,M)**2;
5               MAX(I,M) = AMAX1(MAX(I,M), TEMP(I,M))>>
3           J = J + 1;
3      :L3:  IF J .EQ. LEFTR <
4             J = RIGHTR + 1;
4             UP = (UP~.RLINK);
4             IF (UP~.COARSE) .EQ. P <
5               LEFTR = (UP~.LEFT) - OFFSET;
5               RIGHTR = (UP~.RIGHT) - OFFSET>>>
2         WHILE J .LE. NPTS;
2         P = (P~.RLINK)>
1       UNTIL P .EQ. LEFTMOST(L+1);
1       L = L + 1>
0     UNTIL (P~.COARSE) .EQ. NIL;
0
0     P = LEFTMOST(2);
0     WHILE (P~.COARSE) .EQ. LEFTMOST(1) <
1       FOR J = (P~.LEFT) TO (P~.RIGHT) <
2         CALL EXACT.SOLUTION(A + J*H(1), T, EXACT);
2         DO I = 1, NCOMP <
3           DIFF(I,J) = EXACT(I) - V(I,J)>>
1       P = (P~.RLINK)>
0     DO I = 1, NCOMP <
1       DO M = 1, 2 <
2         L2NORM(I,M) = SQRT(L2NORM(I,M))>>
0
0     RETURN;
0   END; "NORM"
0
  $'USE COM17;' =
0     'COMMON /COM17/ ZZPAGE, ZPAGE, PAGE, BLANK;
0     INTEGER ZZPAGE, ZPAGE(HEIGHTP1), PAGE(HEIGHTP1, PAGEWIDTH),
0     BLANK;'
0
0   SUBROUTINE CLEAR;
0     USE COM3; USE ZERO; USE COM17;
0     DO I = ZERO, HEIGHT <
1       DO J = ZERO, N <
2         PAGE(I, J) = BLANK>>
0     RETURN;  END;
0
0   INTEGER FUNCTION ROUND(X);
0     REAL X;
0     IF X .GE. 0. <ROUND = X + 0.5> ELSE <ROUND = X - 0.5>
0     RETURN;   END;
0
0
0   SUBROUTINE PLOT (TLEVEL, T);
0     "PRINT AND PLOT SOLUTION AND ERROR MEASURES."
0
0     USE SOLN; USE LFTMST; USE COM3; USE STEPSZ; USE ZERO; USE XRATIO;
0     USE DEBUG; USE COM16; USE COM17; USE ERROR; DEFINE RFIN;
0     REAL DIFF(NCOMP, 1), RANGE, T, XJ, XLEFT, XRIGHT;
0     INTEGER BASE, I, J, L, LEFTR, M, NCUT, NPTS, NPTSP2, NRF, OFFSET,
0     RIGHTR, ROUND, TLEVEL, TOP, TOTPTS;
0     INTEGER PERIOD, ZEROCH, LABEL(NCOMP);
0     POINTER TO RFIN: P, UP;
```

```
0     EQUIVALENCE (ESTERROR, DIFF);
0     DATA BLANK /1H /, PERIOD /1H./, ZEROCH /1H0/, LABEL /1HV, 1HW/;
0
0     CALL NORM(T);
0     OUTPUT T, TLEVEL; (/' T =', 1PE15.7, ' =', I5, ' DELTA T');
0     DO I = 1, NCOMP <
1       OUTPUT I, L2NORM(I,1), MAX(I,1), MIN(I); (' NORM OF V(', I1,
1       ') =', 1PE15.7, ' MAX =', 1PE15.7, ' MIN =', 1PE15.7)>
0     DO I = 1, NCOMP <
1       OUTPUT I, L2NORM(I,2), MAX(I,2); (' V(', I1, ') MEAN SQUARE'
1       ' ERROR', 1PE15.7, ' MAXIMUM ERROR', 1PE15.7)>
0     IF TLEVEL .NE. 1 .AND. MOD(DEBUG, 2) .EQ. 1 <
1       DO I = 1, NCOMP <
2         OUTPUT I, T; (/' V(', I1, ') ERRORS AT T =', 1PE15.7);
2         OUTPUT (DIFF(I,J), J = ZERO, N); (1X, 1P10E12.4)>>
0
0     "PRINT SOLUTION AT NEW T-LEVEL"
0     IF MOD(DEBUG, 8)/4 .EQ. 1 <
1       L = 1;
1       P = LEFTMOST(L);
1       REPEAT <
2         REPEAT <
3           BASE = (P~.BASE);
3           TOP =  (P~.TOP);
3           DO I = 1, NCOMP <
4             OUTPUT I, P, L; (/' V(', I1, ') REFINEMENT ', I4,
4             ' LEVEL', I4);
4             OUTPUT (V(I,J), J = BASE, TOP); (1X, 1P8E15.7)>
3           P = (P~.RLINK)>
2         UNTIL P .EQ. LEFTMOST(L+1);
2         L = L + 1>
1       UNTIL (P~.COARSE) .EQ. NIL;
1       >
0
0     IF DEBUG .GT. 0 <
1     "COUNT NUMBER OF DISTINCT MESH POINTS.  ALSO COUNT NUMBER OF
1     (NONEMPTY) REFINEMENTS, EXCLUDING THE COARSEST MESH."
1     P = LEFTMOST(2);
1     TOTPTS = 0;
1     NRF = 0;
1     WHILE (P~.COARSE) .NE. NIL <
2       TOTPTS = TOTPTS + (P~.RIGHT) - (P~.LEFT);
2       NRF = NRF + 1;
2       P = (P~.RLINK)>
1     TOTPTS = N + 1 + TOTPTS*(XRATIO - 1);
1
1     "PLOT SOLUTION ON PRINTER AND GRAFPAC"
1     NCUT = MIN0(N, PAGEWIDTH);
1     DO I = 1, NCOMP <
2       CALL CLEAR;
2       RANGE = MAX(I,1) - MIN(I);
2       IF RANGE .EQ. 0. < RANGE = 1.>
2
2       "INSERT DOTS ON PRINT PLOT TO DENOTE REFINED REGION"
2       P = LEFTMOST(2);
2       WHILE (P~.COARSE) .EQ. LEFTMOST(1) <
3         DO M = ZERO, HEIGHT <
4           PAGE(M, (P~.LEFT)) = PERIOD;
4           PAGE(M, (P~.RIGHT)) = PERIOD>
3         P = (P~.RLINK)>
2
2       IF ROUND(MIN(I)) .LE. 0 .AND. 0. .LE. MAX(1,I) <
3         "INSERT A LINE OF ZEROES INTO THE V PLOT"
3         M = ROUND(-MIN(I) * HEIGHT/ RANGE);
3         DO J = ZERO, NCUT <
4           PAGE(M, J) = ZEROCH>>
2       DO J = ZERO, NCUT <
```

```
3         M = ROUND((V(I,J) - MIN(I)) * HEIGHT / RANGE);
3         PAGE(M, J) = LABEL(I)>
2       OUTPUT; (1H );
2       FOR M = HEIGHT BY -1 TO 0 <
3         OUTPUT (PAGE(M, J), J = ZERO, NCUT); (1X,
3         PAGEWIDTHA1)>
2
2       "PUT OUT NUMBERS FOR GRAFPAC"
2       WRITE (7, :FORM1:) TOTPTS, I;
2       WRITE (7, :FORM2:) T;
2       WRITE (7, :FORM2:) A, B, -1.1, 1.1;
2       WRITE (7, :FORM1:) NRF;
2       "PLOT REFINEMENT BOUNDARIES"
2       P = LEFTMOST(2);
2       L = 2;
2       WHILE (P~.COARSE) .NE. NIL <
3         REPEAT <
4           XLEFT = A + H(L-1)*(P~.LEFT);
4           XRIGHT = A + H(L-1)*(P~.RIGHT);
4           WRITE (7, :FORM3:) L, XLEFT, XRIGHT;
4           P = (P~.RLINK)>
3         UNTIL P .EQ. LEFTMOST(L+1);
3         L = L + 1>
2
2       "PLOT ALL POINTS ON ALL LEVELS IN STRICT LEFT-TO-RIGHT ORDER
2       BY USING A DEPTH-FIRST SEARCH OF THE TREE."
2       P = LEFTMOST(1);
2       L = 1;
2  :RECURSE:
2       J = 0;
2       UP = (P~.FINE);
2       REPEAT <
3         OFFSET = XRATIO*(P~.LEFT);
3         XLEFT = A + H(L-1)*(P~.LEFT);
3         BASE = (P~.BASE);
3         NPTS = (P~.TOP) - BASE;
3         NPTSP2 = NPTS + 2;
3         IF (UP~.COARSE) .NE. P <
4           LEFTR = NPTSP2>
3         ELSE <
4           LEFTR = (UP~.LEFT) - OFFSET;
4           RIGHTR = (UP~.RIGHT) - OFFSET>
3         GO TO :L4:;
3         REPEAT <
4           XJ = XLEFT + J*H(L);
4           WRITE (7, :FORM2:) XJ, V(I,BASE+J);
4           J = J + 1;
4  :L4:       IF J .EQ. LEFTR <
5             P = UP;
5             L = L + 1;
5             GO TO :RECURSE:>>
3         WHILE J .LE. NPTS;
3         UP = P;
3         P = (P~.COARSE);
3         J = (UP~.RIGHT) + 1 - XRATIO*(P~.LEFT);
3         UP = (UP~.RLINK);
3         L = L - 1>
2       UNTIL L .LT. 1;>>
0
0  :FORM1: FORMAT (I5, I2);
0  :FORM2: FORMAT (4E15.7);
0  :FORM3: FORMAT (I1, 2E15.7);
0    RETURN;
0  END; "PLOT"
   $$
   $$
 0 MORTRAN ERRORS ENCOUNTERED
```