Finite-Difference Methods for the Diffusion Equation

Kenneth John Hayman B.Sc.(Maths.Sc.) (Hons.)

Thesis submitted for the degree of

Doctor of Philosophy

Department of Applied Mathematics

University of Adelaide

March 1988

Contents

Summary

The development of accurate finite-difference methods for solving the linear diffusion equation with constant coefficients, in either one or two spatial dimensions, is useful in the study of several physical phenomena, such as underground water flow and diffusion of heat through a solid body. As well as accuracy, however, the amount of computer time taken to generate a solution must be taken into account, since this may be an important practical constraint.

The approach taken has been to thoroughly examine the one-dimensional case and then, having found some good methods to solve this problem, use the knowledge gained to develop methods for solving the more complicated two-dimensional problem. This work can then be extended to solve the variable coefficient diffusion equation, or even the non-linear equation, by considering that over the size of the computational stencil used, the linearised constant coefficient equation is a good approximation to the equation being solved.

In order to determine the accuracy of a given finite-difference equation, the *modified equivalent equation,* developed by Warming and Hyett (1974) for their heuristic stability analysis, has been adapted and used. This approach allows the simple determination of the theoretical order of accuracy of any finite-difference equation, thus allowing methods to be compared with one another. Also, from the truncation error of the modified equivalent equation, it is possible to eliminate the dominant error terms associated with finite-difference equations that contain free parameters (weights), thus leading to more accurate methods.

Several different finite-difference methods for the one-dimensional diffusion equation

are developed, and their theoretical and actual truncation errors, as well as the CPU time required for solution, are compared to determine the most practical methods. To determine the actual order of accuracy of the method, graphs of $\log\{(Gridspacing)\}$ against $\log\{|(Error)|\}$ are plotted for decreasing values of the grid spacing and the results are examined. These graphs should be straight lines, and the slopes of the lines give the actual order of accuracy of the method. In most cases, this order matches the theoretical prediction, and in those cases where this is not so, the reasons for the difference are investigated.

Where the normal derivative at one boundary (or even both boundaries) is specified rather than the boundary value, the approximations at grid points on the boundary must be calculated. It is shown that it is still possible to produce relatively accurate solutions although the results are not as accurate as when the boundary value is known. Also in this case the techniques for handling the problems that arise near the boundary from some of the finite-difference equations having spatially wide computational stencils must be revised.

The same techniques as were used for the one-dimensional case are then applied to developing accurate finite-difference equations for the two-dimensional diffusion equation. In this case the computational stencils contain more grid points, and therefore allow more weights to be included. However, the larger number of low-order error terms in the modified equivalent equation, arising from the added cross-derivative error terms, means that some of the extra weights must be used to maintain the same accuracy as was achieved for the one-dimensional problem. Again, many different stencils and their corresponding finite-difference equations are examined, in order to find the best methods which can be practically applied. The method for determining the actual order of accuracy of the method is the same as that used for the one-dimensional case.

The so-called "locally one-dimensional" methods are examined, where the very accurate methods developed for the one-dimensional case can be applied directly to the two-dimensional problem. The best of the one-dimensional methods used in this manner are then compared with the best of the fully two-dimensional methods, to deter-

mine the preferred solution method. Note that to implement these methods correctly it is necessary to split the two-dimensional diffusion equation into two one-dimensional equations, each of which is solved alternately. Doing this requires special consideration of values on the boundary in the cases where only diffusion in one direction has been modelled. If this is not done then the results are downgraded to second-order accurate, regardless of the order of the finite-difference equation used.

The other class of techniques in common use for solving the two-dimensional diffusion equation is the alternating direction implicit methods, which combine the advantages of implicit methods, particularly large stability ranges, with fast execution speed on a computer, which is the major problem with fully implicit equations for the two-dimensional problem. Two different kinds of equations are considered, those based on the "classical" ADI methods, and those based on a "marching" equation, which must be applied "left-to-right" and then "right-to-left" in each spatial direction, as well as alternating the spatial direction. The potential for generating accurate solutions is examined for each type of equation, since these methods prove to be the only way of obtained generally fourth-order accurate results without using spatially wide computational stencils.

Another important practical problem that arises when solving the two-dimensional problem is an irregular boundary, which results in the specified boundary values not coinciding with the grid points of a uniform grid. This problem can be overcome by developing special finite-difference equations which allow for a non-uniform grid spacing at such a boundary. The effect of using these equations, which have a theoretical accuracy one order lower than their uniform grid analogues, is examined.

In order to make this work feasible, computer programs were developed to perform the time consuming and mechanical tasks involved with developing the finite-difference equations by hand. Using these programs it is possible to start with a desired method of differencing the diffusion equation, and have the computer determine the finite-difference equation corresponding to that differencing, as well as its modified equivalent equation. Weights specified in the original differencing can then be used to eliminate the dominant error terms, which then leads to the optimal form of the finite-difference

equation. This optimal equation can then be checked for such things as time-stepping (von Neumann) stability, solvability and/or marching stability, as appropriate. In some cases it is possible to use some of the weights to enhance the stability region of the equation rather than to increase the accuracy.

Statement of Originality

I certify that this thesis contains no material which has been accepted for the award of any other degree or diploma in any University and, to the best of my knowledge and belief, contains no material that has been previously published or written by any other person, except where due reference has been made in the text.

If this thesis is accepted, I give consent for it to be made available for photocopying and loan as applicable.

Kenneth John HAYMAN

Acknowledgements

I wish to express my sincere gratitude to my supervisor, Dr. B.J. Noye for his guidance, encouragement and support during the period this work was carried out. Thanks also go to David Beard for his invaluable assistance with computing problems as they arose, as well as to other members of the University of Adelaide Mathematics Departments for their help and advice.

The financial support of a Commonwealth Postgraduate Research Award during the period this work was done is gratefully acknowledged.

The use of the symbolic manipulation package Macsyma, by Symbolics Inc., and the numerical IMSL subroutine libraries is also acknowledged.

Chapter 1

Introduction

This work is aimed at producing highly accurate finite-difference methods for solving the one and two-dimensional linear diffusion equations with constant coefficients. The two-dimensional equation can be written in the form

$$\frac{\partial \hat{\tau}}{\partial t} - \alpha_x \frac{\partial^2 \hat{\tau}}{\partial x^2} - \alpha_y \frac{\partial^2 \hat{\tau}}{\partial y^2} = 0, \tag{1.1.1}$$

where α_x and α_y are the constant coefficients of diffusion in the x and y directions respectively. The one-dimensional equation is the special case of this where $\alpha_y = 0$. Since the aim of this work is to produce methods which can be applied to practical problems, it is necessary to produce methods which not only have high theoretical accuracy, but are also practical to implement on a computer without requiring excessive amounts of CPU time.

The ultimate goal is to produce good methods for the two-dimensional case, since this equation has practical uses in such areas as modelling the flow of underground water, the flow of oil in underground reservoirs (Bear, 1972), and the diffusion of heat through solid bodies. To approach this goal, a thorough understanding of the one-dimensional problem is required. Once this simpler problem has been examined in detail and highly accurate and practical solution techniques have been found, the experience gained can be applied to the much more complicated but more practically useful two-dimensional problem.

Extensions of this work into the cases with variable coefficients are also possible. In this case, the fully general form of the diffusion equation, namely

$$\frac{\partial \hat{\tau}}{\partial t} - \frac{\partial}{\partial x}\left(\alpha_x(x,y,t,\hat{\tau})\frac{\partial \hat{\tau}}{\partial x}\right) - \frac{\partial}{\partial y}\left(\alpha_y(x,y,t,\hat{\tau})\frac{\partial \hat{\tau}}{\partial y}\right) = 0 \qquad (1.1.2)$$

in two-dimensions, must be considered. This equation is used in much the same areas as the constant-coefficient equation, but is more applicable in cases where there is a steep "front" involved in the diffusion (Richtmyer and Morton, 1967). To adequately handle problems such as this one, a good understanding of the constant coefficient case, as presented here, is a necessary prerequisite. Constraints on available time, however, prevented this extension from being included in this work.

As already mentioned, the approach taken towards the goal of accurate methods for the two-dimensional problem is to start with the one-dimensional case and find the best methods for dealing with that case, with the various techniques being judged on both the accuracy of the solution generated and the amount of CPU time required to find the solution. From this work, insight is gained into the most advantageous methods for developing solution techniques in the one-dimensional case, so that when attempts are made to solve the more difficult two-dimensional problem, our efforts can be directed in ways most likely to give profitable results.

In all cases, the chosen solution method is via finite-difference techniques, since they offer enough flexibility to generate accurate solutions, and furthermore the numerical errors can be predicted theoretically. Such predictions can then be checked against results generated by test runs of problems with known solutions. In all cases where computer time usage is referred to in this work, the amount of CPU time is meant rather than the elapsed real time, since this latter may vary enormously depending on the system load.

The one-dimensional problem is first approached using *explicit* finite-difference methods, since these are both straight-forward to implement and very fast to run on a computer. Unfortunately, most methods of this type suffer from numerical instabilities, which in some cases place such severe restrictions on the methods that they are of no practical use. This is caused by the stability condition dictating that, for a

given spatial resolution of the the solution domain, the time step must be extremely small, which means that many more time steps are required to find the solution, which in turn increases the CPU time usage. Such problems frequently arise for methods which are extremely accurate, which means that a balance must be struck between high accuracy and keeping the computer time required to generate a solution within reasonable bounds. Balanced with this, however, is the fact that such highly accurate methods can produce very good results using a relatively coarse grid, and this can be used to constrain the CPU time requirements of the solution.

Implicit methods, which in many cases have no limits on their stability, are then investigated. These methods, however, require the solution of a set of linear algebraic equations for each time step, which adds significantly to the amount of computer time used to generate a solution. Also, the solution of this set of algebraic equations may impose limitations on the use of the method, since the coefficient matrix for the set of equations is required to be diagonally dominant to ensure that the solution process is stable. Added to this, it is found that even in cases where neither the stability of the equation itself nor the solution of the equations imposes restrictions on the size of the allowable time step, the error involved in the finite-difference equation increases enormously with an increase in the time step used. This makes it impractical to use large time steps to take advantage of the stability of these equations.

All the above work is done on the assumption that the boundary conditions are specified values on each boundary, which is usually referred to as a *Dirichlet* boundary condition (Duff and Naylor, 1966). In the case where the normal derivative is known instead, which is referred to as a *Neumann* boundary condition, the same solution methods can be employed, but there is the additional problem arising from the requirement to find the actual value at the boundary at each time level. A Neumann boundary condition most frequently arises in practice where one boundary of the solution domain represents an impermeable layer or barrier, which is represented mathematically by saying that the flow velocity across the boundary is zero (Leonard, 1983). Other cases, where there is a known, but non-zero, flow across the boundary also arise in some practical situations.

An added complication in using methods which work well for the Dirichlet case to solve the problem with a Neumann boundary condition is that the boundary value is required by some of the methods used to solve the Dirichlet case, such as happens for implicit equations, so these methods must be re-evaluated to determine whether they are still practical. It is shown that most of the best methods for solving the Dirichlet case still produce accurate results in this case, although the absolute size of the errors has increased by several orders of magnitude, due to the added complication of the problem and the extra approximations required.

From this base, we attempt to find accurate solution techniques for the two-dimensional case. Applying the same methods successfully used in the one-dimensional case, we form generalised equations involving several free weights (parameters), which can be chosen as desired to increase accuracy, numerical stability or both. Several classes of methods are examined, again broadly classed as explicit or implicit. The solution of the set of equations for the implicit methods however has increased in complexity quite dramatically, since the coefficient matrix has now lost its tri-diagonal banded structure which allowed for relatively quick and efficient solution in the one-dimensional problem. This being the case, most of the work on the two-dimensional problem is concentrated on explicit equations.

Another approach to the two-dimensional problem which works quite successfully is to use what are referred to as "locally one-dimensional methods". This involves considering the total two-dimensional problem as a series of one-dimensional problems, firstly by considering a series of constant y values for half a time step, then a series of constant x values for the remaining half time step. In this way, the methods developed for the one-dimensional problem can be directly applied to give a solution to the two-dimensional problem. Note that there is a problem in the implementation of such schemes, since the boundary conditions specified involve diffusion in both spatial directions, but the approximations after the first half time step include diffusion in only one direction. If the boundary conditions are used as given, the order of the solution becomes second-order, regardless of the order of the difference equation being used. This problem can be overcome by noting that some boundary values at

the intermediate time level are not required for a rectangular solution domain, since they are not used in the computations for the next half time step, and the remaining values can be calculated from the previous boundary values using the finite-difference scheme itself, so that the diffusion is the same at the boundaries as in the interior of the region. The case of non-rectangular solution domains is also considered.

One other way of overcoming the problems of fully-implicit equations for the two-dimensional problem is to use an "alternating direction implicit" (ADI) method. These equations are structured so that the set of equations that has to be solved at each time level involves only one spatial dimension. This means that the bandwidth of the coefficient matrix is fixed at three, rather than increasing with the number of grid points used. In this way it is possible to develop more accurate and stable methods than is possible using explicit equations, without the enormous CPU time overhead of the fully implicit equations.

A major part of the work of developing the finite-difference equations was the development of a set of computer programs to speed up the process of finding the finite-difference equation corresponding to a given differencing of either the full advection-diffusion equation (sometimes called the transport equation), namely

$$\frac{\partial \hat{\tau}}{\partial t} + u\frac{\partial \hat{\tau}}{\partial x} + v\frac{\partial \hat{\tau}}{\partial y} - \alpha_x \frac{\partial^2 \hat{\tau}}{\partial x^2} - \alpha_y \frac{\partial^2 \hat{\tau}}{\partial y^2} = 0 \qquad (1.1.3)$$

or one of its special cases, such as the diffusion equation.

Once a particular differencing of the original equation to be solved has been determined, these programs are used to generate the corresponding finite-difference equation, its modified equivalent equation, and hence its truncation error, in a very short period of time. This is a vast improvement over the several days of work often required to determine this information by hand.

Due to the importance of these programs, the documentation describing their use on the VAX is given in Appendix A, as well as a listing of the major program in Appendix C. These programs can also be readily modified to cope with equations other than the advection-diffusion equation, such as the first and second-order wave equations (Noye and Rankovic, 1986).

Once the modified equivalent equation has been determined, a set of equations, usually non-linear and sometimes extremely complicated, can be derived which must then be solved, either by hand or by using a symbolic algebraic manipulation package such as MACSYMA, to determine the values for the free weights that will give the most accurate method possible from the initial differencing. Given these values for the weights, another of the programs substitutes these values into the finite-difference equation to give the optimal form of the equation. The von Neumann stability, and if applicable the solvability of the equation can then be determined, and the equation can then be tried in practice over its usable range, if this range is sufficient to be of practical interest. If the usable range (ie. the range over which the finite-difference equation is consistent, von Neumann stable and solvable) is too severely limited, then this is usually apparent within about half an hour of starting with the initial differencing, which is again much better than the day or two required before these programs were implemented.

Overall, this work provides several practically useful results. Some highly accurate and practically usable finite-difference methods for both the one and two-dimensional diffusion equations with constant coefficients have been derived and analysed, both for the case of known boundary values and also the case where only the derivative is known on one or more of the boundaries. The computer programs developed for this work are also useful for very quickly determining the optimal finite-difference equation for a given differencing, or else for analysing an existing equation to determine its theoretical accuracy.

Chapter 2

The 1-D Diffusion Equation with Dirichlet Boundary Conditions

2.1 Introduction

The one-dimensional linear diffusion equation with constant coefficients can be written as

$$\frac{\partial \hat{\tau}}{\partial t} - \alpha \frac{\partial^2 \hat{\tau}}{\partial x^2} = 0, \tag{2.1.1}$$

where α is the constant diffusion coefficient and $\hat{\tau} = \hat{\tau}(x,t)$. This equation is of interest since it has practical applications in such processes as underground fluid flow, which can be regarded as having no advection component, and in areas such as heat conduction along a thin insulated rod, where the diffusion may be regarded as being one-dimensional. Most practical applications, however, can only be accurately dealt with using the two-dimensional diffusion equation, which is considered later in this work (Chapter 4).

Equation (2.1.1) can be solved numerically by finite-difference techniques. Without loss of generality, we may assume that equation (2.1.1) has been non-dimensionalised, such that the spatial domain is $[0,1]$, and α is a non-dimensional diffusion coefficient. The space domain is then divided into J equal grid spacings of length Δx. The

equation can then be solved on this grid by starting from some known initial state (ie. time $t = 0$) and set of boundary conditions, using these to compute an approximation to the state at a small time $t = \Delta t$ later, and repeating this process until the desired time $t = T$ is reached. It should be noted however that some solution techniques, which are described later, use information from more than one time level to obtain values at the new time level. In such cases some special starting procedure must be used until solutions at enough time levels are available for the desired equation to be employed.

In order to carry out this time stepping process, it is necessary to have both a specified initial condition of the form

$$\hat{\tau}(x, 0) = f(x), \quad \text{for } 0 \leq x \leq 1, \tag{2.1.2}$$

and a pair of boundary conditions at $x = 0$ and $x = 1$. In the first instance, it will be assumed that there is a Dirichlet boundary condition (ie. the boundary *values* are specified), in the form

$$\begin{aligned} \hat{\tau}(0, t) &= g_1(t), \\ \hat{\tau}(1, t) &= g_2(t), \end{aligned} \tag{2.1.3}$$

for all $t > 0$. Later, the case of a Neumann boundary condition (ie where the *normal derivative* is specified) at one boundary, in the form

$$\left.\frac{\partial \hat{\tau}}{\partial x}\right|_{(0,t)} = c_1(t), \quad t > 0 \tag{2.1.4}$$

will also be considered.

As well as the initial and boundary conditions, we need a finite-difference equation that relates the values at both the current and next time levels. Such an equation is derived by approximating all the derivative terms in (2.1.1), applied at the point $(j\Delta x, n\Delta t)$, by combinations of approximate values of $\hat{\tau}$ at the grid points surrounding it. For convenience, the grid point $j\Delta x$ at time $n\Delta t$ will be referred to as the (j, n) grid point, the exact value of $\hat{\tau}$ at this point will be denoted by $\hat{\tau}_j^n$ and the approximate value by τ_j^n. The substitution

$$s = \frac{\alpha \Delta t}{(\Delta x)^2} \tag{2.1.5}$$

is used when writing finite-difference equations. This dimensionless quantity always occurs in finite-difference approximations to equation (2.1.1), due to the form of this equation and the derivatives involved in it.

As an example of a differencing, consider the simplest possible approximations to the derivative terms in equation (2.1.1), namely

$$\left.\frac{\partial \hat{\tau}}{\partial t}\right|_j^n = \frac{\hat{\tau}_j^{n+1} - \hat{\tau}_j^n}{\Delta t} + O\{\Delta t\}, \tag{2.1.6}$$

$$\left.\frac{\partial^2 \hat{\tau}}{\partial x^2}\right|_j^n = \frac{\hat{\tau}_{j-1}^n - 2\hat{\tau}_j^n + \hat{\tau}_{j+1}^n}{(\Delta x)^2} + O\{(\Delta x)^2\}. \tag{2.1.7}$$

When these approximations are substituted into equation (2.1.1), and the terms of $O\{(\Delta x)^2, \Delta t\}$ are dropped to give an equation which is useful in practice, the finite-difference approximation to (2.1.1) is

$$\left(\frac{\tau_j^{n+1} - \tau_j^n}{\Delta t}\right) - \alpha\left(\frac{\tau_{j-1}^n - 2\tau_j^n + \tau_{j+1}^n}{(\Delta x)^2}\right) = 0. \tag{2.1.8}$$

This can be rearranged to give the finite-difference equation

$$\tau_j^{n+1} = s\tau_{j-1}^n + (1-2s)\tau_j^n + s\tau_{j+1}^n, \tag{2.1.9}$$

which is called the FTCS equation, since it is derived from the forward-time (FT) and centred-space (CS) approximations to the derivative terms in equation (2.1.1). This finite-difference equation uses the computational stencil shown in Figure 2.1.

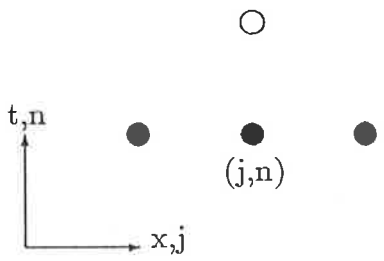


Figure 2.1: *The (1,3) computational stencil of the FTCS method*

Equation (2.1.9) is termed *explicit*, since it can be used directly to compute the value at a grid point at the new time level from values at several grid points at the current and previous time levels. Other equations, which will be examined in detail later, are

termed *implicit*, since they relate the values at several grid points at the new time level to values at the current and previous time levels. This leads to a set of linear algebraic equations that must be solved at each time level.

To simplify the classification of finite-difference equations, methods will be referred to as "a (p, q, r) method" when the computational stencil of the equation involves p grid points at time level $(n+1)$, q points at time level n and r points at time level $(n-1)$. If the method does not involve any points from time level $(n-1)$ then it is simply referred to as an (p, q) method. Thus in this notation, the FTCS equation (2.1.9) is a (1,3) method, since it uses only one grid point at time level $(n+1)$, three points at time level n, and none at all at time level $(n-1)$. The extension of this notation to equations which use more than three time levels is trivial, but such equations are not considered in this work.

2.2 The Modified Equivalent Equation Approach

In order to compare the accuracy of various methods, some measure of the error involved in using a method must be found. The way that has been chosen to do this is a modified equivalent equation approach similar to that used by Warming and Hyett (1974). This approach is chosen because it provides much valuable information about the finite-difference equation, such as the form of the truncation error, how much the solution is shifted spatially by the numerical solution and how much extra diffusion is present, as well as providing for the easy comparison of the errors associated with different finite-difference equations.

Finding the modified equivalent equation is a two-step process. The first step is to expand each term of the finite-difference equation as a Taylor series about the (j, n) grid point. This leads to an infinite order partial differential equation (PDE), which is the actual PDE which is being solved by the finite-difference equation. This equation will be termed the Equivalent Partial Differential Equation (EPDE). If this procedure

is applied to the FTCS equation (2.1.9) above, the EPDE can be shown to be

$$\frac{\partial\tau}{\partial t}+\frac{\Delta t}{2}\frac{\partial^2\tau}{\partial t^2}-\alpha\frac{\partial^2\tau}{\partial x^2}+\frac{(\Delta t)^2}{6}\frac{\partial^3\tau}{\partial t^3}+\frac{(\Delta t)^3}{24}\frac{\partial^4\tau}{\partial t^4}-\frac{\alpha(\Delta x)^2}{12}\frac{\partial^4\tau}{\partial x^4}$$
$$+\frac{(\Delta t)^4}{120}\frac{\partial^5\tau}{\partial t^5}+\frac{(\Delta t)^5}{720}\frac{\partial^6\tau}{\partial t^6}-\frac{\alpha(\Delta x)^4}{360}\frac{\partial^6\tau}{\partial x^6}+\cdots=0, \qquad (2.2.1)$$

where all the derivatives are evaluated at the (j,n) grid point.

The second step is to remove all the time derivative terms from (2.2.1) with the exception of $\partial\tau/\partial t$. This is achieved by repeatedly differentiating the latest form of equation (2.2.1) itself, and adding an appropriate multiple of this back into itself (as it was before it was differentiated) to remove the desired terms. Thus to remove the $\partial^2\tau/\partial t^2$ term, equation (2.2.1) is differentiated once with respect to t, multiplied by $-\Delta t/2$ then added back into the original equation. Note that the terms must be eliminated in order of *increasing* total derivatives, and within an order of total derivatives in order of increasing space derivatives, or else the process will re-introduce previously eliminated terms back into the equation. Thus the first term removed is $\partial^2\tau/\partial t^2$, followed by $\partial^2\tau/\partial t\partial x$ and so on. The resulting equation is termed the Modified Equivalent Partial Differential Equation (MEPDE), and should contain the original partial differential equation as well as some error terms, if the finite-difference equation is consistent with the diffusion equation (2.1.1). If the original PDE is *not* recovered in the MEPDE as the grid spacing $\Delta x, \Delta t \to 0$, then the finite-difference equation is not consistent with the PDE, and so the method is of no practical use.

The MEPDE for the FTCS equation (2.1.9) is thus

$$\frac{\partial\tau}{\partial t}-\alpha\frac{\partial^2\tau}{\partial x^2}+E(s,\alpha,\Delta x,\tau)=0, \qquad (2.2.2)$$

where

$$E(s,\alpha,\Delta x,\tau) = \frac{\alpha(\Delta x)^2}{12}(6s-1)\frac{\partial^4\tau}{\partial x^4}$$
$$+\frac{\alpha(\Delta x)^4}{360}(-120s^2+30s-1)\frac{\partial^6\tau}{\partial x^6}+O\{(\Delta x)^6\}. \qquad (2.2.3)$$

From equations (2.2.2) and (2.2.3) it is seen that the FTCS equation (2.1.9) is consistent with equation (2.1.1) and that the errors involved in using the FTCS equation

are of $O\{(\Delta x)^2\}$. This method is therefore called "second-order accurate", or, in less formal terms, it is simply referred to as a "second-order method". This is all in accordance with the known behaviour of this equation, as is the fact that the leading error term vanishes for the value $s = 1/6$, making the method fourth-order accurate for this particular case (Richtmyer and Morton, 1967). Note that in this work, the term "high-order" will be used to mean a method of high-order accuracy, unless otherwise specified.

In general, the modified equivalent equation for a finite-difference equation that is consistent with the one-dimensional diffusion equation (2.1.1) can be written in the form

$$\frac{\partial \tau}{\partial t} - \alpha \frac{\partial^2 \tau}{\partial x^2} + \sum_{p=0}^{\infty} C_p \frac{\partial^p \tau}{\partial x^p} = 0, \tag{2.2.4}$$

where the condition

$$\lim_{\Delta x, \Delta t \to 0} C_p = 0, \quad p \geq 0 \tag{2.2.5}$$

must hold for consistency. In addition, however, the condition

$$C_p = 0 \quad \text{for } p \leq 2 \tag{2.2.6}$$

is desirable so that the leading error term is at least first order and hence of smaller magnitude than the solution being sought. It is found that the coefficients C_p can be written in the form

$$C_p = \frac{2\alpha(\Delta x)^{p-2}}{p!} \Gamma_p(s) \ , \quad p > 2 \ . \tag{2.2.7}$$

As mentioned by Warming and Hyett (1974), the original PDE must *not* be used in place of the current EPDE to remove the time derivative terms during the "modification" process, since the EPDE represents the finite-difference equation, and in general a solution to the original PDE will not satisfy the finite-difference equation.

The modified equivalent equation for the FTCS equation can be written in the general form (2.2.7), with

$$C_4 = \frac{\alpha(\Delta x)^2}{12}(6s - 1) \tag{2.2.8}$$

which means that in this case

$$\Gamma_4(s) = 6s - 1. \tag{2.2.9}$$

It should be noted that is the leading error term in the MEPDE is C_{q+2} then the method is order q accurate, due to the form of the coefficient given by (2.2.7).

It has also been found (Noye, 1984) that the error terms in the MEPDE can be classified into two sets. Those error terms that are associated with *odd* order derivatives, such as C_3, C_5, etc., represent errors in the wave speed of the solution. Since the diffusion equation does not translate the solution, these errors are better interpreted as a spatial shift of the quantity τ under consideration. It is also worth noting that in the case where the spatial differencing is kept centred about the j^{th} grid point, all of these error terms are automatically zero, since the terms from the Taylor series that produce these error terms cancel out when they are added together. Such centred equations thus involve no artificial translation of the quantity τ. This explains the absence of such error terms in the modified equivalent equation (2.2.3) for the FTCS method above, since the space derivative was approximated by a centred-space difference approximation.

The error terms associated with *even* order derivatives, like C_2, C_4, etc., represent errors in the amplitude of the numerical solution, which means that the numerical solution incorporates either more or less diffusion than is actually present in the original PDE. There is no method of differencing that will force all of these to be zero for the diffusion equation (2.1.1), due to the nature of the equation itself and the relationships between the terms of the Taylor Series of the finite-difference equation.

The computation of the modified equivalent equation corresponding to a given finite-difference equation is in fact a mechanical operation that lends itself to being programmed on a computer. Such a program, written in Pascal under the VAX/VMS operating system, has been developed to produce both the EPDE and MEPDE for a given finite-difference equation. This program uses 32-bit integer arithmetic, and so is limited to a maximum of twelfth-order derivatives, although in practice an overflow can occur for tenth or even eighth-order derivatives for high-order accuracy equations. This program, as well as several others that have been developed into a sophisticated system for generating highly accurate finite-difference equations, is described in Appendix A. This system of programs allows the fast generation of finite-difference

equations, given only the desired approximations to each derivative in the PDE. Also, although these programs were developed to run on a VAX computer, they have been successfully transported to other systems, including various micro-computers such as the Apple Macintosh and machines running the popular CP/M-80 and MS-DOS operating systems. The limitation on the maximum derivative order can be overcome by using extended precision integer arithmetic routines, but these must be coded as part of the program on all the machines where the programs have been implemented, since the required routines are not implemented by the machines themselves. This slows the computation down by a factor of approximately one hundred relative to the version which use integers of the size that the machines usually handle. Since the equations which require this extended precision are usually complicated high-order equations whose MEPDEs require significant amounts of CPU time to calculate anyway, this magnitude of speed reduction is impractical in most situations.

Returning to the modified equivalent equation, it can be seen that this is an extremely useful theoretical tool for determining the accuracy of a given finite-difference equation. It is also desirable, however, to be able to check that these theoretical predictions actually work in practice. In order to do this, it should be noted that the the total error involved in using a finite-difference scheme is dominated by the leading error term of the modified equivalent equation, as succeeding terms contain progressively higher powers of Δx, and thus get smaller. From equations (2.2.4) and (2.2.7), the leading error term at the (j,n) grid point for an order q accurate method can be written in the form

$$\frac{2\alpha(\Delta x)^q}{(q+2)!}\Gamma_{q+2}(s)\frac{\partial^{q+2}\tau}{\partial x^{q+2}}\bigg|_j^n. \tag{2.2.10}$$

Thus for an order q accurate method with a sufficiently small Δx and assuming that the derivative factor is approximately constant, the discretisation error is dominated by the leading error term (2.2.10), so

$$|e| \propto (\Delta x)^q \tag{2.2.11}$$

for a constant s, where e is the discretisation error $(\hat{\tau} - \tau)$ for the method. This leads

to

$$|e| \approx K(\Delta x)^q \tag{2.2.12}$$

$$\Rightarrow \log\{|e|\} \approx q\log\{\Delta x\} + K' \tag{2.2.13}$$

where K is the constant of proportionality in (2.2.11) and $K' = \log\{K\}$. From (2.2.13), it can be seen that a graph of $\log\{|e|\}$ plotted against $\log\{\Delta x\}$ for constant s should produce a straight line of slope q for an order q method (see Noye, 1984). In this work, we plot $-\log\{|e|\}$ against $-\log\{\Delta x\}$ in order to keep most of the numbers positive, as shown in Figure 2.2. In some cases, the points corresponding to small values of J, say 20 or 30, are somewhat off the straight line generated by the remaining points. This is caused by the value of Δx being large enough that higher order error terms make a significant contribution to the discretisation error, which invalidates the above theory. In such cases, these erroneous points are not included in the graphs. Likewise, in the case of sixth or eighth-order methods, the errors generated for large values of J are often smaller than the 14 digits of precision available on the VAX where the tests were done. Again, this leads to erroneous results, due to "subtractive cancellation" between the exact and approximate solutions, and so these points are also excluded in the graphs, to avoid giving a misleading impression.

As another comparison, we can consider the amount of CPU time used. Let $C(\Delta x)$ be the amount of CPU time used in going from one time level to the next, using a space step of Δx and a time step of Δt. The total amount of CPU time required to reach the desired time level, Cp, is given by

$$Cp(\Delta x, \Delta t) = N \times C(\Delta x), \tag{2.2.14}$$

where N is the number of time steps required. If we wish to change the space step to $r\Delta x$, then in order to keep s constant, we also need to change the time step to $r^2\Delta t$. Since the number of operations per time level depends linearly on the number of grid points (at least for the solution methods considered in this work), it can be seen that

$$C(r\Delta x) = \frac{C(\Delta x)}{r}, \tag{2.2.15}$$

since there are now $1/r$ times as many grid points to find values for at the next time level. Thus if $r > 1$ we now have fewer grid points at which to compute values, and

less CPU time will be used. Also, since the time step has been changed, a different number of time steps, N', will now be required to reach the desired time level, where

$$N' = N/r^2. \tag{2.2.16}$$

Overall it can be deduced that

$$Cp(r\Delta x, r^2\Delta t) = \frac{Cp(\Delta x, \Delta t)}{r^3}. \tag{2.2.17}$$

From this relation it follows that

$$\begin{aligned} Cp &\propto 1/(\Delta x)^3 \\ \Rightarrow \quad \log\{Cp\} &= -3\log\{\Delta x\} + \log\{K\} \\ \text{Now} \quad -\log\{\Delta x\} &= -(1/q)\log\{|e|\} + K_1 \quad \text{from above} \\ \Rightarrow \quad -\log\{|e|\} &= (q/3)\log\{Cp\} + K' \end{aligned}$$

where the K terms represent constants. Thus a graph with $-\log\{|e|\}$ plotted against $\log\{Cp\}$, which will be denoted as a $-\log\{|e|\}$ vs $\log\{Cp\}$ graph, should also give a straight line, although in this case the slope of the line is one third of the order of the method. It should be noted however that of the two graphs, the error vs grid spacing one is likely to be more accurate, since CPU time on a time-sharing system, such as the VAX 11-785 used for this work, can be affected by system overheads, like the amount of paging done in response to system load.

One other comparison that can be used is to compare the amount of CPU time that is required to generate solutions of a given accuracy, which corresponds to an intuitive notion of the "efficiency" of the solution. This is only of use for methods of similar orders of accuracy, since all the numerical results are for values in the range $10 \leq J \leq 100$, so a low-order method will never be tested on a fine enough grid to give the same accuracy in the answers as a high-order method produces for even a very coarse grid. Despite this limitation, however, this type of comparison can be very useful.

The finite-difference equations are tested using initial and boundary conditions that provide a known analytic solution to the diffusion equation, which allows the numerical errors from the method to be found from comparison with the exact solution. The exact solution to (2.1.1) used for the numerical tests is the Gauss peak defined by

$$\hat{\tau}(x,t) = \frac{1}{\sqrt{4t+1}} \exp\left\{-\frac{(x-a)^2}{\alpha(4t+1)}\right\}, \tag{2.2.18}$$

where the constant $a = 0.5$ so the peak is centred in the spatial domain of $[0, 1]$. Equation (2.2.18) is used to define the initial and boundary conditions, as well as the exact solution to the problem, by substituting the appropriate values for x and t. The values for the graphs in this work are taken at the point $x = 0.2$, so as to avoid the regions of relatively small slope at both ends and near $x = 0.5$. In such regions the numerical solution seems to be more accurate than over the remainder of the spatial domain, so these regions are avoided to give a better reflection of the true accuracy of the method. The methods are allowed to run to $T = 8$, so that any instabilities or inaccuracies in the equation should be apparent in the results. These values at the point $(0.2, 8)$ can then be used to generate the graphs described above, and allow easy comparison of the difference equations. The exact solution to the test problem at this point and time, using the value $\alpha = 0.01$ is $\hat{\tau} = 0.13....$

If these graphs are drawn for the FTCS equation (2.1.9), it is found that they are indeed straight lines, with slopes as predicted. Examples of these graphs are shown in Figures 2.2 and 2.3. These graphs show the line of best fit through the points for each s, and the slope of this line is the value of M listed in the legends of these graphs.

It is notable from the CPU graph, Figure 2.3, the the amount of CPU time required to generate a solution to a specified accuracy is minimised amongst the second-order cases by the values $s = 0.1$ and $s = 1/3$, which are not the largest values shown in the graph. This is an indication that the dominant error term is much smaller for these values of s than for larger values such as $s = 1/2$. This type of efficiency (ie. the amount of CPU time required to generate a solution to a given accuracy) is a good method of comparing solution techniques, unless there are constraints on either the minimum grid spacing required in the solution or the total amount of CPU time allowed. In such cases, this measure is of less use.

Also of note from the figures is the very much more accurate and efficient solution obtained for the special case $s = 1/6$, where the difference equation is fourth-order accurate. Such results as this indicate enormous gains in accuracy and CPU time usage that may be obtained from the development and use of methods of higher orders of accuracy.

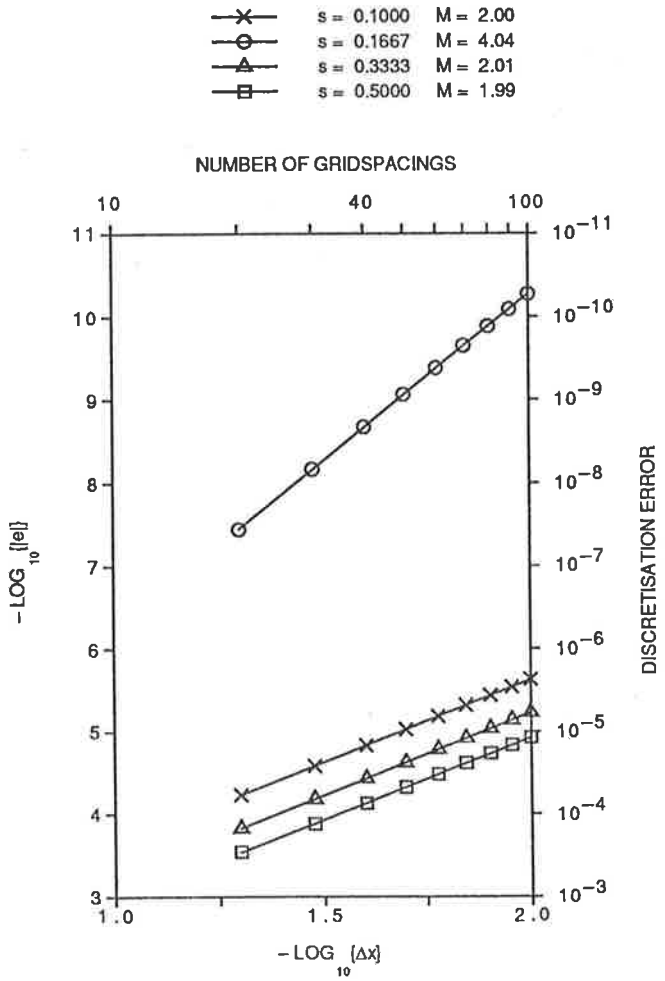


Figure 2.2: *Error vs grid spacing graph for the (1,3) FTCS method (2.1.9)*

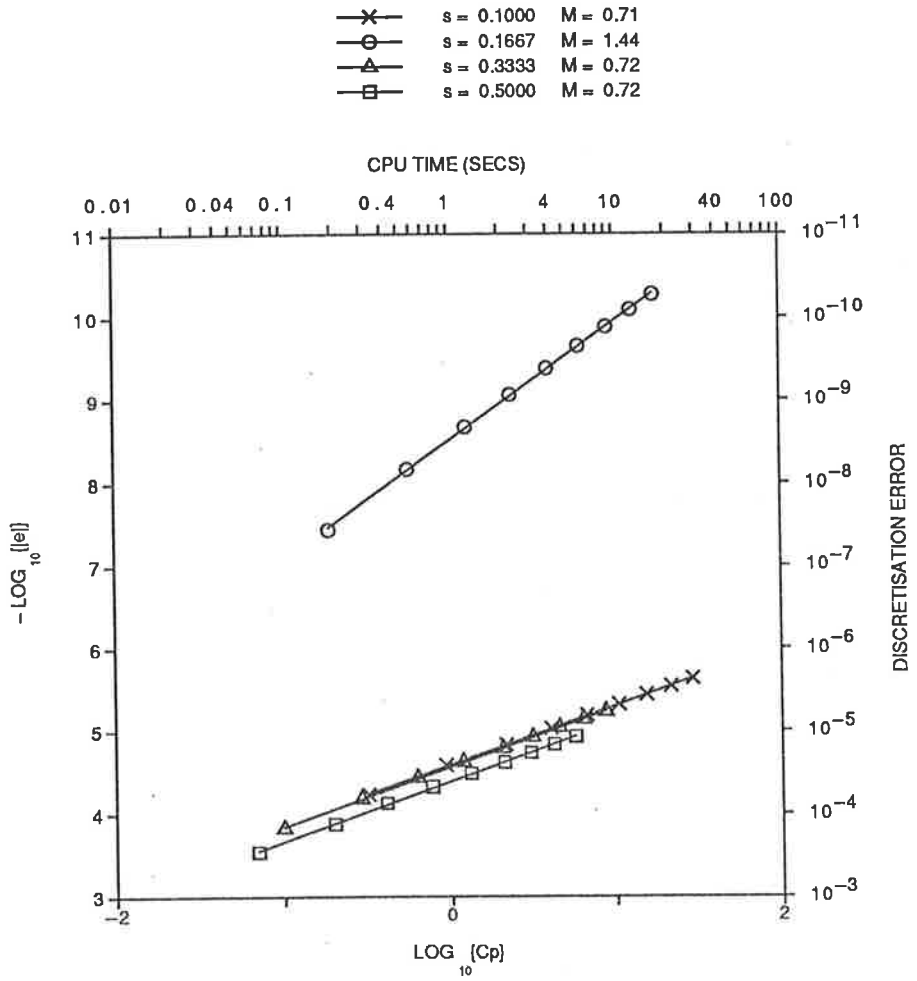


Figure 2.3: *Error vs CPU time graph for the (1,3) FTCS method (2.1.9)*

2.3 Finding Methods of Improved Accuracy

Having found a practical way to compare the errors of different finite-difference equations, we can now look at ways in which more accurate equations can be generated. Higher order equations are desired both because in any practical situation they produce more accurate answers for a given grid spacing, and also because a reduction in the grid spacing produces a much greater increase in accuracy than the same reduction using lower order methods.

Having theoretically developed such higher order methods, it is necessary to run them in practice in order to check that a solution can be generated that is more accurate than the solutions obtained from lower order methods, such as the (1,3) FTCS equation, and also to compare the methods of the same order with one another. Such a comparison is in fact two-fold; one in terms of the absolute error obtained for a given grid spacing, and the other in terms of the amount of CPU time required to find a solution to a given accuracy.

In order to achieve these comparisons, the graphs of $-\log\{|e|\}$ vs $-\log\{\Delta x\}$ and $-\log\{|e|\}$ vs $\log\{Cp\}$, which were discussed in Section 2.2, are generated. As shown there, these graphs should be straight lines with slopes depending on the order of the method.

2.3.1 Using Higher Order Derivative Approximations

The simplest way to try to produce higher order solution finite-difference methods is to employ higher order approximations to the derivative terms in the original PDE (2.1.1). If only two time levels are to be involved in the finite-difference equation, then it is not possible to increase the accuracy of the time derivative approximation. This leaves only the space derivative to be improved, and this can be approximated to fourth-order, rather than the second-order approximation (2.1.7), if five grid points are used instead of three. This produces an equation that uses the (1,5) computational

stencil shown in Figure 2.4. The five-point spatial derivative approximation is

$$\left.\frac{\partial^2 \hat{\tau}}{\partial x^2}\right|_j^n = \frac{-\hat{\tau}_{j-2}^n + 16\hat{\tau}_{j-1}^n - 30\hat{\tau}_j^n + 16\hat{\tau}_{j+1}^n - \hat{\tau}_{j+2}^n}{12(\Delta x)^2} + O\{(\Delta x)^4\}, \tag{2.3.1}$$

which can be used in place of (2.1.7) in (2.1.8) to produce the new finite-difference equation. This equation, developed by Noye (1984), can be written as

$$12\tau_j^{n+1} = -s(\tau_{j-2}^n + \tau_{j+2}^n) + 16s(\tau_{j-1}^n + \tau_{j+1}^n) + (12 - 30s)\tau_j^n. \tag{2.3.2}$$

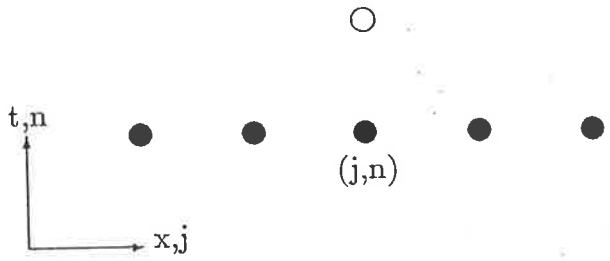


Figure 2.4: *The (1,5) computational stencil*

This equation, having been derived from a fourth-order approximation to the space derivative, may be expected to be more accurate than the three-point FTCS equation (2.1.9). However, when the modified equivalent equation corresponding to (2.3.2) is calculated, the leading error terms are found to involve the factors

$$\begin{aligned} \Gamma_4(s) &= -6s, \\ \Gamma_6(s) &= 4(30s - 1). \end{aligned} \tag{2.3.3}$$

Since (2.3.1) is a centred approximation for the space derivative term, there are no odd-order error terms in the modified equivalent equation.

Despite having used a fourth-order approximation to the space derivative, the resulting finite-difference equation is still only second-order accurate. Also, since $\Gamma_4(s)$ cannot be forced to be zero for any value of $s > 0$, there is no optimal value of s for which the method becomes fourth-order.

The lack of accuracy of the equation (2.3.2) is due to the use of the first-order approximation to the time derivative. The truncation error of $O\{\Delta t\}$ associated with this

approximation is of $O\{(\Delta x)^2\}$ for constant s, which accounts for the second-order nature of this equation. This problem may be overcome, as shown below in Section 2.3.2, by a more careful introduction of the higher order derivative approximation (2.3.1).

In practice, these theoretical predictions are confirmed, with the five-point method producing numerical results of better accuracy for $s < 1/12$ and worse for $s > 1/12$ when compared to the three-point FTCS equation (2.1.9). This is to be expected, since the magnitude of the leading error terms from the respective MEPDEs follows the same trend, as shown in Figure 2.5.

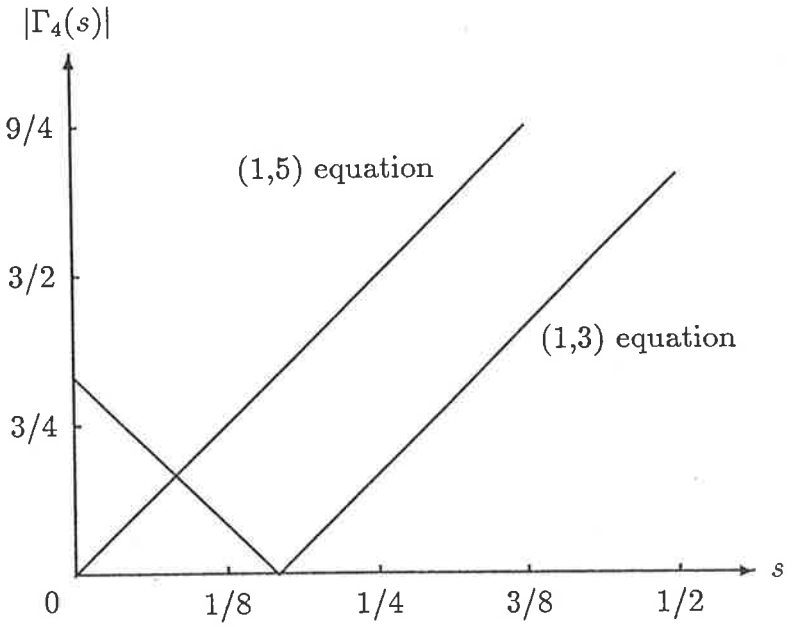


Figure 2.5: *Leading error terms for the (1,3) FTCS method and the standard (1,5) equation.*

From the above, it is apparent that merely using higher order approximations to some of the derivative terms in the original PDE does not necessarily lead to higher order finite-difference equations. A more sophisticated approach is clearly needed to produce finite-difference schemes with a high order of accuracy.

2.3.2 Eliminating the Leading Error Terms

Another approach to generating higher order methods is to generate equations which have some free parameters or weights involved in them, and then eliminate as many of the leading error terms as possible from the modified equivalent equation by choosing suitable values for these weights.

For example, neither the three-point approximation (2.1.7) nor the five-point version (2.3.1) produce a fourth-order method, so we now try a weighted combination of the two, namely

$$\left.\frac{\partial^2 \tau}{\partial x^2}\right|_j^n \approx (1-\varphi) \times [\text{CS3 at } (j,n)] + \varphi \times [\text{CS5 at } (j,n)] \tag{2.3.4}$$

where CS3 is the three-point centred-space approximation (2.1.7) and CS5 is the five-point approximation (2.3.1). If this combination is used to approximate the space derivative in (2.1.1), the resulting finite-difference equation is

$$\begin{aligned} 12\tau_j^{n+1} &= -s\varphi(\tau_{j-2}^n + \tau_{j+2}^n) \\ &+ \{4s(3+\varphi)\}(\tau_{j-1}^n + \tau_{j+1}^n) + \{6(2-5s+(1-\varphi)s)\}\tau_j^n. \end{aligned} \tag{2.3.5}$$

This equation uses the same computational stencil as that used by the (1,5) method developed in the previous section (see Figure 2.4).

The modified equivalent equation corresponding to equation (2.3.5) can be written in the general form (2.2.4) with leading errors involving the terms

$$\begin{aligned} \Gamma_4(s) &= 6s - 1 + \varphi, \\ \Gamma_6(s) &= -1 + 5\varphi(1-6s) + 30s(1-4s). \end{aligned} \tag{2.3.6}$$

Again, since both of the difference approximations used for the space derivative are centred about the j^{th} grid point, the odd-order error terms are automatically eliminated from the MEPDE. The weightings that are used in this work are kept centred for this reason, since using non-centred approximations would mean that more weights would be needed to eliminate the odd-order errors.

From equations (2.3.6), it can be seen that the choice

$$\varphi = 1 - 6s \tag{2.3.7}$$

for the weight φ will force $\Gamma_4(s) = 0$ and thus eliminate the term involving C_4 from the modified equivalent equation. This choice should then produce a method which is fourth-order accurate. If this substitution is made, the finite-difference equation becomes

$$\begin{aligned} 12\tau_j^{n+1} &= \{s(6s-1)\}(\tau_{j-2}^n + \tau_{j+2}^n) \\ &+ \{8s(2-3s)\}(\tau_{j-1}^n + \tau_{j+1}^n) + \{6(2-5s+6s^2)\}\tau_j^n \end{aligned} \tag{2.3.8}$$

which has a modified equivalent equation in the general form (2.2.4) with the leading error term involving the factor

$$\Gamma_6(s) = 4 - 30s + 60s^2. \tag{2.3.9}$$

This shows that the equation (2.3.8) is indeed fourth-order accurate. Since (2.3.7) is the only substitution that makes $\Gamma_4(s) = 0$, equation (2.3.8) is the "optimal" equation for this computational stencil, in the sense that it has the highest order of accuracy possible for this stencil. Thus the weighted combination of the two space derivative approximations has produced a new finite-difference equation that is fourth-order accurate, rather than the second-order accuracy obtained by using either approximation individually. This (1,5) equation is investigated in more detail, and its numerical results examined, in Section 2.6 below.

It appears from this that the technique of first finding a weighted equation, then deriving the corresponding modified equivalent equation and eliminating the leading error terms by a suitable choice of the values for the weights is an extremely useful method for producing high order finite-difference equations.

2.4 Numerical Stability

It seems likely from the above that very accurate finite-difference methods can be developed to solve the diffusion equation, and indeed this can be done in practice. There is no point, however, in developing highly accurate methods if they cannot be implemented to produce results of an acceptable accuracy in a reasonable amount

of CPU time. The ability to do this using a finite-difference equation depends on numerical stability of the difference equation. Thus as well as the accuracy of any given method, we also need to check its stability range, and in this work this check is done using the von Neumann method (O'Brien et. al., 1950).

Checking the stability of our finite-difference equations is also important due to Lax's Equivalence Theorem (Lax and Richtmyer, 1956), which states that "Given a properly posed linear initial value problem and a finite-difference approximation to it that satisfies the consistency condition, stability is the necessary and sufficient condition for convergence". Here "convergence" is used to mean that the solution of the finite-difference equation converges to the solution of the original PDE as the grid spacing tends to zero. Thus we require convergence if our solution is to be of any use, but this property is extremely difficult to prove directly. Instead, we use this theorem, since the diffusion equation with (reasonable) given initial and boundary conditions qualifies as a "properly posed linear initial value problem", which allows us to just prove stability and consistency, which is a much simpler task.

In the case of the FTCS equation (2.1.9) it is well known (Richtmyer and Morton, 1967) that the von Neumann stability restriction is

$$s \leq 1/2 \tag{2.4.1}$$

which, while being somewhat restrictive, allows large enough time steps to obtain results in a reasonable amount of computer time.

For the five-point method (2.3.2) it can be shown (Noye, 1984) that the von Neumann stability range is

$$s \leq 3/8 \tag{2.4.2}$$

which is more restrictive than the three-point FTCS equation (2.1.9). Thus the five-point equation (2.3.2) has a smaller stability range than the three-point equation; in addition it is no more accurate, and also requires special treatment next to the boundaries, since the equation then involves points that are outside the solution domain. Consequently, this method is of little practical use as a solution method and will not be considered further in this work.

For the fourth-order equation (2.3.8), the von Neumann stability range is

$$s \leq 2/3 \tag{2.4.3}$$

(Noye and Hayman, 1986a) which is a larger region than either of the other two equations. This equation, like the basic five-point equation, requires special treatment next to the boundaries, since the computational stencil extends beyond the boundaries when applied to find either τ_1^{n+1} or τ_{J-1}^{n+1}. The equation (2.3.8), however, has both greater accuracy and greater numerical stability than either of the other methods considered so far, and so the extra work to implement it in practice is worthwhile.

Since many finite-difference equations have von Neumann amplification factors which are extremely difficult to obtain in analytic form, many of the stability regions obtained in this work have been found numerically. The computer program to do this (Steinle, 1984) takes a range of values of s and a range of values for one weight, splits each range up into a specified number of subintervals, then for each (s, weight) pair calculates the von Neumann amplification factor, G, for a range of wave numbers. If any of these G values has

$$|G| > 1 \tag{2.4.4}$$

then the equation in unstable for that (s, weight) pair, otherwise it is stable. This program gives a practical way of finding the von Neumann stability range for even extremely complicated finite-difference equations.

2.5 (1,3,3) Methods

The fourth-order equation (2.3.8) is more accurate than the second-order methods, such as the FTCS equation (2.1.9), generally in use at the present time. Before exploring this method further, other ways of achieving fourth-order accuracy will be examined. The advantages of the optimal (1,5) method, namely that it is explicit and has a large stability range, should, if possible, be kept in a new technique. The (1,5) method does however have problems dealing with grid points next to a boundary (this is described fully in Section 2.6), and a new technique should be sought to avoid these.

In an attempt to do this, the three-level (1,3,3) stencil shown in Figure 2.6 is used. This gives an equation which is still explicit, so the advantages of quick computation are still present; it has one extra grid point, which means that a fourth-order scheme should be possible; it is also only three grid-points wide, so it will not require external grid points when used next to the boundaries. Against this, however, is the fact that since the stencil uses grid points from three time levels, some other scheme must be used for the initial time step. This starter method should ideally be of the same order of accuracy as the three-level method so as not to detract from the final numerical results. For example, the fourth-order (1,5) equation (2.3.8) could be used to start a fourth-order three-level method.

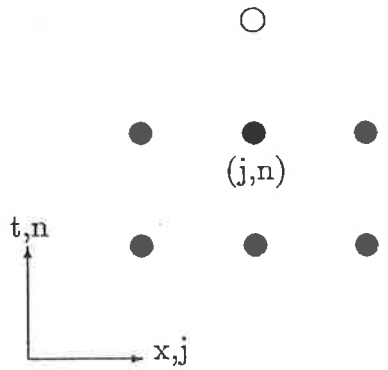


Figure 2.6: *Computational stencil for the (1,3,3) method*

The (1,3,3) stencil, shown in Figure 2.6, allows the introduction of three weights. This is done by differencing the derivative terms in (2.1.1) as follows:

$$\left.\frac{\partial \tau}{\partial t}\right|_j^n \approx \gamma \times \{[\text{BT at } (j-1,n)] + [\text{BT at } (j+1,n)]\}$$
$$+ \lambda \times [\text{FT at } (j,n)] + (1-2\gamma-\lambda) \times [\text{CT at } (j,n)], \qquad (2.5.1)$$
$$\left.\frac{\partial^2 \tau}{\partial x^2}\right|_j^n \approx \varphi \times [\text{CS at } (j,n)] + (1-\varphi) \times [\text{CS at } (j,n-1)], \qquad (2.5.2)$$

where BT, CT and FT represent the backward, centred and forward-time difference approximations to $\partial\hat{\tau}/\partial t$ respectively, and CS denotes the three-point centred-space differencing (2.1.7). The finite-difference equation corresponding to this differencing is

$$\begin{aligned}\{2\gamma-\lambda-1\}\tau_j^{n+1} &= 2\{\gamma-\varphi s\}(\tau_{j-1}^n+\tau_{j+1}^n)+2\{2\varphi s-\lambda\}\tau_j^n \\ &+ 2\{\varphi s-s-\gamma\}(\tau_{j-1}^{n-1}+\tau_{j+1}^{n-1}) \\ &+ \{4s(1-\varphi)+2\gamma+\lambda-1\}\tau_j^{n-1}. \qquad (2.5.3)\end{aligned}$$

Since the distribution of weights was chosen to keep the equation spatially centred, it is expected that the modified equivalent equation corresponding to (2.5.3) will have no odd-order error terms, and indeed the modified equivalent equation, written in the general form (2.2.4) has leading error term involving the factor

$$\Gamma_4(s) = 12\gamma(1-s)+12s-12\varphi s+6\lambda s-1, \qquad (2.5.4)$$

and all the odd-order coefficients are zero.

It can be seen by rearranging equation (2.5.4) that the substitution

$$\varphi = 1-\frac{1}{12s}+\frac{\lambda}{2}+\frac{\gamma(1-s)}{s} \qquad (2.5.5)$$

will make $\Gamma_4(s)=0$ and hence remove the C_4 error term from this modified equivalent equation. This produces an equation which is fourth-order accurate, namely

$$\begin{aligned}-6\{1+(\lambda-2\gamma)\}\tau_j^{n+1} &= \{1-12s-6s(\lambda-2\gamma)\}(\tau_{j-1}^n+\tau_{j+1}^n)\} \\ &+ \{-2+24s+12(s-1)(\lambda-2\gamma)\}\tau_j^n \\ &+ \{6s(\lambda-2\gamma)-1\}(\tau_{j-1}^{n-1}+\tau_{j+1}^{n-1}) \\ &+ \{6(1-2s)(\lambda-2\gamma)-4\}\tau_j^{n-1}. \qquad (2.5.6)\end{aligned}$$

This accuracy of this equation can be verified by finding its modified equivalent equation, which has a leading error involving

$$\Gamma_6(s) = 3/2-15s+60s^2+15s(\lambda-2\gamma)(6s-1). \qquad (2.5.7)$$

By rearranging (2.5.7), $\Gamma_6(s)$ can also be made zero, which would then give a sixth-order equation. It should be noted, however, that despite having two weights left in

equation (2.5.6), it cannot be made eighth-order, since the two weights always occur together in the expression $\lambda - 2\gamma$, and to make it sixth order requires setting

$$\lambda - 2\gamma = \frac{1 - 10s + 40s^2}{10s(1 - 6s)}. \tag{2.5.8}$$

It follows that making the equation sixth-order removes both remaining weights from the equation. The sixth-order equation is

$$\begin{aligned} 6\{20s^2 - 1\}\tau_j^{n+1} &= 4s\{1 - 30s + 120s^2\}(\tau_{j-1}^n + \tau_{j+1}^n) \\ &+ 4\{-3 + 28s - 60s^2 - 240s^3\}\tau_j^n \\ &+ 4s\{60s^2 - 1\}(\tau_{j-1}^{n-1} + \tau_{j+1}^{n-1}) \\ &+ 2\{3 - 56s + 300s^2 - 240s^3\}\tau_j^{n-1}. \end{aligned} \tag{2.5.9}$$

A desirable feature of a finite-difference equation, although clearly of lesser importance than good accuracy, is a large stability range, which allows the spatial resolution to be increased without necessarily having to use much smaller time steps. In fact, the stability region for this equation is

$$s \leq 1/6 \tag{2.5.10}$$

which is only one quarter of that for the fourth-order (1,5) method.

This small stability range means that in order to increase the resolution of the solution (ie. decreasing Δx), the time step must be reduced accordingly, which means that the method may require more CPU time to generate a solution than another method with a larger stability range. Nevertheless, since the method is sixth-order, even relatively coarse grids give very accurate results, so refining the grid spacing may not be required, in which case this equation is very good.

The results obtained by this method, shown in Figures 2.7 and 2.8, are extremely accurate when compared with those from methods of lower order, except for the case of $s = 1/6$. The relatively poor accuracy in this case is explained from the form of the modified equivalent equation of (2.5.9), which has the leading error coefficient

$$C_8 = -\frac{\alpha(\Delta x)^6}{302400}\left\{\frac{50400s^4 - 12600s^3 - 1260s^2 + 330s - 13}{1 - 6s}\right\}, \quad s \neq 1/6. \tag{2.5.11}$$

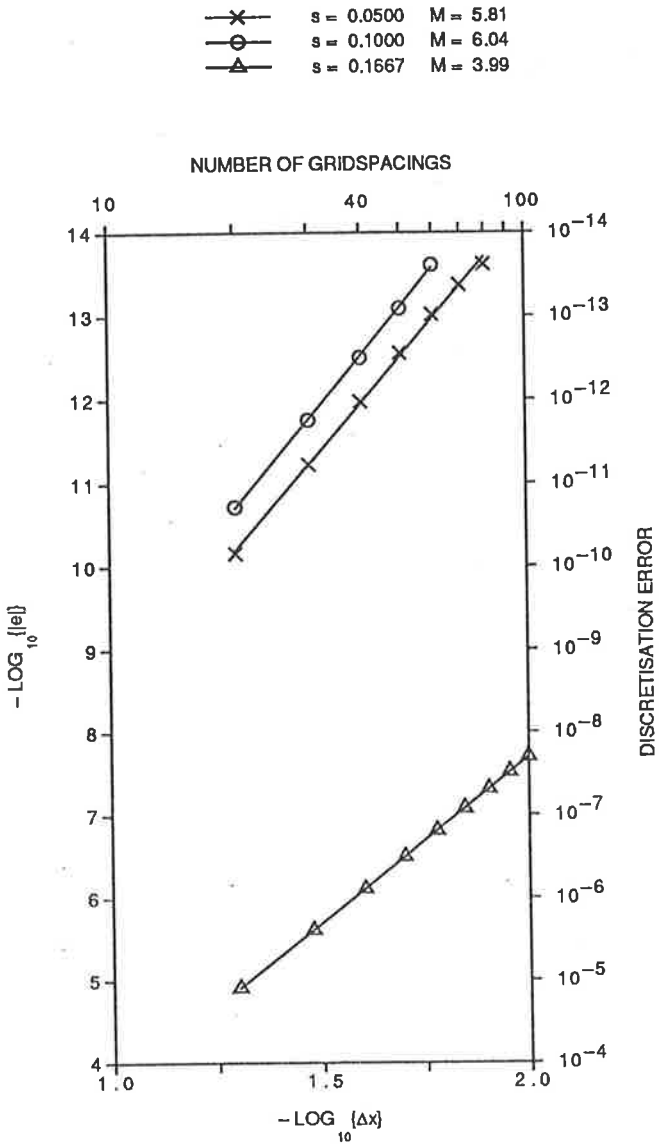


Figure 2.7: *Error vs grid spacing graph for the sixth-order (1,3,3) method (2.5.9)*

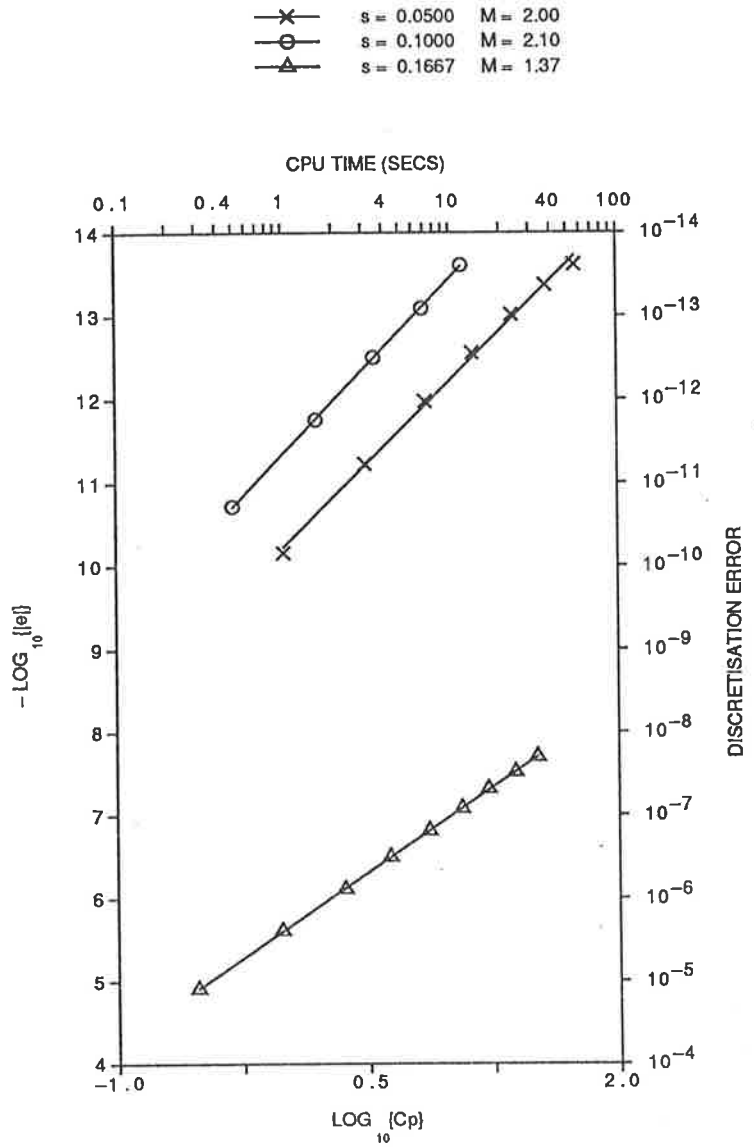


Figure 2.8: *Error vs CPU time graph for the sixth-order (1,3,3) method (2.5.9)*

The denominator term here, $1-6s$, is the result of normalising the modified equivalent equation. If the modified equivalent equation is considered *before* this normalisation process, this term multiplies both $\partial\tau/\partial t$ and $\partial^2\tau/\partial x^2$. Thus at $s = 1/6$, the modified equation does not contain these derivatives, and the finite-difference equation is consistent with some other partial differential equation, not the one-dimensional diffusion equation. This causes the very bad results obtained from the numerical tests in this case. To avoid this problem, the value of s is restricted to exclude the value $s = 1/6$. Note that, for a given grid spacing and using the maximum allowable value of s in each case, the amount of CPU time required to obtain a solution with the (1,3,3) method is about four times that required by the (1,5) equation (see Section 2.6 below).

Against this, however, is the fact that the sixth-order (1,3,3) equation with $s = 0.1$ gives a more accurate answer in slightly less CPU time than the (1,5) equation with $s = 2/3$, which makes the (1,3,3) method more attractive if this advantage can be utilised. Thus whether or not the (1,3,3) equation is practical to use is dependent on any CPU and/or accuracy restrictions on the solution, such as a minimum spatial resolution being required.

To cover cases where the sixth-order (1,3,3) equation is not practical to use, further investigation of the weighted equation (2.5.3) is warranted. The choice of

$$\begin{aligned} \gamma &= s(\varphi - 1) \\ \lambda &= 2\varphi s \end{aligned} \tag{2.5.12}$$

leads to the well known equation of DuFort and Frankel (1953) , namely

$$\{1 + 2s\}\tau_j^{n+1} = 2s(\tau_{j-1}^n + \tau_{j+1}^n) + \{1 - 2s\}\tau_j^{n-1}. \tag{2.5.13}$$

This equation has the advantage that is is von Neumann stable for all $s > 0$, so there is no limit on the size of the time step that may be taken for a given grid spacing. In order to determine the accuracy of this equation, the modified equivalent equation is derived, and the leading error terms are found to contain the factors

$$\begin{aligned} \Gamma_4(s) &= 12s^2 - 1 \\ \Gamma_6(s) &= -(720s^4 - 120s^2 + 1) \end{aligned} \tag{2.5.14}$$

Thus the DuFort-Frankel equation is only second-order accurate in general, although it becomes fourth-order in the special case $s = 1/\sqrt{12}$, when the factor $\Gamma_4(s)$ vanishes. In the "optimal" case, however, the value of s is fixed, and so no use can be made of the extra stability range. In the general case, the equation is only second-order accurate, and as can be seen from $\Gamma_4(s)$, the magnitude of the error coefficient rises quadratically as s is increased. Thus the numerical error associated with this method increases enormously if s is increased to take advantage of the extra stability.

Although it has been successfully applied by many workers in the past, including Fromm and Harlow (1963) and Hung and Macagno (1966), the DuFort-Frankel equation is of little practical use due to its being only second-order accurate for general values of s.

A better method than DuFort-Frankel, although not as accurate as the sixth-order equation investigated earlier, is the fourth-order weighted equation (2.5.6). This equation can be re-written in terms of a single weight, by substituting

$$\theta = \lambda - 2\gamma, \tag{2.5.15}$$

giving the equation

$$\begin{aligned} -6\{1+\theta\}\tau_j^{n+1} &= \{1 - 12s - 6\theta s\}(\tau_{j-1}^n + \tau_{j+1}^n)\} \\ &+ \{-2 + 24s + 12\theta(s-1)\}\tau_j^n \\ &+ \{6\theta s - 1\}(\tau_{j-1}^{n-1} + \tau_{j+1}^{n-1}) \\ &+ \{6\theta(1-2s) - 4\}\tau_j^{n-1}\ , \quad \theta \neq -1. \end{aligned} \tag{2.5.16}$$

This equation is at least fourth-order accurate for all $s > 0$ (it includes the sixth-order equation discussed above as a special case) and still contains one free weight, which can be used to try to increase the stability of the method.

In the case $\theta = -1$, the value τ_j^{n+1} is eliminated from the finite-difference equation. What remains is in fact a fourth-order implicit equation, written at time levels n and $(n-1)$, which was first derived by Crandall (1955). This equation, which is unconditionally stable (Figure 2.9), is discussed in more detail in Section 2.8 below.

Equation (2.5.16) can now be entered into the numerical von Neumann stability program, to try to find a form for θ, which may depend on s, that will maximise the stability of the resulting method. The resulting plot, shown in Figure 2.9, shows that the stability range increases as $\theta > 0$ increases, although the range never exceeds

$$s < 1/2, \tag{2.5.17}$$

and only approaches this value slowly as θ becomes large.

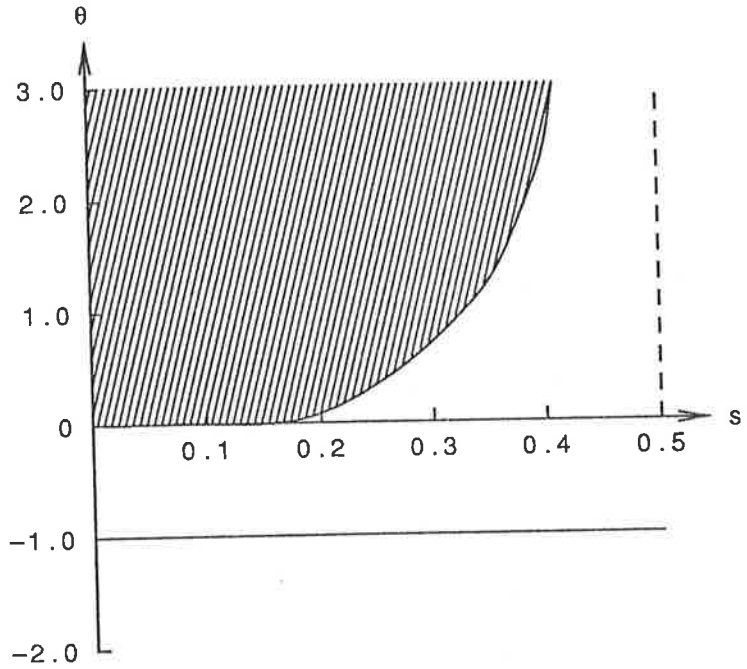


Figure 2.9: *Stability plot in the s-θ plane of the weighted (1,3,3) equation (2.5.16). The dashed line is the upper bound on the stability region for $\theta > 0$ (2.5.17).*

The problem with letting θ become large is, as for the DuFort-Frankel method above, use of the increased stability range will degrade the accuracy of the solution. To see this, the fourth-order error coefficient

$$C_6 = \frac{\alpha(\Delta x)^4}{240}(1 - 10s + 40s^2 + 10\theta s(6s - 1)) \tag{2.5.18}$$

is examined, and it is clear that for $s > 1/6$ this coefficient increases linearly with θ. Also, as s is increased for a given $\theta > 0$, the error coefficient increases quadratically. Thus the gain in stability, which was aimed at reducing the CPU time required, is offset by a loss of accuracy, which is unacceptable.

These theoretical expectations are clearly demonstrated by a numerical test of this technique. If the method is run with $\theta = 100$, which gives stability up to $s < 0.4+$, the results shown in Figures 2.10 and 2.11 are obtained.

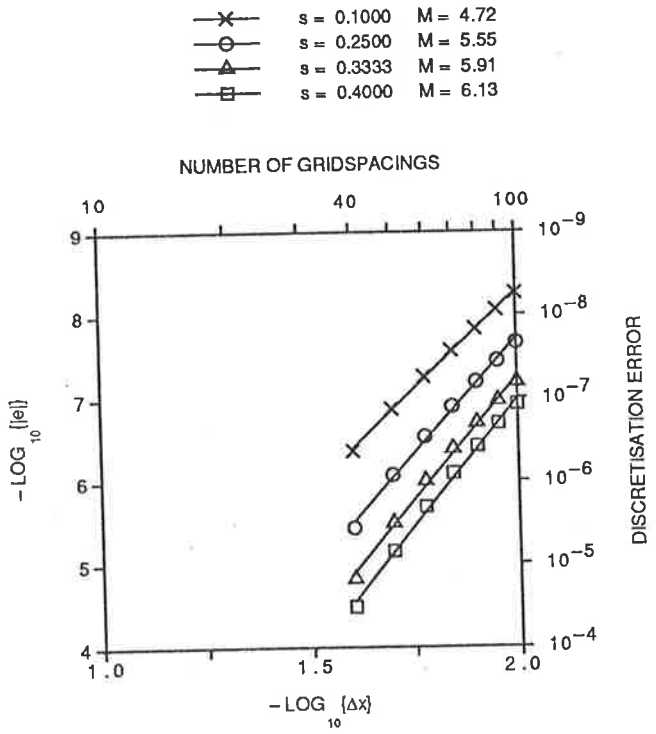


Figure 2.10: *Error vs grid spacing graph for the fourth-order (1,3,3) method (2.5.16), with* $\theta = 100$

Two points are obvious from these results. Firstly, the graphs, while being close to straight lines, have slopes somewhat larger than the expected values, which indicates that for this set of initial and boundary conditions some cancellation of errors is occurring. Secondly, and more importantly, is the drop in accuracy as s is increased towards the stability limit, which confirms the predictions that using the extra stability range would cause a loss of accuracy. This loss is not so great as to render the method useless, however, since s is constrained by the stability limit, so depending on the circumstances of the solution its use may be considered.

Overall, both the sixth-order and the weighted fourth-order (1,3,3) equations provide practical solutions, although the practicality of each is determined by the requirements at the time of solution. Note that while the stability of the fourth-order (1,3,3) equa-

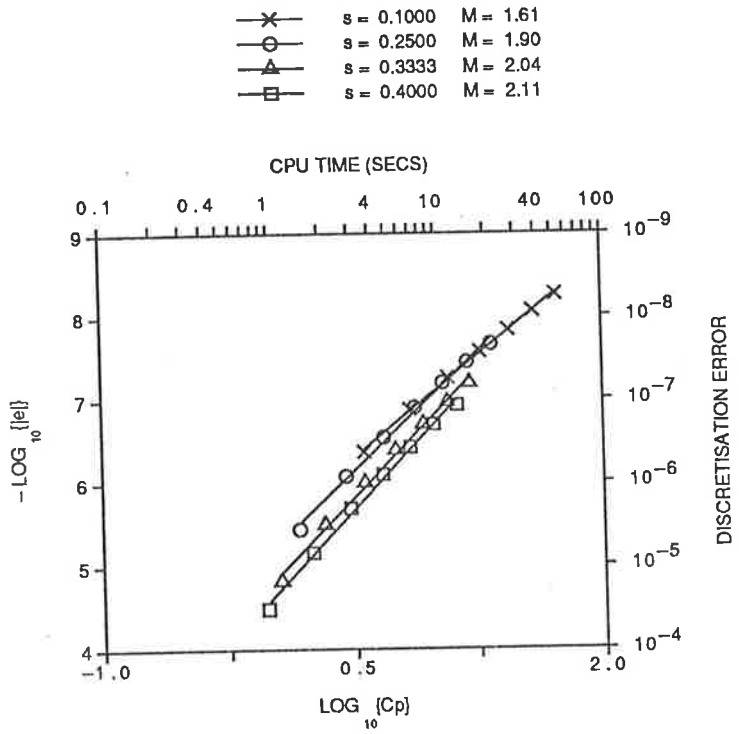


Figure 2.11: *Error vs CPU time graph for the fourth-order (1,3,3) method (2.5.16), with* $\theta = 100$

tion is somewhat less than that for the (1,5) equation, the (1,3,3) equation does not require special treatment near the boundaries, which may be an advantage in some situations.

2.6 The Optimal (1,5) Method

As discussed in Section 2.5 above, an alternative explicit method to the fourth-order (1,3,3) equation is the fourth-order (1,5) equation (2.3.8), namely

$$\begin{aligned} 12\tau_j^{n+1} &= \{s(6s-1)\}(\tau_{j-2}^n + \tau_{j+2}^n) \\ &+ \{8s(2-3s)\}(\tau_{j-1}^n + \tau_{j+1}^n) + \{6(2-5s+6s^2)\}\tau_j^n \end{aligned} \tag{2.6.1}$$

However, there are problems with implementing this equation next to the boundaries, since the equation then involves grid points outside the solution domain.

Nevertheless, it does avoid the stability and error trade-off encountered with the

fourth-order (1,3,3) method. To show this, we find the von Neumann amplification factor for the (1,5) equation, which is given by

$$G(s,\cos\beta) = \frac{s(6s-1)}{3}\cos^2\beta + \frac{4s(2-3s)}{3}\cos\beta + \frac{3-7s+6s^2}{3} \qquad (2.6.2)$$

where $\beta = m\pi\Delta x$ and m is a wave number. For von Neumann stability, we require

$$|G| \leq 1 \quad \text{for all } \beta. \qquad (2.6.3)$$

In order to satisfy (2.6.3), given that $s > 0$ for the equation to be of practical use, we require

$$0 < s \leq 2/3, \qquad (2.6.4)$$

which is the result stated in Section 2.4 above. The complete derivation of this result is given by Noye and Hayman (1986a). This is the largest stability range of any of the methods developed so far, except for the DuFort-Frankel. More importantly, the numerical error is independent of the stability limit, so the method should produce reasonably accurate results even if $s = 2/3$ is chosen.

Since the equation (2.3.8), when applied at $j = 1$ or $j = J-1$, involves grid points outside the solution domain, an alternative method must be employed to find the values at τ_1^{n+1} and τ_{J-1}^{n+1}.

This method must be of the same accuracy as that used for the rest of the solution domain, and should also be von Neumann stable over at least the range defined in (2.6.4). One way to create such a method is to approximate the $\partial^2\tau/\partial x^2$ term in an off-centred fashion that contains enough grid points still to be fourth-order. Such a method, developed by Noye and Hayman (1986a), is

$$\begin{aligned} 12\tau_j^{n+1} &= 2s(5+6s)\tau_0^n + 3(4-5s-18s^2)\tau_1^n + 4s(24s-1)\tau_2^n \\ &+ 14s(1-6s)\tau_3^n + 6s(6s-1)\tau_4^n + s(1-6s)\tau_5^n \end{aligned} \qquad (2.6.5)$$

at the $x = 0$ boundary with the mirror-image equation

$$\begin{aligned} 12\tau_J^{n+1} &= s(1-6s)\tau_{J-5}^n + 6s(6s-1)\tau_{J-4}^n + 14s(1-6s)\tau_{J-3}^n \\ &+ 4s(24s-1)\tau_{J-2}^n + 3(4-5s-18s^2)\tau_{J-1}^n + 2s(5+6s)\tau_J^n \end{aligned} \qquad (2.6.6)$$

being used at the $x = 1$ boundary. To verify the accuracy of these equations, their modified equivalent equations are examined. The two equations are found to have the same MEPDE, and the leading error term contains the factor

$$\Gamma_6(s) = 2(30s^2 + 75s - 13), \tag{2.6.7}$$

which verifies the fourth-order accuracy of the equations.

The von Neumann stability range, however, is not large enough to allow the use of these equations directly. The range, computed numerically, is

$$0 < s \leq 0.29\ldots\ , \tag{2.6.8}$$

which is a significantly smaller range than that given in (2.6.4).

The only way around this problem is to use the formulae (2.6.5) and (2.6.6) with k smaller time steps at the boundary, which then necessitates calculating values at the time levels $(n + i/k)\Delta t$ for $i = 1, 2, 3, \ldots, k-1$, $j = 0, 1, 2, 3, 4, 5$ and also for $j = J-5, J-4, J-3, J-2, J-1, J$. Alternatively, a single formula can be derived, which is the equivalent of k steps using, say, (2.6.5). This formula however is extremely complicated, with coefficients that are polynomials of large degree in s, so a simpler method was sought.

Crandall (1955) developed an optimal (3,3) implicit method, based on the computational stencil shown in Figure 2.12. This equation, which is the most accurate possible with this computational stencil, is

$$\begin{aligned} &\{1-6s\}(\tau_{j-1}^{n+1} + \tau_{j+1}^{n+1}) + 2\{5+6s\}\tau_j^{n+1} \\ = \ &\{1+6s\}(\tau_{j-1}^{n} + \tau_{j+1}^{n}) + 2\{5-6s\}\tau_j^{n}, \end{aligned} \tag{2.6.9}$$

which has a modified equivalent equation that can be written in the general form (2.2.4), with a leading error term which involves the factor

$$\Gamma_6(s) = \frac{3}{2}(1 - 20s^2), \tag{2.6.10}$$

which shows that the method is fourth-order accurate. When used on its own to solve the diffusion equation (2.1.1), this method, being implicit, requires the solution of a set

of linear algebraic equations at each time level. Even though the relatively efficient Thomas (1949) algorithm can be used, this is a time-consuming process compared to the use of explicit methods, and is the main disadvantage of implicit methods in general, particularly for solving multi-dimensional problems.

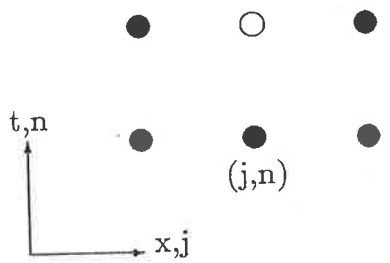


Figure 2.12: *Computational stencil for the boundary scheme based on Crandall's (3,3) method*

For the purposes of overcoming the boundary problems inherent in the (1,5) explicit scheme, however, Crandall's equation can be used in an explicit fashion to find the values τ_1^{n+1} and τ_{J-1}^{n+1}. This is possible because the values τ_0^{n+1} and τ_J^{n+1} at the boundaries are known from the Dirichlet boundary condition (the case of the Neumann condition is discussed below in Section 3), the values τ_2^{n+1} and τ_{J-2}^{n+1} are known from the main formula and all the required values at time level n are also known. This leaves only one value unknown in Crandall's formula at either boundary, so the equation can be re-arranged to give this value explicitly, namely

$$\begin{aligned} 2\{5+6s\}\tau_j^{n+1} &= \{6s-1\}(\tau_{j-1}^{n+1}+\tau_{j+1}^{n+1}) \\ &+ \{1+6s\}(\tau_{j-1}^{n}+\tau_{j+1}^{n})+2\{5-6s\}\tau_j^{n} \end{aligned} \qquad (2.6.11)$$

where $j = 1$ is used at the $x = 0$ boundary, and $j = J-1$ is used at the $x = 1$ boundary.

One of the main reasons for trying this method is that Crandall's implicit equation is both fourth-order accurate for all $s > 0$ and also unconditionally von Neumann stable, so it is hoped that this stability will also be apparent in the rearranged form. When

equation (2.6.11) is checked numerically for stability, remembering that two of the values come straight from the boundary conditions and hence play no part in error accumulation, and the other known value at time level $(n+1)$ is calculated from values at time level n via the main formula, the von Neumann stability region is found to be

$$0 < s \leq 0.95. \tag{2.6.12}$$

This is much larger than the stability region for the interior of the solution domain, so use of this boundary scheme imposes no extra restrictions on the method. This being the case, this technique of overcoming the boundary problems associated with the (1,5) equation is to be preferred over the use of the off-centred approximations and multiple steps at the boundaries described above.

The actual results, obtained for the Gaussian peak defined by (2.2.18) and shown in Figures 2.13 and 2.14, are very close to the expected ones; the graphs are straight lines with slopes very close to the predicted values of 4 and 4/3 for the error and CPU graphs respectively. The errors are much smaller than those of any other fourth-order method that has been considered so far. Even compared to the FTCS equation in the optimal case when $s = 1/6$, which gives results of similar accuracy to the (1,5) method, the (1,5) method is superior in terms of computer time used. This is because the (1,5) method with $s = 1/3$ gives the same accuracy as the FTCS method with $s = 1/6$, but can use twice the time step for a given spatial grid, due to the larger value of s. This means that the same time level can be reached in only half as many time steps, which significantly reduces the computational time required. It is interesting to note, however, that of all the values shown, the value $s = 0.25$ uses the smallest amount of CPU time to achieve a given accuracy. This coincides with the minimal value of the error coefficient $\Gamma_6(s)$, given in equation (2.3.9), although experience with other equations of different accuracies shows that in general terms the two values are merely related and not necessarily equal. Compared with this is the fact that the smallest amount of CPU time required to generate a solution (independent of the accuracy) occurs for $s = 2/3$, which is the expected result since larger values of s allow larger time steps and hence use less computer time.

Compared to the fourth-order (1,3,3) equation (2.5.16) the (1,5) equation is superior

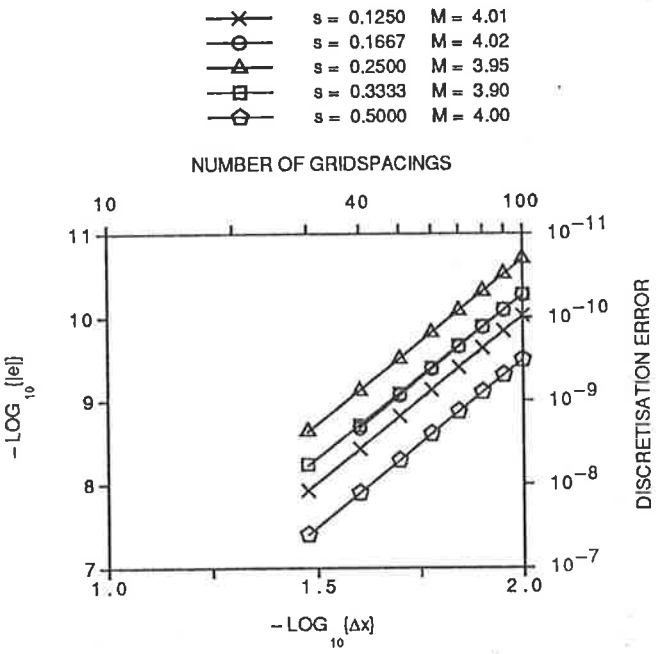


Figure 2.13: *Error vs grid spacing graph for the fourth-order (1,5) method (2.3.8)*

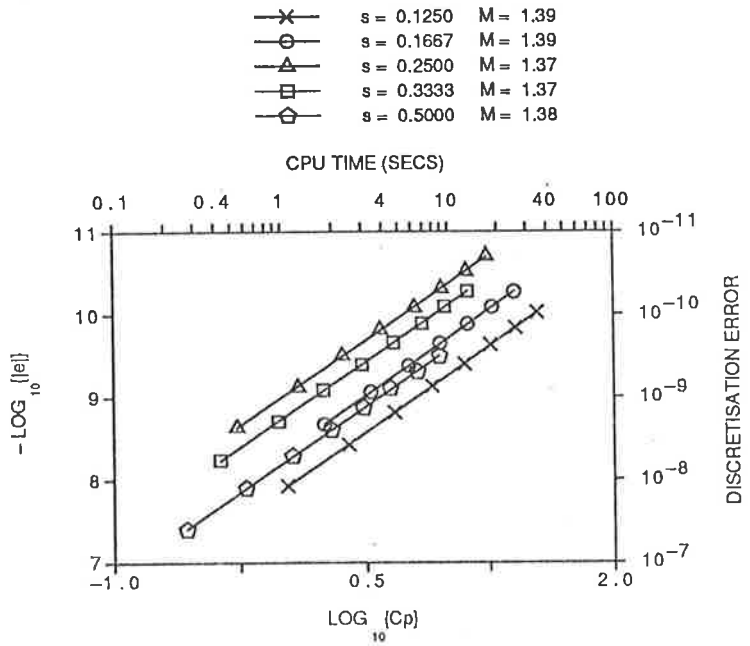


Figure 2.14: *Error vs CPU time graph for the fourth-order (1,5) method (2.3.8)*

in terms of both accuracy and computer time used. The use of Crandall's formula to overcome problems next to the boundary means that there is no need to use any grid points beyond the physical solution domain. Also, the von Neumann stability range of the (1,5) equation is greater than that for the (1,3,3), which allows larger time steps to be taken for the same grid spacing, so reducing the amount of CPU time required in cases where a specified spatial resolution of the solution is required.

In summary, the preferred method of solution of the diffusion equation (2.1.1), amongst the explicit methods discussed above, is either the sixth-order (1,3,3) equation (2.5.9), if its stability restriction is not a problem or the fourth-order (1,5) equation (2.3.8), using the rearranged Crandall formula (2.6.11) at the boundaries. These have been shown to give the most accurate answers, using only modest amount of CPU time and are very straight-forward to implement.

2.7 The (1,5,1) Method

Although the optimal (1,5) equation presented above provides high accuracy and a good usable region, it is possible that even better methods may be possible using a larger computational stencil. An attempt has been made to produce an even more accurate method by extending the computational stencil to include the $(j, n-1)$ grid point. This leads to a finite-difference equation based on the (1,5,1) stencil shown in Figure 2.15.

The (1,5,1) stencil allows the use of two weights, one for the spatial derivative in exactly the same way as was done for the (1,5) equation, and another for the time derivative. Thus the weighting used is

$$\begin{aligned} \left.\frac{\partial \tau}{\partial t}\right|_j^n &\approx \theta \times [\text{FT at } (j,n)] + (1-\theta) \times [\text{CT at } (j,n)], \\ \left.\frac{\partial^2 \tau}{\partial x^2}\right|_j^n &\approx \varphi \times [\text{CS3 at } (j,n)] + (1-\varphi) \times [\text{CS5 at } (j,n)]. \end{aligned} \tag{2.7.1}$$

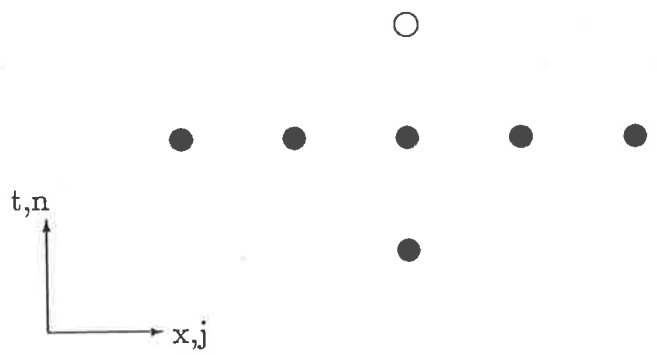


Figure 2.15: *The (1,5,1) computational stencil*

This weighting produces the finite-difference equation

$$\begin{aligned} 6\{\theta+1\}\tau_j^{n+1} &= s\{\varphi-1\}(\tau_{j-2}^n+\tau_{j+2}^n)+4s\{4-\varphi\}(\tau_{j-1}^n+\tau_{j+1}^n) \\ &+ 6\{\varphi s+2\theta-5s\}\tau_j^n+6\{1-\theta\}\tau_j^{n-1}\ , \quad \theta\neq -1, \end{aligned} \tag{2.7.2}$$

which has a corresponding modified equivalent equation whose leading error terms contain the factors

$$\begin{aligned} \Gamma_4(s) &= 6\theta s-\varphi, \\ \Gamma_6(s) &= 4-5\varphi+30\theta\varphi s+60s^2(1-3\theta^2). \end{aligned} \tag{2.7.3}$$

From this, it is apparent that the equation(2.7.2) is consistent with the one-dimensional diffusion equation (2.1.1), and is, in general, second-order accurate. However, the choice of weights

$$\begin{aligned} \theta &= \frac{2+30s^2}{15s}, \\ \varphi &= \frac{4+60s^2}{5} \end{aligned} \tag{2.7.4}$$

forces the terms $\Gamma_4(s)=\Gamma_6(s)=0$, so the equation resulting from the substitution of these values should be sixth-order accurate. Note also that for all real s, the substitution (2.7.4) satisfies the restriction $\theta\neq -1$ on equation (2.7.2). In fact, the

resulting equation is

$$\begin{aligned} -2\{30s^2+15s+2\}\tau_j^{n+1} &= s^2\{1-60s^2\}(\tau_{j-2}^n+\tau_{j+2}^n) \\ &- 16s^2\{4-15s^2\}(\tau_{j-1}^n+\tau_{j+1}^n) \\ &- 2\{180s^4-3s^2+4\}\tau_j^n \\ &+ 2\{30s^2-15s+2\}\tau_j^{n-1}. \end{aligned} \tag{2.7.5}$$

It can be shown that this equation is in fact sixth-order, as expected, and that its modified equivalent equation contains the factor

$$\Gamma_8(s) = \frac{8400s^4-700s^2+16}{5} \tag{2.7.6}$$

in its leading error term. The von Neumann stability range of the equation (2.7.5) is given by

$$s \leq 0.51\ldots\ , \tag{2.7.7}$$

which, while it is not as large a range as that for the fourth-order (1,5) equation, is still large enough to be useful in practice, especially since there should be a significant gain in accuracy due to the increased order of accuracy of the finite-difference equation.

One complication with practical implementation of this method is the problem of determining the approximation at the grid point next to a boundary. This was solved for the (1,5) equation by using Crandall's fourth-order (3,3) equation, but this may detract from the sixth-order accuracy of the (1,5,1) equation. Other possible solutions to the problem, such as sixth-order methods with a more compact or off-centred stencil or interpolation of the correct accuracy, severely limit the von Neumann stability range of the whole implementation, which may detract from the effectiveness of the method. For this reason, Crandall's equation will be used initially, in the same way as it was for the (1,5) equation.

Another problem, associated with the fact that this equation uses values from three time levels, is the need to use another technique to compute the first time step. Since explicit two-level sixth-order methods tend to be very unstable, their use would unnecessarily complicate the starting procedure, if there is some other less complicated starting method, which gives acceptable results. In fact, there are several such starting methods. The best methods are either the FTCS equation with $s = 1/6$, which is

simple to implement but which would require several time steps to reach the required time equivalent to one time step of the (1,5,1) equation with a larger value of s, or the fourth-order (1,5) equation, which requires extra work at the boundaries, but is stable over the entire stability range of the (1,5,1) equation and so needs to be used for only one time step to find the required starting values. It is found in practice that the actual (fourth-order) starting scheme which is used makes little difference to the final results obtained.

The results from implementing this scheme in practice are shown in Figures 2.16 and 2.17. The most obvious thing about the graphs is that the slopes of the lines are all as would be expected for a sixth-order method. Thus the use of the Crandall equation at the boundary and the fourth-order starting scheme have not detracted from the accuracy of the method in this case. Note however that in a different application this may not be the case, and then some other technique such as those mentioned above may be required at the boundary. Whether the increase in CPU time usage arising from the much smaller stability range is justified by the high accuracy method would need to be determined for the particular application.

Notice that the minimum CPU time required to generate a solution of a particular accuracy occurs for $s = 0.2$, which is very close to a local minimum of the coefficient $\Gamma_8(s)$ given in equation (2.7.6). Another feature of Figure 2.17 is that the CPU usage is not much greater than that for the two fourth-order methods discussed earlier (see Figure 2.14), so this method appears to be a better overall method than the fourth-order ones.

If the sixth-order (1,3,3) equation (2.5.9) is used to fill in the value next to the boundary at each time step, then the entire method, after the starting procedure, is sixth-order. This does introduce, however, a much more restricted von Neumann stability range, since it was shown in Section 2.5 above that equation (2.5.9) is only stable and consistent for

$$s < 1/6. \tag{2.7.8}$$

The results of doing this, shown in Figures 2.18 and 2.19, are very similar to those from

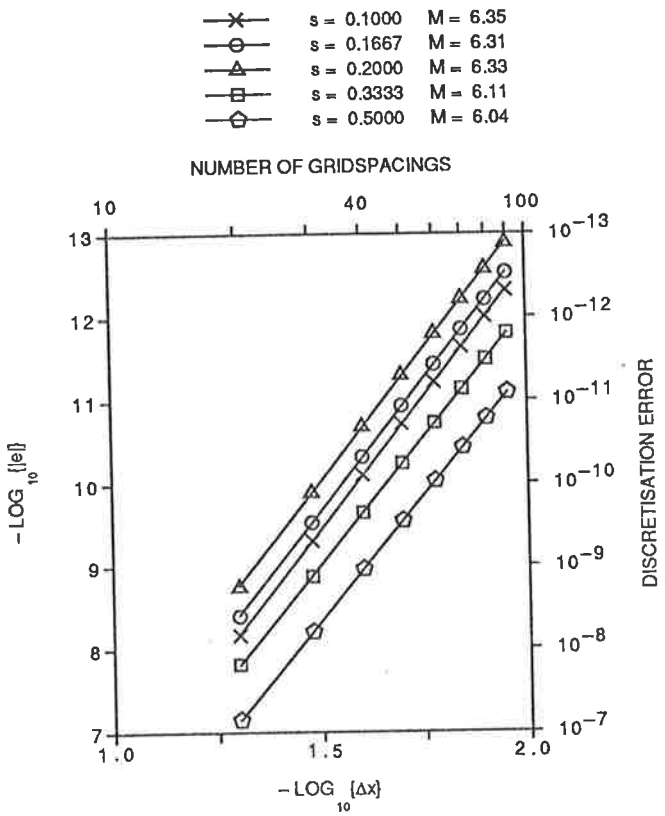


Figure 2.16: *Error vs grid spacing graph for the sixth-order (1,5,1) method (2.7.5), using Crandall next to the boundaries*

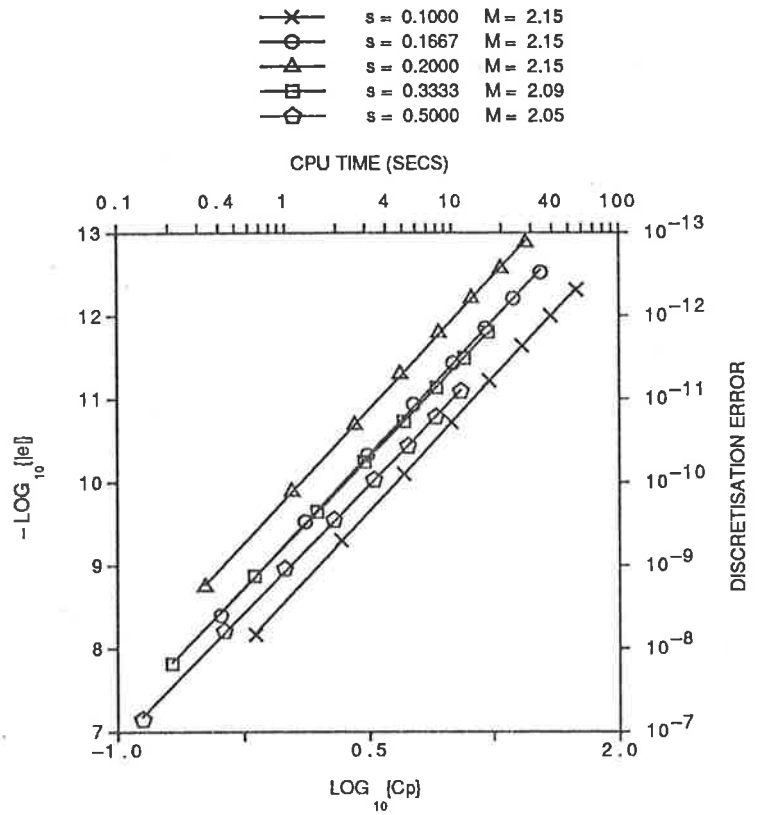


Figure 2.17: *Error vs CPU time graph for the sixth-order (1,5,1) method (2.7.5), using Crandall next to the boundaries*

using Crandall. The difference is that this technique is slightly more accurate, and also uses somewhat less CPU time to gain a specified accuracy. Thus it is seen here again that a restrictive stability range does not necessarily increase the CPU requirements of the method, unless there is some minimum spatial resolution required.

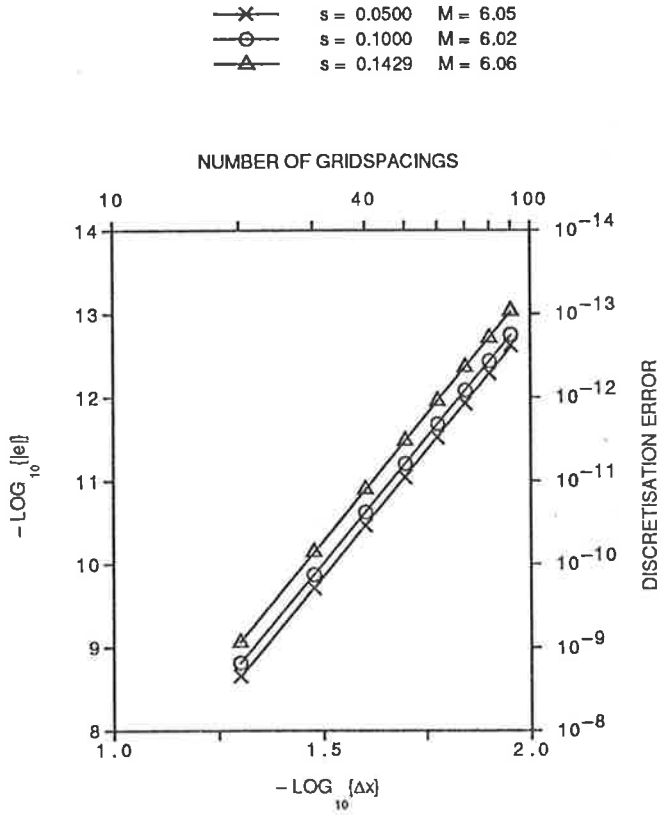


Figure 2.18: *Error vs grid spacing graph for the sixth-order (1,5,1) method (2.7.5), using sixth-order (1,3,3) next to the boundaries*

Attempts at producing even higher-order methods, particularly one based on a (1,5,3) stencil, produce finite-differences equations which tend to have severely limited von Neumann stability ranges and also have modified equivalent equations which involved coefficients which were too large to be dealt with on the VAX system being used in the time available. Such equations, while they may be of use after further investigation, are thus not considered here.

Overall, of the explicit methods examined, the preferred method is to use either the sixth-order (1,3,3) equation or the (1,5,1) equation with the problems near the bound-

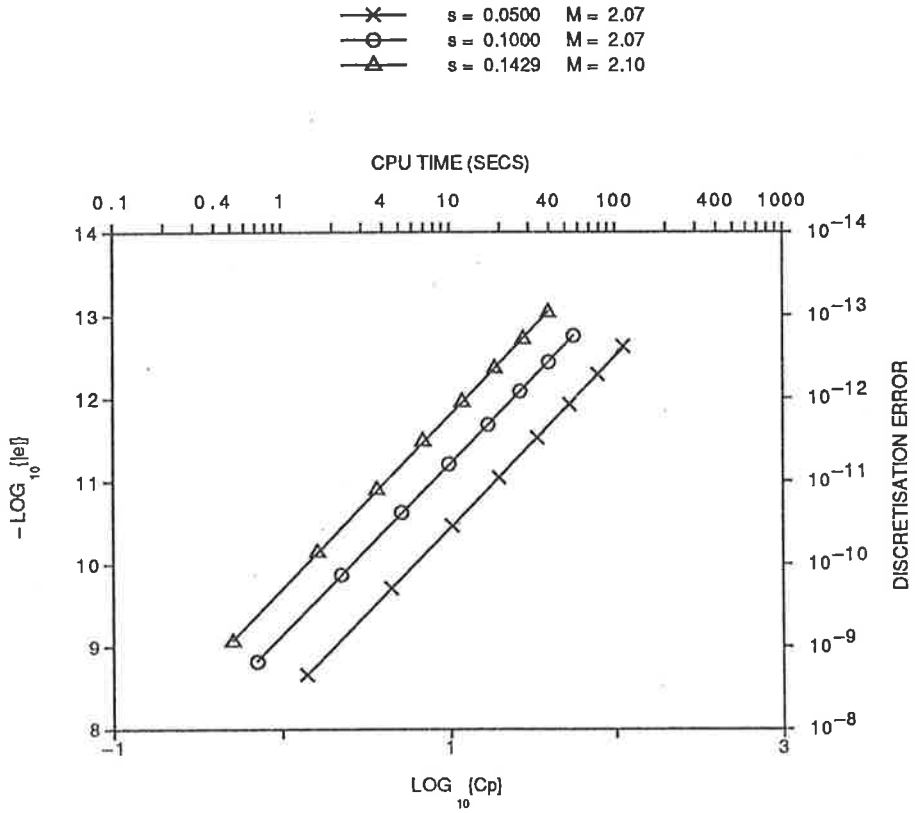


Figure 2.19: *Error vs CPU time graph for the sixth-order (1,5,1) method (2.7.5), using sixth-order (1,3,3) next to the boundaries*

aries overcome using either the sixth-order (1,3,3) equation or Crandall's equation. The fourth-order (1,5) and (1,3,3) equations, however, also produce very accurate results for the test problem, and may be useful tools depending on circumstances.

2.8 Implicit Methods

The methods that have been examined so far have all been *explicit* in nature. It is possible to employ *implicit* methods, where the finite-difference equation relates several values at the new time level to values at previous time levels.

Implicit methods have enjoyed considerable popularity, since in general they tend to have much larger stability ranges than explicit methods. The major drawback with implicit methods however is that they require the solution of a set of linear algebraic equations for each time step, which requires both extra storage space and extra computation time on a computer. Furthermore, the solution of this set of equations can impose extra restrictions on the use of a method, since the algorithms used to solve such equations are also subject to numerical instabilities. It is also worth noting again that an equation with a large stability range does not necessarily use less CPU time to generate a solution to a given accuracy.

Another undesirable feature of implicit methods is that they cannot make full use of array processors. For an explicit method, any value at the new time level can be computed directly from values at previous time levels, so the full power of array processors can be used. This is not the case for implicit methods, where values at the new time level are expressed in terms of other values at the same time level, and require the solution of a set of linear equations to find the individual values.

For implicit methods that use only three grid point at time level $(n+1)$, the most efficient way to solve the set of equations is the Thomas algorithm (Thomas, 1949). This is a specialised case of Gauss elimination with back substitution for the tri-diagonal systems of equations which such methods generate. A sufficient condition that this algorithm gives correct results is that the coefficient matrix of the system of

equations be diagonally dominant, which means that for every row of the coefficient matrix $A = [a_{ij}]$

$$|a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{J-1} |a_{ij}|, \tag{2.8.1}$$

with strict inequality holding for at least one row. While there does exist a stronger definition of diagonal dominance, the condition (2.8.1) is adequate here due to the form of the implicit finite-difference equations. This extra condition will be referred to as *solvability*, and when combined with the von Neumann stability condition will define the total region for which the method is actually *usable*.

One of the earliest implicit methods, and one which is still widely known and utilised, is that due to Crank and Nicolson (1947). It is derived by splitting the spatial differencing evenly between time levels n and $(n+1)$, which leads to the equation

$$\begin{aligned} - \quad & \{s/2\}(\tau_{j-1}^{n+1} + \tau_{j+1}^{n+1}) + \{1+s\}\tau_j^{n+1} \\ = \quad & \{s/2\}(\tau_{j-1}^{n} + \tau_{j+1}^{n}) + \{1-s\}\tau_j^{n}. \end{aligned} \tag{2.8.2}$$

This equation has the advantage that it is both von Neumann stable and solvable for

$$s > 0, \tag{2.8.3}$$

(Noye, 1984) which means that in theory the time steps can be made arbitrarily large. The modified equation corresponding to equation (2.8.2) has leading errors involving the terms

$$\begin{aligned} \Gamma_4(s) &= -1 \\ \Gamma_6(s) &= -(1+30s^2). \end{aligned} \tag{2.8.4}$$

From this, it is seen that the Crank-Nicolson equation is only second-order accurate, but it does have the advantage that the size of the leading error term is independent of s. This means that large time steps can be used, although the results are constrained by the second-order accuracy.

Numerical results for this method, shown in Figures 2.20 and 2.21, verify the theory above. The error graphs are almost identical, independent of s, except for a slight

deviation for $s = 8$ with $J = 30$. This is caused by the magnitude of the coefficient of the fourth-order error term, $\Gamma_6(s)$, which is given above, becoming large and thus increasing the fourth-order error to the point where it makes a significant contribution to the overall error. Once this happens, the errors increase in proportion to s^2, and the accuracy of the method is destroyed. Also, the errors themselves are much worse than those obtained by the fourth-order explicit methods, and the savings in terms of CPU time used are small at best. Overall, this method is of little practical use given the accuracy and efficiency of the explicit methods developed earlier.

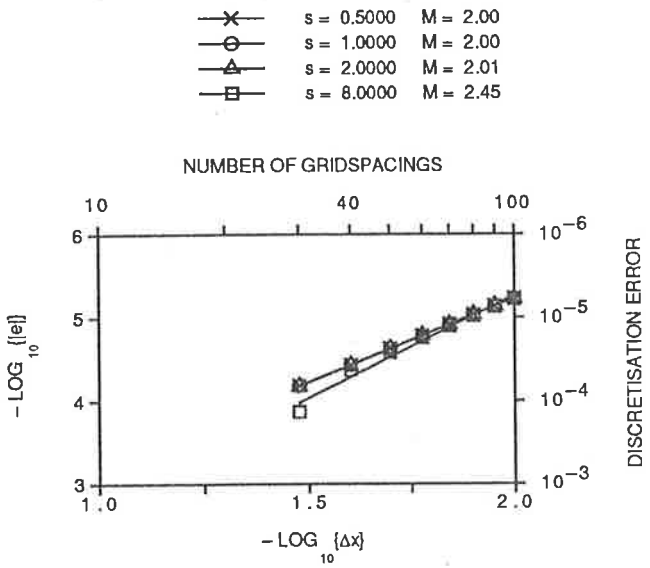


Figure 2.20: *Error vs grid spacing graph for the (3,3) Crank-Nicolson method (2.8.2)*

In order to obtain more accuracy while still using the same computational stencil, the differencing of the spatial derivative can be split between time levels n and $(n+1)$ in a more general fashion than used by Crank and Nicolson. This is done using one weight, θ, by expressing

$$\left.\frac{\partial^2 \tau}{\partial x^2}\right|_j^n \approx \theta \times [\text{CS at } (j,n)] + (1-\theta) \times [\text{CS at } (j,n+1)]. \tag{2.8.5}$$

The resulting finite-difference equation is

$$\begin{aligned} & s\{\theta - 1\}(\tau_{j-1}^{n+1} + \tau_{j+1}^{n+1}) + \{1 + 2s(1-\theta)\}\tau_j^{n+1} \\ = \;& \theta s(\tau_{j-1}^n + \tau_{j+1}^n) + \{1 - 2\theta s\}\tau_j^n, \end{aligned} \tag{2.8.6}$$

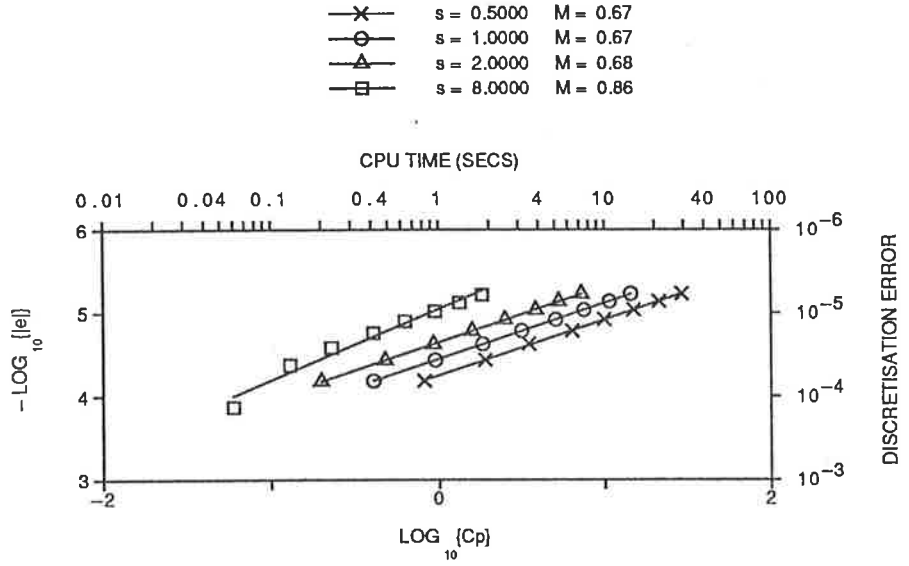


Figure 2.21: *Error vs CPU time graph for the (3,3) Crank-Nicolson method (2.8.2)*

which reduces to the Crank-Nicolson equation (2.8.2) in the particular case of $\theta = 1/2$. The modified equivalent equation corresponding to (2.8.6) has leading errors involving

$$\Gamma_4(s) = 12\theta s - 6s - 1, \tag{2.8.7}$$

from which it can be seen that the method can be made fourth-order accurate if θ is chosen as

$$\theta = \frac{1}{2} + \frac{1}{12s}. \tag{2.8.8}$$

The result of this substitution is Crandall's fourth-order equation

$$\begin{aligned} &\{1-6s\}(\tau_{j-1}^{n+1} + \tau_{j+1}^{n+1}) + 2\{5+6s\}\tau_j^{n+1} \\ = \;&\{1+6s\}(\tau_{j-1}^{n} + \tau_{j+1}^{n}) + 2\{5-6s\}\tau_j^{n}, \end{aligned} \tag{2.8.9}$$

which was used earlier to solve the boundary problems of the fourth-order (1,5) equation. This method is both von Neumann stable and solvable for all values of $s > 0$ and it is fourth-order accurate for all s except for $s = 1/\sqrt{20}$, when it is sixth-order. This optimal value, which can be derived from the modified equivalent equation coefficient (2.6.10), can be used in practice to give good results, but the small value of s requires small time steps and so increases the amount of CPU time required. Thus practical use of this optimal value depends on the acceptability of this restriction.

Indiscriminate use of the extended usable region of equation (2.8.9) is again not possible, since the fourth-order error coefficient contains a term involving s^2. Thus as s is increased to use larger time steps, the size of the dominant error term increases quadratically, which eventually degrades the solution to the point where the results are useless.

Again, numerical experiments verify the above theory. By comparing Figures 2.22 and 2.23 with Figures 2.20 and 2.21, it is obvious that the answers are much more accurate than those produced by the Crank-Nicolson method, although for $s \geq 1$ they are not as good as the results for the (1,5) explicit method in its stable range. This means that any reduction in CPU time due to using large time steps is obtained at a direct cost to the accuracy of the final solution. It is apparent that the increased CPU time involved in solving the set of linear algebraic equations is minimising the saving due to the larger time steps.

A comparison of efficiency of this method against the fourth-order (1,5) equation is also useful. To get accuracy of 10^{-8} using the fourth-order (1,5) equation, Figure 2.14 indicates that less than one second of CPU time is required. To get the same accuracy from Crandall's equation requires at least three or four seconds of CPU time, from the results of Figure 2.23. This comparison too indicates that the implicit methods is not as efficient for practical use as the explicit methods discussed previously.

Higher order implicit methods (sixth and even eighth order accurate) are possible using such computational stencils as a (3,3,3) or a (5,5). However, methods that use spatially wide stencils, such as a (5,5) equation, have extra problems near the boundaries, and as yet no way has been found to deal with these that does not detract from either the accuracy of the final solution or the stability of the method or both. As well as this, these methods however tend to have very small usable ranges such as

$$s \leq 1/6 \tag{2.8.10}$$

or even less. This means that extremely small time steps must be used, which dramatically increases the amount of computer time needed to find a solution. Such methods, being implicit in nature, also require, as previously mentioned, the solution of a set

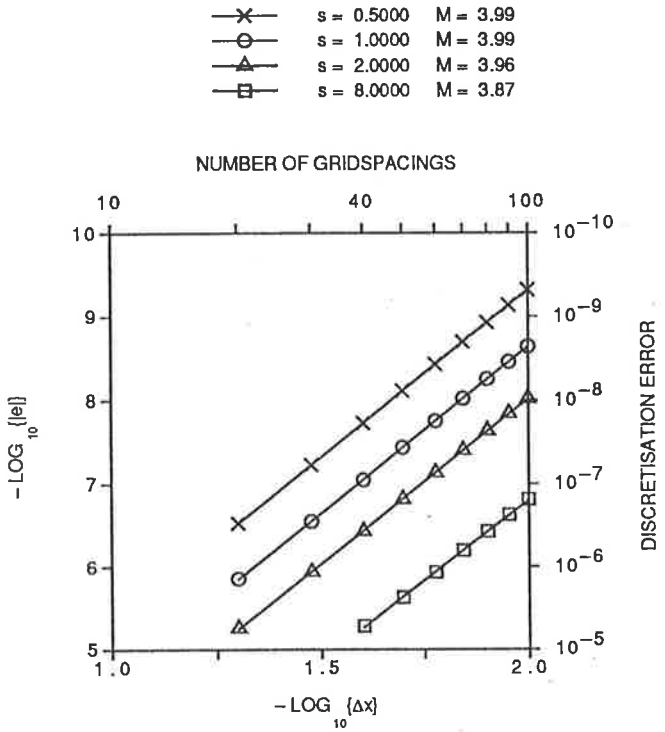


Figure 2.22: *Error vs grid spacing graph for Crandall's (3,3) method (2.6.9)*

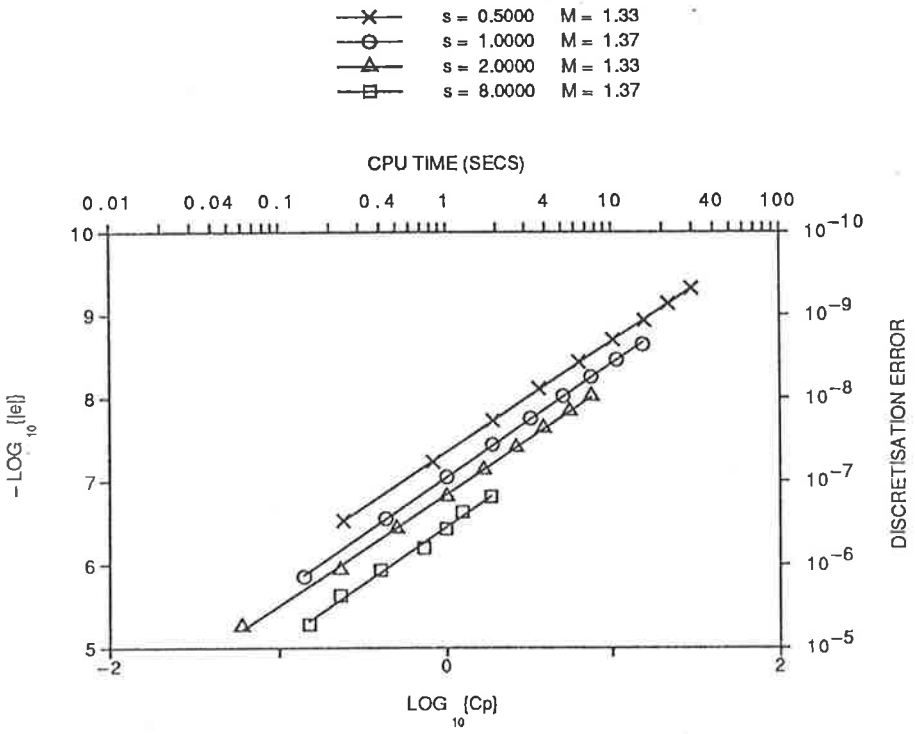


Figure 2.23: *Error vs CPU time graph for Crandall's (3,3) method (2.6.9)*

of equations for each time step taken, which further adds to the computer time used. In view of these as yet unresolved problems, such methods are not considered in this work.

2.9 Summary

From the above, it is apparent that Crandall's method, while being by far the best of the practical implicit methods, is not in general an improvement over the explicit methods developed earlier, in terms of either accuracy or CPU time used. Given this, the explicit methods are still preferred since they do not require the solution of sets of equations, and are thus easier to implement. Also, since our ultimate goal is to produce methods for solving the two-dimensional diffusion equation, the solution of a set of equations at each time step is undesirable, because the extra CPU time required to do this increases enormously in two dimensions due to the more complex structure of the coefficient matrix and the much larger number of equations to be solved. The most obvious advantage of the implicit methods, that of having unlimited usable ranges, has been shown to be counteracted by a large increase in the size of the errors if the value of s is increased. For small values of s, the extra CPU time required to solve the set of linear algebraic equations at each time step, as well as the slightly larger errors for the implicit methods, make the explicit methods better to use.

Explicit methods appear to be the best methods to use to solve a practical problem. Exactly which method is the "best" is dependent on circumstances, there being two different choices. If a small von Neumann stability range is acceptable, which is the case where the use of coarse grids is possible and/or accuracy is the prime concern, then one the sixth-order equations is clearly the best choice. If, however, CPU time is a prime concern, then the fourth-order (1,5) equation will be the best choice of solution technique, since it offers the largest von Neumann stability range.

The lack of a single "best" technique is a reflection of the fact that there are many different factors involved in a solution, some of which will be more important than others in each specific case. The important thing found here is that restricted stability

ranges do not necessarily count against a high-order method, since such a method can produce extremely accurate results of a relatively coarse grid. Amongst methods of the same order of accuracy, however, the one with the largest stability range will generally use less CPU time to generate solutions.

Chapter 3

The 1-D Diffusion Equation with a Neumann Boundary Condition

3.1 Introduction

The preceding work is based on the assumption of having Dirichlet boundary conditions, namely conditions of the form

$$\begin{aligned} \hat{\tau}(0,t) &= g_1(t) \\ \hat{\tau}(1,t) &= g_2(t). \end{aligned} \tag{3.1.1}$$

The other case that needs to be considered is the Neumann boundary condition, where at least one of the boundary conditions is given in the form

$$\frac{\partial \hat{\tau}}{\partial x} = c_1(t). \tag{3.1.2}$$

The following work assumes that such a boundary condition is given only at the $x = 0$ boundary, although the principles used to solve this case can be applied at the $x = 1$ boundary as well. Such a boundary condition usually arises in practice where, for example, there is an impermeable barrier or layer, which is represented mathematically as there being zero velocity across the boundary (Bear, 1972). Other cases where the (non-zero) velocity across a boundary is known are, however, possible as well.

The additional problems with this type of boundary condition arise from the fact that the value on the boundary at the new time level is unknown. This means that one of the values τ_0^{n+1} and τ_1^{n+1} must be found without requiring knowledge of the other value. This extra value to be calculated near the boundary requires different handling of the boundary problems to that used for the Dirichlet case.

In order to incorporate a boundary condition of the form (3.1.2) into a finite-difference scheme, the derivative $\partial\tau/\partial x$ must be expressed as a combination of known values of τ at nearby grid points. Care must also be taken that this approximation does not degrade the accuracy of the solution near the boundary, as such a loss of accuracy will eventually reduce the accuracy throughout the solution domain.

To determine the necessary accuracy for an approximation, consider that the approximation is accurate to order q. This in general involves one of the forms

$$\left.\frac{\partial\hat{\tau}}{\partial x}\right|_0^n = \frac{F(\hat{\tau}_0^n, \hat{\tau}_1^n, \hat{\tau}_1^n, \ldots, \hat{\tau}_q^n)}{\Delta x} + O\{(\Delta x)^q\}, \tag{3.1.3}$$

$$\left.\frac{\partial\hat{\tau}}{\partial x}\right|_0^n = \frac{G(\hat{\tau}_{-1}^n, \hat{\tau}_0^n, \hat{\tau}_1^n, \ldots, \hat{\tau}_{q-1}^n)}{\Delta x} + O\{(\Delta x)^q\}, \tag{3.1.4}$$

where the functions F and G are linear functions. Note that in (3.1.4) a value $\hat{\tau}_{-1}^n$ at the exterior grid point $(-\Delta x, n\Delta t)$ is included, and this value must also be determined if this form is used.

When (3.1.3) or (3.1.4) is rearranged to give an approximation for $\hat{\tau}_j^n$, $j = -1, 0, 1, \ldots$, the error term in that value becomes $O\{(\Delta x)^{q+1}\}$. When such an approximation is substituted into the difference equation, it is clear from the form of equation (2.1.8) that there is a decrease in accuracy of two orders, corresponding to division by either $(\Delta x)^2$ or Δt, since

$$\Delta t \propto (\Delta x)^2 \tag{3.1.5}$$

for a fixed value of s.

To verify that this last assertion about the required order of accuracy is correct, consider substituting the fourth-order approximation for τ_{j-2}^n, namely

$$\tau_{j-2}^n \approx 4\tau_{j-1}^n - 6\tau_j^n + 4\tau_{j+1}^n - \tau_{j+2}^n \tag{3.1.6}$$

into the fourth-order equation (2.3.8). If the above theory is correct, the fourth-order approximation (3.1.6) should be reduced by the substitution to second-order, as a result making the finite-difference equation second-order. In fact, the resulting finite-difference equation is

$$\tau_j^{n+1} = s\tau_{j-1}^n + (1-2s)\tau_j^n + s\tau_{j+1}^n, \tag{3.1.7}$$

which is the three-point FTCS equation (2.1.9). This has already been shown to be, in general, second-order accurate, which is the expected result. If however, the sixth-order approximation to τ_{j-2}^n is used, the resulting equation is

$$\begin{aligned} 12\tau_j^{n+1} &= 2s(6s+5)\tau_{j-1}^n + 3(4-5s-18s^2)\tau_j^n \\ &+ 4s(24s-1)\tau_{j+1}^n + 14s(1-6s)\tau_{j+2}^n \\ &+ 6s(6s-1)\tau_{j+3}^n + s(1-6s)\tau_{j+4}^n, \end{aligned} \tag{3.1.8}$$

which is fourth-order accurate for all values of s, since its modified equivalent equation, written in the form (2.2.4) has a leading error term containing the factor

$$\Gamma_6(s) = 60s^2 + 150s - 26. \tag{3.1.9}$$

Again, this is in accordance with our expectation, and illustrates that to substitute a value into a finite-difference equation of order p with no loss of accuracy, an order $p+2$ approximation to that value is required.

Returning to the derivative boundary condition problem, we have shown that to use a derivative boundary condition approximation to substitute a value into a finite-difference equation which is of order p without loss of formal accuracy, the approximation to the derivative must have a truncation error of $(\Delta x)^{p+1}$. Given this information, attention can now given to solution techniques incorporating the derivative boundary condition.

The numerical test used for this problem is almost identical to that used for the Dirichlet condition, except that the peak described by (2.2.18) has been moved from $a = 0.5$ to $a = 0.25$. This is done so that the derivative at $x = 0$ is not close to zero, since if this derivative *is* close to zero, artificially good results are obtained.

The error measurements are still taken at $x = 0.2$ after time $T = 8$, as was the case for the Dirichlet condition. Note however that the exact solution has changed to $\hat{\tau} = 0.17....$ Since this is the same order of magnitude as that for the Dirichlet condition, comparing the magnitude of the absolute errors obtained for the Neumann and Dirichlet conditions is still reasonable.

3.2 Using External Grid Points

One way to approach the problem of not knowing the actual values at the boundary, regardless which fourth-order finite-difference equation used in the interior of the solution domain, is to add a set of extra grid points outside the solution domain at $x = -\Delta x$. The fictitious value τ^n_{-1} at these grid points can be found from the fifth-order approximation

$$\left.\frac{\partial \hat{\tau}}{\partial x}\right|_0^n = \frac{-12\hat{\tau}^n_{-1} - 65\hat{\tau}^n_0 + 120\hat{\tau}^n_1 - 60\hat{\tau}^n_2 + 20\hat{\tau}^n_3 - 3\hat{\tau}^n_4}{60\Delta x} + O\{(\Delta x)^5\}, \tag{3.2.1}$$

by rearranging this and dropping the error terms of $O\{(\Delta x)^6\}$ to give the explicit formula

$$12\tau^n_{-1} = -60\Delta x c^n_1 - 65\tau^n_0 + 120\tau^n_1 - 60\tau^n_2 + 20\tau^n_3 - 3\tau^n_4, \tag{3.2.2}$$

where $c^n_1 = c_1(n\Delta t)$. Given these external values, the (1,3,3) equation (2.5.16) can be used to calculate the actual values on the boundary at the new time level.

Also, since the values at the boundary at the current time level and all previous time levels are known, the (1,3,3) equation can be used to find the value τ^{n+1}_1 without any added complication arising from the derivative boundary condition.

In theory then, this use of the (1,3,3) equation appears to be a simple way to overcome the problems associated with the derivative boundary condition, while still maintaining fourth-order accuracy. Unfortunately, the use of the extrapolation formula (3.2.2) results in a large reduction of the stability range of the method. In fact, the stability range is now only

$$s \leq 0.23\ldots, \tag{3.2.3}$$

for $\theta = 100$. This compares badly with the case of the Dirichlet boundary condition, where the stability range was more than double this value. Also, the most efficient value, in terms of using the least CPU time to achieve a given absolute error, was seen to be at about $s = 0.25$ for the Dirichlet condition, and that value is now outside the stability range. Increasing the value of θ does not significantly improve the stability region, and again detracts from the accuracy of the solution, as in the case of a Dirichlet boundary condition.

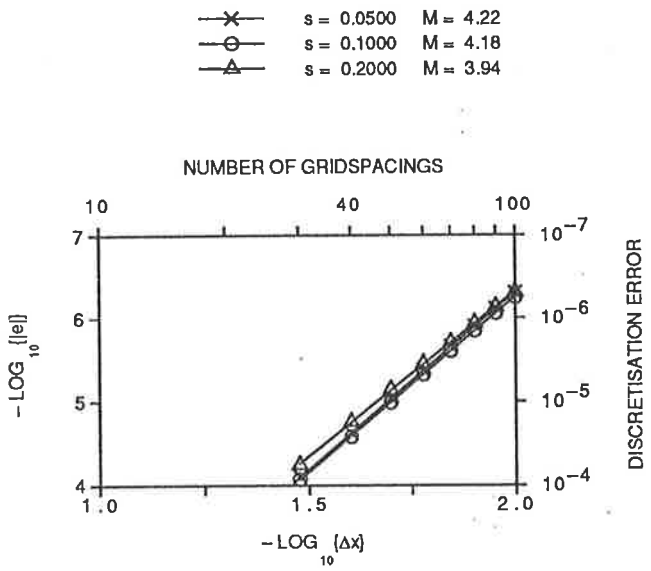


Figure 3.1: *Error vs grid spacing graph for the extrapolated (1,3,3) equation with* $\theta = 100$, *stable for* $s < 0.23^{+}$.

The accuracy obtained by this method is also not good, as shown in Figures 3.1 and 3.2, the errors being between one and two orders of magnitude larger than for the Dirichlet case studied earlier (compare with Figures 2.10 and 2.11). It is also apparent that the amount of CPU required to achieve a given error is decreasing as s increases over the values shown, so a technique that increased the stability range could well improve the efficiency of the method from this point of view as well.

Given that this technique appears to reduce the stability of finite-difference equations, the sixth-order (1,3,3) equation is not considered since a reduction of like magnitude to that for the fourth-order equation would render it impractical, even given its high

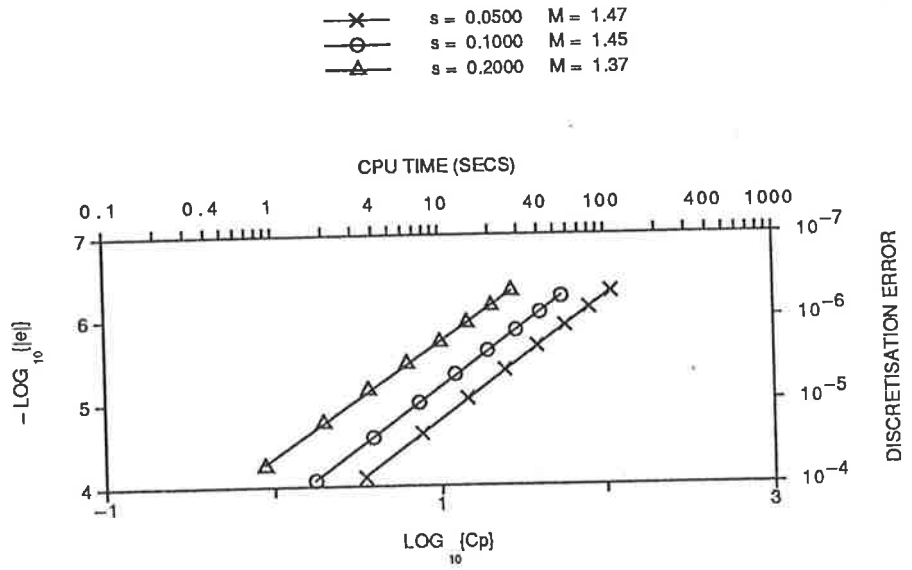


Figure 3.2: *Error vs CPU time graph for the extrapolated (1,3,3) equation with* $\theta = 100$, *stable for* $s < 0.23^{+}$.

accuracy.

The results for the spatially wide finite-difference equations, such as the fourth-order (1,5) equation (2.3.8) or the sixth-order (1,5,1) equation (2.7.5), are similar to those for the (1,3,3) equation. The unknown values τ_0^{n+1} and τ_1^{n+1} can be found from the known values τ_j^n, $j = 0(1)J$ as follows. Firstly, the value τ_{-1}^n at the exterior grid point is found from the sixth-order approximation (3.2.2). The (1,5) equation (2.3.8) can then be used to determine the value τ_1^{n+1}. In order to analyse this method, the two steps can be combined into a single finite-difference equation, namely

$$\begin{aligned} -144\tau_1^{n+1} &= 60s(6s-1)\Delta x\, c_1^n + s(678s-257)\tau_0^n - 48(24s^2-10s+3)\tau_1^n \\ &+ 36s(18s-7)\tau_2^n - 64s(6s-1)\tau_3^n + 3s(6s-1)\tau_4^n. \end{aligned} \tag{3.2.4}$$

The value τ_0^{n+1} at the boundary can then be determined from a rearrangement of Crandall's fourth-order equation (2.6.9) into the form

$$\begin{aligned} (1-6s)\tau_0^{n+1} &= (1+6s)(\tau_0^n + \tau_2^n) + 2(5-6s)\tau_1^n \\ &- 2(5+6s)\tau_1^{n+1} - (1-6s)\tau_2^{n+1}\ , \quad s \neq 1/6. \end{aligned} \tag{3.2.5}$$

When implementing this scheme in practice, it is found that as higher order approximations to the derivative at $x = 0$ are used, there is a reduction in the maximum value of s for which the resulting finite-difference scheme is von Neumann stable. In particular, use of the approximation (3.2.1) gives a scheme which is only von Neumann stable for $s \leq 0.21\ldots$. This limit, while being only an heuristic stability measure (Trapp and Ramshaw, 1976) has been found to be very close to correct in practice. It is also sufficiently small that, like the fourth-order (1,3,3) equation above, the most efficient values of s, found for the Dirichlet case, are outside the stability range.

Such a small stability range thus severely limits the usefulness of the scheme. One way to overcome this problem, mentioned above in Section 2.6, is to use k time steps each of $\Delta t/k$ at the boundary, so the effective value of s is divided by k, and the stability range is multiplied by k. Putting $k = 3$ thus provides a stability range of approximately $s \leq 0.63\ldots$, which does not detract significantly from the stability of either of the equations (2.3.8) or (2.7.5) used at other interior grid points. In order to implement this scheme, values must be calculated at two intermediate time levels, these values being used in the formulae (2.3.8) and (3.2.5) to finally obtain the desired results for τ_0^{n+1} and τ_1^{n+1}.

Alternatively, this substitution can be done algebraically to produce formulae that go directly from time level n to time level $(n+1)$ without the explicit use of intermediate time levels, which saves storage space and CPU time on a computer. The resulting formulae are very complicated, involving grid points as far as $j = 8$ from the boundary ($j = 0$), multiplied by polynomial coefficients of order 6 in s. While these could be implemented in practice, other methods have been sought in order to avoid the complicated forms of the single equation and the extra storage requirements of using multiple steps.

Overall, the use of exterior grid points has been found to be unsatisfactory, due to the severe von Neumann stability restrictions that are placed on the solution process due to having to extrapolate the values at the exterior grid points.

3.3 Using Interior Grid Points Only

Given the unacceptable stability restrictions imposed by the use of exterior grid points, alternative methods which do not require such points must be examined.

One such alternative method which can be used in conjunction with a fourth-order (1,3,3) equation is to use the fifth-order approximation to $\partial\hat{\tau}/\partial x$, namely

$$\left.\frac{\partial\hat{\tau}}{\partial x}\right|_0^n = \frac{-137\hat{\tau}_0^n + 300\hat{\tau}_1^n - 300\hat{\tau}_2^n + 200\hat{\tau}_3^n - 75\hat{\tau}_4^n + 12\hat{\tau}_5^n}{60\Delta x} + O\{(\Delta x)^5\}, \qquad (3.3.1)$$

applied at time level $(n+1)$ to give an explicit formula for the boundary value τ_0^{n+1}, namely

$$137\tau_0^{n+1} = -60\Delta x c_1^{n+1} + 300\tau_1^{n+1} - 300\tau_2^{n+1} + 200\tau_3^{n+1} - 75\tau_4^{n+1} + 12\tau_5^{n+1}. \qquad (3.3.2)$$

When this method is tested for von Neumann stability, it is found to be stable over at least the same range as the original fourth-order (1,3,3) method, so in this case, handling the Neumann boundary condition imposes no extra restrictions over the Dirichlet case, and the most efficient values can be chosen.

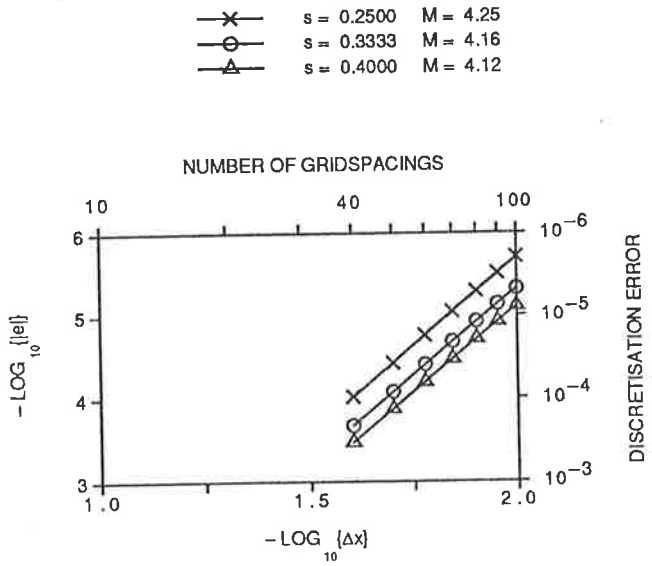


Figure 3.3: *Error vs grid spacing graph for the fourth-order (1,3,3) equation, using interior points only, with* $\theta = 100$

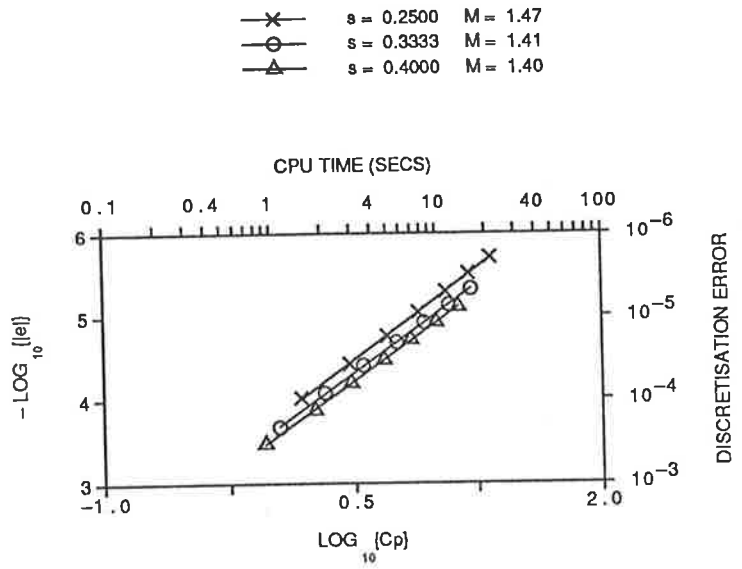


Figure 3.4: *Error vs CPU time graph for the fourth-order (1,3,3) equation, using interior points only, with* $\theta = 100$

The numerical results shown in Figure 3.3 and 3.4 show that the method is fourth-order, as expected, but the actual errors are two orders of magnitude worse than for the Dirichlet boundary condition (see Figure 2.10). This decrease in the accuracy of the generated solution is largely due to having to approximate the value on the boundary. For the Dirichlet condition there was an exact value being included at every time step, which will tend to reduce the build-up of errors. In the case of the Neumann condition, the boundary value is approximated and this benefit is not gained.

Given that the fourth-order (1,3,3) equation can be successfully implemented with no more restrictions than for the Dirichlet case, the sixth-order (1,3,3) equation should also be tried. In order to approximate the boundary condition to the correct order, a seventh-order approximation to $\partial\hat{\tau}/\partial x$ must be derived. This leads to the formula

$$\begin{aligned} 1089\tau_0^{n+1} &= -420\Delta x c_1^{n+1} + 2940\tau_1^{n+1} - 4410\tau_2^{n+1} + 4900\tau_3^{n+1} \\ &- 3675\tau_4^{n+1} + 1764\tau_5^{n+1} - 490\tau_6^{n+1} + 60\tau_7^{n+1}. \end{aligned} \tag{3.3.3}$$

which is the sixth-order analogue of equation (3.3.2) above. If this extrapolation is used with the sixth-order (1,3,3) equation, there is no reduction in the numerical stability range, since the range is already very restrictive.

The numerical results, shown in Figures 3.5 and 3.6, display a number of interesting features. Most obvious is the fact that the slopes of the lines are much greater than expected theoretically, and that the value of s has very little impact on the accuracy obtained for a given grid spacing. This may be caused by a combination of the error terms of the (1,3,3) equation, the method of handling the boundary and the test problem itself, but further investigation is required to determine the exact cause with certainty.

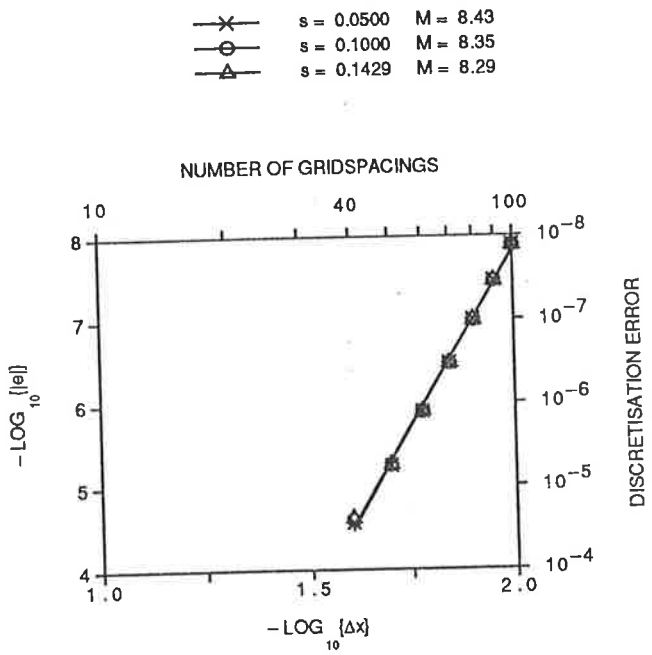


Figure 3.5: *Error vs grid spacing graph for the sixth-order (1,3,3) equation, using interior points only*

Also worth noting is that not only are the results clearly more accurate than those for the fourth-order (1,3,3) equation, which is to be expected, but the amount of CPU time required to generate a solution to a given accuracy is also better in the sixth-order case, indicating that, in this case, the restricted von Neumann stability range should not detract from the method's practicality.

If this technique is tested with lower-order boundary approximations being used, it is found that the results deteriorate to unacceptable levels, even with the use of a sixth-order approximation rather than the seventh-order one used above.

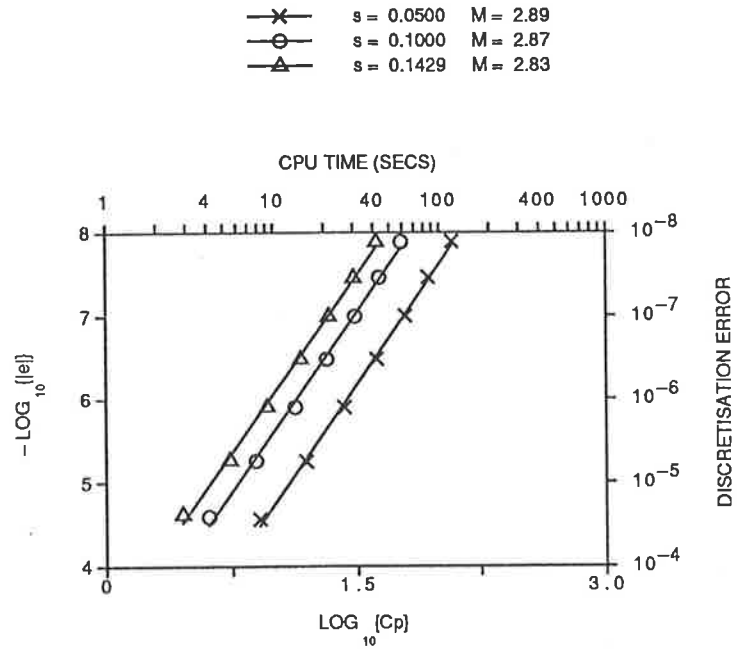


Figure 3.6: *Error vs CPU time graph for the sixth-order (1,3,3) equation, using interior points only*

The fourth-order (1,5) equation (2.3.8) presents more obvious difficulties than either form of the (1,3,3) equation discussed above, since it cannot be used next to the boundary without requiring exterior grid points. However, this problem can be overcome by using a combination of the Neumann boundary condition and appropriate rearrangement of Crandall's fourth-order equation.

In particular, Crandall's formula (2.6.9) can be rearranged to obtain a formula for the value τ_1^{n+1}, namely

$$\begin{aligned}\{1-6s\}\tau_1^{n+1} &= \{1+6s\}(\tau_1^n+\tau_3^n)+2\{5-6s\}\tau_2^n \\ &- 2\{5+6s\}\tau_2^{n+1}-\{1-6s\}\tau_3^{n+1}\ , \quad s\neq 1/6. \end{aligned} \tag{3.3.4}$$

In the case where $s = 1/6$, Crandall's equation reduces to the fourth-order special case of the (1,3) FTCS equation, and this equation can be used to find τ_1^{n+1} without loss of accuracy. The fifth-order approximation to $\partial\tau/\partial x$ (3.3.1) is then applied at time level $(n+1)$ to give the explicit formula for the boundary value (3.3.2) for τ_0^{n+1}. This boundary approximation has been applied over the entire range of stability of

the interior method, namely $0 < s \leq 2/3$, and has shown no signs of instability. In fact, numerically examining the von Neumann stability of the equation involving the boundary point that is used to step to the next time level, shows that it is stable up to at least $s = 1$, so this boundary technique imposes no extra stability restrictions.

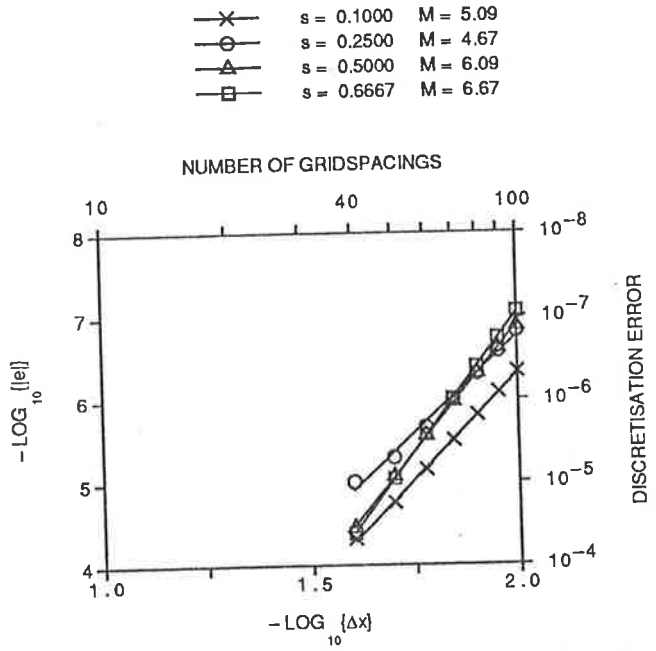


Figure 3.7: *Error vs grid spacing graph for the (1,5) equation using only interior grid points*

Note that this method, while still based on the idea of using both the derivative boundary condition and a rearrangement of Crandall's formula, does not use an exterior grid point, nor is its use dependent on the use of a particular finite-difference equation in the interior of the region. Any finite-difference equation could be used in the interior of the region, as long the von Neumann stability of the boundary technique is not exceeded.

The numerical results for this technique, shown in Figures 3.7 and 3.8, are more accurate than those results obtained by the fourth-order (1,3,3) equation, but not as accurate as the sixth-order (1,3,3) equation. Of interest, however, is that the CPU usage of the (1,5) equation to obtain a solution of a given accuracy is only about half that for the sixth-order (1,3,3) equation. Thus if the accuracy of the (1,5) equation is

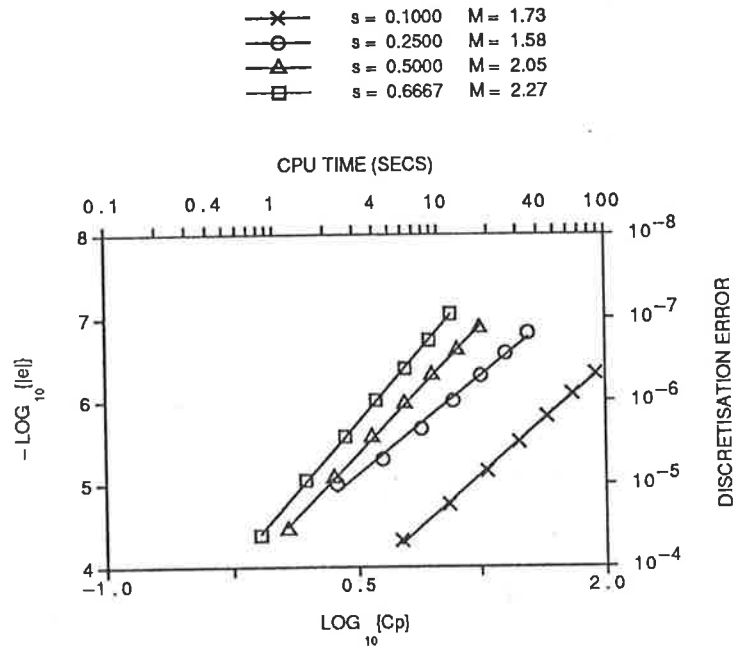


Figure 3.8: *Error vs CPU time graph for the (1,5) equation using only interior grid points*

acceptable for the problem being solved, this appears to be a better technique than the earlier (1,3,3) equations.

The effect of using less accurate approximations to the derivative at the boundary can be examined here. The fourth-order approximation

$$\left.\frac{\partial\hat{\tau}}{\partial x}\right|_0^n = \frac{-25\hat{\tau}_0^n + 48\hat{\tau}_1^n - 36\hat{\tau}_2^n + 16\hat{\tau}_3^n - 3\hat{\tau}_4^n}{12\Delta x} + O\{(\Delta x)^4\}, \tag{3.3.5}$$

can be applied at time level $(n+1)$ and rearranged to give the explicit form

$$25\tau_0^{n+1} = -12\Delta x c_1^{n+1} + 48\tau_1^{n+1} - 36\tau_2^{n+1} + 16\tau_3^{n+1} - 3\tau_4^{n+1}, \tag{3.3.6}$$

which can be used in place of (3.3.2). In this case the results, shown in Figures 3.9 and 3.10, are less accurate than for the fifth-order derivative approximation, although they are still good, as indicated by the fact that errors plotted against grid spacing on a logarithmic scale still have slope greater than the predicted value of four. In this case, however, the CPU efficiency advantage over the sixth-order (1,3,3) method has been lost, so the drop in accuracy of the boundary approximation has somewhat

damaged the practicality of the method. Note that this is in contrast to the sixth-order (1,3,3) equation, where it was found that a drop of even one order of accuracy in the boundary approximation degraded the solution unacceptably.

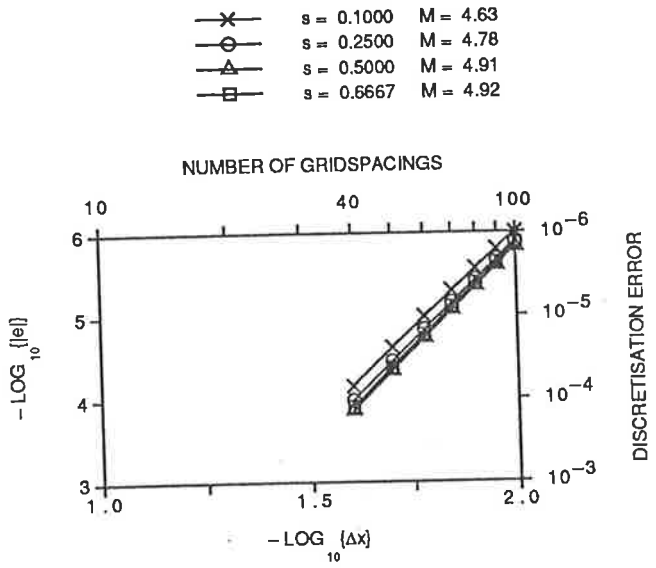


Figure 3.9: *Error vs grid spacing graph for the (1,5) equation using an $O\{4\}$ boundary approximation*

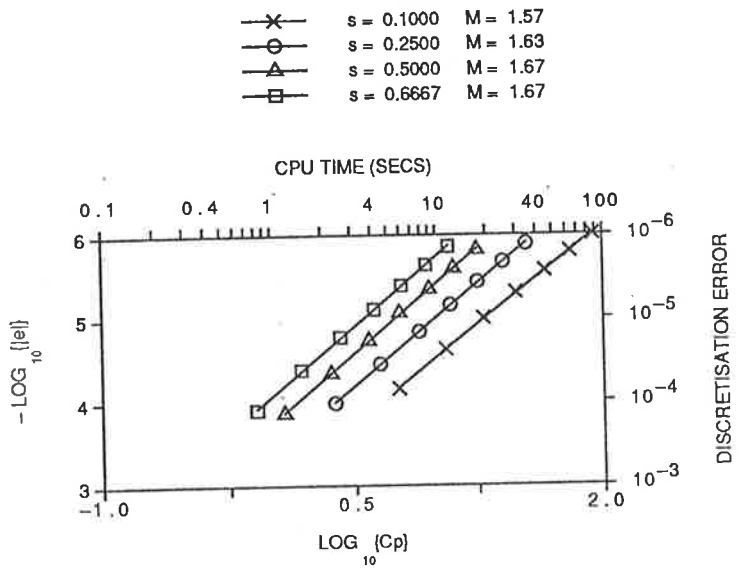


Figure 3.10: *Error vs CPU time graph for the (1,5) equation using an $O\{4\}$ boundary approximation*

The use of still lower-order approximations, such as the third-order approximation

$$\left.\frac{\partial \hat{\tau}}{\partial x}\right|_0^n = \frac{-11\hat{\tau}_0^n + 18\hat{\tau}_1^n - 9\hat{\tau}_2^n + 2\hat{\tau}_3^n}{6\Delta x} + O\{(\Delta x)^3\}, \tag{3.3.7}$$

leads to results that are even less accurate, and of lower than fourth-order, as shown by the slopes of the graphs of Figures 3.11 and 3.12, which show the results of using equation (3.3.7). Such low-order boundary handling is therefore entirely impractical.

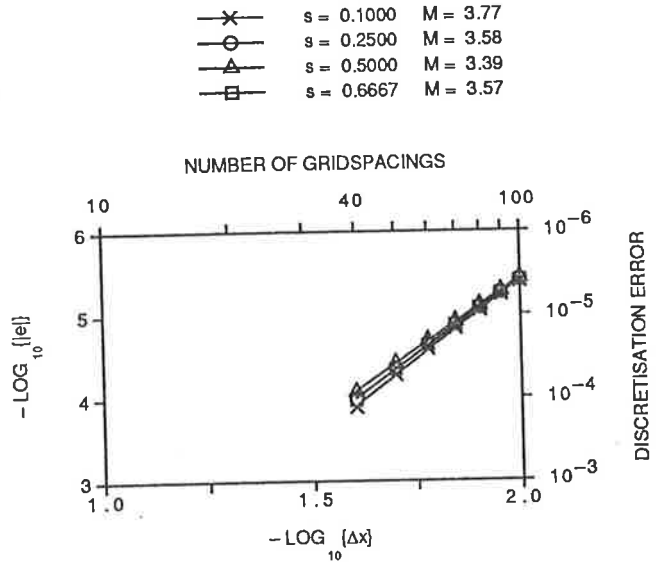


Figure 3.11: *Error vs grid spacing graph for the (1,5) equation using an $O\{3\}$ boundary approximation*

Thus it appears that boundary approximations that are one order of accuracy less that that theoretically required can be used in some cases. Some loss of accuracy is evident if this is done, and the CPU advantage of the (1,5) equation is lost, so this idea is not considered practical to use.

Given the above experience, the sixth-order (1,5,1) equation (2.7.5) can be handled in the same manner as the fourth-order (1,5) equation Although the (1,5,1) equation is sixth-order accurate, and so the boundary treatment should also be at least as accurate, it was seen for the Dirichlet condition that a fourth-order boundary treatment sufficed to give sixth-order results.

For the Neumann condition, however, this proves *not* to be the case. The results for this equation, shown in Figures 3.13 and 3.14, show a marked difference from the

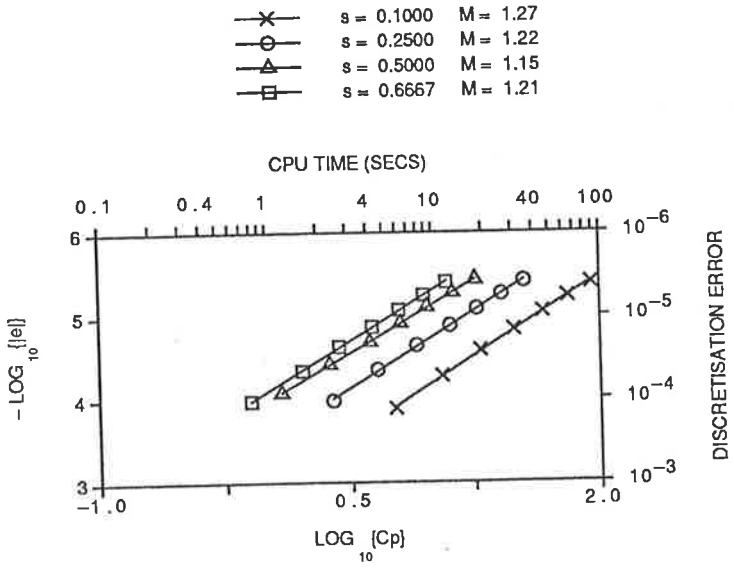


Figure 3.12: *Error vs CPU time graph for the (1,5) equation using an $O\{3\}$ boundary approximation*

expected straight lines of slope six. In fact, the derivative boundary condition has reduced this equation to being only fourth-order accurate for all values of s, except for $s = 1/2$, and even this is not much better than fifth-order. Also worth noting is the fact that the absolute errors are in fact bigger for this equation that for those for the (1,5) equation, which were given above in Figures 3.7 and 3.8.

To overcome these problems would require some sort of sixth-order handling of the boundary, but sixth-order finite-difference equations based on such stencils as (1,3,3) and (3,3,3) have restricted von Neumann stability ranges. Examining the CPU graph for the (1,5,1) equation (Figure 3.14), it is noted that the most efficient values of s are the larger values, with efficiency dropping off dramatically towards $s = 0.1$, as indicated by the much longer CPU times required to get a particular accuracy. Since the use of sixth-order finite-difference equations would force us to use such inefficient values of s to achieve stability, this type of approach was not considered in more detail, especially since efforts in this direction indicated that there was almost no increase in accuracy.

An alternative is to develop an off-centred stencil which is sixth-order accurate, but

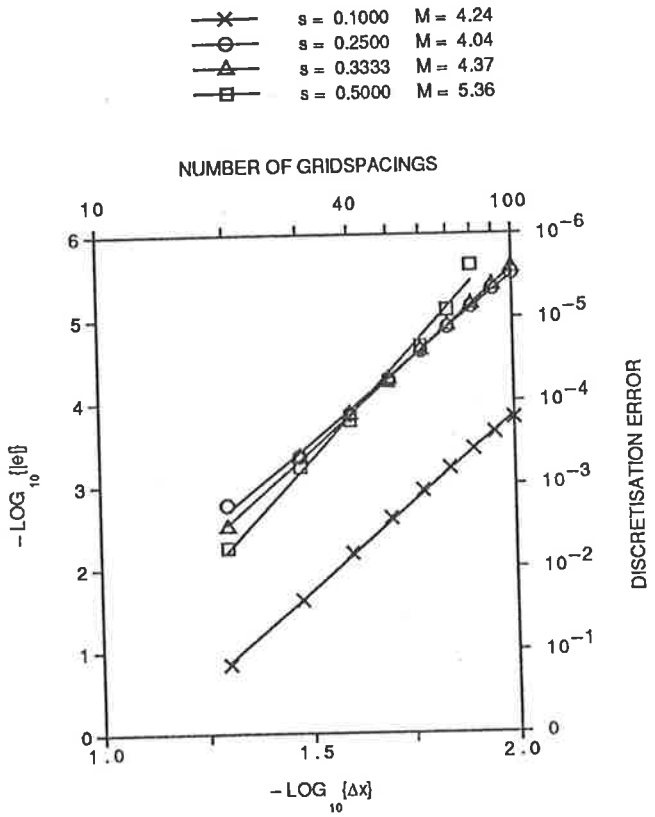


Figure 3.13: *Error vs grid spacing graph for the (1,5,1) equation using only interior grid points*

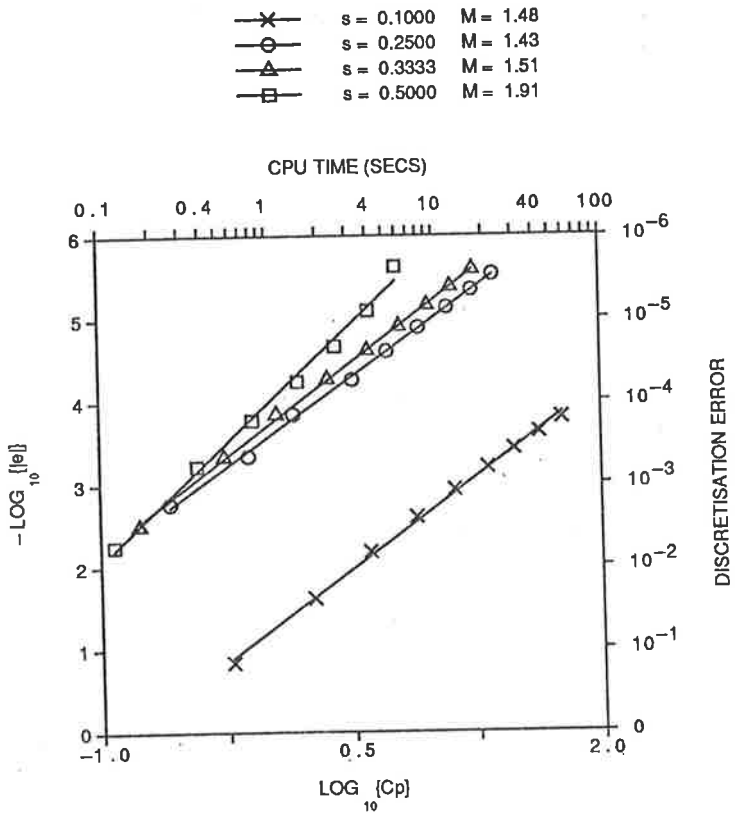


Figure 3.14: *Error vs CPU time graph for the (1,5,1) equation using only interior grid points*

can be used next to the boundaries. To do this, the approximation

$$\left.\frac{\partial^2\hat{\tau}}{\partial x^2}\right|_j^n = \frac{10\hat{\tau}_{j-1}^n - 15\hat{\tau}_j^n - 4\hat{\tau}_{j+1}^n + 14\hat{\tau}_{j+2}^n - 6\hat{\tau}_{j+3}^n + \hat{\tau}_{j+4}^n}{12(\Delta x)^2} + O\{(\Delta x)^4\} \qquad (3.3.8)$$

may be used. The finite-difference equation is developed in exactly the same manner as the centred (1,5,1) equation, using the same weights in the same manner. The resulting equation can be made sixth-order by suitable choice of the weights. The sixth-order finite-difference equation that results is

$$\begin{aligned}\{60s^2 - 150s - 26\}\tau_j^{n+1} &= s^2\{120s^2 - 302\}\tau_{j-1}^n \\ &- \{540s^4 - 729s^2 + 52\}\tau_j^n + s^2\{960s^2 - 316\}\tau_{j+1}^n \\ &+ s^2\{14 - 840s^2\}\tau_{j+2}^n + s^2\{360s^2 - 6\}\tau_{j+3}^n \\ &+ s^2\{1 - 60s^2\}\tau_{j+4}^n + \{26 - 150s - 60s^2\}\tau_j^{n-1}. \qquad (3.3.9)\end{aligned}$$

This equation is found to be von Neumann stable only over the approximate range

$$s \leq 1/3, \qquad (3.3.10)$$

which is somewhat less than the range of the main equation. Numerical results obtained with this equation, in this more restricted stability region, have also proven to be even less accurate. This is despite using the "correct", seventh-order, approximation to the derivative condition, in place of the fifth-order one used for the (1,5) equation. This result may be due to the fact that the equation (3.3.9) is off-centred, so there are odd-order error terms in the modified equivalent equation which are not present for the centred equations. This off-centred technique appears to offer no solution to the problems with the use of the (1,5,1) equation, and is thus not considered further.

Overall, the (1,5,1) equation has produced results of disappointing accuracy with no major CPU time advantage, and so must be considered as unsuitable for use with Neumann boundary conditions until better methods of handling those boundary conditions are found.

3.4 Summary

The addition of a derivative boundary condition to the problem has significantly complicated the solution process, as might be expected. Having re-evaluated the approaches used for the known boundary condition in view of this, it is apparent that the errors are several orders of magnitude larger with the derivative boundary condition included than for the case without it. Despite this, the errors are still small in absolute terms, being of $O\{10^{-7}\}$ in numbers of $O\{10^{-1}\}$, if a grid spacings of $\Delta x = 1/80$ or finer are used.

It is of particular interest that the (1,5,1) equation, which produced very good results for the case of a known boundary value, gives relatively poor results with a derivative boundary condition, despite all efforts at handling the boundary to the correct order of accuracy. Further work is needed before this particular problem can be satisfactorily be overcome.

Overall, the solution process that gives the most accurate answers is the sixth-order (1,3,3) equation (2.5.9), using a seventh-order approximation to the derivative boundary condition. Against this, however, is the CPU time efficiency of using the fourth-order (1,5) equation (2.3.8) in the interior of the solution domain, with the Crandall variant used both to solve the problems at the known boundary at $x = 1$ and to find τ_1^{n+1}. This is followed by an extrapolation based on the derivative approximation (3.3.1) to find the value τ_0^{n+1} at the $x = 0$ boundary. Once again, which of these two methods is preferred is dependent on circumstances, since each has both advantages and disadvantages compared to the other.

Chapter 4

The 2-D Diffusion Equation

4.1 Introduction

The information gained from the preceding study of the one-dimensional diffusion equation, and attempts to produce fast and accurate solution schemes for it, can now be applied to a more general and physically meaningful case, namely the two-dimensional constant-coefficient diffusion equation. This equation can be written as

$$\frac{\partial \hat{\tau}}{\partial t} - \alpha_x \frac{\partial^2 \hat{\tau}}{\partial x^2} - \alpha_y \frac{\partial^2 \hat{\tau}}{\partial y^2} = 0, \tag{4.1.1}$$

where α_x and α_y are considered to be constants.

To compare different methods for solving this equation, the modified equivalent equation approach used for the one-dimensional case can again be used, but some changes need to be made. These changes are necessary because of the fact that the general form of the modified equivalent equation is now

$$\frac{\partial \tau}{\partial t} - \alpha_x \frac{\partial^2 \tau}{\partial x^2} - \alpha_y \frac{\partial^2 \tau}{\partial y^2} + \sum_{p=0}^{\infty} \sum_{q=0}^{p} C_{p,q} \frac{\partial^p \tau}{\partial x^{p-q} \partial y^q} = 0. \tag{4.1.2}$$

From this form, it can be seen that as long as

$$\lim_{\Delta x, \Delta y, \Delta t \to 0} C_{p,q} = 0, \quad p \geq 0 \tag{4.1.3}$$

then the finite-difference scheme is consistent with the two-dimensional diffusion equation (4.1.1). In order for the leading errors to be smaller in magnitude than the solution

values, the condition

$$C_{p,q} = 0 \quad \text{for } p \leq 2 \tag{4.1.4}$$

is also desirable. It is also apparent from this general form that instead of there being only one n^{th} order error term, as for the one-dimensional case, there may now be $(n+3)$ such error terms. This means that to create a method of a given order using a general weighted scheme and choosing appropriate weights, many more weights are now required so that all the desired error terms can be removed. These extra weights can be readily introduced, however, due to the greater number of grid points available to the difference schemes.

The difference schemes to be investigated here will be again written in terms of non-dimensional diffusion parameters, which, in a manner analogous to the one-dimensional case, are defined as

$$s_x = \frac{\alpha_x \Delta t}{(\Delta x)^2} \quad , \quad s_y = \frac{\alpha_y \Delta t}{(\Delta y)^2} \quad . \tag{4.1.5}$$

Given this, the error coefficients $C_{p,q}$ in the modified equivalent equation can be written in the form

$$C_{p,q} = \frac{2\alpha_s (\Delta s)^{p-2}}{p!} \Gamma_{p,q}(s_x, s_y) \qquad \text{if } q = 0 \text{ or } q = p, \tag{4.1.6}$$

$$= \frac{4(\Delta x)^{p-q}(\Delta y)^q}{(p-q)!q!(\Delta t)} \Gamma_{p,q}(s_x, s_y) \qquad \text{if } p \neq 2q, \tag{4.1.7}$$

$$= \frac{4(\Delta x)^q (\Delta y)^q}{(q!)^2 (\Delta t)} \Gamma_{p,q}(s_x, s_y) \qquad \text{if } p = 2q, \tag{4.1.8}$$

where $s \equiv x$ if $q = 0$ and $s \equiv y$ if $p = q$. This particular form has been chosen so that the functions $\Gamma_{p,q}(s_x, s_y)$ are as simple as possible. In particular, they tend to avoid having s_x or s_y in a denominator.

As was noted in earlier chapters, if centred finite difference forms are used to approximate $\partial^2 \hat{\tau}/\partial x^2$ and $\partial^2 \hat{\tau}/\partial y^2$, then the resulting computational stencil is spatially symmetric, and there are no "odd-order" derivative terms in the modified equivalent equation for that method. In the two-dimensional case the derivative term with coefficient $C_{p,q}$ may be considered of "odd-order" if either p is odd or else one (or possibly both) of q and $(p-q)$ is odd. Thus if the stencil is centred, there are no error terms

involving $\partial^3\tau/\partial x^q \partial y^{3-q}$, nor will there be terms like $\partial^4\tau/\partial x \partial y^3$. This simplifies the form of the resulting methods, as well as greatly reducing the number of weights that must be included in a method to force it to be of high order. Most of the methods discussed in this chapter will have centred stencils to take advantage of this fact.

Methods for solving the equation (4.1.1) can be divided into two classes. The first of these are two-dimensional methods, that use a stencil that is itself two-dimensional. These can be further subdivided into explicit and implicit methods, based on whether or not they require the solution of a system of equations at each time step, in exactly the same manner as was done in the one-dimensional case. The other class comprises the one-dimensional methods, which use one-dimensional stencils (either explicit or implicit) to solve the problem.

The method used for denoting two-dimensional methods is exactly the same as that used in one dimension; namely, a method described as being "a (p, q, r) method" uses p points the $(n+1)^{\text{th}}$ time level, q points at the n^{th} and r at the $(n-1)^{\text{th}}$, with the r term being omitted in the case of the method involving only two time levels.

In order to check the numerical accuracy of two-dimensional methods, a test problem is required. The particular problem chosen is the two-dimensional analogue of the Gauss peak (2.2.18) used in the one-dimensional case, namely

$$\hat{\tau}(x, y, t) = \frac{1}{(4t+1)} \exp\left\{-\frac{(x-a)^2}{\alpha_x(4t+1)}\right\} \exp\left\{-\frac{(y-b)^2}{\alpha_y(4t+1)}\right\}, \tag{4.1.9}$$

where the constants a and b are both set to 0.5, so that the peak is centred in the spatial domain. As for the one-dimensional case, the equation (4.1.9) is used to define both the initial and boundary conditions, as well as to generate the exact solution to compare the numerical answers against. The errors for the graphs presented in this chapter are taken at the point $(0.2, 0.2)$, for the same reasons as the choice of the point $x = 0.2$ in the one-dimensional case, and the diffusion parameters used are $\alpha_x = \alpha_y = 0.01$. In this case, however, the tests are only run to $T = 2$, since the amount of CPU time required to use $T = 8$ was found to be excessive, while no extra information was obtained.

4.2 Two Level Explicit Methods

Since explicit methods do not require the solution of a set of linear equations for each time level, they have the potential to be very much faster on a computer than implicit methods. This is more of a consideration in two or more dimensions than it is in one dimension, owing to the vastly increased number of equations that must be solved for the implicit methods, as well as the more complex form of the system of equations in two or more dimensions.

4.2.1 (1,5) Forward-Time Centred-Space Method

The direct analogue in the two-dimensional case to the FTCS method for solving the one-dimensional problem is to use a forward-time approximation to the time derivative at the (j, k, n) grid point, and to approximate both the spatial derivatives by their appropriate second-order centred-space approximations. This gives the two-dimensional finite-difference equation, based on the computational stencil shown in Figure 4.1,

$$\tau_{j,k}^{n+1} = s_x\tau_{j-1,k}^{n} + s_y\tau_{j,k-1}^{n} + (1 - 2s_x - 2s_y)\tau_{j,k}^{n} + s_x\tau_{j+1,k}^{n} + s_y\tau_{j,k+1}^{n}. \qquad (4.2.1)$$

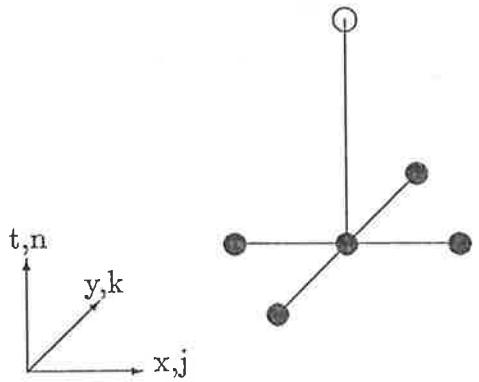


Figure 4.1: *Computational stencil for the (1,5) method*

This method, like its one-dimensional counterpart, is easy to implement, requiring no

special treatment near the boundaries if the boundary values are given and the region is rectangular. Also, being explicit, this method does not require the solution of a set of equations at each time level. If the boundary values are not specified (ie. we have a derivative or mixed boundary condition) then extra work is required to find approximations at grid points on the boundary, but this is true of any method.

A von Neumann stability analysis of equation (4.2.1) leads to the stability criterion

$$s_x + s_y \leq 1/2, \tag{4.2.2}$$

which can be seen to be very restrictive (Roache, 1974). In the symmetric case where $s_x = s_y = s^*$ this condition becomes

$$s^* \leq 1/4 \tag{4.2.3}$$

which is twice as restrictive as the one-dimensional case.

The modified equivalent equation for equation (4.2.1) can be written in the general form (4.1.2) where the leading non-zero error terms contain the factors

$$\begin{aligned} \Gamma_{4,0} &= 6s_x - 1 \\ \Gamma_{4,2} &= s_x s_y \\ \Gamma_{4,4} &= 6s_y - 1. \end{aligned} \tag{4.2.4}$$

Thus this method is seen to be second-order accurate for general values of s_x and s_y. It is also worth noting that although the terms $\Gamma_{4,0}$ and $\Gamma_{4,4}$ can be made to vanish by the choice of values $s_x = s_y = 1/6$, the second-order error term $\Gamma_{4,2}$ is not zero for these values (or indeed any other values of $s_x > 0$ and $s_y > 0$). Thus there are no optimal values of s_x and s_y which make the method fourth-order, which contrasts with the one-dimensional case where $s = 1/6$ makes the method fourth-order.

The actual numerical results for this method, shown in Figures 4.2 and 4.3, show that the method is performing exactly as expected from the theory. The error vs. grid spacing graphs are all straight lines with slopes close to two, indicating that the method is second-order accurate. Also, by similar reasoning to the one-dimensional case, it can be shown that for an order q method,

$$-\log\{|e|\} = (q/4)\log\{Cp\} + K', \tag{4.2.5}$$

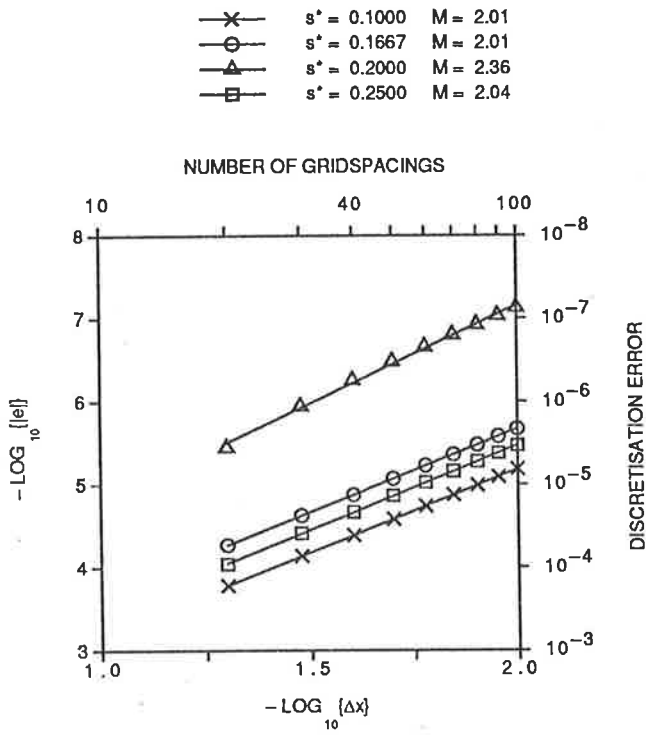


Figure 4.2: *Error vs grid spacing graph for the (1,5) FTCS method (4.2.1)*

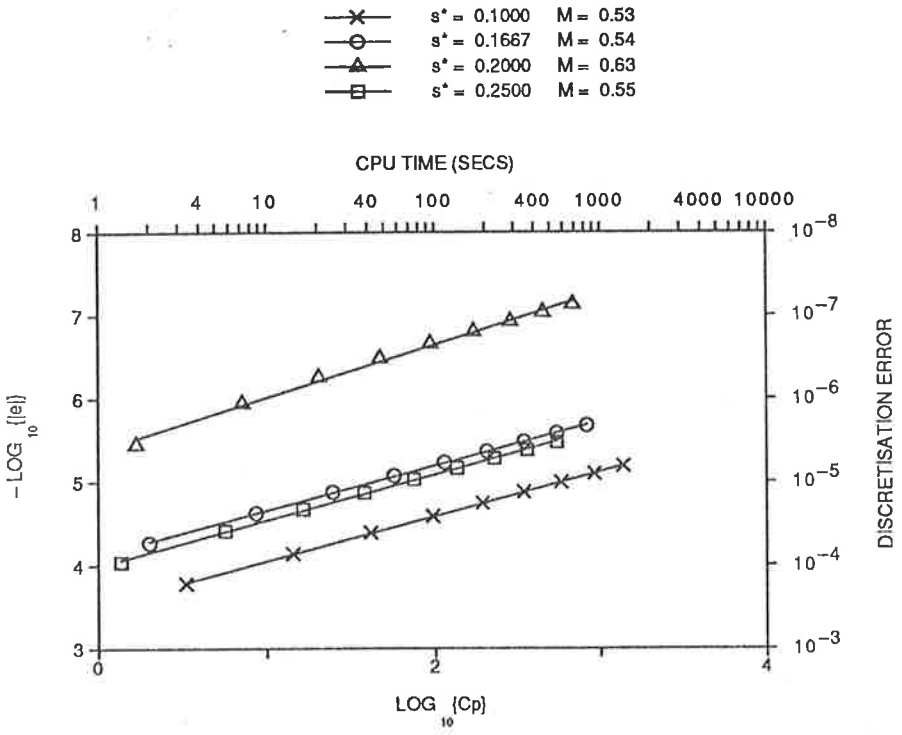


Figure 4.3: *Error vs CPU time graph for the (1,5) FTCS method (4.2.1)*

and once again the graphs agree with this theoretical expectation.

The most important feature of these graphs is the fact that the CPU time taken to generate a solution is so much larger than for the one-dimensional problem. This justifies the selection of the preferred methods in one dimension based on CPU usage as well as error, since a method which uses too much CPU time on the one-dimensional problem will be totally impractical to use when it is generalised into two dimensions.

Note that, as for in the one-dimensional case, increasing the value of s^* and hence the size of the time step does not necessarily decrease the CPU time required to generate a solution. As shown in Figure 4.3, the least amount of CPU time required to find a solution of a given accuracy for the parameters tested occurs for $s^* = 0.200$.

This gives some insight into the problems involved in developing accurate methods for the two-dimensional case, based on accurate methods for the one-dimensional problem. The error terms involved with the cross-derivatives in the modified equivalent equation (such as $\Gamma_{4,2}$ above) have no analogue in the one-dimensional case, and often exhibit quite different behaviour from the pure x or y derivative terms. This must be taken into account when developing methods, for example, by adding extra weights into the general form of a method to permit these error terms to be eliminated.

4.2.2 (1,9) Weighted Explicit Method

The (1,5) method above lacks sufficient grid points to allow the use of weights to try to produce more accurate methods. In order to introduce some weights, more grid points need to be introduced into the computational stencil, so as to allow weighted differencing of the spatial derivative terms in (4.1.1). The (1,9) stencil shown in Figure 4.4 allows for the introduction of two weights, while still retaining a compact stencil. Centred difference approximations with symmetric weighting are used for the spatial derivative terms in order to keep the resulting equation spatially centred. The space derivative in the x direction can be approximated by a weighted combination of centred-space approximations at $k-1$, k and $k+1$. A similar weighting scheme can be used for the space derivative in the y direction.

The weighted scheme that is thus arrived at is

$$\begin{aligned}
\left.\frac{\partial^2\tau}{\partial x^2}\right|_{j,k}^{n} &\approx \varphi \times \{[\text{ CS at } (j,k-1,n)] + [\text{ CS at } (j,k+1,n)]\} \\
&+ (1-2\varphi) \times [\text{ CS at } (j,k,n)], \\
\left.\frac{\partial^2\tau}{\partial y^2}\right|_{j,k}^{n} &\approx \gamma \times \{[\text{ CS at } (j-1,k,n)] + [\text{ CS at } (j+1,k,n)]\} \\
&+ (1-2\gamma) \times [\text{ CS at } (j,k,n)], \qquad (4.2.6)
\end{aligned}$$

where CS is used to denote a three-point centred space difference approximation about the specified grid point. This differencing leads to the (1,9) weighted explicit equation

$$\begin{aligned}
\tau_{j,k}^{n+1} &= \{\varphi s_x + \gamma s_y\}(\tau_{j-1,k-1}^{n} + \tau_{j+1,k-1}^{n} + \tau_{j-1,k+1}^{n} + \tau_{j+1,k+1}^{n}) \\
&+ \{s_y - 2(\varphi s_x + \gamma s_y)\}(\tau_{j,k-1}^{n} + \tau_{j,k+1}^{n}) \\
&+ \{s_x - 2(\varphi s_x + \gamma s_y)\}(\tau_{j-1,k}^{n} + \tau_{j+1,k}^{n}) \\
&+ \{1 - 2(s_x + s_y) + 4(\varphi s_x + \gamma s_y)\}\tau_{j,k}^{n}, \qquad (4.2.7)
\end{aligned}$$

which uses the computational stencil shown in Figure 4.4.

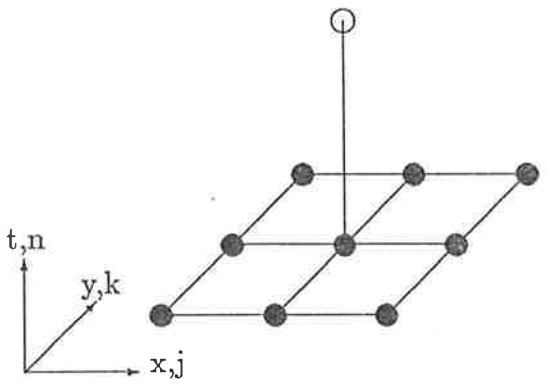


Figure 4.4: *Computational stencil for the (1,9) method*

From this equation, the leading error terms can be examined in order to determine the theoretical order of accuracy of this method, and what values of the weights, if any, will improve this. The modified equivalent equation can be written in the general form (4.1.2), with the leading error terms being

$$\begin{aligned}\Gamma_{4,0} &= 6s_x - 1 \\ \Gamma_{4,2} &= s_x s_y - \varphi s_x - \gamma s_y \\ \Gamma_{4,4} &= 6s_y - 1. \end{aligned} \tag{4.2.8}$$

From this form of the modified equivalent equation, it can be seen that the weights are not involved in the second-order error terms $\Gamma_{4,0}$ and $\Gamma_{4,4}$, so this method cannot be made fourth-order accurate by any choice of weights. What can be done, however, is to remove the cross-derivative term $\Gamma_{4,2}$, which will reduce the second-order error. To do this we set $\Gamma_{4,2} = 0$, which gives

$$\gamma = \frac{s_x(s_y - \varphi)}{s_y} \tag{4.2.9}$$

as the condition on the weights. Since equation (4.1.1) is symmetrical with respect to x and y, it is desirable that a finite-difference equation to solve this equation should possess similar symmetry. In this case, no particular effort is required to achieve this, since the weights φ and γ in equation (4.2.7) only occur in the expression

$$(\varphi s_x + \gamma s_y) \tag{4.2.10}$$

which reduces to $s_x s_y$ on substitution of the expression (4.2.9). Thus removing the error term $\Gamma_{4,2}$ gives the explicit (1,9) finite-difference equation

$$\begin{aligned}\tau_{j,k}^{n+1} &= s_x s_y(\tau_{j-1,k-1}^n + \tau_{j-1,k+1}^n + \tau_{j+1,k-1}^n + \tau_{j+1,k+1}^n) \\ &+ s_y\{1 - 2s_x\}(\tau_{j,k-1}^n + \tau_{j,k+1}^n) \\ &+ s_x\{1 - 2s_y\}(\tau_{j-1,k}^n + \tau_{j+1,k}^n) \\ &+ \{(1 - 2s_x)(1 - 2s_y)\}\tau_{j,k}^n. \end{aligned} \tag{4.2.11}$$

Despite this method being only second-order accurate, it can be shown to be von Neumann stable in the region

$$s_x \leq 1/2 \quad \text{and} \quad s_y \leq 1/2 \tag{4.2.12}$$

which is a marked improvement over the (1,5) FTCS equation (4.2.1), for which the stability limit was $s_x + s_y \leq 1/2$, or in the symmetric case $s_x = s_y = s^* \leq 1/4$. For

equation (4.2.11) the stability criterion for the symmetric case, $s_x = s_y = s^* \leq 1/2$, means that for a given grid spacing and diffusion coefficients, time steps can be twice as large as those used for the (1,5) equation without introducing numerical instabilities, and so a given time can be reached in half as many time steps. Offset against this expectation is the fact that the leading error terms are linear in s_x and s_y, so the error may be expected to increase as s^* is increased, possibly to an unacceptable level.

The fact that no gain of accuracy has been achieved can be verified by looking at the modified equivalent equation corresponding to equation (4.2.11) which can be written in the general form (4.1.2) with leading error terms containing the factors

$$\begin{aligned} \Gamma_{4,0} &= 6s_x - 1 \\ \Gamma_{4,2} &= 0 \\ \Gamma_{4,4} &= 6s_y - 1 \end{aligned} \tag{4.2.13}$$

as expected from the derivation of equation (4.2.11). Note that the choice $s_x = s_y = 1/6$ will make this scheme fourth-order.

The numerical results for the method, shown in Figures 4.5 and 4.6, largely reflect the theoretical findings, including the special case of $s^* = 1/6$ where the method is fourth-order, as opposed to the second-order which is apparent for the other values shown. In the general case, however, the results are somewhat less accurate than those from the (1,5) equation (4.2.1), shown in Figures 4.2 and 4.3. Also, the most efficient value of s^* (excluding $s^* = 1/6$), in terms of minimising the amount of CPU time required to generate answers of a given accuracy, appears to be $s^* = 0.25$. In general, therefore, the (1,9) method requires much more CPU time to generate a solution of a given accuracy than the (1,5) equation, so the (1,9) equation is of little practical use.

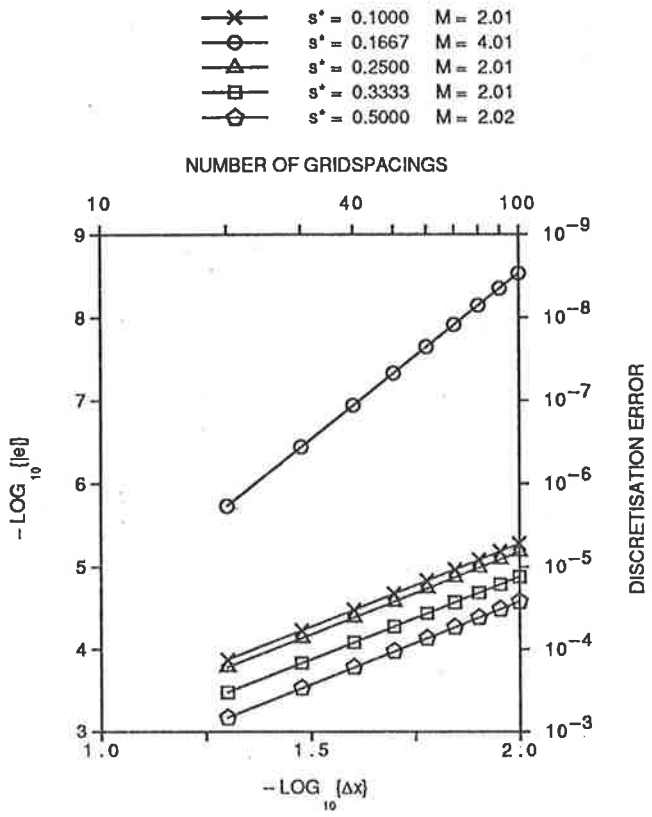


Figure 4.5: *Error vs grid spacing graph for the (1,9) method (4.2.11)*

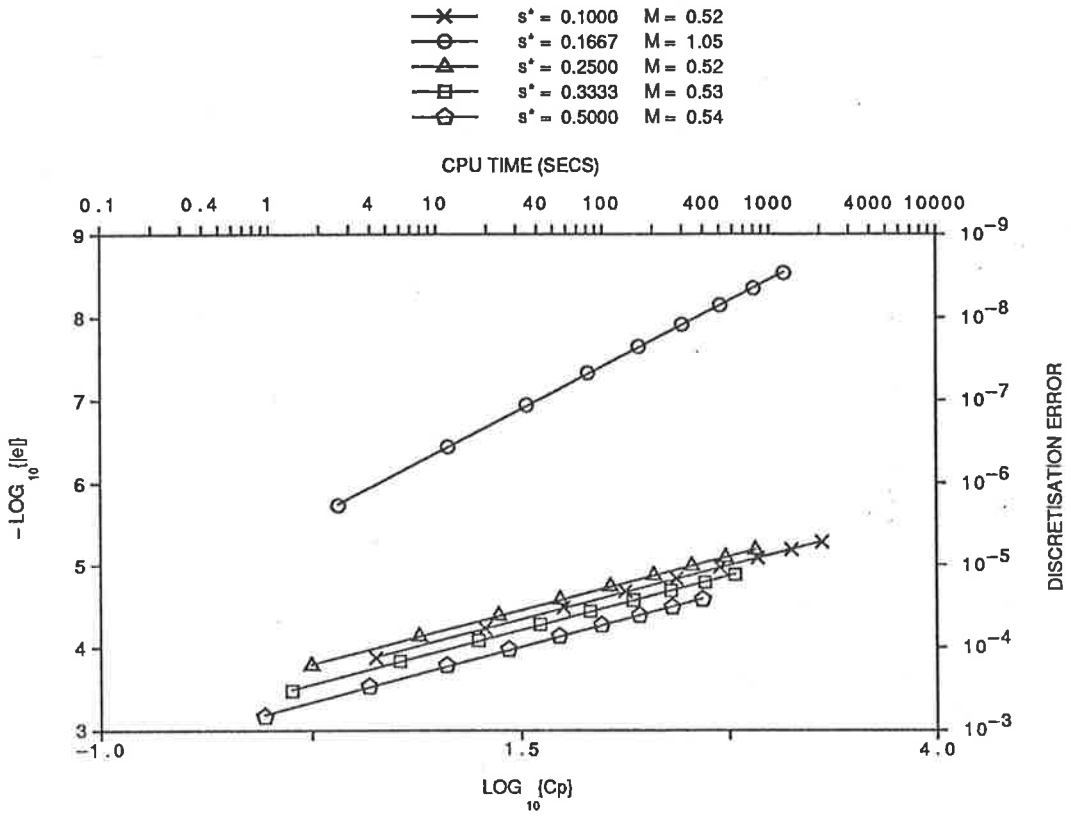


Figure 4.6: *Error vs CPU time graph for the (1,9) method (4.2.11)*

4.2.3 (1,13) Weighted Explicit Method

As happened for the one-dimensional case, the previous two sections have shown that, using two-level explicit stencils which are only three spatial grid points wide to approximate the spatial derivatives, finite-difference equations cannot be derived to give more than second-order accuracy for general s_x and s_y. Although there are specific values of s_x and s_y which will make some of these methods fourth-order, for a method to be of practical use it should allow for variations of these values without degrading the accuracy of the solution. Also, these specific values restrict the size of the time step allowed to values which are much smaller than is desirable in practice.

Again, five-point approximations to the spatial derivatives can be used to overcome these limitations. This again leads to problems with finding values at grid points next to a boundary, and these problems must be addressed in order for any method to be useful in practice. In order to keep these problems to a minimum, the first case considered will use only two five-point approximations; one for $\partial^2\tau/\partial x^2$ and the other for $\partial^2\tau/\partial y^2$, both about the (j,k,n) grid point. This leads to a (1,13) stencil, shown in Figure 4.7, which is the minimum possible explicit extension of the (1,9) case, given that the equation must be kept spatially centred so as to force as many of the error terms as possible to be zero.

The weighted differencing used in this case is

$$\begin{aligned}
\left.\frac{\partial^2\tau}{\partial x^2}\right|_{j,k}^{n} &\approx \varphi \times \{[\text{ CS3 at } (j,k-1,n)] + [\text{ CS3 at } (j,k+1,n)]\} \\
&+ \gamma \times [\text{ CS5 at } (j,k,n)] \\
&+ (1-2\varphi-\gamma) \times [\text{ CS3 at } (j,k,n)], \\
\left.\frac{\partial^2\tau}{\partial y^2}\right|_{j,k}^{n} &\approx \theta \times \{[\text{ CS3 at } (j-1,k,n)] + [\text{ CS3 at } (j+1,k,n)]\} \\
&+ \epsilon \times [\text{ CS5 at } (j,k,n)] \\
&+ (1-2\theta-\epsilon) \times [\text{ CS3 at } (j,k,n)], \qquad (4.2.14)
\end{aligned}$$

where CS3 and CS5 are used to represent three and five-point centred-space approximations respectively to the spatial derivative.

Using these weighted approximations gives the weighted (1,13) explicit finite-difference

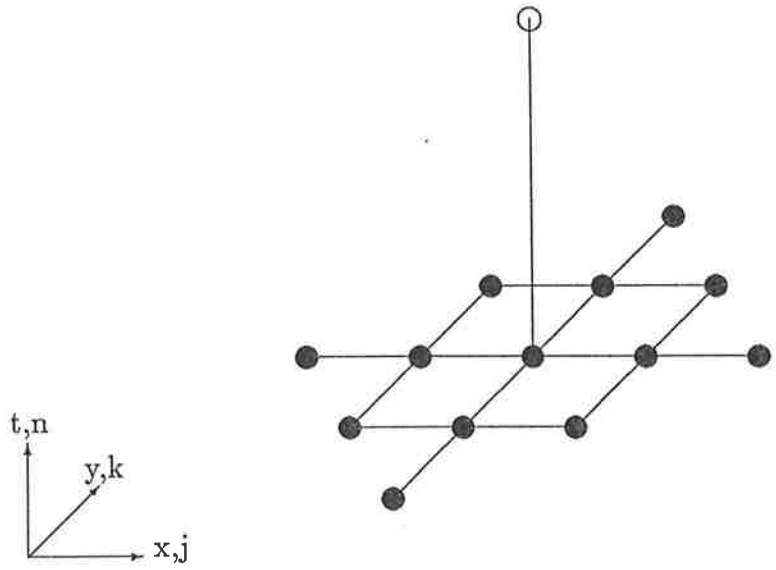


Figure 4.7: *Computational stencil for the (1,13) method*

equation

$$
\begin{aligned}
-12\tau^n_{j,k} &= \gamma s_x(\tau^n_{j-2,k} + \tau^n_{j+2,k}) + \epsilon s_y(\tau^n_{j,k-2} + \tau^n_{j,k+2}) \\
&- 12\{\varphi s_x + \theta s_y\}(\tau^n_{j-1,k-1} + \tau^n_{j-1,k+1} + \tau^n_{j+1,k-1} + \tau^n_{j+1,k+1}) \\
&+ 4\{6(\varphi s_x + \theta s_y) - (3+\gamma)s_x\}(\tau^n_{j-1,k} + \tau^n_{j+1,k}) \\
&+ 4\{6(\varphi s_x + \theta s_y) - (3+\epsilon)s_y\}(\tau^n_{j,k-1} + \tau^n_{j,k+1}) \\
&+ 6\{(4 - 8\varphi + \gamma)s_x + (4 - 8\theta + \epsilon)s_y - 2\}\tau^n_{j,k}. \qquad (4.2.15)
\end{aligned}
$$

The modified equivalent equation corresponding to (4.2.15) can be written in the general form (4.1.2) with leading error terms containing the factors

$$
\begin{aligned}
\Gamma_{4,0} &= \gamma + 6s_x - 1 \\
\Gamma_{4,2} &= s_x s_y - \varphi s_x - \theta s_y \\
\Gamma_{4,4} &= \epsilon + 6s_y - 1 \qquad (4.2.16)
\end{aligned}
$$

so in general the method can be seen to be second-order accurate. It can be seen from (4.2.16) however, that if the weights are chosen to have values which satisfy

$$\begin{aligned} \gamma &= 1 - 6s_x \\ \varphi &= \frac{s_y(s_x - \theta)}{s_x} \\ \epsilon &= 1 - 6s_y \end{aligned} \tag{4.2.17}$$

then all of the second-order error terms will be zero, and so the resulting method must be at least fourth-order accurate, as $\Gamma_{5,q} = 0$ for all q.

On substituting the expressions (4.2.17) into equation (4.2.15), the resulting equation is

$$\begin{aligned} -12\tau_{j,k}^{n+1} &= s_x\{1 - 6s_x\}(\tau_{j-2,k}^n + \tau_{j+2,k}^n) + s_y\{1 - 6s_y\}(\tau_{j,k-2}^n + \tau_{j,k+2}^n) \\ &- 12s_xs_y(\tau_{j-1,k-1}^n + \tau_{j-1,k+1}^n + \tau_{j+1,k-1}^n + \tau_{j+1,k+1}^n) \\ &+ 8s_x\{3s_x + 3s_y - 2\}(\tau_{j-1,k}^n + \tau_{j+1,k}^n) \\ &+ 8s_y\{3s_x + 3s_y - 2\}(\tau_{j,k-1}^n + \tau_{j,k+1}^n) \\ &+ 6\{s_x(5 - 6s_x - 4s_y) + s_y(5 - 6s_y - 4s_x) - 2\}\tau_{j,k}^n. \end{aligned} \tag{4.2.18}$$

The thing to be noted about this equation is the fact that the fourth weight θ, has also been eliminated from this equation. As in the case of the (1,9) method, this is due to the fact that it was only present in (4.2.15) as part of the sub-expression

$$(\varphi s_x + \theta s_y) \tag{4.2.19}$$

and this expression simplifies to s_xs_y on substitution of (4.2.17).

The modified equivalent equation corresponding to equation (4.2.18) can be written in the general form (4.1.2) with leading error terms

$$\begin{aligned} \Gamma_{6,0} &= 2(2 - 15s_x + 30{s_x}^2) \\ \Gamma_{6,2} &= s_xs_y(6s_x - 1) \\ \Gamma_{6,4} &= s_xs_y(6s_y - 1) \\ \Gamma_{6,6} &= 2(2 - 15s_y + 30{s_y}^2), \end{aligned} \tag{4.2.20}$$

which verifies that the method is fourth-order accurate for all values of s_x and s_y. A numerical von Neumann stability analysis of the method reveals that the equation

(4.2.18) is stable for

$$s_x + s_y \leq 2/3, \tag{4.2.21}$$

which is a larger region of the (s_x, s_y) plane than that for the (1,5) equation (4.2.1) but not as large as that for the (1,9) equation (4.2.11). This slight disadvantage relative to the latter case is more than offset by the fact that this method is fourth-order accurate, whereas both of the previous methods were only second-order. It should also be noted that although the choice of $s_x = s_y = 1/6$ will force the cross-derivative error terms $\Gamma_{6,2}$ and $\Gamma_{6,4}$ to be zero, the other fourth-order error terms cannot be eliminated, so there are no special values of s_x and s_y which make this method sixth-order.

Having shown that this method is more accurate than the others considered so far, the problems near the boundary, due to the large spatial spread of grid points, must now be overcome, or else the method is of no practical use. The difficulty, as in the one-dimensional case, is that when the equation (4.2.18) is used to compute the values at the grid point next to a boundary, a reference is made to a grid point outside the boundary, and the value of τ at this point is not known.

This problem can be overcome in several ways in the case where the boundary values are known. The first method involves extrapolating values of τ at external grid points (at the n^{th} time level) to sixth-order, using the formula

$$\tau^n_{-1,k} \approx 6\tau^n_{0,k} - 15\tau^n_{1,k} + 20\tau^n_{2,k} - 15\tau^n_{3,k} + 6\tau^n_{4,k} - \tau^n_{5,k}, \tag{4.2.22}$$

which produces the same accuracy as the finite-difference method applied at the interior grid points. This value can then be used directly in equation (4.2.18) to find the value next to the boundary at the new time level. This has the advantage that it is easy to implement and does not use large amounts of CPU time. It may, however, significantly detract from the theoretical stability region of the method, particularly as the order of accuracy of the extrapolation increases.

Looking at actual numerical results, shown in Figures 4.8 and 4.9, the fourth-order accuracy of equation (4.2.18) is apparent. It is also apparent that the results from this method are much more accurate, in absolute terms, than those produced by the second-order methods discussed above. It should be noted that there is no sign of

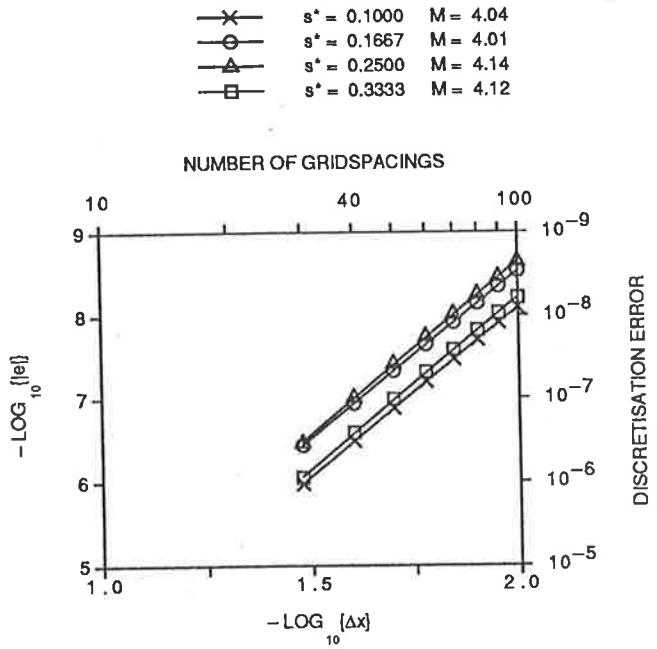


Figure 4.8: *Error vs grid spacing graph for the (1,13) method (4.2.18), using extrapolation at the boundaries*

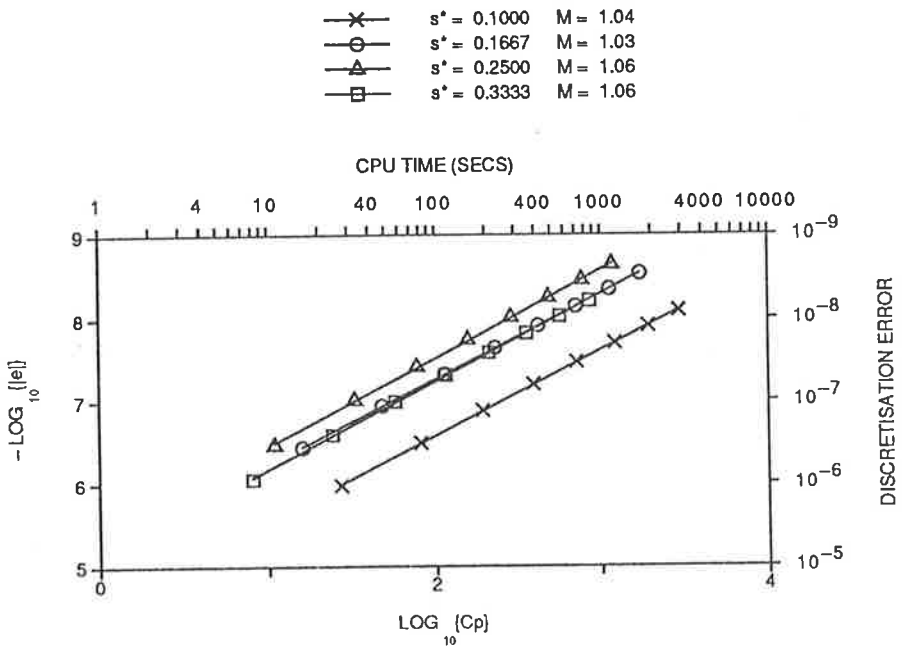


Figure 4.9: *Error vs CPU time graph for the (1,13) method (4.2.18), using extrapolation at the boundaries*

any numerical instability in these results, such as may have been introduced by the extrapolation to the exterior grid points. Any instability introduced by the extrapolation appears to have been damped out by the main finite-difference equation, so this boundary technique is practical to use. Note that this may not be so for other finite-difference equations which may be used in the interior of the region.

Another way to overcome the boundary problem is to use a compact three-level method, such as a (1,9,9) method, to find the value next to the boundary. The method chosen must be of the same order of accuracy as the method used for the rest of the region, namely fourth-order, and should also be stable over at least the same region as the main method. Some three-level methods that may be used for this purpose are described later in Section 4.3. It is shown there that such three-level methods, as in the one-dimensional case, tend to have more restricted stability ranges than two-level methods. In particular, the fourth-order (1,9,9) equation has a stability range that is much smaller than that for the (1,13) equation itself. Since there are other practical, fourth-order schemes for handling the boundaries, the use of three-level methods has not been investigated further.

Another different approach is to calculate all the values at the next time level except those next to the boundaries, then use interpolation with sixth-order accuracy at the new time level to fill in these values. This technique has the advantage of being easy to implement and quick to run on a computer, so the computing time advantage of using an explicit method is preserved. Again the effect on numerical stability of using this scheme must be examined to ensure that the method is still of practical use. In this case, an adverse affect on the stability of the whole scheme is evident, and this technique also has thus been pursued no further.

The last method to be considered here involves the use of an implicit (9,9) centred stencil to find a set of equations relating the values next to the boundary. One advantage of this is that it is possible to produce a method based on this stencil that is unconditionally stable, so the overall method remains stable for the same region as the interior method. The (9,9) stencil is used by first calculating all the values at the new time level except those next to the boundary (which are the values to be

found). Starting from any corner position, number the unknown grid points from 1 to $(2J+2K-8)$, where there are J grid points in the x direction and K in the y direction (the direction of numbering is unimportant). An example of such a numbering scheme is shown in Figure 4.10. Then the use of a (9,9) centred stencil (ie. a 3×3 square of grid points at each time level) in the corner of the region with grid point 1 gives an equation for points labelled $(2J+2K-8)$, 1 and 2. Shifting the stencil one grid point along, in the same direction as the points were numbered, gives an equation involving the same three points with point 3 added. Another shift produces an equation involving points 2, 3 and 4 only. As the stencil is shifted around the boundary of the region, a set of $(2J+2K-8)$ linear equations is built up.

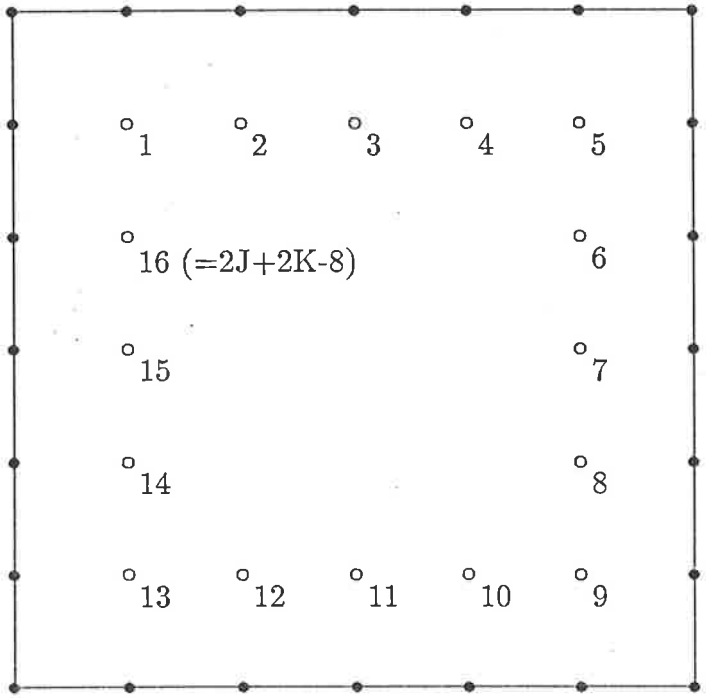


Figure 4.10: *One possible numbering scheme of grid points for the boundary solution scheme based on a (9,9) stencil, with* $J = K = 6$

The structure of these equations is almost tri-diagonal, but there are ten non-zero values which are not part of a normal tri-diagonal system. Two of these (corresponding to the presence of point $(2J+2K-8)$ in the first equation and point 1 in the last one) can be left in the system if a special cyclic solver is employed (Evans and Hatzopoulos,

1976), and the other eight values can be easily removed by elimination with other equations.

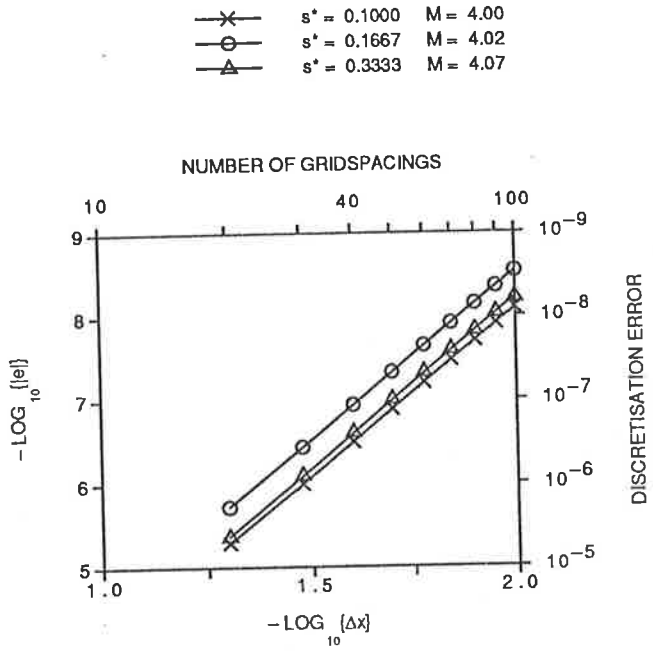


Figure 4.11: *Error vs grid spacing graph for the (1,13) method (4.2.18), using a (9,9) equation at the boundaries*

Since the (9,9) method used to generate these equations is fourth-order the accuracy of the values found next to the boundary will match that of the values in the rest of the region. The numerical results for this boundary scheme, shown in Figures 4.11 and 4.12, are very similar to the results obtained for previous boundary techniques, both in terms of the accuracy of the solutions and the amount of CPU time required to generate the solution. This is somewhat surprising, since the extra requirement to set up and solve a set of linear equations at each time step could be expected to add significantly to the CPU time required. This result, with only a marginal increase in the CPU usage, shows that the vast majority of the CPU time required to generate the solution is used in the interior of the region, so the extra time used at the boundary is not a significant factor.

Note that the value $s^* = 0.25$ is omitted from the numerical trials of the (9,9) boundary technique, because in this case the $O\{4\}$ (9,9) equation used has several zero coefficients, which changes the structure of the system of equations that needs to be solved.

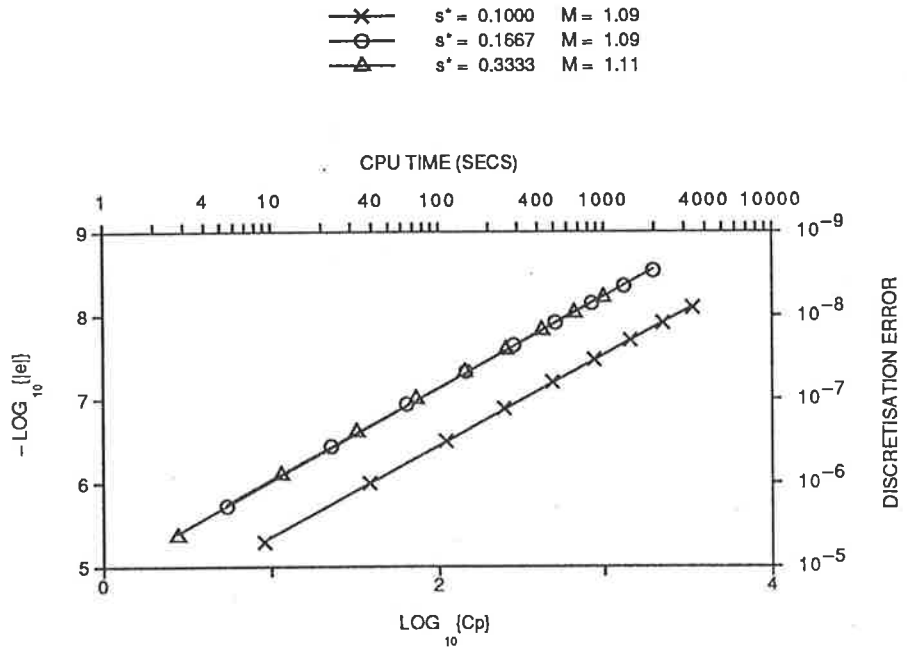


Figure 4.12: *Error vs CPU time graph for the (1,13) method (4.2.18), using a (9,9) equation at the boundaries*

This leads to an attempt to divide by zero in the solution process, and while a special case could be made for this, doing this would make meaningful comparisons of CPU time impossible. Also note that the (9,9) method of handling the boundary problems is useful since it may be applied without changes to both the logical extensions of the (1,13) stencil, namely the (1,21) and (1,25) stencils, which are discussed later. This contrasts with techniques such as extrapolation, where several extra external values must be computed for each of these extensions.

4.2.4 (1,21) Weighted Explicit Method

Having found one reasonably stable, fourth-order accurate method for solving equation (4.1.1), consideration is now given to finding methods which are either more accurate then this (ie. sixth-order), or which possess a larger stability range, or both. Methods should not disproportionately increase the CPU time used; ideally the same amount of CPU time should give solutions of comparable accuracy. This objective requires more points to be introduced into the computational stencil, but once again care must

be taken to ensure that all spatial differencing is kept centred. This is achieved by including five-point approximations to the spatial derivatives, one grid spacing either side of the (j,k,n) grid point. By doing this, it is now possible to incorporate extra weights into the scheme, which can be used to either increase the order of accuracy, if possible, or to try to increase the stability range of the method.

In fact, it is possible to include eight weights into the computational stencil, by weighting the spatial derivative terms as follows:

$$\begin{aligned}
\left.\frac{\partial^2\tau}{\partial x^2}\right|^n_{j,k} &\approx \varphi\times\{[\text{ CS3 at } (j,k-2,n)]+[\text{ CS3 at } (j,k+2,n)]\}\\
&+ \chi\times[\text{ CS3 at } (j,k,n)]+(1-2(\varphi+\gamma+\pi)-\chi)\times[\text{ CS5 at } (j,k,n)],\\
&+ \gamma\times\{[\text{ CS3 at } (j,k-1,n)]+[\text{ CS3 at } (j,k+1,n)]\}\\
&+ \pi\times\{[\text{ CS5 at } (j,k-1,n)]+[\text{ CS5 at } (j,k+1,n)]\}\\
\left.\frac{\partial^2\tau}{\partial y^2}\right|^n_{j,k} &\approx \theta\times\{[\text{ CS3 at } (j-2,k,n)]+[\text{ CS3 at } (j+2,k,n)]\}\\
&+ \eta\times[\text{ CS3 at } (j,k,n)]+(1-2(\theta+\epsilon+\lambda)-\eta)\times[\text{ CS5 at } (j,k,n)]\\
&+ \epsilon\times\{[\text{ CS3 at } (j-1,k,n)]+[\text{ CS3 at } (j+1,k,n)]\}\\
&+ \lambda\times\{[\text{ CS5 at } (j-1,k,n)]+[\text{ CS5 at } (j+1,k,n)]\}. \qquad (4.2.23)
\end{aligned}$$

This appears to give much more flexibility than even the (1,13) scheme above, due to the extra number of weights which can be used to eliminate error terms or increase stability.

The finite-difference equation that results from this differencing is

$$\begin{aligned}
-12\tau^{n+1}_{j,k} &= \{\pi s_x-12\theta s_y\}(\tau^n_{j-2,k-1}+\tau^n_{j-2,k+1}+\tau^n_{j+2,k-1}+\tau^n_{j+2,k+1})\\
&+ \{(1-2\varphi-2\gamma-2\pi-\chi)s_x+24\theta s_y\}(\tau^n_{j-2,k}+\tau^n_{j+2,k})\\
&+ \{(1-2\theta-2\epsilon-2\lambda-\eta)s_y+24\varphi s_x\}(\tau^n_{j,k-2}+\tau^n_{j,k+2})\\
&+ \{\lambda s_y-12\varphi s_x\}(\tau^n_{j-1,k-2}+\tau^n_{j+1,k-2}+\tau^n_{j-1,k+2}+\tau^n_{j+1,k+2})\\
&- 4\{(4\pi+3\gamma)s_x+(4\lambda+3\epsilon)s_y\}(\tau^n_{j-1,k-1}+\tau^n_{j-1,k+1}+\tau^n_{j+1,k-1}+\tau^n_{j+1,k+1})\\
&+ \{4(\chi+8\varphi+8\gamma+8\pi-4)s_x+6(4\epsilon+5\lambda)s_y\}(\tau^n_{j-1,k}+\tau^n_{j+1,k})\\
&+ 6\{(5-\chi-10\varphi-10\gamma-10\pi)s_x+(5-\eta-10\theta-10\epsilon-10\lambda)s_y-2\}\tau^n_{j,k}\\
&+ \{4(\eta+8\theta+8\epsilon+8\lambda-4)s_y+6(4\gamma+5\pi)s_x\}(\tau^n_{j,k-1}+\tau^n_{j,k+1}) \qquad (4.2.24)
\end{aligned}$$

which is fully centred about the (j,k) grid position, as was intended by the differencing chosen. The computational stencil used by this equation is shown in Figure 4.13. The modified equivalent equation corresponding to this finite-difference equation can be written in the general form (4.1.2), with leading error terms which contain the factors

$$\begin{aligned}\Gamma_{4,0} &= 6s_x - \chi - 2\gamma - 2\varphi \\ \Gamma_{4,2} &= s_x s_y - (\gamma + \pi + 4\varphi)s_x - (\epsilon + \lambda + 4\theta)s_y \\ \Gamma_{4,4} &= 6s_y - \eta - 2\epsilon - 2\theta. \end{aligned} \tag{4.2.25}$$

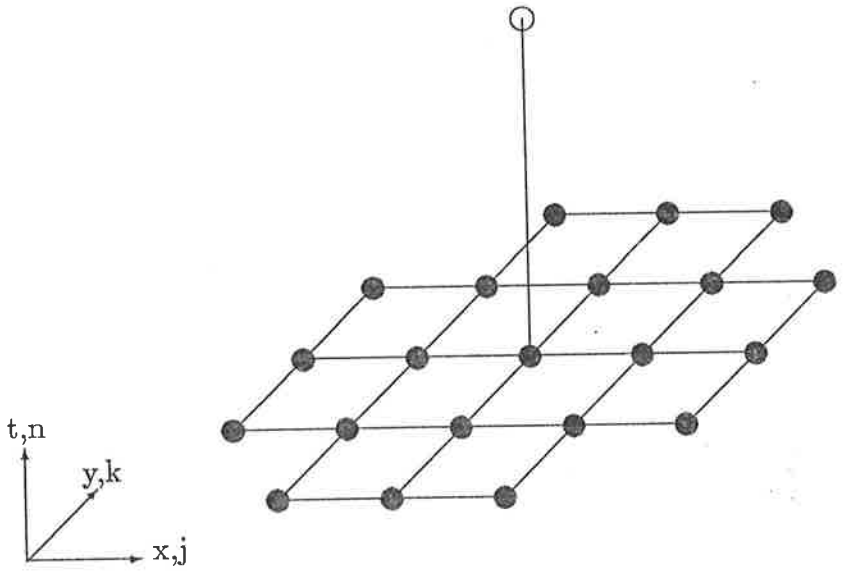


Figure 4.13: *Computational stencil for the (1,21) method*

All these terms involve some of the weights from the method, so it may be possible to use these weights to remove the error terms (4.2.25) and produce a method of at least fourth, if not sixth, order accuracy. Solution of the equations to make (4.2.24) fourth-order leads to the values

$$\begin{aligned}\chi &= 6s_x - 2\varphi - 2\gamma \\ \lambda &= s_x - (\pi + \gamma + 4\varphi)(s_x/s_y) - (\epsilon + 4\theta) \\ \eta &= 6s_y - 2\theta - 2\epsilon \end{aligned} \tag{4.2.26}$$

being required for three of the weights. Note however that this choice of weights will remove the symmetry with respect to x and y from the equation. To overcome this

lack of symmetry, the expression for λ is replaced by the pair of more specialised conditions

$$\begin{aligned} \pi &= s_y/2 - \gamma - 4\varphi. \\ \lambda &= s_x/2 - \epsilon - 4\theta \end{aligned} \tag{4.2.27}$$

This substitution leaves four weights free in the equation, and since there are four fourth-order error terms in the modified equivalent equation (since it is centred), deriving a sixth-order method may still be possible. However, examination of the modified equivalent equation corresponding to the new equation with the values (4.2.26) and (4.2.27) substituted shows that the new leading error terms, written in the general form (4.1.2), contain the factors

$$\begin{aligned} \Gamma_{6,0} &= 60{s_x}^2 - 30s_x + 4 \\ \Gamma_{6,2} &= s_x(6s_x s_y - 4\varphi - \gamma - s_y/2) - 12\theta s_y \\ \Gamma_{6,4} &= s_y(6s_x s_y - 4\theta - \epsilon - s_x/2) - 12\varphi s_x \\ \Gamma_{6,6} &= 60{s_y}^2 - 30s_y + 4, \end{aligned} \tag{4.2.28}$$

from which it can be seen that the terms $\Gamma_{6,0}$ and $\Gamma_{6,6}$ have no weights in them, and so cannot be removed by any choice of values for the weights.

Thus the (1,21) method is no more accurate than the (1,13) method, and needs more effort near the boundaries to cope with the additional points that would ordinarily fall outside the boundary of the region when the stencil is used at grid points next to the boundaries. Given this, there is very little use looking at this scheme further, unless it can be shown to be much more stable than the (1,13) scheme for some choice(s) of the remaining weights.

The resulting (1,21) equation has too many weights involved to effectively investigate its numerical stability, so the error terms $\Gamma_{6,2}$ and $\Gamma_{6,4}$ will be eliminated first. In order to do this, the values

$$\begin{aligned} \gamma &= (6s_x - 1/2 - 12\theta/s_x)s_y - 4\varphi \\ \epsilon &= (6s_y - 1/2 - 12\varphi/s_y)s_x - 4\theta \end{aligned} \tag{4.2.29}$$

are substituted into equation (4.2.24) with the other weights given above. This removes the fourth-order cross derivative terms involving $\Gamma_{6,2}$ and $\Gamma_{6,4}$ from the modified equation, as well as simplifying the finite-difference equation by removing two more weights.

The finite-difference that results from all the above substitutions is given by

$$\begin{aligned}
-12\tau_{j,k}^{n+1} &= s_x s_y\{1-6s_x\}(\tau_{j-2,k-1}^n + \tau_{j-2,k+1}^n + \tau_{j+2,k-1}^n + \tau_{j+2,k+1}^n) \\
&+ s_x\{(1-6s_x)(1-2s_y)\}(\tau_{j-2,k}^n + \tau_{j+2,k}^n) \\
&+ s_y\{(1-2s_x)(1-6s_y)\}(\tau_{j,k-2}^n + \tau_{j,k+2}^n) \\
&+ s_x s_y\{1-6s_y\}(\tau_{j-1,k-2}^n + \tau_{j-1,k+2}^n + \tau_{j+1,k-2}^n + \tau_{j+1,k+2}^n) \\
&+ 4s_x s_y\{6s_x + 6s_y - 5\}(\tau_{j-1,k-1}^n + \tau_{j-1,k+1}^n + \tau_{j+1,k-1}^n + \tau_{j+1,k+1}^n) \\
&+ 2s_x\{12s_x + 19s_y - 24s_x s_y - 18{s_y}^2 - 8\}(\tau_{j-1,k}^n + \tau_{j+1,k}^n) \\
&+ 2s_y\{19s_x + 12s_y - 24s_x s_y - 18{s_x}^2 - 8\}(\tau_{j,k-1}^n + \tau_{j,k+1}^n) \\
&+ 6\{-2 + 5(s_x + s_y) - 12s_x s_y - 6({s_x}^2 + {s_y}^2) + 12s_x s_y(s_x + s_y)\}\tau_{j,k}^n.
\end{aligned} \tag{4.2.30}$$

Note that here, as in previous cases, the substitution of specified values for most of the weights has eliminated all the weights. The von Neumann stability of this equation can be determined numerically and is found to be convex region bounded by

$$\begin{aligned}
0 \leq\ & s_x & \leq 2/3 \\
& s_x + s_y & < 1 \\
0 \leq\ & s_y & \leq 2/3,
\end{aligned} \tag{4.2.31}$$

shown in Figure 4.14. This is a slightly larger stability region than the (1,13) equation, so a numerical test of the method is required to determine which is the preferred method of solution. The best method of handling the boundary problems in this case is to use the (9,9) implicit scheme, since it used no more CPU for the (1,13) case than the other boundary techniques examined, it can be used here in exactly the same form, and imposes no extra stability restrictions on the method.

The results from a numerical experiment, shown in Figures 4.15 and 4.16, show that the results are very similar to those for the (1,13) equation. The accuracy obtained

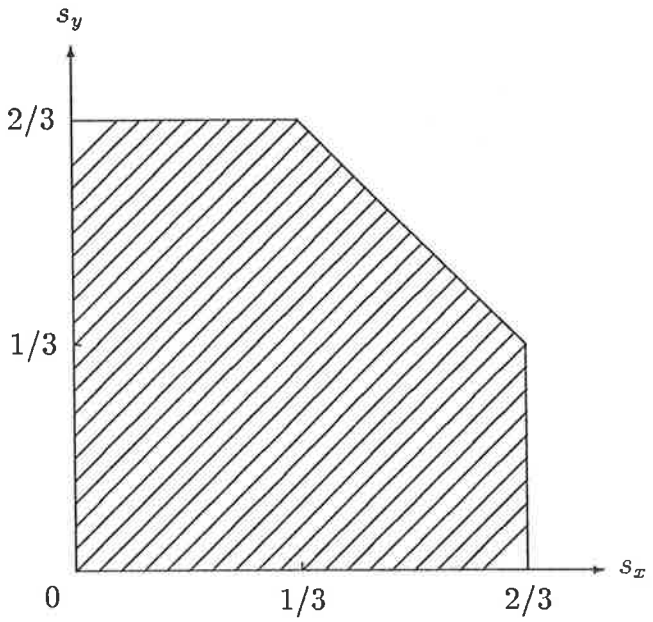


Figure 4.14: *Von Neumann stability region for the optimal (1,21) equation. Note that the line* $s_x + s_y = 1$ *is excluded from the region*

is practically identical, as is the CPU time required for each run. Overall, there is little to distinguish this method from the (1,13) method described above. The (1,13) equation may be preferred because it involves fewer grid points and is thus somewhat simpler to implement, but even this is only a marginal difference.

4.2.5 (1,25) Weighted Explicit Method

A final attempt to produce an explicit sixth-order method is now made by using the full 5×5 grid of points at the n^{th} time level to produce a weighted (1,25) method. This can be done by using the same basic weighting as for the (1,21) method described above, with the addition of the five-point approximations at the edge of the stencil. This adds an extra weight in each spatial direction, which will be denoted by ω in the x-direction and σ in the y-direction.

Such a method still has the same problems as the (1,13) and (1,21) equations near

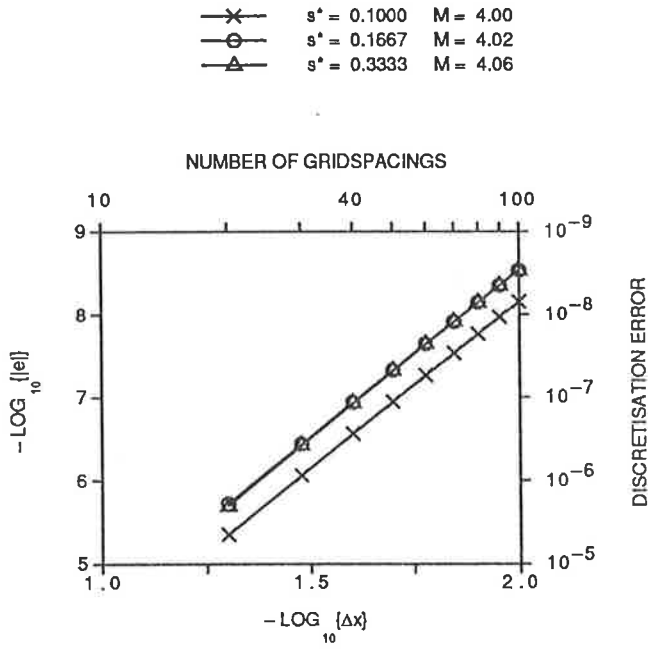


Figure 4.15: *Error vs grid spacing graph for the (1,21) method (4.2.30), using a (9,9) equation at the boundaries*

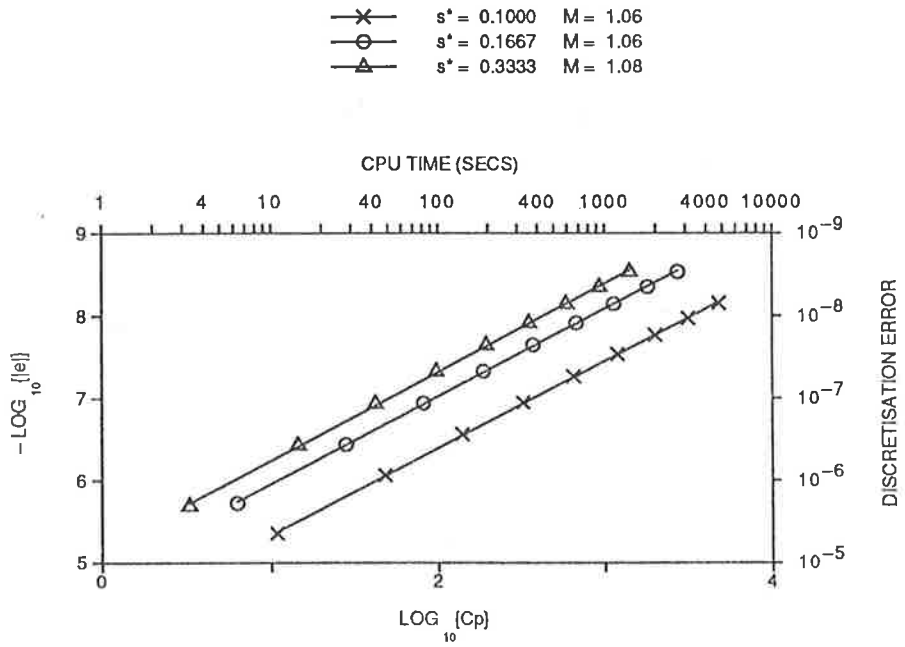


Figure 4.16: *Error vs CPU time graph for the (1,21) method (4.2.30), using a (9,9) equation at the boundaries*

the boundaries of the solution region, with the added complication that when used in a corner of the region there is one point of the stencil that is outside the region in *both* spatial directions, rather than just one as has been the case with the methods discussed so far. This adds to the difficulties of implementing these methods, as well as introducing an additional source of errors, unless the (9,9) implicit method is used, in which case these extra problems never arise.

The finite difference equation that corresponds to this differencing is

$$\begin{aligned}
12\tau^n_{j,k} &= \{\omega s_x + \sigma s_y\}(\tau^n_{j-2,k-2} + \tau^n_{j-2,k+2} + \tau^n_{j+2,k-2} + \tau^n_{j+2,k+2}) \\
&+ \{\pi s_x - 4(3\theta + 4\sigma)s_y\}(\tau^n_{j-2,k-1} + \tau^n_{j-2,k+1} + \tau^n_{j+2,k-1} + \tau^n_{j+2,k+1}) \\
&+ \{\lambda s_y - 4(3\varphi + 4\omega)s_x\}(\tau^n_{j-1,k-2} + \tau^n_{j-1,k+2} + \tau^n_{j+1,k-2} + \tau^n_{j+1,k+2}) \\
&+ \{(1 - 2\varphi - 2\omega - 2\gamma - 2\pi - \chi)s_x + 6(4\theta + 5\sigma)s_y\}(\tau^n_{j-2,k} + \tau^n_{j+2,k}) \\
&+ \{6(4\varphi + 5\omega)s_x + (1 - 2\theta - 2\sigma - 2\epsilon - 2\lambda - \eta)s_y\}(\tau^n_{j,k-2} + \tau^n_{j,k+2}) \\
&- 4\{(3\gamma + 4\pi)s_x + (3\epsilon + 4\lambda)s_y\}(\tau^n_{j-1,k-1} + \tau^n_{j-1,k+1} + \tau^n_{j+1,k-1} + \tau^n_{j+1,k+1}) \\
&+ \{4(\chi + 8\varphi + 8\omega + 8\gamma + 8\pi - 4)s_x + 6(4\epsilon + 5\lambda)s_y\}(\tau^n_{j-1,k} + \tau^n_{j+1,k}) \\
&+ \{6(4\gamma + 5\pi)s_x + 4(\eta + 8\theta + 8\sigma + 8\epsilon + 8\lambda - 4)s_y\}(\tau^n_{j,k-1} + \tau^n_{j,k+1}) \\
&- 6\{(\chi + 10\varphi + 10\omega + 10\gamma + 10\pi - 5)s_x \\
&\quad +(\eta + 10\theta + 10\sigma + 10\epsilon + 10\lambda - 5)s_y + 2\}\tau^n_{j,k}. \qquad (4.2.32)
\end{aligned}$$

The modified equivalent equation corresponding to equation (4.2.32) can be written in the general form (4.1.2) with leading error terms

$$\begin{aligned}
\Gamma_{4,0} &= 6s_x - \chi - 2\gamma - 2\varphi \\
\Gamma_{4,2} &= s_x s_y - (4\omega + 4\varphi + \pi + \gamma)s_x - (4\sigma + 4\theta + \lambda + \epsilon)s_y \\
\Gamma_{4,4} &= 6s_y - \eta - 2\epsilon - 2\theta. \qquad (4.2.33)
\end{aligned}$$

These error coefficients can be made zero by the choice of

$$\begin{aligned}
\chi &= 6s_x - 2\gamma - 2\varphi \\
\pi &= s_x - (4\omega + 4\varphi + \gamma) - (4\sigma + 4\theta + \lambda + \epsilon)(s_y/s_x) \\
\eta &= 6s_y - 2\epsilon - 2\theta \qquad (4.2.34)
\end{aligned}$$

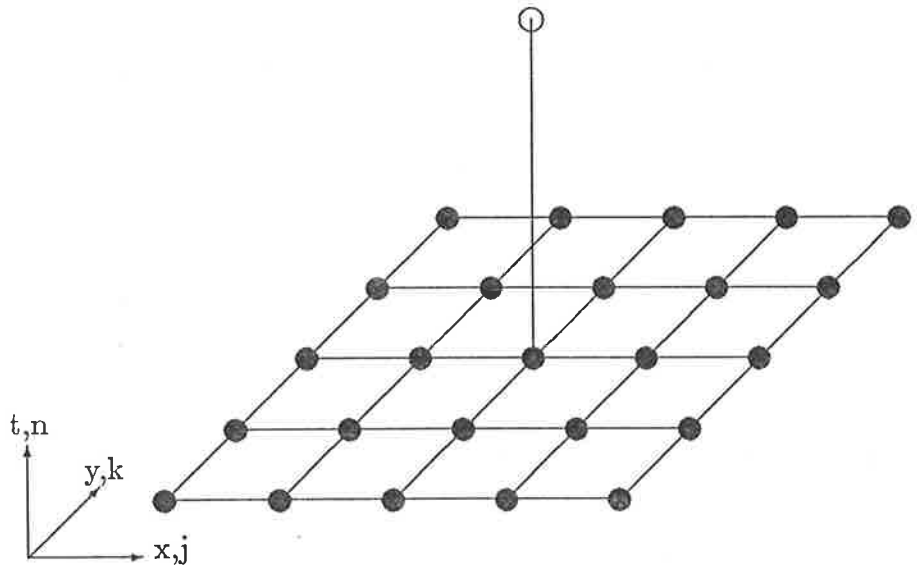


Figure 4.17: *Computational stencil for the (1,25) method*

which leads to a fourth-order finite difference equation. As was done for the (1,21) method however, the equation for π is split into two equations, in order to keep the symmetry of the difference equation. This is achieved by

$$\begin{aligned} \lambda &= s_x/2 - 4\sigma - 4\theta - \epsilon, \\ \pi &= s_y/2 - 4\omega - 4\varphi - \gamma. \end{aligned} \tag{4.2.35}$$

If the modified equivalent equation corresponding to this fourth-order equation is examined however, it is found that

$$\Gamma_{6,0} = 4 - 30s_x + 60{s_x}^2 \tag{4.2.36}$$

which contains no weights, and so the method cannot be made sixth-order by any choice of weights. It can be seen that this will happen no matter what weights we choose to make (4.2.32) fourth-order, as for this equation

$$\Gamma_{6,0} = 4 - 5(\chi + 2\gamma + 2\varphi) + 30(\chi + 2\gamma + 2\varphi)s_x - 120{s_x}^2, \tag{4.2.37}$$

and to eliminate the term $\Gamma_{4,0}$ we must set

$$\chi + 2\gamma + 2\varphi = 6s_x. \tag{4.2.38}$$

Substituting this into equation (4.2.37) gives equation (4.2.36) unconditionally, so that any choice of weights that makes equation (4.2.32) fourth-order makes eliminating further weights to give a sixth-order method impossible.

Thus this equation cannot be made any more accurate than the (1,13) and (1,21) equations discussed earlier and is significantly more complicated, due to the additional weights. These extra weights also mean that the von Neumann stability of the equation cannot be usefully examined numerically with the current programs. Since, based on previous experience, the method is unlikely to be significantly more stable than previous methods (the best likely result being $s_x \leq 2/3$, $s_y \leq 2/3$), the possible small saving in CPU time is not worth the added complication of this equation.

4.2.6 Summary

Of all the two-level explicit methods, the best fourth-order equations found were the (1,13) and (1,21) equations. The (1,21) equation has the advantage of a somewhat bigger von Neumann stability range than the (1,13) equation, but uses more grid points and so is generally more complicated to implement next to a boundary. Note that this disadvantage may be overcome by using the (9,9) implicit equation to find the values at grid points next to the boundary, with very little if any CPU time penalty compared to other methods, so this is the preferred technique of handling the boundary problems.

It is not practical to extend the computational stencil any further spatially, due to the increasing problems of points outside the region when the stencil is used near a boundary. Given this, attention is now shifted towards three-level explicit methods, where more grid points, and hence more weights, can be introduced into the computational stencil without extending it spatially.

4.3 Three-Level Explicit Methods

Three level methods are worth considering for a number of reasons. The main one is that more grid points can be included in the computational stencil while still keeping the stencil compact spatially, thus avoiding (or at least minimising) the problems discussed above that occur near the boundaries of the solution domain.

This also leads to an increase in the number of weights that can be used, both because there are now two time levels on which to difference the spatial derivatives, and also because backward-time differencing can be used to weight the time derivative as well, which was not possible for the two-level methods. It should be kept in mind, however, that three-level methods were, in general, less stable than two-level methods for the one-dimensional problem, and this may also be the case here. If this does carry over into the two-dimensional problem, then a three-level equation would need to be at least sixth-order accurate to make up for a reduced stability range.

4.3.1 (1,5,5) Weighted Explicit Method

The simplest possible three-level method is based on the (1,5) two-level method described in Section 4.2.1 above, with an extra 5 grid points included at the $(n-1)^{\text{th}}$ time level, as shown in Figure 4.18.

This stencil allows for the inclusion of five weights in centred fashion, although, judging by past experience, the number of points in the stencil is likely to be insufficient to obtain a very accurate method. The differencing used to include these weights is

$$\begin{aligned}
\left.\frac{\partial \tau}{\partial t}\right|_{j,k}^{n} &\approx \gamma \times \{[\text{ BT at } (j-1,k,n)] + [\text{ BT at } (j+1,k,n)]\} \\
&+ \lambda \times \{[\text{ BT at } (j,k-1,n)] + [\text{ BT at } (j,k+1,n)]\} \\
&+ \omega \times [\text{ CT at } (j,k,n)] \\
&+ (1-2\gamma-2\lambda-\omega) \times [\text{ FT at } (j,k,n)], \\
\left.\frac{\partial^2 \tau}{\partial x^2}\right|_{j,k}^{n} &\approx \varphi \times [\text{ CS3 at } (j,k,n)] + (1-\varphi) \times [\text{ CS3 at } (j,k,n-1)],
\end{aligned}$$

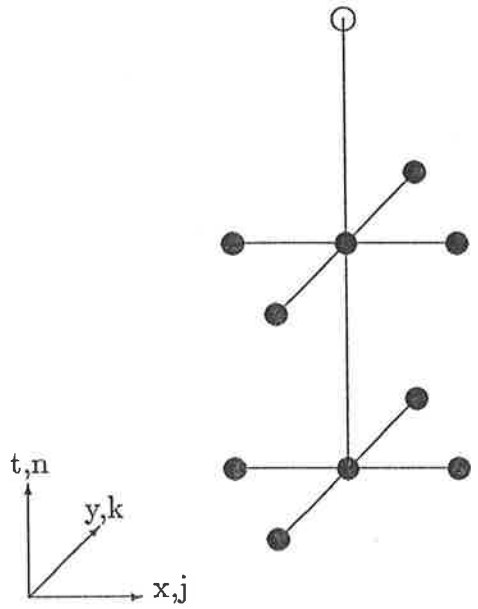


Figure 4.18: *Computational stencil for the (1,5,5) method*

$$\left.\frac{\partial^2\tau}{\partial y^2}\right|_{j,k}^{n} \approx \theta \times [\text{ CS3 at } (j,k,n)] + (1-\theta) \times [\text{ CS3 at } (j,k,n-1)]. \quad (4.3.1)$$

It can be seen that this differencing is not centred in time. It is however still centred in space, which is in fact sufficient for the resulting finite-difference equation to possess the desirable features of centred equations, namely that the odd-order error terms are all zero. The finite-difference equation that results from the differencing (4.3.1) is

$$\begin{aligned}
\{\omega + 4\gamma + 4\lambda - 1\}\tau_{j,k}^{n+1} &= 2\{\gamma - \varphi s_x\}(\tau_{j-1,k}^{n} + \tau_{j+1,k}^{n}) \\
&+ 2\{\lambda - \theta s_y\}(\tau_{j,k-1}^{n} + \tau_{j,k+1}^{n}) \\
&+ 2\{2\gamma + 2\lambda + \omega + 2\varphi s_x + 2\theta s_y - 1\}\tau_{j,k}^{n} \\
&+ 2\{(\varphi - 1)s_x - \gamma\}(\tau_{j-1,k}^{n-1} + \tau_{j+1,k}^{n-1}) \\
&+ 2\{(\theta - 1)s_x - \lambda\}(\tau_{j,k-1}^{n-1} + \tau_{j,k+1}^{n-1}) \\
&+ \{4(1-\varphi)s_x + 4(1-\theta)s_y - 1\}\tau_{j,k}^{n-1} \qquad (4.3.2)
\end{aligned}$$

This equation gives rise to a modified equivalent equation which has leading error

terms containing the factors

$$\begin{aligned}\Gamma_{4,0} &= 12\gamma - 1 + 6(3 - 2\varphi - 4\gamma - 4\lambda - \omega)s_x \\ \Gamma_{4,2} &= \lambda s_x + \gamma s_y + (3 - \theta - 4\gamma - 4\lambda - \omega - \varphi)s_x s_y \\ \Gamma_{4,4} &= 12\lambda - 1 + 6(3 - 2\theta - 4\gamma - 4\lambda - \omega)s_y. \end{aligned} \tag{4.3.3}$$

There are no values of the weights that eliminate all three of these errors simultaneously for general values of s_x and s_y. This can be verified by eliminating the terms $\Gamma_{4,0}$ and $\Gamma_{4,4}$, which is done by setting

$$\begin{aligned}\varphi &= \frac{12\gamma - 1 + s_x(18 - 6\omega - 24\lambda - 24\gamma)}{12s_x} \\ \theta &= \frac{12\lambda - 1 + s_y(18 - 6\omega - 24\lambda - 24\gamma)}{12s_y}. \end{aligned} \tag{4.3.4}$$

If this substitution is made and the modified equivalent equation of the resulting equation is examined, it is found to contain factors in the leading error terms of

$$\begin{aligned}\Gamma_{4,0} &= 0 \\ \Gamma_{4,2} &= \frac{s_x + s_y}{12} \\ \Gamma_{4,4} &= 0. \end{aligned} \tag{4.3.5}$$

The absence of any of the remaining weights in these expressions means that $\Gamma_{4,2}$ cannot be eliminated, for general s_x and s_y, so the method is only second-order accurate. Thus this method is, as expected, significantly less accurate than the fourth-order methods presented in previous sections.

4.3.2 (1,9,9) Weighted Explicit Method

If the (1,5,5) computational stencil used above is extended to a (1,9,9) stencil, eight more grid points are involved in the stencil, which should allow a more accurate method to be developed. This is done by adding extra weights into the differencings for both the space and time derivatives, which should allow at least all the second-order error terms to be eliminated, to produce a fourth-order equation.

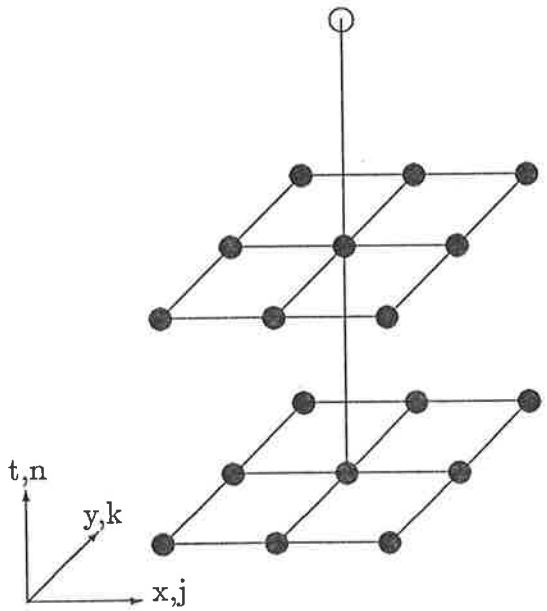


Figure 4.19: *Computational stencil for the (1,9,9) method*

The differencing used is basically the same as that used for the (1,5,5) stencil, with some extra differencings introduced to incorporate the extra points added to the computational stencil. To keep the stencil spatially centred, the same weight must be used for all four of the additional backward time derivatives introduced, but two additional weights can be added in the differencing for each of the space derivative terms, giving a total of ten weights for the method. The weighting used is thus

$$\begin{aligned}
\left.\frac{\partial \tau}{\partial t}\right|_{j,k}^{n} &\approx \gamma \times \{[\text{ BT at } (j-1,k,n)] + [\text{ BT at } (j+1,k,n)]\} \\
&+ \lambda \times \{[\text{ BT at } (j,k-1,n)] + [\text{ BT at } (j,k+1,n)]\} \\
&+ \chi \times \{[\text{ BT at } (j-1,k-1,n)] + [\text{ BT at } (j-1,k+1,n)] \\
&\qquad + [\text{ BT at } (j+1,k-1,n)] + [\text{ BT at } (j+1,k+1,n)]\} \\
&+ \omega \times [\text{ FT at } (j,k,n)] + (1-2(\gamma+\lambda+2\chi)-\omega) \times [\text{ CT at } (j,k,n)], \\
\left.\frac{\partial^2 \tau}{\partial x^2}\right|_{j,k}^{n} &\approx \sigma \times \{[\text{ CS3 at } (j,k-1,n)] + [\text{ CS3 at } (j,k+1,n)]\} \\
&+ \varphi \times [\text{ CS3 at } (j,k,n)] \\
&+ \pi \times \{[\text{ CS3 at } (j,k-1,n-1)] + [\text{ CS3 at } (j,k+1,n-1)]\} \\
&+ (1-2\sigma-2\pi-\varphi) \times [\text{ CS3 at } (j,k,n-1)],
\end{aligned}$$

$$\begin{aligned}\left.\frac{\partial^2\tau}{\partial y^2}\right|_{j,k}^{n} &\approx \nu\times\{[\text{ CS3 at }(j-1,k,n)]+[\text{ CS3 at }(j+1,k,n)]\}\\ &+ \theta\times[\text{ CS3 at }(j,k,n)]\\ &+ \beta\times\{[\text{ CS3 at }(j-1,k,n-1)]+[\text{ CS3 at }(j+1,k,n-1)]\}\\ &+ (1-2\nu-2\beta-\theta)\times[\text{ CS3 at }(j,k,n-1)].\end{aligned}\tag{4.3.6}$$

This differencing leads to the finite-difference equation

$$\begin{aligned}\{\omega+4\gamma+4\lambda+8\chi-2\}\tau_{j,k}^{n+1} &= 2\{\chi-\sigma s_x-\nu s_y\}\\ &\quad(\tau_{j-1,k-1}^{n}+\tau_{j-1,k+1}^{n}+\tau_{j+1,k-1}^{n}+\tau_{j+1,k+1}^{n})\\ &+ 2\{\gamma-\varphi s_x+2\nu s_y\}(\tau_{j-1,k}^{n}+\tau_{j+1,k}^{n})\\ &+ 2\{\lambda+2\sigma s_x-\theta s_y\}(\tau_{j,k-1}^{n}+\tau_{j,k+1}^{n})\\ &+ 2\{\omega+2\gamma+2\lambda+4\chi+2\varphi s_x+2\theta s_y-1\}\tau_{j,k}^{n}\\ &- 2\{\chi+\pi s_x+\beta s_y\}\\ &\quad(\tau_{j-1,k-1}^{n-1}+\tau_{j-1,k+1}^{n-1}+\tau_{j+1,k-1}^{n-1}+\tau_{j+1,k+1}^{n-1})\\ &+ 2\{(2\sigma+2\pi+\varphi-1)s_x+2\beta s_y-\gamma\}(\tau_{j-1,k}^{n-1}+\tau_{j+1,k}^{n-1})\\ &+ 2\{2\pi s_x+(2\nu+2\beta+\theta-1)s_y-\lambda\}(\tau_{j,k-1}^{n-1}+\tau_{j,k+1}^{n-1})\\ &+ \{4(1-\varphi-2\sigma-2\pi)s_x\\ &\quad+4(1-\theta-2\nu-2\beta)s_y-\omega\}\tau_{j,k}^{n-1},\end{aligned}\tag{4.3.7}$$

which can be shown to have a modified equivalent equation that is in the general form (4.1.2) with leading error terms involving the factors

$$\begin{aligned}\Gamma_{4,0} &= -1+24\chi+12\gamma-6(4\sigma+2\varphi+8\chi+4\gamma+4\lambda+\omega-3)s_x\\ \Gamma_{4,2} &= (2\chi+\lambda-\sigma-\pi)s_x+(2\chi+\gamma-\nu-\beta)s_y\\ &+ (3-2\sigma-\varphi-2\nu-\theta-\omega-4\gamma-4\lambda-8\chi)s_xs_y\\ \Gamma_{4,4} &= -1+24\chi+12\lambda-6(4\nu+2\theta+8\chi+4\gamma+4\lambda+\omega-3)s_y.\end{aligned}\tag{4.3.8}$$

These second-order error terms can be removed by a suitable choice of values for some of the weights. In order to keep the equation spatially symmetrical, four weights are

used instead of the minimum three, the choice being

$$\begin{aligned}
\pi &= (3-2\sigma-\varphi-2\nu-\theta-\omega-4\gamma-4\lambda-8\chi)(s_y/2)-\sigma+\lambda+2\chi,\\
\beta &= (3-2\sigma-\varphi-2\nu-\theta-\omega-4\gamma-4\lambda-8\chi)(s_x/2)-\nu+\gamma+2\chi,\\
\varphi &= \{-1+24\chi+12\gamma-6(4\sigma+8\chi+4\gamma+4\lambda+\omega-3)s_x\}/(12s_x),\\
\theta &= \{-1+24\chi+12\lambda-6(4\nu+8\chi+4\gamma+4\lambda+\omega-3)s_y\}/(12s_y), \qquad (4.3.9)
\end{aligned}$$

where the values given for φ and θ must also be substituted in the expressions for π and β. This substitution leads to a fourth-order finite-difference equation in which several of the weights appear only in common sub-expressions. To simplify the form of the equation, we can, without loss of generality, replace each of these sub-expressions with a new "weight", which can then be treated exactly like any other weight. The substitution made is

$$\begin{aligned}
\mu &= \omega+4(\gamma+\lambda+2\chi)\\
\epsilon &= \chi-\sigma s_x-\nu s_y, \qquad (4.3.10)
\end{aligned}$$

which yields the spatially centred fourth-order finite-difference equation

$$\begin{aligned}
6\{\mu-2\}\tau_{j,k}^{n+1} &= 12\epsilon(\tau_{j-1,k-1}^n+\tau_{j-1,k+1}^n+\tau_{j+1,k-1}^n+\tau_{j+1,k+1}^n)\\
&+ \{1-18s_x-24\epsilon+6\mu s_x\}(\tau_{j-1,k}^n+\tau_{j+1,k}^n)\\
&+ \{1-18s_y-24\epsilon+6\mu s_x\}(\tau_{j,k-1}^n+\tau_{j,k+1}^n)\\
&+ \{36(s_x+s_y)-16+12\mu(1-s_x-s_y)+48\epsilon\}\tau_{j,k}^n\\
&- \{s_x+s_y+12\epsilon\}(\tau_{j-1,k-1}^{n-1}+\tau_{j-1,k+1}^{n-1}+\tau_{j+1,k-1}^{n-1}+\tau_{j+1,k+1}^{n-1})\\
&+ \{8s_x+2s_y-1-6\mu s_x+24\epsilon\}(\tau_{j-1,k}^{n-1}+\tau_{j+1,k}^{n-1})\\
&+ \{2s_x+8s_y-1-6\mu s_y+24\epsilon\}(\tau_{j,k-1}^{n-1}+\tau_{j,k+1}^{n-1})\\
&+ \{4-16(s_x+s_y)-6\mu(1-2s_x-2s_y)-48\epsilon\}\tau_{j,k}^{n-1}, \qquad (4.3.11)
\end{aligned}$$

with the proviso that $\mu \neq 2$ being required so as to retain the explicit nature of the finite-difference equation.

The fourth-order error terms corresponding to this finite-difference equation then contain the factors

$$
\begin{aligned}
\Gamma_{6,0} &= (2/3)\{1 - 20s_x + 100{s_x}^2 + 10\mu s_x(1 - 6s_x)\} \\
\Gamma_{6,2} &= (1/12)\{s_x(6s_x - 1 - 144\epsilon) + 6s_y(\mu - 1) + 36s_x s_y(5 - 3\mu)\} \\
\Gamma_{6,4} &= (1/12)\{s_y(6s_y - 1 - 144\epsilon) + 6s_x(\mu - 1) + 36s_x s_y(5 - 3\mu)\} \\
\Gamma_{6,6} &= (2/3)\{1 - 20s_y + 100{s_y}^2 + 10\mu s_y(1 - 6s_y)\}. \qquad (4.3.12)
\end{aligned}
$$

It can be seen from coefficients (4.3.12) that the (1,9,9) equation cannot be made sixth-order for general s_x and s_y, since to force $\Gamma_{6,0} = \Gamma_{6,2} = 0$, we require

$$
\begin{aligned}
\mu &= \frac{20s_x(1 - 5s_x) - 1}{10s_x(1 - 6s_x)} \\
\epsilon &= \frac{6s_x - 1 + 6s_y(\mu - 1) + 36s_x s_y(5 - 3\mu)}{144}, \qquad (4.3.13)
\end{aligned}
$$

and this substitution removes all the weights from the expressions for $\Gamma_{6,4}$ and $\Gamma_{6,6}$, without, in general, making these zero. Note, however, that in the special case $s_x = s_y = s^*$, the substitution (4.3.13) *does* also make $\Gamma_{6,4} = \Gamma_{6,6} = 0$, so a sixth-order equation in possible in this case.

If the substitution (4.3.13) is made, the "optimal" (in the sense that as many of the low order error terms as possible have been eliminated) finite-difference equation for the (1,9,9) computational stencil is found to be

$$
\begin{aligned}
-36\{1 - 20{s_x}^2\}\tau^{n+1}_{j,k} &= \{2s_x(7 - 300s_x + 1260{s_x}^2) - 6s_y(1 - 28s_x + 100{s_x}^2)\} \\
&\quad (\tau^n_{j-1,k} + \tau^n_{j+1,k}) \\
&+ \{5s_x(1 - 12s_x + 36{s_x}^2) + 3s_y(1 - 28s_x + 100{s_x}^2)\} \\
&\quad (\tau^n_{j-1,k-1} + \tau^n_{j-1,k+1} + \tau^n_{j+1,k-1} + \tau^n_{j+1,k+1}) \\
&+ \{10s_x(5 - 24s_x - 36{s_x}^2) - 6s_y(7 + 32s_x - 380{s_x}^2)\} \\
&\quad (\tau^n_{j,k-1} + \tau^n_{j,k+1}) \\
&+ \{4s_x(143 - 240s_x - 1260{s_x}^2) \\
&\quad + 12s_y(7 + 32s_x - 380{s_x}^2) - 72\}\tau^n_{j,k} \\
&+ \{-5s_x(1 - 36{s_x}^2) - 3s_y(1 - 8s_x - 20{s_x}^2)\} \\
&\quad (\tau^{n-1}_{j-1,k-1} + \tau^{n-1}_{j-1,k+1} + \tau^{n-1}_{j+1,k-1} + \tau^{n-1}_{j+1,k+1})
\end{aligned}
$$

$$
\begin{aligned}
&+ \{-2s_x(7-540{s_x}^2)+6s_y(1-8s_x-20{s_x}^2)\} \\
&\quad (\tau^{n-1}_{j-1,k}+\tau^{n-1}_{j+1,k}) \\
&+ \{-10s_x(5-36s_x+36{s_x}^2)+6s_y(7-68s_x+220{s_x}^2)\} \\
&\quad (\tau^{n-1}_{j,k-1}+\tau^{n-1}_{j,k+1}) \\
&+ \{36-4s_x(143-720s_x+540{s_x}^2) \\
&\quad -12s_y(7-68s_x+220{s_x}^2)\}\tau^{n-1}_{j,k} \quad , s_x \neq \frac{1}{\sqrt{20}}. \qquad (4.3.14)
\end{aligned}
$$

The condition $s_x \neq 1/\sqrt{20}$ corresponds to the condition $\mu \neq 2$ from equation (4.3.11). Equation (4.3.14) is still symmetric with respect to each spatial dimension individually, but is not totally symmetric, due to the asymmetric nature of the substitution (4.3.13). It should be noted that there is no symmetric substitution which removes any of the error terms (4.3.12), except in the case where $s_x = s_y = s^*$, due to the form of the coefficients. This asymmetry does not, however, introduce any extra error terms into the modified equivalent equation. The factors from the coefficients of the leading error terms in the modified equation corresponding to (4.3.14) are given by

$$
\begin{aligned}
\Gamma_{6,0} = \Gamma_{6,2} &= 0 \\
\Gamma_{6,4} &= \frac{s_y(20{s_x}^2-1)(s_x-s_y)}{20s_x(1-6s_x)} \\
\Gamma_{6,6} &= \frac{(1-6s_x-6s_y+20s_xs_y)(s_x-s_y)}{10s_x(1-6s_x)}. \qquad (4.3.15)
\end{aligned}
$$

From this, it is clear that the equation is sixth-order accurate in the case $s_x = s_y$, since both of the remaining fourth-order error terms contain $(s_x - s_y)$ as a factor.

If the von Neumann stability of equation (4.3.14) is examined, it is found to be stable for only the small region *approximately* defined by

$$
\begin{aligned}
s_x &\leq 1/6 \\
s_y &\leq 1/3. \qquad (4.3.16)
\end{aligned}
$$

For some values of s_x the upper limit for s_y is slightly more than 1/3, in some cases as high as 0.4. In the case $s_x = s_y = s^*$, for which the equation is sixth-order, the stability limit is $s^* = 1/6$, which is far too restrictive for the method to be of practical

use, especially since it is only fourth-order accurate. Given this stability range, the (1,13) equation (4.2.18) is a more practical method, since it has the same fourth-order accuracy and twice the stability range in the x-direction.

Notice that there seems to be some correspondence between this method and the sixth-order (1,3,3) equation (2.5.9) for the one-dimensional case, in that both are three-level explicit equations, both use all available points in a spatially three point wide stencil and both restricted to $s^* \leq 1/6$ for von Neumann stability. Note, however, that the one-dimensional (1,3,3) equation was sixth-order accurate, whereas the best that can be done here is fourth-order accuracy.

What can be done to attempt to overcome the small stability range is to look at the von Neumann stability of the fourth-order weighted equation (4.3.11), to see if this is suitably stable for some choice of the weights. To be of practical use, the fourth-order equation would have to be stable over a range which is comparable to that obtained for the fourth-order methods discussed earlier. However, a numerical stability analysis of equation (4.3.11) shows that there are no values for the weights μ and ϵ that produce a von Neumann stable scheme for $s_x = s_y = s^* = 1/2$, or even $s^* = 1/3$. Since any lesser range would be overly restrictive, given the ranges of earlier methods, we must conclude that the (1,9,9) stencil cannot produce a practically useful equation.

Experience with unsuccessfully trying to develop several spatially wide three-level methods for the one-dimensional case, with stencils such as (1,5,5), shows that such equations generally have very restrictive von Neumann stability ranges, even when the desired accuracy could be obtained. Given this, and the analogy between the one-dimensional (1,3,3) equation and the two-dimensional (1,9,9) equation, it is expected that attempting to find more accurate methods by extending the three-level stencils spatially will be unsuccessful.

4.3.3 Other Three-Level Equations

Given the success of the (1,5,1) stencil in the one-dimensional case it is worth trying to develop an analogous equation, which uses fewer points at time level $(n-1)$ than

at time level n, for the two-dimensional problem.

As a first attempt, a (1,5,1) stencil can be considered. This stencil only allows one weight, on the time derivative, and the resulting equation cannot be made fourth-order by any choice of this weight. Thus this equation is only second-order accurate for all s_x and s_y, and so is not worth considering.

An equation based on a (1,9,1) computational stencil can include three weights, one for each space derivative and one for the time derivative, which may possibly be used to eliminate all three second-order error terms and make a fourth-order method. If this is tried, however, it is found that $\Gamma_{4,0}$ and $\Gamma_{4,4}$ depend only on the time weight, and both are simultaneously zero only in the case $s_x = s_y = s^*$. Thus this equation too is only second-order accurate in the general case.

Closer to a direct analogy of the one-dimensional (1,5,1) stencil is a spatially centred (1,13,5) stencil, which can incorporate nine weights. Despite the large number of weights, it is found that the resulting equation cannot be made sixth-order accurate, and the fourth-order version of the equation has no advantages over the earlier explicit methods in terms of accuracy or von Neumann stability, so this equation is not considered in any more detail.

The last possible approach is to use a spatially centred (1,13,9) stencil, which can incorporate twelve weights. The resulting finite-difference equation can be made sixth-order by suitable choices for the values of the weights. The problems arise from the fact that the sixth-order equation has coefficients containing numbers which are too large to allow development and analysis of this equation on the VAX computer used for this work. Further work may produce a good sixth-order equation from this stencil, but since the analysis cannot be carried out on the computers available this has not been done here.

4.3.4 Summary

It can be seen from the above that three-level methods, which provided the very accurate and von Neumann stable (1,5,1) equation to solve the one-dimensional problem, are not as practically useful for the two-dimensional problem. There is a similar tendency to that seen in the one-dimensional case for there to be a severe reduction in von Neumann stability for spatially wide three-level equations. Also, the only real analogue of the sixth-order (1,5,1) equation for the one-dimensional case contains coefficients which are sufficiently large to prevent analysis on the available computing facilities.

Thus overall, while three-level methods may appear to offer more accurate solutions from compact computational stencils, in practice no more accuracy is obtained than from the two-level methods.

4.4 Two-Level Implicit Methods

Another way of trying to incorporate more grid points into a stencil, and hence to get more accuracy, is to use implicit methods. By analogy with the one-dimensional case, it is expected that such methods would be very much more stable than the explicit methods discussed in Section 4.2 above, and, for a given spatial extent of a computational stencil, more accurate. Balanced against this is the fact that at each time level, a set of linear algebraic equations must be solved to give the values at the new time level, and in two dimensions this becomes an extremely large time-consuming problem, due to the large number of unknown values at the new time level.

It may, however, be expected that implicit methods could be made more accurate than the explicit ones discussed above, as there is more possibility of introducing weights into the equations. This is due to the extra points at the new time level, which allow forward-time differences to be used at several space positions, rather than just the (j, k, n) grid point, as was the case for two-level explicit methods.

4.4.1 (5,5) Implicit Method

In order to check whether the general results for one-dimensional finite-difference methods will carry over into the two-dimensional case, a very simple (5,5) method is considered. While this is likely to have insufficient grid points in the stencil to be even fourth-order accurate, it can still serve to investigate such properties as the von Neumann stability of such methods.

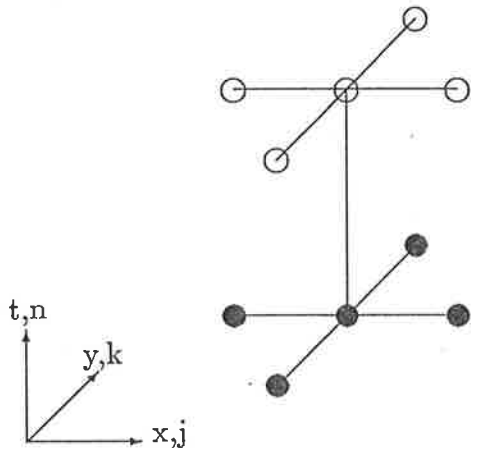


Figure 4.20: *Computational stencil for the (5,5) method*

Using such a computational stencil, illustrated in Figure 4.20, the only way of weighting the spatial derivative terms is by splitting the differencing between time levels, namely

$$\begin{aligned}
\left.\frac{\partial^2\tau}{\partial x^2}\right|_{j,k}^{n} &\approx \lambda \times [\text{ CS at } (j,k,n+1)] + (1-\lambda) \times [\text{ CS at } (j,k,n)], \\
\left.\frac{\partial^2\tau}{\partial y^2}\right|_{j,k}^{n} &\approx \theta \times [\text{ CS at } (j,k,n+1)] + (1-\theta) \times [\text{ CS at } (j,k,n)]. \qquad (4.4.1)
\end{aligned}$$

The time derivative is also limited in terms of potential ways to introduce weights, since the stencil is to be kept centred. One way of differencing is

$$\begin{aligned}
\left.\frac{\partial\tau}{\partial t}\right|_{j,k}^{n} &\approx \varphi \times \{[\text{ FT at } (j-1,k,n)] + [\text{ FT at } (j+1,k,n)]\} \\
&+ \gamma \times \{[\text{ FT at } (j,k-1,n)] + [\text{ FT at } (j,k+1,n)]\} \\
&+ (1-2\varphi-2\gamma) \times \{[\text{ FT at } (j,k,n)]\}. \qquad (4.4.2)
\end{aligned}$$

This differencing leads to the finite-difference equation

$$\begin{aligned} & \{\varphi - \lambda s_x\}(\tau_{j-1,k}^{n+1} + \tau_{j+1,k}^{n+1}) + \{\lambda - \theta s_y\}(\tau_{j,k-1}^{n+1} + \tau_{j,k+1}^{n+1}) \\ + \; & \{1 - 2\varphi - 2\gamma + 2\lambda s_x + 2\theta s_y\}\tau_{j,k}^{n+1} \\ = \; & \{\lambda s_x - \varphi - s_x\}(\tau_{j-1,k}^{n} + \tau_{j+1,k}^{n}) + \{\theta s_y - \gamma - s_y\}(\tau_{j,k-1}^{n} + \tau_{j,k+1}^{n}) \\ + \; & \{2\varphi + 2\gamma + 2s_x(1-\lambda) + 2s_y(1-\theta)\}\tau_{j,k}^{n} \end{aligned} \tag{4.4.3}$$

which can be shown to have a modified equivalent equation that has leading error terms that contain the factors

$$\begin{aligned} \Gamma_{4,0} &= (12\varphi + 6s_x(1-2\lambda) - 1) \\ \Gamma_{4,2} &= (\gamma s_x + s_x s_y(1-\theta-\lambda) + \varphi s_y) \\ \Gamma_{4,4} &= (12\gamma + 6s_y(1-2\theta) - 1). \end{aligned} \tag{4.4.4}$$

As expected, these error terms cannot be removed simultaneously to make the method fourth-order. If this is tried, then $\Gamma_{4,0} = \Gamma_{4,4} = 0$ leads to

$$\begin{aligned} \varphi &= \frac{1 + 6s_x(2\lambda - 1)}{12} \\ \gamma &= \frac{1 + 6s_y(2\theta - 1)}{12} \end{aligned} \tag{4.4.5}$$

and substituting these back into $\Gamma_{4,2}$ gives

$$\Gamma_{4,2} = \frac{s_x + s_y}{12}. \tag{4.4.6}$$

There are no weights left in this expression, and no specific values of s_x and s_y which make it zero, except in the case where $s_x = -s_y$, which means that $s_x = s_y = 0$, which is of no practical use as it forces the time steps to be of zero size. Thus this method cannot be made fourth-order, but is instead second-order for all values of s_x and $s_y > 0$.

The equation that results from substituting the values (4.4.5) into equation (4.4.3) is

$$\begin{aligned} & \{1 - 6s_x\}(\tau_{j-1,k}^{n+1} + \tau_{j+1,k}^{n+1}) + \{1 - 6s_y\}(\tau_{j,k-1}^{n+1} + \tau_{j,k+1}^{n+1}) \\ + \; & 4\{2 + 3s_x + 3s_y\}\tau_{j,k}^{n+1} \\ = \; & \{1 + 6s_x\}(\tau_{j-1,k}^{n} + \tau_{j+1,k}^{n}) + \{1 + 6s_y\}(\tau_{j,k-1}^{n} + \tau_{j,k+1}^{n}) \\ + \; & 4\{2 - 3s_x - 3s_y\}\tau_{j,k}^{n} \end{aligned} \tag{4.4.7}$$

which has no weights left in it. A theoretical von Neumann stability analysis of this method shows that it is unconditionally stable, and furthermore it can be shown that the method is also unconditionally solvable. This is obviously much better than any of the explicit methods. This is also consistent with the observation from the one-dimensional case that two-level implicit methods tend to be more stable than explicit ones. The major problem with such methods is the large system of linear algebraic equations that must be solved at each time step.

Running a numerical test for this equation produces results as shown in Figures 4.21 and 4.22. These results were generated using the (5,5) equation, with the sets of linear equations being solved at each time level by a banded equation solver, since this uses many less operations than a full Gauss elimination process (Reid, 1971). The main feature of these graphs is the enormous amount of CPU time required to generate the solutions, even for such small values as $J = K = 40$. Another run, that included the value $J = K = 50$ as well required over two days of CPU time to run. This level of CPU time usage is unacceptable for a method in most practical situations. Since the only real difference between this and the explicit methods discussed previously is the requirement to solve a system of linear equations at each time step, it can be concluded that implicit methods in general are impractical, since even with more efficient solution techniques, the size of the sets of equations will consume significant amounts of CPU time.

Given this problem with the amount of CPU time required to run implicit methods, no further work will be done on them, except to generate the fourth-order (9,9) equation which can be used earlier to overcome the boundary problems which arise from such explicit equations as the (1,13) equation.

4.4.2 (9,9) Implicit Method

To be useful as a boundary technique for explicit equations with spatially wide computational stencils, a (9,9) implicit equation must be at least fourth-order accurate.

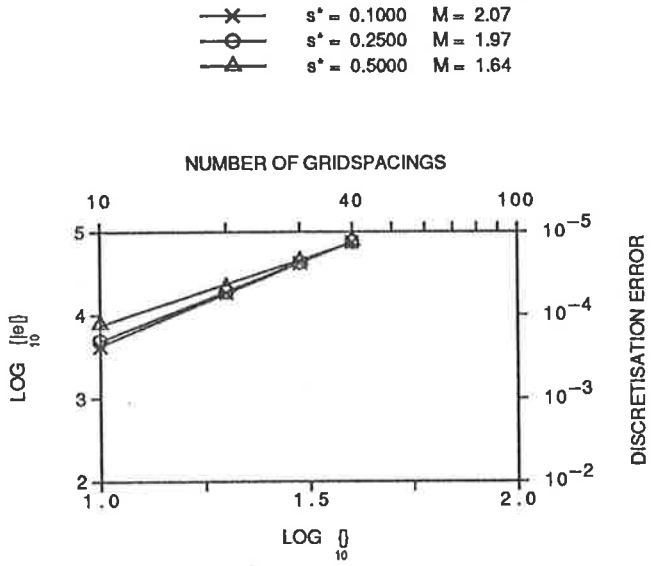


Figure 4.21: *Error vs grid spacing graph for the (5,5) implicit method (4.4.7)*

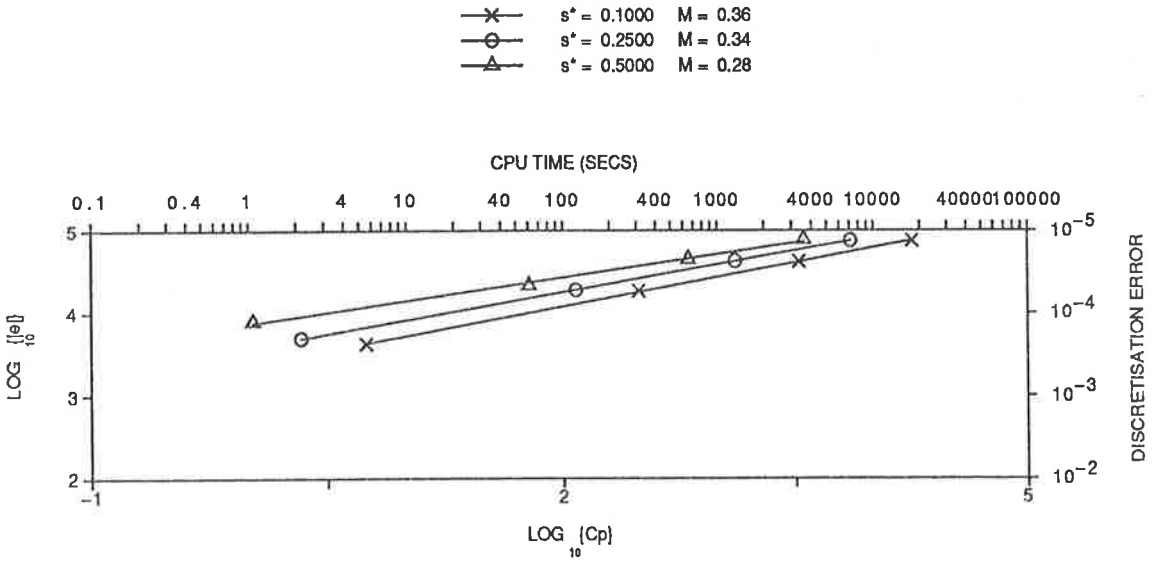


Figure 4.22: *Error vs CPU time graph for the (5,5) implicit method (4.4.7)*

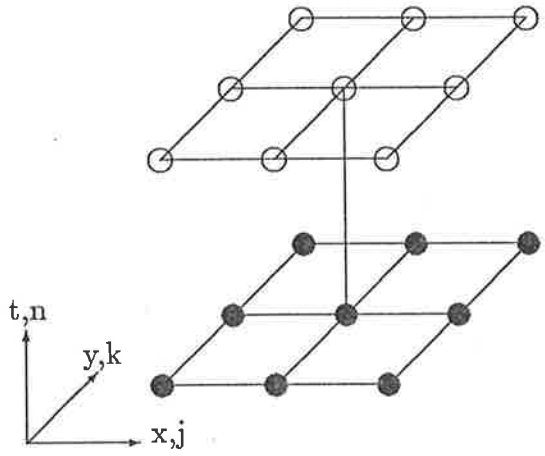


Figure 4.23: *Computational stencil for the (9,9) method*

Although as many as ten weights could be incorporated into a (9,9) computational stencil, most of these could not be usefully used, since we seek only a fourth-order equation, which requires only three or four weights to eliminate the second-order error terms. To ensure that the second-order errors *can* be eliminated, six weights will be used, distributed among the spatial derivatives as follows:

$$\begin{aligned}
\left.\frac{\partial^2\tau}{\partial x^2}\right|^n_{j,k} &\approx \eta \times \{[\text{ CS at } (j,k-1,n+1)] + [\text{ CS at } (j,k+1,n+1)]\} \\
&+ \epsilon \times [\text{ CS at } (j,k,n)] \\
&+ \lambda \times \{[\text{ CS at } (j,k-1,n)] + [\text{ CS at } (j,k+1,n)]\} \\
&+ (1-2\eta-2\lambda-\epsilon) \times [\text{ CS at } (j,k,n)], \\
\left.\frac{\partial^2\tau}{\partial y^2}\right|^n_{j,k} &\approx \chi \times \{[\text{ CS at } (j-1,k,n+1)] + [\text{ CS at } (j+1,k,n+1)]\} \\
&+ \pi \times [\text{ CS at } (j,k,n)] \\
&+ \theta \times \{[\text{ CS at } (j-1,k,n)] + [\text{ CS at } (j+1,k,n)]\} \\
&+ (1-2\chi-2\theta-\pi) \times [\text{ CS at } (j,k,n)]. \qquad (4.4.8)
\end{aligned}$$

Combined with this, the forward-time differencing is used for the temporal derivative to give the weighted finite-difference equation

$$
\begin{aligned}
& \{\eta s_x + \chi s_y\}(\tau^{n+1}_{j-1,k-1} + \tau^{n+1}_{j-1,k+1} + \tau^{n+1}_{j+1,k-1} + \tau^{n+1}_{j+1,k+1}) \\
+ \; & \{\epsilon s_x - 2\chi s_y\}(\tau^{n+1}_{j-1,k} + \tau^{n+1}_{j+1,k}) \\
+ \; & \{\pi s_y - 2\eta s_x\}(\tau^{n+1}_{j,k-1} + \tau^{n+1}_{j,k+1}) - \{1 + 2\epsilon s_x + 2\pi s_y\}\tau^{n+1}_{j,k} \\
= \; - \; & \{\lambda s_x + \theta s_y\}(\tau^{n}_{j-1,k-1} + \tau^{n}_{j-1,k+1} + \tau^{n}_{j+1,k-1} + \tau^{n}_{j+1,k+1}) \\
+ \; & \{(2\eta + 2\lambda + \epsilon - 1)s_x + 2\theta s_y\}(\tau^{n}_{j-1,k} + \tau^{n}_{j+1,k}) \\
+ \; & \{2\lambda s_x + (2\chi + 2\theta + \pi - 1)s_y\}(\tau^{n}_{j,k-1} + \tau^{n}_{j,k+1}) \\
+ \; & \{(2 - 4\eta - 4\lambda - 2\epsilon)s_x + (2 - 4\chi - 4\theta - 2\pi)s_y - 1\}\tau^{n}_{j,k}. \qquad (4.4.9)
\end{aligned}
$$

The modified equivalent equation corresponding to this equation can be written in the general form (4.1.2) with the leading errors terms

$$
\begin{aligned}
\Gamma_{4,0} &= 6(1 - 2\epsilon - 4\eta)s_x - 1, \\
\Gamma_{4,2} &= -(\lambda + \eta)s_x - (\theta + \chi)s_y + (1 - 2\chi - 2\eta - \pi - \epsilon)s_x s_y \\
\Gamma_{4,4} &= 6(1 - 2\pi - 4\chi)s_y - 1. \qquad (4.4.10)
\end{aligned}
$$

These leading errors can be forced to be zero, while still maintaining the symmetry of the finite-difference equation, by the choice of weights

$$
\begin{aligned}
\theta &= \frac{(1 - \pi - \epsilon - 2\chi - 2\eta)s_x}{2} - \chi \\
\lambda &= \frac{(1 - \pi - \epsilon - 2\chi - 2\eta)s_y}{2} - \eta \\
\pi &= \frac{1 - 4\chi}{2} - \frac{1}{12 s_y} \\
\epsilon &= \frac{1 - 4\eta}{2} - \frac{1}{12 s_x}. \qquad (4.4.11)
\end{aligned}
$$

Although this scheme cannot be made sixth-order accurate, since there are insufficient weights, it is useful to look at the coefficients in the leading error terms, after the above substitution is made. These terms can be written in the form (4.1.2) with

$$
\begin{aligned}
\Gamma_{6,0} &= 3/2 - 30{s_x}^2, \\
\Gamma_{6,2} &= \frac{6(1 - 24\eta){s_x}^2 + 6(1 - 24\chi)s_x s_y - s_x - 36{s_x}^2 s_y}{12}, \\
\Gamma_{6,4} &= \frac{6(1 - 24\chi){s_y}^2 + 6(1 - 24\eta)s_x s_y - s_y - 36{s_y}^2 s_x}{12}, \\
\Gamma_{6,6} &= 3/2 - 30{s_y}^2. \qquad (4.4.12)
\end{aligned}
$$

From this, it can be seen that the value

$$\eta = -\frac{1 - 6(s_x + s_y) + 166\chi s_y + 36 s_x s_y}{144 s_x} \tag{4.4.13}$$

removes the remaining two weights from the finite-difference equation, and also eliminates the cross-derivative error terms $\Gamma_{6,2}$ and $\Gamma_{6,4}$. The resulting fourth-order "optimal" finite-difference equation is

$$\begin{aligned}
& \{1 - 6s_x - 6s_y + 36 s_x s_y\}(\tau^{n+1}_{j-1,k-1} + \tau^{n+1}_{j-1,k+1} + \tau^{n+1}_{j+1,k-1} + \tau^{n+1}_{j+1,k+1}) \\
+ \; & \{10 - 60s_x + 12s_y - 72 s_x s_y\}(\tau^{n+1}_{j-1,k} + \tau^{n+1}_{j+1,k}) \\
+ \; & \{10 + 12s_x - 60s_y - 72 s_x s_y\}(\tau^{n+1}_{j,k-1} + \tau^{n+1}_{j,k+1}) \\
+ \; & \{100 + 120(s_x + s_y) + 144 s_x s_y\}\tau^{n+1}_{j,k} \\
= \; & \{1 - 6s_x - 6s_y - 36 s_x s_y\}(\tau^{n}_{j-1,k-1} + \tau^{n}_{j-1,k+1} + \tau^{n}_{j+1,k-1} + \tau^{n}_{j+1,k+1}) \\
+ \; & \{-10 - 60s_x + 12s_y + 72 s_x s_y\}(\tau^{n}_{j-1,k} + \tau^{n}_{j+1,k}) \\
+ \; & \{-10 + 12s_x - 60s_y + 72 s_x s_y\}(\tau^{n}_{j,k-1} + \tau^{n}_{j,k+1}) \\
+ \; & \{-100 + 120(s_x + s_y) - 144 s_x s_y\}\tau^{n}_{j,k},
\end{aligned} \tag{4.4.14}$$

which can be used to overcome the boundary problems of the spatially wide (1,13) and (1,21) explicit equations discussed earlier. Although using this equation to overcome problems near the boundary involves solving a set of equations at each time step, the set of equations in this case is basically tri-diagonal in nature, and so can be solved very efficiently by the Thomas algorithm, exactly as was done for the one-dimensional implicit methods. This is in contrast to using the implicit method to solve the entire problem, where the band width of the coefficient matrix increases with the values of J and K, which then uses extremely large amounts of CPU time.

Attempts to produce a sixth-order (9,9) equation, which may be useful to overcome boundary problems if a sixth-order explicit equation is developed by using a weighted differencing for the time derivative as well gives similar results to the above; namely, making the method fourth-order eliminates all the weights from some of the fourth-order error terms, thus making it impossible to eliminate all these terms and make the method sixth-order.

4.5 Locally One-Dimensional Methods

Another solution technique which is worth consideration here is the class of methods referred to as "locally one-dimensional" (LOD) methods, developed by people such as D'Yakonov (1963) and Marchuk (1975). In such techniques, rather than using a two-dimensional finite-difference equation to solve the two-dimensional diffusion equation, the two-dimensional problem is split into a series of one-dimensional problems. Because this one-dimensional approach only allows for diffusion in one direction, the direction in which the one-dimensional equation is applied must be constantly swapped, from the x-direction to the y-direction then back again, between each time step.

The main advantage of this technique is that the accurate techniques that were developed for the one-dimensional case can be applied directly. From a development point of view this is desirable since the one-dimensional equations contain fewer weights, and are thus usually much easier to work with than the full two-dimensional equations.

In order to implement such a scheme in practice, it is necessary to determine when to swap the direction of the one-dimensional scheme. It is found that it is best to split each time step in half. For the first half time step, the one-dimensional technique is used in, say, the y-direction, then for the second half time step it is changed to the x-direction. Combined with this, however, is the need to correctly model the amount of diffusion in each direction. In a real situation, there is an amount of diffusion α_y in the y-direction, and this is present over the whole time. In the numerical implementation, however, the y-direction diffusion is only present in the first half time step, when the y-direction one-dimensional equation is in use. To overcome this, the amount of diffusion in the numerical scheme is doubled. In similar fashion, the x-diffusion is doubled over the second half time step, to account for the fact that there was none in the first half time step. The net effect of this is to leave the values of s_x and s_y unaltered, since although the diffusion has been doubled, the time step is only half as large, and these two factors cancel each other.

Another factor that must be taken into account is that the boundary values which apply at the intermediate time level are not those specified for the "complete" (ie.

non-intermediate) time levels. Firstly, it should be noted that the boundary values at $y = 0$ and $y = 1$ at the intermediate time level are not required for the second half-step to the new time level, and so need not be calculated. If the values on the boundaries $x = 0$ and $x = 1$ are found by application of the specified boundary condition applied at the intermediate time level, it is found that any finite-difference method degenerates to second-order accuracy. This is due to the fact that the boundary value incorporates diffusion in both the x and y-directions, whereas the interior values computed at the intermediate time level include diffusion only in the y-direction, due to the use of the one-dimensional equation. This effect is clearly shown by running the (1,3) FTCS equation with values of $s = 1/6$ and $s = 1/2$, the results of which are shown in Figures 4.24 and 4.25. Note that results are shown only up to $J = K = 80$, since computational time rises dramatically for higher values of J and K, and no more information is obtained.

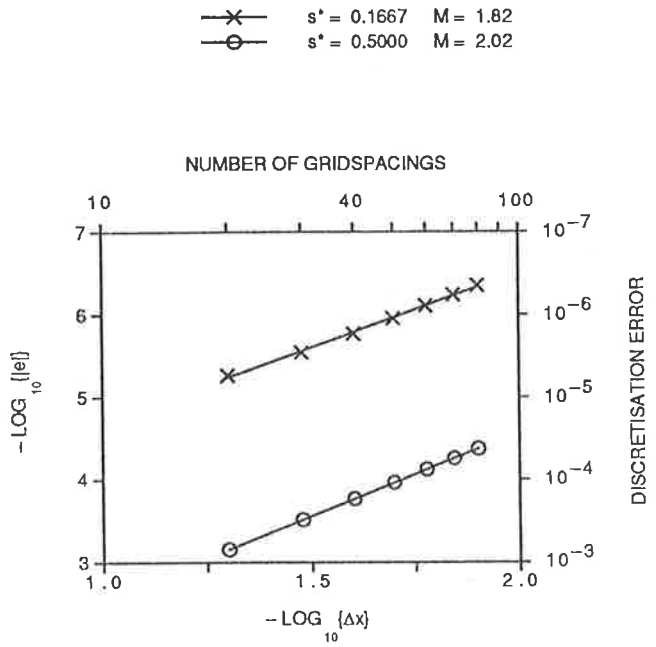


Figure 4.24: *Error vs grid spacing graph for the (1,3) FTCS LOD method, using the boundary condition at the intermediate time level.*

The alternative to using the boundary condition at $x = 0$ and $x = 1$ at the intermediate time step is to take the boundary values from the previous time level and apply the

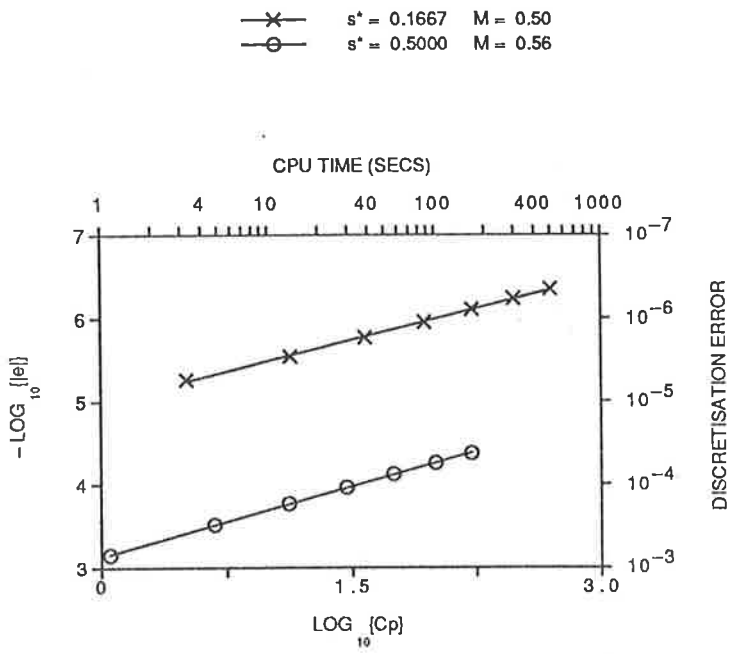


Figure 4.25: *Error vs CPU time graph for the (1,3) FTCS LOD method, using the boundary condition at the intermediate time level.*

finite-difference equation being used elsewhere in the interior of the region to give the values along the boundaries at the intermediate level. Note that the endpoint values at $y = 0$ and $y = 1$ are not required. This technique has the advantage that the values at the intermediate time level all incorporate consistent amounts of diffusion in both spatial directions. The numerical results obtained after making this change to the FTCS LOD method used above are shown in Figures 4.26 and 4.27. Note that the solution for $s^* = 0.5$ is theoretically only second-order accurate, so this solution suffered little degradation in accuracy. This explains the evident lack of improvement in this solution.

The above results clearly show that using the correct treatment of the boundaries at the intermediate time level is extremely important in the generation of the final solution. In particular, the given boundary condition should not be applied at the intermediate time level, which avoids the problems shown above. This is particularly relevant for implementing equations which use spatially wide computational stencils, such as the (1,5) and (1,5,1) equations. In these cases, we need to find the values at

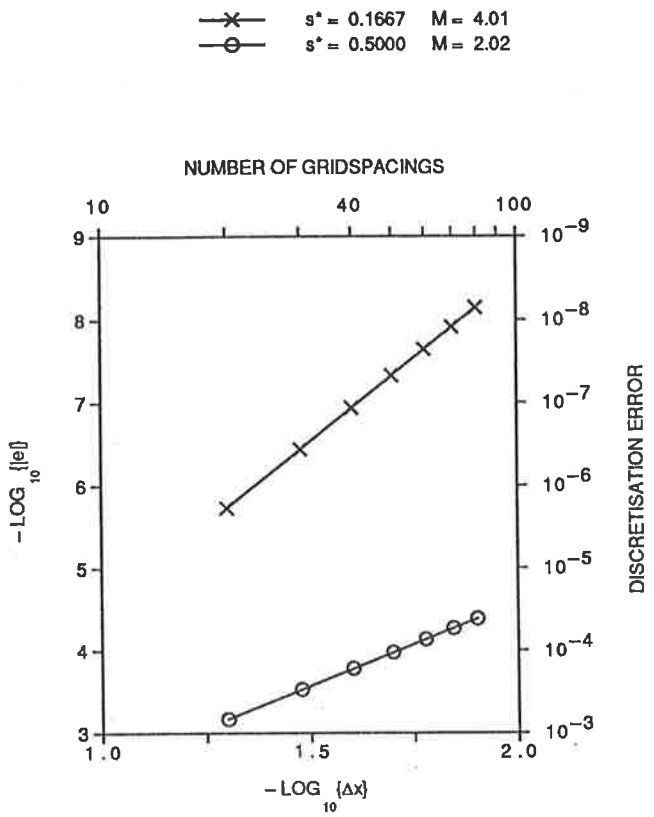


Figure 4.26: *Error vs grid spacing graph for the (1,3) FTCS LOD method, using the (1,3) equation on the boundary at the intermediate time level.*

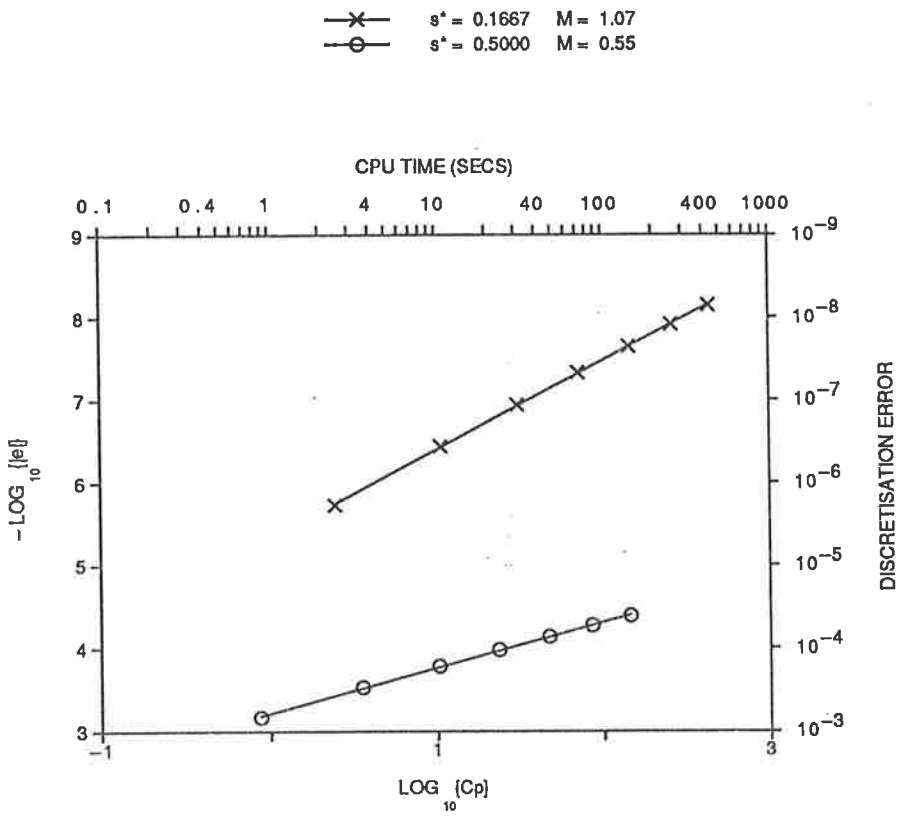


Figure 4.27: *Error vs CPU time graph for the (1,3) FTCS LOD method, using the (1,3) equation on the boundary at the intermediate time level.*

the grid points next to the boundary, without requiring the value at the grid point on the boundary.

This problem can be overcome in exactly the same way as was done for the Neumann boundary condition case for the one-dimensional problem. The re-arrangement of Crandall's fourth-order implicit equation (3.3.4), namely

$$\begin{aligned}\{1-6s\}\tau_1^{n+1} &= \{1+6s\}(\tau_1^n+\tau_3^n)+2\{5-6s\}\tau_2^n \\ &- 2\{5+6s\}\tau_2^{n+1}-\{1-6s\}\tau_3^{n+1}\ , \quad s\neq 1/6, \end{aligned} \tag{4.5.1}$$

can be used to find the missing end point. In the case $s = 1/6$, Crandall's equation reduces to the fourth-order special case of the (1,3) FTCS equation, and this equation can then be applied to find the required value in this case, without any loss of accuracy. The numerical results from applying this technique to the fourth-order (1,5) equation are shown in Figures 4.28 and 4.29.

These results show that this technique is producing answers that are as accurate as the fourth-order two-dimensional equations, such as the (1,13) and (1,21) equations discussed earlier. Also notice that the CPU times are slightly less than for the (1,13) scheme, and that to achieve a specific accuracy requires much less CPU time. Overall, use of the LOD method seems to be better than using the corresponding fully two-dimensional finite-difference equations.

The effective use of such three-level one-dimensional equations as the sixth-order (1,5,1) and (1,3,3) equations requires several problems to be overcome. Firstly, a two-level starter must be employed for the first half time step, and this must produce diffusion in the correct direction. Ideally, this starting method should be of the same order of accuracy as the three-level equation to be used, but for sixth-order equations appropriate two-level starter methods are not usually available. As was the case in the one-dimensional case, a fourth-order starter, such as the fourth-order (1,5) equation, should be used. Having completed the first half time step using the starter, computations over the next half time step can be carried out with either the starter method or the three-level method to be used for the rest of the time steps, depending on the method of implementation. In either case, the diffusion must be in the opposite

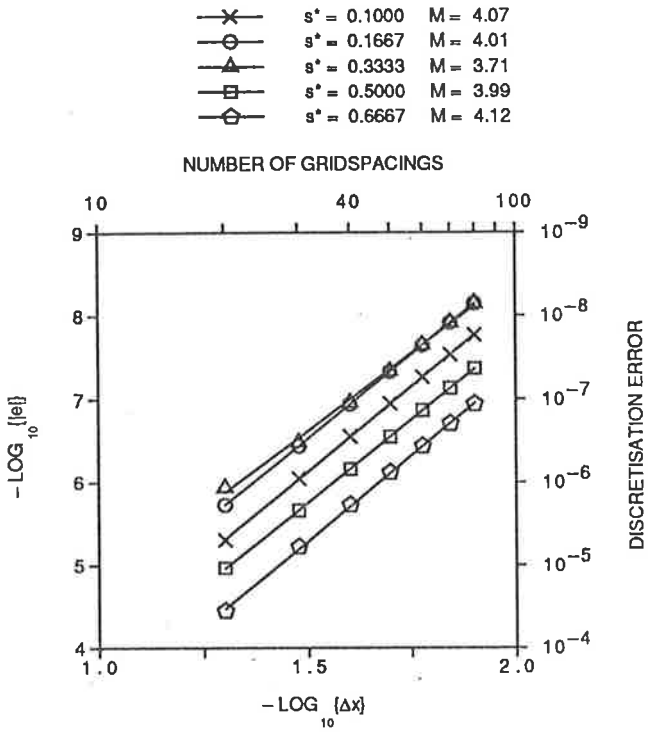


Figure 4.28: *Error vs grid spacing graph for the fourth-order (1,5) LOD method, using Crandall's equation next to the boundary.*

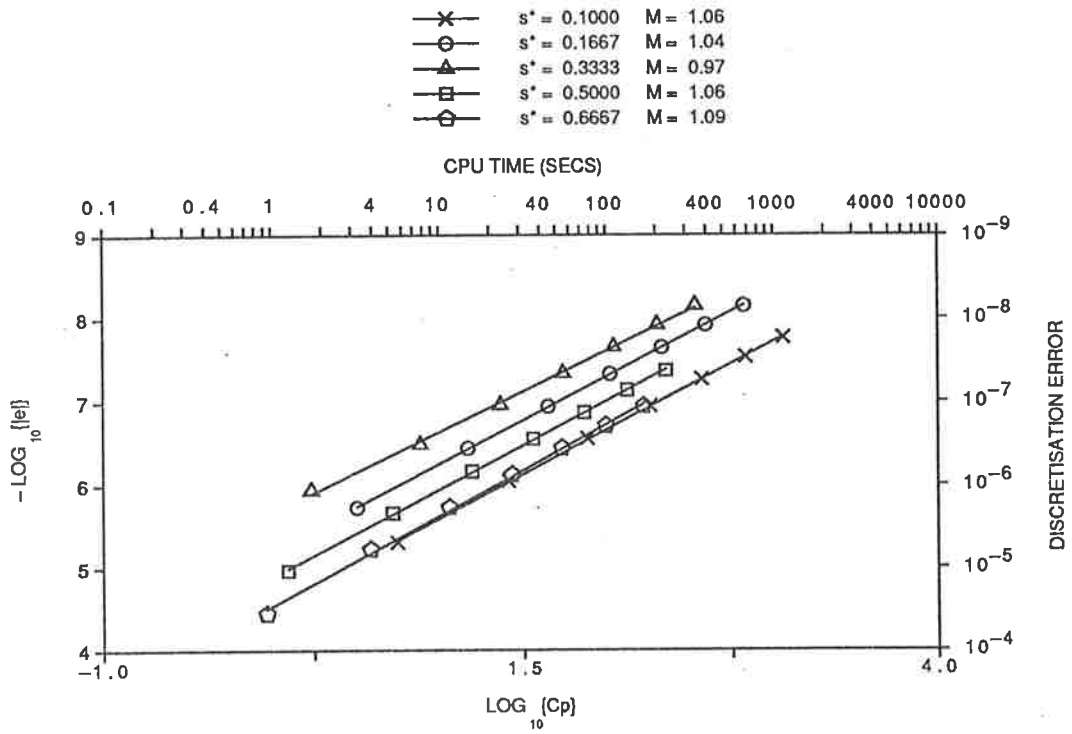


Figure 4.29: *Error vs CPU time graph for the fourth-order (1,5) LOD method, using Crandall's equation next to the boundary.*

direction to that which was initially used by the starting scheme.

The results obtained using such equations show the error vs. grid spacing graphs having slopes around two, despite the finite-difference equations being sixth-order. This is due to the fact that a three-level equation uses values from two previous time steps. For the one-dimensional diffusion equation, this is no problem, but when applied to the two-dimensional case, it results in values that include diffusion in only one spatial direction being combined with values that include diffusion in both directions to try to produce an approximation that alternates between requiring diffusion in only one direction and requiring it in both directions. This "mixing" of values results in inaccurate approximations and degrades the accuracy of the solution to second-order irrespective of the accuracy of the finite-difference equation employed.

Overall, it can be seen that locally one-dimensional techniques provide an effective way to solve the two-dimensional problem. In particular, these techniques are much simpler to develop and implement than fully two-dimensional equations, due to the absence of the cross-derivative error terms that must be eliminated to produce accurate fully two-dimensional equations. At the moment, the locally one-dimensional technique is restricted to two-level finite-difference equations, which in turn limits accuracy to fourth-order. If the problems associated with "mixing" values from several time levels can be overcome, however, then the more accurate sixth-order three-level equations could also be employed.

4.6 Alternating Direction Implicit Methods

The final solution technique to be considered in this work is the alternating direction implicit (ADI) approach. This class of techniques, originally developed by Peaceman and Rachford (1955) and Douglas (1955), is based on a desire to obtain the useful properties of implicit methods, most notably the extended stability range, without incurring the enormous CPU time overheads of using fully two-dimensional implicit equations. This is done by using an equation which is implicit in one spatial direction only, while points from the current and previous time levels can be used as required,

since these do not affect the explicit or implicit nature of the equation. In practice, this means that for each value of $j = 1..J-1$, the ADI equation can be used to generate a set of $K-1$ equations that must be solved to find the values $\tau_{j,1}^{n+1}..\tau_{j,K-1}^{n+1}$. Thus $J-1$ of these sets of equations must be solved to step from time level n to time level $n+1$. This contrasts with the fully implicit equations which require the solution of a single set of $(J-1)(K-1)$ equations. The big advantage of the ADI technique is that the bandwidth of the sets of equations is a fixed number that depends only on the spatial extent of the computational stencil, which allows the use of very efficient solving techniques such as the Thomas algorithm can be used. This compares with the fully implicit equation where the bandwidth of the set of equations is proportional to the product of the numbers of grid spacings in either spatial direction, which leads to enormous amounts of CPU time being required to solve the equations as J and K are increased.

Also of note is the fact that by alternating the spatial direction in which the equation is applied, it is possible to get cancellation of errors between the two sweeps. This means that ADI equations that by themselves are only say second-order accurate may actually produce results that are fourth-order accurate, if the second-order errors can be made to cancel each other out on sweeps of alternate direction.

The original ADI equation, which is based on the (3,3) computational stencil shown in Figure 4.30, is the simplest possible form for an ADI equation, and there is no possibility of introducing any weights into an equation based on this stencil. Note that, unlike the LOD methods discussed earlier, an ADI equation is consistent with the complete two-dimensional diffusion equation, so there are no problems with having to split the time steps in half, and any boundary conditions specified will be correct after each step of the ADI process. In some situations, however, it may be desirable to actually split the time steps in half, particularly in the case where the desired time level occurs after an odd number of full time steps.

The finite-difference equation based on the stencil shown in Figure 4.30 is

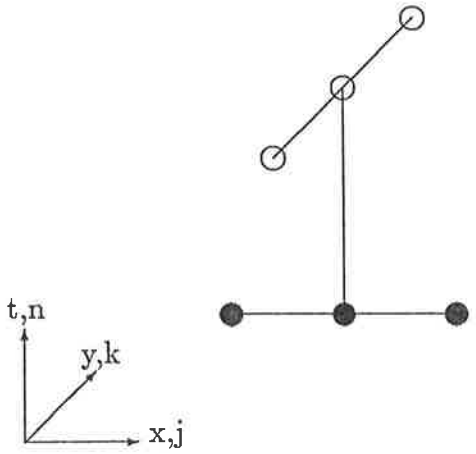


Figure 4.30: *Computational stencil for the Classical (3,3) ADI method*

$$\begin{aligned} s_y(\tau^{n+1}_{j,k-1} &+ \tau^{n+1}_{j,k+1}) - (1+2s_y)\tau^{n+1}_{j,k} \\ &= -s_x(\tau^n_{j-1,k} + \tau^n_{j+1,k}) - (1-2s_x)\tau^n_{j,k}, \end{aligned} \tag{4.6.1}$$

with a very similar equation with the roles of x and y reversed holding for the alternate direction sweep. The modified equivalent equation corresponding to this equation has leading error terms containing the factors

$$\begin{aligned} \Gamma_{4,0} &= 6s_x - 1 \\ \Gamma_{4,2} &= 0 \\ \Gamma_{4,4} &= -(6s_y + 1), \end{aligned} \tag{4.6.2}$$

while the factors for the alternate direction equation are

$$\begin{aligned} \Gamma_{4,0} &= -(6s_x + 1) \\ \Gamma_{4,2} &= 0 \\ \Gamma_{4,4} &= 6s_y - 1. \end{aligned} \tag{4.6.3}$$

It can be readily seen from this that either equation by itself is second-order accurate, and no choice for the values of $s_x > 0$ and $s_y > 0$ will make it fourth-order. It is worth noting that these error terms are *not* symmetric, which is one of the main reasons for

using the equation with the implicit portion alternating between the spatial directions with every time step. In this manner, the errors tend to balance each other, and in the case of some equations they can even be made to cancel each other to produce more accurate results.

The other reason for using the alternating direction is the von Neumann stability of the equation. Equation (4.6.1) itself is only conditionally stable (Noye, 1984), but it can be shown that the use of this equation over two time steps, one in each spatial direction, is *unconditionally* stable, which makes this technique more attractive in some practical situations.

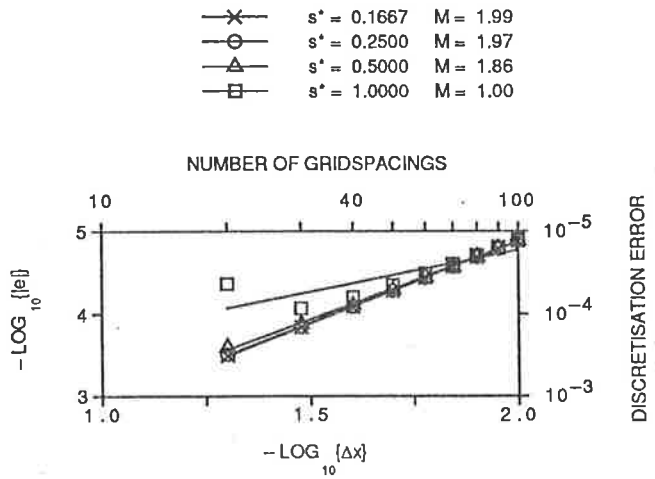


Figure 4.31: *Error vs grid spacing graph for the classical (3,3) ADI method.*

The results from this method, shown in Figures 4.31 and 4.32 bear out the theoretical predictions. The results are very close to being second-order accurate, with the exception of the unexpectedly accurate result for $s = 1$ with $J = K = 20$, which may be due to some cancellation amongst the leading error terms of the modified equivalent equation. It should be noted that alternating the direction of use of the stencil has the effect of adding components of the modified equivalent equation, so for the double sweep the coefficients are

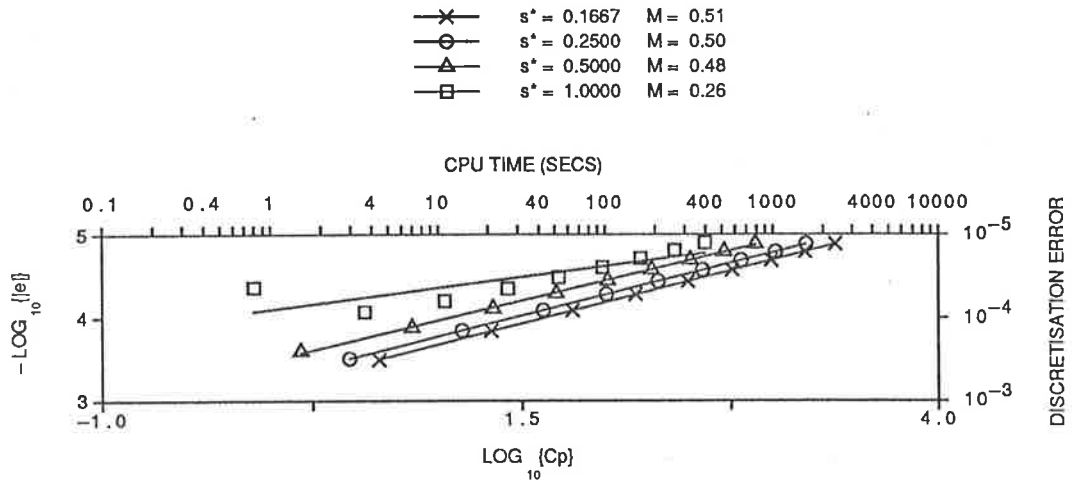


Figure 4.32: *Error vs CPU time graph for the classical (3,3) ADI method.*

$$\begin{aligned} \Gamma_{4,0} &= -2 \\ \Gamma_{4,2} &= 0 \\ \Gamma_{4,4} &= -2. \end{aligned} \tag{4.6.4}$$

The fact that the leading error terms are independent of s_x and s_y explains why the numerical errors are so close together regardless of the value of $s_x = s_y = s^*$ chosen.

Thus ADI methods appear to be viable in practice, but we need to look at ways of getting better accuracy from such techniques if they are to be a useful alternative to the methods discussed earlier.

If all the grid points $(j \pm 1, k \pm 1, n)$ are incorporated into a (3,9) computational stencil, then four weights can be introduced in the finite-difference equation; one on the time derivative, one on one of the spatial directions and two on the other spatial direction (which is the direction in which the equation is implicit). If this is done, the leading error terms are found to have coefficients that contain the factors

$$\begin{aligned} \Gamma_{4,0} &= 6s_x - 1 \\ \Gamma_{4,2} &= (\lambda - \theta)s_x - \varphi s_y + (1 - \gamma)s_x s_y \\ \Gamma_{4,4} &= 6s_y - 1 + 12\lambda - 12\gamma s_y \end{aligned} \tag{4.6.5}$$

While this would appear to be of little use, since $\Gamma_{4,0}$ contains no weights, the property

of adding the errors over a double sweep can be used to advantage here. In particular, if we can force $\Gamma_{4,2} = 0$ and $\Gamma_{4,4} = 1 - 6s_y$, then when the two errors are added over a double sweep, all the second-order errors will cancel, and the resulting method will be fourth-order.

In order to achieve this fourth-order accuracy, the values

$$\begin{aligned} \lambda &= 1/6 \\ \theta &= 1/6 \\ \gamma &= 1 \\ \varphi &= 0 \end{aligned} \tag{4.6.6}$$

must be chosen, which leads to the "optimal" finite-difference equation

$$\begin{aligned} \{6s_y - 1\}(\tau^{n+1}_{j,k-1} + \tau^{n+1}_{j,k+1}) \; - \; & 4\{1 + 3s_y\}\tau^{n+1}_{j,k} = \\ -s_x(\tau^n_{j-1,k-1} \; + \; & \tau^n_{j-1,k+1} + \tau^n_{j+1,k-1} + \tau^n_{j+1,k+1}) - 4s_x(\tau^n_{j-1,k} + \tau^n_{j+1,k}) \\ +4\{2s_x - 1\}\tau^n_{j,k} \; + \; & \{2s_x - 1\}(\tau^n_{j,k-1} + \tau^n_{j,k+1}). \end{aligned} \tag{4.6.7}$$

The double sweep of this equation can been shown numerically to be unconditionally von Neumann stable, and its unconditional solvability, even over a single sweep, is very simply shown analytically from the coefficients at time level $n + 1$, namely by showing that

$$4|1 + 3s_y| \geq 2|6s_y - 1| \tag{4.6.8}$$

for all $s_y > 0$. Thus this method, used in an ADI fashion, is usable for all values of $s_x > 0$ and $s_y > 0$.

The results of a numerical experiment with this equation are shown in Figures 4.33 and 4.34, from the fourth-order nature of the solutions can be seen. This is worthy of comment because it is the first time that we have been able to obtain fourth-order accuracy using only a compact nine-point computational stencil at each time level; all the other methods considered have used spatially wide stencils. The interest here lies not so much in this fact for itself, since we have found accurate ways to overcome the problems of spatially wide stencils, but in the fact that maybe this could lead to

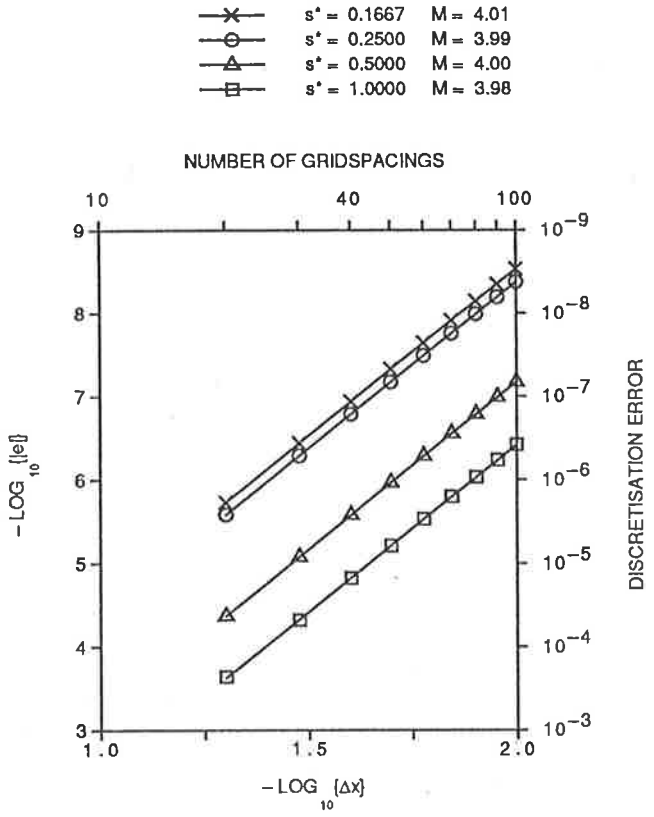


Figure 4.33: *Error vs grid spacing graph for the fourth-order (3,9) ADI method.*

sixth-order equations using a manageable sized computational stencil (ie. at most a 5×5 square of grid points at any one time level).

Also worth noting is that the size of the error increases with s^*, so the most accurate results are obtained for the smallest values of s^*, which contrasts with some of the earlier methods. This is due to the fourth-order error term of the double sweep, which contains the factors

$$\begin{aligned} \Gamma_{6,0} &= -2(1 - 30s_x + 120{s_x}^2) \\ \Gamma_{6,2} = \Gamma_{6,4} &= 0 \\ \Gamma_{6,6} &= -2(1 - 30s_y + 120{s_y}^2) \end{aligned} \tag{4.6.9}$$

which vanish (giving a sixth-order method) for $s_x = s_y = s^* \approx 0.039...$ and $s^* \approx 0.210...$, and increase in magnitude as s^* increases above 0.210.... From the CPU graph it can also be seen that these small values of s^* are the most efficient values, since they provide a solution to a specified accuracy in the smallest amount of CPU time. In absolute terms, however, the efficiency of this technique is not as good as

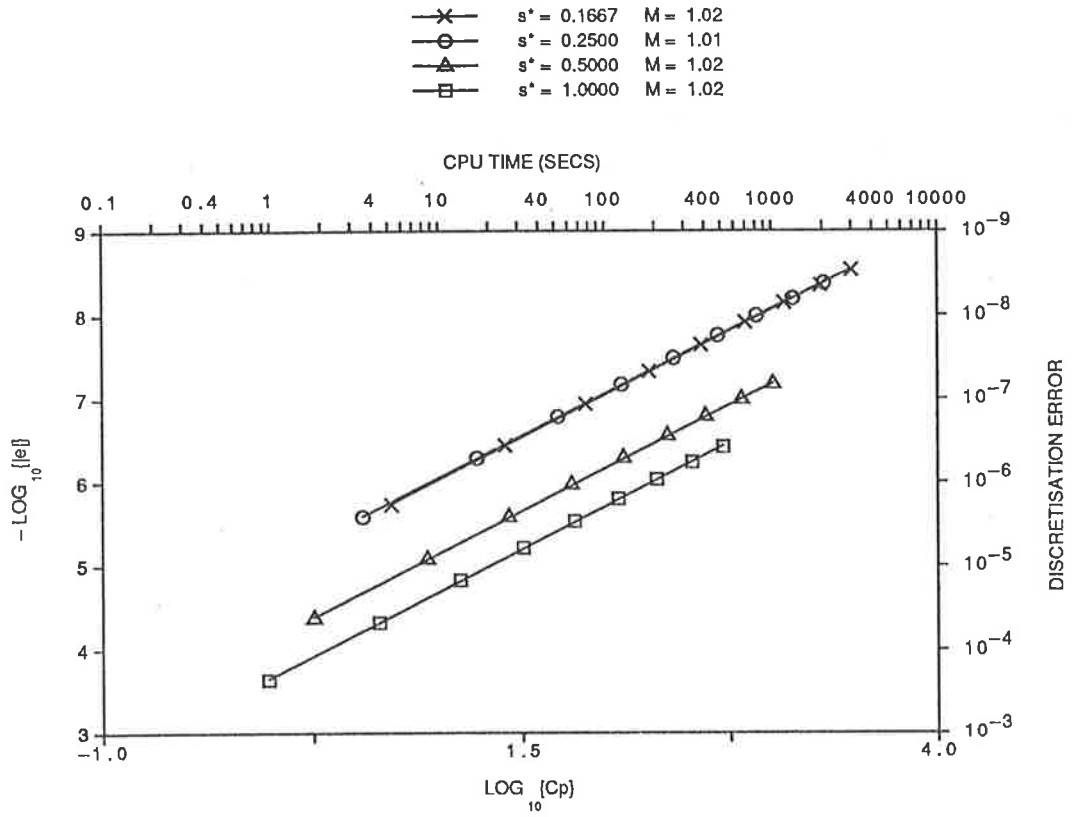


Figure 4.34: *Error vs CPU time graph for the fourth-order (3,9) ADI method.*

that of the explicit (1,13) and (1,21) equations; up to 30% more CPU time is required to generate solutions of a given accuracy, most of which is due to the time required to solve the sets of equations at each time level.

Further extensions to the computational stencil, such as a (3,13) stencil or even three-level stencils such as (3,9,5), (3,9,9) or (3,9,13) could be developed in exactly the same fashion in an attempt to gain such a sixth-order equation. Care should be taken, however, as to which errors are eliminated, since some errors may be automatically taken care of by the alternate-direction step, so removing these using the weights is unnecessary. The development of such equations is not done here, largely because the emphasis of the work has been towards explicit methods, since these can use the full potential of array processors whereas implicit methods cannot, but also due to time limitations.

Another variety of ADI method is that based on the computational stencil shown in Figure 4.35, which is a loose two-dimensional analogue of the technique used by

Saul'yev (1964) for the one-dimensional problem. Such an equation can still be considered as ADI because, given the values at the $x = 0$ boundary at time level $(n+1)$ from the boundary condition, this equation can be used to find the values at $x = \Delta x$ by application of the ADI process. These values can then be used to find those at $x = 2\Delta x$, and so on across the grid.

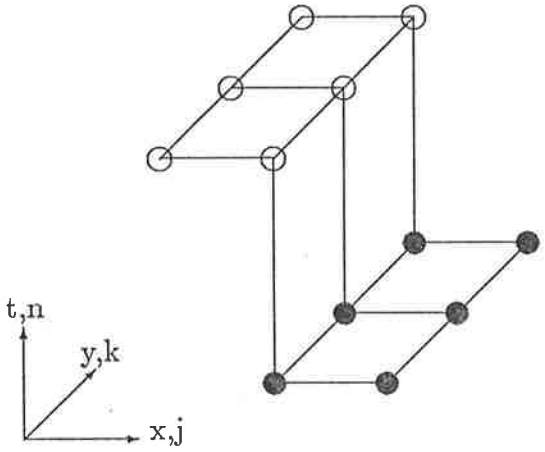


Figure 4.35: *Computational stencil for the (6,6) ADI method*

There is an immediately obvious problem in that there is no set of three points at the same time level that can be used to approximate the $\partial^2\hat{\tau}/\partial x^2$ term in the diffusion equation. The easiest way around this problem is to consider using the full (9,9) stencil, on which the differencing is straight-forward, and adding and subtracting an appropriate multiple of $\partial\hat{\tau}/\partial x$ to the diffusion equation. If the weighted scheme used for $\partial^2\hat{\tau}/\partial x^2$ at *both* time levels is say $\gamma\times$ CS at $k \pm 1$ and $(1-2\gamma)\times$ CS at k, then both extra $\partial\hat{\tau}/\partial x$ terms must be multiplied by γ; the extra term which is subtracted is approximated by forward-space differencing at time level $(n+1)$ and the one which is added is approximated by backward-space differencing at time level n. All this has the effect of removing the unwanted grid points from the resulting finite-difference equation, leaving an equation which uses only the desired grid points.

As usual, the modified equivalent equation corresponding to this equation must be examined, and in this case the leading error terms contain the factors

$$\begin{aligned}
\Gamma_{3,0} &= s_x \\
\Gamma_{3,1} &= 0 \\
\Gamma_{3,2} &= s_x s_y \\
\Gamma_{3,3} &= 0,
\end{aligned} \tag{4.6.10}$$

which appears to make the method only first-order accurate. In fact, however, the $C_{3,0}$ and $C_{3,2}$ terms of which the non-zero factors are part both contain the factor Δx as well as the factor given. This means that if the equation is run both "left-to-right" and "right-to-left" for each spatial direction, these errors will cancel each other out, since the second sweep is the equivalent of the first with Δx replaced by $-\Delta x$. Thus the method can be made second-order, even before any alternating direction techniques are applied.

This method can be further improved, since there are still weights in the equation. The second-order error terms contain the factors

$$\begin{aligned}
\Gamma_{4,0} &= -1 - 12{s_x}^2 \\
\Gamma_{4,2} &= (\gamma - \varphi)s_x + \epsilon s_y + {s_x}^2 s_y \\
\Gamma_{4,4} &= -1 + 12\varphi.
\end{aligned} \tag{4.6.11}$$

It should be noted that since these are multiplied by $(\Delta x)^2$ in the appropriate $C_{4,n}, n = 0, 2, 4$ terms, the double sweep to remove the first-order error terms will only introduce a factor of two to all of these, and this does not affect the following calculations.

We wish to make $\Gamma_{4,2} = 0$ and $\Gamma_{4,4} = 1 + 12{s_y}^2$ in order to achieve fourth-order with this ADI scheme. This can be readily done by the choice of weights

$$\begin{aligned}
\gamma = \varphi &= (1 + 6{s_y}^2)/6 \\
\epsilon &= -{s_x}^2
\end{aligned} \tag{4.6.12}$$

which leads to an "optimal" finite-difference equation. Unfortunately, this equation is unstable for even such values as $s^* = 1/2$, and is found numerically to lack a sufficiently large von Neumann stability range to be of practical use, given that we have already

found a fourth-order ADI method that is unconditionally stable. Other equations based on this stencil, but which are only second-order accurate can be developed which are von Neumann stable and thus could be used to solve the two-dimensional diffusion equation, but the lack of accuracy makes this undesirable in practice.

Despite this, however, it may still be possible to develop stable finite-difference equations based on the ideas above, that require "marching" across the grid in a certain direction and obtain accurate solutions to the two-dimensional diffusion equation. Such an investigation, however, is not carried out in this work.

4.7 Summary

It has been seen that much of the information gained by the study of the one-dimensional problem has helped in the study of the two-dimensional case. In particular, the modified equivalent equation approach has allowed the analysis of the errors associated with various finite-difference equations, which in turn allows the development of more accurate equations. Also of great use were general techniques for dealing with spatially wide computational stencils next to the boundaries of the solution domain.

Unlike the one-dimensional case, it has not been possible with the time and computing resources available to develop a sixth-order accurate technique for solving the two-dimensional case. What has been done, however, is to compare the various classes of solution techniques via finite-differences, namely explicit and implicit two-dimensional equations, locally one-dimensional (LOD) equations and alternating direction implicit (ADI) methods. Implicit two-dimensional equations have been found to require enormous amount of CPU time, both relative to the other techniques and also in absolute terms. Such equations are therefore not considered to be practical solution techniques. ADI methods have not been considered in much detail here, although further work should be possible along the lines of the examples given above. The main reasons for not considering these methods in detail are that this work is directed mainly at explicit techniques, since implicit techniques tend by their nature to require more CPU

time, and also because explicit techniques can use the full computing potential of array processors; many values at the new time level can be calculated simultaneously from the known values at the old time level, whereas for implicit methods which require the solution of a set of linear equations this is not possible. This will further add to the CPU time difference between the two types of techniques.

There is little to separate the fourth-order explicit two-dimensional equations from the fourth-order LOD equations, since they both solve the test problem with the same degree of accuracy and in similar amounts of CPU time. The LOD methods are probably slightly better for several reasons, however. Firstly, the small differences in accuracy and CPU usage between two-dimensional equations and LOD methods tend to favour the LOD methods. Also, the development and implementation of one-dimensional equations tends to be simpler, due to the absence of cross-derivative error terms. Also, if the problems associated with using three-level methods in an LOD method are overcome this would provide a simple way to obtain a sixth-order accurate solution. This is potentially a much more elegant method of obtaining sixth-order accuracy for the two-dimensional problem than attempting to find a fully two-dimensional sixth-order equation.

Chapter 5

Irregular Boundaries

5.1 Introduction

The preceding work has all been based on solving the diffusion equation, in either one or two dimensions, on linear or rectangular spatial regions respectively which are covered by a rectangular grid of uniform spacing. Such a region is ideal for the application of finite-differences, since the rectangular grid completely covers the region, and it can be arranged so that grid points lie exactly on the boundaries, so the boundary conditions can be easily incorporated.

In a practical problem, however, it is unlikely that the region over which a solution is required will be perfectly rectangular. Instead, the boundary can be expected to be irregular in shape, and so in general the grid points will not lie exactly on the boundary. This creates more problems, since a technique for incorporating the boundary conditions into the solution process must be found that does not detract from the accuracy of the rest of the solution.

One way that can be used to overcome this is to find a transformation of coordinates that maps the solution domain onto a rectangular region, which also alters the governing partial differential equation accordingly, after which the problem is solved on the new, rectangular grid. The transformation usually alters the governing diffusion

equation into an advection-diffusion equation, the solution to which is somewhat more complicated than the diffusion equation being discussed in this work. This solution must then be transformed back into the original coordinates to get the solution to the actual problem. This approach is outside the scope of this work.

The other approach to the problem is to develop finite-difference equations which deal with the case where one grid spacing at the boundary is not the same size as the other grid spacings used. This creates problems of its own, since the finite-difference equation that results is no longer centred in space.

5.2 Variable Grid Equations

In order to investigate finite-difference equations to be used in such circumstances, it is necessary to define the grid to be used, and then develop appropriate approximations to the space derivative term in the diffusion equation. The grid next to the irregular boundary is set up as shown in Figure 5.1, assuming that the irregular boundary is on the right-hand edge of the region. If it is on the left, then a mirror-image of this figure applies. Note that the boundary is always assumed to lie in the interval $((J-1)\Delta x,\ J\Delta x]$. In the case where $r = 1$, the grid becomes the uniform grid we have been dealing with before. For $r > 1$, the grid point immediately to the left of the boundary itself would be $(J-2)\Delta x$, with the other points placed accordingly.

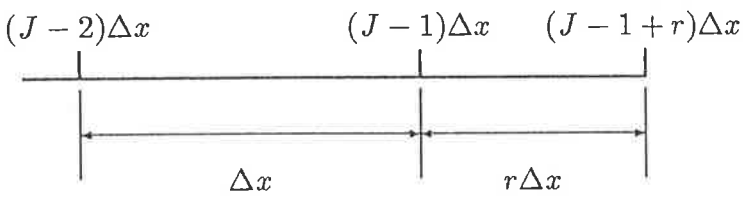


Figure 5.1: *Modified grid for an irregular boundary ($0 < r \leq 1$ here).*

The basic three-point approximation to $\partial^2\hat{\tau}/\partial x^2$ now becomes

$$\left.\frac{\partial^2\hat{\tau}}{\partial x^2}\right|_j^n = \frac{2r\hat{\tau}_{j-1}^n - 2(r+1)\hat{\tau}_j^n + 2\hat{\tau}_{j+r}^n}{r(r+1)(\Delta x)^2} + O\{(r-1)(\Delta x), r(\Delta x)^2\}, \tag{5.2.1}$$

where the $(j+r,n)$ grid point is the boundary point, which has a space position of $(j+r)\Delta x$ (cf. Noye, 1984). It is apparent that this is one order of accuracy less than the corresponding equation for the uniform grid (2.1.7). In the case where $r = 1$, this reduces to the uniform grid case, as expected. If equation (5.2.1) is used with the standard forward-time difference (2.1.6) for the time derivative (since the grid is still uniform in time), a forward-time variable-space (FTVS) (1,3) finite-difference equation, is obtained, namely

$$r(r+1)\tau_j^{n+1} = 2rs\tau_{j-1}^n + (r+1)(r-2s)\tau_j^n + 2s\tau_{j+r}^n. \tag{5.2.2}$$

The computer program used to evaluate the modified equivalent equation corresponding to a finite-difference equation has been adapted to cope with the case where the grid spacing is no longer uniform at one point, by allowing for the change in the Taylor series calculated. It is found that the modified equivalent equation corresponding to (5.2.2) has leading error terms which can be written in the general form (2.2.7) with coefficients

$$\begin{aligned} \Gamma_3(s,r) &= (1-r) \\ \Gamma_4(s,r) &= 6s - 1 + r(1-r). \end{aligned} \tag{5.2.3}$$

Note that these coefficients in general depend on both s and r. It can be seen that in the uniform grid case, where $r = 1$, the equations (5.2.3) reduce to the single original formula (2.2.9), but in the general case, equation (5.2.3) is only first-order accurate, due to the coefficient $\Gamma_3(s,r)$ being non-zero.

Thus the non-uniform grid spacing at the boundary has lowered the accuracy of this equation by one order. This means that it may be necessary to include more weights in such stencils just to achieve the same order of accuracy that was done for the uniform grid.

The non-uniform analogue of the five-point approximation (2.3.1), which again is one

order of accuracy lower than its uniform grid counterpart, is given by

$$\begin{aligned}\left.\frac{\partial^2\hat{\tau}}{\partial x^2}\right|_j^n &= \{-r(r+1)(r+2)\hat{\tau}_{j-2}^n + 3r(r+1)(r+3)^2\hat{\tau}_{j-1}^n \\ &- 3r(r+2)(r+3)(2r+3)\hat{\tau}_j^n + (r+1)(r+2)(r+3)\hat{\tau}_{j+1}^n - 6\hat{\tau}_{j+1+r}^n\} \\ &/ \{3r(r+1)(r+2)(r+3)(\Delta x)^2\} + O\{(\Delta x)^3\}. \end{aligned} \tag{5.2.4}$$

In similar fashion to the development of the fourth-order (1,5) equation for the uniform grid case, this can be used in weighted fashion with the centred three-point approximation (for the uniform grid), to give the weighted finite-difference equation

$$\begin{aligned}3r(r+1)(r+2)(r+3)\tau_j^{n+1} &= -\varphi sr(r+1)(r+2)\tau_{j-2}^n \\ &+ 3sr(r+1)(r+3)(r+2+\varphi)\tau_{j-1}^n \\ &+ 3r(r+2)(r+3)((r+1)(1-2s) - s\varphi)\tau_j^n \\ &+ sr(r+1)(r+2)(r+3)(\varphi+3)\tau_{j+1}^n \\ &+ 6\varphi s\tau_{j+1+r}^n. \end{aligned} \tag{5.2.5}$$

The modified equivalent equation corresponding to this finite-difference equation has a leading error term involving the coefficient

$$\Gamma_4(s,r) = \varphi - 1 + 6s \tag{5.2.6}$$

which means that the second-order error term can be eliminated by the choice of weight

$$\varphi = 1 - 6s. \tag{5.2.7}$$

It is of interest to note that this value is independent of the value of r, and hence that this is the same weight as that used for the uniform grid case. If this substitution is made, the finite-difference equation becomes

$$\begin{aligned}3r(r+1)(r+2)(r+3)\tau_j^{n+1} &= sr(r+1)(r+2)(6s-1)\tau_{j-2}^n \\ &+ 3sr(r+1)(r+3)(r-6s+3)\tau_{j-1}^n \\ &+ 3r(r+2)(r+3)((r+1)(1-2s) - s(1-6s))\tau_j^n \\ &+ s(r+1)(r+2)(r+3)(3r-6s+1)\tau_{j+1}^n \\ &+ 6s(6s-1)\tau_{j+1+r}^n, \end{aligned} \tag{5.2.8}$$

which is the non-uniform equivalent of the fourth-order (1,5) equation (2.3.8). In this case, however, examination of the modified equivalent equation shows that the leading error term contains the factor

$$\Gamma_5(s,r) = (r-1)(1-6s), \tag{5.2.9}$$

which means that the equation (5.2.8) is only third-order accurate, except in the special cases where either $r = 1$ or $s = 1/6$. In either of these cases, the leading error term is eliminated and the equation becomes fourth-order.

The von Neumann stability range of this equation is worth considering. Except for the impractical values $r = 0, -1, -2, -3$, which remove the grid value from time level $(n+1)$ from the difference equation, the stability range for $r \leq 0$ is approximately $s \leq 0.1$. As r increases from 0 to 1, this range increases to $s \leq 2/3$, which is the restriction present in the uniform grid case. What is more interesting, however, is that as r is increased from 1 to 2, the stability range increases even more, until at $r = 2$ it is nearly $s \leq 1$. This suggests that, if this equation is to be used in practice, the value of r used should be greater than one. Thus next to the boundary, the last grid spacing should be *larger* than the rest of the grid spacings, rather than smaller.

Once again, to implement this scheme in practice requires another method to be used next to the boundary, due to the spatial extent of the (1,5) stencil. Taking the analogue from the one-dimensional case, a non-uniform version of Crandall's (3,3) equation will be developed. Thus to solve the diffusion equation, the uniform grid (1,5) equation can be used in the interior of the solution domain, the variable grid (1,5) equation is used two grid points in from the irregular boundary and the variable grid Crandall equation is used next to the boundary.

The optimal-order variable grid (3,3) equation can be developed in the same way as in the uniform grid case. In this case, however, the forward-time approximations to the time derivatives at $(j \pm 1, n)$ are included with weights λ and θ respectively, as well as the spatial weight θ. This increases the number of weights available to eliminate unwanted error terms from the modified equivalent equation. The weighted equation

that results is

$$\begin{aligned}
&\{2r\theta s - \lambda r(r+1)\}\tau_{j-1}^{n+1} \\
+(r+1)&\{r(\gamma - \lambda - 1) - 2\theta s\}\tau_j^{n+1} \\
+\{2\theta s - \gamma r(r+1)\}&\tau_{j+r}^{n+1} = \{2rs(\theta-1) - \lambda r(r+1)\}\tau_{j-1}^n \\
&+ (r+1)\{r(\lambda+\gamma-1) + 2s(1-\theta)\}\tau_j^n \\
&+ \{2s(\theta-1) - \gamma r(r+1)\}\tau_{j+r}^n. \qquad (5.2.10)
\end{aligned}$$

This weighted equation can be analysed by finding its modified equivalent equation, which is found to have complicated coefficients C_3 and C_4 of the leading error terms. In order to eliminate these terms, the values

$$\begin{aligned}
\lambda &= \{r(12\theta s - 6s - r^2 + r + 1\}/\{6r(r+1)\} \\
\gamma &= \{12\theta s - 6s + r^2 + r - 1\}/\{6r(r+1)\} \qquad (5.2.11)
\end{aligned}$$

must be chosen. If these values are substituted back into equation (5.2.10), it is found that the weight θ is also eliminated from the finite-difference equation, which becomes

$$\begin{aligned}
&\{r(6s + r^2 - r - 1)\}\tau_{j-1}^{n+1} \\
-\{(r+1)&(6s + r^2 + 3r + 1)\}\tau_j^{n+1} \\
+\{6s - r^2 - r + 1\}&\tau_{j+r}^{n+1} = \{-r(6s - r^2 + r + 1)\}\tau_{j-1}^n \\
&+ \{(r+1)(6s - r^2 - 3r - 1)\}\tau_j^n \\
&- \{6s + r^2 + r - 1\}\tau_{j+r}^n. \qquad (5.2.12)
\end{aligned}$$

This equation can be analysed using the modified equivalent equation approach, which shows that the leading error term involves the factor

$$\Gamma_5(s,r) = \frac{(r-1)(r+2)(2r+1)}{3}, \qquad (5.2.13)$$

which indicates that this equation, like the optimal case of the (1,5) equation on a variable grid, is only third-order accurate. This consistency of orders allows the optimal (3,3) equation to be used to overcome the boundary problems associated with the (1,5) equation.

Despite the fact that these methods are only third-order accurate for general values of r and s, it is worth running some numerical tests to see what results are actually obtained by using them. This is because the real problem we wish to solve is the two-dimensional case, and it was seen in the last chapter that three-level methods for the one-dimensional problem require more work before they can be used in locally one-dimensional fashion. Thus the high-order (1,3,3) and (1,5,1) equations developed earlier are of no interest in the current context, until such problems with LOD methods are resolved.

To test the variable grid (1,5) and (3,3) equations in practice, we again use the same problem as for the one-dimensional uniform grid case, except that the left-hand boundary is no longer at $x = 0.0$, but at some point $x = x_L$, where $0 \leq x_L < \Delta x$ and $1 < r \leq 2$. The exact value for x_L is determined by the value of r being used, since the coefficient of the leading error term in the modified equivalent equation is now $\Gamma_q(r, s)$ rather than just $\Gamma_q(s)$ as has been the case so far. Thus for our error graphs to remain as straight lines, the values of both s and r must be specified, which means that the position of the left hand boundary must vary as the grid spacing is changed. The numerical results are still taken from the point $x = 0.2$ at time $T = 8$.

The results from such a numerical test are shown in Figures 5.2 to 5.5, which shows that the method is still close to fourth-order accurate, despite using the third-order variable grid equations at the boundary. The exceptionally good value for $r = 1.25$ with $J = 70$, which is not present for $r = 1.75$, is an indication that for this set of values and this problem, there is in some sense an "optimal" case, with some error terms cancelling each other out to produce a better than expected result. Other than this, however, the results are showing that the accuracy of the fourth-order (1,5) has not been diminished by the introduction of the irregular grid spacing at the boundary.

Given this good result in the one-dimensional case, this technique can be readily extended to solve the two-dimensional problem by means of the LOD technique. This would then provide an efficient means of solving the two-dimensional problem on an irregular region, such as may be found in some physical region being modelled. One big advantage is that this method requires only two forms of the equations, one for the

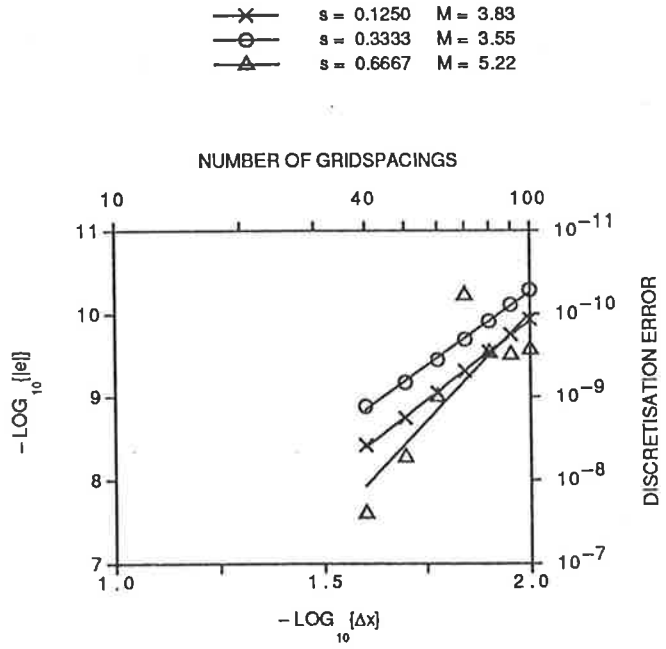


Figure 5.2: *Error vs grid spacing graph for the one-dimensional problem using the variable grid (1,5) equation with an irregular left-hand boundary,* $r = 1.25$.

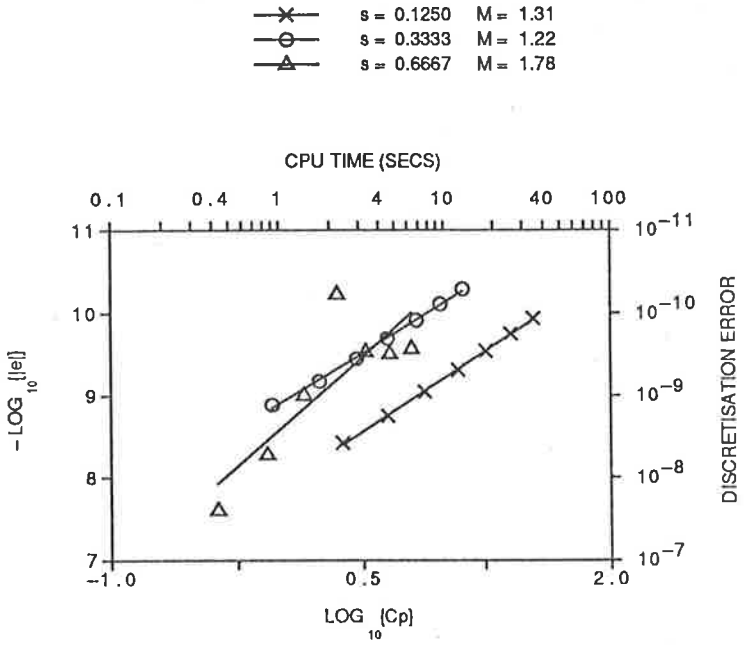


Figure 5.3: *Error vs CPU time graph for the one-dimensional problem using the variable grid (1,5) equation with an irregular left-hand boundary,* $r = 1.25$.

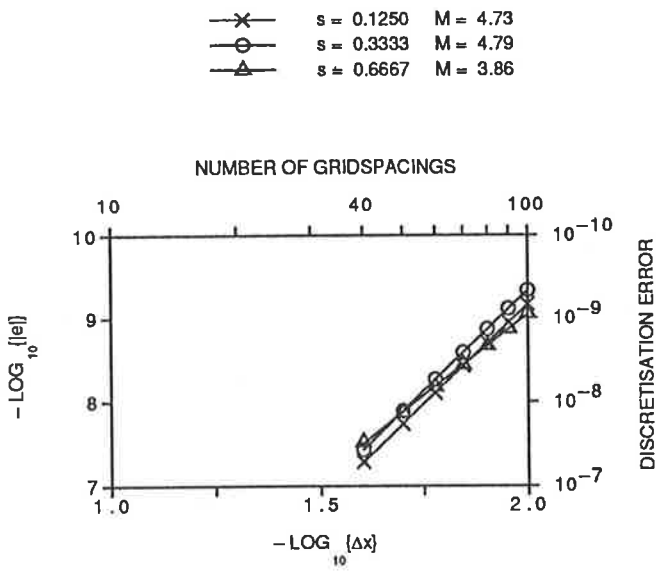


Figure 5.4: *Error vs grid spacing graph for the one-dimensional problem using the variable grid (1,5) equation with an irregular left-hand boundary,* $r = 1.75$.

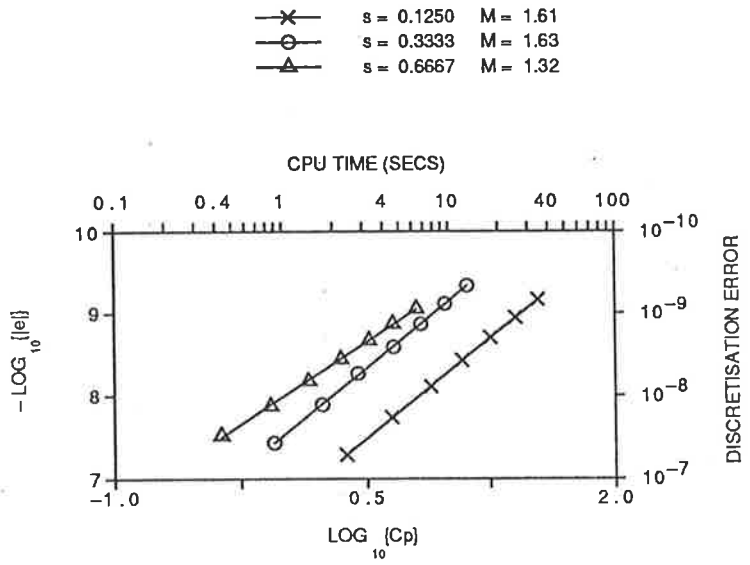


Figure 5.5: *Error vs CPU time graph for the one-dimensional problem using the variable grid (1,5) equation with an irregular left-hand boundary,* $r = 1.75$.

left-hand boundary and its mirror image for the right-hand boundary. If a fully two-dimensional method was to be developed, it would require the development of many different equations, since the geometry of the boundary relative to the grid being used leads to many different grid spacings not being the "uniform" size. For example, on a square region that has different grid spacings at the edges to the "uniform" spacing in the interior, we require one equation which has a variable grid spacing in one spatial direction, for general use next to a boundary, three more equations which are rotations of the first equation, for use next to the other three boundaries, as well as four more equations which have a variable grid spacing in both spatial directions, for use in the corners of the region. This is a total of eight equations, and it should be noted that the square geometry has lead to a much smaller number of equations than would be needed for a general irregular region.

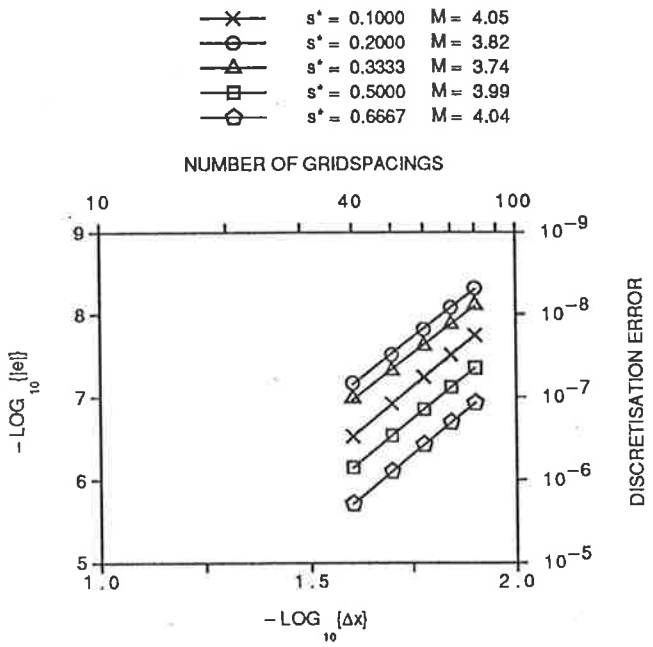


Figure 5.6: *Error vs grid spacing graph for the two-dimensional problem using the variable grid (1,5) equation in an LOD fashion,* $r = 1.25$.

The numerical results for the use of the variable grid, third-order (1,5) and (3,3) equations, shown in Figures 5.6 to 5.9, were generated by using the same test problem as for the general two-dimensional problem, but with the boundaries at $x = x_L$ and

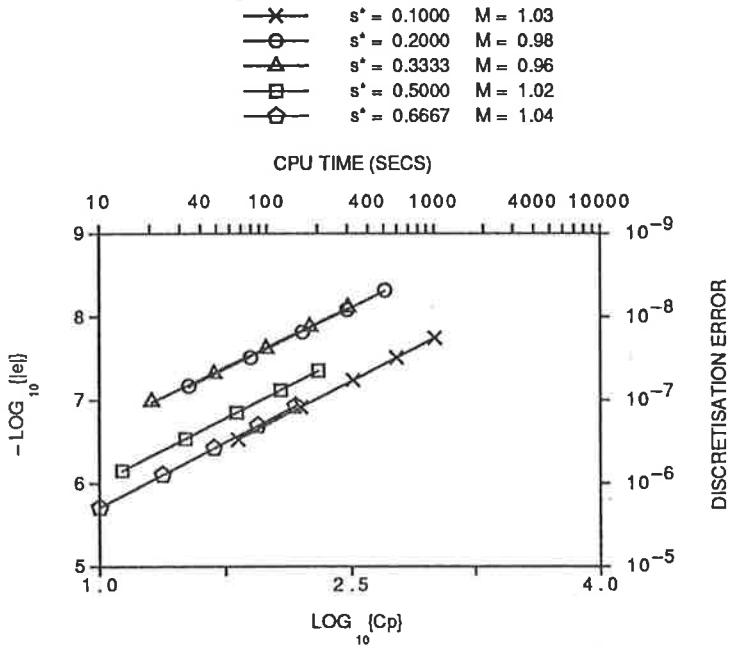


Figure 5.7: *Error vs CPU time graph for the two-dimensional problem using the variable grid (1,5) equation in an LOD fashion,* $r = 1.25$.

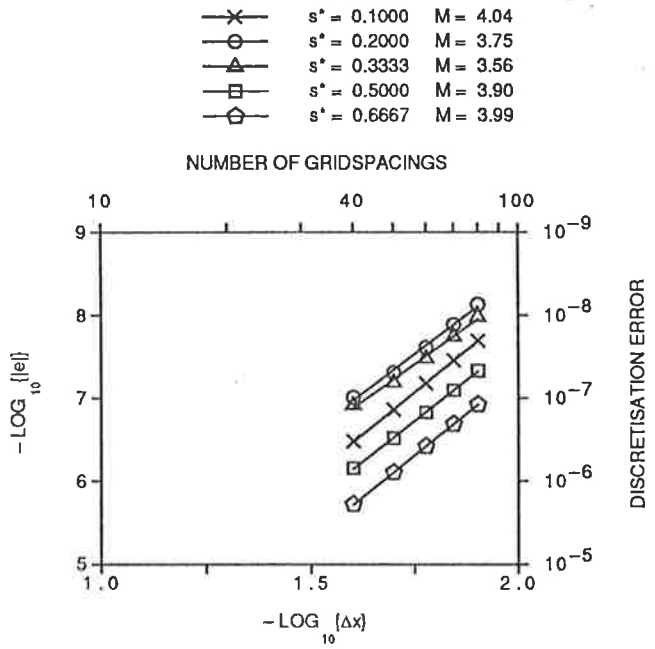


Figure 5.8: *Error vs grid spacing graph for the two-dimensional problem using the variable grid (1,5) equation in an LOD fashion,* $r = 1.75$.

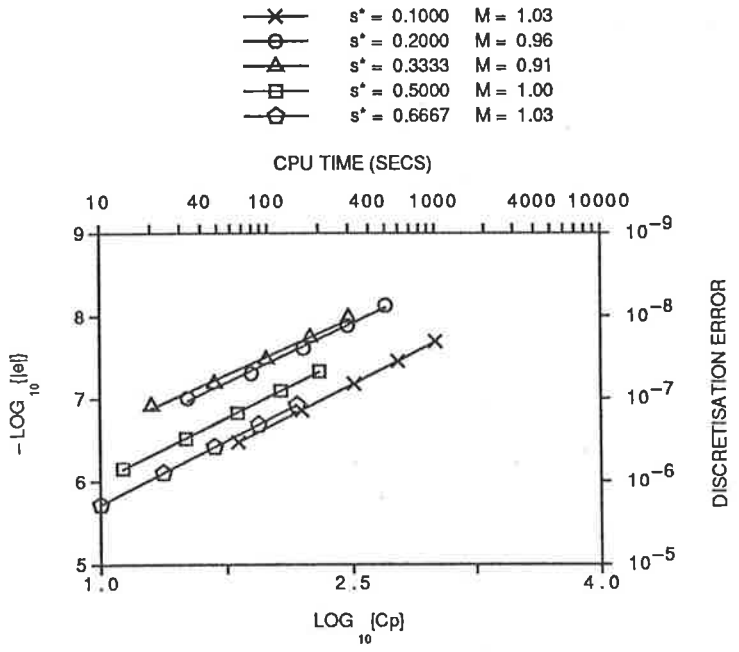


Figure 5.9: *Error vs CPU time graph for the two-dimensional problem using the variable grid (1,5) equation in an LOD fashion,* $r = 1.75$.

$y = y_L$ where $0 \leq x_L < \Delta x$ and $0 \leq y_L < \Delta y$. The other boundaries were unchanged, and the results are still taken from the point $(0.2, 0.2)$ at time $T = 2$. As in the case of the one-dimensional test, this problem was run with two different values of r, to judge the effects of the value of r on the results.

The results show that the order of accuracy, as judged by the slopes of the lines, has only been decreased slightly due to the third-order nature of the variable grid equations used next to the boundary. It is worth noting, however, that the actual accuracy of the solutions generated is still the same as that for the uniform grid case. The omission of values for $J = K = 20$ and $J = K = 30$ on these graphs was necessary to present the correct overall impression of these results; the actual points that were removed gave errors that were significantly *more* accurate than expected. For instance, the error for $J = K = 30$ and $s = 0.2$ was approximately $10^{-8.5}$, rather than the $10^{-6.5}$ that would be expected from the rest of the results. The cause of such anomalous results has not been determined, and requires extra work beyond what is done here.

For a general curved boundary, much the same techniques can be applied, except that

along each one-dimensional space line we must keep track of the value of r required at each end. The major problem introduced by this extension, however, is that the boundary can no longer be split into two halves, each aligned in one of the spatial directions. Thus all the boundary values must be found at the intermediate time level, rather than only some, and these values cannot be found from the finite-difference equation since the boundary is neither straight, even over the length of the computational stencil, nor aligned in one of the primary spatial directions. This problem can be overcome either by careful application of the given boundary condition, or the use of extrapolation from the interior values. If the boundary condition is separable into parts involving each of the space variable separately, as is the case with our test problem since it is just the product of a Gauss peak in each of the x and y directions, then this form can be used to find the boundary conditions. This is done by substituting different values of t into each part of the equation, to reflect the fact that at the intermediate time level the solution has diffused in only one spatial direction. It should be noted here, however, that an analytic form of the boundary condition is not usual in practical situations; in most cases the boundary conditions are given as a set of numerical data, so this method cannot be used. This method is used here, however, since this allows for a better analysis of the results.

The other alternative for modelling the boundary at the intermediate time level is to use either some other one-dimensional finite-difference equation or extrapolation to find the boundary value from the known values in the interior of the region. This technique is undesirable as it tends to limit the von Neumann stability of the final scheme, or in the worst cases makes it totally unstable. Nevertheless, the form of some boundary conditions may dictate that this is the best technique to be used on a particular problem. Since the boundary condition is specified in a closed analytic form for our test problem, it will be used to approximate the values on the boundary at the intermediate time level.

It is also worth noting that in this case, there is no way that the value of r can be kept constant; indeed the value varies for each one-dimensional time line that the finite-difference method is applied to, and in the general case, the values of r at the

"left" and "right" boundaries will also differ. This being the case, the graphs of error vs grid spacing and CPU time can no longer be expected to be straight lines, since the leading coefficient of the modified equivalent equation depends on s and r. The actual numerical results from running the (1,5) LOD scheme on a circular region, namely the circle $(x - 1/2)^2 + (y - 1/2)^2 = 1/4$, using the usual two-dimensional Gauss peak to provide initial and boundary conditions, are shown in Figures 5.10 and 5.11.

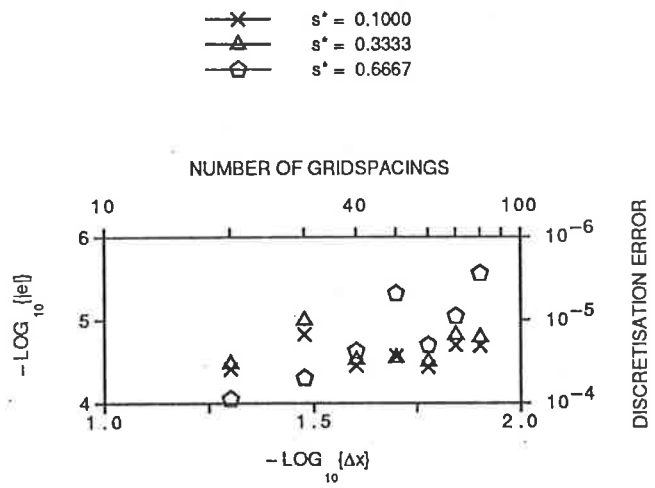


Figure 5.10: *Error vs grid spacing graph for the circular boundary problem using the variable grid (1,5) equation in an LOD fashion.*

The most obvious feature of these graphs is that they are no longer straight lines of slope four, and indeed that fact that the solutions obtained with $J = K = 80$ are not much more accurate than those for $J = K = 20$ is somewhat disappointing. Also note that while the absolute accuracy is still quite good, being approximately four decimal digits, the accuracy of the solution has fallen quite markedly from the case where r was fixed. These results show that while the (1,5) LOD technique is a good method for solving two-dimensional problems, further investigation needs to be done in the case where the boundary of the solution domain is curved or irregular. In particular, most of the problems evident in the results for the circular region arise from the necessity to evaluate all the points on the boundary at the intermediate time level. Clearly, further work is necessary to determine the best way that this can be done, hopefully without compromising either the accuracy or the stability of the method as a whole.

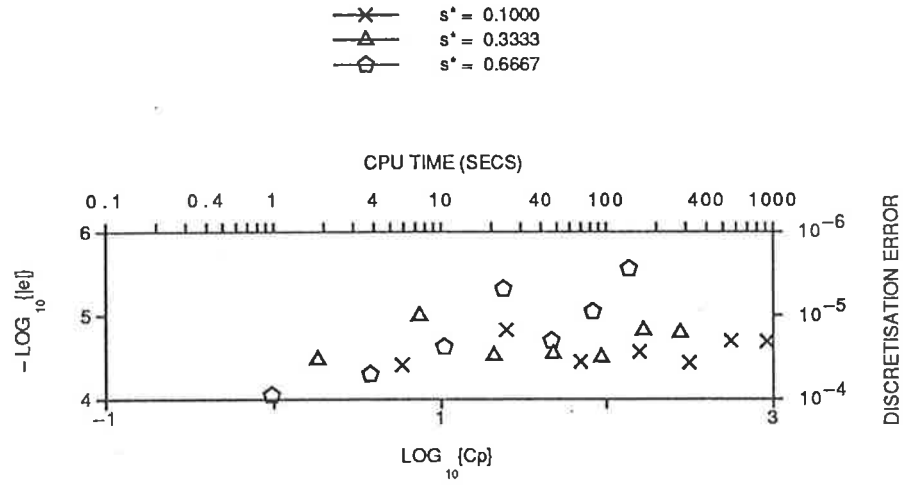


Figure 5.11: *Error vs CPU time graph for the circular boundary problem using the variable grid (1,5) equation in an LOD fashion.*

5.3 Summary

It has been seen that the problems arising from the region not being perfectly rectangular, which means that there is not necessarily a grid point exactly on the boundary with a uniform grid, can be overcome by using variants of our existing finite-difference equations. These new equations incorporate the fact that the grid is non-uniform next to the boundary, and are in general one order less accurate than their uniform grid counterparts. Note that many of the properties of the uniform grid equations are still true of the variable grid versions, since the latter must reduce to the former in the special case $r = 1$.

The numerical results for the two-dimensional problem, which is the case for which an irregular boundary is most likely to occur in practice, show that the accuracy of the solution on a square grid, with different grid spacings at the boundaries is close to that obtained using the LOD scheme on a uniform grid, despite the theoretical accuracy being one order less, and the actual order being slightly less than four in general. The results on a circular region indicate that other considerations, which sometimes tend to be considered as only minor factors in the solution process, such as the modelling the boundary values at the intermediate time step, can be of great

importance in generating an accurate final solution, and further investigation of such factors is needed to produce more accurate solutions in this case.

Nevertheless, it is thus seen that the locally one-dimensional solution technique is useful in practice for solving such problems with irregular boundaries, where a fully two-dimensional method would be very much more complicated to develop and implement, since it would require many different equations to deal with all the different combinations of variable grid spacings which are possible.

Chapter 6

Conclusions

It has been shown that the "best" way to solve the diffusion equation by finite-difference techniques, in either one or two spatial dimensions, is dependent on the circumstances requiring the solution. In general, the higher the theoretical order of accuracy of the solution technique the better, but constraints on CPU time available or the minimum required resolution may dictate the use of a less accurate method which has a larger von Neumann stability range.

The basis of analysis of the various finite-difference equations considered here is the modified equivalent equation approach, developed from the 1974 work of Warming and Hyett. This allows direct and simple comparison of the errors associated with the equations as well as providing a means to develop more accurate equations. This is achieved by incorporating free weights into a generalised equation, then eliminating the dominant error terms from the modified equivalent equation by suitable choice of values for these weights, to give more accurate finite-difference equations. Since the generation of equations and their corresponding modified equivalent equations is a time consuming, tedious and error-prone task by hand, a suite of computer programs, written in Pascal, has been developed to carry out the mechanical operations. This results in large savings in the time required to develop new finite-difference equations.

For the one-dimensional problem, the most accurate method found is the sixth-order (1,5,1) equation, which is also von Neumann stable over the range $s < 0.51...$, which

is much better than the other sixth-order methods considered. While even higher order equations can be developed, these are all implicit in nature, which means that there is the extra overhead of the solution of a set of linear algebraic equations to be solved at each time level which must be considered. In addition to this, such equations have very restrictive von Neumann stability ranges, so increasing the value of s somewhat to offset the CPU time overhead is not possible. Achieving higher accuracy by increasing the width of the computational stencil is not practical either, since the problems near the boundaries become correspondingly greater, with more points near the boundary having to be calculated by some alternative means, preferably without a loss of accuracy from this process.

The fourth-order (1,5) equation is also very useful for obtaining a solution in cases where the sixth-order equations are unsuitable. It is simpler to implement than the (1,5,1) equation, since it uses values from only two time levels and so does not require a different technique for starting. Also, the boundary problems can be overcome to the correct order of accuracy by use of Crandall's equation, without the loss of accuracy imposed on the (1,5,1) equation by the use of the sixth-order (1,3,3) equation next to the boundary.

If a boundary condition is given as a derivative (a Neumann condition) rather than a known value (the Dirichlet condition), much the same techniques can still be used, but extra effort is required to accurately determine the values at grid points on the boundary. This involves extra approximations, and somewhat different techniques for handling the problems associated with spatially wide computational stencils must also be used. All this leads to a decrease in accuracy which shows up very clearly in the numerical results. The worst affected method was the (1,5,1) equation, for reasons which are not clear, but may be connected with it using both values from three time levels and a spatially wide stencil. Further investigation of this problem is required to obtain a better understanding of the problems in this case. The rest of the "good" method for the Dirichlet case, namely the fourth-order (1,5) and (1,3,3) and the sixth-order (1,3,3) equations, still produce quite accurate answers for the Neumann boundary condition. Which of these methods is the "best" is again dependent on the

circumstances of the solution.

For the two-dimensional problem, it has been found much harder to develop high-order finite-difference equations. This is mainly due to the existence of extra "cross-derivative" error terms in the modified equivalent equation which must be also be eliminated in order to generate high-order equations. These extra error terms force an increase in the number of weights which must be used to eliminate them, which greatly increases the complexity of the finite-difference equations. This extra complexity prevented the development of a sixth-order equation with the computing resources available, although with the use of a larger, more powerful computer such an equation may be developed successfully.

Many of the features discovered in the one-dimensional problem carry over into the two-dimensional case. In particular, using two-level explicit computational stencils limited to three grid points in each spatial direction, it is not possible to develop a finite-difference equation which is more than second-order accurate in general. Likewise, for computational stencils which extend over five grid points spatially, the best possible equation is fourth-order accurate.

Three-level methods were in general disappointing, since all the equations that could be analysed were no better than fourth-order, rather than sixth-order as was obtained in the one-dimensional case. This was due, however, to the complex nature of the weighted finite-difference equations, in particular that for the (1,13,9) computational stencil, which cannot be analysed on the available computing resources.

Implicit methods for the two-dimensional problem, as was expected, require enormous amounts of CPU time, due to the requirement of solving a set of linear algebraic equations at each time level. The problem is that the although the coefficient matrix of the equation set is banded, the bandwidth is directly proportional to the number of equations, rather than fixed as was the case in one dimension. This massive CPU time overhead makes implicit equations totally impractical to use, so the only equation seriously developed was the fourth-order (9,9) equation, which can be used to solve the boundary problems associated with spatially wide explicit equations like the (1,13)

and (1,21) equations. Used in this fashion, the bandwidth of the set of equations to be solved is fixed at three, so the CPU time overhead is reduced to quite acceptable levels.

"Locally one-dimensional" (LOD) techniques, which are based on splitting the two-dimensional equation into two one-dimensional problems then using the established one-dimensional solution techniques on each of them, avoid the need for the more complex fully two-dimensional equations. It has been shown that these techniques require less CPU time to run than full two-dimensional equations, and the accuracy of the solutions obtained is similar. The one problem encountered is that finite-difference equations that use values from three time levels cannot be used, since the time levels generated here involve diffusion in only one and both spatial directions alternately. This prevents three-level methods from generating accurate results, and so the preferred solution technique here is the fourth-order (1,5) equation.

Overall, the LOD techniques are seen to offer somewhat better prospects than the fully two-dimensional equations for solving the two-dimensional diffusion equation, since they generate solutions of the same accuracy in less CPU time, and are much simpler to develop and implement. Extra work needs to be directed at solving the problems associated with using three-level equations in this fashion, which may be the best way to obtain a sixth-order solution technique for the two-dimensional problem. The alternative is to develop the sixth-order (1,13,9) equation to the point where is can be implemented in practice, since this will undoubtedly give more accurate answers than the fourth-order techniques presented here.

Alternating direction implicit (ADI) methods provide another good way to obtain accurate solutions to the two-dimensional diffusion equation in a moderate amount of CPU time. These equations are typically of a fairly low order of accuracy when considered in isolation, but if they are considered after two applications, one in each spatial direction, it is found that some of the lower-order error terms have cancelled each other out. The development of such equations needs to take this into account, since it provides a convenient way to generate spatially compact high-order equations. Such an equation was shown to be the only fourth-order equation that was found that

did not require a spatially wide computational stencil, which immediately removes the problems near the boundaries. It is found, however, that since the solution process involves the solution of many sets of linear equations at each time level, the amount of CPU time required to find a solution is somewhat more than that needed by the explicit equations. This amount of CPU time is, however, still moderate and only a fraction of the CPU time required by the fully implicit methods, so stable ADI techniques should be considered favourably when choosing a solution technique.

Irregular boundaries, such as may be found when modelling physical regions, can be dealt with either by mapping the irregular region onto a rectangular region, when the solution techniques already discussed can be applied, or else by developing special finite-difference equations that take the irregularity into account. The incorporation of a variable grid spacing is found to reduce the formal order of accuracy of the finite-difference equations by one, so the optimal (1,5) and (3,3) equations are now third-order instead of fourth. Numerical tests have shown, however, that this has only a slight impact on the solutions generated by using these equations next to the boundaries. Give this, these equations can be applied in a locally one-dimensional fashion to generate a solution to the two-dimensional diffusion equation on an irregular region. This is much simpler than developing a fully two-dimensional equation that must incorporate a different variable grid spacing in each spatial direction, especially since there are only two (mirror-image) forms of the one-dimensional equations (for the left and right-hand boundaries), rather than the many different equations required of the two-dimensional equation. This is another illustration of the practicality of the LOD approach to solving the two-dimensional problem.

Overall, the modified equivalent equation approach has proven to be extremely useful and practical, both to analyse existing finite-difference equations and to develop new and more accurate ones to solve both the one and two-dimensional linear diffusion equations with constant coefficients. Several very accurate equations have been developed, discussed and compared on the basis of both accuracy of solution and the CPU time required to generate that solution. There is, however, no clear "best" method for solving any of the problems discussed here; the method to be preferred must de-

pend on constraints such as the available computing capacity and the required spatial resolution and accuracy.

Further work remains to be done in several areas, such as more accurate techniques for the Neumann boundary condition, using three-level equations as LOD equations, the development of more accurate ADI methods and also the development of sixth-order fully two-dimensional equations. The extension of this work into the variable coefficient diffusion equations in both one and two dimensions should also be examined, since these equations provide a better approximation to many of the physical processes that we set out to model.

Appendix A

User Guide for the FDE Development Programs

A.1 Introduction

This appendix describes a system for the development of accurate finite-difference equations (FDEs) to solve the one or two dimensional linear advection-diffusion equations with constant coefficients (sometimes referred to as the transport equations).

The two-dimensional form of this equation is

$$\frac{\partial \hat{\tau}}{\partial t} + u\frac{\partial \hat{\tau}}{\partial x} + v\frac{\partial \hat{\tau}}{\partial y} - \alpha_x \frac{\partial^2 \hat{\tau}}{\partial x^2} - \alpha_y \frac{\partial^2 \hat{\tau}}{\partial y^2} = 0 \qquad \text{(A.1.1)}$$

where u, v, α_x and α_y are constants. This equation can be used to describe such things as the spread of pollutant in a stream or heat transfer in a solid object (in which case u and v are both zero). Two special cases of this equation occur, firstly when $\alpha_x = \alpha_y = 0$, which leads to the advection equation, and secondly when $u = v = 0$, which is the diffusion equation.

The equation (A.1.1) can be solved numerically by finite-difference techniques. Without loss of generality we can assume that (A.1.1) is written in a non-dimensional form, such that the space domain is $[0, 1]$ in both the x and y directions, while u, v, α_x and α_y refer to non-dimensional quantities. The space domain is then divided up into a

rectangular grid, with J grid spacings each of length Δx in the x direction, and K grid spacings of length Δy in the y direction. Alternative grid geometries, such as triangular and hexagonal are possible, but the programs described in this appendix deal only with the rectangular case. The equation (A.1.1) can then be solved on this grid by starting from a known initial state (at time $t = 0$), then using this information to approximate the state at time $t = \Delta t$, and so on until the desired time $t = T$ is reached.

This stepping process is achieved by approximating the derivative terms in (A.1.1) by combinations of approximate values of $\hat{\tau}$ at the grid points defined above. For convenience, the grid point $(j\Delta x, k\Delta y)$ at time $n\Delta t$ is referred to as the (j, k, n) grid point, and the approximate value of $\hat{\tau}$ at this point is $\tau^n_{j,k}$. When the derivative terms are approximated and rearranged into useable formulae, several dimensionless ratios are found to occur. These are denoted by:

$$c_x = \frac{u\Delta t}{\Delta x} \quad , \quad c_y = \frac{v\Delta t}{\Delta y} \quad , \quad s_x = \frac{\alpha_x \Delta t}{(\Delta x)^2} \quad , \quad s_y = \frac{\alpha_y \Delta t}{(\Delta y)^2} \tag{A.1.2}$$

As an example of this, consider the diffusion equation

$$\frac{\partial \hat{\tau}}{\partial t} - \alpha_x \frac{\partial^2 \hat{\tau}}{\partial x^2} - \alpha_y \frac{\partial^2 \hat{\tau}}{\partial y^2} = 0, \tag{A.1.3}$$

and the finite-difference approximations

$$\frac{\partial \hat{\tau}}{\partial t} = \frac{\tau^{n+1}_{j,k} - \tau^n_{j,k}}{\Delta t} + O\{\Delta t\}, \tag{A.1.4}$$

$$\frac{\partial^2 \hat{\tau}}{\partial x^2} = \frac{\tau^n_{j-1,k} - 2\tau^n_{j,k} + \tau^n_{j+1,k}}{(\Delta x)^2} + O\{(\Delta x)^2\}, \tag{A.1.5}$$

$$\frac{\partial^2 \hat{\tau}}{\partial y^2} = \frac{\tau^n_{j,k-1} - 2\tau^n_{j,k} + \tau^n_{j,k+1}}{(\Delta y)^2} + O\{(\Delta y)^2\}. \tag{A.1.6}$$

If these forms are substituted into (A.1.1) and rearranged, the resulting equation is

$$\tau^{n+1}_{j,k} = s_x \tau^n_{j-1,k} + s_y \tau^n_{j,k-1} + (1 - 2s_x - 2s_y)\tau^n_{j,k} + s_x \tau^n_{j+1,k} + s_y \tau^n_{j,k+1} \tag{A.1.7}$$

after the terms of $O\{\Delta t, (\Delta x)^2, (\Delta y)^2\}$ are dropped. This equation can then be used to find values of τ at one time level from the values at the previous time level.

In order to check that this difference scheme is actually solving the correct partial differential equation (PDE) (called being *consistent*), and also to find the formal error

involved in this approximation, we now find the corresponding modified equivalent partial differential equation (MEPDE). This is done by expanding all the terms in (A.1.7) as Taylor Series about the (j,k,n) grid point, then differentiating the resulting equation and adding the result back into the original equation repeatedly to remove all the derivative terms involving $\partial/\partial t$, except for the term $\partial\tau/\partial t$.

In the case of the example above, the resulting MEPDE is

$$\frac{\partial\tau}{\partial t} - \alpha_x\frac{\partial^2\tau}{\partial x^2} - \alpha_y\frac{\partial^2\tau}{\partial y^2} + E(\tau, s_x, s_y, \Delta x, \Delta y) = 0 \qquad \text{(A.1.8)}$$

where

$$\begin{aligned} E(\tau, s_x, s_y, \Delta x, \Delta y) &= -\frac{\alpha_x(\Delta x)^2}{12}(1-6s_x) - \frac{(\Delta x)^2(\Delta y)^2}{\Delta t}(s_x s_y) \\ &- \frac{\alpha_y(\Delta y)^2}{12}(1-6s_y) + O\{(\Delta x)^4, (\Delta y)^4\}. \qquad \text{(A.1.9)} \end{aligned}$$

Thus it can be seen that (A.1.7) is consistent with the two-dimensional diffusion equation (A.1.3) with an error of $O\{(\Delta x)^2, (\Delta y)^2\}$, as all the other errors terms in E are much smaller than the leading error terms given in (A.1.9). A method with errors of this form is called second-order accurate; in general if the leading error terms involve $(\Delta x)^a(\Delta y)^b(\Delta t)^c$ where $a+b+2c=d$ then the method is said to be of order d.

As can be readily appreciated from the above example, working out the finite-difference equation that corresponds to a given set of approximations to the derivative terms in (A.1.1) and then finding the corresponding MEPDE can be a very long and tedious process in all but the very simplest of cases. To overcome these problems (and the associated errors which invariably creep into hand calculations), a set of computer programs has been developed to perform these mechanical tasks, thus removing most of the possibility of errors, and saving large amounts of time. The relationship between the programs themselves and the files they use and produce is shown in Figure A.1.

The current implementation of this set of programs is written in Vax Pascal V3.5 and runs under VAX/VMS. Despite this, the programs are written in almost standard Pascal. The only VAX extensions used are exponentiation, the use of the "_" character in identifier names and the use of error trapping and file name defaulting for opening

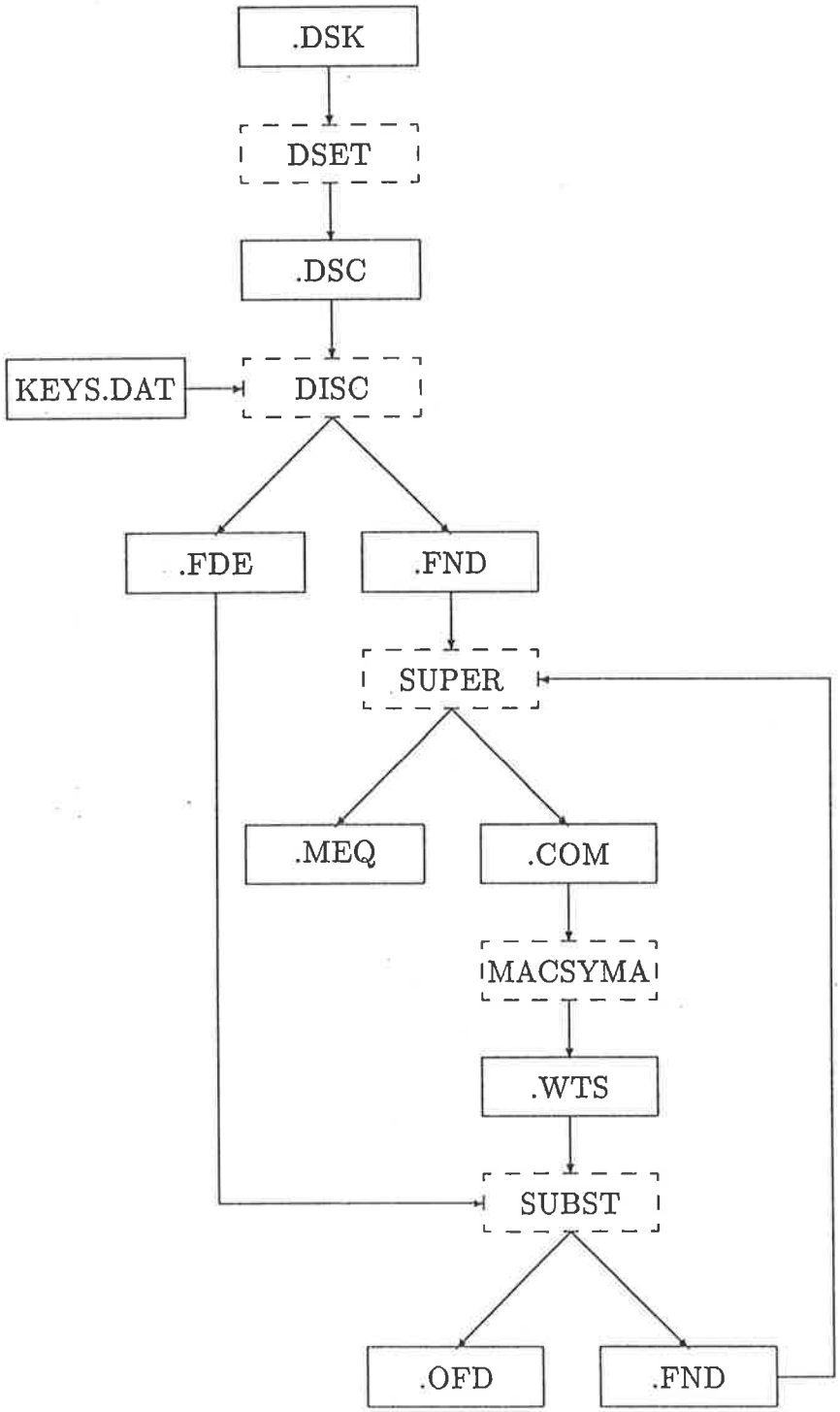


Figure A.1: *Relationship between programs and their input and output files. The program names are explained in the text.*

files. All these are reasonably common extensions to Pascal, and things like name defaulting and "_" characters can be removed without changing the program behaviour significantly. Thus it is a reasonably simple matter to get these programs to run on other machines and/or operating systems. This has in fact been done in the case of SUPER, which was converted to run on a micro-computer under CP/M-80, using Turbo Pascal V2. It has also been successfully ported to the Apple Macintosh.

All input and output files for the set of programs are ordinary text files that may be changed with a text editor, and all information is passed from one program to the next by means of these files. Thus each program can be run independently of the others; there is no need to run any other programs if output from one only is needed, and input files for that program can be created using a text editor (eg. Ludwig on University of Adelaide computers).

The programs described here fall into two categories. The first category includes those designed for the one-dimensional advection-diffusion equation, which have the program names referred to in this document and are located, along with the required data files, in the directory `D2:[KHAYMAN.PHD.MODPDE.DVL_SYS]` on Vax E. The second category of programs includes those for use with the two-dimensional advection-diffusion equation, which are located in the directory `D2:[KHAYMAN.PHD.MODPDE.DVL_SYS.TDMOD]` and have the prefix "2D" before the names given in the text.

The development process is started by selecting the desired computational stencil for the method, then choosing how to difference each derivative in the equation over that stencil. This may involve several weights, either because it is desired to split a derivative approximation between two or more different grid points and/or to allow for optimisation of the resulting method by choosing optimal values for the weights. These optimal values are usually chosen to remove the leading error terms from the MEPDE corresponding to the method, although they can also be used to achieve other desirable features for the method, such as an increase in the numerical stability region. The program DSET is used to convert the desired differencing and weighting into an input file suitable for DISC to read. From this input file, the program DISC is used to find the weighted form of the finite-difference equation (FDE) for the given

differencing.

Having done this, the error involved in using the finite-difference equation must be determined. This is done by using the program SUPER to find the MEPDE for the method. The input file for this program is generated by the program DISC. From the MEPDE, the aim is usually to eliminate as many of the leading error terms as possible by a suitable choice of the values of the weights, although, as noted above, this may not always be the case. In some cases, one weight may be required to ensure that the method involves no artificial diffusion. The optimal weight values may be found either by hand, which becomes difficult once there are more than one or two weights, or using a symbolic manipulation package such as MACSYMA.

The values of the weights, as well as the weighted finite-difference equation, are then used by the program SUBST to find the optimal finite-difference equation (ie. the one with the highest formal order) corresponding to the original differencing. If it is necessary to check the MEPDE of the optimal scheme, an input file to SUPER is also generated by SUBST. This is also useful in cases where two schemes of the same order of accuracy are to be combined to give a scheme of higher order.

All the programs have a similar input format. Each input file, and output file, is prompted for in turn, and the user can enter the corresponding file name. Values that are given in square brackets (``[ ]'') at the end of the prompt indicate default values that are added to whatever the user specifies to generate the final file name to be used. Note that these defaults are overridden if the user specifies the field in question, and that all files are assumed to be in the current default directory unless otherwise specified by the user. The only exception to this last rule is the KEYS file for the program DISC, which is assumed to be in the same directory as the program DISC.

Values in parentheses (``( )'') just prior to the default strings are defaults if the user merely presses "RETURN" in response to the prompt (ie. enters no file name). This is usually "(keyboard)" for an input file, which indicates that the input from that file will be prompted for from the user's terminal, and "(screen)" for an output

file, which means that output to that file will appear on the terminal screen. A value of "(none)" means that the file will not be generated unless the user gives the file a name. If there is no default value given, then the file is required by the program, and thus must exist (once the defaults mentioned above have been added).

If a file cannot be opened or created with the name as specified, then the file is prompted for again in most cases. In the case of an input file, the user should check that the file exists and is accessible in the specified directory (or the current one if none was specified). For an output file, the user should check that the file can be created in the specified (or current) directory and that there is sufficient available disk quota available to create the file.

The last section of this appendix describes several programs that, while not being part of the system for developing accurate finite-difference *equations*, are essential for creating practically usable finite-difference *methods*. They were written in Vax Fortran by Peter Steinle and are used for checking various stability characteristics of a finite-difference equation. Details of their use and input file format, which differs from the rest of the programs described in this document, are given in Section A.5.

Except for the program DISC, which was written by Mark Rankovic, and the Fortran stability characteristic programs written by Peter Steinle, all the programs described here were developed and written myself. They represent a large but worthwhile investment of effort, since there is an enormous saving of time possible with their use, especially in the more complicated two-dimensional cases. The most important of the programs is SUPER, which actually takes a finite-difference equation and determines its modified equivalent equation; the listing for the one-dimensional form of this program is given in Appendix C.

Example

To illustrate the use of each program, an optimal (1,5,1) method based on the stencil shown in Figure A.2 will be developed to solve the one-dimensional diffusion equation

$$\frac{\partial \tau}{\partial t} - \alpha \frac{\partial^2 \tau}{\partial x^2} = 0. \tag{A.1.10}$$

At the end of each section, this stencil will be used to illustrate the use of the program discussed in that section. Where appropriate, the output files generated by the programs are listed as well.

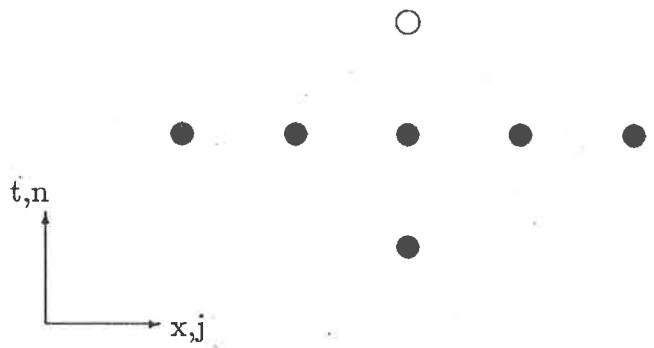


Figure A.2: *The (1,5,1) Computational Stencil*

The (1,5,1) stencil allows two weights to be used, as follows:

$$\left.\frac{\partial \tau}{\partial t}\right|_j^n \approx \theta \times [\text{ FT at } (j,n)] + (1-\theta) \times [\text{ CT at } (j,n)], \tag{A.1.11}$$

$$\left.\frac{\partial^2 \tau}{\partial x^2}\right|_j^n \approx \varphi \times [\text{ CS3 at } (j,n)] + (1-\varphi) \times [\text{ CS5 at } (j,n)] \tag{A.1.12}$$

where CS3 is used to denote a three-point centred space difference approximation about the specified point, CS5 represents a five-point centred space approximation, FT represents a forward time approximation and CT represents a three-point centred time approximation.

A.2 Differencing a Partial Differential Equation

The first step towards finding an accurate solution method for the equation (A.1.1) is to substitute finite-difference forms for the derivative terms in the equation, then simplify the result to give the corresponding finite-difference equation. The program DISC is designed to do this, based on a set of pre-defined differencings of various derivative terms. The input format for this program however is not particularly easy to work with, so the program DSET was written to convert an easily created file that describes the differencings and weightings to be used into a form suitable for input into DISC.

Generating the input file for DISC

```
Program Name:   DSET
Author:         Ken Hayman
Date:           November 1986
```

This program is designed to make it easier to create input files for DISC. DSET takes an input file which describes the weighting to be used. The format of this file consists of a one line title to describe the equation to be developed, followed by, for each differencing to be used, an expression in parentheses, which may extend over several lines, followed by "* KEY(n)", where n is the number of the discretisation key, obtained from the tables given in Appendix B. At the end of the file, there must be a line that consists of "= 0", to emphasise that the equation is written in LHS= 0 format. Note that the coefficient terms must be enclosed in parentheses, regardless of how simple they are. Thus "(1) * KEY(13)" must be written instead of just "KEY(13)". Also note that no sign is allowed before the opening parenthesis of the expression, so "(-(<expression>)) * KEY(13)" is the correct form for entering a negative coefficient.

The output from this program is a file suitable for input into DISC. It should be realised that this file itself is not the shortest possible representation of the differencing, as

collection and simplification of like terms has not been done. However, this makes no difference to the operation of DISC, as this simplification is done by DISC itself as it processes its input file.

Example

The discretisation keys for the example problem can be obtained from a listing like that given in Appendix B below. This listing contains all the differencings that are available in the standard key file, and should be updated if any new differencings are added to this file. This file of standard differencings is called KEYS.DAT, and is kept in the same directory as the program DISC. If the required differencing for a method is not listed, then it is necessary to create a customised file of differencings (called a *key file*), which contains the desired new differencing, as well as any of the standard ones that are used by the method. The differencings in the new file are given key numbers starting from 1 and increasing sequentially. Care should be taken to correctly calculate the key numbers of the differencings in the new file, as they will be quite different to those in the standard file.

For the current example, however, the standard key file is sufficient, since the method uses only the following differencings:

$$\begin{aligned} \text{FT} &: \text{Key } 3 \\ \text{CT} &: \text{Key } 13 \\ \text{CS3} &: \text{Key } 41 \\ \text{CS5} &: \text{Key } 44. \end{aligned} \tag{A.2.1}$$

From these key numbers, and the weightings (A.1.11) and (A.1.12) given above, the input file for DSET can be formed. The file is

```
1-D Diffusion: Weighted (1,5,1) Method

(Theta) * Key(3)
```

```
(1-Theta) * Key(13)

(-Alpha*Phi) * Key(41)

(-Alpha*(1-Phi)) * Key(44)

= 0
```

(remembering that the equation must be in LHS = 0 form). DSET takes this file and produces an input file suitable for DISC to read and evaluate the corresponding FDE.

Finding the Finite Difference Equation

```
Program Name:   DISC
Author:         Mark Rankovic
Date:           September 1985
                Modifications since by Ken Hayman
```

This program takes a differencing on a given computational stencil and produces the corresponding finite-difference equation, as well as an input file for the program SUPER.

In order to make the program as flexible as possible, the differencings that it "knows" about are read in from a file. This file has the following format:

<x-space offset> <y-space offset> <time-offset>
<numerator> <denominator> <delta-x> <delta-y> <delta-t>

where <x-space-offset> is the spatial offset in the x-direction of the grid point from j, <y-space-offset> is a similarly defined offset in the y-direction and <time-offset> is the offset of the grid point from time level n. The terms <numerator> and <denominator> form a multiplying factor, and <delta-x>, <delta-y> and <delta-t> are the powers of Δx, Δy and Δt respectively. In the the one-dimensional case,

the numbers <y-space-offset> and <delta-y> are absent from the file. Note that the line structure of the file is important, although individual items on the lines may be separated by an arbitrary number of spaces.

This information is repeated for each grid point involved in the differencing, and each differencing is terminated by a line containing a single asterisk (*). This asterisk is required even after the last differencing in the file. The differencings in the key file are referred to by number, with the first differencing in the file being number one. To use this system effectively, a printed copy of the differencings in the file and their corresponding numbers should be kept, as the numbers are not readily apparent from the file itself, particularly in the two-dimensional version, where there are currently more than 450 different keys. Copies of the key files for the one and two-dimensional cases are given in Appendix B.

The program DISC takes as input both the key file described above and a file that describes the differencings to be used. The latter file provides information about the differencings to be applied to the various derivative terms in the partial differential equation. This file can either be generated by hand, or produced by the program DSET, which is described above. The first line of this file is the name of the method (up to 80 characters long), then on successive lines are the number of weights used, and their names (which are converted internally to upper case). Following this, and again with one item per line, are the (integer) numerator and denominator (with any sign included in the numerator rather than the denominator) and the powers of u, α (for the one-dimensional case) or u, v, α_x and α_y (for the two-dimensional case) and the weights (if any) which multiply the term. Following this is the number of the discretisation to be applied, which is obtained from the key file listing, then a "Y" if there is another term to be input or a "N" if this was the last one. Extra terms are entered in exactly the same format, starting with the numerator and ending with the "Y"/"N" response as appropriate.

The program takes the input data and finds the corresponding finite-difference equation by combining together and simplifying all the contributions at each grid point by adding terms with all the same powers together into one term and eliminating any

term whose coefficient becomes zero. Once all the differences from the input file have been processed, the resulting equation can be written out without any further changes.

The program produces two output files from the simplified data. The first of these is a file that gives the resulting finite-difference equation in an easily understood form, which can readily be used to implement the method. This file is also used as input to the program SUBST, described later, and so should be kept. The second file is an input file to the program SUPER, used for finding the MEPDE corresponding to the difference equation. This file is an optional output, and may be suppressed by just pressing RETURN when prompted for its name. If it is produced, it will prompt for the order of the highest derivative SUPER should work with. This value should be chosen with care; restrictions on its value are explained later in Section A.3.

Example

For the example problem given at the end of Section A.1, the finite-difference equation generated by DISC, using the input file generated by DSET, is as follows:

```
FDE for : 1-D Diffusion: Weighted (1,5,1) Method

(   1/2 * THETA
  + 1/2 ) * TAU(n+1,j)

(   1/12 * S
  - 1/12 * S * PHI ) * TAU(n,j-2)

(   1/3 * S * PHI
  - 4/3 * S ) * TAU(n,j-1)

( - 1 * THETA
  - 1/2 * S * PHI
  + 5/2 * S ) * TAU(n,j)
```

```
(   1/3 * S * PHI
  - 4/3 * S ) * TAU(n,j+1)

(   1/12 * S
  - 1/12 * S * PHI ) * TAU(n,j+2)

( - 1/2
  + 1/2 * THETA ) * TAU(n-1,j)

= 0
```

which corresponds to the finite-difference equation

$$\begin{aligned} 6\{\theta+1\}\tau_j^{n+1} &= s\{\varphi-1\}(\tau_{j-2}^n+\tau_{j+2}^n)+4s\{4-\varphi\}(\tau_{j-1}^n+\tau_{j+1}^n) \\ &+ 6\{\varphi s+2\theta-5s\}\tau_j^n+6\{1-\theta\}\tau_j^{n-1} \end{aligned} \tag{A.2.2}$$

on rearrangement into the more usual form.

A.3 Finding the Modified Equivalent Equation

```
Program Name:   SUPER (or SUPERSLOW)
Author:         Ken Hayman
Date:           March 1984 (original program FINDMOD)
                (see source listing below for update details)
```

This program will calculate the modified equivalent equation corresponding to a given finite-difference equation, up to the term containing a derivative of a specified order. This modified equivalent equation can be used to determine the theoretical accuracy of the finite-difference equation, as well as to produce optimal versions of weighted finite-difference equations, by choosing values of the weights that eliminate some of the leading error terms in the MEPDE.

SUPER takes its input either straight from the keyboard, in which case each piece of information required is prompted for, or from a file, which has the necessary input set out with one item to a line. Note that input should not come straight from a batch job's main input file, as the accumulated lengths of the prompts may cause a run-time error in the program. To overcome this problem, either specify that input comes from SYS$INPUT, which will suppress the prompts but still read the data from the batch job input file, or else use a separate input file. Input files for SUPER are generated both by the program DISC, described above, and by SUBST, described in Section A.4. Although both these automated programs produce files that are somewhat longer than those that may be produced manually, SUPER will collect like terms together and simplify them properly, so there is no loss incurred by using these programs to generate input files.

Although a user should never have to know about the format of the input file for SUPER (since these files are generated by both DISC and SUBST), a knowledge of this format may be useful to either make minor modifications directly or to track down problems. The file starts with a one-line title for the method, which may be up to 80 characters long, followed by the maximum order of the derivative to be used in the calculations. This number should be chosen with care, with regard to the expected order of the method, and the available amount of computer time. As this number increases, the amount of work SUPER must do also increases greatly, which is then reflected in the execution time. More importantly, however, the magnitude of the numbers it must work with increases, which in turn increases the chance of encountering integers beyond the machine's range. Note that with 32-bit integers such as those used on the VAX range of computers, there is an absolute upper limit of twelve on this number, due to the fact that the program uses the factorial of this number in the calculations.

If this restricted range of integers is a problem, the program SUPERSLOW can be used instead of SUPER. SUPERSLOW uses arrays of integers to represent multiple-precision integers, thus extending the range of numbers available. The major disadvantage here is that all arithmetic operations on such integers are simulated as part of

the program, which results in a massive increase in the amount of CPU time needed to find results. This increase has been found to be around 100 fold in some small test cases where a direct comparison has been possible. Apart from the size of the numbers they will handle, SUPER and SUPERSLOW are identical, so no further distinction between them will be made.

Following the derivative order in the input file (each item must be on a separate line) is the number of weights used, and their names (which are converted to upper case internally by the program). Following this is a repeated construct that specifies the finite-difference equation. It starts off by specifying the numerator, denominator and powers of Δx, Δt, u and α (or Δx, Δy, Δt, u, v, α_x and α_y for the two-dimensional case) followed by the powers of the weights (if any weights are being used, the powers must be given in the same order that the weights were specified at the top of the input file). This information makes up a coefficient, which may (indeed should) multiply several τ values at various grid points. These grid points are now specified, by giving the x space position, relative to j, the y space position relative to k (for the two-dimensional case only), followed by the time level, relative to n, and an *integer* which multiplies this particular term. Any denominator that multiplies a term must be taken out as a factor and specified with the rest of the coefficient part above. This is followed by either a "Y", to indicate there is another grid point value that multiplies the current coefficient, or a "N" to indicate that there is not. Once all the grid points that a given coefficient multiplies have been entered, a "Y"/"N" response is required to indicate whether there is another coefficient to enter. If so, then the whole construct just described is repeated, until all coefficients are entered.

The program works by expanding all the terms out as Taylor Series about the (j, n) grid point (or (j, k, n) grid point, for the two-dimensional case), then collecting up any like terms to produce a truncated version of the equivalent partial differential equation (EPDE) corresponding to the finite-difference equation. The EPDE is then normalised by dividing by the coefficient of $\partial\tau/\partial t$, and is then repeatedly differentiated with the result being added back into the equation so as to remove all the derivatives with a time dependence in them, with the exception of the $\partial\tau/\partial t$ term. This produces the

MEPDE, which has only spatial derivatives, except for the $\partial\tau/\partial t$ term. This form can then be used to verify the consistency of the difference equation, while the additional terms form the truncation error of the method, denoted by E.

This technique for finding the modified equivalent equation assumes that the order of differentiation is unimportant when eliminating terms. If this assumption is not valid (for instance, if the initial condition contains a discontinuity) then any results obtained using this MEPDE may be invalid.

The modified equivalent equation is stored internally as a set of linked lists, so there is no specific limit on the number of terms in the coefficient of any particular derivative. Also, any components that are no longer required are returned to the available storage by means of the Pascal `Dispose` function, so as to keep the overall storage requirements to a minimum. In the case of a large problem on a machine with a small amount of memory, the program may terminate with an "out of memory" type error, but on a machine such as a VAX this is extremely unlikely, although it can happen with early versions of SUPER that don't release unneeded storage space with the Pascal `Dispose` procedure.

One of the steps in producing the modified equivalent equation is to divide the equivalent equation by the coefficient of $\partial\tau/\partial t$, as mentioned above. If this coefficient is an expression involving addition and/or subtraction then the division is beyond the scope of this program. To overcome this restriction, the coefficient is given the name `DENOM`, and the equation is divided by this. In cases where this happens, the output will contain the expression that has been replaced at the top of the output, then the modified equivalent equation will refer to powers of `DENOM`. So that the program can achieve the maximum simplification (and not produce results that are highly misleading to a casual observer), the powers of `DENOM` in any given coefficient in the modified equivalent equation are equalised by multiplying appropriate terms by `DENOM`. This allows the program's simplification routines to work on the expression, in particular, correctly representing some of the leading coefficients as zero rather than extremely complicated expressions that simplify to zero after much extra work. This allows easy verification that these coefficients *are* zero, for checking optimal difference equations.

The output from this program consists of two files. The first contains the complete form of the EPDE and the MEPDE, which can be used to analyse the difference equation. The second is a file that forms the basis for an input to MACSYMA. It contains equations that are the cancelled forms of each of the error terms from the MEPDE, with right hand sides that are just "= 0". Each of the equations is numbered "Fn", where n is the order of the derivative that term was multiplying in the MEPDE (or "Fny", where y is the order of the y derivative of the term, in the two-dimensional case). Thus to find optimal values for the weights that make some of the leading error terms zero, this file could be input to MACSYMA, and the ALGSYS command in that package used to solve the desired subset of the equations.

Before doing this however, it is necessary to modify the F2 equation if the original equation involved diffusion, as this term will contain the diffusion term $-\alpha\ \partial^2\tau/\partial x^2$. Thus for the pure diffusion equation, there is a line

```
F2: (- 1 ) = 0$
```

which must be deleted. If there is more than one term involved in the modified equivalent equation coefficient C2, say a scheme for the transport equation that has artificial diffusion, then just the one term needs to be deleted, not the whole equation. If the whole equation is deleted, then the reference to the equation must also be removed from the list of equations to be solved. In the above case, the line that solves for the weights is

```
algsys([f2,f4,f6],[THETA,PHI]);
```

which must be changed to

```
algsys([f4,f6],[THETA,PHI]);
```

since the F2 equation was deleted entirely.

Note that the output of the MACSYMA file is optional, and can be suppressed by just pressing RETURN when its name is prompted for. In the case of the full two-

dimensional advection-diffusion equation, however, this MACSYMA file has not yet been implemented, and alternate means should be used to find optimal values for the weights. The easiest way to do this is to edit the main output file, removing the EPDE, and setting up the desired equations to be solved, then using this as input to MACSYMA. This procedure is usually satisfactory, as the reduction of any answers from the form involving Δx's etc. to the simpler form involving s_x's and s_y's is usually very simple, and can be done by the user as the answer is read.

It should be noted that the output from MACSYMA is not suitable for input directly into SUBST as a weight file (ie. .WTS extension). The values of the weights must be read from the MACSYMA output and re-formatted into the format of a weights file using a text editor such as Ludwig. The format of this file is described in Section A.4. Also worth noting is the fact that if the output from MACSYMA is something of the form

```
(Cn)                    []
```

then MACSYMA was unable to find any solutions to the given set of equations for the weights specified. Note also that the "(Cn)" in output by MACSYMA (like the above) has no relationship to the coefficients of the modified equivalent equation which have been denoted by Cn in this work.

Example

For the example problem, given the input file for SUPER generated by DISC, the main output file is

```
1-D Diffusion: Weighted (1,5,1) Method

Initial coefficient of DTau/DT is :

(    1 * DELTA_T )
```

```
Equivalent Partial Differential Equation :
------------------------------------------

(    1 ) * DTau / DT

(    1/2 * THETA * DELTA_T ) * D2Tau / DT2

( - 1 * ALPHA ) * D2Tau / DX2

(    1/6 * DELTA_T**2 ) * D3Tau / DT3

(    1/24 * THETA * DELTA_T**3 ) * D4Tau / DT4

( - 1/12 * ALPHA * PHI * DELTA_X**2 ) * D4Tau / DX4

(    1/120 * DELTA_T**4 ) * D5Tau / DT5

(    1/720 * THETA * DELTA_T**5 ) * D6Tau / DT6

(    1/90 * ALPHA * DELTA_X**4
   - 1/72 * ALPHA * PHI * DELTA_X**4 ) * D6Tau / DX6

"Modified" Partial Differential Equation :
------------------------------------------

(    1 ) * DTau / DT

( - 1 * ALPHA ) * D2Tau / DX2

( - 1/12 * ALPHA * PHI * DELTA_X**2
   + 1/2 * ALPHA**2 * THETA * DELTA_T ) * D4Tau / DX4
```

```
(   1/90 * ALPHA * DELTA_X**4
  - 1/72 * ALPHA * PHI * DELTA_X**4
  + 1/12 * ALPHA**2 * THETA * PHI * DELTA_X**2 * DELTA_T
  - 1/2 * ALPHA**3 * THETA**2 * DELTA_T**2
  + 1/6 * ALPHA**3 * DELTA_T**2 ) * D6Tau / DX6
```

It is evident from this that the finite-difference equation is consistent with the one-dimensional diffusion equation, with a truncation error

$$E(\tau, s, \Delta x) = -\frac{\alpha(\Delta x)^2}{12}(\varphi - 6s\theta)\frac{\partial^4 \tau}{\partial x^4} + \frac{\alpha(\Delta x)^4}{360}(4 - 5\varphi(1 - 6s\theta) + 60s^2(1 - 3s\theta))\frac{\partial^6 \tau}{\partial x^6} + O\{(\Delta x)^6\} \quad \text{(A.3.1)}$$

Thus the weighted (1,5,1) method is in general second-order accurate, although in Section A.4 below, values will be chosen for the weights that make the method more accurate.

The MACSYMA file produced for this method is

```
$ set noverify
$ macsyma

F2: (- 1 ) = 0$

F4: (- 1/12 * PHI
  + 1/2 * S * THETA ) = 0$

F6: (  1/90
  - 1/72 * PHI
  + 1/12 * S * THETA * PHI
  - 1/2 * S**2 * THETA**2
  + 1/6 * S**2 ) = 0$
```

```
algsys([f2,f4,f6],[THETA,PHI]);
quit();
$ if "''notify'" .eqs. "" then notify = "$sys_ute:notify"
$ notify "<MACSYMA finished>"
$ exit
```

In this case it is clear that the F2 equation must be removed (as explained above) for the system of equations to be consistent. It should be noted however that if the method incorporates numerical diffusion, then only one term needs to be removed from the F2 equation, rather than the whole equation as was the case here.

A.4 Evaluating the Optimal Equation

```
Program Name:   SUBST
Author:         Ken Hayman
Date:           July 1986
```

The purpose that program SUBST serves is two-fold. Firstly, it can be used to create an input file for SUPER from a file which is much easier to enter correctly than the SUPER input file itself. Secondly, and its main use, however, is to substitute specific values of weights into a weighted difference equation and simplify the result to produce the new equation. These values of the weights may be generated from MACSYMA, as described above, or by some other convenient method, such as another package, or by hand.

The main input file to SUBST is the file that specifies the finite-difference equation to be used. This is exactly the output file produced by the program DISC described in Section A.2 above, so if DISC was used to generate the initial finite-difference equation, then the output file produced by it can be used directly. Otherwise, the file must be created using a text editor. This file is very similar to the input file described above for DSET. Its format consists of a one-line title for the method, followed by

the finite-difference equation itself, which is specified by a coefficient *in parentheses* multiplied by the value of τ at a given grid point, repeated for each grid point that the equation includes. For example, one term from an equation may look like

```
(1
-2 * S) * TAU(N,J)
```

for the one-dimensional case, or

```
(1
-2 * Sx
-2 * Sy) * TAU(N,J,K)
```

for the two-dimensional case, which specifies $(1-2s)\tau_j^n$ in the first case or $(1-2s_x-2s_y)\tau_{j,k}^n$ in the second. Note that each grid point in the computational stencil of the equation must appear exactly once in the input file, and while the coefficient may extend over multiple lines, it must be enclosed in parentheses, no matter how simple it is (eg. τ_j^n must be specified as `(1) * TAU(N,J)`). Each new grid point has an implicit "+" sign in front of its coefficient, so the signs of the terms inside the coefficient should be written accordingly. Also, after the last grid point of the equation, an "= 0" is required. This requirement is both to accommodate the output from DISC, and also to serve as a reminder that the equation must be in LHS = 0 form when entered into this program.

The other input file into this program is a file that specifies the values of weights to be substituted into the equation. If no weights are to be substituted (ie. this run is only to generate an input file for SUPER) then RETURN should be pressed when this file name is prompted for. If, however, it is desired to substitute some weight values, this file is required. The format of the file is

<weight-name> = <expression>

where <weight-name> is one of the weights in the equation, and <expression> is the expression to be substituted in its place. This expression can extend onto multiple

lines, and need not be in parentheses. Each new weight name should start on a new line, however.

One restriction on the expressions that may be entered into the program SUBST, either as a coefficient of the difference equation or the value of a weight, is that any exponentiation must be to an *integer* power (ie. no weights or expressions are allowed as powers). This is due to the internal representation used, and should not be a problem in any forseeable application of this program to developing finite-difference equations, and has been mentioned for completeness only. Also, division by expressions containing addition and/or subtraction is not allowed directly, but can be achieved by dividing by the pre-defined name DENOM, which can then be defined to be the expression to be divided by. This is also not a restriction in practice, as the sets of weights from a weighted scheme have been found to either all share a common denominator, or can be made so very easily, so one DENOM suffices. In the unlikely event that this is not the case, several runs of SUBST, using different weight files with different values of DENOM could be used.

SUBST works by building up a tree form of each weight to be substituted, then building a tree for each coefficient in turn, substituting the weights into this by referencing the appropriate weight sub-tree at the correct place in the expression tree, and simplifying the resulting expression. This new expression is then written to the output file.

The output files from this program are exactly the same as those from DISC, namely a formatted version of the new finite-difference equation, and an optional input file for the program SUPER (which may be suppressed by pressing RETURN when prompted for its name). Note that the file type for the finite-difference equation file produced by SUBST is different from that produced by DISC (.OFD rather than .FDE) to distinguish the two files. This allows the original weighted finite-difference equation to be used with a different weight file at a later time, if desired. The input file for SUPER allows the user to run the resulting finite-difference equation back through SUPER to verify that the given weighting has removed or modified the leading error terms as expected, and also to look at the simplified form of the modified equivalent equation for the new method.

Example

From Equation A.3.1 above, either by hand calculation or using a package like MACSYMA, it can be found that the leading error terms in the modified equivalent equation for the weighted (1,5,1) method are removed by the choice of weights

$$\theta = \frac{2+30s^2}{15s} \quad , \quad \varphi = \frac{4+60s^2}{5}. \tag{A.4.1}$$

To verify that this choice does in fact remove the leading errors, and to find the corresponding FDE, the following weight file is input to SUBST, along with the FDE file produced earlier by DISC:

```
Theta = (2 + 30*s^2) / (15*s)

Phi = (4+60*s^2) / 5
```

The optimal FDE, as produced by SUBST is

```
FDE for : 1-D Diffusion: Weighted (1,5,1) Method - Optimal Version

(   4 * S**-1
  + 60 * S
  + 30 ) * TAU(n+1,j)

(   1 * S
  - 60 * S**3 ) * TAU(n,j-2)

( - 64 * S
  + 240 * S**3 ) * TAU(n,j-1)

( - 8 * S**-1
  + 6 * S
  - 360 * S**3 ) * TAU(n,j)
```

```
( - 64 * S
  + 240 * S**3 ) * TAU(n,j+1)

(   1 * S
  - 60 * S**3 ) * TAU(n,j+2)

( - 30
  + 4 * S**-1
  + 60 * S ) * TAU(n-1,j)

 = 0
```

which corresponds to the FDE

$$\begin{aligned} -2\{30s^2+15s+2\}\tau_j^{n+1} &= s^2\{1-60s^2\}(\tau_{j-2}^n+\tau_{j+2}^n) \\ &- 16s^2\{4-15s^2\}(\tau_{j-1}^n+\tau_{j+1}^n) \\ &- 72\{180s^4-3s^2+4\}\tau_j^n \\ &+ 2\{30s^2-15s+2\}\tau_j^{n-1}. \end{aligned} \tag{A.4.2}$$

Note that (A.4.2) has been multiplied through by s from the output from SUBST, so as to simplify the form of the resulting FDE.

If this FDE is examined (by using SUPER with the corresponding input file, also generated by SUBST), it is found that this new equation is still consistent with the one-dimensional diffusion equation, and the leading error term is of $O\{(\Delta x)^6\}$ for arbitrary values of s, the error terms shown in Equation A.3.1 having been eliminated by the choice of weights. This means that this finite-difference equation is sixth-order accurate.

Since there is only one choice of weights that eliminates both the second and fourth-order error terms, the above FDE is in fact the highest order method possible for the (1,5,1) stencil. Other methods, of lower order, can be generated by other choices

of the weights, and still other methods are possible using other differencings of the derivative terms of the diffusion equation. It should be noted, however, that the use of off-centred derivatives leads to the inclusion of more low-order error terms, which must all be removed to generate a high-order method, which in turn requires the use of more weights. Such problems were not apparent in the example case, as centred approximations were used for all derivatives.

A.5 Numerical Stability and Solvability

```
Program Name:   VNS, SOLVABLE
Author:         Peter Steinle
Date:           1983
```

Finding a finite-difference equation which is of high-order accuracy in theory does not mean that the equation will give highly accurate numerical solutions. For the solution to be accurate, the equation must not allow the uncontrolled accumulation of round-off errors, in which case the method is termed *stable*. For implicit methods that require the solution of a set of linear algebraic equations at each time step, the solution of this set of equations must be stable, in which case the method is *solvable*. Overall, a finite-difference method is only of practical use in the smallest region where it is von Neumann stable and solvable, if the latter is applicable to the method.

The von Neumann stability of a method can be determined analytically in some cases, most notably when the computational stencil is centred about the (j, n) grid point. This approach, however, is extremely complicated for all but the most basic finite-difference equations, so a numerical equivalent has been developed. This is based on the von Neumann stability test, discussed more fully in Noye (1985), which requires that

$$|G(s, \beta)| \leq 1 \qquad \text{for all } \beta \in [0, 2\pi] \tag{A.5.1}$$

where G is the amplification factor associated with the method. The program for this test, which is called VNS, works by evaluating G for many values of β in the range

$[0, 2\pi]$ for a given value of s, and checking whether $|G| \leq 1$ or not. If $|G| > 1$ for any value of β, then the method is unstable for that value of s, otherwise it is stable. The results are output as a graph suitable for printing on a line printer. Note that while there are versions of this program for both the one and two-dimensional cases, the other programs described later in this section are at present only available for the one-dimensional case.

The solvability of a method is determined solely by the coefficients at the $(n+1)^{\text{th}}$ time level. The criterion used is that the coefficient matrix for the system of linear equations must be diagonally dominant, which means that the magnitude of the coefficient on the leading diagonal of the matrix must not be exceeded by the sum of the magnitudes of the other coefficients on that row of the matrix. This may be written mathematically as

$$|a_{i,i}| \geq \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{i,j}| \quad , \qquad i = 1(1)n, \tag{A.5.2}$$

where the $A = [a_{i,j}]$ is the $n \times n$ coefficient matrix for the system of equations to be solved. This condition is checked by the program SOLVABLE. Note that this definition of diagonal dominance is adequate in this context due to the form of the implicit finite-difference equations being used.

The input for the programs for the one-dimensional case is in a different format to the previous programs, and is achieved by editing the Fortran source program `COEFF.FOR` and entering data about the method. A listing of this program is given below in Section A.5.1. The variable `Nlevels` must be set to indicate the number of time levels involved in the computational stencil, and should be either two or three in most cases. A three line title can be entered into the variable `Title`, and this appears at the top of the result file. The last data needed are the coefficients of the finite-difference equation itself. As for the previous programs, the finite-difference equation must be in the form of LHS = 0. The variable `Coeff(a,b)` contains the coefficient of τ_{j+a}^{n+b}, while the variables `Stencil(b,1)` and `Stencil(b,2)` hold the number of grid points to the left and right respectively of the (j,n+b) grid point. This array `Stencil` is used to speed up the resulting program by eliminating redundant operations. Note that when entering values for `Coeff`, the variable to go along the horizontal axis of the output

must be referred to as `C` and the one on the vertical axis `S`, regardless of the label these axes have in the output.

The input for the two-dimensional von Neumann stability program is very similar to the above. The title and the variable `Nlevels` are exactly the same, but now the stencil is split up into "planes" running in the x direction. For each of these "planes" at each time level, the value `Stencil(a,b,1)` and `Stencil(a,b,2)` hold the number of grid points in the computational stencil to the left and right respectively of $\tau_{j,k+a}^{n+b}$. The values `Coeff(a,b,c)` hold the coefficients of $\tau_{j+a,k+b}^{n+c}$. Although there are four parameters available, namely $Cx = c_x$, $Cy = c_y$, $Sx = s_x$ and $Sy = s_y$, $Cx = Cy$ and $Sx = Sy$ in order that the result can be output in a useful graphical format. As was the case for the one-dimensional program, the variable $Cx = Cy$ is output on the horizontal axis, while the variable $Sx = Sy$ is output on the vertical axis.

Having set up the correct coefficients in the appropriate program, this program must then be compiled and linked with both the double precision IMSL library (the single precision IMSL library is used for the two-dimensional program, to reduce the running time to an acceptable value) and either the program VNS to examine the von Neumann stability of the method, SOLVABLE to examine the solvability of an implicit method. As an example of how to do this, the command to link a coefficient file called MY_COEFF for a von Neumann stability plot is

```
$ LINK MY_COEFF,VNS,SYS_PACK:IMSLIBD/L
```

after the file VNS has been copied to the current default directory and compiled. The single precision IMSL library is called IMSLIBS rather than IMSLIBD in the above. When the program is run, several questions must be answered. The first is the name for the output file, followed by the labels to put on the horizontal and vertical axes. Following this is the range of values for each of the variables, followed by the number of (equally spaced) values for each of these to use. If there is no second variable (eg in the case of the advection or diffusion equation with no weight involved), then specifying a value of zero points for the vertical variable will produce the correct output.

Example

To find the region in which the sixth-order finite-difference equation (A.4.2) is von Neumann stable, the coefficients of this equation should be edited into a copy of `COEFF.FOR`, along with the other information required (title, number of levels, etc.). This is then linked with the `VNS` program, and the resulting output is as follows, using an interval of [0,1] with 50 subintervals.

```
(1,5,1) Explicit Method O{6}
1-D Diffusion Equation

        W
        ^
        |
        |
 0.0000 +SSSSSSSSSSSSSSSSSSSSSSSS.........................
        -+---------+---------+---------+---------+---------+---> S
      0.0000    0.2000    0.4000    0.6000    0.8000    1.0000
```

This indicates that the method is von Neumann stable in the region

$$s \leq 1/2. \tag{A.5.3}$$

In order to get a more accurate stability region, the interval of interest can be narrowed down to the area around $s = 1/2$. The output for the interval [0.45,0.55] with 50 subintervals is

```
(1,5,1) Explicit Method O{6}
1-D Diffusion Equation

        W
         ^
         |
         |
 0.0000 +SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS.................
        -+---------+---------+---------+---------+---------+---> S
      0.4500    0.4700    0.4900    0.5100    0.5300    0.5500
```

which shows that the actual upper limit for stability is between 0.516 and 0.518. Further investigation using successively smaller intervals gives a more accurate von Neumann stability range for the method as

$$s \leq 0.51638. \tag{A.5.4}$$

Since this particular method is explicit, there is no need to check the solvability of the method. Thus the region defined by (A.5.4) gives the usable region for the method.

A.5.1 Listing of Module COEFF.FOR

This section contains a listing of the code module COEFF.FOR, which is the file that must be edited to input the finite-difference equation into the stability checking programs. Note that in VAX Fortran, an exclamation mark (!) means the rest of that line is a comment.

```
      Subroutine Get_Coeffs (C, s)

* ************************************************************************
* This is the only module that needs to be changed by a user. It       *
* contains the details of the FDE. Note that the FDE should be         *
* entered in LHS = 0 form.                                             *
*                                                                      *
* The first variable will be the one appearing on the horizontal axis  *
* if this is used with stability programs                              *
*                                                                      *
* Written by Peter Steinle                                             *
* Modified by Ken Hayman, July 1987                                    *
*   Changed to LHS = 0 format, for compatability with the rest of the  *
*   FDE development system. Removed useless third dimension from       *
*   Coeff array.                                                       *
************************************************************************
      Implicit None
      Integer Nlevels, Stencil(-2:1, 2)
      Double Precision C, s, Coeff(-3:3, -2:1)
      Character*(70) Title(3)
      Common /FDE/ Coeff, Stencil, Title, Nlevels

* Set up the number of levels that the method involves here
      Nlevels = 3              ! No. of levels involved
* And the three line title to be printed above the stability
* plot is entered here.
      Title(1) = '== Enter name here========================='
      Title(2) = ' Name of eqn  '
      Title(3) = '  and anything else here '
* STENCIL(m,1) holds the number of points to the left and
* STENCIL(m,2) holds the number of points to the right of the
```

```
*  (j,m)th grid point
*
*  COEFF(x,y,1) holds the coefficient of Tau(j+x,n+y)
        Stencil(1,1) = 0
        Coeff(-2,1) = 0d0                       ! Tau(j-2,n+1)
        Coeff(-1,1) = 0d0                       ! Tau(j-1,n+1)
        Coeff( 0,1) = 1 + 2*s                   ! Tau(j  ,n+1)
        Coeff( 1,1) = 0d0                       ! Tau(j+1,n+1)
        Coeff( 2,1) = 0d0                       ! Tau(j+2,n+1)
        Stencil(1,2) = 0

        Stencil(0,1) = 1
        Coeff(-2,0) = 0d0                       ! Tau(j-2,n  )
        Coeff(-1,0) = C + 2*s                   ! Tau(j-1,n  )
        Coeff( 0,0) = 0d0                       ! Tau(j  ,n  )
        Coeff( 1,0) = -C + 2*s                  ! Tau(j+1,n  )
        Coeff( 2,0) = 0d0                       ! Tau(j+2,n  )
        Stencil(0,2) = 1

        Stencil(-1,1) = 0
        Coeff(-2,-1) = 0d0                      ! Tau(j-2,n-1)
        Coeff(-1,-1) = 0d0                      ! Tau(j-1,n-1)
        Coeff( 0,-1) = 1 - 2*s                  ! Tau(j  ,n-1)
        Coeff( 1,-1) = 0d0                      ! Tau(j+1,n-1)
        Coeff( 2,-1) = 0d0                      ! Tau(j+2,n-1)
        Stencil(-1,2) = 0

        Return
        End
```

Appendix B

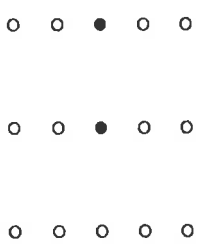
Key Files for Program DISC

B.1 Introduction

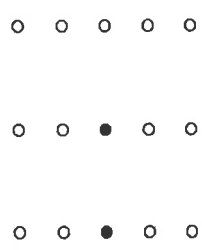
This appendix lists the available differencings and their corresponding key numbers for use with the program DISC. If differencings are required that are not listed here, then a new file must be created that contains both the required "standard" differencings, from those listed here, as well as the extra ones that are required. This new file can then be used with DISC to difference the partial differential equation in the desired manner. This process is described in more detail in the section on program DISC above.

B.2 Keys for 1-Dimensional Program

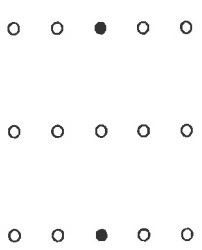
Time derivatives:



$$\frac{\partial \tau}{\partial t} \qquad j = -2(1)2 \qquad \text{KEY} = 1 + (j+2)$$

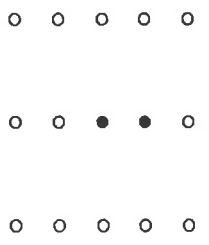


$$\frac{\partial \tau}{\partial t} \qquad j = -2(1)2 \qquad \text{KEY} = 6 + (j+2)$$

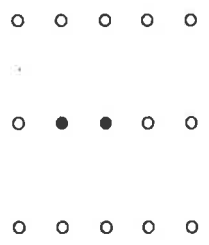


$$\frac{\partial \tau}{\partial t} \qquad j = -2(1)2 \qquad \text{KEY} = 11 + (j+2)$$

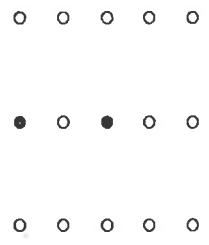
Space derivatives:



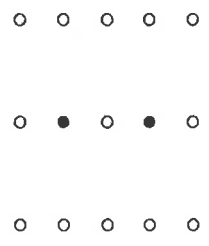
$$\frac{\partial \tau}{\partial x} \qquad n = -1(1)1 \qquad \text{KEY} = 16 + (n+1)$$



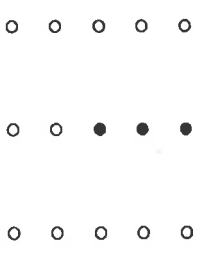
$$\frac{\partial \tau}{\partial x} \qquad n = -1(1)1 \qquad \text{KEY} = 19 + (n+1)$$



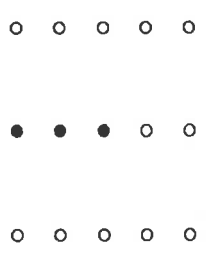
$$\frac{\partial \tau}{\partial x} \qquad n = -1(1)1 \qquad \text{KEY} = 22 + (n+1)$$



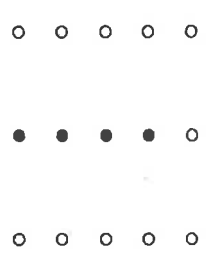
$$\frac{\partial \tau}{\partial x} \qquad n = -1(1)1 \qquad \text{KEY} = 25 + (n+1)$$



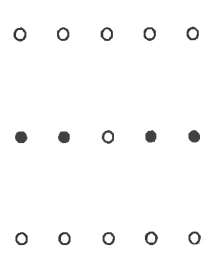
$$\frac{\partial \tau}{\partial x} \qquad n = -1(1)1 \qquad \text{KEY} = 28 + (n+1)$$



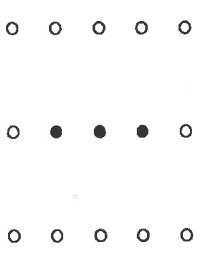
$$\frac{\partial \tau}{\partial x} \qquad n = -1(1)1 \qquad \text{KEY} = 31 + (n+1)$$



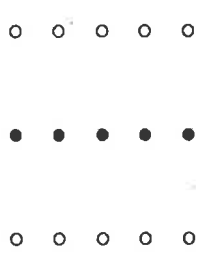
$$\frac{\partial \tau}{\partial x} \qquad n = -1(1)1 \qquad \text{KEY} = 34 + (n+1)$$



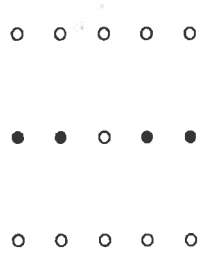
$$\frac{\partial \tau}{\partial x} \qquad n = -1(1)1 \qquad \text{KEY} = 37 + (n+1)$$



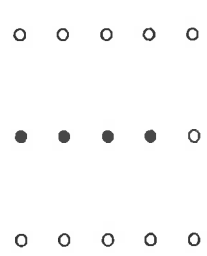
$$\frac{\partial^2 \tau}{\partial x^2} \qquad n = -1(1)1 \qquad \text{KEY} = 40 + (n+1)$$



$$\frac{\partial^2 \tau}{\partial x^2} \qquad n = -1(1)1 \qquad \text{KEY} = 43 + (n+1)$$



$$\frac{\partial^3 \tau}{\partial x^3} \qquad n = -1(1)1 \qquad \text{KEY} = 46 + (n+1)$$

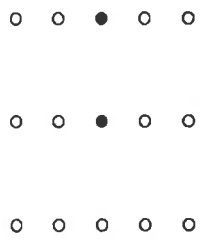


$$\frac{\partial^3 \tau}{\partial x^3} \qquad n = -1(1)1 \qquad \text{KEY} = 49 + (n+1)$$

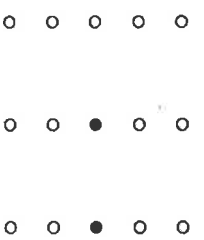
o o o o o			
• • • • •	$\frac{\partial^4 \tau}{\partial x^4}$	$n = -1(1)1$	KEY $= 52 + (n+1)$
o o o o o			

B.3 Keys for 2-Dimensional Program

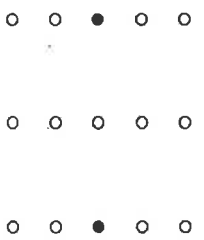
Time derivatives:



$$\frac{\partial \tau}{\partial t} \qquad \left.\begin{array}{l} j = -2(1)2 \\ k = -2(1)2 \end{array}\right\} \text{KEY} = 1 + 5(j+2) + (k+2)$$



$$\frac{\partial \tau}{\partial t} \qquad \left.\begin{array}{l} j = -2(1)2 \\ k = -2(1)2 \end{array}\right\} \text{KEY} = 26 + 5(j+2) + (k+2)$$



$$\frac{\partial \tau}{\partial t} \qquad \left.\begin{array}{l} j = -2(1)2 \\ k = -2(1)2 \end{array}\right\} \text{KEY} = 51 + 5(j+2) + (k+2)$$

Space derivatives:

$$\left.\begin{array}{ll} & k=-2(1)2 \\ s\equiv x: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY} = 76+3(k+2)+(n+1)$$

o o o o o

$$\frac{\partial\tau}{\partial s}$$

o o • • o

$$\left.\begin{array}{ll} & j=-2(1)2 \\ s\equiv y: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY} = 91+3(j+2)+(n+1)$$

o o o o o

$$\left.\begin{array}{ll} & k=-2(1)2 \\ s\equiv x: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY} = 106+3(k+2)+(n+1)$$

o o o o o

$$\frac{\partial\tau}{\partial s}$$

o • • o o

$$\left.\begin{array}{ll} & j=-2(1)2 \\ s\equiv y: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY} = 121+3(j+2)+(n+1)$$

o o o o o

$$\left.\begin{array}{ll} & k=-2(1)2 \\ s\equiv x: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY} = 136+3(k+2)+(n+1)$$

o o o o o

$$\frac{\partial\tau}{\partial s}$$

• o • o o

$$\left.\begin{array}{ll} & j=-2(1)2 \\ s\equiv y: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY} = 151+3(j+2)+(n+1)$$

o o o o o

$$\left.\begin{array}{ll} & k = -2(1)2 \\ s \equiv x: & \\ & n = -1(1)1 \end{array}\right\} \text{KEY} = 166 + 3(k+2) + (n+1)$$

o o o o o

$$\frac{\partial \tau}{\partial s}$$

o • o • o

o o o o o

$$\left.\begin{array}{ll} & j = -2(1)2 \\ s \equiv y: & \\ & n = -1(1)1 \end{array}\right\} \text{KEY} = 181 + 3(j+2) + (n+1)$$

$$\left.\begin{array}{ll} & k = -2(1)2 \\ s \equiv x: & \\ & n = -1(1)1 \end{array}\right\} \text{KEY} = 196 + 3(k+2) + (n+1)$$

o o o o o

$$\frac{\partial \tau}{\partial s}$$

o o • • •

o o o o o

$$\left.\begin{array}{ll} & j = -2(1)2 \\ s \equiv y: & \\ & n = -1(1)1 \end{array}\right\} \text{KEY} = 211 + 3(j+2) + (n+1)$$

$$\left.\begin{array}{ll} & k = -2(1)2 \\ s \equiv x: & \\ & n = -1(1)1 \end{array}\right\} \text{KEY} = 226 + 3(k+2) + (n+1)$$

o o o o o

$$\frac{\partial \tau}{\partial s}$$

• • • o o

o o o o o

$$\left.\begin{array}{ll} & j = -2(1)2 \\ s \equiv y: & \\ & n = -1(1)1 \end{array}\right\} \text{KEY} = 241 + 3(j+2) + (n+1)$$

$$\left.\begin{array}{rl} & k=-2(1)2 \\ s\equiv x: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 256+3(k+2)+(n+1)$$

o o o o o

$$\frac{\partial\tau}{\partial s}$$

• • • • o

$$\left.\begin{array}{rl} & j=-2(1)2 \\ s\equiv y: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 271+3(j+2)+(n+1)$$

o o o o o

$$\left.\begin{array}{rl} & k=-2(1)2 \\ s\equiv x: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 286+3(k+2)+(n+1)$$

o o o o o

$$\frac{\partial\tau}{\partial s}$$

• • o • •

$$\left.\begin{array}{rl} & j=-2(1)2 \\ s\equiv y: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 301+3(j+2)+(n+1)$$

o o o o o

$$\left.\begin{array}{rl} & k=-2(1)2 \\ s\equiv x: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 316+3(k+2)+(n+1)$$

o o o o o

$$\frac{\partial^2\tau}{\partial s^2}$$

o • • • o

$$\left.\begin{array}{rl} & j=-2(1)2 \\ s\equiv y: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 331+3(j+2)+(n+1)$$

o o o o o

$$\left.\begin{array}{ll} & k=-2(1)2 \\ s\equiv x: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 346+3(k+2)+(n+1)$$

o o o o o

$$\frac{\partial^2\tau}{\partial s^2}$$

• • • • •

$$\left.\begin{array}{ll} & j=-2(1)2 \\ s\equiv y: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 361+3(j+2)+(n+1)$$

o o o o o

$$\left.\begin{array}{ll} & k=-2(1)2 \\ s\equiv x: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 376+3(k+2)+(n+1)$$

o o o o o

$$\frac{\partial^3\tau}{\partial s^3}$$

• • o • •

$$\left.\begin{array}{ll} & j=-2(1)2 \\ s\equiv y: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 391+3(j+2)+(n+1)$$

o o o o o

$$\left.\begin{array}{ll} & k=-2(1)2 \\ s\equiv x: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 406+3(k+2)+(n+1)$$

o o o o o

$$\frac{\partial^3\tau}{\partial s^3}$$

• • • • o

$$\left.\begin{array}{ll} & j=-2(1)2 \\ s\equiv y: & \\ & n=-1(1)1 \end{array}\right\} \text{KEY } = 421+3(j+2)+(n+1)$$

o o o o o

$$\left.\begin{array}{rl} & k = -2(1)2 \\ s \equiv x: & \\ & n = -1(1)1 \end{array}\right\} \text{KEY} = 436 + 3(k+2) + (n+1)$$

o o o o o

$$\frac{\partial^4 \tau}{\partial s^4}$$

• • • • •

o o o o o

$$\left.\begin{array}{rl} & j = -2(1)2 \\ s \equiv y: & \\ & n = -1(1)1 \end{array}\right\} \text{KEY} = 451 + 3(j+2) + (n+1)$$

Appendix C

Listing of Program SUPER

C.1 Introduction

This Appendix gives the listing of the program SUPER, described in Appendix A above, for the one-dimensional case. This is the main program of the finite-difference equation development package, which takes a finite-difference equation and determines its modified equivalent equation. The program for the two-dimensional case is very similar to the one given here, but has an extra subscript on some arrays and extra loops in the main processing routine to handle the extra spatial dimension.

The program is written in largely standard Pascal, although the following non-standard features are used, and may have to be changed to implement the program on another system:

- Use of the VARYING [n] OF CHAR type for varying length strings. The function LENGTH(str) returns the current length of "str". Such strings may also be read in directly from a file of type TEXT in VMS Pascal.

- The underscore character (_) is used in identifier names, to enhance readability.

- Use of the OPEN procedure to open a named file. The options in this call allow such things as providing default values for unspecified portions of the file name,

whether the file should already exist or is to be created and what sharing options, if any, are to be allowed.

- Use of error recovery procedures from I/O calls, to prevent program termination if an error arises, such as a named file not existing. The function STATUS(file) returns a non-zero code if an error in fact occurred during the last I/O operation on "file".

- Use of the exponentiation operator "**". Note that for systems which do not support this, "$A**B$" can be written as "$\exp(B*\log(A))$".

Many compilers have some or all of these features allowed, despite their being non-standard. On systems where one or more of them are missing, they are not hard to implement, or in some cases, such as the error trapping,they can be omitted altogether, since they are not essential to the operation of the program.

C.2 The Program SUPER

```
Program Modified_Equation(input,output,infile,outfile,macfile);

{  Author   : K.J. Hayman
   Date     : 8 March 1984
   System   : VAX/VMS V3.4
   Language : Pascal V2.3  }

{ As of December 1987:
   System   : VAX/VMS V4.5
   Language : Pascal V3.5  }

{  Modified :
      23 August 1984 - produce output file suitable for
                       running through MACSYMA
```

```
      September 1984 - allow coefficient of DTau/DT to be
                       multi-nomial
      17 April 1986  - incorporate simplification of modified
                       equation where coeff of DTau/DT is multi-nomial.
                       Also use DISPOSE to control use of VM to
                       some extent.
      16 August 1986 - incorporate into FDE development system, give
                       the various files default types
      25 August 1986 - fix bug with non-Var parameter to Write_Entry
                       in relation to FIX_ORDER call
       November 1986 - fix MACSYMA output file, remove denominator
                       terms from it
       16 April 1987 - Allow simplification of C2 term for multinomial
                       denominators
         27 May 1987 - Make MACSYMA file a valid .COM file }

{This program is designed to find the Modified PDE for a given FDE,
which gives the coefficients C2,C3,C4,... . The program will allow
coefficients up to C12 to be calculated (except that intermediate
results may generate integer overflows if this is attempted).}

Const max_wts = 15;       {maximum number of weights allowed}
      top_deriv = 12;     {maximum order of derivative}
      src = 0;
      dst = 1;
      min_mac_term = 2;   { min deriv term to output to MACSYMA file }

Type  wt_type = array [1..max_wts] of integer;
      nm_type = varying [30] of char;
                          { for the names of the weights }
```

```
      entry_ptr = ^entry_type;  { the main data type }
      entry_type = record
                     num,den : integer;
                                  {numerator+denominator of coeff}
                     u,alpha : integer;  {power of U and ALPHA}
                     dx,dt   : integer;  {power of DX and DT}
                     wt      : wt_type;  {power of weights}
                     fden    : integer;
                                 {power of multi-nomial denominator}
                     next    : entry_ptr;{next term}
                   End;
                                 { TABLE represents main equation }
Var   table     : array [0..top_deriv,0..top_deriv,0..1] of entry_ptr;
      exists    : array [0..top_deriv] of boolean;
      exist1    : boolean;              { >= 1 exists flag is true }
      den_ptr   : entry_ptr;
      max_deriv : integer;    {max deriv used in current problem}
      cnt,
      wts       : integer;              {number of weights being used}
      infile    : text;                 {input file}
      outfile   : text;                 {output file}
      macfile   : text;                 {MACSYMA file}
      iname     : varying [80] of char; {VAX/VMS input file name}
      oname     : varying [80] of char; {VAX/VMS output file name}
      mname     : varying [80] of char; {VAX/VMS MACSYMA file name}
      title     : varying [80] of char; {title for output file}
      inter     : boolean;              {input is direct from terminal}
      wt_name   : array [1..max_wts] of nm_type;{names for the weights}
      too_hard  : boolean;              {something wrong with the data}
      simple    : boolean;              {can do the division}
      ch        : char;
```

```
Procedure Kill_Chain(Var ptr : entry_ptr);

{ Free up the storage used by a linked list which is no
  longer required. }

Var     ptr1 : entry_ptr;

Begin
  while (ptr <> nil) do
    Begin
      ptr1 := ptr^.next;
      dispose(ptr);
      ptr := ptr1;
    End;
End;    { Kill_Chain }

Procedure Initialise(which : integer);
{Clear out the half of TABLE pointed to by WHICH}
Var i,j : integer;

Begin
  for i:=0 to max_deriv do
    for j:=0 to max_deriv do
      kill_chain(table[i,j,which]);     { release any used space }
End; {Initialise}

Function Getgcd(n1,n2 : integer) : integer;
{Finds the GCD of N1 and N2. Note that both are assumed to be positive}

Var r : integer;
```

```
Begin
  while n2<>0 do
    Begin
      r := n1 mod n2;
      n1 := n2;
      n2 := r;
    End;
  getgcd := n1;
End; {Getgcd}

Procedure Cancel(Var n1,n2 : integer);
{Cancel the fraction n1/n2 into its lowest terms}

Var gcd : integer;

Begin
  gcd := getgcd(abs(n1),abs(n2));
  if gcd<>0 then
    Begin
      n1 := n1 div gcd;
      n2 := n2 div gcd;
    End;
End;

Procedure Add(num1,num2,den1,den2 : integer;
                Var resnum,resden : integer);
{ Procedure to add two fractions (num1/den1 and num2/den2) together.
  Note DIV is done before * to avoid overflow problems. }

Var gcd,tl1,tl2 : integer;
```

```
Begin
  gcd := getgcd(den1,den2);
  resden := den1*(den2 div gcd);
  tl1 := num1*(resden div den1);
  tl2 := num2*(resden div den2);
  resnum := tl1+tl2;
  cancel(resnum,resden);
End; {Add}

Procedure Enter(tab,x_deriv,t_deriv : integer; ent_ptr : entry_ptr;
                  wts : integer);
{Place an entry into the main equation table, adding into another
 term if possible}

Var ptr,oldptr : entry_ptr;
    found : boolean;
    i : integer;

Begin
  ptr := table[x_deriv,t_deriv,tab];    { start with existing entry }
  oldptr := nil;
  found := false;
  while (ptr<>nil) and (not found) do   { look for matching entry }
    with ptr^ do
      Begin
        found := (ent_ptr^.dx=dx) and (ent_ptr^.dt=dt)
                 and (ent_ptr^.u=u) and (ent_ptr^.alpha=alpha)
                 and (ent_ptr^.fden=fden);
        for i:= 1 to wts do
          found := found and (ent_ptr^.wt[i]=wt[i]);
```

```
        if found then           {add this into the existing entry}
          Begin
            add(ent_ptr^.num,num,ent_ptr^.den,den,num,den);
            if num=0 then   {the entries cancelled, so delete them}
              Begin
                if oldptr=nil then
                  table[x_deriv,t_deriv,tab] := ptr^.next
                else
                  oldptr^.next := ptr^.next;
                dispose(ptr);
              End;
          End
        else
          Begin    {not found, so get next link in chain, if any}
            oldptr := ptr;
            ptr := ptr^.next;
            End;
      End;
  if not found then  {doesn't exist, so must add an additional term}
    Begin
      new(ptr);
      with ptr^ do
        Begin
          cancel(ent_ptr^.num,ent_ptr^.den); {put coefficient in lowest
terms}
          num := ent_ptr^.num;
          den := ent_ptr^.den;
          dx := ent_ptr^.dx;
          dt := ent_ptr^.dt;
          u := ent_ptr^.u;
          alpha := ent_ptr^.alpha;
```

```
          for i := 1 to wts do
            wt[i] := ent_ptr^.wt[i];
          fden := ent_ptr^.fden;
          next := nil;
        End;
      if oldptr <> nil then
        oldptr^.next := ptr
      else                {if entry did not already have one term}
        table[x_deriv,t_deriv,tab] := ptr;
    End;
End; {Enter}

Function Compare(Ptr1, Ptr2 : Entry_Ptr;
                 Extra_U, Extra_A, Extra_D, Mult : Integer) : Boolean;

{ Compare the terms Ptr1 and Ptr2 for equality (with Ptr2 multiplied
by Mult, U**(Extra_U) and ALPHA**(Extra_A). The two are equal if they
have the same number of components, and each component of one can be
found in the other. Note that this relies on both of them being in
simplified form }

Var     SPtr  : Entry_Ptr;
        Found : Boolean;
        I     : Integer;

Function Count_Terms(Ptr : Entry_Ptr) : Integer;

{ Count the number of components in the term Ptr }

Var     Tally : Integer;
```

```
Begin
  Tally := 0;
  While (Ptr <> Nil) Do          { for each term in the entry }
    Begin
      Tally := Tally + 1;
      Ptr := Ptr^.Next;
    End;
  Count_Terms := Tally;
End;    { Count_Terms }

Begin   { Compare}
  Found := False;
                          { compare numbers of terms }
  If Count_Terms(Ptr1) = Count_Terms(Ptr2) Then
    Begin
      Found := True;
      While (Ptr1 <> Nil) And (Found) Do
        Begin
          SPtr := Ptr2;
          Found := False;
                          { look for term in other chain }
          While (SPtr <> Nil) And (Not Found) Do
            Begin
              Found := (Ptr1^.Num = Mult * SPtr^.Num)
                    And (Ptr1^.Den = SPtr^.Den)
                    And (Ptr1^.Dx = SPtr^.Dx) And (Ptr1^.Dt = SPtr^.Dt)
                    And (Ptr1^.U = SPtr^.U + Extra_U)
                    And (Ptr1^.Alpha = SPtr^.Alpha + Extra_A)
                    And (Ptr1^.FDen = SPtr^.FDen + Extra_D);
              For I := 1 To Wts Do
                Found := Found And (Ptr1^.Wt[I] = SPtr^.Wt[I]);
```

```
            SPtr := SPtr^.Next;
          End;
        Ptr1 := Ptr1^.Next;
      End;
  End;
 Compare := Found;              { return the result }
End;    { Compare }

Procedure Enter_data;

{ Reads in the FDE from either an input file or the terminal. }

Var i : integer;
    time_level,space_posn,minus : integer;
    another,lst_trm : boolean;
    ans : char;
    ptr : entry_ptr;

Function Factorial(n : integer):integer;

{ Returns the factorial of n. Note that values of n > 12 will
  cause an overflow error. }

Begin
  if n<0 then                   { Bad input value }
    writeln(outfile,'N < 0 - can''t find N!')
  else if n<=1 then             { trivial case }
    factorial := 1
  else                          { work it out recursively }
    factorial := n*factorial(n-1);
End;    {Factorial}
```

```
Function Binomial(n,r : integer):integer;

{ Find the binomial coefficient C(n,r). This way is ok, since this
  routine hasn't ever bombed on an overflow and it's the clearest
  way to do it, but it could be coded differently if necessary to
  avoid such errors }

Begin   {Binomial}
  if n<r then
    writeln(outfile,'N < R - can''t find binomial coefficient')
  else
    binomial := (factorial(n) div factorial(r)) div factorial(n-r);
End;    {Binomial}

Procedure Generate(space,time : integer; ptr : entry_ptr;
                        minus : integer);

{ Derive the Equivalent PDE from the initial data, by expanding things
  as Taylor series and collecting up the terms. }

Var total_deriv,t_deriv,k : integer;
    ptr2 : entry_ptr;

Procedure Put_In(deriv,t_deriv : integer; ptr,ptr2 : entry_ptr);

{ Add a term into the main equation }

Var x_deriv,i : integer;
```

```
Begin   {Put_in}
  x_deriv := deriv-t_deriv;
  with ptr2^ do
    Begin
      num := num*ptr^.num;
      den := den*ptr^.den;
      cancel(num,den);
      dx := dx+ptr^.dx;
      dt := dt+ptr^.dt;
      u := u+ptr^.u;
      alpha := alpha+ptr^.alpha;
      for i := 1 to wts do
        wt[i] := wt[i]+ptr^.wt[i];
    End;
  Enter(src,x_deriv,t_deriv,ptr2,wts);
End;    {Put_in}

Begin   {Generate}
  for total_deriv:=0 to max_deriv do    { for each total deriv ... }
    for t_deriv:=0 to total_deriv do    { for each time deriv ... }
      Begin
        new(ptr2);
        with ptr2^ do
          Begin
{ Here we must be careful to avoid 0**0, since the results are
  somewhat nasty in Vax Pascal (or were when this was written!). }
            num := minus * binomial(total_deriv,t_deriv);
            if (space=0) and (t_deriv<>total_deriv) then
              num := 0
            else if (space<>0) or (t_deriv<>total_deriv) then
                num := num * space**(total_deriv-t_deriv);
```

```
            if (time=0) and (t_deriv<>0) then
              num := 0
            else if (time<>0) or (t_deriv<>0) then
                num := num * time**t_deriv;
            den := factorial(total_deriv);
            cancel(num,den);
            dx := total_deriv-t_deriv;  { set powers up properly }
            dt := t_deriv;
            u := 0;
            alpha := 0;
            for k := 1 to wts do
              wt[k] := 0;
            fden := 0;
          End;
        if ptr2^.num<>0 then    { have got term, now insert }
          put_in(total_deriv,t_deriv,ptr,ptr2);
        dispose(ptr2);
      End;
End;    {Generate}

Procedure Upcase(Var string : nm_type);

{ Convert the passed name into upper case characters }

Var i : integer;

Begin   {Upcase}
  for i := 1 to length(string) do
    if string[i] in ['a'..'z'] then
      string[i] := chr(ord(string[i]) - ord('a') + ord('A'));
End;    {Upcase}
```

```
Begin   {Enter_Data}
  if inter then
    write('Enter Title for method : ');
  readln(infile,title);
  repeat
    if inter then
      write('Enter order of highest derivative : ');
    readln(infile,max_deriv);
  until (max_deriv>0) and (max_deriv<=top_deriv);
  repeat
    if inter then
      write('Enter number of weights : ');
    readln(infile,wts);
  until (wts>=0) and (wts<=max_wts);
  for i := 1 to wts do
    Begin
      repeat
        if inter then
          write('Enter name for weight ',i:1,' : ');
        readln(infile,wt_name[i]);
      until length(wt_name[i])>0;
      upcase(wt_name[i]);
    End;
  repeat                  { get each different term }
    new(ptr);
    with ptr^ do
      Begin
        if inter then
          write('Enter numerator of coefficient : ');
        readln(infile,num);
```

```
      if inter then
        write('Enter denominator of coefficient : ');
      readln(infile,den);
      if den<0 then          {Make sure sign is in numerator}
        Begin
          num := -num;
          den := -den;
        End;
      if inter then
        write('Enter power of DELTA X : ');
      readln(infile,dx);
      if inter then
        write('Enter power of DELTA T : ');
      readln(infile,dt);
      if inter then
        write('Enter power of U : ');
      readln(infile,u);
      if inter then
        write('Enter power of ALPHA : ');
      readln(infile,alpha);
      for i:=1 to wts do
        Begin
          if inter then
            write('Enter power of ',wt_name[i],' : ');
          readln(infile,wt[i]);
        End;
      fden := 0;
    End;
  repeat      { get each grid point that this term applies to }
    if inter then
      write('Enter space position, relative to j : ');
```

```
      readln(infile,space_posn);
      if inter then
        write('Enter time level, relative to n : ');
      readln(infile,time_level);
      if inter then
        write('Coefficient : ');
      readln(infile,minus);
      generate(space_posn,time_level,ptr,minus); { add into equation }
      if inter then
        Begin
          writeln;
          write('Another term (Y/N) : ');
        End;
      readln(infile,ans);
      if not (ans in ['N','Y','n','y']) then
        writeln('Invalid response "',ans,'" - assuming NO');
      lst_trm := (ans='Y') or (ans='y');
    until (not lst_trm);
    dispose(ptr);                { release unneeded storage }
    if inter then
      Begin
        writeln;
        write('Another coefficient (Y/N) : ');
      End;
    readln(infile,ans);
    if not (ans in ['N','Y','n','y']) then
        writeln('Invalid response "',ans,'" - assuming NO');
    another := (ans='Y') or (ans='y');
  until (not another);
End;    {Enter_data}
```

```
Procedure Normalise;
{This makes sure that the coefficient of DTau/DT is 1}

Var i,j,dgcd : integer;
    ptr      : entry_ptr;

Procedure Reduce(ptr : entry_ptr; simple : boolean);

{ Divide the specified term by the coefficient of DTau/Dt, so as to
make this coefficient 1 in the modified equation }

Var k : integer;

Begin   {Reduce}
  while ptr<>nil do
    Begin
      if simple then    { Simple denom - can do the division direct }
        Begin
          with ptr^ do
            Begin
              num := num*table[0,1,src]^.den;
              den := den*table[0,1,src]^.num;
              if den<0 then
                Begin
                  num := -num;
                  den := -den;
                End;
              cancel(num,den);
              dx := dx-table[0,1,src]^.dx;
              dt := dt-table[0,1,src]^.dt;
```

```
            u := u-table[0,1,src]^.u;
            alpha := alpha-table[0,1,src]^.alpha;
            for k:=1 to wts do
              wt[k] := wt[k]-table[0,1,src]^.wt[k];
          End;
      End
    else
      with ptr^ do    { must use the FDEN trick }
        Begin
          fden := 1;
          den := den*dgcd;
          cancel(num,den);
        End;
      ptr := ptr^.next;
  End;
End;    {Reduce}

Begin   {Normalise}
  too_hard := false;
  simple := true;
  if table[0,0,src]<>nil then   { check for valid coefficients }
    Begin
      writeln(outfile);
      writeln(outfile,'*** Coefficient of Tau(n,j) is non-zero. ***');
      writeln('*** Coefficient of Tau(n,j) is non-zero. ***');
      too_hard := true;
    End
  else
    Begin
      ptr := table[0,1,src];
      if ptr=nil then
        Begin
```

```
      writeln(outfile);
      writeln(outfile,'*** Coefficient of DTau/DT is zero. ***');
      writeln('*** Coefficient of DTau/DT is zero. ***');
      too_hard := true;
    End
  else
    Begin
      simple := (ptr^.next=nil);     { can divide easily or ...}
      if not simple then             { we must work hard }
        Begin
          dgcd := abs(ptr^.num);
          ptr := ptr^.next;
          while ptr<>nil do
            Begin
              dgcd := getgcd(dgcd,abs(ptr^.num));
              ptr := ptr^.next;
            End;
        End;
      for i:=0 to max_deriv do       { simplify each term in turn }
        for j:=0 to max_deriv do
          if (i<>0) or (j<>1) then
            reduce(table[i,j,src],simple);
      den_ptr := nil;
      if not simple then    { a few brute-force simplifications }
        Begin
          new(ptr);
          with ptr^ do
            Begin
              num := 1;
              den := 1;
              dx := 0;
```

```
        dt := 0;
        u := 0;
        alpha := 0;
        for i := 1 to wts do
          wt[i] := 0;
        fden := 0;
        next := nil;
      End;
    den_ptr := table[0,1,src];
    table[0,1,src] := ptr;
    ptr := den_ptr;
    while ptr<>nil do
      with ptr^ do
        Begin
          den := den*dgcd;
          cancel(num,den);
          ptr := next;
        End;

{ Coefficient of DTau/Dx is really U, so make it look right }

    if compare(table[1,0,src],den_ptr,1,0,1,1) then
      Begin
        new(ptr);
        with ptr^ do
          Begin
            num := 1;
            den := 1;
            dx := 0;
            dt := 0;
            u := 1;
```

```
                    alpha := 0;
                    for i := 1 to wts do
                      wt[i] := 0;
                    fden := 0;
                    next := nil;
                  End;
                table[1,0,src] := ptr;
              End;
          End
        else
          reduce(table[0,1,src],true);
      End;
  End;
End;    {Normalise}

Procedure Multiply(ptr,sptr : entry_ptr; j,k : integer;
                   isden,fix : boolean);

{ Multiply two terms together }

Var rptr : entry_ptr;
    l : integer;

Begin
  while ptr<>nil do
    Begin
      with ptr^ do
        Begin
          new(rptr);
          rptr^.num := -num*sptr^.num;
          if fix then rptr^.num := -rptr^.num;
```

```
          rptr^.den := den*sptr^.den;
          cancel(rptr^.num,rptr^.den);
          rptr^.u := u+sptr^.u;
          rptr^.alpha := alpha+sptr^.alpha;
          rptr^.dx := dx+sptr^.dx;
          rptr^.dt := dt+sptr^.dt;
          for l:=1 to wts do
            rptr^.wt[l] := wt[l]+sptr^.wt[l];
          if (isden) then
            rptr^.fden := sptr^.fden+1
          else
            rptr^.fden := fden+sptr^.fden;
        End;
      enter(dst,j,k,rptr,wts);
      dispose(rptr);
      ptr := ptr^.next;
    End;
End;    {Multiply}

Procedure Process_data;
{Main driving routine to find the modified equation}

Var i,j,k,cd,x_deriv,t_deriv : integer;
    sptr : entry_ptr;

Procedure Add_in(source,dest : integer);
{add in the last line done into the total equation}

Var i,j : integer;
    ptr : entry_ptr;
```

```
Begin
  for i:=0 to max_deriv do
    for j:=0 to max_deriv do
      Begin
        ptr := table[i,j,source];    {start of the term to be added in}
        while ptr<>nil do
          Begin
            enter(dest,i,j,ptr,wts);
                                {enter the term in the master equation}
            ptr := ptr^.next;        {point to the next term}
          End;
      End;
End; {Add_in}


Begin
  for i:=2 to max_deriv do              {total derivative}
    for x_deriv:=0 to i-1 do            {X derivative}
      Begin
        t_deriv := i-x_deriv;           {T derivative}
        sptr := table[x_deriv,t_deriv,src];
                                        {point to term to be removed}
        initialise(dst);
        while sptr<>nil do
          Begin
            for cd := i to max_deriv do
              Begin
                for j := 0 to cd do
                  Begin
                    k := cd-j;
                    if (j>=x_deriv) and (k>=t_deriv-1) then
                                                {we have a new term}
```

```
                    multiply(table[j-x_deriv,k-t_deriv+1,src],
                             sptr,j,k, false,false);
                                        {entry to be multiplied}
                End;                    {for j ...}
            End;                        {for cd ...}
          sptr := sptr^.next;
        End;                            {while sptr ...}
      add_in(dst,src);                  {update master equation}
    End;                                {for x_deriv ...}

{ Coefficient of D2Tau/DX2 is really ALPHA, so make it look right }

  if (not simple) and compare(table[2,0,src],den_ptr,0,1,1,-1) then
    Begin
      new(sptr);
      with sptr^ do
        Begin
          num := -1;
          den := 1;
          dx := 0;
          dt := 0;
          u := 0;
          alpha := 1;
          for i := 1 to wts do
            wt[i] := 0;
          fden := 0;
          next := nil;
        End;
      table[2,0,src] := sptr;
    End;
End; {Process_data}
```

```
Procedure Fixup;

{This is called in the case where we have a DENOM term, and it makes
sure that all references to denom in a term have the same power. This
has the effect of getting some simplification done. }

Var     ptr,ptr1,last,
        ptr2,one        : entry_ptr;
        deriv,x_deriv,
        t_deriv,
        i,j,
        max_denom       : integer;
        first           : boolean;

Begin
  initialise(dst);
  new(one);
  with one^ do              {The value 1, as a record. This is used}
    Begin                   {to multiply terms back into the equation.}
      num   := 1;
      den   := 1;
      u     := 0;
      alpha := 0;
      dx    := 0;
      dt    := 0;
      for i := 1 to wts do
        wt[i] := 0;
      fden := 0;
      next := nil;
    End;
```

```
  for deriv:=1 to max_deriv do
    for x_deriv:=0 to deriv do
      Begin
        t_deriv := deriv-x_deriv;
        ptr := table[x_deriv,t_deriv,src];
        if (ptr <> nil) then
          Begin
{ First work out what the maximum denominator is ... }
            max_denom := -2;
            ptr1 := ptr;
            while (ptr1 <> nil) do
              Begin
                if (ptr1^.fden > max_denom) then
                  max_denom := ptr1^.fden;
                ptr1 := ptr1^.next;
              End;
{Now we must multiply what we have by the denominator polynomial as
many times as necessary to get this whole term over a common power of
the denominator. Then the various Enter routines will take care of the
 desired simplification.}
            while (ptr <> nil) do
              Begin
                if (ptr^.fden = max_denom) then {enter it as it is}
                  multiply(one,ptr,x_deriv,t_deriv,false,true)
{ We must NOT step through terms of PTR, so it's specified it second}
                else
                  Begin
                    ptr1 := table[x_deriv,t_deriv,dst];
                                        {save current values}
                    ptr2 := ptr;        {current term}
```

```
        first := true;      {use only 1 term}
        for i := 1 to (max_denom - ptr^.fden) do
          Begin
            table[x_deriv,t_deriv,dst] := nil;
                              {initialise}
            while (ptr2 <> nil) do
              Begin
                multiply(den_ptr,ptr2,x_deriv,t_deriv,
                          true,true);
                           {multiply term * denominator}
                if first then
                  Begin
                    first := false; {only use one term}
                    ptr2 := nil;
                  End
                else
                  Begin
                    last := ptr2;
                              {get next term to multiply}
                    ptr2 := ptr2^.next;
                    dispose(last);  {clean up the garbage}
                  End;
              End;
            ptr2 := table[x_deriv,t_deriv,dst];
                                          {final result}
          End;
        multiply(ptr1,one,x_deriv,t_deriv,false,true);
                          {add in what was already there}
      End;
    ptr := ptr^.next;       {and step on to next term}
  End;
```

```
          End;
      End;
  dispose(one);                    {clean up now}
  initialise(src);
  for i := 0 to max_deriv do       {copy back to right area}
    for j := 0 to max_deriv do
      table[i,j,src] := table[i,j,dst];
End;    {Fixup}


Procedure Write_Entry(Var ptr : entry_ptr; x_deriv,t_deriv : integer);
{Writes out a single entry in the table. ALL terms are written by one
call}

Var xpwr,tpwr,
    acnt,
    k : integer;
    save_ptr : entry_ptr;
    isu,
    isalpha,
    justchanged,
    changed,
    domac,
    first : boolean;

Procedure Order_Entry(var ptr : entry_ptr);
{ This makes sure that the terms invloving U are all before the terms
without a U, thus making life easier to write MACSYMA files for the
Transport Equn }

Var     uptr,aptr,luptr,laptr : entry_ptr;
```

```
Begin
  uptr := nil;
  aptr := nil;
  luptr := nil;
  laptr := nil;
  while (ptr <> nil) do
    Begin
      with ptr^ do
        if (u <> 0) then
          Begin
            if (uptr = nil) then
              uptr := ptr
            else
              luptr^.next := ptr;
            luptr := ptr;
          End
        else
          Begin
            if (aptr = nil) then
              aptr := ptr
            else
              laptr^.next := ptr;
            laptr := ptr;
          End;
      ptr := ptr^.next;
    End;
  isu := (uptr <> nil);          { There is a term involving U }
  isalpha := (aptr <> nil);      { There is a term not involving U }
  if isu then
    luptr^.next := aptr;
```

```
  if isalpha then
    laptr^.next := nil;
  if isu then
    ptr := uptr
  else
    ptr := aptr;
End;    { Order_Entry }

Begin   { Write_Entry }
  order_entry(ptr);
  save_ptr := ptr;
  first := true;
  domac := (x_deriv+t_deriv) >= min_mac_term;
  if isu then
    changed := false
  else
    changed := true;
  while ptr<>nil do     { for each term ... }
    with ptr^ do
      Begin
        if (not first) then
          Begin
            write(outfile,'  ');
            if domac then
              write(macfile,'  ');
            if num>=0 then
              Begin
                write(outfile,'+ ');
                if domac then
                  if not justchanged then
                    write(macfile,'+ ')
```

```
            else
              write(macfile,'  ');
        End
      else
        Begin
          write(outfile,'- ');
          if domac then
            write(macfile,'- ');
        End;
    End
  else
    Begin
      writeln(outfile);
      if domac then
        Begin
          writeln(macfile);
          exists[x_deriv+t_deriv] := true;
          exist1 := true;
        End;
      write(outfile,'( ');
      if domac then
        Begin
          write(macfile,'F',(x_deriv+t_deriv):1,': (');
          if (isu and isalpha) then
            write(macfile,'(');
        End;
      if num>=0 then
        Begin
          write(outfile,'  ');
          if domac then
            write(macfile,'  ');
```

```
        End
      else
        Begin
         write(outfile,'- ');
         if domac then
           write(macfile,'- ');
        End;
      first := false;
    End;
  write(outfile,abs(num):1);
  if domac then
    write(macfile,abs(num):1);
  if (den<>1) then
    Begin
      write(outfile,'/',den:1);
      if domac then
        write(macfile,'/',den:1);
    End;
  xpwr := dx;      { Copy of powers of DELTA_X and DELTA_T }
  tpwr := dt;

  if u<>0 then
    Begin
      if changed then
        writeln('Consistency failure in output routine')
      else
        Begin
          xpwr := xpwr - (x_deriv + t_deriv - 1);
          write(outfile,' * U');
          if u<>1 then
            Begin
```

```
            write(outfile,'**',u:1);
            if domac then
              Begin
                write(macfile,' * C');
                if u<>2 then
                  write(macfile,'**',(u-1):1);
                xpwr := xpwr + u - 1;
                tpwr := tpwr - u + 1;
              End;
          End;
      End;
  End;

if alpha<>0 then
  Begin
    if changed then
      xpwr := xpwr - (x_deriv + t_deriv - 2);
    write(outfile,' * ALPHA');
    if alpha<>1 then
      write(outfile,'**',alpha:1);
    if ((alpha>1) or not changed) and domac then
      Begin
        write(macfile,' * S');
        if not changed then
          acnt := alpha
        else
          acnt := alpha-1;
        if acnt<>1 then
          write(macfile,'**',acnt:1);
        xpwr := xpwr + 2*acnt;
        tpwr := tpwr - acnt;
```

```
            End;
        End;

{ Now we check for any stray DELTA_X or DELTA_T terms that weren't
  soaked up by the C and S terms }

      if (xpwr <> 0) and domac then
        Begin
          write(macfile,' * DELTA_X');
          if xpwr<>1 then
            write(macfile,'**',xpwr:1);
        End;

      if (tpwr <> 0) and domac then
        Begin
          write(macfile,' * DELTA_T');
          if tpwr<>1 then
            write(macfile,'**',tpwr:1);
        End;

      for k:=1 to wts do
        if wt[k]<>0 then
          Begin
            write(outfile,' * ',wt_name[k]);
            if domac then
              write(macfile,' * ',wt_name[k]);
            if wt[k]<>1 then
              Begin
                write(outfile,'**',wt[k]:1);
                if domac then
                  write(macfile,'**',wt[k]:1);
```

```
            End;
        End;

    if dx<>0 then
      Begin
        write(outfile,' * DELTA_X');
        if dx<>1 then
          write(outfile,'**',dx:1);
      End;

    if dt<>0 then
      Begin
        write(outfile,' * DELTA_T');
        if dt<>1 then
          write(outfile,'**',dt:1);
      End;

{ Note that we don't have to write out "* DENOM**-x" to the MACSYMA
  file, as all the terms have the same power of DENOM, so we multiply
  through and remove it entirely!!! }

    if fden<>0 then
      Begin
        write(outfile,' * DENOM');
        if fden<>-1 then
          write(outfile,'**',-fden:1);
      End;
    justchanged := false;

    if (next <> nil) and (not changed) and domac then
      if next^.u = 0 then
```

```
      Begin
        changed := true;
        justchanged := true;
        writeln(macfile,' )');
        write(macfile,'    + (S/C)*(');
      End;


  if (next=nil) and (x_deriv+t_deriv>0) then
    Begin
      write(outfile,' ) * D');
      if domac then
        Begin
          if isu and isalpha then
            write(macfile,' )) = 0$')
          else
            write(macfile,' ) = 0$');
        End;
      if (x_deriv+t_deriv)>1 then
        write(outfile,(x_deriv+t_deriv):1);
      write(outfile,'Tau / ');
      if x_deriv>0 then
        Begin
          write(outfile,'DX');
          if x_deriv>1 then
            write(outfile,x_deriv:1);
          write(outfile,' ');
        End;
      if t_deriv>0 then
        Begin
          write(outfile,'DT');
          if t_deriv>1 then
```

```
                write(outfile,t_deriv:1);
              End;
          End
        else if next=nil then
          Begin
            write(outfile,' )');
            if domac then
              write(macfile,' )');
          End;

        writeln(outfile);
        if domac and not justchanged then
          writeln(macfile);
        ptr:=next;
      End;

  ptr := save_ptr;
End; {write_entry}

Procedure Write_results;

{Displays the table, in "correct" order}

Var deriv,x_deriv,t_deriv : integer;

Begin   {Write_results}
  if not simple then
    Begin
      write(outfile,'DENOM = ');
      write_entry(den_ptr,0,0);
    End;
```

```
  for deriv:=1 to max_deriv do
    for x_deriv:=0 to deriv do
      Begin
        t_deriv := deriv-x_deriv;           {write out the entry}
        write_entry(table[x_deriv,t_deriv,src],x_deriv,t_deriv);
      End;
End; {Write_results}

Begin   { main program }
  inter := false;
  repeat
    write('Enter input file    (keyboard) [.FND] : ',error:=continue);
    readln(iname);
    if (length(iname)=0) then
      Begin
        iname := 'SYS$INPUT';
        inter := true;
      End;
    open(infile,iname,default:='.FND',history:=readonly,
         sharing:=readonly,error:=continue);
  until (status(infile)=0);
  reset(infile);
  repeat
    write('Enter output file     (screen) [.MEQ] : ',error:=continue);
    readln(oname);
    if (length(oname)=0) then
      oname := 'SYS$OUTPUT';
    open(outfile,oname,default:='.MEQ',history:=new,error:=continue);
  until (status(outfile)=0);
  rewrite(outfile);
    repeat
```

```
    write('Enter MACSYMA file      (none) [.COM] : ',error:=continue);
    readln(mname);
    if (length(mname)=0) then
      mname := 'NL:';            {Don't write MACSYMA file}
    open(macfile,mname,default:='.COM',history:=new,error:=continue);
  until (status(macfile)=0);
  rewrite(macfile);
  max_deriv := top_deriv;
  Initialise(src);      {initialise both halves of the table}
  Initialise(dst);
  Enter_data;           {read in the data}
  writeln(outfile,title); {write title onto output file}
  writeln(outfile);
  writeln(outfile,'Initial coefficient of DTau/DT is :');
  write_entry(table[0,1,src],0,0);
  Normalise;            {make sure coeff of DTau/DT is 1}
  writeln(outfile);
  writeln(outfile,'Equivalent Partial Differential Equation :');
  writeln(outfile,'----------------------------------------');
  Write_Results;        {write out the initial equation, for checking}
  if not too_hard then
    Begin
      Process_data;         {find the modified PDE}
      writeln(outfile);
      writeln(outfile,'"Modified" Partial Differential Equation :');
      writeln(outfile,'----------------------------------------');
      exist1 := false;
      for cnt := min_mac_term to top_deriv do
        exists[cnt] := false;
      rewrite(macfile);     {only want final results on MACSYMA file}
      writeln(macfile,'$ set noverify');
```

```
    writeln(macfile,'$ macsyma');
    if not simple then fixup;
    Write_results;    {write out the final results}
    writeln(macfile);
    if exist1 and (wts > 0) then
      Begin           { there are some weights to remove }
        write(macfile,'algsys(');
        ch := '[';
        for cnt := min_mac_term to top_deriv do
          if exists[cnt] then
            Begin
              write(macfile,ch:1,'f',cnt:0);
              ch := ',';
            End;
        write(macfile,'],');
        ch := '[';
        for cnt := 1 to wts do
          Begin
            write(macfile,ch,wt_name[cnt]);
            ch := ',';
          End;
        writeln(macfile,']);');
      End;
    writeln(macfile,'quit();');
    writeln(macfile,
       '$ if "''''notify''" .eqs. "" then ',
       'notify = "$sys_ute:notify"');
    writeln(macfile,'$ notify "<MACSYMA finished>"');
    writeln(macfile,'$ exit');
  End
else if (table[0,0,src] <> nil) then
```

```
    Begin
      writeln(outfile);
      writeln(outfile,'Coefficient of Tau(n,j) is :');
      write_entry(table[0,0,src],0,0);
    End;
End.
```

Bibliography

Ames, W.F. (1977), "Numerical Methods for Partial Differential Equations", Second Edition, Academic Press, Thomas Nelson and Sons.

Bear, J. (1972), "Dynamics of Fluids in Porous Media", Elsevier Publishers

Caussade, B.H. and Renard, G. (1977), "Contribution to the Numerical Solution of Nonlinear Parabolic Partial Differential Equations", *Advances in Computer Methods for Partial Differential Equations II*, editor R. Vichnevetsky, IMACS, pp. 62 – 64.

Colgan, L.H. (1981), "Iterative Methods for Solving Large Sparse Linear Systems", *Numerical Solutions of Partial Differential Equations*, editor J. Noye, North-Holland Publishing, pp. 367 – 396.

Crandall, S.H. (1955), "An Optimum Implicit Recurrence Formula for the Heat Conduction Equation", *Quarterly of Applied Mathematics*, Vol. 13, No. 3, pp. 318 – 320.

Crank, J. (1975), "The Mathematics of Diffusion", Second Edition, Oxford University Press, London.

Crank, J. and Nicolson, P. (1947), "A Practical Method for Numerical Evaluation of Solutions of Partial Differential Equations of the Heat-Conduction Type", Proceedings of the Cambridge Philosophical Society, Vol. 43, No. 50, pp. 50 – 67.

Croft, D.R. and Lilley, D.G. (1977), "Heat Transfer Calculations Using Finite Difference Equations", Applied Science Publishers, London.

Douglas, J. Jr. (1955), "On the Numerical Integration of $\partial^2 u/\partial x^2 + \partial^2 u/\partial y^2 = \partial u/\partial t$ by Implicit Methods", *Journal Society of Industrial Applied Mathematics*, Vol. 3, No. 1, pp. 42–65.

Duff, G.F.D., and Naylor, D. (1966), "Differential Equations of Applied Mathematics", John Wiley and Sons.

DuFort, E.C. and Frankel, S.P. (1953), "Stability Conditions in the Numerical Treatment of Parabolic Differential Equations", Mathematical Tables and Other Aids to Computation, Vol. 7, pp. 135–152.

D'Yakonov, E.G. (1963), "Difference Schemes with Split Operators for Multi Dimensional Unsteady Problems", *USSR Computational Mathematics*, Vol. 4, No. 2, pp. 92–110.

Evans, D.J. and Hatzopoulos, M. (1976), "The Solution of Certain Banded Systems of Linear Equations using the Folding Algorithm", The Computer Journal, Vol. 19, No. 2, pp. 184 – 187.

Evans, D.J. and Abdullah, A.R. (1985), "A new explicit method for the diffusion-convection equation", *Computers and Mathematics with Applications*, Vol. 11, pp. 145 – 154.

Fromm, J.E. and Harlow, F.H. (1963), "Numerical Solution of the Problem of Vortex Sheet Development", Physics of Fluids, Vol. 6, No. 7, pp. 975–982.

Graffi, D. (1980), "Nonlinear Partial Differential Equations in Physical Problems", Pitman Advanced Publishing Program.

Hung, T.K. and Macagno, E.O. (1966), "Laminar Eddies in a Two-Dimensional Conduit Expansion", La Houille Blanche, Vol. 21, No. 4, pp. 391–400.

Lax, P.D. and Richtmyer, R.D. (1956), "Survey of the Stability of Linear Finite Difference Equations", *Communications on Pure and Applied Mathematics*, Vol. 9, pp. 267 – 293.

Leonard, B.P. (1983), "A Convectively Stable, Third-Order Accurate Finite-Difference Method for Steady Two-Dimensional Flow and Heat Transfer", *Numerical Properties and Methodologies in Heat Transfer*, editor T.M. Shih, Springer-Verlag.

Mann, K.J. (1981), "Inversion of Large Sparse Matrices - Direct Methods", *Numerical Solutions of Partial Differential Equations*, editor J. Noye, North-Holland Publishing, pp. 311 – 355.

Marchuk, G.I. (1975), "Methods of Numerical Mathematics", Springer-Verlag, New York.

Noye, B.J. (1984), "Finite Difference Techniques for Partial Differential Equations", *Computational Techniques for Differential Equations*, editor J. Noye, North-Holland Mathematics Studies 83, pp. 95 – 354.

Noye, B.J. and Hayman, K.J. (1986a), "An Accurate Five-Point Explicit Finite-Difference Method for Solving the One-Dimensional Linear Diffusion Equation", *Computational Techniques and Applications: CTAC-85*, editors B.J. Noye and R.L. May, North-Holland Publishing Co., pp. 205 – 216.

Noye, B.J. and Hayman, K.J. (1986b), "Accurate Finite Difference Methods for Solving the Advection-Diffusion Equation", *Computational Techniques and Applications: CTAC-85*, editors B.J. Noye and R.L. May, North-Holland Publishing Co., pp. 137 – 158.

Noye, B.J. and Rankovic, M.J. (1986), "An Accurate Explicit Finite-Difference Technique for solving the One-Dimensional Wave Equation", *Communications in Applied Numerical Methods*, Vol. 2, pp. 557 – 561.

O'Brien, G.G., Hyman, M.A. and Kaplan, S. (1950), "A Study of the Numerical Solution of Partial Differential Equations", *Journal of Mathematics and Physics*, Vol. 29, pp. 223 – 251.

Patankar, S.V. and Baliga, B.R. (1978), "A New Finite-Difference Scheme for Parabolic Differential Equations", *Numerical Heat Transfer*, Vol. 1, pp. 27 – 37.

Peaceman, D.W. and Rachford H.H. Jr. (1955), "The Numerical Solution of Parabolic and Elliptic Differential Equations", *Journal Society of Industrial Applied Mathematics*, Vol. 1, No. 1, March 1955, pp 28 – 41.

Reid, J.K. (1971), "Large Sparse Sets of Linear Equations", Academic Press.

Richtmyer, R.D. and Morton, K.W. (1967), "Difference Methods for Initial-Value Problems", Second Edition, Interscience Publishers.

Roache, P.J. (1974), "Computational Fluid Dynamics", Hermosa Publishers, Albuquerque.

Rosinger, E.E. (1982), "Nonlinear Equivalence, Reduction of PDEs to ODEs and Fast Convergence Numerical Methods", Pitman Advanced Publishing Program.

Saul'yev, V.K. (1964), "Integration of Equations of Parabolic Type by the Methods of Nets", Translated by G.J. Tee, Pergamon Press.

Steinle, P.J. (1984), Computer Program to find the von Neumann Stability Region for a Finite-Difference Equation, Personal Communication.

Thomas, L.H. (1949), "Elliptic Problems in Linear Difference Equations over a Network", Watson Scientific Computing Laboratory, Columbia University, New York.

Trapp, J.A. and Ramshaw, J.D. (1976), "A Simple Heuristic Method for Analyzing the Effect of Boundary Conditions on Numerical Stability", *Journal of Computational Physics*, Vol. 20, pp. 238 – 242.

Warming, R.F. and Hyett, B.J. (1974), "The Modified Equation Approach to the Stability and Accuracy Analysis of Finite-Difference Methods", *Journal of Computational Physics*, Vol. 14, pp. 159 – 179.