ON THE IMPLEMENTATION OF MULTIGRID METHODS FOR THE NUMERICAL SOLUTION OF PARTIAL DIFFERENTIAL EQUATIONS

By

ALLEN DANIEL DELANEY

B.Sc., McGill University, 1964

Ph.D., McGill University, 1969

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

The Department of Computer Science

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

November 1984

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date November 19, 1984

ABSTRACT

A number of experimental implementations of the multigrid algorithm for the solution of systems of partial differential equations have been produced. One program is applicable to simple nonlinear scalar equations, the others to linear equations, scalar and systems, which may be mildly stiff. All use nested grids and residual extrapolation techniques to compute solution and error estimates very economically. One version implements list based adaptive grids to further decrease both computation and storage needed for comparable problems. Each experiment was demonstrated using a set of problems with known solutions and the program performance or nonperformance discussed. Several techniques were examined to ensure that the system of difference equations representing a given problem would be convergent. The use of artificial viscosity was found to be practical in the general case, though for linear problems the use of one-sided differencing may be superior.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

My supervisor, Dr. Uri Ascher, deserves many thanks for his patience, since I was so infrequent in my visits to discuss this project. Also, I much appreciated the flexibility allowed me by my employer, Dr. Ryk Ward, both in time and in the use of computer facilities. The use of the same system I normally work on was quite a boon. Last, but not least, I must thank Tricia for not letting me throw in the towel when life got hectic and everything was taking too long.

1. Introduction

1.1. The method

The numerical solution of boundary value problems for partial differential equations has traditionally been performed by the discretization of the domain upon which the problem is defined and the use of iterative solution methods for the resulting large sparse system of linear equations.

Multigrid methods for the computation of solutions of such systems have been discussed by many authors, e.g. [5-8, 15, 21]. The idea behind such methods, as suggested by Brandt, [3, 6] is to consider the discretizations of the problem on a nested sequence of grids and to exploit the relationships among the corresponding discrete solutions. Thus we can divide the solution process into three elements:

(1) removal of non-smooth error components from fine grids.

(2) improvement of smooth error by corrections computed on a coarser grid.

(3) the use of nested iteration to provide initial estimates for each finer grid.

The removal of non-smooth, i.e. high frequency, errors from fine grids is normally achieved by one of the iterative relaxation techniques often used alone for these problems, such as the Gauss Seidel iteration. These relaxations have been shown to reduce high frequency error components very effectively, but low frequency components much less efficiently. In the multigrid process, only a few relaxation cycles on the finest grid are required, and then coarse grid corrections handle the rest of the error for that grid. Alternate

relaxation methods, such as the preconditioning incomplete LU decomposition methods, have been shown by several authors, [16-18, 23, 24], to be robust smoothing algorithms.

For the coarse grid corrections, what better way could be found than to recursively apply the multigrid algorithm at the lower level. Communication between grids at different levels is carried out by restriction and prolongation operators. Suitable such operators are discussed by Brandt and Mol[5, 19]. Eventually, of course, we reach a coarsest grid on which the discretized problem must be solved exactly. This grid is sufficiently small that the use of the relaxation technique for the accurate solution is an efficient alternative, or the solution for this grid can be determined by direct methods such as Gaussian elimination. Except for cases where convergence is a problem, convenience dictates the use of the relaxation technique, since it must already be available.

The third element of the full multigrid algorithm, nested iteration, provides an efficient way to get an accurate starting value for the finer grids. Once the difference system has been approximated as well as possible on a given grid, the approximation is cubically interpolated to the next finer grid and the multigrid process repeated on the new level. The availability of this nested set of grids allows us to use extrapolation and defect correction methods to accelerate the convergence and improve the accuracy of the approximation of the differential solution [2, 10, 14] to better than that of the algebraic solution of the finest grid[3, 5]. The iterative use of nested grids and the use of coarse grid corrections in the multigrid algorithm improve the efficiency of the multigrid process [6, 20, 22] such that it becomes an $O(N)$ algorithm, where N is the number of

grid points in the finest grid. This compares to $O(N \log N)$ for the most favourable cases or $O(N^{3/2})$ for the more conventional methods.

Multigrid methods have the added advantage that they are readily amenable to adaptive refinement of grids, i.e., during the execution of the algorithm the parts of the mesh which are sufficiently accurate do not need to be considered on finer meshes, only those areas of the grid which require more work are included in the more expensive finer grids.

A disadvantage of the multigrid method is that the implementation of the algorithm is much more complex than the usual iterative solution methods. It is this disadvantage which makes this thesis a profitable venture, that is, to provide a background or framework upon which further experiments using the multigrid architecture can be tried. In this work I have leaned heavily on Hemker's description[15] of the recursive character of the multigrid algorithm, rather than the iterative description of Brandt[5,6].

1.2. Other Implementations

A few other groups have presented implementations of the multigrid method. Brandt has been the "father" of multigrid methods, publishing very prolifically[3-10]. He and his colleagues have discussed every aspect of the process, both theoretically and for specific problems, and have published a software package claimed to be effective for general problems. This package is a development tool, providing an environment and data structure for further multigrid experiments. It suffers from the fact that much of the algorithmic content of the package is devoted to working within the

confines of a Fortran environment.

Another work, by Dendy[11,12], allows the user to code only the rectangular matrix problem, leaving the grid manipulation to a "black box". It seems, though, that there must be a new "black box" for each class of problems, the ones dealt with by Dendy being symmetric and nonsymmetric linear scalar equations. His package is more concerned with dealing with arbitrary grids than with general problems.

Stuben and Trottenberg[21] have produced a very extensive discussion of the theory and application of multigrid methods and present a demonstration of another multigrid package. This offering is an example of the library approach as opposed to the "general package" approach. Foerster and Witsch[13] have offered to distribute this collection of programs for the solution of partial differential equations using multigrid methods.

1.3. This Implementation

The aim of this work, therefore, has been to produce an implementation (program) of the multigrid method for the solution of partial differential equations and to experiment with a variety of aspects arising from such attempts. The package which has been produced is applicable to systems of equations and does tackle the two major problems in this area. These are the fact that simple discretizations do not always lead to systems of equations which can be solved using iterative techniques and that the size of the systems required for partial differential equations may tax or exceed the available computational space and time, virtual or otherwise.

The former problem requires the use of techniques such as one-sided differences or artificial viscosity[1]. Both have been tried

and this work shows that the use of artificial viscosity is a practical approach. The use of one-sided differences has been found to be useful, even superior to the viscosity approach, for linear problems, but more work is needed before any conclusions concerning nonlinear problems can be made.

The latter problem, computer time and space demands, requires that some sort of adaptive mesh selection be used so that parts of the domain which are "easy" can be solved with a coarse mesh, requiring little storage space or computational time, and the "hard" areas of the domain solved with a progressively finer mesh. An implementation presented with this thesis uses a list based data structure which allows completely adaptive meshes to be used.

The choice of which language to use to best implement an algorithm is a question of general interest. Numerical analysts have often favoured Fortran, because it is familiar to most other numerical analysts, it is widely available, and its compilers are often very efficient. On the contrary side, Fortran's lack of data structures and of recursion severely restrict the programming style and clarity of the code produced. An algorithmic, structured language like Algol allows the production of clearer code, probably with less programmer effort, particularly for list based and recursive algorithms. Pascal is an Algol-like language which is beginning to approach Fortran in popularity. With this popularity and with hardware design beginning to favor recursive programming, the efficiency of Pascal compilers and their code has become less of a disadvantage.

For these reasons, Pascal was chosen as the implementation language for this project. An early implementation was actually coded in the C language, but then the decision to switch was made. C has some advantages, particularly in the dynamic allocation of arrays, but has disadvantages in code clarity and in the breadth of its acceptance.

Although the later versions of the package are applicable only to linear systems, the adaptation to nonlinear problems should not be difficult. The basic algorithm was implemented in the earliest version, and the transfer of these details to the final version should be straightforward.

No attempt at optimizing performance has been made, but the literature contains many examples of high frequency smoothing schemes and low frequency correction techniques which might be tried in the attempt to optimize performance for specific difficult problems. In general the literature [7, 9, 17, 19] deals with the theory of performance and admits that practical problems require individual attention to efficiency for each case. This package is designed as a framework upon which to build the specialized systems for real problems; it is modular and extensible. Of course a number of classes of problems are solvable using the presented package, but the future user should be able to add his own classes of problems with minimal difficulty.

As to efficiency, a major design goal for this program was to allow flexible use of adaptive grids. This approach to efficiency seems much more crucial than doing a few less relaxation sweeps on full grids.

Consideration has not been given to problems with arbitrary boundaries, but the data structure used here could be adapted to the irregular edges of a domain in a straightforward manner. The technique would involve adding finer and finer mesh points as the boundary is approached until a grid point is sufficiently close to the boundary, but is still part of the organized grid. These grid points near the boundary would have to be flagged so they could be included in all grids.

This thesis is actually a report on the evolution of a number of experimental codes and the experience I have had with them. The next section presents the general algorithm and discusses the implementation details for a sequence of programs, each with a different emphasis, but each growing from the previous. After that, some results for each code are presented to demonstrate their performance, or non-performance. In the discussion I have tried to summarize what I have done and outline the many avenues which could be traveled in a continuation of this work.

2. A Multigrid Method and Its Implementation

2.1. The Algorithm

The numerical solution of a system of partial differential equations is generally accomplished by solving the system of difference equations arising from a discretization of the domain into a set of grid points, i.e., a mesh (or grid). One wishes to find an "optimal" mesh where the error in the computed solution is satisfactorily small, and the number of grid points is small enough to keep the computational cost reasonable. One way of approaching this optimum is to solve on very coarse grids, use the solution to approximate a starting point for the next finer mesh, and repeat the cycle until the estimate of the error is sufficiently low. The multigrid method presented here does this and it uses the availability of multiple grid solutions to improve both efficiency and accuracy.

This multigrid method is outlined in Figure 1. In this algorithm L^h denotes the difference operator representing the system of equations on a grid with mesh size h. The letter n denotes the number of divisions in one dimension of the grid, so $h = 1/n$ on the unit square. We will refer to n as the grid size. L^H represents the same operator, but working on the next coarser grid, with mesh size $H = 2h$. The symbol f represents the right hand side of the system.

Several parameters and descriptive constants provide the limits for the algorithm. The constants nmin and nmax are the grid sizes for the coarsest and finest grids. η and α and EF are parameters dependent on the order of the discretization used. EF is the extrapolation factor, which is always unity on correction grids, but may be

```
procedure multigrid;

   let n = nmin

   initialize u and f on the grid;

   solve exactly L^h u^h = f^h ;

   while n < nmax and an error estimate > an error tolerance do
    begin

      let n = 2 n ;

      interpolate u to level n from level n/2 ;

      solve at level n the problem L^h u^h = f^h ;

   end ;

end multigrid;


procedure solve ( at level n ) ;

   let τ_n = ∞

   while τ_n > α τ_{n/2} do
   begin

     while convergence < η  do relax at level n

     let r^H = f^H - EF I^h_H ( L^h u^h - I^H_h L^H u^H ) ;

     solve at level n/2 the problem L^H u^H = r^H ;

     let u^h = u^h + I^H_h u^H ;

   end;

end solve;
```

Figure 1: The Multigrid Algorithm

An algorithmic description of the multigrid method. See the text for a discussion of the parameters defining and limiting the process.

greater than unity on top-level grids. It allows the over-correction

of the residual to obtain higher order accuracy for smooth problems.

The notation I_H^h and I_h^H refers to restriction and prolongation operators. In one part of the multigrid process residual estimates computed using u values from the finer of a pair of grids must be added to the u values of the coarser grid, and in another part a correction computed on the coarser of the pair of grids must be added to the finer grid. These two operators compute the u values for a grid at the required level from the u values at the known level. In this package restriction may be either injection, i.e., fine grid values are merely copied to the corresponding locations of the coarse grid, or an averaging of 9 points. Prolongation is linear interpolation of points in the coarse grid to produce a fine grid, which is equivalent to 9-point prolongation.

The relaxations used by this implementation in the solve routine of the algorithm are Gauss Seidel iterations over the system of equations for the grid. For linear problems the system of linear equations at each grid point is solved using Gaussian elimination. If the version of the program handles nonlinear systems, the relaxations actually do one or more Newton iterations to relax the nonlinear system at each grid point.

The heart of the multigrid system is the set of relaxations, and the fact that these relaxations, though they have very poor convergence properties overall, have an excellent, i.e. fast, convergence rate for the highest frequency components of the error.

When working with discretizations, as we are now, it is appropriate, under certain conditions, to represent the error as a Fourier series, i.e., a sum of trigonometric terms with differing

wavelengths. We can divide the error, then, into three groups of components. The low frequency error components are those with wavelengths less than $h/2$, the high frequency components are those with wavelengths between $h/2$ and $2h$, and the invisible components are those with still higher frequencies. These invisible errors must be avoided, rather than dealt with, as we shall see later.

The idea then, is to do a few relaxation cycles at the finest grid level, until the convergence criterion returned by the relax routine indicates that further relaxations are of little use, and then to correct the low frequency errors by solving on the next coarser grid. The program parameter η controls the definition of the relaxation convergence criterion. When the ratio of the changes for two consecutive relaxations is greater than η, high frequency convergence is said to be obtained.

This process is recursively continued back down to the coarsest grid level, which is solved exactly. In this way usually only two or three relaxations are done at the finest level on each cycle and the rest of the work is done on lower, cheaper levels.

The extrapolation factor is dependent on the order of the difference approximations used. If the grid has been solved accurately, then this extrapolation increases the accuracy of a second order difference solution to fourth order. The value used in the extrapolation of the residual is an estimate of the residual, evaluated from two grid computations as per Figure 1. This estimate, τ, is used to decide when further coarse grid corrections are useless, and, if yes, to go on to a finer grid. For second order differences, it can be expected that the residual will decrease by a factor of one fourth

for each grid level.

The program parameter α is given the value of this expected ratio and is used in the computation of the grid tolerance for the estimated residual. This tolerance is α times that obtained for the previous, coarser, grid.

Another constant often used in the multigrid analysis, δ, is the factor by which the error is expected to decrease for each grid level. For a fourth order method this would be one sixteenth, and the error estimate would be

$$\frac{\delta(u_c - u_f)}{1-\delta}$$

My implementations are very conservative, in that they do not use δ. Instead the correction term $u_c - u_f$ is reported, thus overestimating the error. The reader may note that that the value of δ changes with the order of differencing, and that δ has little meaning when artificial viscosity is being used.

Extrapolation is only performed at the finest level. The function of the lower level solves is to approximate the solution of the algebraic system of equations at the next finer grid. The function of the extrapolation is to improve the top level algebraic solution as an approximation to the solution of the differential system.

The algorithm described in Figure 1, is known as the FAS algorithm [5,6]. All grids in the algorithm are dealing with a similar problem, an approximation of the problem being solved on the finest grid. The correction to be applied to the finer grid is computed, in the FAS case, by subtraction from the best fine grid solution. This contrasts with the residual correction approach, where the coarse grids are used to solve a problem related to the error in the fine

grid, and the correction is directly calculated. The residual correction approach is therefore applicable only to linear problems.

2.2. Grid Implementations

One property of multigrid processes is that they perform similar operations on grids of varying size. In most of this study, the largest grid size used was 64, and normally the minimum grid size was 4. During the course of a computation, only one of the finest grids, with $n = 64$, is required. For the coarse grid corrections, though, one grid of each size $n = 4,8,16$, and 32 are required. Interpolation, prolongation, and restriction operations must use these different sized grids. I will discuss three possible data structures for these grids.

2.2.1. Minimum storage

The obvious approach, if we are using only uniform grids, is to define an array for each grid which will contain that grid and no more. A grid with $n = 4$ would, in Pascal, be defined:

```
array [0..4] of array [0..4] of real;
```

and one with $n = 8$ as:

```
array [0..8] of array [0..8] of real;
```

This uses a minimum of storage, so long as full grids are used. The disadvantage of this system is that the translation of the integer indices i,j to the (x,y) values they represent changes from grid to grid, and so the interpolation and restriction functions are more complicated than is desirable.

From a strictly programming viewpoint, since Pascal was chosen as the implementation language, problems of data typing and dynamic array allocation make this type of data structure very inconvenient. The Pascal procedures which work with grids will not allow grids of different size to have the same type, and it is not straightforward to give arrays of different size the same type. There are ways around this problem, but they are unlikely to be portable, and portability was one of the major reasons for choosing Pascal as the implementation language.

2.2.2. Minimum trouble

At a considerable sacrifice in storage, and given that we are using uniform grids, all the disadvantages of the minimum storage approach can be eliminated. Here all grids have the same type, and the relation between (x,y) and (i,j) is the same regardless of the grid size. We merely leave empty spaces for all the unused grid points on a given level, and all grids occupy a maximum amount of space.

The storage cost can be easily computed. If we let $m = \log_2(nmax/nmin)$ then, using the minimum storage data structure the grids would require

$$nmax^2 \sum_{i=0}^{m} 0.5^{2i} \leqslant \frac{4}{3}\, nmax^2$$

If each grid required the full $nmax^2$ grid locations, the proportional cost of the minimum trouble grid relative to the minimum storage grid would be

$$\frac{m\ \ nmax^2}{\frac{4}{3}\ nmax^2} = \frac{3m}{4}$$

Figures 2 and 3 present pictures of the set of grids which would be

used by a computation with *nmin*=4, *nmax*=32. Every grid contains all the (x,y) co-ordinates which are present in the previous grid, thus the coarsest grid points are present in all grids. With all four of these grids superimposed, and with the respective grid points represented by "-", "*", "+", and "@", the representation becomes Figure 4.

2.2.3. Adaptive Grids

A third data structure for the grids does not use arrays at all. It uses a list oriented structure to hold the grid information. With the exception of the overhead for pointers, the storage cost for this data structure is the same as for the minimum storage array method.

Each grid point in Figures 1-4 is represented by a single node of the data structure. This node contains the current value of u at that point, the indices (i,j), from which the (x,y) values can be computed, pointers to the nodes to the right and above, the value of the right hand side for that node, and other information which is used in the computation, such as error and residual estimates. The actual pointer to the grid is the pointer to the lower left corner node of the grid.

It would have simplified programming quite a bit if each node had four pointers, one for each of its neighbors, but a decision to save those eight bytes of memory per grid point was made. Hindsight indicates that, at least for the prototype implementation, it would have been more efficient of my time to work out the implementation using four pointers and rewrite for efficiency later, if at all.

This type of data structure allows the flexibility required for the implementation of adaptive grid algorithms. For problems which

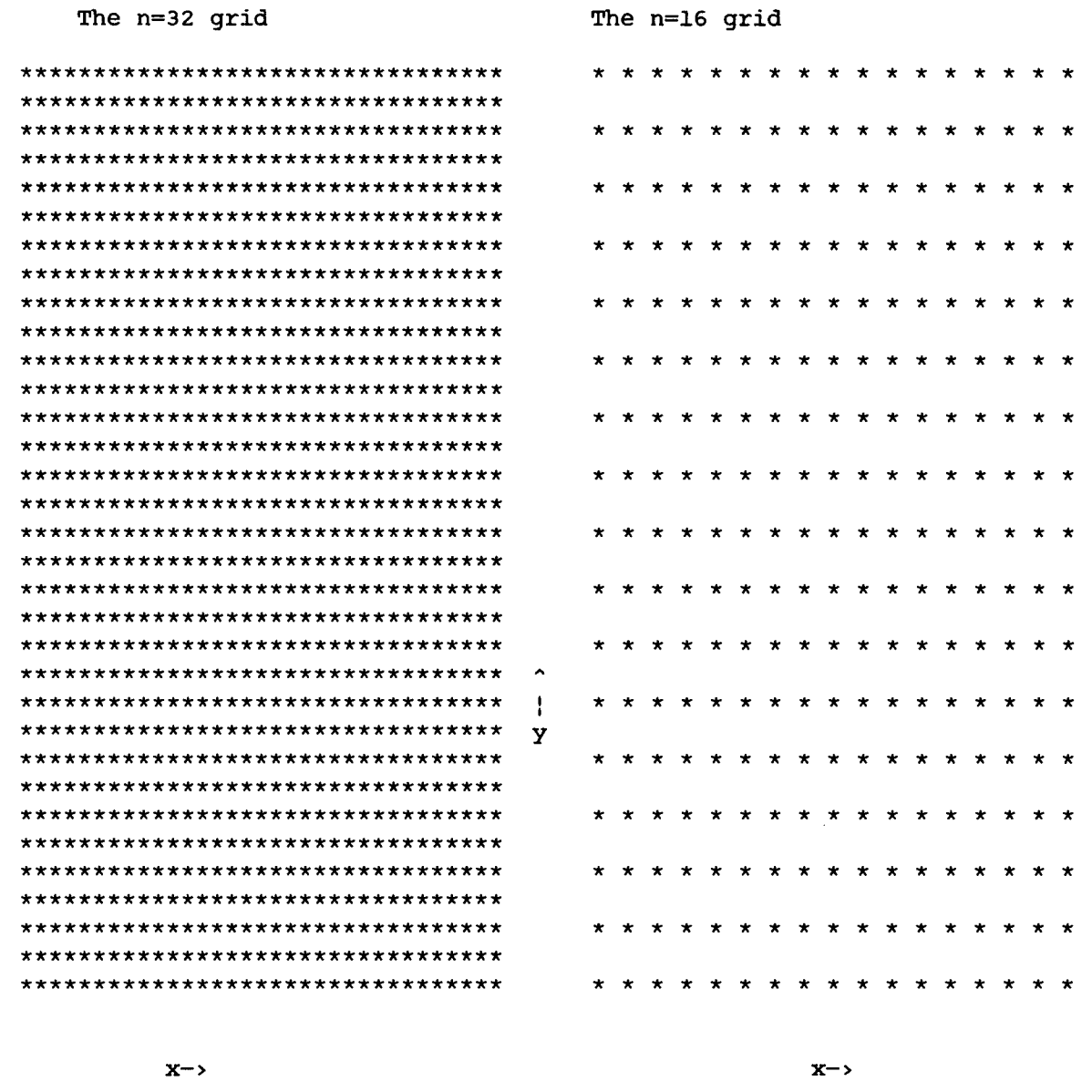

Figure 2: Finest grid examples

A diagram representing the grid points used at the two finest levels when *nmax*=32

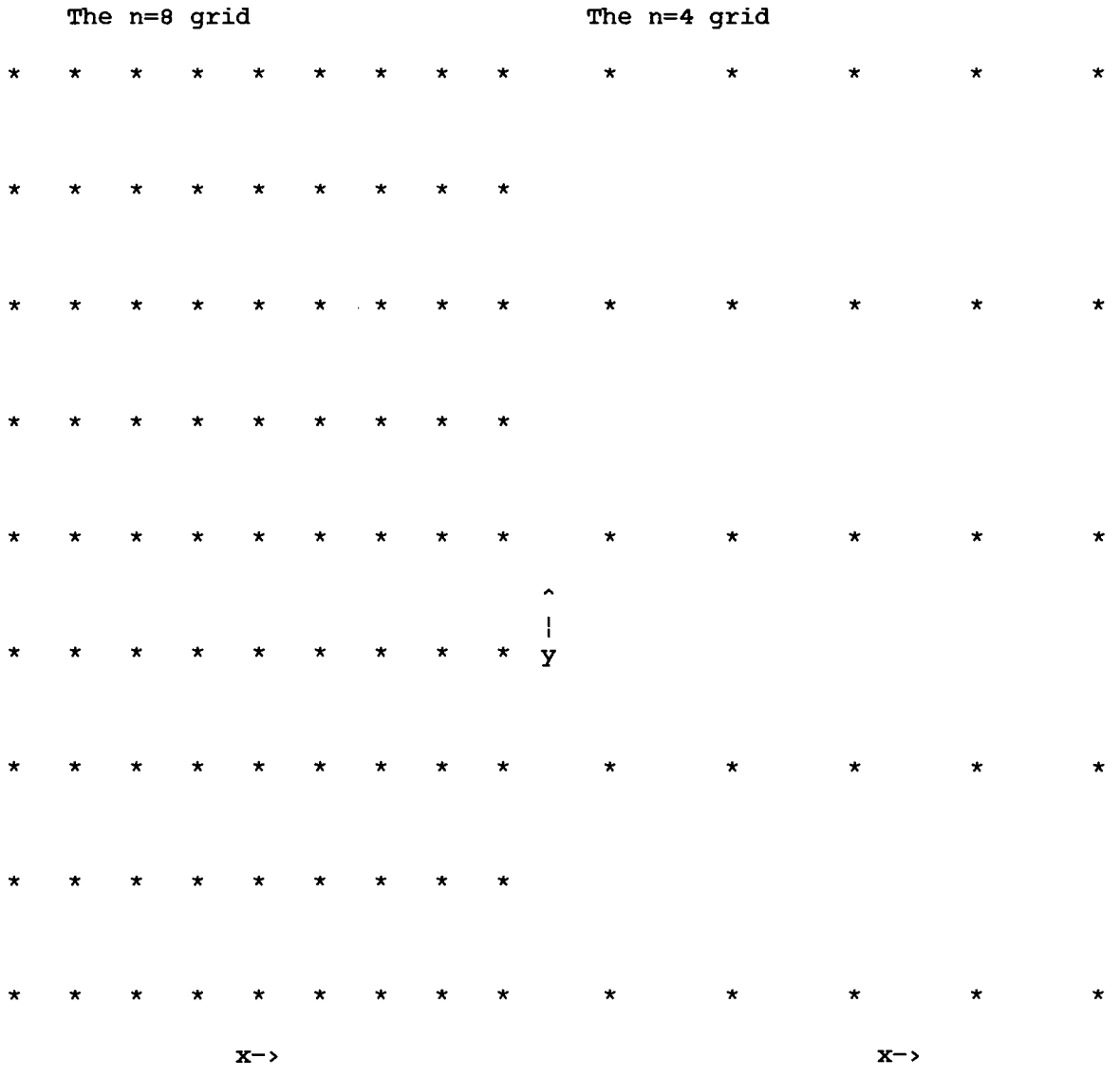

Figure 3: Coarsest grid examples

A diagram representing the grid points used at the two coarsest levels when *nmin*=4

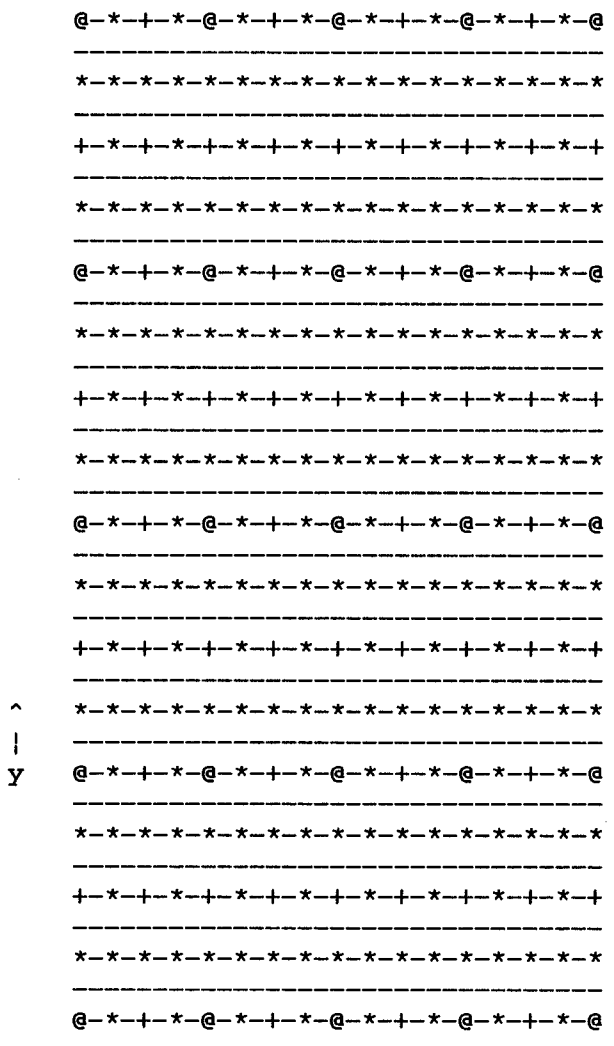

Figure 4: Composite of many grids

A representation of a grid with size $n=32$. All points are members of the finest grid. Only the "*" points are in the $n=16$ grid, the "+" points are the $n=8$ grid, and the "@" points are the coarsest grid, at $n=4$.

require very fine meshes in selected areas of the grid, it is essential that the fine meshes need not encompass the entire domain. A

list oriented data structure allows us to allocate only those regions of the grids for which error estimates indicate that more refinement is necessary.

In the final version of the program presented with this thesis, the error estimates at each grid point are used by the interpolation procedure to decide which new points are needed at the next level. Figure 5 shows a hypothetical pair of grids, demonstrating how the adaptive interpolation proceeds. On the n=8 grid the "@" grid points are those with large error estimates. These same grid points are represented by "@" symbols again in the n=16 grid. On this finer grid, there are two types of new points, in addition to all the coarse grid points. The points represented by "x" characters are those which will be relaxed on the fine grid. The other new points are there merely to allow the relaxations to proceed on the "x" points.

If any point has an error estimate larger than the tolerance for the grid, all points within the square defined by the 9 point group around the grid point in question will be placed in the resulting grid. This kind of arrangement will lead to some grid points which do not have the neighbors required for relaxation. Such grid points, however, will always either have satisfactory error estimates, or be the cubic interpolate of two such points. The relaxation sweeps cover the entire grid each time, but the computations are done only on the "x" and "@" grid points.

The reader may notice that there are seemingly unnecessary points on the boundaries. These are not used for the computation, but are needed for traversing the grid, ensuring orderly access to all the

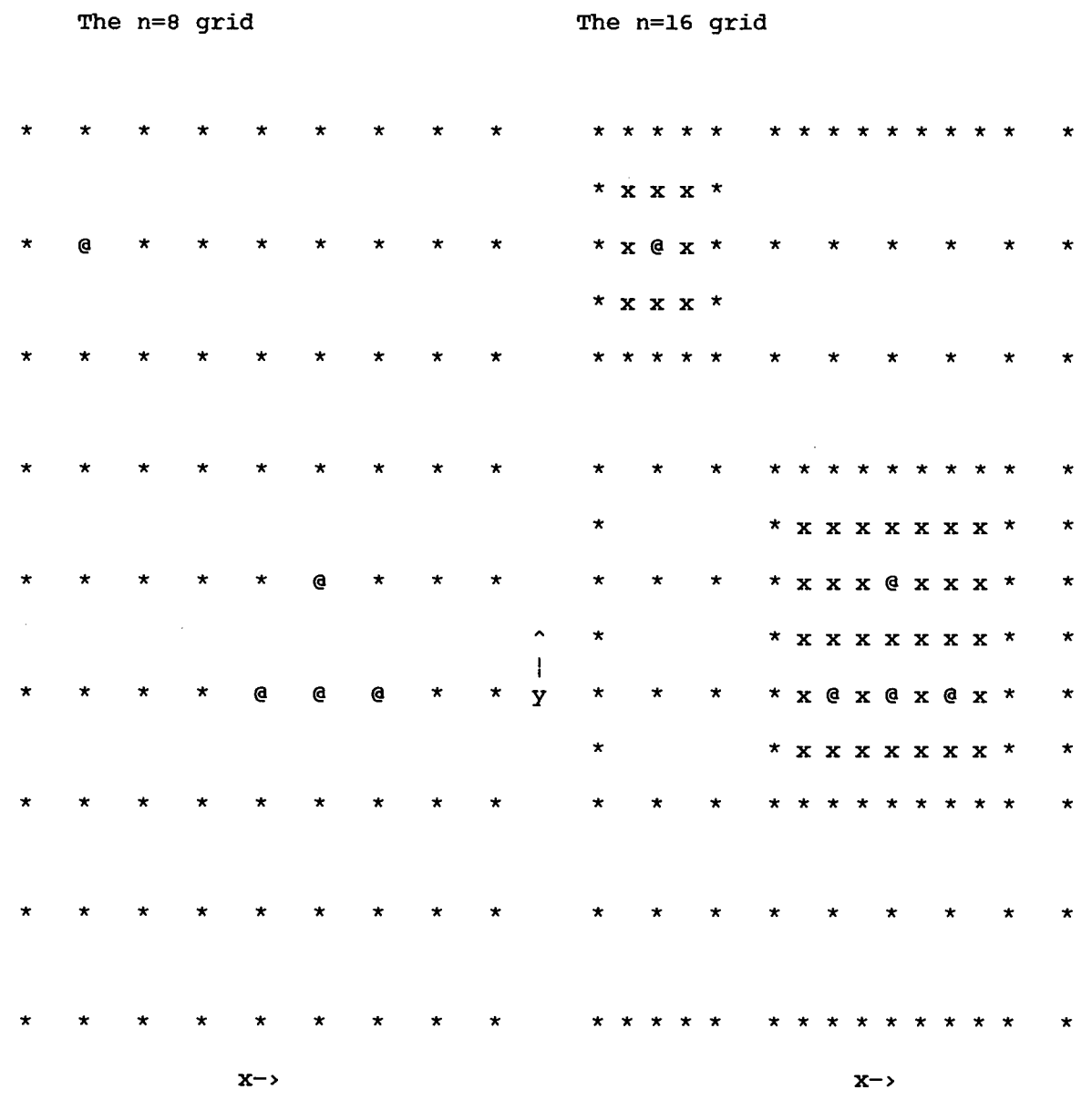

Figure 5: Adaptive interpolation

A representation of a pair of grids, demonstrating the adaptive interpolation procedure used to produce the next finer grid.

grid points.

During the course of this work, four program versions were implemented. The first three of these used the Minimum Trouble storage method, the final version used the adaptive, list based, data structure.

3. Numerical experiments and their results

As mentioned before, several different multigrid implementations were produced, each growing from the previous one. Here is a description of the programs I have built, the problems incurred, the strategies tried, and the results obtained.

3.1. Simple nonlinear scalar equations

This initial version of the package attacked the problem:

$$a\ u_{xx} + c\ u_{yy} + d\ u_x + e\ u_y + g = 0$$

where a, c, d, e, and g are scalar functions of x, y, and u. Since g can be an arbitrary function of x, y, u, it was deemed unnecessary to include a right hand side in the implementation. The user was required to provide the functions for $g(x,y,u)$ and its partial derivative with respect to u, as well as a function specifying the boundary values of the solution. Figure 6 lists some example problems of this prototype which were used to demonstrate this first implementation.

The motivation for presenting these particular results is to demonstrate that at least one version of these implementations was effective on nonlinear problems. Table I summarizes the results of this study. It can be seen that the convergence of the process is close to fourth order for most cases. Example 2 is an exception, and this is because the exact solution is a low order polynomial, hence the coarse grid solutions are close to the exact one not just because of small h. In all cases the convergence rate dropped off at a grid-size of 64. This behaviour was caused by the parameter settings in the program being tuned for efficiency, so a minimal number of relax-

$$u_{xx} + u_{yy} - u^2 = -\sin^2(x)\, e^{2y} \qquad (1)$$

Solution: $u = \sin(x)\, e^{y}$

$$u_{xx} + u_{yy} - e^{u} = 4 - e^{x^2+y^2} \qquad (2)$$

Solution: $u = x^2 + y^2$

$$u_{xx} + u_{yy} - u^2 + 2u = -\sin^2(x)\sin^2(y) \qquad (3)$$

Solution: $u = \sin(x)\sin(y)$

$$u_{xx} + u_{yy} - u^3 + 2u = -\sin^3(x)\sin^3(y) \qquad (4)$$

Solution: $u = \sin(x)\sin(y)$

$$u\,u_{xx} + u_{yy} - e^{u} + 2u = -e^{(\sin(x)\sin(y))} \qquad (5)$$

Solution: $u = \sin(x)\sin(y)$

Figure 6: Sample problems (1) to (5)

Example problems 1 through 5. These were used in the demonstration of the first implementation, which could handle nonlinear problems but only for equations producing a diagonally dominant discretization matrix.

ation sweeps were done on the finest grids. A readjustment of the parameters to allow more multigrid cycles would improve the convergence at the cost of additional computation.

An attempt at adaptive grids was made with this version, and it was very educational. The idea was to have two limits, SMALL and BIG. All multigrid operations were performed on grids of size SMALL or less. When it became necessary to go from the SMALL grid to that of size 2*SMALL the grid was partitioned into four grids of size SMALL and each grid was treated separately. Once the four grids were solved, a region covering the boundaries between the grids was relaxed a few times. If the error estimate on a given grid was

Problem	grid	error	error ratio
1	8	3.0e-5	16.6
	16	2.4e-6	12.8
	32	1.8e-7	13.1
	64	2.7e-8	7.0
2	8	3.3e-7	8.2
	16	6.4e-8	5.2
	32	5.1e-9	12.7
	64	1.1e-9	4.6
3	8	1.1e-5	16.0
	16	8.0e-7	13.8
	32	6.4e-8	12.5
	64	1.4e-8	4.6
4	8	1.1e-5	16.0
	16	8.0e-7	13.8
	32	6.4e-8	12.5
	64	1.4e-8	4.6
5	8	1.1e-5	16.0
	16	8.0e-7	13.8
	32	6.4e-8	12.5
	64	1.4e-8	4.6

Table I: Single nonlinear equations

A summary of the errors obtained and the error ratio between adjacent grids for several problems, using the original implementation which was applicable to single nonlinear equations.

satisfactory, it was not necessary that it be solved.

If the error was concentrated in one or more of the corners, this approach might have been feasible, but it was found that error on the

boundaries between subgrids did not improve with decreasing mesh size. In retrospect this could have been expected, since the overlapping relaxations can remove only the high frequency error from the subgrid boundaries, and there is no coarse grid correction to remove the low frequency errors.

This approach was abandoned for a number of reasons: It is not applicable to systems of equations, the use of adaptive meshes was not successful, and no provision is made for problems which yield a system of equations which is not diagonally dominant, which leaves out many interesting problems. Also, the fineness of the mesh is limited due to the very inefficient use of storage space for the grids. This storage method was chosen for ease of programming, and it severely limits practical grid size.

There were some accomplishments from this part of the project, besides experience. It was demonstrated that the use of FAS and a newton iteration in relaxation is a feasible way to approach nonlinear problems. Further, the accuracy obtained using the τ extrapolation method was demonstrated to be fourth order for the smooth test problems. One approach to adaptive grids was eliminated from consideration.

3.2. Linear scalar equations with mild stiffness

Proceeding in a stepwise fashion, the next program implementation addressed the question of robustness. Several different approaches were used for problems which, with the usual centered differencing, did not yield a system of equations which had a diagonally dominant matrix. To simplify programming, the same storage inefficient data structure was used, and problems were limited to a single linear

equation. Thus the class of equations was

$$a\ u_{xx} + c\ u_{yy} + d\ u_x + e\ u_y + f\ u = g$$

Again the user was required to provide the functions defining the problem, but the function $g(x,y,u)$ described in the previous section was split into $g(x,y,u) = f(x,y)\ u - rhs(x,y)$. Since the problems must be linear, no derivatives were necessary. The problems approached with this package are shown in Figure 7.

Examples (8) to (12) have the same representation, using R_e value of 1, 8, 16, 32, and 64. To avoid overflow during computation the solution in the last two cases was scaled by dividing by e^{R_e}. With the values of mesh size h used here, the latter three problems yielded systems of equations which were not diagonally dominant on one or more of the coarsest grids when the usual centered differences were used.

$$u_{xx} + u_{yy} - 2u = 0 \qquad (6)$$

Solution: $u = e^{(x+y)}$

$$u_{xx} + u_{yy} = 0 \qquad (7)$$

Solution: $u = sin(x)\ e^{y}$

$$\frac{u_{xx}}{R_e} + \frac{u_{yy}}{R_e} - u_x + u_y = 0 \qquad (8\text{–}12)$$

Solution: $u = (1-e^{xR_e})(1-e^{-yR_e})$.

Figure 7: Sample problems (6) to (12)

The sample problems used to test the implementations which could handle single linear equations. Examples (8) to (12) use R_e values of 1, 8, 16, 32, and 64.

Several methods were utilized in order to solve this convergence problem. Two new relaxation routines were implemented. One group of methods utilized one-sided differences at grid points which do not satisfy the diagonal dominance criterion. Another method utilized only second order differences, but used artificial viscosity on the grid points which required it. In the constant coefficient problems examined here, where any part of a grid requires artificial viscosity, the entire grid would require it.

As a side issue in this study, two types of correction grid solve routines were examined. The original such routine is different from the top level solve in that relaxations are done both before and after the coarse grid correction step. The other approach was to use identical solve routines at both stages, but avoiding the τ extrapolation in the correction routines. The latter involves fewer relaxation cycles in total, is less complicated in programming, and allows more control of the correction cycles. My findings were that the second approach was equivalent, if not slightly better, in accuracy, than the first, so it was adopted for general use.

Using first order differences in the relaxations created some significant problems. When, during the course of multigrid execution, it became necessary to solve on a finer grid, this being the first grid which is fine enough to use centered differencing, the jump from a first order grid to a second order grid did not proceed smoothly. It was desirable to use the FAS algorithm, since it is needed for extension of the method to nonlinear problems but, with FAS, the problem solved on each correction grid is a close approximation to the original problem and the error on a first order correction grid was found to be too large to be useful as a starting point

for the next grid.

For linear problems, the use of the corrections to solve only the residual problem allowed first order grids to be useful. In order to preserve the fourth order convergence properties of the process, it was necessary to use an extrapolated residual value on the finest grid.

example	n	cgerr[n]	err[n/2]	ratio	
6)	8	1.6e-5	4.9e-4	31	
	16	1.1e-6	1.7e-5	15	
7)	8	1.1e-5	3.2e-4	29	
	16	8.6e-7	1.2e-5	14	
8)	8	5.0e-6	8.3e-5	17	R_e=1
9)	8	3.8e-2	8.2e-1	22	R_e=8
	16	4.8e-3	1.0e-1	21	
	32	5.0e-4	5.5e-3	11	
	64	5.3e-5	5.0e-4	9.4	
10)	8	2.8e-1	1.3e-1	0.5	R_e=16
	16	2.4e-1	2.8e-1	1.1	
	32	2.4e-1	2.4e-1	1.0	
	64	2.4e-1	2.4e-1	1.0	

Table II: One-sided differences with FAS for linear single equations

Results for some of the sample problems from the implementation for linear single equations which used one sided differencing when necessary, using the FAS correction scheme and using either centered or one sided differences when computing τ values. Presented in the table are the example number, the grid level n, the maximum error on the coarse grid at level n, the maximum error on the entire previous grid, and their ratio. At the left are some descriptive comments about some of the examples.

Tables II to VI present summaries of the results from some of the combinations tried. In each of the tables are presented the error found at a given grid level, the error found at the previous grid level, and their ratio. Comparisons were made only at grid points which occurred in both grids, since the newest gridpoints on the n-level grid have no counterparts for comparison. A ratio of sixteen would indicate fourth order convergence, four would indicate second order convergence, and two first order.

The results in Table II are from a program which used first order relaxation when necessary, but used centered differencing exclusively when computing the value of τ. This being the case, extrapolation of 1.333 was used at the top level in all cases. Results were not too encouraging, indicating that coarse grid errors in correction cycles were propagated up to the fine grid levels. Results for the easy problems, those which do not require the use of first order differencing, were good, the others were not.

The problems (11) and (12), with R_e of 32 and 64, were both very similar to (10), with slightly higher errors.

Identical results were obtained from the implementation where first order differences were used when necessary, and the computation of τ used first order differences when the fine grid required it. Extrapolation of 1.333 was used for grid points which are fully second order, and a value of 2.0 was used for grid points which were first order, at the top level only. Again the easy problems were fine, the more difficult problems had error which was not removed on fine grids, and the results were essentially identical to those presented above.

A third implementation used 1st order differencing only in the relaxations and only when necessary to obtain diagonal dominance, but the coarse grid correction was a residual correction only. This is not the FAS algorithm, but corresponds more closely to Brandt's cycle C algorithm [5,10]. At the top level the residual computed is an extrapolated one, which improves the order of the top level accuracy from second to fourth order. The results, presented in Table III, are somewhat more encouraging.

Example problem (10) requires first order differencing only on the grid at $n = 4$, (11) at $n = 4$ and $n = 8$ and (12) when $n = 4,8,$ and 16. Results are much improved over the first attempts. The lowest grid is solved exactly, and this seems to yield a better error than expected. After this grid, we seem to be getting approximately first order convergence until both grids involved in the transfer are second order.

It is unfortunate that this residual correction is applicable only to linear problems. The behaviour noted here is reasonable, considering that the error generated on the coarse grids is $O(h)$ in the residual, which itself is $O(h)$, so the total correction should be $O(h^2)$. In the FAS case, each correction cycle solves a problem very close to the original, and a subtraction is performed to get the correction factor. If an $O(h)$ error is generated on a coarse grid, this entire error would be brought up with the correction.

In Table IV result summaries are presented for a method which uses first order differencing when necessary, in both the relaxations and in the computation of the residual. Comments about applicability to linear problems only are relevant here as well.

example	n	cgerr[n]	err[n/2]	ratio	
6)	8	1.6e-5	4.9e-4	31	
	16	1.1e-6	1.7e-5	15	
	32	7.3e-8	1.1e-6	15	
7)	8	1,1e-5	3.2e-4	29	
	16	8.5e-7	1.2e-5	14	
	32	5.7e-8	8.6e-7	15	
8)	8	5.0e-6	8.3e-5	17	R_e=1
	16	4.7e-7	6.2e-6	13	
9)	8	3.8e-2	8.2e-1	22	R_e=8
	16	4.8e-3	1.0e-1	21	
	32	5.2e-4	5.5e-3	11	
	64	5.5e-5	5.2e-4	9.4	
10)	8	3.2e-2	1.3e-1	4.0	R_e=16
	16	4.9e-3	8.1e-2	17	
	32	3.6e-4	1.0e-2	28	
	64	4.5e-5	9.9e-4	22	
11)	8	1.5e-1	9.0e-2	0.6	R_e=32
	16	2.7e-2	1.5e-1	5.6	
	32	5.0e-3	8.5e-2	17	
	64	3.6e-4	1.0e-2	28	
12)	8	4.0e-2	5.0e-2	1.2	R_e=64
	16	2.0e-2	1.3e-1	6.5	
	32	2.9e-2	1.6e-1	5.5	
	64	5.0e-3	8.7e-2	17	

Table III: One-sided differences with residual corrections only in the relaxations

Results for some of the sample problems from the implementation using first order differences when necessary, using coarse grid corrections to estimate residuals only.

example	n	cgerr[n]	err[n/2]	ratio	
10)	8	3.2e-2	1.3e-1	4.0	R_e=16
	16	4.9e-3	8.1e-2	17	
	32	3.6e-4	1.0e-2	28	
	64	4.5e-5	9.9e-4	22	
11)	8	1.9e-2	9.0e-2	4.7	R_e=32
	16	2.7e-2	1.3e-1	4.8	
	32	5.0e-3	8.5e-2	17	
	64	3.6e-4	1.0e-2	28	
12)	8	2.4e-2	5.0e-2	2.1	R_e=64
	16	1.7e-2	8.1e-2	4.8	
	32	2.7e-2	1.4e-1	5.2	
	64	5.0e-3	8.5e-2	17	

Table IV: One-sided differences with residual corrections in relaxations and in coarse grid corrections.

Results for some of the sample problems from the implementation using first order differences when necessary, in both the relaxations and the coarse grid corrections. Here again the correction was a residual correction only.

Problems (6) to (9) were identical to the above, since no first order differencing is used, and the results for the other three problems are in Table IV.

For problems (10),(11),and (12) the error was smoothed out over the first order grids, but the end result when the second order grids were reached was the same. This may imply that the extra effort of doing first order coarse grid corrections may not be worth the trouble. Enough experimentation has not been done to indicate whether this approach would be preferable for even higher values for R_e.

As far as the number of relaxations was concerned, this scheme was observed to be slightly more expensive than the corresponding scheme without the matching coarse grid corrections, and it would

have more overhead, in the testing and recomputing at first order grid points.

Another implementation makes use of artificial viscosity to handle the diagonal dominance requirement. Table V summarizes the results for the problems of interest. Again results for examples (6) to (9) were identical to those for the rest of the schemes. For examples (10) to (12) the results in Table V show that the error is not improving at all on the grids which require the use of artificial viscosity, but convergence bounces back very well once that constraint is gone. This is actually reasonable, since at those grid points different problems are being solved on the two grids.

For completeness, another experiment was done where relaxations used artificial viscosity, as above, but the τ computation and coarse grid correction did not take this into account. As usual, results for examples (6) to (9) were identical to previous results, but for the other examples, results were terrible, the coarse grid correction obviously spoiling things. This contrasts with the residual correction methods discussed above, where the matching of the coarse grid correction with the relaxations seemed to be of little importance.

3.3. Linear systems with mild stiffness

After completing the experiments outlined in the previous section, it was time to modify the package to operate on systems of differential equations. The class of problems handled by this version is:

$$a\ u_{xx} + c\ u_{yy} + d\ u_x + e\ u_y + f\ u = g$$

Here, however, the coefficients are matrices, the right hand side and

example	n	cgerr[n]	err[n/2]	ratio	
10)	8	1.8-2	1.5e-2	0.83	R_e=16
	16	5.0-3	1.1e-1	22	
	32	3.6e-4	1.0e-2	28	
	64	4.5e-5	1.0e-3	22	
11)	8	3.4e-4	2.9e-4	0.85	R_e=32
	16	1.8e-2	1.8e-2	1.0	
	32	5.0e-3	1.2e-1	25	
	64	3.6e-4	1.0e-2	28	
12)	8	1.1e-7	9.8e-8	0.89	R_e=64
	16	3.4e-4	3.3e-4	0.97	
	32	1.8e-2	1.8e-2	1.0	
	64	5.1e-3	1.2e-1	23.5	

Table V: Artificial viscosity with FAS in both relaxations and coarse grid correction

Results for some of the sample problems from the implementation using artificial viscosity to ensure that the linear system representing the problem is diagonally dominant. Centered differences are used in both relaxations and in the computation of τ for the coarse grid correction.

solution are vectors. The example problems upon which this version was tested are presented in Figure 8. Problem sample (14) used R_e = 1, sample (15) R_e = 16, and sample (16) R_e = 32.

The only version of this program which has been implemented is one using artificial viscosity when a grid point yields a system of equations which does not meet the diagonal dominance criterion. At each grid point the diagonal dominance test used was

$$\frac{a_{ii}}{h^2} \geqslant \frac{d_{ii}}{2h}$$

$$u_{1xx} + u_{1yy} = 0 \qquad (13)$$

$$u_{2xx} + u_{2yy} - u_{1x} - 2u_2 = 0$$

Solution: $u_1 = 4\cos(x)\sinh(y)$

$u_2 = 2x\sin(x)\sinh(y)$

$$u_{1xx} + u_{1yy} - 2u_1 = 0 \qquad (14\text{-}16)$$

$$\frac{u_{2xx}}{R_e} + \frac{u_{2yy}}{R_e} - u_{2x} + u_{2y} = 0$$

Solution: $u_1 = e^{(x+y)}$

$u_2 = (1-e^{xR_e})(1-e^{-yR_e})$

Figure 8: Sample problems (13) to (16)

The sample problems used to test the implementations which could handle systems of linear equations. Examples (14) to (16) used R_e values of 1, 16, and 32.

$$\frac{c_{ii}}{h^2} \geqslant \frac{e_{ii}}{2h}$$

for each i.

Table VI presents the errors and convergences. Examples (13) and (14) require no artificial viscosity and results are excellent. Examples (15) and (16) give results very similar to the single equation examples discussed above. The error is not improved on the grids requiring artificial viscosity, but the stability of the grids is maintained until sufficiently fine grids are reached.

example	n	cgerr[n]	err[n/2]	ratio	
13)	8	1.6e-5	4.7e-4	29.4	
	16	1.1e-6	1.6e-5	14.5	
	32	6.9e-8	1.1e-6	15.9	
	16	4.4e-9	6.9e-8	15.7	
14)	8	1.6e-5	4.9e-4	30.6	R_e=1
	16	1.1e-6	1.7e-5	15.5	
	32	7.4e-8	1.1e-6	14.9	
	64	4.8e-9	7.4e-8	15.7	
15)	8	1.8e-2	1.5e-2	0.83	R_e=16
	16	5.0e-3	1.2e-1	24	
	32	3.6e-4	1.0e-2	28	
	64	4.5e-5	1.0e-3	22	
16)	8	3.4e-4	4.9e-4	1.4	R_e=32
	16	1.8e-2	1.8e-2	1.0	
	32	5.1e-3	1.2e-1	23.5	
	64	3.6e-4	1.0e-2	28	

Table VI: Artificial viscosity and FAS for systems of equations

Results are presented for some sample problems from the implementation which will handle systems of equations, and uses artificial viscosity. This implementation is not adaptive and uses the inefficient storage method discussed in the text.

3.4. Linear systems, with adaptive meshes

One of the conclusions from the results so far obtained is that grids finer than those used above will be necessary for many problems. For this it is necessary that adaptive grids be implemented as part of an adaptive program which will check error estimates on a grid point basis, rather than on a whole grid basis, and make the grid finer only in the regions which require this. The latest version of this multigrid package has the ability to adaptively utilize finer meshes only in regions where it is necessary.

The same set of problems was treated as in the previous section, with a tolerance sufficiently low that full grids were used at all levels. The results are tabulated in Table VII. Problem (13) is an example of a problem which has a non-diagonal term in one of the coefficient matrices. This example is smooth and this is reflected in the excellent convergence result.

Results for problem (14) are approximately as expected, for (15) and (16) there seems to be a "catch up" grid at the second grid which does not use artificial viscosity. As long as the problem requires

example	n	cgerr[n]	err[n/2]	ratio	
13)	8	9.1e-6	4.7e-4	51.6	
	16	8.1e-7	1.6e-5	19.8	
	32	5.4e-8	1.1e-6	20.4	
	64	3.5e-9	6.8e-8	19.4	
14)	8	1.7e-5	4.9e-4	28.8	R_e=1
	16	1.1e-6	1.0e-5	9.1	
	32	5.9e-8	9.6e-7	16.3	
	64	3.8e-9	5.9e-8	15.5	
15)	8	1.8e-2	1.5e-2	0.83	R_e=16
	16	4.7e-3	1.2e-1	25.5	
	32	1.0e-4	1.0e-2	100	
	64	4.5e-5	9.9e-4	22.0	
16)	8	3.4e-4	4.9e-4	1.4	R_e=32
	16	1.8e-2	1.8e-2	1.0	
	32	4.7e-3	1.2e-1	25.5	
	64	1.1e-4	1.0e-2	90.9	

Table VII: The use of the adaptive grid implementation with a very low tolerance

Results from the adaptive implementation, but with the tolerance set low enough that no adaptive grid formations were used.

artificial viscosity, we cannot really expect an improvement in the maximum error. What happens is that we get closer and closer to the singularity and keep the error there reasonably small.

When the same problems are attacked with a low tolerance limit, with the idea of sacrificing high accuracy to achieve better computation time, the accuracy results for example (14) are as in Table VIII. Runs for the other examples terminated at grid sizes of 8 or 16 because their error estimates were sufficiently small. As can be seen, once we reach the tolerance level, we see no improvement in the actual error. As a matter of fact, there is some backward movement in

---

Problem (15) : R_e=16

n	cgerr[n]	err[n/2]	ratio	internal points	not solved	proportional grid size
8	1.8e-2	1.5e-2	1.2	49	43	43/49 = 0.88
16	4.8e-3	1.2e-1	25.0	279	130	130/256 = 0.51
32	1.1e-4	1.0e-2	90.9	789	368	368/1024 = 0.36
64	6.6e-3	9.9e-4	0.16	2158	1169	1169/4096 = 0.29
128	1.8e-2	1.8e-2	1.0	6402	4887	4887/16384 = 0.30

Table VIII: Actual use of adaptive grids(high error tolerance)

Results from the adaptive implementation, using a tolerance low enough that incomplete grids were used in some cases. Presented here are the grid size n, the maximum error found on the coarse grid at level n, the maximum error found on the grid at level $n/2$, which corresponds to the coarse grid at level n, the ratio of these two errors, the total number of grid points which exist in this grid, the number of these grid points which are accurate enough that they will not be interpolated to the finer grid, and the proportional grid size relative to a full grid.

---

the error obtained. This may be due to some leverage of error from points considered to be solved satisfactorily. This behaviour is the sacrifice made for the computational savings. The last three columns show the number of grid points existing in the grids at the various levels, the number of grid points being relaxed, and the ratio of the latter to the maximum number of points in the grid at that point. This last value is approximately the proportional work being done at that grid level relative to the non-adaptive method. Since each successive grid is four times the size of the previous, the relative values on the finest grid are the most important.

In Figures 9 to 12 we give grid diagrams representing the grid points at the four levels which were actually used in the adaptive solution of example (14). The asterisks represent the grid points where a solution was found to an estimated accuracy better than the tolerance and the "@" characters represent those which were not. These latter points are interpolated to provide the finer grid points for the next level. As can be seen, only about a quarter of the $n = 64$ grid needed interpolation, which has allowed the program to proceed to even finer grids for the more difficult areas of the domain.

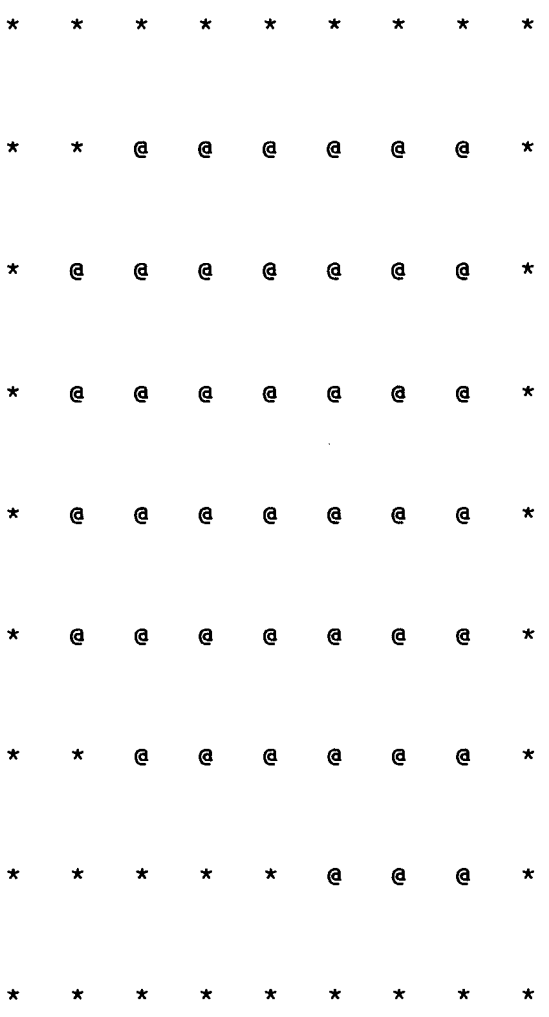

Figure 9: Level 8 grid, example (15)

The grid at level $n=8$ used on example (15). This is a full grid, and the error estimates on the asterisk points are less than the tolerance, while those on the "@" points are greater than the tolerance.

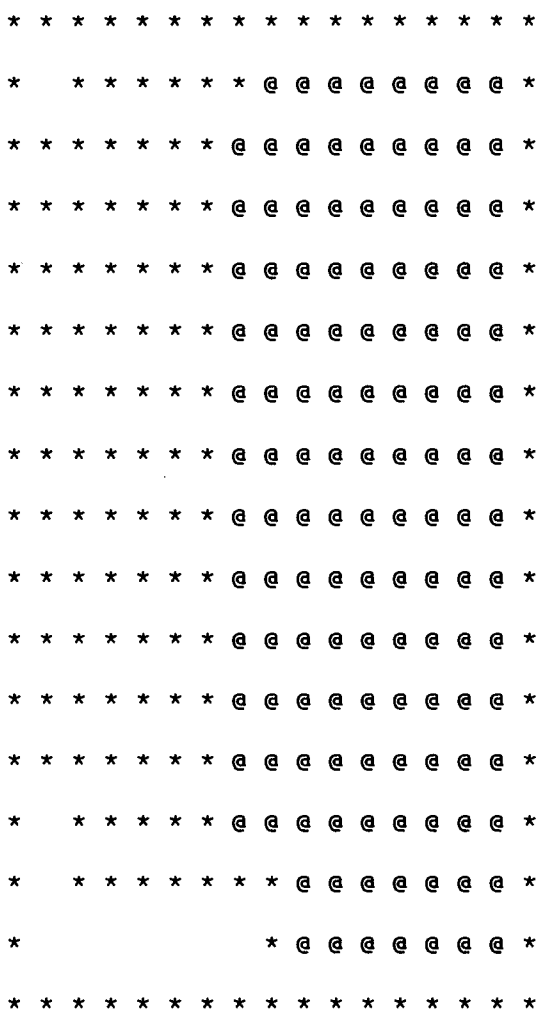

Figure 10: Level 16 grid, example (15)

The grid at level n=16 used on example (15). This is not a full grid, and the error estimates on the asterisk points are less than the tolerance, while those on the "@" points are greater than the tolerance. The blank areas of the grid have no grid points allocated.

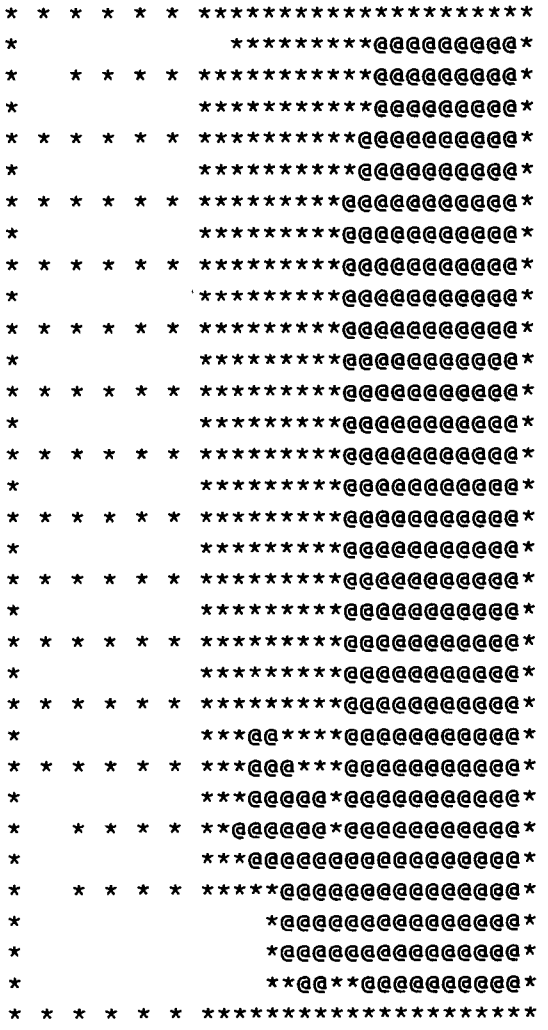

Figure 11: Level 32 grid, example (15)

The grid at level n=32 used on example (15). This is not a full grid, and the error estimates on the asterisk points are less than the tolerance, while those on the "@" points are greater than the tolerance. The blank areas of the grid have no grid points allocated.

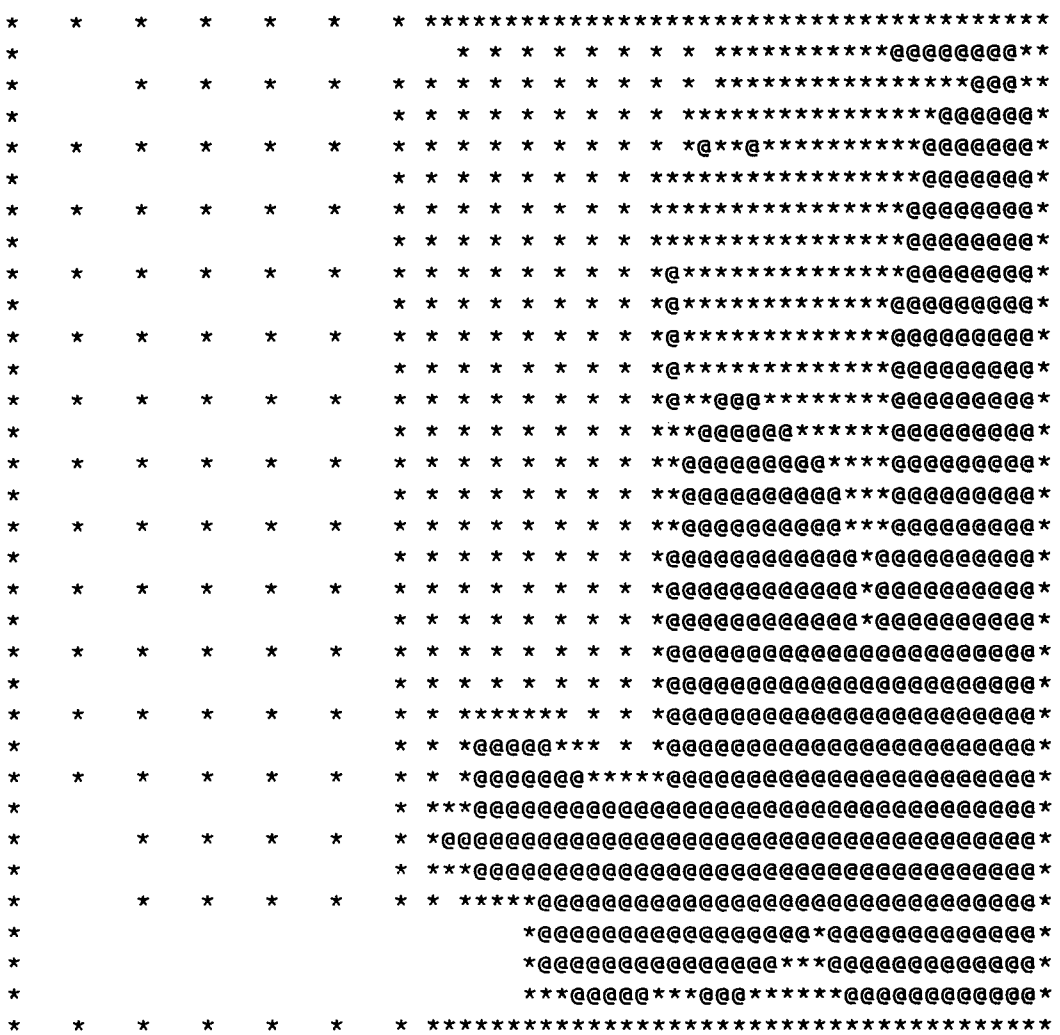

Figure 12: Level 64 grid, example (15)

The grid at level n=64 used on example (15). Only half of the rows in this grid are displayed here, due to space limitations. The error estimates on the asterisk points are less than the tolerance, while those on the "@" points are more than the tolerance. The blank areas of the grid have no grid points allocated. The width scale of the diagram has been doubled for this diagram, to accommodate the larger number of points displayed here relative to the previous diagrams.

4. Discussion

4.1. What has been done

Several implementations of the multigrid method for the solution of systems of partial differential equations have been produced. The family of problems to which the latest of these implementations may be applied is written as

$$a\ u_{xx} + c\ u_{yy} + d\ u_x + e\ u_y + f\ u = g$$

where a, c, d, e, and f are matrices, g is a vector, and all are functions of x and y. Some of these implementations use a data structure which is very wasteful of storage, but is simple to utilize. The latest version uses a list based, space efficient data structure which allows adaptive grid interpolations.

A variety of combinations of relaxations and coarse grid corrections were tried. Some of these were:

i) 2nd order differencing, FAS coarse grid correction (CGC)

ii) 1st order differencing when necessary, FAS CGC

iii) 1st order differencing when necessary, residual CGC

iv) 2nd order differencing, with artificial viscosity

The last three options were used for mildly stiff problems. We can define such a problem as one which, with the usual centered differencing, would yield a difference system which does not meet the diagonal dominance criterion on one or more of the grids, but which does meet the criterion on at least the finest grid.

It was found that, though it is advantageous to use the FAS method (it allows τ extrapolation and easy adaptation to nonlinear problems), difficulties arise when 1st order differencing is needed

on the correction grids, *viz.* the extra error is propagated from the coarse grids up to the finest grid, ruining the convergence. If residual correction is used instead of FAS, the error on the coarse grids does improve according to theory, and it does not propagate and ruin the fine grid convergence. The error which propagates in the FAS case is 1st order with respect to u, while the propagating error in the residual correction case is 1st order with respect to the error in u. Fourth order accuracy is attainable with residual correction, but adaptation to nonlinear problems will not be straightforward.

The use of artificial viscosity on the coarse grids was found to work well, but the extrapolation cannot be expected to work at the grid points using the viscosity, so the approximation and error improvement on those gridpoints is poor. Whether this will become a significant difficulty when entire grids need viscosity has yet to be examined. One certainly must worry about the "leverage" of error, where the error estimate on a given grid point may be sufficiently low and may even be a correct estimate, but may cause a disproportionately large error in nearby grid points which are still being processed. In this case the algorithm may proceed nicely but yield an invalid result.

4.2. Take Home Message

The indications from this study are that the use of artificial viscosity is a practical method for ensuring convergence in the solution of mildly stiff boundary value problems by the multigrid method. It is easier to program than the one sided differences approach and when used in conjunction with the FAS correction scheme, is more

robust with respect to the propagation of error from coarse to fine grid. This possibility of using the FAS algorithm is a requirement for extending the implementation to nonlinear systems.

For linear systems, the use of residual coarse grid corrections seems to be preferable.

4.3. Omissions from this work

This project can be extended in a number of ways which, using hindsight, are beyond the scope of this thesis:

(1) Tune up the adaptive strategy used by the package. If we limited the proportion of grid points which could be interpolated at each level, the total work to be done would be more predictable than the present system of using a simple tolerance on the error estimate. Also, the strategy should be more flexible, so the system could change its mind about points that it once considered to be solved.

(2) Implement a version using 1st order differencing in combination with the FAS algorithm, or demonstrate that this combination cannot work. There should be some way of preventing the 1st order coarse grids from interfering with the finer 2nd order grids. It may even be that I have not examined the distribution of error over the breadth of the grid in sufficient detail.

(3) If (1) is impractical, implement a residual correction version for use on linear problems. For some linear problems a one-sided algorithm may be advantageous, and it is always preferable to have the choice.

(4) Implement a version for nonlinear problems. This would possibly use the FAS correction scheme, and Newton iteration at each grid point, as did the first implementation discussed in this thesis.

(5) Implement some of the relaxation techniques suggested in the literature, such as red-black ordering Gauss Seidel (RBGS), line Gauss Seidel(LGS), and incomplete LU decomposition (ILU). One might include in this category experiments in types of restrictions, prolongations, interpolations, and coarse grid corrections, all components of the multigrid method.

(6) Try some realistic problems, as opposed to ones I have constructed from known solutions. Some examples might be the Cauchy Riemann problem, the Steady State Stokes system, and the Steady State Incompressible Navier Stokes system.

(7) Try 9 point averaging restriction for problems with oscillatory behaviour. This could be considered to be a subset of the aspect of trying out difficult problems to gauge the power of this implementation. This type of restriction was implemented in between the first two versions, but no conclusive results were obtained, due to the nature of the test problems.

(8) Extend the algorithm to handle arbitrary boundaries. This would involve setting up boundary grid points by binary search until the boundary was sufficiently close to a point, and then including these boundary points in the coarse grids.

5. References

1. E. J. Asselt, The multigrid method and artificial viscosity, in *Multigrid Methods Proceedings, Koln-Porz, November 1981*, vol. 960, W. Hackbush and U. Trottenberg (ed.), Springer-Verlag, Berlin, 1982.
2. W. Auzinger and H. J. Stetter, Defect Correction and multigrid iterations, in *Multigrid Methods Proceedings, Koln-Porz, November 1981*, vol. 960, W. Hackbush and U. Trottenberg (ed.), Springer-Verlag, Berlin, 1982.
3. A. Brandt, Multi-level adaptive technique (MLAT) for fast numerical solution to boundary value problems. , in *Proceedings Third International Conference on Numerical Methods in Fluid Mechanics, Paris, 1972*, vol. 18, H. Cabannes and R. Teman (ed.), Springer-Verlag, Berlin, 1973, 82-89.
4. A. Brandt, Multi-level adaptive techniques (MLAT) I. The Multigrid Method, Research Report RC 6026, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1976.
5. A. Brandt, Multi-level adaptive solutions to partial differential equations - Ideas and Software, in *Proceedings of Symposium on Mathematical Software*, J. Rice (ed.), Academic Press, New York, March 1977, 277-318.
6. A. Brandt, Multi-level adaptive solutions to boundary value problems, *J. Math. Comp. 31*, (1977), 333-390.
7. A. Brandt, Multi-level Adaptive Techniques (MLAT) for Singular Perturbation Problems, in *Numerical Analysis of Singular Perturbation Problems*, P. W. Hemker and J. J. H. Miller (ed.), Academic Press, London, 1979.
8. A. Brandt, Stages in Developing Multigrid Solutions, in *Numerical Methods for Engineering*, E. Absi, R. Glowinski and P. Lascaux (ed.), Dunod, Paris, 1980, 23-44.
9. A. Brandt and S. Ta'asan, Multi-grid methods for highly oscillatory problems, Research Report, Dept. of Applied Mathematics, Weizmann Institute of Science, Rehovot, 1981.
10. A. Brandt, A Guide to Multigrid Development, in *Multigrid Methods Proceedings, Koln-Porz, November 1981*, vol. 960, W. Hackbush and U. Trottenberg (ed.), Springer-Verlag, Berlin, 1982.
11. J. E. Dendy, Black Box Multigrid, *J. Comp. Phys. 48*, 3 (1982), 366-386.
12. J. E. Dendy, Black Box Multigrid for NonSymmetric Problems, *J. Appl. Math. and Comp. 13*, (1983), 261.
13. H. Foerster and K. Witsch, Multigrid Software for the Solution of Elliptic Problems on Rectangular Domains, in *Multigrid Methods Proceedings, Koln-Porz, November 1981*, vol. 960, W. Hackbush and U. Trottenberg (ed.), Springer-Verlag, Berlin, 1982.
14. W. Hackbusch, On Multigrid Iterations with Defect Correction, in *Multigrid Methods Proceedings, Koln-Porz, November 1981*, vol. 960, W. Hackbush and U. Trottenberg (ed.), Springer-Verlag, Berlin, 1982.
15. P. W. Hemker, On the Structure of an Adaptive Multi-level Algorithm. BIT, , 1980.

16. P. W. Hemker, *The Incomplete LU Decomposition as a Relaxation Method in Multigrid Algorithms Boundary and Interior Layers - Computational and Asymptotic Methods*, Boole Press, Dublin, 1980.

17. R. Kettler, Analysis and comparison of relaxation schemes in robust multigrid and preconditioned conjugate gradient methods., in *Multigrid Methods Proceedings, Koln-Porz, November 1981*, vol. 960, W. Hackbush and U. Trottenberg (ed.), Springer-Verlag, Berlin, 1982.

18. W. J. A. Mol, Smoothing and Coaarse Grid Approximation Properties of Multigrid Methods, NW 110/81, Department of Numerical Mathematics, Stichting Mathematisch Centrum, Amsterdam, 1981.

19. W. J. A. Mol, On the Choice of Suitable Operators and Parameters in Multigrid Methods, NW 107/81, Department of Numerical Mathematics, Stichting Mathematisch Centrum, Amsterdam, 1981.

20. R. A. Nicolaides, On the observed rate of convergence of an iterative method applied to a model elliptic defference equation, *Math. Comp. 32*, (1978), 127-133..

21. K. Stuben and U. Trottenberg, Multigrid Methods: Fundamental Algorithms, Model Problem Analysis, and Applications, in *Multigrid Methods Proceedings, Koln-Porz, November 1981*, vol. 960, W. Hackbush and U. Trottenberg (ed.), Springer-Verlag, Berlin, 1982.

22. P. Wesseling, A convergence proof for a multiple grid method, NA-21, Delft Technical University, Delft, The Netherlands., 1978.

23. P. Wesseling, Theoretical and practical aspects of a multigrid method, NA-37, Department of Mathematics, Delft University of Technology, Delft, 1980.

24. P. Wesseling, A Robust and Efficient Multigrid Method, in *Multigrid Methods Proceedings, Koln-Porz, November 1981*, vol. 960, W. Hackbush and U. Trottenberg (ed.), Springer-Verlag, Berlin, 1982.

6. APPENDICES

Appendix A: User Instructions

The *mgrid* program is used to numerically solve systems of linear elliptic partial differential equations, as:

$$a\ u_{xx} + c\ u_{yy} + d\ u_x + e\ u_y + f\ u = g$$

where a, c, d, e, f are matrices, g is a vector, and all are functions of x and y, The user must provide these functions as pascal functions and link them together with the rest of the package(See appendix II for sample source). The makefile and several function examples in the package source code can act as examples.

A sample run command on the unix system used for development is :

```
mgrida title < prmfile > resultsfile
```

The title is printed on the output, and is optional. The standard input contains the execution parameters, one per line, in the following order:

nmin	minimum grid size used (a multiple of 2)
nmax	maximum grid size used (a larger multiple of 2)
alpha	α, the ratio expected of tau from grid to grid (usually 0.25)
delta	δ, the ratio expected of error from grid to grid (usually use 0.125)
eta	slow convergence criterion η, usually use 0.5, (0 < eta < 1)
tol	tolerance on error estimate, an exit criterion.
tautol	tolerance on the change in τ, an exit criterion within the solve routine.
cyclelimit	another exit criterion within the solve routine, if τ ratio does not reach alpha within this number of cycles, solve stops trying, accepting the value of τ so far obtained.
extraptau	the τ extrapolation factor, usually set to 1.33
lbx	the lower bound of the x component of the domain
ubx	the upper bound of the x component of the domain
lby	the lower bound of the y component of the domain
uby	the upper bound of the y component of the domain

In the parameter file, comments may be placed on the line after the value. Unlike this example, additional comment lines may not be inserted.

Standard output contains the level by level summaries of the process of the computation. The maximum values of τ and error estimates are printed each time they are computed at the fine grid level.

At the conclusion of each grid level computation in the nested set, a detailed output file is created with the name "grid.nnn", where the "nnn" is the grid level. This file contains a record for each grid point in the grid, with its x, y, and u values, as well as the error estimates and the actual errors. This implementation computes the actual error using a supplied function which computes

the actual solution, a feature which would probably not be carried over into a production model, since in real life an analytical solution is rarely known.

Appendix B contains sample source for the required functions, appendix C a sample parameter file, and appendix 4 sample output for that particular run.

Appendix B: Sample Source Code

Below is a listing of an example of the source which must be supplied by the user. Of course the problem must fit into the family of problems to which this package may be applied, linear second order systems of equations. The routines include a boundary value function(fbc), a solution function(usol), and six functions defining the problem, *a*, *c*, *d*, *e*, *f*, and *g*.

```
#include "defs.h"
#include "ext.h"

procedure fbc; { ( x,y:real; var v : vector);}
const
  Re = 16;
begin
    v[1] := (exp(-Re)-exp(Re*(x-1)))*(1-exp(-Re*y));
    v[2] := exp(x+y);
end;

procedure usol; { ( x,y:real; var v : vector);}
const
  Re = 16;
begin
    v[1] := (exp(-Re)-exp(Re*(x-1)))*(1-exp(-Re*y));
    v[2] := exp(x+y);
end;

procedure a; {(x,y:real; var v:matrix);}
const
  Re = 16;
begin
    v[1][1] := 1/Re;
    v[1][2] := 0;
    v[2][1] := 0;
    v[2][2] := 1;
end;

procedure c; {(x,y:real; var v:matrix);}

const
  Re = 16;
begin
    v[1][1] := 1/Re;
    v[1][2] := 0;
    v[2][1] := 0;
    v[2][2] := 1;
end;
```

```
procedure d; {(x,y:real; var v:matrix);}

begin
    v[1][1] := -1;
    v[1][2] := 0;
    v[2][1] := 0;
    v[2][2] := 0;
end;

procedure e; {(x,y:real; var v:matrix);}

begin
    v[1][1] := 1;
    v[1][2] := 0;
    v[2][1] := 0;
    v[2][2] := 0;
end;

procedure f; {(x,y:real; var v:matrix);}

begin
    v[1][1] := 0;
    v[1][2] := 0;
    v[2][1] := 0;
    v[2][2] := -2;
end;

procedure g; {(x,y:real; var v:vector);}

begin
    v[1] := 0;
    v[2] := 0;
end;
```

Appendix C: A Sample Parameter File

Below is a sample of a parameter file. This parameter file and the source in the previous appendix were used to produce the results in the next appendix.

```
4 nmin
32 nmax
0.25 alpha
0.125 delta
0.5 eta
1.0e-8 tol
0.01 tolerance on the change in tau value
3 cyclelimit
0.1 lowerbndx
0.9 upperbndx
0.1 lowerbndy
0.9 upperbndy
```

Appendix D: A Sample Output File

Here is the descriptive output for the run specified by the previous 2 appendices:

```
the parameters were: delta=  1.250e-01, alpha=  2.500e-01, Eta=  5.000e-01
error and tau tolerances:  1.000e-04 1.000e-02, limit on # cycles     3
nmin=     4, nmax=    32
The upper and lower bounds of x:  0.000e+00 1.000e+00
The upper and lower bounds of y:  0.000e+00 1.000e+00


printerr: n=     4, max error  1.535e-02, at x=  7.500e-01, and y=  2.500e-01
printerr: n=     4, max cg error  4.478e-04, at x= 5.000e-01, and y= 5.000e-01

solve: n= 8 maximum tau value[1]3.2e-02, at x= 7.5e-01 y= 7.5e-01

solve: n= 8 maximum tau value[2]3.4e-02, at x= 7.5e-01 y= 7.5e-01
solve: relaxation count(top level total) 7

solve: n= 8 maximum tau value[1]3.2e-02, at x= 7.5e-01 y= 7.5e-01

solve: n= 8 maximum tau value[2]3.4e-02, at x= 7.5e-01 y= 7.5e-01
solve: relaxation count(top level total) 9

solve: n= 8 maximum tau value[1]3.2e-02, at x= 7.5e-01 y= 7.5e-01

solve: n= 8 maximum tau value[2]3.4e-02, at x= 7.5e-01 y= 7.5e-01
solve: relaxation count(top level total) 11
solve: tau tolerance attained
printerrest: for grid n=     4, max error correction is  3.959e-03
printerrest: for grid n=     2, max error correction is  3.959e-03

printerr: n=     8, max error  1.150e-01, at x=  8.750e-01, and y=  2.500e-01
printerr: n=     8, max cg error  1.772e-02, at x= 7.500e-01, and y= 2.500e-01


solve: n= 16 maximum tau value[1]8.7e-01, at x= 8.8e-01 y= 3.8e-01

solve: n= 16 maximum tau value[2]1.8e-02, at x= 8.8e-01 y= 8.8e-01
solve: relaxation count(top level total) 2

solve: n= 16 maximum tau value[1]7.7e-01, at x= 8.8e-01 y= 5.0e-01

solve: n= 16 maximum tau value[2]1.1e-02, at x= 8.8e-01 y= 8.8e-01
solve: relaxation count(top level total) 5
```

```
solve: n= 16 maximum tau value[1]7.4e-01, at x= 8.8e-01 y= 8.8e-01

solve: n= 16 maximum tau value[2]1.1e-02, at x= 8.8e-01 y= 8.8e-01
solve: relaxation count(top level total) 7
solve: reached limit of number of cycles
printerrest: for grid n=     8, max error correction is  1.025e-01
printerrest: for grid n=     4, max error correction is  9.661e-02

printerr: n=    16, max error  1.007e-02, at x=  9.375e-01, and y=  8.125e-01
printerr: n=    16, max cg error  4.789e-03, at x= 8.750e-01, and y= 5.000e-01


solve: n= 32 maximum tau value[1]4.1e-01, at x= 9.4e-01 y= 6.3e-01

solve: n= 32 maximum tau value[2]1.2e+00, at x= 5.0e-01 y= 8.8e-01
solve: relaxation count(top level total) 4

solve: n= 32 maximum tau value[1]4.0e-01, at x= 9.4e-01 y= 8.8e-01

solve: n= 32 maximum tau value[2]1.1e+00, at x= 5.0e-01 y= 8.8e-01
solve: relaxation count(top level total) 6

solve: n= 32 maximum tau value[1]4.0e-01, at x= 9.4e-01 y= 6.3e-01

solve: n= 32 maximum tau value[2]1.2e+00, at x= 5.0e-01 y= 8.8e-01
solve: relaxation count(top level total) 8
solve: reached limit of number of cycles
printerrest: for grid n=    16, max error correction is  8.108e-03
printerrest: for grid n=     8, max error correction is  8.069e-03

printerr: n=    32, max error  9.936e-04, at x=  9.688e-01, and y=  9.063e-01
printerr: n=    32, max cg error  5.366e-04, at x= 5.000e-01, and y= 8.750e-01


mgrid: execution finished, using finest grid    32
```

Appendix E: Source for the central routines solve, relax, taufunc

If the user wishes to apply a new differential operator, or use a different coarse grid correction algorithm, new versions of the relaxation routine and the coarse grid correction routine would have to be written. Possibly some changes would be required in the solve routine as well, in the parts concerned with the coarse grid correction routine. Here is the source for those three routines. preceded by the definitions file used by all routines.

```
label 999;

const
   TINY = 1.0e-30;
   BIG = 1.0e10;
   GRIDSIZE = 16384;     { max grid division : 2^14 : keep i,j to 2 bytes }
   VECTORSIZE = 2;       { should be the only change for more equations }
   FIELDSPERPAGE = 8;    { just used for formatting output }
   EXTRAPTAU = 1.3333333333333333;  { extrapolation factor}
   UNKNOWN = 1.0E35;    { use this to test for unknown tau value}
type
  positive = 1..maxint;
  nonnegative = 0..maxint;
  gridrange = 0..GRIDSIZE;
  vector = array[1..VECTORSIZE] of real;

  ivector = array[1..VECTORSIZE] of integer;

  matrix = array[1..VECTORSIZE] of vector;

  atvlist = ^vlist;
  vlist = record
      u : vector;
      link: atvlist
      end;

  atgrid = ^gridtyp;
  gridtyp = record       { grid is made of these nodes}
      i,j : gridrange;
      u,                 { the result vector}
      grhs,              { right hand side for this gridpoint}
        tau,             { tau value at last CGC, valid on coarse
                           grid, Value from the CCG (coarse coarse
                           grid, are to right of the CCG grid points }
      errest : vector;   { error estimate, could be flag or scalar
                           if we wanted to save memory?}
      smrate,                 { smoothing rate, see if further relaxing
                           is needed. could be a flag?}
```

```
      rlxchange : real; { last change which produced u in the
                          relaxations, also used as a flag
                          if negative, this point has not been
                          interpolated and should not be used for
                          further interpolation }
        tauflag : boolean;     { is tau sufficient at this point?}
      xlink,ylink : atgrid     { pointers to right(x) and up(y) directions}
      end;


  longstring = array[1..100] of char;

var    { the external declarations, signified by capital first letter
         as opposed to constants which are all caps. }
  Nmin,Nmax : 1..GRIDSIZE;
  Cyclelimit:nonnegative;
  Alpha,Delta,Eta,Tol,Tautol,Lbndx,Ubndx,Lbndy,Ubndy,Hx,Hy : real;
  Zero : vector;  { used for zeroing vectors}
  Maingrid : atgrid;          { points to gridpoint at x=Lbx,y=Lby}
  Title : longstring;


{ Routine which does the main work of the multigrid method for
solving a discretized form of a boundary value ordinary
differential equation }

#include "defs.h"
#include "ext.h"

procedure solve;
{ (n:gridrange; gr:atgrid; corrgridflag:boolean);
}
var
  x,y,change,conv,t :real;
  int2,int4,l,count,cyclecount : nonnegative;
  maxi,maxj : ivector;
  subgr,nextx,nexty,subnextx,subnexty,below,above,right,left : atgrid;
  ulist,savedgrid : atvlist;
  first,tflag,returnflag,tauflag : boolean;
  maxtau,prevtau : vector;
```

```
begin   { 0 }
if n=Nmin then solved(gr)     { solve directly and to high accuracy }
else
  begin   { 1 }
  copytau(n,gr);   { copy tau expectation on CCG to adjacent gridpoints
                     and initialize grid flags }
  for l:= 1 to VECTORSIZE do prevtau[l] := BIG;
  cyclecount := 1; first:=true;
  change:=BIG;              { signal for first sweep }
  int2 := 2*(GRIDSIZE div n); int4:=2*int2;
  count:=0;
  returnflag := false;
  while not returnflag do
    begin    { 2 }
    change:=BIG;     { signal for first sweep }
    conv:=0;

    while (conv < Eta) do
      begin   { 3 }
      relax (gr, n, change, conv); count:=count+1;
      end;   { 3 }


    taufunc(n,gr,first);{ get tau for each grid point }
    first:=false;

    restrict (n,gr,subgr);    { including the boundaries; create subgr}

    for l := 1 to VECTORSIZE do maxtau[l] := 0;
    tauflag := true;
    nexty:=gr^.ylink; subnexty:=subgr^.ylink; below:=gr;
    while subnexty^.ylink <> nil do    { boundary values not used later }
      begin   { 3 }
      while nexty^.i <> subnexty^.i do
        begin below:=nexty;nexty:=nexty^.ylink;   { 4 }
        end;   { 4 }
      above:=nexty^.ylink;

      if above^.i - below^.i = int2 then
        begin   { 4 }
        left:=nexty; nextx := nexty^.xlink; subnextx := subnexty^.xlink;
        while subnextx^.xlink <> nil do
          begin   { 5 }
          while nextx^.j <> subnextx^.j do
            begin left:=nextx; nextx:=nextx^.xlink;   { 6 }
            end;   { 6 }
          while below^.j < nextx^.j do below:=below^.xlink;
          if below^.j = nextx^.j then
            begin   { 6 }
            above:=nextx^.ylink; right:=nextx^.xlink;
```

```
        if ( above^.i - below^.i = int2) and
        ( right^.j - left^.j = int2) then
          begin     { we are at a gridpoint in a fine grid region}  { 7 }
          if tauflag then
            if (nextx^.j mod int4 = 0) and (nextx^.i mod int4 = 0) then
              for l := 1 to VECTORSIZE do
              if (abs(nextx^.tau[l]) > abs(nextx^.xlink^.tau[l])) then
               tauflag:=false;
          if corrgridflag then
            for l := 1 to VECTORSIZE do
              subnextx^.grhs[l] := nextx^.tau[l] + nextx^.grhs[l]
          else
            for l := 1 to VECTORSIZE do
             subnextx^.grhs[l] := EXTRAPTAU*nextx^.tau[l] + nextx^.grhs[l];
        for l:=1 to VECTORSIZE do
          begin { 8 }
            t := abs(nextx^.tau[l]);
            if t > maxtau[l] then
              begin  { 9 }
              maxtau[l] := t;
              maxi[l] := nextx^.i;
              maxj[l] := nextx^.j;
              end;  { 9 }
          end; { 8 }
          end;   { 7 }
        end;   { 6 }
      subnextx := subnextx^.xlink;
      end;   { 5 }
    end;   { 4 }
  subnexty:=subnexty^.ylink;
  end;   { 3 }

savegrid (subgr,savedgrid);            { save uH for ops after solving }
{ necessary if restriction is not injection }
{ merely copies grid u values into a list to be read back in again.
the grid structure will not change before it is used again}
if not corrgridflag then
for l := 1 to VECTORSIZE do
  begin   { 3 }
  x := Lbndx + maxj[l] * Hx;
  y := Lbndy + maxi[l] * Hy;
  writeln;
  writeln('solve: n= ',n:1,' maximum tau value[',l:0,']',
  maxtau[l]:4,', at x= ',x:4, ' y= ',y:4);
  end;   { 3 }

solve(n div 2,subgr,true);
```

```
nexty:=subgr; ulist:=savedgrid;
while nexty<>nil do
  begin             { put the correction into the grid}   { 3 }
  nextx:=nexty;     { instead of the residual solution }
  while nextx<>nil do
    begin   { 4 }
    for l := 1 to VECTORSIZE do
    nextx^.u[l]:=nextx^.u[l] - ulist^.u[l];
    ulist:=ulist^.link;
    nextx:=nextx^.xlink;
    end;   { 4 }
  nexty:=nexty^.ylink;
  end;   { 3 }

ulist:=savedgrid;   { get rid of the save storage}
while ulist<>nil do
  begin   { 3 }
  savedgrid:=savedgrid^.link; dispose(ulist);  ulist:=savedgrid;
  end;   { 3 }

prolongrid(gr, subgr); { prolongate the grid to level n }

{ gr is used to ensure identical structure of both grids
for efficiency could avoid expanding subgr by doing the correction at the
same step, i.e. compute the fine grid corrections as one goes. This would
involve combining prolongrid and the following correction code}

nexty:=gr^.ylink; subnexty:=subgr^.ylink;
while nexty^.ylink<>nil do  { uH = 0 at boundaries, so don't add them }
  begin   { 3 }
  nextx:=nexty^.xlink; subnextx:=subnexty^.xlink;
  while nextx^.xlink<>nil do
    begin   { 4 }
  if nextx^.rlxchange >= 0 then { signal that this point }
                          { is being relaxed}
      for l := 1 to VECTORSIZE do
        begin    { 5 }
        nextx^.u[l] := nextx^.u[l] + subnextx^.u[l];
        if not corrgridflag then
        nextx^.errest[l] :=
          nextx^.errest[l] + abs(subnextx^.u[l])/(1+abs(nextx^.u[l]));
                    { relative error for u >> 1 }
      end;   { 5 }
    subnextx:=subnextx^.xlink; nextx:=nextx^.xlink;
    end;   { 4 }
  subnexty:=subnexty^.ylink; nexty:=nexty^.ylink;
  end;   { 3 }

disposegrid(subgr);
```

```
    if not corrgridflag then
      writeln('solve: relaxation count(top level total) ', count:1);

    if tauflag then           { successfully attained tau goal }
      begin   { 3 }
      if not corrgridflag then writeln('solve: tau goal attained');
      returnflag := true;
      end   { 3 }
    else
      begin  { 3 }
      tflag:=true;
      for l:= 1 to VECTORSIZE do
      if abs(1-abs(maxtau[l]/prevtau[l])) > Tautol then tflag:=false;
      if tflag then
        begin  { 4 }
        if not corrgridflag then writeln('solve: tau tolerance attained');
        returnflag := true;
        end  { 4 }
      else
      if cyclecount = Cyclelimit then
        begin   { 4 }
        if not corrgridflag then
          writeln('solve: reached limit of number of cycles');
        returnflag:=true;
        end   { 4 }
      else
      cyclecount := cyclecount+1;
      prevtau:=maxtau;
      end;   { 3 }
    end;   { 2 }
  end;   { 1 }
end;
```

```
#include "defs.h"
#include "ext.h"

{
This is a routine to calculate the difference between one element
of the vector resulting from the application of the difference operator
on grid level n/2 and grid level n, on the vector u.
n is the fine grid level.
This function calculates the difference between the grid operator
vector product at a given grid point, between level n and level n/2.

             h h h    H h h
    So tau := - I L u  + L I u
             H          H

The operator is the discretized operator corresponding to
the perturbed laplace equation

      a*u  + c*u + d*u + e*u  + f*u = g
           xx     yy    x     y

The operator does not use g, and in the difference the coefficient of
u, i.e. the f function, cancel, so they are not used here.

This routine presumes that the restriction operator is injection, so
it just uses the coarse grid values present, not computing them.

In this routine only fine grid regions are processed, i.e., coarse grid
points which are surrounded by fine grid points.
}
procedure taufunc;
{ (n : gridrange; gr:atgrid; first:boolean);
}
var
  nexty,cbelow,below,above,cabove,left,cleft,right,cright,here : atgrid;
  l,int,int2 : integer;
  amatrix,camatrix,cmatrix,ccmatrix,dmatrix,ematrix : matrix;
  tvect,tvect1,tvect2,txx,txxc,tyy,tyyc,tx,txc,ty,tyc : vector;
  ta,tc,td,te,t : real;
  x,y,hx,hy,hhx,hhy : real;
begin { 0 }
  int := GRIDSIZE div n;
  int2 := 2*int;
  hx := int*Hx; hhx:=hx*hx;
  hy := int*Hy; hhy:=hy*hy;
  { need 5 in a fine grid row on the left boundary}
  nexty:=gr;
  cbelow:=nexty; below:=cbelow^.ylink; cleft:=below^.ylink;
  above:=cleft^.ylink; cabove:=above^.ylink;
```

```
while cabove<>nil do
begin { 1 }
  if ( above^.i - below^.i = int2 ) then   { fine grid region}
  begin left:=cleft^.xlink; here:=left^.xlink; right:=here^.xlink; { 2 }
    cright:=right^.xlink;   { x line is set up}
    while cright<>nil do
    begin { 3 }
      if ( right^.j - left^.j = int2 ) then          { 5 in a row}
      begin { 4 }
        while cbelow^.j < here^.j do cbelow:=cbelow^.xlink;
        if (cbelow^.j = here^.j) and (cbelow^.ylink <> here) then
        begin above:=here^.ylink; below:=cbelow^.ylink; { 5 }
          if (above^.i - below^.i = int2 ) then
          begin cabove:=above^.ylink; { 6 }
            y := here^.i*Hy+Lbndy;
            x := here^.j*Hx+Lbndx;

            a(x,y,amatrix);      { for matrix computations procedures }
            c(x,y,cmatrix);      { used below have prefixes: }
            d(x,y,dmatrix);      { mt signifies matrix operation, }
            e(x,y,ematrix);      { vt signifies vector operation }

          mtcopy(amatrix,camatrix);
          mtcopy(cmatrix,ccmatrix);
            for l := 1 to VECTORSIZE do
            begin { 7 }
              ta := amatrix[l][l]; tc := cmatrix[l][l];
              td := dmatrix[l][l]; te := ematrix[l][l];
              t := abs(td*hx/2);
              if ta < 2*t then
            camatrix[l][l] := 2*t;
              if ta < t then
            amatrix[l][l] := t;
              t := abs(te*hy/2);
              if tc < 2*t then
            ccmatrix[l][l] := 2*t;
              if tc < t then
            cmatrix[l][l]:=t;

              txx[l] := (left^.u[l] - 2*here^.u[l] + right^.u[l])/hhx;
              txxc[l] := (cleft^.u[l] - 2*here^.u[l] + cright^.u[l])/(4*hhx);
              tyy[l] := (below^.u[l] - 2*here^.u[l] + above^.u[l])/hhy;
              tyyc[l] := (cbelow^.u[l] - 2*here^.u[l] + cabove^.u[l])/(4*hhy);

              tx[l] := (right^.u[l] - left^.u[l])/(2*hx);
              txc[l] := (cright^.u[l] - cleft^.u[l])/(4*hx);
              ty[l] := (above^.u[l] - below^.u[l])/(2*hy);
              tyc[l] := (cabove^.u[l] - cbelow^.u[l])/(4*hy);
              end;  { 7 }
```

```
            if first then right^.tau := here^.tau;
          mtvtmult(camatrix,txxc,tvect1);
          mtvtmult(amatrix,txx,tvect2);
          vtsub(tvect1,tvect2,here^.tau);
          mtvtmult(ccmatrix,tyyc,tvect1);
          mtvtmult(cmatrix,tyy,tvect2);
          vtsub(tvect1,tvect2,tvect);
          vtadd(tvect,here^.tau,here^.tau);
          vtsub(txc,tx,tvect);
          mtvtmult(dmatrix,tvect,tvect1);
          vtadd(tvect1,here^.tau,here^.tau);
          vtsub(tyc,ty,tvect);
          mtvtmult(ematrix,tvect,tvect1);
          vtadd(tvect1,here^.tau,here^.tau);

          end; { 6 }
        end; { 5 }
      end; { 4 }
    if (left^.j-cleft^.j = int) then
    begin
      cleft:=here; left:=right; here:=cright; right:=cright^.xlink;
      if right <> nil then cright:=right^.xlink
    else cright:=nil;
      end
    else
    begin
      cleft:=left; left:=here; here:=right; right:=cright;
      cright:=cright^.xlink;
    end;
    end; { 3 }
  end; { 2 }
  if (nexty^.ylink^.i - nexty^.i = int) then
    begin
    cbelow := nexty^.ylink^.ylink; nexty:=cbelow;
    below:=cbelow^.ylink; cleft:=below^.ylink; above:=cleft^.ylink;
    if above<> nil then cabove:=above^.ylink
    else cabove:=nil;
    end
  else
    begin
    cbelow := nexty^.ylink; nexty:=cbelow;
    below:=cbelow^.ylink; cleft:=below^.ylink; above:=cleft^.ylink;
    cabove:=above^.ylink;
    end;
 end; { 1 }
end; { 0 }
```

```
#include "defs.h"
#include "ext.h"

{ routine which does one Gauss-Seidel relaxation sweep,
  actually solving the necessary system of n equations for
  the gridpoint for the n simultaneous pdes.

  designed to be used in solving the general problem

       au  + cu + d*u  + e*u  + f*u = g
         xx    yy    x      y

where a,c,d,e,f, and g are functions of (x,y),
}
procedure relax;
{(gr:atgrid; n:gridrange; var change,conv : real);
}
var
  nextx,nexty,right,left,above,below : atgrid;
  l : 1..VECTORSIZE;
  int,int2 : gridrange;
  tmp,newchg,thischange,hx,hy,hhx,hhy,x,y,tta,ttc,ttd,tte,h2x,h2y : real;
  ta,tc,td,te,amatrix,cmatrix,dmatrix,ematrix,fmatrix : matrix;
  beta0,beta1,beta2,beta3,beta4 : matrix;
  rhs,uprev,w1,w2,w3,w4 : vector;

begin
  newchg:=0; int:=GRIDSIZE div n; hx:=int*Hx; hy:=int*Hy;
  hhx:=hx*hx; hhy:=hy*hy; h2x:=2*hx; h2y:=2*hy; int2:=2*int;
  nexty:=gr^.ylink; below:=gr;
  while nexty^.ylink<>nil do
  begin
    nextx:=nexty^.xlink; left:=nexty;
    while nextx^.xlink<>nil do
    begin
      right:=nextx^.xlink; above:=nextx^.ylink;
      below:=below^.xlink;
      if below^.ylink <> nextx then below:=belowlink(gr,nextx);
      if (above^.i - below^.i = int2) and ( right^.j - left^.j = int2) then
      begin
      vtcopy (nextx^.u,uprev);
      x:=nextx^.j*Hx+Lbndx; y:=nextx^.i*Hy+Lbndy;
      a(x,y,amatrix);
      c(x,y,cmatrix);
      d(x,y,dmatrix);
      e(x,y,ematrix);
      f(x,y,fmatrix);
```

```
    { put in the artificial viscosity}
    for l := 1 to VECTORSIZE do
      begin
      tta := amatrix[l][l]/hhx; ttd := dmatrix[l][l]/h2x;
      ttc := cmatrix[l][l]/hhy; tte := ematrix[l][l]/h2y;
      if tta < abs(ttd) then amatrix[l][l] := hhx*abs(ttd);
      if ttc < abs(tte) then cmatrix[l][l] := hhy*abs(tte);
      end;

    scmtmult(-2/hhx,amatrix,ta);
    scmtmult(-2/hhy,cmatrix,tc);
    mtadd(ta,tc,beta0);
    mtadd(beta0,fmatrix,beta0);
          { beta0 is the matrix for our gridpoint }
    scmtmult(0.5/hx,dmatrix,td);
    scmtmult(1/hhx,amatrix,ta);
    mtadd(ta,td,beta1);
    mtsub(ta,td,beta2);
    scmtmult(1/hhy,cmatrix,tc);
    scmtmult(0.5/hy,ematrix,te);
    mtadd(tc,te,beta3);
    mtsub(tc,te,beta4);
          { beta[1-4] are for the rhs of this grid point}
    mtvtmult(beta1,right^.u,w1);
    mtvtmult(beta2,left^.u,w2);
    mtvtmult(beta3,above^.u,w3);
    mtvtmult(beta4,below^.u,w4);

    vtsub(nextx^.grhs,w1,rhs);
    vtsub(rhs,w2,rhs);
    vtsub(rhs,w3,rhs);
    vtsub(rhs,w4,rhs);

{ call the gaussian elimination routine to solve the system }

    gauss ( beta0,rhs, nextx^.u );

    thischange := 0;
    for l := 1 to VECTORSIZE do
      begin
      tmp := abs(nextx^.u[l] - uprev[l])/(1+abs(nextx^.u[l]));
      if tmp > thischange then thischange:=tmp;
      end;
```

```
      if nextx^.rlxchange > 0 then
        nextx^.smrate:=thischange/nextx^.rlxchange;
      if nextx^.rlxchange >= 0 then  { don't muck up the boundary flag}
        nextx^.rlxchange:=thischange;
      if thischange > newchg then newchg:=thischange;
      end;
      left:=nextx; nextx:=nextx^.xlink;
    end;
    below:=nexty; nexty:=nexty^.ylink;
  end;

  if newchg=0 then
    begin
    conv:=1;
    writeln('relax: newchg = 0');
    end
  else
    if (change > 1) or (newchg < 1 ) then conv:=newchg/change
  else
    if (change/newchg = 0 ) then
      begin conv:=1;
      writeln('relax: conv would have overflowed');
      end
  else
    conv:=newchg/change;
  change := newchg;
end;
```