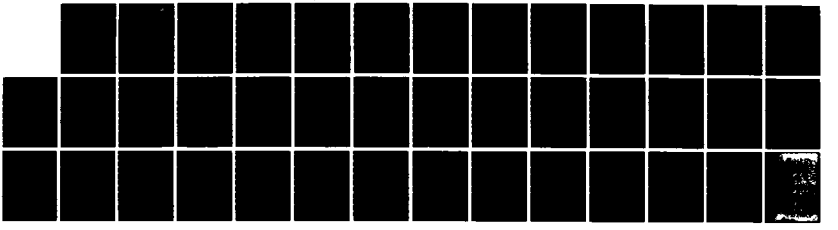
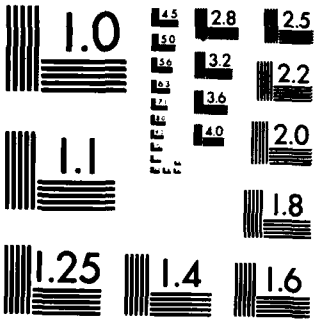
AD-A147 290 GRIDPACK: TOWARD UNIFICATION OF GENERAL GRID PROGRAMMING(U) WEIZMANN INST OF SCIENCE REHOVOTH (ISRAEL) DEPT OF APPLIED MATHEMATICS A BRANDT ET AL. 1/1

UNCLASSIFIED JUN 83 DAJA37-79-C-0504 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

GRIDPACK: TOWARD UNIFICATION OF GENERAL GRID PROGRAMMING

Final Technical Report

by

Achi Brandt and Dan Ophir

June 1983

US ARMY RESEARCH AND STANDARDIZATION GROUP (EUROPE)

London, England

Contract No. DAJA37-79-C-0504
Department of Applied Mathematics
Weizmann Institute of Science
Rehovot, Israel

Approved for public release; distribution unlimited

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. AD A147290
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) Toward Unification of General Grid Programming	5. TYPE OF REPORT & PERIOD COVERED Final Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Achi Brandt and Dan Ophir	8. CONTRACT OR GRANT NUMBER(s) DAJA37-79-C-0504
9. PERFORMING ORGANIZATION NAME AND ADDRESS Weizmann Institute of Science Dept. of Mathematics, Rehovot, Israel	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 1T161102BH57-05
11. CONTROLLING OFFICE NAME AND ADDRESS USARDSG-UK Box 65, FPO NY 09510	12. REPORT DATE June 1983
	13. NUMBER OF PAGES Thirty Seven (37)
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved For Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

GRIDPACK; numerical solution of PDE; grid generation; grid programming; boundary programming; finite difference equations; Multigrid; fast solvers; interpolation; display

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

GRIDPACK is a fully portable Fortran extension, including economic data structures, routines and macro-statements, for simplifying and unifying the programming of grid operations in general two-dimensional domains. These operations include the introduction and modification of grids, boundaries, inner and outer subgrids, grid functions, boundary functions and finite-difference operators, including operators near and on boundaries; interpolation between grids; sweeping conveniently and with full efficiency over grids;

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CONTENTS

1. INTRODUCTION

The main purpose of GRIDPACK is to supply the general Fortran programmer with convenient and fully efficient facilities for handling two-dimensional grids and for communicating grid routines to other programmers. The user of this software can, within his usual Fortran code and by means of simple macro-statements (or simple calls to GRIDPACK routines), perform a variety of grid operations, such as:

- Create the logical structure (set of pointers) for a grid by specifying its domain (or boundary curves), meshesizes and ordering.
- Create similar structures for boundary grids (the intersections of any given boundary curves with given gridlines). Such structures can in fact be used for treating any other curves cutting through the grid, such as material interfaces and traced discontinuities (shocks, wakes, etc).
- Create new grids from exisiting ones by various operations such as unions, intersection, transposition (changing row ordering into column ordering), coarsening and refinement; or create a new grid which is the "inner" part of a given grid (containing all gridpoints whose all neighbors with respect to a given templet are also gridpoints), or its "outer" part (the grid without its inner part).
- Define templets (neighborhoods of gridpoints).
- Allocate storage for grid functions, including boundary-grid functions. The resulting storage is fully efficient and flexible: the amount of pointers is small compared to the amount of real numberical data, but still allow grid changes costing small CPU times compared to the CPU times of numerical processes over the grids.

- Delete any of the above grids or storage allocations. Garbage-collection routines ensure deletion of holes in the data structure as soon as their total size becomes significant.
- Introduce numerical values, specified as Fortran functions, into the grid functions.
- Add, subtract or do any other arithmetic with grid functions.
- Interpolate from a grid function (or a boundary grid function) on a coarser grid to one on a finer grid, with any desired order of interpolation.
- Transfer information between the interior grid and the corresponding boundary grid.
- Sweep conveniently over grids, or parts of grids, or simultaneously over several grids, by various means, including various DO-like statements or calls to KEY routines which set up convenient-to-use arrays (see Fig. 1). The resulting sweeps are fully efficient, with single-indexed DO loops performing the sweeps over the inner strings. Inside the DO-like statements, grid functions can be programmed as if they were rectangular arrays and efficient automatic branching is available for separate treatment at irregular points (gridpoints some of whose neighbors, by a given templet, are not in the grid).
- Program, once for all grids and all programs, various tasks, such as various difference operators near boundaries involving both boundary and interior data, relaxation of general types of interior finite difference equations, relaxation of boundary conditions (independently or together with the interior relaxation), etc.
- Communication between curved grids (used locally, along boundaries or interfaces or discontinuities, in a multi-grid framework) and Cartesian grids. This feature is still under development.

- Display grids, grid functions, boundary shapes, templets, etc., showing also their relative geometric positions.

More important perhaps than any one of these specific operations is the fact that the GRIDPACK software offers a general framework for communicating grid programs. Any user of this software can program once for all various grid operations, that can then be reused for any other grid in his program, or in any other program using this software. Moreover, GRIDPACK routines can even be used by programs which do not employ neither GRIDPACK nor its data structure, provided they incorporate some simple interface routines (the KEY routines; see Fig. 1, and Sec. 4.3). Enormous programming repetition can thus be saved, and high degree of portability gained.

To obtain full portability, GRIDPACK is entirely written in Fortran and as a Fortran extension. That is, the basic part of GRIDPACK is a collection of Fortran routines which can serve any Fortran program. In addition, a special collection of macro-statements, called Grid Language (GL), has been developed, which can much simplify various programming tasks. These macro-statements can be used within a usual Fortran code: They are expanded into conventional Fortran statements, including calls to GRIDPACK routines, by a special preprocessor yclept PREPACK. The preprocessor itself is written in Fortran, thus maintaining the full portability of the system.

The development of GRIDPACK is of particular importance for the evolving technology of *multi-grid programming*. Multigrid techniques (or, more generally, Multi-Level Adaptive Techniques - MLAT) are already used quite extensively as fast (in fact, the fastest) solvers for discretized partial differential equations of general types on general

domains. They can also be used for many other purposes, such as the flexible creation of locally refined discretization and local coordinate transformations, all integrated into the fast-solvers. (See, for example [8] for a brief discussion of multi-level capabilities, potential and research, and [7] for a more complete introduction and survey of recent developments. The geometrical capabilities of local refinement and coordinate transformation are described in [6, Sec. 9], as well as in previous publications [2], [3], [4].) All multigrid algorithms, and in particular those with highly developed geometrical capabilities, are based on standard grid operations, of the types mentioned above, which are used time and again, for different grids in the same program and in many different multigrid programs. Thus, the GRIDPACK system may very much facilitate programming in this field, encourage the use of its full potential, and promote portability, communication and cooperation.

Some multigrid algorithms are indeed provided together with GRIDPACK (see Sec. 4.7). A collection of other multigrid programs comes on the same magnetic tape, called MUGTAPE (multi-grid tape) on which GRIDPACK is distributed. The detailed description of GRIDPACK [1] appears as another file on that tape. The tape also includes MUGPACK, a precursor of GRIDPACK that can be used both for learning the programming techniques and as a special-purpose program for the multigrid solution of the 5-point Poisson equation on a general domain. Still much simpler programs, for rectangular domains, are also included on MUGTAPE. Some of them are very short and very suitable for first acquaintance with multigrid techniques.

A new version of MUGTAPE, including the completely revised version of GRIDPACK described in this article, will hopefully be released by the end of 1983. It will be available from the Department of Appled Mathematics, Weizman Institute of Science, Rehovot, Israel.

The first description of GRIDPACK appeared as Section 4 in [3] and a complete description of its first phase was given in [5]. In addition to the present authors, several routines were contributed by Fred Gustavson and Alan Goodman (see Sec. 4.1), some boundary-treatment routines are modified ELLPACK routines [9], and some interpolation routines were contributed by Markus von Cube (see Sec. 4.3). The sponsors of this project, in addition to the Weizmann Institute of Science, are the United States Army, through its European Research Office under contract DAJA37-79-C-0504, and the Gesellschaft für Mathematik und Datenverarbeitung (GMD) in St. Augustin, FRG, through its Institut für Mathematics (IMA), where one of the authors (Ophir) worked for two years. The interest, assistance and encouragement of the entire GMD-IMA group is especially acknowledged.

2. DEFINITIONS

The following terms are used throughout GRIDPACK:

The *lattice* with *origin* (x_0,y_0) and *meshsizes* (h_x,h_y) in the (x,y) plane is the set:

$$L(x_0,y_0;h_x,h_y) = \{(x_i,y_j) \mid x_i=x_0+ih_x,\ y_j=y_0+jh_y;\ i,j \text{ integers}\}.$$

A *uniform grid*, $G = G(D;x_0,y_0;h_x,h_y)$ is the intersection of the domain D and the lattice $L(x_0,y_0;h_x,h_y)$. A *uniform subgrid* G', of grid G, is a uniform grid defined on a subdomain D' of the domain D, with the same lattice. They are formally two different grids, but they may share functions (see routine POINTO in Sec. 4.2), in which case G is called the *parent* of G'.

The *column* j of grid G defined above is the set of gridpoints;

$$\{(x,y) \mid y=y_0+jh;\ (x,y) \text{ is in } G\}.$$

The *row* i of grid G is the set of gridpoints:

$$\{(x,y) \mid x=x_0+ih,\ (x,y) \text{ is in } G\}.$$

A *grid-line* is a grid column or a grid row. A grid-line is composed of strings. A *point string* is a sequence of consecutive gridpoints in a grid column (*vertical string*) of in a grid row (*horizontal string*). The internal GRIDPACK representation of any uniform grid is organized in terms of either vertical strings or horizontal ones. In the first case the grid is said to be *vertical* or *in column construction order*, in the latter it is *horizontal* or *in row construction order*.

A *column string* is a sequence of consecutive columns. A *row string* is a sequence of consecutive rows. For brevity, the term *set* is used in GRIDPACK for column string (in case of vertical grids), or for row string (in horizontal grids). See Fig. 2. A *single-string grid* is a uniform grid with only one string per line and only one set. A *multi-string grid* is any uniform grid which is not single-string.

The following terms are used by GRIDPACK in treating boundaries of two-dimensional domains.

A *continuous piece* P in the (x,y) plane is any set of the form:

$$P = P(X,Y,t^B,t^E) = \{(x,y) \mid x = X(t),\ y = Y(t),\ t^B<t<t^E\}.$$

The piece *end-points* are the points

$$(X(t^B),Y(t^B)) \quad \text{and} \quad (X(t^E),\ Y(t^E)).$$

A *continuous curve* C is a union $U_{i=1}^{n} P_i$ of consecutive continuous pieces P_i; i.e.,

$$P_i = P(X_i,Y_i,t_i^B,\ t_i^E),\ X_i(t_i^B) = X_{i-1}(t_{i-1}^E)$$

$$Y_i(t_i^B) = Y_{i-1}(t_{i-1}^E),\quad (i = 1,\ldots,n)\ .$$

The curve is called *closed* if

$$X_n(t_n^E) = X_1(t_1^B) \quad \text{and} \quad Y_n(t_n^E) = Y_1(t_1^B),$$

otherwise the curve is called *open*.

A *boundary* B is a union $U_{i=1}^{m} C_i$ of continuous curves C_i. Sometimes there may be several boundaries in a program. Each boundary is therefore given a number, the *boundary number*, which should be distinguished from the grid number of the corresponding boundary-grid. A boundary is called *open* if at least one of its curves is open. The *boundary end-points* are the end-points of the boundary pieces.

A *boundary-grid* $BG = BG(B;x_0,y_0;h_x,h_y)$ is the intersection of the boundary B with the lines in the lattice $L(x_0,y_0;h_x,h_y)$, together with the set of boundary end-points. All these points are called *boundary-grid-points*. A *discrete piece*, or briefly a *piece*, is the sequence of boundary-grid-points on a given continuous piece, ordered by an ascending order of

the piece parameter (t). A *discrete curve*, or briefly a *curve*, is the sequence of discrete pieces corresponding to the continuous pieces of a given continuous curve.

A *grid* is either a uniform grid (or uniform subgrid) or a boundary-grid. In any given program any number of such grids may be defined; each is given a unique number called the *grid (ordinal) number*. We briefly say "grid n" instead of "the grid whose number is n". When no confusion may arise, the general term "grid" is sometimes used for a uniform grid.

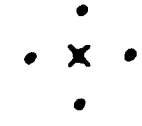
A *boundary point ordinal number* is the ordinal number of boundary grid-points when they are all ordered in their natural sequence; i.e., in ascending order of curves, in ascending order of pieces within each curve, and in ascending t (including t^E and t^B) within each piece. In closed curves the first and last boundary points have differ ordinal numbers although they correspond to the same physical point.

A *grid-function* is a function defined on a grid, i.e., an ordered set of values with one to one correspondence to grid-points. Often it will represent some approximate solution of a discretized problem. There are two kinds of grid functions corresponding to the two kinds of grids: a *uniform-grid-function* and a *boundary-grid-function*. On any grid any number of grid functions may be defined. Each grid function is thus uniquely specified by giving its grid number together with its *grid-function ordinal number*, the latter specifying its ordinal number relative to other functions defined on the same grid. A grid function is illustrated in Fig. 2.

A *function-string* is the sequence of function values that correspond to a point-string, which is a vertical string in case of a vertical uniform

grid, a horizontal string in case of a horizontal uniform grid, and a discrete piece in case of a perimetric boundary grid. (See Sec. 3.2. For function strings of boundary grids in other construction orders, see Sec. 3.3.) If several functions are defined on the grid, all function values corresponding to the same gridpoint are taken consecutively, before all those corresponding to the next gridpoint in the string. We thus view all the functions defined on the grid as a vector of functions, with a vector of numerical values corresponding to each gridpoint. All these functions should be of the same type: either real or integer. (The type can be changed, though. It is determined at run time by using the QSPACE or LSPACE routines. See Sec.4.2.) If both integer and real functions are desired on the same grid, it should be treated formally as two different grids. (Similarly, an interior grid and its boundary intersections are formally viewed as two different grids. Super-structures may later be defined for briefly describing operations involving different but related grids.)

A *templet* is a set of shifts on a grid, used to describe the neighborhood of a grid-point. For example, the neighborhood



(the standard 5-point neighborhood of the point x, used for example in approximating the Laplace or Poisson equation at x) is represented by the templet ((0,0),(0,1),(0,-1),(-1,0),(1,0)). Each templet has a unique *templet (ordinal) number* (τ).

The τ-*inner subgrid* of a given grid G is the grid composed of all the points belonging to G whose neighbors according to the given templet

number τ are also all in G. The τ-*outer subgrid* of a given grid G is a grid composed of all the points of G without the points of the τ-inner subgrid.

3. GRID INTERNAL DATA STRUCTURES

All the grids are organized using linked allocation in two general vectors ("garbage collectors"): The *L space* for all logical and integer data (mainly pointers), and the *Q space* for all type-real data (mainly numerical values of grid functions).

All the information related to grid n branches from a fixed list of integer parameters starting at L(L(n)). This list (fully given in [1, App. II]) specifies the number of functions defined on the grid and their type (real or integer), the grid's construction order (vertical or horizontal), its type (a uniform or a boundary grid), its parent grid m (i.e., the grid whose storage is used by grid n - see POINTO in Sec. 4.2; if regular storage was allocated by QSPACE(n, NF), then m = n; if no storage was allocated, then m = 0), the address in Q of its list of real parameters (meshsizes and origin), various basic grid statistics (such as its total number of sets, columns and strings, or curves and pieces, total number of grid points, total L and Q spaces occupied by the grid and its functions, and its smallest and largest row and column numbers), and the addresses of its chief data structures: For a uniform grid the address is given of the first set-quad (see Sec. 3.1). For a boundary grid, three addresses are given, corresponding to the three construction orders in which each boundary grid is constantly kept: The perimetric order (Sec. 3.2) and the vertical and horizontal orders (Sec. 3.3).

3.1 Uniform-grid: The QUAD structure

A uniform grid is organized either vertically (in columns) or horizontally (in rows), depending on the order in which it is to be used. (See Sec. 4.3, including the remark about transposing from vertical to horizontal.) For definiteness, the *vertical* construction is here described.

Each vertical grid is described as a sequence of column strings (sets) with each column being described as a sequence of *vertical* strings (see Fig. 2). Each sequence, either of strings or of sets, is represented internally by a chain of *quads* (integer quadruples), where each string (or set) is represented by a quad of the form

QUAD = | LOC | INDEX | LENGTH | NEXT | .

LOC is the location where the string is stored (in the L space if it is a set or a type-integer function string; in the Q space if it is a type-real function string). INDEX and LENGTH describe the lattice position of the string. INDEX is its first index, i.e., the column number of its first column if this is a column string, or the row number of its first grid-point if this is a function string. LENGTH is the cardinal number of columns or grid-points in the string. NEXT is the location (in L) of the next quad in the chain; NEXT = 0 if the present quad is the last in its chain.

Thus, every grid is represented by a chain of *set-quads*, corresponding to its sequence of sets, and every column is represented by a chain of *string-quads*, corresponding to its sequence of vertical strings. Each set-quad points to a set (a sequence of consecutive columns), i.e., its INDEX and LENGTH pointers indicate which and how many columns are included in the set, and its LOC pointer show the address in L where the representation of that set starts. Namely, at L(LOC) a sequence of LENGTH string-quads starts, corresponding to the LENGTH columns of the set; each string-quad in that sequence is in fact the first in the chain of string-quads representing the corresponding column. Each string-quad points to a function string; i.e., its INDEX and LENGTH pointers indicate on which consecutive grid-points the function string

is defined, while its LOC pointer shows the location in Q (or in L, if the function is type-integer) where the function string is stored.

This "QUAD structure" is relatively *inexpensive*, since a group of four integers represents a whole string of function values. Moreover, it is very *efficient for grid sweeping operations*. If the sweep is made in the construction lines (e.g., in columns when the construction is vertical), then it can be made in terms of efficient singly indexed DO loops (see Fig. 1). To sweep that efficiently in the other direction, too, one can transpose the grid. The QUAD structure may be somewhat inefficient in separately accessing a single grid-point of a multi-string grid, but this is an unlikely event. All the usual multigrid processes, for example, are sweep processes; an isolated access to a point is seldom needed.

Moreover, the QUAD structure is very *suitable for grid changes*, like those required by adaptive-grid procedures. Every string can reside anywhere in the L or Q space, and its length can therefore easily be changed by reconstructing it in a new location if necessary. After many changes of this type, many gaps (unused locations) may appear in the Q and L spaces. The system keeps track of the total length of these gaps. When it accumulates to a serious proportion, the gaps are automatically eliminated by one pass of "Garbage Collection" (described in [1, App. II]). This pass is very inexpensive relative to the numerical processes that would create those gaps (e.g. processes of grid adaptation, which normally follow at least a couple of relaxation sweeps over the adapted grids).

3.2 Boundary grid: Perimetric order

To obtain full efficiency and flexibility in applications, each boundary grid information is constantly kept in three different data structures. The

first and most basic structure organizes the boundary grid as a sequence of curves, each curve being organized as a sequence of pieces for which function strings are ordered in boundary ordinal numbers.

Thus, this "perimetric" construction order starts with a sequence of lists, corresponding to the sequence of curves of the grid. For each curve the corresponding list specifies its closure type (open, closed clockwise, or closed counter-clockwise), the ordinal number of its first piece, how many pieces it includes, and the addresses of two sequences of lists, one in L and one in Q, each sequence corresponding to the sequence of pieces in the curve. For each piece, its corresponding Q-list gives the limits of its parameter (t^B and t^E in Sec. 2). Its corresponding L-list specifies the ordinal number of the pair of FORTRAN functions defining the piece (X(t) and Y(t) in Sec. 2), the ordinal and cardinal number of the points in the piece, and the address of the function string corresponding to the piece.

Special functions can be defined on this construction order (some of them are naturally produced together with the creation of the data structure) to store various geometrical information and relations to (e.g., addresses of function strings of) corresponding interior grids (see Sec. 3.5).

3.3 Boundary grid: Vertical and horizontal orders

The second data structure constantly kept for each boundary-grid is the vertical one, describing the grid as a sequence of sets, each set being a sequence of consecutive columns and each column a sequence of boundary intersections (the boundary intersections with the column, ordered in ascending ordinates). It employs a system of pointers similar to the structures above, leading to the addresses of function strings. Each function string here is a

sequence of (possibly vectorial) values corresponding to the sequence of a column's boundary intersections.

The third structure is the horizontal one, similar to the vertical, rows replacing columns.

Special functions can be defined on these construction orders to store various geometrical information and relations to the perimetric construction order (see Sec. 3.5).

3.4 Templets

The templet number τ is stored as a string of integers starting at $L(L(NT + \tau))$, where NT is fixed by the initialization routine INIT. The first integer in the string specifies how many shifts are included in τ, and the shifts themselves follow, represented by a pair of integers each.

3.5 Handling boundary operators and geometry

One of the heaviest tasks usually confronted in general-domain grid programs is that of storing and using the geometrical information related to the boundary, producing, for example, difference approximations at irregular interior points near the boundary, or approximating Neumann boundary conditions, etc. The data structures presented above, together with a library of routines, part of which has already been written, can alleviate this task, and systematize it so that much of the programming can be done once and for all. The main vehicle is the introduction of a collection of standard geometrical functions, some of which are boundary-grid functions, others are τ-outer-subgrid functions.

Some boundary-grid functions are produced during the construction of the boundary-grid, and can be useful again later. These include the following functions, shown as functions of the boundary-point ordinal number k:

BTYPE(k) - the boundary type of k (an intersection with a column, or with a row, or a piece end-point, or a combination of these types).

NPOINT(k) - the ordinal number (within the piece) of k.

NPIECE(k) - the ordinal number (within the curve) of the piece to which k belongs.

NCURVE(k) - the ordinal number (within the boundary grid) of the curve to which k belongs.

TCOOR(k) - the value of the piece parameter (t in Sec. 2) at k.

XCOOR(k) - the value of the x-coordinate at k.

YCOOR(k) - the value of the y-coordinate at k.

Generally, the user of GRIDPACK can cancel any function defined on any grid, freeing their Q (or L) storage for other uses. Or he can create new functions, changing if needed the number of functions defined on any grid (see Sec. 4.2). In particular, he may cancel any of the above functions when no longer needed, or rebuild them. Other geometrical functions of a similar nature may be added to the system, such as functions that give, at each boundary point k, the slopes dX/dt, dY/dt or dX/dY, or the curvature of the piece, higher-order derivatives, etc.

Also, a collection of type-integer and type-real functions can be built that form, for each grid-point k, a finite-difference formula approximating the normal derivative at k, with the type-integer functions supplying the addresses in Q of the function values participating in that formula. Similar finite-difference boundary capabilities can be added to the system. Once programmed, they can be used for any grid by any GRIDPACK user.

Another type of geometrical information can be given as functions of the uniform-grid points near the boundary, i.e., as functions defined on

τ-outer subgrids. These functions may include horizontal and vertical distances to the boundary, the addresses of function values defined on neighboring boundary intersections (see POINTE and POINTB in Sec. 4.3, for example), etc. These elementary functions can then be used by various GRIDPACK users to construct more advanced functions, such as formulas for various finite-difference approximations to various differential operators, with type-integer functions giving addresses of values participating in the formulae. The creation of such functions could again be programmed once and for all, making Cartesian-coordinate differencing near curved boundaries much more standard and easier.

4. GRIDPACK ROUTINES

In this section we briefly describe a selection of those GRIDPACK subroutines which are available to the usual user. Lists of routine parameters are sometimes abbreviated here for the sake of clarity. All parameters are allowed to be variables. We do not describe here error messages. For the full description of all GRIDPACK routines, including internal routines not normally accessible to the user, see [1, Appendix II].

4.1 Initialization and logical grid operations

Logical grid operations are routines (whose name therefore usually starts with an L) which deal with the logical structures (the pointers) of the grid, not with allocating or using storage for functions defined on the grid. Thus, the LOC pointers of the string quads remain undefined by these routines. The computer time expended by these routines (and also by those of Secs. 4.2 and 4.3) is negligible, because it is proportional to the number q of *strings* in the grid, whereas numerical processes (relaxation, etc.) expand many operations per *grid point*. Exceptions are routines LTPØSE, where the work is O(q log q), and LGRDCR, where the characteristic function should be evaluated at each lattice point in the rectangle $[x_1,x_2] \times [y_1,y_2]$.

As a general rule, all the routines below assume (and check) that when different grids participate in the same logical operation they have the same construction order and the same lattice (except in coarsening and refinement operations, of course). In grid operations creating new grids, if the new grid number coincides with a previously defined one, the old one is thereby deleted.

Several of the routines in this section and in Sec. 4.2 were designed in collaboration with F. Gustavson and written by F. Gustavson and A. Goodman at IBM T. J. Watson Research Center, Yorktown Heights, New York. Some of the internal routines used by LGRDBR were modified ELLPACK routines [9].

INIT - Initialization routine, should be called before any other GRIDPACK routine. It gets from the user several basic parameters, such as the maximum number of grids, maximum number of templets and constants affecting the frequency of garbage collection. The user can sometimes also make changes in a long list of GRIDPACK constants held in certain COMMONs. The INIT routine uses all these parameters and constants to set up various pointers into the L and Q spaces. (cf., e.g., Sec. 3.4)

LGRDCR $(n,x_0,y_0,h_x,h_y,I_c,x_1,x_2,y_1,y_2,$CHAR$)$ - Create uniform-grid $G(D;x_0,y_0;h_x,h_y)$ with ordinal grid number n, where the domain D is defined by the characteristic function CHAR(x,y):

$$D = \{(x,y) \mid x_1 \le x \le x_2,\ y_1 \le y \le y_2,\ \text{CHAR}(x,y) = 1\}.$$

I_c is the index of construction order; $I_c = 1$ if vertical, $I_c = 0$ if horizontal.

LGRD2F $(n,x_0,y_0,h_x,h_y,I_c,x_1,x_2,f_1,f_2)$ - Create uniform-grid number ℓ given the 2 functions $f_1(x)$ and $f_2(x)$. Similar to LGRDCR, except that

$$D = \{(x,y) \mid x_1 \le x \le x_2,\ f_1(x) \le y \le f_2(x)\}.$$

LGRDBR (n,h_x,h_y,x_0,y_0,I_c,NB,BCOORD,INITB) - Create boundary-grid n which is the intersection of the lines of $L(x_0,y_0;h_x,h_y)$ with the continuous boundary defined by the ordinal number NB and by the user's boundary-specification routines INITB and BCOORD, which are structured as similar ELLPACK routines.

LGRIDB(n,m,I_c,h_x,h_y) - Create the uniform-grid n bounded by the boundary-grid m, in construction order I_c. The meshsizes (h_x, h_y) of n should be integer multiples of those of m.

DELETE(n) - Delete logical grid number n.

LCOARS(m,n,I_x,I_y) - Create grid n which is the coarsening of grid m by factors I_x and I_y in the x and y directions, respectively.

LUNION(ℓ,m,n) - Create union of two uniform grids: $n = \ell \cup m$.

LSECT(ℓ,m,n) - Create the uniform-grids intersection $n = \ell \cap m$.

LMINUS(ℓ,m,n) - Create the uniform-grids difference $n = \ell - m$.

BTEMPL(τ,IT) - Build templet number τ by the information in array IT, simply by copying from IT to $L(L(NT+\tau))$ (see Sec. 3.4).

DELT(τ) - Deletes templet τ.

LINNER(m,n,τ) - Create n as the τ-inner subgrid of grid m.

LOUTER(m,n,τ) - Create n as the τ-outer subgrid of grid m.

LINRBT($n,m,\tau_x,\tau_y,I_c,h_x,h_y$) - Create internal uniform-grid n bounded by boundary-grid m with construction order I_c and meshsizes h_x and h_y, "internal" being defined in terms of the templets τ_x and τ_y: An internal point is a grid point inside the domain bounded by m, whose all τ_x-neighbors and all τ_y-neighbors are also in that domain,

and none of whose τ_x-neighborhood horizontal links and τ_y-neighborhood vertical links is intersected by the boundary grid.

LTPOSE(m,n) - Create n as the transpose of m, i.e., the same uniform grid but in the opposite construction order.

COMPARE(ℓ,m) - Compare the constants of grids ℓ and m (their origins, meshsizes, total number of points, strings and columns, etc.) and print them out, usually for debugging purposes.

ORIGIN(n,x_0,y_0) - Change the origin of grid n to (x_0, y_0).

Function NGFREE(n) is the smallest grid number greater than n which is free (not assigned to undeleted grid).

Function NTFREE(τ) is the smallest templet number above τ which is free.

4.2 Function storage allocation

The routines in this section specify, or change, the number of functions defined on a (uniform or boundary) grid, and allocate for them suitable Q or L storage, supplying accordingly the values for the LOC pointers in the string quads. Allocating storage in the Q space defines the functions, and hence the grid, to be type-real, while allocating in L defines them as type-integer.

QSPACE(n,NF) - Allocate Q space for a vector of NF grid functions on grid n.

LSPACE(n,NF) - Similar allocation in L space.

POINTO(n,m) - Set the pointers of uniform-grid n to point into the function strings of uniform-grid m. Usually grid n will represent a subgrid m, created for example by LINNER(m,n,τ) or LSECT(ℓ,m,n). This routine thus enables us to operate on a proper part of a grid. The

number of functions (NF) defined on n is automatically set to equal that of m.

QOFF(n) - The same as QSPACE(n,o).

4.3 KEY interface

Here is a collection of routines designed to interface between the user and the grid functions and other grid data. Starting from the grid number specified by the user, the KEY routines will trace various pointers, make some short calculations and then load convenient interface information into arrays named by the user. With this information the user can perform fully efficient DO loops over columns of vertical grids or over rows of hotizontal grids. If the grid construction order is not compatible with the interior loops, one can obtain full efficiency by first transposing the grid (see LTPOSE in Sec. 4.1 and TFERTP in Sec. 4.4). The transposition is very inexpensive (relative to the cost of a relaxation sweep, for example).

KEYS(n,I_1,I_2,IR,J_1,J_2,JR,...) - Load constants I_1, I_2, and IR, and vectors J_1, J_2 and JR so that if u is the only function defined on the *vertical* single-string grid n, its value at the (i,j) gridpoint will be given by

$$u_{ij} = u(x_0 + ih_x, \quad y_0 + jh_y) = Q(JR(IR+i) + j),$$

valid (i.e., (i,j) is indeed a gridpoint) for

$$I_1 \le i \le I_2, \qquad J_1(i) \le j \le J_2(i).$$

See an example for the use of this routine in Fig. 1 above. For a *horizontal* grid n, KEYS(n,J_1,J_2,JR,I_1,I_2,IR,...) would similarly

give $u_{ij} = Q(IR(JR+j)+i)$. The unshown parameters of KEYS above include the meshsizes of grid n, and sometimes additional information.

KEYG(n,...) - Similar to KEYS, but for a general (possibly multi-string) grid n. In this case, for a vertical grid,

$$u_{ij} = Q(JR(KR(IR(\ell)+i)+k)+j),$$

where k is the string number within the column and ℓ is the set number. IR, KR and JR, as well as other vectors giving the ranges of j, k, i and ℓ in the grid, are all specified by the user and loaded by KEYG.

KEYI(n,τ,...) - Similar to KEYG, but giving the ranges for the τ-inner subgrid of n.

KEYBG(n,...) - Give similar access to boundary grid n in perimetric order.

KEYP(n,NC,NP,...) - Give access to the NP-th piece in the NC-th curve of boundary grid n.

KEYBI(n,IC,...) - Give access to the boundary grid number n, in I_c construction order (vertical or horizontal).

POINTE(n,m) - Point the function on uniform-grid n to the east neighbor on boundary-grid m; i.e., for each gridpoint of n insert the *address* of the boundary function value corresponding to the nearest boundary point lying east to the gridpoint, at a distance less than the meshsize. If the distance is greater than the meshsize, the value ITOP (a standard default ingeter, usually the largest integer defined on the computer) is instead inserted. Usually grid n for

such a purpose will be a τ-outer subgrid, the templet τ containing (only) the east shift (0,1). In this example grid n should have a single type-integer function defined on it, by having called LSPACE(n,1).

POINTB(n,m,ν_E,ν_W,ν_N,ν_S) - Point the ν_E-th function on grid n to the east neighbor on boundary-grid m, the ν_W-th function to the west neighbor, ν_N-th to the north, and ν_S-th to the south.

4.4 Function initialization, operation and transfer

The subroutines in this section and in the next are part of a growing collection of routines for putting values into, and transferring values between, grid functions. Functions mentioned in Sec. 3.5 are also part of that collection.

In the description below we denote the ν-th function defined on n by $u^{n,\nu}$, and its value at gridpoint (i,j) by $u^{n,\nu}_{ij}$. u^n stands for the vector of functions (or the single function) defined on grid n.

PUTSC(n,ν,C) - Put the scalar constant C into $u^{n,\nu}$.

PUTSF(n,ν,F) - Put the scalar function F(x,y) into $u^{n,\nu}$; i.e.,

$$u^{n,\nu}_{ij} \leftarrow F(x_i, y_j) = F(x_o + ih_x, y_o + ih_y).$$

PUTVC(n,V) - Put the constant vector V into u^h. The length of V is assumed to equal the number of functions defined on n.

PUTC(n,C) - Same as PUTSC(n,1,C).

PUTZ(n) - Same as PUTC(n,0).

PUTF(n,F) - Same as PUTSF(n,1,F).

In the following transfers, where several grids participate, it is assumed that all grids have the same lattice, the same number of

functions, and the same construction order (except in TFERTP, of course). The transfer is always made at all lattice points (i,j) common to all participating grids.

TFER(m,n) - Transfer u_{ij}^m into u_{ij}^n.

SUBF(ℓ,m,n) - Put $u_{ij}^\ell - u_{ij}^m$ into u_{ij}^n.

ADDF(ℓ,m,n) - Put $u_{ij}^\ell + u_{ij}^m$ into u_{ij}^n.

PUTF2F(ℓ,m,n,F) - Put $F(u_{ij}^\ell, u_{ij}^m)$ into u_{ij}^n.

TFERTP(m,n) - Transfer u_{ij}^m into u_{ij}^n, uniform grids m and n having opposite (transposed) construction orders. By using this routine between two applications of a line relaxation routine, for example, alternating-direction line relaxation procedure is automatically obtained. The transfer costs only one addition per gridpoint, plus some operations per string.

4.5 Interpolation

The collection of interpolation routines turned out to be perhaps the most difficult to develop, mainly due to the attempted generality. There are many possible cases to be considered regarding the relative positions of the involved grids and boundaries. Situations may arise, especially with very coarse grids and near special boundary configurations, where the desired order of interpolation is difficult, or even impossible, to achieve, since not enough points are available to interpolate from. Non-standard (e.g., non-central) interpolation formulas are then tried in various ways, searching for a rule as close to central as possible which still gives the highest possible interpolation order (up to the designated order). Therefore, uniform-grid interpolation routines have

so far been developed only for the standard coarse-to-fine case, i.e., from a coarse grid m to a finer grid n where $(h_x^m : h_x^n, h_y^m : h_y^n)$ is either (2,2), (2,1), or (1,2), and where every coarse-grid line coincides with a fine-grid line. The routines are devised to have full speed at interior parts of the two grids, with fully efficient DO loops. Near the boundaries, however, the routines are still engaged in very inefficient searches. Detailed account and timing of the interpolation algorithms is given in [1, App. II].

Routines INT2, INT2AD and INT4 were contributed by Markus von Cube at GMD, St. Augustin, Germany.

INTAD(n,m,ν,μ,I) - Interpolate I-order bipolynomial interpolation from the ν-th function on the coarse grid n and add the interpolant to the μ-th function on the fine grid m. Both grids are uniform; this routine uses no boundary information, hence extrapolation would frequently be used at marginal points even for bilinear (I=2) interpolation. Special subroutines are used by INTAD for important cases which can be executed faster than the general case, such as single-string grids or the special orders I=2 (bilinear) and I=4 (bicubic).

INTRB2(n,m,ν,μ) - Perform linear (order 2) interpolation from the μ-th function on boundary grid m to the ν-th function on boundary grid n, where both grids are defined on the same continuous boundary, but their lattices need not have any particular relation to each other. The interpolation is performed with respect to the piece parameter (t in Sec. 2).

INT2(n,m,ℓ) - Interpolate linear (order 2) interpolation from uniform grid m and boundary grid ℓ to uniform grid n, all assumed to have the same lattice.

INT4(n,m,ℓ) - Similar, with cubic (4-th order) interpolation.

INT2AD(n,m,ℓ) - Like INT2, but the interpolated function is added to the previous function on grid n.

4.6 Display

The routines below are used for displaying grids (also showing their relative positions), grid functions, templets and boundaries.

DISPG(n,LW,ICAR,LU) - Display uniform grid n by printing the character ICAR in page positions corresponding to grid points. LW is the line width (number of characters per line); if it is not large enough, the grid is shown in bands of LW characters each. LU is the logical output unit.

DISPGS(N,NL,LW,ICAR,LU) - Similarly display NL uniform grids simultaneously. N is a vector of NL grid numbers and ICAR is a vector of NL corresponding display characters. The grids are printed on top of each other, thus showing their relative positions. It is assumed that all grids have the same construction order and matching lattices; i.e., all lattices are subsets of one underlying lattice, used to determine the page positions.

DISPT(τ,LU) - Similarly display templet number τ.

DISPF(n,ν,LW,IFE,w,d,LU) - Display the values of the ν-th function on grid n, using Fortran format Fw.d (if IFE=0) or Ew.d (if IFE≠0), LW and LU meaning as above.

PLTLAT(n,XL,YL,IPEN) - Plot uniform grid n, in IPEN color and on XL x YL paper.

PLTBND(n,ν,IPL,INDB,IPEN,XL,YL) - Plot boundary grid n and the ν-th function on it. If INDB=0, the ordinal numbers of the gridpoints are shown. If INDB=1, their types of intersection are printed. If INDB=2, the values of the function are printed. If INDB=3, the pieces are drawn and the piece ordinal numbers are shown. IPEN is the color and XL x YL is the size of the paper. If IPL=1 this is the last plot on the present paper. It is thus possible to combine several such plots on top of each other, in a variety of colors.

DOMPLT(...,NB) - Plots the continuous boundary number NB with ELLPACK-type data.

DUMPQL - Dump the Q and L vectors (for debugging purposes).

4.7 Multigrid drivers and subroutines

GRIDPACK comes with a collection of multigrid programs that are used for testing the whole package and are also useful for solving various PDE problems and can easily be extended to solve many others. An example:

MULTIG(x_1,x_2,y_1,y_2,CHAR,x_0,y_0,H_x,H_y,M,...,RELAX,RESCAL) - Solve a differential equation using M grids with lattices

$$L(x_0,y_0;\ H_x * 2^{-k},\ H_y * 2^{-k}),\quad (1 \le k \le M).$$

A sequence of additional parameters specified by the user controls the algorithm to be an accommodative or fixed Full Multi-Grid (FMG) algorithm employing the Correction-Scheme (CS) or the Full Approximation Scheme (FAS). RELAX(n,RES) should be a routine that performs

a relaxation sweep (thus also defining the difference equations, including boundary conditions) and outputs the residual norm RES. It should be written for a general grid number n (using a KEY routine, as in Fig. 1). RESCAL(n,m) should be a routine that computes residuals in a fine grid n and transfer weighted averages of them to the coarser grid m.

Routines RELAX and RESCAL should generally be supplied by the user of MULTIG. Some fairly general routines of these kinds, called RELG and RESG, come with GRIDPACK and can be easily modified and generalized to various other cases. A library of such routines, all written in terms of general grid numbers, should, in time, be developed. An important aid in easy writing of such routines is the Grid Language (GL), described next.

5. GRID LANGUAGE (GL)

To facilitate the programming of grid "sweeping operations" (operations which are done sequentially at all or most gridpoints; e.g., a relaxation sweep), and to simplify various other aspects of using GRIDPACK and other grid-oriented programming, a special extension of Fortran called Grid Language (GL) has been developed. The GL macro-statements are written within the usual Fortran code, each macro being identified by "C*" in the first two columns, and may be continued to several card images by the usual Fortran continuation convention. These macro-statements are expanded into usual Fortran statements, including suitable calls to various GRIDPACK routines described above, by a special preprocessor named PREPACK. This preprocessor itself is written in basic Fortran, and thus enjoys full portability.

A representative list of GL macro-statements is given below. In addition to these, GL statements will also be used to have more convenient calls to GRIDPACK routines. For example, n=UNION(ℓ,m) will be used instead of CALL UNION(ℓ,m,n). The list of such statements will not appear here since it is so far only partly implemented.

For a detailed list of limitations and restrictions currently placed on GL statements, and for a detailed description of PREPACK with its three modes, see [1, Appendix III].

5.1 GL DO statements

The three dots (...) appearing in a statement below mean that indefinite number of additional groups, similar to the groups subscripted by 1 and 2, could be added to the statement. Also, we denote by (x_i^n, y_j^n) the values of the (x,y) coordinates at the (i,j) gridpoint of grid n.

DO 5 I = I1, I2, ΔI - The usual Fortran DO statement, except that I1,I2 and the increment ΔI may be any arithmetic expression, and may assume any (negative, zero or positive) integer value.

DO 5 (I,J) = GRID(n), ΔI, ΔJ, $n_1(I_1,J_1)$, $n_2(I_2,J_2)$,... - The row index I and the column index J traverse all the gridpoints (I,J) of grid n, in its construction order and in (positive or negative) increments ΔI and ΔJ, while (I_ℓ,J_ℓ) simultaneously traverse the corresponding gridpoints on n_ℓ, (ℓ=1,2,...); i.e., at each pass of this DO loop (performing the statements following the DO, up to the statement labelled 5) (I,J) is another point on grid n, and (I_ℓ, J_ℓ) is set so that $(x^{n_\ell}_{I_\ell}, y^{n_\ell}_{J_\ell}) = (x^n_I, y^n_J)$, if this is possible. When impossible, close values are set by some rules [1, AIII-4.2.2]. The grid numbers n and n_ℓ can be integer constants or variables, but should not, of course, change inside the DO loop.

DO 5 I = COLS(n), ΔI, $n_1(I_1)$, $n_2(I_2)$, ... - I traverses the columns of grid n, in increments ΔI, while I_ℓ traverses the corresponding columns of grid n_ℓ.

DO 5 J = ROWS(n), ΔJ, $n_1(J_1)$, $n_2(J_2)$,... - Similar.

DO 5 J = COL(I,n), ΔJ, $n_1(J_1)$, $n_2(J_2)$,... - J traverses the gridpoints of column I in grid n, J_ℓ assuming corresponding values on grid n_ℓ.

DO 5 I = ROW(J,n), ΔI, $n_1(I_1)$, $n_2(I_2)$, ... - Similar.

5.2 Statements inside GL DO loops

The following GL statements can typically be used inside DO-GRID or DO-COL or DO-ROW loops. The ASSIGN statement can also be used outside the loop.

ASSIGN u AS ν ON n - Means that in the present subprogram, the letter(s) u stand for the ν-th function on grid n. The function index ν and the grid number n can be either constant or variable.

u(I,J) = u(I+1,J)+u(I-1,J)-u_1(I_1,J_1)*H2 - An example of a usual Fortran replacement statement, except that u and u_1 may be grid functions. u is defined as a grid function by an "ASSIGN u AS ν ON n" statement, in which case n should appear as a grid number in the preceding DO statement. Also, u is automatically designated as a grid function if it coincides with a (constant or variable) grid number appearing in the DO statement, in which case u stands for the *first* function of that grid. Expressions like u(IX,JX) can be used, with IX and JX any arithmetic expressions in I and J, respectively, where (I,J) are the indices in the DO statement traversing the grid on which u is defined.

REGULAR(τ)
IRREGULAR 15 - The sequence of statements between these two should be executed only for gridpoints of the τ-inner subgrid (of the principal grid in the DO statement), while the sequence between IRREGULAR and the statement labelled 15 should only be executed for the points of the τ-outer subgrid. This is a way to differentiate between the treatment of irregular and regular points while still maintaining high-speed DO loops for the latter.

IF((I_1+1,J_1).IN(τ_1).n_1) GO TO 5 - Test whether (I_1 + 1, J_1) is in the τ_1-inner subgrid of grid n_1. This is a slow way to differentiate types of points and should therefore usually be used only in the sequence following an IRREGULAR statement.

6. Future Extensions

The GRIDPACK and Grid-Language system should be extended in many directions. First, improvement in the current routines, and chiefly in the interpolation collection (see Sec. 4.5) is needed, and the library of boundary-treatment routines (see Sec. 3.5) and multigrid programs (see Sec. 4.7) should significantly be expanded. Much of this may be contributed by various GRIDPACK users.

Next, *staggered grids* (interacting grids whose lattices are shifted a constant shift from each other) should be admitted. This would require an additional collection of interpolation routines, some additional GL DO statements to conveniently access shifted neighbors, and more boundary-operator programs. Then, *rotated grids* and *curved grids* should similarly be introduced. It is enough to introduce the simple family of curved grids described in [3, Sec. 2.3] or [6, Sec. 9.3], which are fully characterized by single-variable functions. This will make it possible to use *local* coordinate transformations in multigrid fast solvers, allowing very effective treatment of curved boundaries, interfaces, etc.

Moving boundary capabilities should also be added. This would make it possible to trace interfaces and interior discontinuites (e.g., shocks). Again, a collection of new interpolation and boundary-operator programs will have to be developed for this purpose.

Another, ambitious task is the extension of the current two-dimensional system to deal with *three-dimensional* grids.

References

[1] A. Brandt and Dan Ophir: GRIDPACK SYSTEM. A detailed, voluminous description, to appear as a file on MUGTAPE84, which will carry the system itself as another file.

[2] A. Brandt: Multi-level adaptive solutions to boundary-value problems. Math. Comp. 31 (1977) 333-390.

[3] A. Brandt: Multi-level adaptive techniques (MLAT) for partial differential equations: ideas and software. In: *Mathematical Software III* (John R. Rice, ed.) Academic Press, New York 1977, pp. 273-314.

[4] A. Brandt: Multi-level adaptive solutions to singular-perturbation problems. In: *Numerical Analysis of Singular Perturbation Problems* (P. W. Hemker and J. J. H. Miller, eds.), Academic Press 1979, pp. 53-142.

[5] D. Ophir: Language for processes of numerical solutions to differential equations. Ph.D. Thesis (1979), The Weizman Institute of Science, Rehovot, Israel.

[6] A Brandt: Guide to multigrid methods. In [7], pp. 220-312.

[7] W. Hackbusch and U. Trottenberg (eds.): *Multigrid Methods*. Springer-Verlag, 1982.

[8] A Brandt: Introductory remarks on multigrid methods. In: *Numerical methods for Fluid Dynamics* (K. W. Morton and M. J. Baines, eds.), Academic Press 1982, pp. 127-134.

[9] J. R. Rice: ELLPACK 80, Users Guide - Preliminary Version, CSD-TR 306, May 9, 1980.

```
      SUBROUTINE PUTZ(N)
      COMMON J1(65), J2(65), JR(65)
      CALL KEYS (N,I1,I2,IR,J1,J2,JR,...)
      DO 1 I=IR+I1, IR+I2
      DO 1 IJ=JR(I)+J1(I),JR(I)+J2(I)
    1 Q(IJ)=0
      END
```

Figure 1. PUTZ: An illustration of using a KEY routine.

PUTZ is a general routine for setting to zero the function defined on grid number N. Notice that this routine can be used for any uniform-grid function, and in any program which uses GRIDPACK; in fact, it is enough that the program includes KEYS. Routines like PUTZ, and far more complicated ones, can thus be written once for all; having been written by one GRIDPACK user, all others can benefit from it. Notice also the full efficiency of the central DO loop, and the ease of writing: Q(IJ) stands for $U^N_{I,J}$. Generally, Q(JR(IR+i)+j) would stand for any u^N_{ij}, with expressions like JR(IR+i), for all i needed in a central loop, being evaluated before the loop. If the central loops are on i, the grid can be transposed (see Sec. 4.3).

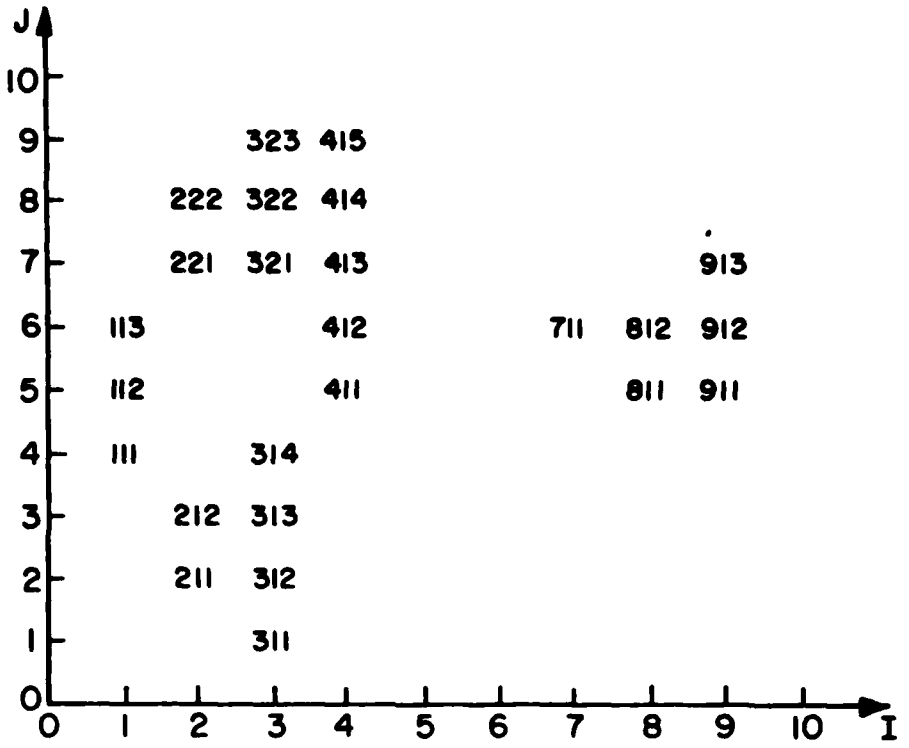
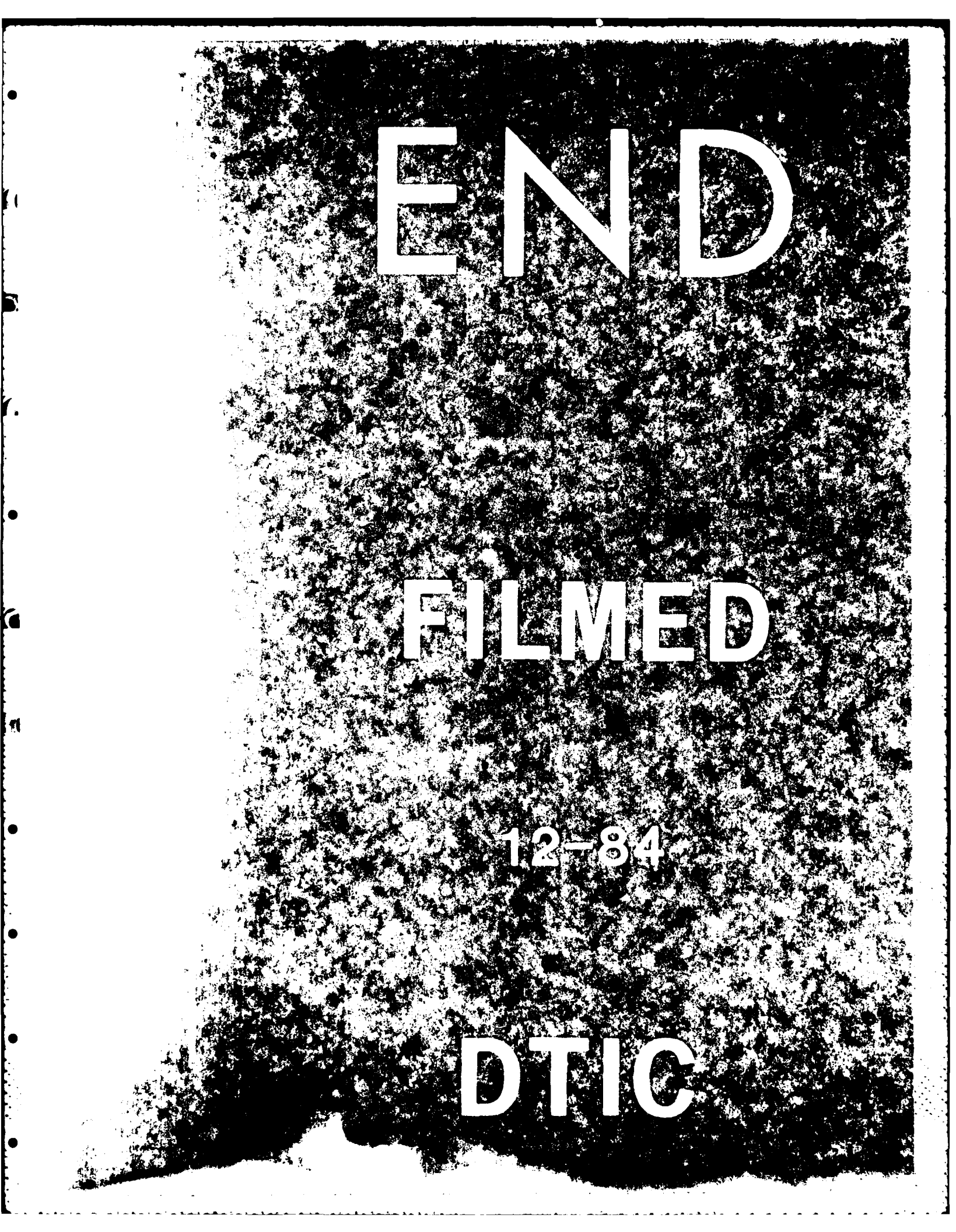


Figure 2. The structure of a vertical uniform grid and a function on it.

The grid has two sets (strings of consecutive columns): (1,2,3,4) and (7,8,9). At gridpoint (I,J) the figure shows the value of a function $U_{IJ} = \nu_1\nu_2\nu_3$, where $\nu_1 = I$ is the column number, the digit ν_2 is the vertical-string ordinal number within the column, and ν_3 is the gridpoint ordinal number within the string (not its row number J).


END
FILMED
12-84
DTIC