(1)

David Taylor Research Center

Bethesda, Maryland 20084-5000

SSPD-90-175-14 January 1990

Ship Structures and Protection Department

Research and Development Report

A FAST PC-BASED SOLVER FOR SIMULTANEOUS LINEAR EQUATIONS

by

Peter N. Roth

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

MAJOR DTRC TECHNICAL COMPONENTS

CODE 011	DIRECTOR OF TECHNOLOGY, PLANS AND ASSESSMENT
12	SHIP SYSTEMS INTEGRATION DEPARTMENT
14	SHIP ELECTROMAGNETIC SIGNATURES DEPARTMENT
15	SHIP HYDROMECHANICS DEPARTMENT
16	AVIATION DEPARTMENT
17	SHIP STRUCTURES AND PROTECTION DEPARTMENT
18	COMPUTATION, MATHEMATICS & LOGISTICS DEPARTMENT
19	SHIP ACOUSTICS DEPARTMENT
27	PROPULSION AND AUXILIARY SYSTEMS DEPARTMENT
28	SHIP MATERIALS ENGINEERING DEPARTMENT

David Taylor Research Center

Bethesda, Maryland 20084-5000

SSPD-90-175-14

January 1990

A FAST PC-BASED SOLVER FOR SIMULTANEOUS LINEAR EQUATIONS

by

Peter N. Roth

CONTENTS

1. Abstract

An indirect (pointer-based) datastructure for the storage of a symmetric, positive definite matrix of coefficients, and the corresponding "right hand side" vector is presented. This datastructure is useful in many kinds of engineering work, most notably finite element analysis. Procedures for accessing the datastructure are demonstrated via the small Turbo Pascal program KxR.

2. Introduction

Matrix methods in general, and the finite element method (FEM) in particular, are now common engineering tools. The equations to be solved in the vast majority of finite element stress analyses are

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} \\ k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} \\ k_{31} & k_{32} & k_{33} & k_{34} & k_{35} & k_{36} \\ k_{41} & k_{42} & k_{43} & k_{44} & k_{45} & k_{46} \\ k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} \\ k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix} = \begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{Bmatrix} \qquad (1)$$

(KR)

or, in matrix notation,

$$\mathbf{Kx} = \mathbf{R}$$

where K is the positive definite† stiffness matrix, x is the vector of unknowns, and R is the vector of knowns. In the general case, of course, the number of equations is much larger than 6, often numbering into the thousands.

The majority of techniques for solving (1) are variants of a method introduced by the mathematical genius Carl F. Gauss. See, for example, [Ralston]‡ or [Carnahan], for an explication of the technique. The demonstrations in [Bathe] and [Wilson] are outstanding. The original Gaussian method has been supplanted in recent times by methods that are appropriate for computer implementation.

2.1 Motivation

Since there are published programs, why bother writing yet another solver? The major need was for a solver of "reasonable" capacity, in Pascal, for a personal computer (PC).

2.1.1 Pascal in general Pascal, a programming language invented in the early 1970's by Niklaus Wirth, is intrinsicly legible code. That is, a reader and writer of the code can understand a Pascal program more easily than code written in other languages. Pascal also facilitates data structuring, a language capability that is absent from FORTRAN and BASIC. Finally, Pascal is a readable step along the pathway to C, the inscrutable (but ubiquitous) language in which it seems most programs will eventually be written.

2.1.2 Turbo Pascal in particular Version 5.0 of the Turbo Pascal compiler manufactured by Borland International [BorlandPU], [BorlandR] provides a superb built-in debugger. Borland has implemented the Modula-2 *module* concept, which allows separate compilation of code modules while retaining the strong type-checking of Pascal.† The module concept, present in

† A matrix $\mathbf{Q}$ is *positive definite* if the product $\mathbf{x}^T\mathbf{Q}\,\mathbf{x} > 0$ for any non-zero $\mathbf{x}$.

‡ References are listed at the end of the paper.

† Borland's spelling for the word "module" is "Unit."

almost all languages except BASIC and standard Pascal, speeds up program development enormously. Overlays are available in Turbo Pascal to allow very large programs to fit into the DOS3x memory limitation of 640K bytes. And, finally, Turbo Pascal is faster than any other compiler for the PC, for *any* language, especially FORTRAN. This extremely high speed makes for a good environment for program development, for "small" programs.

2.1.3 Drawbacks to Turbo Pascal Turbo Pascal is *not* Pascal: some of Turbo Pascal is not portable to other machines.

3. Data Storage on PC's

3.1 Capacity

The maximum addressable storage on a DOS machine is usually abbreviated to the number 640K bytes. The architecture of the 80x86 series machines accesses storage in *segments*. The maximum size of a single segment is 65520 bytes.

3.2 Floating Point Numbers

Turbo Pascal allows five representations of floating point numbers, as shown in Table 1:

Table 1: Floating Point in Turbo Pascal

Type	Bytes	N/segment
single	4	16380
real	6	10920
double	8	8190
extended	10	6552
comp	10	6552

The first column gives the name of the Pascal floating point type, and the second column shows the number of bytes required for that type. The third column of the table shows the maximum number of floating point variables allowed in a segment. This effectively limits the maximum size of data a program can use, and the maximum size of a single array, unless other techniques are used. Some PC FORTRAN compilers map arrays to appropriate portions of segments, but control of this process is not available in the language directly.

Interestingly enough, computations are faster using the Turbo Pascal "double" than the "real", (using a numerical co-processor chip) because the truncation of the result of floating point arithmetic operations is avoided. Certainly, computations are more accurate in "double" than in "real" or "single".

4. A Comparison of Matrix Storage Techniques

The crucial datastructure in a FEM program is the stiffness matrix. Because the majority of the procedures not involved with reading input or printing output are related to accessing this great datastructure, the form of the stiffness matrix dictates the structure of the FEM code.

Usually, one wishes to mesh a structure with the economically largest possible number of elements (*numel*) because accuracy of analysis is directly related to the fineness of the grid. Imagine a mesh of 2D planar 8-noded isoparametric elements. These elements have two degrees of freedom per node, and therefore, 16 equations per element. In order to solve a 2D FEM mesh with ONE of these elements, one must be able to store and solve at least 16 equations. Let us consider the following four storage forms for K: Square, Triangular, Banded, and Skyline.

4.1 Square

An n-dimensional square matrix may be illustrated with

$$\begin{matrix}
11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\
21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 \\
31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\
41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \\
51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 \\
61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 & 69 \\
71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 & 79 \\
81 & 82 & 83 & 84 & 85 & 86 & 87 & 88 & 89 \\
91 & 92 & 93 & 94 & 95 & 96 & 97 & 98 & 99
\end{matrix}$$

where $n=9$ in this case. Assuming that calculations will be performed using "doubles", the *total storage for this matrix is* $n^2 \leq 8190$ where n is the size of K. Thus, the maximum n is 90, and the upper bound on *numel* in a 2D mesh is about 8, which is interesting, but not very useful.

A complete (and naive) Pascal program to solve equation (1) is developed in [Wirth] using the stepwise refinement technique.

```
program gauss_elimination ;
const
        n =6 ;
var
        i, j, k: 1 .. n ;
        p : real ;
        A : array [ 1 .. n, 1 .. n ] of real ;
        B : array [ 1 .. n ] of real ;
        X : array [ 1 .. n ] of real ;
begin
        { assign values to A and B }
        for k :=1 to n do begin
                p :=1.0 / A[k,k ] ;
                for j :=k + 1 to n do
                        A[k,j ] :=p * A[k,j ] ;
                B[k ] :=p * B[k ] ;
                for i :=k + 1 to n do begin
                        for j :=k + 1 to n do
                                A[i,j ] :=A[i,j ] - A[i,k ] * A[k,j ] ;
                        B[i ] :=B[i ] - A[i,k ] * B[k ]
                end { i }
        end { k } ;
        k :=n ;
        repeat
                p :=B[k ] ;
                for j :=k + 1 to n do
                        p :=p - A[k,j ] * X[j ] ;
                X[k ] :=p ;
                k :=k - 1
        until k =0
        { X[1 ] ... X[n ] are the solutions }
end.
```

Program 1: Gaussian Solution of Simultaneous Equations

Program 1 gives the flavor of Gaussian elimination, but needs some improvement. There should at least be a check for division by zero.

4.2 Triangular

Because K is symmetric, or can be made so by appropriate multiplications, only "half" the array, either above or below the diagonal, need be stored.

```
11  12  13  14  15  16  17  18  19
    22  23  24  25  26  27  28  29
        33  34  35  36  37  38  39
            44  45  46  47  48  49
                55  56  57  58  59
                    66  67  68  69
                        77  78  79
                            88  89
                                99
```

The storage required for K is $n(n+1)/2 \leq 8190$, so $n \leq 127$, and the maximum *numel* is about 12. This is an improvement over the square scheme, but hardly enough to make the derivation of a structure to store the array worthwhile.

4.3 Banded

A further observation of FEM stiffness equations shows that for judiciously numbered meshes, the matrix appears to be "banded"; *i.e.*, all of the non-zero terms in K are within a certain distance from the diagonal. *E.g.*,

$$\begin{array}{ccccccccc}
11 & 12 & 13 & 14 & 0 & 0 & 0 & 0 & 0 \\
 & 22 & 23 & 24 & 25 & 0 & 0 & 0 & 0 \\
 & & 33 & 34 & 35 & 36 & 0 & 0 & 0 \\
 & & & 44 & 45 & 46 & 47 & 0 & 0 \\
 & & & & 55 & 56 & 57 & 58 & 0 \\
 & & & & & 66 & 67 & 68 & 69 \\
 & & & & & & 77 & 78 & 79 \\
 & & & & & & & 88 & 89 \\
 & & & & & & & & 99
\end{array}$$

If it is known that certain values of K are zero, then they need not be stored. This leads to the auxiliary descriptor of K: the maximum bandwidth, commonly represented in programs as the variable *maxbw*. The smaller the bandwidth, the less storage required.

K can then be stored in a rectangular array of the form

$$\begin{array}{cccc}
11 & 12 & 13 & 14 \\
22 & 23 & 24 & 25 \\
33 & 34 & 35 & 36 \\
44 & 45 & 46 & 47 \\
55 & 56 & 57 & 58 \\
66 & 67 & 68 & 69 \\
77 & 78 & 79 & * \\
88 & 89 & * & * \\
99 & * & * & *
\end{array}$$

where the asterisks indicate "wasted" space. Storage required: $n \cdot maxbw \leq 8190$. Thus, n varies from 90 (with a *maxbw* of 90) to 8190 (with a *maxbw* of 1). Recalling the simplistic 2D mesh, the maximum number of elements is 8190/16 =511. Now we're getting somewhere! Of course, the algorithm that solves equation (1) when K is stored in banded form will be more complicated than Program 1, because it will have to map the subscripts of K to the coordinates of the rectangular array.

4.4 Skyline

Wilson, and others, have observed that K stored in banded form may store unnecessary zeroes. The bandwidth at any location in K may vary from small to large, forming a "skyline", as shown typically below for n =17. Note that the solvers described in [Wilson] and [Bathe] eliminate computations that use terms outside the skyline, and are among the fastest solvers in common use.

```
x   x                           x
    x   x   x                   x
        x   x                   x
            x   x               x
                x   x           x                       x       x
                    x   x       x                       x       x
                        x   x   x                       x       x
                            x   x   x   x               x       x
                                x   x   x   x           x       x
                                    x   x   x   x       x       x
                                        x   x   x       x       x
                                            x   x   x   x       x
                                                x   x   x       x
                                                    x   x   x   x
                                                        x   x   x
                                                            x   x
                                                                x
```

Since the "columns" are of arbitrary height, so a little more information about K than *n* and *maxbw* is needed. Unfortunately, FORTRAN has no direct facility for storing this datastructure.

The technique used in [Wilson] and [Bathe] to store the skyline array in the programs SAP4, NONSAP and ADINA conceptually numbers each element of the array consecutively. The indexing scheme proceeds from the diagonal element, numbered first, through each element in the column, numbering towards the skyline, *viz.*,

```
1   3                           25
    2   5   8                   24
        4   7                   23
            6   10              22
                9   12          21                      54      70
                    11  14      20                      53      69
                        13  16  19                      52      68
                            15  18  28  32              51      67
                                17  27  31  36          50      66
                                    26  30  35  40      49      65
                                        29  34  39      48      64
                                            33  38  43  47      63
                                                37  42  46      62
                                                    41  45  57  61
                                                        44  56  60
                                                            55  59
                                                                58
```

K is then stored in a one-dimensional (linear) array, with an auxiliary integer array D to point to the diagonal terms of K, thus:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
D[i]	1	2	4	6	9	11	13	15	17	26	29	33	37	41	44	55	58	71

For example, the indexes of all terms in the upper portion of column 9 are defined by the limits D[9] to D[10]−1 (= 17 to 25). The last term (D[18]) is a sentinel pointer to the *n*+1th column. Wilson uses these FORTRAN "pointers" to calculate column heights, and requires an additional pointer to enable calculation of the height of the last column.

The *profile* P is defined mathematically as

$$P = \sum_{i=1}^{n} D_i$$

which is simply the number of non-zero terms within the skyline. The total storage (in bytes) required to store K must now include storage for D:

$$\text{Storage} = P \cdot \text{Sizeof(float)} + (n+1) \cdot \text{Sizeof(integer)}$$

where *Sizeof* returns the number of bytes required to store its argument. The skyline technique will probably be a gain over the banded storage scheme although K is still stored in a single segment.

Note that the trade-offs made to gain storage space will make the solution algorithm more complex and harder to understand than Program 1. Since Wilson uses FORTRAN, that language's built-in features to clearly calculate subscripts must be supplemented by array access mechanisms which map K to the skyline. Now consider Pascal, a language that has the facility for direct access to unusual and arbitrary datastructures.

5. Implementation of the Skyline Data Structure in Turbo Pascal

So far, about the best that any of the storage schemes yield is an upper limit on the number of 2D elements somewhere around 500, certainly enough for programs analyzing small models. Is it possible to do any better without resorting to a file-based storage scheme? The answer is of course "yes", but it requires that the use of *pointers,* a data structuring tool present in most modern languages.

5.1 Pointers

From [Jensen], Chapter 10:

> "A *static* variable (staticly allocated) is one which is declared in a program and subsequently denoted by its identifier. It is called static, for it exists (*i.e.,* memory is allocated for it) during the entire execution of the block to which it is local. A variable may, on the other hand, be generated dynamically (without any correlation to the static structure of the program) by the procedure **new**. Such a variable is consequently called a *dynamic variable.*
>
> Dynamic variables do not occur in an explicit variable declaration and cannot be referenced directly by identifiers. Instead, generation of a dynamic variable introduces a *pointer* value (which is nothing other than the storage address of the newly allocated variable). Hence, a pointer type P consists of an unbounded *(sic)* set of of values pointing to elements of a given type T. P is then said to be *bound* to T. The value *nil is always an element of* P and points to no element at all.
>
> type <identifier> = ^ <type identifier>
>
> If, for example, p is a pointer to a variable of type T by the declaration
>
> var p : ^ T
>
> then p is a reference to a variable of type T, and p^ denotes that variable.† In order to create or allocate such a variable, the standard procedure **new** is used. The call new(p) allocates a variable of type T and assigns its address to p.
>
> Pointers are a simple tool for the construction of complicated and flexible data structures. If the type T is a record structure that contains one or more fields of type ^T, then structures equivalent to arbitrary finite graphs may be built, where the T's represent the nodes, and the pointers are the edges."

Pointers have several advantages: they permit programmers to develop the most flexible of data structure; they permit fast access to data items without need for subscript computation; they do not require recourse to assembly language; and they allow access to all of available memory. The latter is crucial. For example, the memory available to a PC program is typically 470 Kbytes, (approximately 58750 doubles as opposed to the 8190 indicated in Table 1). By using

† If p is a pointer to a type T, the expression p^ is said to *dereference* the pointer. Alternatively, p is the address, p^ is the "value" at the address.

pointers to access this storage, the upper bound on *numel* for our simple 2D problem becomes 5874 elements! Furthermore, the upper bound on *numel* for a 20-node 3D isoparametric element mesh is 1631. The pointer data storage technique therefore affords the capability to solve significant FEM problems in main storage on the PC.

The First Law of Thermodynamics paraphrased states that you don't get something for nothing. Great flexibility means that the programmer must impose discipline to control code and data. (Pointers have been called the datastructure equivalent of the GOTO statement.) The pointer concept is foreign to the FORTRAN programmer, so a significant number of engineers are unlikely to easily understand pointer code without sufficient time to study the concept. The program becomes more intellectually dense, since more concepts are embedded in each character. Access to a data item is by indirection, rather than directly. Both programmer and code readers not only move their lips when reading pointer code, but read out loud, assume awkward postures, and draw little diagrams. So much for "intrinsicly legible code" in Pascal. However, pointers *are* supported by the language itself, so learning and using pointers becomes more natural and straight-forward with practice.

5.2 Memory allocation in Turbo Pascal

The memory available to a Turbo Pascal programmer comes from the "Heap", an area in memory accessible by several Turbo procedures and functions.

Dispose	Releases storage previously allocated to a variable. Used in cooperation with New.
Freemem	returns to the heap exactly the number of bytes allocated to a pointer by Getmem.
Getmem	allocates any number of bytes (and therefore, *partial records*) to a pointer. Used in cooperation with Freemem.
Mark	establishes a pointer to a location on the heap; used in cooperation with Release.
Memavail	returns the memory available at the current point in the program's execution.
Maxavail	returns the maximum memory available on the heap.
New	allocates storage to a variable. Used in cooperation with Dispose.
Release	frees *all* memory above a pointer established by Mark.

The cooperating procedure pairs are therefore: New, Dispose; Mark, Release; and Getmem, Freemem. Getmem and Freemem are non-standard Pascal, and are the closest to the C language storage allocation routines.

5.3 The New Data Structure

The ingredients include:

1. Memavail from Turbo Pascal. This procedure will indicate the total storage available.
2. Mark and Release from Turbo Pascal. Mark will allocate the total heap storage to the arrays needed for the solution of (1). Release will return the heap storage to the operating system on completion of the job. Although this may seem unnecessarily fastidious, my machine has run out of memory during the development of KxR, because "no one told him" that he regained the memory.
3. Getmem from Turbo Pascal (the key ingredient). Getmem will be used to allocate to each column *only that storage needed to contain the column.*

4. **height**, a pointer to a variable length integer array that describes the height of each column. This replaces the D array of Wilson's method, and all the explicit calculations of columnar bounds are handled directly by the language.
5. K, a pointer to an array of pointers containing the elements of K. The stiffness matrix is therefore accessed doubly indirectly (pointers to pointers). Although this may seem complicated, it actually makes the code simpler and clearer than the linear FORTRAN technique.

The skyline structure in Pascal will therefore be represented in a manner similar to Wilson's. However, each column of the array will be numbered from the diagonal to the uppermost element, rather than numbering each element of the array. This alternative can be demonstrated:

```
  1   2                           9
      1   2   3                   8
          1   2                   7
              1   2               6
                  1   2           5                      11      13
                      1   2       4                      10      12
                          1   2   3                       9      11
                              1   2   3   4               8      10
                                  1   2   3   4           7       9
                                      1   2   3   4       6       8
                                          1   2   3       5       7
                                              1   2   3   4       6
                                                  1   2   3       5
                                                      1   2   3   4
                                                          1   2   3
                                                              1   2
                                                                  1
```

Figure 1: Conceptual Layout of K

The array of column heights is then

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
height^[i]	1	2	2	3	2	2	2	2	9	3	4	4	4	3	11	3	13

where we write "height^[i]" to show what the *values* are. Remember, "height" is just a pointer; the presence of the "^" dereferences the pointer.

With this groundwork laid, it is possible to write the code to implement this structure.

6. The code

The allocation of the data structure is discussed first, followed by the solver itself. Additional utility routines for access to the data structure follow.

6.1 Allocation of Storage

The allocation of storage follows this algorithm:

1. Assume the number of equations *neq* is known.
2. Then an array of *neq* integers is needed to hold *height*. Allocate this array.
3. Determine the height of each column and store it in the appropriate location of *height*.
4. Since *neq* and *height* are known, the shape of K is completely determined. Allocate storage for an array of *neq* pointers to columns of K (as *i* varies from 1 to *neq*, K^[*i*] points to column *i*). To each pointer K^[*i*], allocate *height*^[*i*] floating point values. With this scheme, the term on the diagonal in the *i*-th column is written K^[*i*]^[1] and the *j*-th term from the diagonal in the *i*-th column is written K^[*i*]^[*j*].
5. Allocate *neq* floating point values for the right hand side vector R .

The Turbo Pascal procedures to accomplish the algorithm outlined above are shown below.

6.1.1 Allocation of height The data *types* involved are

```
TYPE
          float                      = double ; (* change to suit accuracy *)

CONST
          MAXEQNS                    = 65520 div Sizeof( float ) ;

TYPE
          column_height_array        = array [ 1 .. MAXEQNS ] of word ;
          column_height_type         = ^column_height_array ;

VAR
          height :                   column_height_type ;
```

Note that Turbo Pascal allows the declarative matter of a program to be in arbitrary order (*not* standard Pascal). Also, simple computations are permitted in the declarations: div is the Pascal "integer divide" operation. The type *word* permits variables of that type to assume values between 0 and 65532. *Float* is defined as *double* only once in the solver; thus the accuracy may be changed with the alteration of this single switch. MAXEQN is arbitrarily defined as the maximum number of floats that will fit in a single memory segment.

The procedure to allocate height:

```
(*****
 *
 *.  allocate_height - allocate storage for the column heights.
 *
 *)
procedure allocate_height ;
  VAR
    c : longint ;
  begin (* allocate_height *)

    c := neq * sizeof( word ) ;

    if  c > MemAvail  then begin
       writeln( 'Out of memory attempting to allocate ', c,
              ' bytes  to column heights - allocate_height.' ) ;
       abort
    end
    else
      Getmem ( height, c )

  end   (* allocate_height *) ;
```

In addition to the procedures already mentioned, **allocate_height** makes use of **abort**:

```
(*****
 *
 *.  abort - release memory and die.
 *
 *)
procedure abort ;
  begin (* abort *)

    release( origin ) ;
    halt( 1 )

  end   (* abort *) ;
```

which presumes that the initialization of the program includes the statement

```
mark( origin ) ;
```

Halt is the Turbo Pascal procedure which terminates program execution.

6.1.2 Allocation of K Declarations of the data types associated with the storage of K and R :

```
TYPE
          column_type          = array [ 1 .. MAXEQNS ] of float ;
          column_ptr           = ^column_type ;

          column_array         = array [ 1 .. MAXEQNS ]  of column_ptr ;
          matrix               = ^column_array ;

VAR
          K : matrix ;
          R : column_ptr ;
```

The procedure to allocate the storage for K begins with "allocate_K" which allocates storage to the pointers to the columns:

```
(*****
 *
 *.  allocate_K - allocate storage for the matrix of coefficients.
 *
 *)
procedure allocate_K ;
  VAR
    c : longint ;
    i : word ;

  begin (* allocate_K *)

    c := neq * sizeof( pointer ) ;

    if  c > MemAvail  then begin
      writeln ( 'Out of memory attempting to allocate ', c,
               ' bytes to [K] -  allocate_K.' ) ;
      abort
    end
    else
      Getmem ( K, c ) ;

    for  i := 1  to  neq  do
      allocate_column( i, height^[i] )

  end   (* allocate_K *) ;
```

The first part of this procedure is similar to **allocate_height**. In the second part, the **for** statement calls **allocate_column** to allocate the storage for each column of K. This procedure is:

```
(*****
 *
 *.  allocate_column - arrange storage for the i-th column.
 *
 *)
procedure allocate_column ( i : integer ; (* the column of interest *)
          r : word    (* the size desired *)
         ) ;
  VAR
    c : longint ;
  begin

    c := r * sizeof_float ;

    if  c > MemAvail  then begin
      writeln ( 'Out of memory attempting to allocate ', c,
             ' bytes  to column ', i, ' - allocate_column.' ) ;
      abort
    end
    else begin
      Getmem ( K^[i], c ) ;
    end

  end (* allocate_column *) ;
```

Allocate_column uses a previously computed *sizeof_float*, which is

```
sizeof_float :=Sizeof(float);
```

Note that the neat arrangement of K in Figure 1 is not necessarily how the datastructure is arranged in memory. **Getmem** searches the heap for the next block of the appropriate size, and assigns it to the pointer. Because the *language* accommodates this process, the actual locations in storage assigned to each column of K are of little concern.

6.1.8 Allocation of R The right hand side vector is allocated by the following:

```
(*****
*
*.  allocate_vector - allocate storage for a 'right-hand-side-like' vector.
*
*)
procedure allocate_vector ( VAR R : column_ptr ) ;

  VAR
    c : longint ;
  begin (* allocate_vector *)

    c := neq * sizeof_float ;

    if  c > MemAvail  then begin
      writeln ( 'Out of memory attempting to allocate ', c,
              ' bytes to column vector - allocate_vector.' ) ;
      abort
    end
    else
      Getmem ( R, c )

  end  (* allocate_vector *) ;
```

All storage necessary to the solution of (1) has now been allocated.

6.2 The solver

The solver itself is an adaptation of the LDL^T technique demonstrated in [Wilson] and [Bathe]. Control of the solution is via the executive procedure **active_column_solver:**

```
(*****
 *
 *.  active_column_solver - Gaussian solution of simultaneous equations.
 *
 *)
procedure active_column_solver ;

  begin (* active_column_solver *)

    write( 'Decomposition     ...' ) ;
    decompose ;
    writeln( ' done.' );

    write( 'Reduction         ...' ) ;
    reduce_rhs ;
    writeln( ' done.' ) ;

    write( 'Back substitution ...' ) ;
    back_substitute ;
    writeln( ' done.' )

  end  (* active_column_solver *) ;
```

Program 2a: Active Column Solver - Executive

Messages surround the major phases of the solution process so the solution may be monitored from the screen.

6.2.1 Decomposition The decomposition is handled by decompose:

```
(*****
 *
 *.  decompose - LDU decomposition of coefficient matrix.
 *
 *)
procedure decompose ;
  VAR
    b, c : float ;
    n, p, q, s, t, u, v : word ;
  begin (* decompose *)
    for n := 1  to  neq  do begin
                (*
                 * propagate effect of previous
                 * off-diagonal terms to the current column
                 *)
      if  height^[n] > 1  then begin
        (* p = first previous column of interest *)
        p := n - height^[n] + 2 ;
        for  q := height^[n] - 1  downto  2  do begin
          s := 2;
          t := q + 1 ;
           (* u = number of multiplies to perform *)
          u := min( height^[p] - 1,  height^[n] - q ) ;
          c := 0.0 ;
          for  v := 1  to  u  do begin
            c := c + K^[p]^[s] * K^[n]^[t] ;
            Inc( s ) ;
            Inc( t )
          end (* v *) ;
          K^[n]^[q] := K^[n]^[q] - c ;
          Inc( p )   (* the next "previous column" *)
        end (* q *)
      end (* if *) ;
                (*
                 * propagate effect of previous
                 * diagonal terms to the current column
                 *)
      p := n ;
      b := 0.0 ;
      for  q := 2  to  height^[n]  do begin
        Dec( p ) ;
        c := K^[n]^[q] / K^[p]^[1] ;        (* the "Lij" terms *)
        b := b + c * K^[n]^[q] ;
        K^[n]^[q] := c
      end (* q *) ;
      K^[n]^[1] := K^[n]^[1] - b ;
      if  K^[n]^[1] <= 0.0  then  solerr( n )
    end (* n *)
  end  (* decompose *) ;
```

Program 2b: Active Column Solver - Decomposition

Although the "^" notation may look a little strange, the code bears a striking resemblance to non-pointer code (compare with **Program 1**). The pointers obviate explicit calculation of subscripts of terms in K, however, so the code is relatively clean. While studying **decompose**, bear in mind that K^[p]^[1] is the diagonal term in column p, and K^[p]^[2] is the first off-diagonal term. Probably, the most efficient way to observe how the solver works is to prepare a small test case (similar to the example demonstrating use of KxR), and watch the program run with the Turbo Debugger.

The function **min** is simply:

```
(*****
 *
 *.  min - returns smallest integer value.
 *
 *)
function min ( a, b : integer ) : integer ;
  begin (* min *)
    if  a <= b  then  min := a  else  min := b
  end  (* min *) ;
```

Inc and Dec are Turbo Pascal procedures that, respectively, increment, and decrement, their arguments. They map directly to machine instructions, so are quite fast.

In case of trouble, **decompose** calls **solerr**:

```
(*****
 *
 *.  solerr - prints error message and dies.
 *
 *)
procedure solerr ( n : integer ) ;
  begin (* solerr *)
    writeln( '*** Diagonal term in column ', n, ' = ', K^[n]^[1] ) ;
    writeln( '    Coefficient matrix is NOT positive definite. - solerr.' ) ;
    halt( 1 )
  end  (* solerr *) ;
```

Solerr should probably use **abort**, rather than **halt**.

6.2.2 Reduction The reduction of the right hand side is performed by **reduce_rhs:**

```
(*****
 *
 *.  reduce_rhs - using factors computed by 'decompose'.
 *
 *)
procedure reduce_rhs ;
  VAR
    c : float ;
    n, p, q : word ;
  begin (* reduce_rhs *)
    for n := 1 to neq do begin
      p := n - 1 ;        (* first "previous column" *)
      c := 0.0 ;
      for q := 2 to height^[n] do begin
        c := c + K^[n]^[q] * R^[p] ;
        Dec( p )      (* next "previous column" *)
      end (* q *) ;
      R^[n] := R^[n] - c
    end (* n *) ;
  end  (* reduce_rhs *) ;
```

Program 2c: Active Column Solver - Reduction

6.2.3 Back Substitution The back substitution step is handled by **back_substitute:**

```
(*****
 *
 *.  back_substitute - complete solution of the equations.
 *
 *)
procedure back_substitute ;
VAR
  n, p, q : word ;
  begin (* back_substitute *)
    for n := 1 to neq do
      R^[n] := R^[n] / K^[n]^[1] ;
    for n := neq downto 2 do begin
      p := n - 1 ;
      for q := 2 to height^[n] do begin
        R^[p] := R^[p] - K^[n]^[q] * R^[n] ;
        Dec( p )
      end (* q *)
    end (* n *)
  end  (* back_substitute *) ;
```

Program 2d: Active Column Solver - Back Substitution

6.3 Accessing the Data Structure via Utility Routines

Several of the procedures and functions that provide an interface to the datastructure are commented on below. A more complete set is listed in the Appendix.

6.3.1 Clearing K In order to use the datastructure, most FEM programs will require a "clean slate." The following procedure sets the contents of K to 0.0:

```
(*****
 *
 *.  clear_K - set the contents of K to 0.0.
 *
 *)
procedure clear_K ;
  VAR
    i : word ;
  begin (* clear_K *)
    for i := 1  to  neq  do
      set_vector( K^[i], height^[i], 0.0 )
  end  (* clear_K *) ;
```

Clear_K makes use of the more general procedure **set_vector** (below). The danger lurking here is that undisciplined use of **set_vector** will clear more storage than has been allocated, possible destroying the program or operating system code.

```
(*****
 *
 *.  set_vector - set  V^[1..n]  to  W.
 *
 *)
procedure set_vector ( V : column_ptr;   n : word;   W : float ) ;
  begin (* set_vector *)
    for  n := 1  to  n  do
      V^[n] := W
  end  (* set_vector *) ;
```

6.3.2 Adding a Term to K Given the location *at_row, at_col* in a "full" K, this procedure adds *term* to the appropriate location in K stored in skyline form. That is, add_to_K maps the standard matrix subscripts to the skyline. Relatively cheap insurance is provided by the if statements, which avoid storing a term in a random memory location.

```
(*****
 *
 *.  add_to_K - add 'term' at 'at_row,at_col'
 *           where at_row, at_col give the location in
 *           a "full" matrix.
 *
 *)
procedure add_to_K ( term : float ;  at_row, at_col : word ) ;
  VAR
    i : word ;
  begin (* add_to_K *)
    if  at_row <= at_col  then begin      (* K is upper triangle only *)
      i := Succ( at_col - at_row ) ;
      if  i <= height^[at_col]  then      (* insurance *)
        K^[at_col]^[i] := K^[at_col]^[i]  +  term
    end (* if *)
  end  (* add_to_K *) ;
```

6.3.3 The Product $f = K * x$ The usual definition of a matrix multiplying a vector is $\mathbf{f} = \mathbf{Kx}$ or, using indices, $f_i = \sum_{j=1}^{n} k_{ij} x_j$. Let K be symmetric, and consider the equation

$$\begin{Bmatrix} f_1 \\ f_2 \\ f_3 \end{Bmatrix} = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{12} & k_{22} & k_{23} \\ k_{13} & k_{23} & k_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}$$

Performing the multiplication yields the following terms for **f**:

$$\begin{aligned} f_1 &= k_{11}x_1 + k_{12}x_2 + k_{13}x_3 \\ f_2 &= k_{12}x_1 + k_{22}x_2 + k_{23}x_3 \\ f_3 &= k_{13}x_1 + k_{23}x_2 + k_{33}x_3 \end{aligned}$$

The datastructure only stores "half" of K, however. Hence, the computation of the product is reordered such that all the terms on and above the diagonal are used first, thus:

$$\begin{aligned} f_1 &= k_{11}x_1 \\ f_2 &= k_{12}x_1 + k_{22}x_2 \\ f_3 &= k_{13}x_1 + k_{23}x_2 + k_{33}x_3 \end{aligned}$$

This of course is equivalent to using all the terms to the *left* of the diagonal in a full matrix.

Then, each column is considered in turn; the terms at the appropriate "row" (= height) are used to accumulate the missing terms into **f**: Using column 2,

$$f_1 = f_1 + k_{12}x_2$$

and using column 3,

$$\begin{aligned} f_1 &= f_1 + k_{13}x_3 \\ f_2 &= f_2 + k_{23}x_3 \end{aligned}$$

It is seen that the resulting **f** is correct. This algorithm is implemented by the following procedure. Note that the numbering of K in the procedure agrees with that of the datastructure.

```
(*****
 *
 *.  KtimesX - column vector F := skyline matrix K times column vector X.
 *          uses global variables 'neq', 'K' and 'height'
 *
 *   K is symmetric:
 *     k11        k44
 *         k21    k43
 *             k31 k42
 *     (sym)       k41
 *
 *   Part 1 uses each column from the diagonal to the maximum
 *   height in each column as a row.  This ensures that
 *   the terms to the left of the diagonal are used in the
 *   multiplication.
 *
 *   Part 2 adjusts the previous sums by using the terms above
 *   the diagonal as the remainder of the row.
 *
 *)
procedure KtimesX ( F, X : column_ptr ) ;
  VAR
    i, j, q : word ;
    s : float ;
  begin (* KtimesR *)
    for i := 1  to  neq  do begin
                  (* Part 1 *)
      q := height^[i] ;
      s := 0.0 ;
      for  j := 1  to  height^[i]  do begin
        s := s + K^[i]^[q] * X^[i] ;
        dec( q )
      end ;
      F^[i] := s ;
                  (* Part 2 *)
      q := i - 1 ;
      for  j := 2  to  height^[i]  do begin
        F^[q] := F^[q] + K^[i]^[j] * X^[i] ;
        dec( q )
      end
    end (* for *) ;
  end  (* KtimesX *) ;
```

7. Using the KxR Program

7.1 Command line

```
kxr input output
```

7.2 Input file

The input is free format. Each entry separated by blank, tab, or <cr>. The maximum number of numbers on a line will usually depend on the operating system, and most likely, requires that a line be less than 256 characters.

It is good practice to begin each kind of data on a new line.

7.2.1 Input File Format First, on a line by itself, the number of equations.

Second, the height array. This may use as many lines as necessary to completely describe the array.

Third, the terms of K, in order from the diagonal towards the top of each column.

Fourth, the terms of R, from first equation to last.

For example, consider the set of simultaneous equations

$$\begin{bmatrix} 5.0 & -4.0 & 1.0 & 0.0 \\ -4.0 & 6.0 & -4.0 & 1.0 \\ 1.0 & -4.0 & 6.0 & -4.0 \\ 0.0 & 1.0 & -4.0 & 5.0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{Bmatrix}$$

Pertinent data for this set:

— The number of equations *neq* is 4.

— The column heights are, respectively, 1, 2, 3, and 3.

— The terms of column 1: 5.0.

— The terms of column 2: 6.0, and -4.0.

— The terms of column 3: 6.0, -4.0, and 1.0

— The terms of column 4: 5.0, -4.0, and 1.0

— Finally, the terms of the right hand side are: 0.0, 1.0, 0.0, and 0.0

The input file for this case might therefore look like:

```
4
1      2      3      3
5.0
6.0    -4.0
6.0    -4.0   1.0
5.0    -4.0   1.0
0.0    1.0    0.0    0.0
```

7.3 Output file

The output file contains the solution vector, one term per line. Since the solution vector will most likely be used by another program, there is no label or "pretty-print" information written to this file.

To use the solution vector **R** in another program:

1. Allocate storage for *neq* floats, to hold **R**.
2. Open the file access to the *text* file containing **R**. Turbo Pascal statements would have the form

```
var
   f : text ;
...
assign( f, 'solution.dat' ) ;
reset( f ) ;
```

3. Read **R** with statements of the form

```
for i := 1 to neq do
   read( f, R^[i] ) ;
```

8. Acknowledgements

The author sincerely thanks Professor Theodore Toridis for his careful reading of, and his helpful comments on, the text.

9. References

Bathe	Bathe, Klaus-Jürgen, and Edward L. Wilson, **Numerical Methods in Finite Element Analysis,** Prentice-Hall, 1976.
BorlandD	—, **Turbo Debugger,** Version 1.0, Borland International, 1988.
BorlandPR	—, **Turbo Pascal Reference Guide,** Version 5.0, Borland International, 1988.
BorlandPU	—, **Turbo Pascal User's Guide,** Version 5.0, Borland International, 1988.
Carnahan	Carnahan, Brice, H.A. Luther, & James O. Wilkes, **Applied Numerical Methods,** Wiley, 1969.
Jensen	Jensen, Kathleen and Niklaus Wirth, **Pascal User Manual and Report,** Springer-Verlag, 1974.
Ralston	Ralston, Anthony, **A First Course in Numerical Analysis,** McGraw-Hill, 1965.
Wilson	Wilson, Edward L., *et. al*, "Direct Solution of Large Systems of Linear Equations," *Computers & Structures,* Vol. 4, pp. 363-372, Pergamon Press, 1974.
Wirth	Wirth, Niklaus, **Systematic Programming: An Introduction,** Prentice-Hall, 1973.

APPENDIX

The following pages contain the complete source code for KxR.

APPENDIX - Source code for KxR

```
**********  float.inc  ************************************

TYPE
   float = double ;    (* change to suit accuracy *)


**********  globals.pas  **********************************

Unit globals ;

(*. globals - where all the global variables are kept. *)


Interface              (* public declarations *)

{$I float.inc }        (* defines the numerical precision *)

CONST
   MAXEQNS = 65520 div Sizeof( float ) ;

TYPE
   column_height_array = array [ 1 .. MAXEQNS ] of word ;
   column_height_type  = ^column_height_array ;

   column_type         = array [ 1 .. MAXEQNS ] of float ;
   column_ptr          = ^column_type ;

   column_array        = array [ 1 .. MAXEQNS ]  of column_ptr ;
   matrix              = ^column_array ;

   matrix_file         = file of float ;

   item = record case boolean of
                true : ( w : word ) ;
                false: ( f : float )
             end (* item *) ;

VAR

   column : column_ptr ;          (* pointer to the K column *)

   f : text ;           (* input file *)
   g : text ;           (* output file *)

   height : column_height_type ;

   K      : matrix ;              (* matrix of coefficients *)

   neq    : word ;                (* # of equations, this problem *)

   mfile  : matrix_file ;
   mfile_name : string[12] ;

   origin : pointer ;             (* marks the start of storage *)

   pre_dmax,  pre_dmin  : float ;
   post_dmax, post_dmin : float ;

   R : column_ptr ;               (* "Right hand side" of equation *)
   sizeof_float : longint ;

   X : column_ptr ;                (* used to check accuracy *)


(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
Implementation                     (* private declarations *)

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
begin (* globals initialization code *)

   height := nil ;
   K      := nil ;
   R      := nil ;

end   (* globals *)




**********  kxr.pas  ***************************************

{$O+ }
program KxR ;

(*****
 *
 *
     KxR - solve the matrix equation [K]{x} = {R}
           where [K] is symmetric and stored in
           skyline form per Bathe/Wilson.

     Copyright (c) Peter N Roth - January 1989.
 *
 *
 *****)


Uses overlay
   , globals
   , kxrutil
   , init
   , calc
   , fini
   ;

{$O kxrutil }
{$O init }
{$O calc }
{$O fini }

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
begin
   ovrinit( 'kxr.ovr' ) ;
   initialize ;         (* unit  INIT *)
   calculate ;          (* unit  CALC *)
   finish               (* unit  FINI *)
end   (* KxR *).
```

```
**********  init.pas  ***************************************

Unit init ;

(*. init -  set up the machine, read input, etc. *)

Interface              (* public declarations *)

Uses overlay
   , globals
   , kxrutil
   ;

{$O kxrutil }

procedure initialize ;

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
Implementation                      (* private declarations *)

   VAR
      m1, m2 : longint ;(* temporary variables for storage calculations *)


(*****
 *
 *.   usage - 'how to invoke the program.'
 *
 *)
procedure usage ;
   begin (* usage *)

      writeln( '*** usage: ', paramstr( 0 ), '  input  output ' )

   end   (* usage *) ;



(*****
 *
 *.   command_line - get and vet command line parameters.
 *
 *)
procedure command_line ;
   begin (* command_line *)

      if  paramcount <= 1  then begin
            usage ;
            halt( 1 )
      end ;

      if  paramstr( 1 ) = paramstr( 2 )  then begin
            usage ;
            writeln( '    Input and output file names must differ.' ) ;
            halt( 1 )
      end ;

      assign( f, paramstr( 1 ) ) ;
      reset( f ) ;
      if  eof( f )  then begin
            usage ;
            writeln( '    Immediate end of file on ', paramstr( 1 ) ) ;
            usage ;
            abort
        end ;

        assign( g, paramstr( 2 ) ) ;          (* will overwrite an existing file *)
        rewrite( g )

    end   (* command_line *) ;


(*****
 *
 *.   get_neq - read the number of equations from <f>.
 *
 *)
procedure get_neq ;
    begin (* get_neq *)

        readln( f, neq ) ;

        if  neq > MAXEQNS  then begin
            writeln( 'Number of equations exceeds current capacity (',
                         MAXEQNS, ') - get_neq.' ) ;
          halt  ( 1 )
        end
        else if  neq <= 0  then begin
            writeln( 'Number of equations <= 0  ???  Can''t solve. - get_neq.' ) ;
            abort
        end (* if *)

    end   (* get_neq *) ;


(*****
 *
 *.   get_heights -  read the heights of each column.
 *
 *)
procedure get_heights ;
    VAR
        i : word ;
    begin (* get_heights *)

        for  i := 1  to  neq  do begin
            read( f, height^[i] ) ;
            if  height^[i] <= 0  then begin
              writeln( 'Non-positive column height at entry ',i ) ;
              abort
            end (* if *)
        end (* for *)

    end   (* get_heights *) ;
```

```
(*****
 *
 *.   get_K -  read the coefficient matrix K.
 *
 *)
procedure get_K ;
   VAR
      i, j : word ;

   begin (* get_K *)

      for  i := 1  to  neq  do begin
            for  j := 1  to  height^[i]  do begin
               read( f, K^[i]^[j] )
            end
      end (* for *)

   end   (* get_K *) ;


(*****
 *
 *.   get_R -  read the right hand side R.
 *
 *)
procedure get_R ;
   VAR
      i : word ;
   begin (* get_R *)

      for  i := 1  to  neq  do
            read(f, R^[i] )

   end   (* get_R *) ;
```

```
(*****
 *
 *.   initialize - perform the initializations.
 *
 *)
procedure initialize ;
   begin (* initialize *)

      command_line ;              (* get & vet the command line *)

      m1 := memavail ;
      sizeof_float := sizeof( float ) ;

      mark( origin ) ;            (* where the heap begins *)

      get_neq ;

      allocate_height ;
      get_heights ;
                                  (*
                                   * allocate ALL space first, to ensure
                                   * that we can compute, THEN read
                                   * the stuff.
                                   *)
      allocate_K ;
      allocate_vector( R ) ;
      allocate_vector( X ) ;
                                  (*
                                   * It is not necessary to clear the arrays,
                                   * since we will re-initialize the arrays
                                   * via the READ.
                                   *)
      clear_K ;
      set_vector( R, neq, 0.0 ) ;
      set_vector( X, neq, 0.0 ) ;

      get_K ;
      get_R ;

      m2 := memavail ;
      write( 'Memory available (bytes): ', m1 ) ;
      writeln( ';    used: ', m1 - m2 ) ;

      assign( mfile, 'junk.mat' ) ;
      rewrite( mfile ) ;

   end   (* initialize *) ;

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)

end   (* init *).
```

```
**********  calc.pas  *************************************

Unit calc ;

(*. calc -  the meat of the crunching. *)

Interface              (* public declarations *)

Uses overlay
   , globals
   , kxrutil
   , solver
   ;

{$O kxrutil }
{$O solver}

procedure calculate ;

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
Implementation                     (* private declarations *)


(*****
 *
 *.   calculate - control the crunching.
 *
 *)
procedure calculate ;
   begin (* calculate *)

      diagonal_extremes( pre_dmax, pre_dmin ) ;

      active_column_solver ;                (* unit SOLVER *)

      diagonal_extremes( post_dmax, post_dmin ) ;

   end   (* calculate *) ;


(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)

end   (* calc *).
```

```
**********  fini.pas  **************************************

Unit fini ;

(*. fini -  clean up & go home. *)

Interface               (* public declarations *)

Uses overlay
   , globals
   , kxrutil
   ;

{$O kxrutil }

procedure finish;

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
Implementation                     (* private declarations *)


(*****
 *
 *.   write_R - put the solution vector to <g>.
 *
 *)
procedure write_R ;
VAR
   n : word ;
   begin (* write_R *)

      for  n := 1  to  neq  do
            writeln( g, R^[n] ) ;

   end   (* write_R *) ;


(*****
 *
 *.   finish - out of here.
 *
 *)
procedure finish ;
   begin (* finish *)

      write_R ;

      writeln( 'Diagonal extremes before decomposition:' ) ;
      writeln( pre_dmax:30, pre_dmin:30 ) ;

      writeln( 'Diagonal extremes after decomposition:' ) ;
      writeln( post_dmax:30, post_dmin:30 ) ;

      release( origin ) ;        (* memory back to system *)

      close( f ) ;
      close( g )
   end   (* finish *) ;(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)

end   (* fini *).
```

```
**********  solver.pas  *************************************

{$O+}
Unit solver ;

(*. solver. *)

Interface              (* public declarations *)

Uses overlay
   , globals
   , kxrutil
   ;



procedure active_column_solver ;
procedure decompose ;
procedure reduce_rhs ;
procedure back_substitute ;



(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
Implementation                     (* private declarations *)
```

```
(*****
 *
 *.   decompose - LDU decomposition of coefficient matrix.
 *
 *)
procedure decompose ;
   VAR
      b, c : float ;
      n, p, q, s, t, u, v : word ;

   begin (* decompose *)

      for  n := 1  to  neq  do begin
                                 (*
                                  * propagate effect of previous
                                  * off-diagonal terms to the current column
                                  *)
         if  height^[n] > 1  then begin
                                 (* p = first previous column of interest *)
            p := n - height^[n] + 2 ;
            for  q := height^[n] - 1  downto  2  do begin
               s := 2;
               t := q + 1 ;
                                 (* u = number of multiplies to perform *)
               u := min( height^[p] - 1,  height^[n] - q ) ;
               c := 0.0 ;
               for  v := 1  to  u  do begin
                  c := c + K^[p]^[s] * K^[n]^[t] ;
                     Inc( s ) ;
                     Inc( t )
               end (* v *) ;
               K^[n]^[q] := K^[n]^[q] - c ;
               Inc( p )              (* the next "previous column" *)
            end (* q *)
         end (* if *) ;
                                     (*
                                      * propagate effect of previous
                                      * diagonal terms to the current column
                                      *)
         p := n ;
         b := 0.0 ;
         for  q := 2  to  height^[n]  do begin
            Dec( p ) ;
            c := K^[n]^[q] / K^[p]^[1] ;  (* the "Lij" terms *)
            b := b + c * K^[n]^[q] ;
            K^[n]^[q] := c
         end (* q *) ;
         K^[n]^[1] := K^[n]^[1] - b ;

         if  K^[n]^[1] <= 0.0  then  solerr( n )

      end (* n *)

   end   (* decompose *) ;
```

```
(*****
 *
 *.   reduce_rhs - using factors computed by 'decompose'.
 *
 *)
procedure reduce_rhs ;
   VAR
      c : float ;
      n : word ;
      p : word ;
      q : word ;

   begin (* reduce_rhs *)

      for  n := 1  to  neq  do begin
         p := n - 1 ;              (* first "previous column" *)
         c := 0.0 ;
         for  q := 2  to  height^[n]  do begin
            c := c + K^[n]^[q] * R^[p] ;
            Dec( p )               (* next "previous column" *)
         end (* q *) ;
         R^[n] := R^[n] - c
      end (* n *) ;

   end   (* reduce_rhs *) ;


(*****
 *
 *.   back_substitute - complete solution of the equations.
 *
 *)
procedure back_substitute ;
VAR
   n : word ;
   p : word ;
   q : word ;

   begin (* back_substitute *)

      for  n := 1  to  neq  do
         R^[n] := R^[n] / K^[n]^[1] ;

      for  n := neq  downto  2  do begin
         p := n - 1 ;
         for  q := 2  to  height^[n]  do begin
            R^[p] := R^[p] - K^[n]^[q] * R^[n] ;
            Dec( p )
         end (* q *)
      end (* n *)

   end   (* back_substitute *) ;
```

```
(*****
 *
 *.   active_column_solver - Gaussian solution of simultaneous equations.
 *
 *)
procedure active_column_solver ;

   begin (* active_column_solver *)

      write( 'Decomposition     ...' ) ;
      decompose ;
      writeln( ' done.' );

      write( 'Reduction         ...' ) ;
      reduce_rhs ;
      writeln( ' done.' ) ;

      write( 'Back substitution ...' ) ;
      back_substitute ;
      writeln( ' done.' )

   end   (* active_column_solver *) ;
(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)

end   (* solver *).



**********  kxrutil.pas  ***********************************

{$O+}
Unit kxrutil ;

(*. kxrutil -  utility routines for the KxR package. *)

Interface             (* public declarations *)

Uses overlay
   , globals
   ;

procedure abort ;
procedure add_to_K ( term : float ;  at_row, at_col : word ) ;
procedure allocate_column ( i : integer ; r : word ) ;
procedure allocate_height ;
procedure allocate_K ;
procedure allocate_vector ( VAR R : column_ptr ) ;
procedure clear_K ;
function  diagonal_average ( VAR davg : float ) : float ;
procedure diagonal_extremes ( VAR dmax, dmin : float ) ;
procedure KtimesX ( F, X : column_ptr ) ;
function  min ( a, b : integer ) : integer ;
function  norm_K ( VAR vnorm : float ) : float ;
procedure print_K ;
procedure retrieve_K ( VAR h : matrix_file ) ;
procedure solerr ( n : integer ) ;
procedure set_vector ( V : column_ptr;   n : word;   W : float ) ;
procedure store_K    ( VAR h : matrix_file ) ;

(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)
Implementation                      (* private declarations *)
```

```
(*****
 *
 *.   abort - release memory and die.
 *
 *)
procedure abort ;
   begin (* abort *)

      release( origin ) ;
      halt( 1 )

   end   (* abort *) ;



(*****
 *
 *.   add_to_K - add 'term' at 'at_row,at_col'
 *                where at_row, at_col give the location in
 *                a "full" matrix.
 *
 *)
procedure add_to_K ( term : float ;  at_row, at_col : word ) ;
   VAR
      i : word ;
   begin (* add_to_K *)

      if  at_row <= at_col  then begin       (* K is upper triangle only *)
         i := Succ( at_col - at_row ) ;
         if  i <= height^[at_col]  then      (* insurance *)
            K^[at_col]^[i] := K^[at_col]^[i] + term
      end (* if *)

   end   (* add_to_K *) ;



(*****
 *
 *.   allocate_column - arrange storage for the i-th column.
 *
 *)
procedure allocate_column ( i : integer ; (* the column of interest *)
                            r : word      (* the size desired *)
                            ) ;
   VAR
      c : longint ;
   begin

      c := r * sizeof_float ;

      if  c > MemAvail  then begin
            writeln ( 'Out of memory attempting to allocate ', c,
                      ' bytes  to column ', i, ' - allocate_column.' ) ;
            abort
      end
      else begin
            Getmem ( K^[i], c ) ;
      end

   end (* allocate_column *) ;
```

```
(*****
 *
 *.   allocate_height - allocate storage for the column heights.
 *
 *)
procedure allocate_height ;
   VAR
      c : longint ;
   begin (* allocate_height *)

      c := neq * sizeof( word ) ;

      if  c > MemAvail  then begin
            writeln( 'Out of memory attempting to allocate ', c,
                     ' bytes  to column heights - allocate_height.' ) ;
            abort
      end
      else
            Getmem ( height, c )

   end   (* allocate_height *) ;


(*****
 *
 *.   allocate_K - allocate storage for the matrix of coefficients.
 *
 *)
procedure allocate_K ;

   VAR
      c : longint ;
      i : word ;

   begin (* allocate_K *)

      c := neq * sizeof( pointer ) ;

      if  c > MemAvail  then begin
            writeln ( 'Out of memory attempting to allocate ', c,
                      ' bytes to [K] - allocate_K.' ) ;
            abort
      end
      else
            Getmem ( K, c ) ;

      for  i := 1  to  neq  do
            allocate_column( i, height^[i] )

   end   (* allocate_K *) ;
```

```
(*****
 *
 *.   allocate_vector - allocate storage for a 'right-hand-side-like' vector.
 *
 *)
procedure allocate_vector ( VAR R : column_ptr ) ;

   VAR
      c : longint ;
   begin (* allocate_vector *)

      c := neq * sizeof_float ;

      if  c > MemAvail  then begin
            writeln ( 'Out of memory attempting to allocate ', c,
                      ' bytes to column vector - allocate_vector.' ) ;
            abort
      end
      else
            Getmem ( R, c )

   end   (* allocate_vector *) ;



(*****
 *
 *.   clear_K - set the contents of K to 0.0.
 *
 *)
procedure clear_K ;
   VAR
      i : word ;
   begin (* clear_K *)

      for  i := 1  to  neq  do
         set_vector( K^[i], height^[i], 0.0 )

   end   (* clear_K *) ;



(*****
 *
 *.   diagonal_average - return average of values on diagonal of K.
 *
 *)
function diagonal_average ( VAR davg : float ) : float ;
   VAR
      i : word ;
   begin (* diagonal_average *)

      davg := 0.0 ;
      for  i := 1  to  neq  do
         davg := davg + K^[i]^[1] ;
      davg := davg / neq ;
      diagonal_average := davg

   end   (* diagonal_average *) ;
```

```
(*****
 *
 *.   diagonal_extremes - return maximum and minimum values on diagonal of K.
 *
 *)
procedure diagonal_extremes ( VAR dmax, dmin : float ) ;
   VAR  i : word ;
   begin (* diagonal_extremes *)
      dmax := K^[1]^[1] ;
      dmin := K^[1]^[1] ;
      for  i := 1  to  neq  do begin
         if  K^[i]^[1] > dmax  then  dmax := K^[i]^[1] ;
         if  K^[i]^[1] < dmin  then  dmin := K^[i]^[1] ;
      end
   end   (* diagonal_extremes *) ;



(*****
 *
 *.   KtimesX - column vector F := skyline matrix K times column vector X.
 *              uses global variables 'neq', 'K' and 'height'
 *
 *    K is symmetric:
 *       k11         k44
 *           k21     k43
 *               k31 k42
 *        (sym)      k41
 *
 *    Part 1 uses each column from the diagonal to the maximum
 *    height in each column as a row.  This ensures that
 *    the terms to the left of the diagonal are used in the
 *    multiplication.
 *
 *    Part 2 adjusts the previous sums by using the terms above
 *    the diagonal as the remainder of the row.
 *
 *)
procedure KtimesX ( F, X : column_ptr ) ;
   VAR
      i, j, q : word ;
      s : float ;
   begin (* KtimesR *)
      for  i := 1  to  neq  do begin
                          (* Part 1 *)
         q := height^[i] ;
         s := 0.0 ;
         for  j := 1  to  height^[i]  do begin
            s := s + K^[i]^[q] * X^[i] ;
            dec( q )
         end ;
         F^[i]  := s ;
                          (* Part 2 *)
         q := i - 1 ;
         for  j := 2  to  height^[i]  do begin
            F^[q]  := F^[q] + K^[i]^[j] * X^[i] ;
            dec( q )
         end

      end (* for *) ;

   end   (* KtimesX *) ;
```

```
(*****
 *
 *.   min - returns smallest integer value.
 *
 *)
function min ( a, b : integer ) : integer ;
   begin (* min *)

      if  a <= b  then  min := a  else  min := b

   end   (* min *) ;


(*****
 *
 *.   norm_K - a norm of K.
 *
 *)
function norm_K ( VAR vnorm : float ) : float ;
   VAR
      i : word ;
   begin (* norm_K *)

      vnorm := K^[1]^[1] ;
      for  i := 1  to  neq  do
         if  K^[i]^[1] > vnorm  then  vnorm := K^[i]^[1] ;

      norm_K := vnorm

   end   (* norm_K *) ;


(*****
 *
 *.   print_K - print the coefficient matrix (for debugging purposes).
 *
 *)
procedure print_K ;
VAR
   n : word ;
   p : word ;
   begin (* print_K *)

      for  n := 1  to  neq  do begin
         for  p := 1  to  height^[n]  do begin
            write( K^[n]^[p] )
         end (* p *) ;
         writeln
      end (* n *)

   end   (* print_K *) ;
```

```
(*****
 *
 *.   retrieve_K - from <h>.
 *
 *)
procedure retrieve_K ( VAR h : matrix_file ) ;
   VAR
      i, j : word ;
      x : item ;
   begin (* retrieve_K *)

      read( h, x.f ) ;
      neq := x.w ;

      for  i := 1  to  neq  do begin
         read( h, x.f ) ;
         height^[i] := x.w
      end ;

      for  i := 1  to  neq  do
         for  j := 1  to  height^[i]  do
               read( h, K^[i]^[j] )

   end   (* retrieve_K *) ;


(*****
 *
 *.   set_vector - set  V^[1..n]  to  W.
 *
 *)
procedure set_vector ( V : column_ptr;   n : word;   W : float ) ;
   begin (* set_vector *)

      for  n := 1  to  n  do
         V^[n] := W

   end   (* set_vector *) ;


(*****
 *
 *.   solerr - prints error message and dies.
 *
 *)
procedure solerr ( n : integer ) ;
   begin (* solerr *)

      writeln( '*** Diagonal term in column ', n, ' = ', K^[n]^[1] ) ;
      writeln( '    Coefficient matrix is NOT positive definite. - solerr.' ) ;
      halt( 1 )

   end   (* solerr *) ;
```

```
(*****
 *
 *.   store_K - save K on <h>.
 *
 *)
procedure store_K ( VAR h : matrix_file ) ;
   VAR
      i, j : word ;
      x : item ;
   begin (* store_K *)

      x.w := neq ;  write( h, x.f ) ;

      for  i := 1  to  neq  do begin
         x.w := height^[i] ;
         write( h, x.f )
      end ;

      for  i := 1  to  neq  do
         for  j := 1  to  height^[i]  do
            write( h, K^[i]^[j] )

   end   (* store_K *) ;


(* ----- + ----- + ----- + ----- + ----- + ----- + ----- *)

end   (* kxrutil *).
```