THE DESIGN AND IMPLEMENTATION OF A SPARSE MATRIX PACKAGE†

by

Alan George

and

Joseph W.H. Liu

Research Report CS-77-21

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

June 1977

† Research supported in part by the Canadian National Research Council under grant A8111.

ABSTRACT

Software for solving sparse systems of linear equations typically involves fairly complicated data structures and storage management. In many cases the user of such software simply wants to solve a system of equations, and should not have to be concerned with the way this storage management is actually done, or the way the matrix components are actually stored. In this paper we describe a sparse matrix package which effectively insulates the user from these considerations, but which still allows the user to conveniently use the package in a variety of ways.

§1 Introduction

Computer subroutines for solving the dense linear equations problem involve conventional numerical data types such as one and two dimensional arrays of floating point numbers, which are already available in the programming languages normally used for numerical computation. The array storage for these subroutines is known as soon as the number of equations to be solved is prescribed, and the number of parameters to such procedures is usually quite modest, typically only four or five. Finally, the number of subroutines which are used to solve a problem of a particular type is seldom greater than two, and often only one. All these aspects of software for solving the dense linear equations problem make the intellectual overhead involved in learning to use the subroutines fairly small. The user is already familiar with the way the data is stored, the number of data items to keep in mind is modest, and the number of subroutines used is also small.

Unfortunately, almost none of the above is true about subroutines which implement algorithms for solving sparse linear systems. Most of the data structures used are sufficiently unconventional that they are not provided as data types in the language of implementation. The amount of storage required is usually unpredictable, and is not known until at least part of the overall sequence of subroutines has been executed. The overall method of solution is multiphase, with ample opportunity (and usually the necessity) for storage overlays. Finally, due to the complicated data structures employed, the subroutines tend to have long argument lists, most of which have little or no meaning to the user of the subroutine unless he cares to know how the data is stored.

In this article we describe the design and implementation of what we loosely refer to as a user interface for a Fortran sparse matrix package developed at the University of Waterloo. This interface is a layer of software between the user, who has a sparse linear system to solve, and the numerous subroutines which implement the various phases of the solution procedure (described in detail in section 2). This layer of software provides storage management services, insulates the user from the complicated data structures used by the subroutines, and provides a convenient means of communication between the user and the subroutines. It also provides sequencing control, so that subroutines are called in the correct order, and convenient checkpoint/restart facilities. Although we describe the interface only as it relates to solving sparse positive definite systems of equations, it will be obvious that the method used to implement the interface is applicable in a much wider context. The actual package handles both symmetric and unsymmetric problems, but it assumes that the matrix structure is symmetric, and that row and/or column interchanges are not required to maintain numerical stability.

§2 Problem Overview

In order to appreciate the need for an interface as suggested in the introduction, we review the important subtasks in the overall solution scheme, and identify the ways the package might be used. For definiteness, and to provide notation and a framework for subsequent discussion, suppose the sparse N by N,positive definite system of equations to be solved is

(2.1) $Ax = b.$

For some N by N permutation matrix P, Cholesky's method is applied to PAP^T, yielding the triangular factorization

(2.2) $PAP^T = LL^T,$

where L is lower triangular. Then the two triangular systems $Ly = Pb$ and $L^T(Px) = y$ are solved. Thus, instead of (2.1), the formally equivalent system

(2.3) $(PAP^T)(Px) = Pb$

is solved. The crucial practical point is that the reordered system can lead to enormous reductions in storage and/or computation compared to applying Cholesky's method to the original problem (2.1), (assuming of course, that sparsity is exploited).

It is natural to view the above procedure as consisting of four distinct phases:

Step 1 (Order) Find a "good" ordering (permutation P) for A.

Step 2 (Data structure set-up) Determine the location of the nonzeros in L, where $PAP^T = LL^T$, and set up the appropriate data structures.

Step 3 (Factor) Factor PAP^T into LL^T.

Step 4 (Triangular solution) Solve $Ly = Pb$ and $L^Tz = y$, and then set $x = P^Tz$.

Our situation is that we possess a collection of subroutines for finding orderings, determining the corresponding structure of L, setting up various data structures, and performing the actual numerical computation. That is, we have several computer implementations for each of the phases above; for purposes of discussion we will label these program modules ORDERi, SET-UPi, FACTORi, and SOLVEi, $i = 1,2,\ldots,m$, where m is the number of distinct order/set-up/factor/ solve combinations we have available. We refer to each combination collectively, as a method or strategy. In specific circumstances, each method has advantages over the others; however, note that we are not concerned here with the development of these various ordering algorithms, storage schemes, etc. Our objective is to package this software so that it can be easily used by those who have problems to solve, but who do not have the time or the inclination to learn any details about how it is done.

In order to make this packaging effective, we must identify how these modules might be used, assuming a user was prepared to learn the appropriate calling sequences and the exact nature of their parameters. The most obvious way would be to choose method i, and call the sequence of subroutines ORDERi, SET-UPi, FACTORi, and SOLVEi. However, different methods will in general require different storage and computation, and the best method to use depends on the size and origin of the problem. In principle, these quantities are known at the end of step 2, so the user could execute

ORDERi and SET-UPi for several different i, choose the most attractive method k, and then execute FACTORk and SOLVEk. Ideally, to avoid re-executing ORDERk and SET-UPk, the results from the currently best SET-UPi should be saved until k is determined.

When the user has selected a method i and executed SET-UPi, he may use FACTORi and SOLVEi in several ways other than the normal case where he has a single problem to solve. In some contexts many problems having the same zero-nonzero structure must be solved. Since ORDERi and SET-UPi depend only on the <u>structure</u> of A and not its numerical values, they do not have to be executed more than once, and if FACTORi and SOLVEi are to be executed over an extended time period, the output from SET-UPi should be saved so that execution can begin with FACTORi for each new numerical problem. In still other situations many problems which differ only in their right hand sides must be solved, again sometimes over an extended time period. In this case only SOLVEi need be used repeatedly, and the output of FACTORi may be saved and recovered for each new right side.

The above discussion identifies four modes in which a user might use the software modules which implement steps 1-4 above. In addition to the normal case where four modules corresponding to one method are called in sequence, additional modules may be called, and/or some may be repeatedly called, in order to a) select the most desirable method, b) solve many problems having the same structure, or c) solve many problems differing only in their right hand side.

We have now discussed some important problems a user could deal with, given the software modules and the patience to learn how to call each of them. Our claim is that this investment in time is likely to be high. As we noted earlier, the elaborate data structures usually mean that the argument lists are lengthy. In order to use the modules efficiently, the user must determine which arrays must be preserved as input to the next module, and which ones can be destroyed (reused) by that module. In all but the standard mode of usage discussed above, the user must be aware of all information that needs to be preserved if the computation is to be restarted at a later time. Finally, perhaps the most persuasive argument for insulating the user from all these considerations is that storage requirements for some of the sparse matrix subroutines is often unpredictable. The user must guess how large some arrays must be declared; if he guesses low, the module will not execute successfully, and if he guesses high, some storage will be wasted.

§3 Design Considerations

3.0 Introduction

In the previous section we identified what could be loosely described as the functional capabilities that our sparse matrix package should possess. In this section we descend to the operational level and examine the actual components of the interface, from the user's view. Although we defer most of the discussion of the actual implementation of the interface until section 4, it is helpful in this section to be aware of the basic approach. Briefly, the user of the package supplies a single one dimensional array S along with its declared size MAXS. All array storage is allocated from S by the interface, and the origins of these arrays are transmitted from module to module in a COMMON area. Other information about the data structures along with control information is also passed from module to module through a COMMON block.

We now describe the interface components. We first augment the list of steps discussed in section 2 to include the points of information exchange between the user and the software modules discussed in the previous sections. We then discuss the responsibilities, desirable features, and possible abnormal completions of the implementation of each step in this augmented list.

Step 1 (Initialization)

Step 2 (Input nonzero structure) Input the (i,j) pairs for which $A_{ij} \neq 0$.

Step 3 (Order) Find a "good" ordering for A (i.e., find P).

Step 4 (Data structure set-up) Determine the location of the nonzeros in L and set up the appropriate data structures.

Step 5 (Input A) Input the numerical values for A_{ij}.

Step 6 (Factor) Factor PAP^T into LL^T.

Step 7 (Input b) Input the numerical values for b.

Step 8 (Triangular solution) Solve $Ly = Pb$ and $L^Tz = y$, and then set $x = P^Tz$.

The discussion in section 2 concerning the various modes in which the package might be used strongly suggest the need for a save/restart facility. The possible abnormal terminations discussed in the following subsections also illustrate the need for such a capability. Thus, we include the following two steps, which (loosely speaking) can be inserted anywhere in the list above.

Step S (Save) Save the current results on a specified input/output unit.

Step R (Restart) Read results written by execution of Step S from a specified input/output unit.

3.1 Initialization Step

This module is called before any part of the package is used. Its role is to initialize certain variables, to set default values for options, and perform some installation dependent functions such as setting the logical unit number for the printer, initializing the timing routine,etc. The Fortran statement is simply

```
CALL INIT(S)
```

where S is the working storage array declared by the user for use by the package.

3.2 Input of the Matrix Structure

This step represents the first serious communication with the package. Our algorithms for finding orderings represent the matrix structure as a symmetric graph in the array pair (XADJ, ADJNCY); the nodes adjacent to node i (columns subscripts of the nonzeros in row i of A) are stored in ADJNCY(k), for k = XADJ(i), XADJ(i) + 1,...,XADJ(i+1) - 1. However, it is unlikely (or at least not at all clear) that the user will have this representation available. In many situations the (i,j) pairs for which $A_{ij} \neq 0$ will become available to the user in a more or less arbitrary order, and thus the values of XADJ cannot be determined until the entire structure of A is known. Thus, the user would like to be able to communicate the (i,j) pairs in any order he chooses. Our package records this information, eliminating duplications, and when all pairs have been communicated, the appropriate internal representation is generated. The method used to communicate the fact that $A_{ij} \neq 0$ is the simple Fortran statement CALL INIJ(I,J,S). When all pairs have been input, the statement CALL IJEND(S) is executed to carry out the transformation to the internal (XADJ, ADJNCY) format. Note that the user is insulated from all the various internal data structures used in the recording and manipulation of the information. The parameter S in the subroutine references is again the working storage array.

Recall from our discussion in section 2 that a user may wish to investigate the relative merits of several different methods, which implies that we should be able to restart the computation at the beginning of the ordering phase, and avoid re-inputting the matrix structure. To do this we execute the Fortran statement

```
CALL SAVE(K,S)
```

where K is the Fortran logical unit on which the output of IJEND is to be written, along with other information needed to restart the computation at this point. If execution is then terminated, the state of the computation can be re-established by executing the two statements

```
CALL INIT(S)
CALL RESTRT(K,S).
```

If a user wishes to return to this state of the computation during the same execution, the subroutine INIT does not have to be executed.

§3.3 Finding the Ordering and Setting Up the Data Structures

For our class of positive definite problems, there are important reasons for performing the ordering and data structure set-up tasks serially, in separate subroutines. An important reason is that temporary storage used by the ordering subroutine can be reused by the data structure set-up subroutine. However, from the user's view, there seems little reason to segregate the two steps. The output from the ordering routine is in itself of little interest to the user, since it is simply a permutation vector (and for some methods a small amount of partitioning information). The user is really interested in the implications, in terms of storage and computation, of using the ordering, and these are only known after the analysis of the structure of L has been performed. Thus, in our package the user invokes the execution of steps 3 and 4 (ordering and data structure set-up) by executing the Fortran statement

```
        CALL ORDERi(S)
```

where i is a numerical digit indicating the method (order/set-up/factor/solve sequence) to be used.

What can go wrong? First, there may not be enough storage in S to execute the ordering algorithm. In this case the user can execute SAVE (if he has not already done so at the end of the previous step) and after declaring a larger S, he can execute RESTRT and call ORDERi again. The output of the unsuccessful execution of ORDERi tells the user how large S must be to execute the ordering subroutine. The same SAVE/RESTRT strategy can be employed if the ordering algorithm aborts during execution. The ordering subroutines currently in the package do not terminate abnormally as a result of exceeding the storage provided, since they all use a fixed predictable amount. However, some implementations of ordering algorithms do require unpredictable amounts of storage, and some of these might be included in the package later.

When the ordering is obtained, the appropriate subroutine is called from ORDERi to determine the structure of L and set up its data structure. A disagreeable but inevitable characteristic of many of these subroutines is that their storage requirements are unpredictable, because the number of data structure pointers etc. is not known until the structure of L has been fully determined. There may be enough storage available to execute the subroutine and thereby determine the storage <u>needed</u> for the data structure even though the data structure itself cannot be saved.

Thus, the interface module ORDERi may terminate in several distinctly different ways:

a) there was not enough storage to execute the ordering subroutine.

b) the ordering was successfully obtained, but there was insufficient storage to even initiate execution of the data structure set-up subroutine.

c) the data structure set-up subroutine was executed, and the storage required for the data structure pointers etc. was determined, but there was insufficient storage for those pointers.

d) the data structure was successfully generated, but there is insufficient storage for the actual numerical values, so the next step cannot be executed.

e) ORDERi was successfully executed, and there is sufficient storage to proceed to the next step.

If any of the above conditions occurs, the user may execute SAVE, and reinitiate the computation after adjusting his storage declarations (either up or down) and executing RESTRT. If a) or b) occurs, information is supplied indicating the minimum value of MAXS needed so that c) will occur upon re-execution. If c) occurs, the minimum value of MAXS needed for d) and e) is provided.

When c) occurs, after executing SAVE, adjusting our storage declarations, and executing RESTRT, we must again call ORDERi. However, the interface will detect that the ordering has already been found, and will skip that part of the computation. Note that if the user is simply using the package to select a particular method, c) may be an acceptable termination state.

§3.4 Input of the Numerical Values for A and b

After having successfully set up the data structure for L, and determined that enough storage for the numerical values is available, the user may now input the actual numerical values for A and b. The position of step 7 in the sequence of steps in section 3.0 is arbitrary; the only restriction is that numerical values for b should be input after step 4 has been executed, and before step 8 is executed. Numbers can be transmitted by subroutine calls of the form

```
CALL INRHS(I,VALUE,S)
```

where I refers to the subscript of the original given ordering, and not Pb. Similarly, input of the numerical values of A is achieved by repeated subroutine calls of the form

```
CALL INAIJi(I,J,VALUE,S)
```

where again I and J refer to the subscripts of the unpermuted A, and VALUE is the numerical value of A_{ij}. Thus, the user is insulated from the fact that the problem, as he knows it, has been permuted. Note that there is a different matrix input subroutine for each method, because the data structures used are different. However, the parameter lists for all the methods are the same, and the subroutine names are the same except for the last digit which distinguishes the method.

In some situations, such as in certain finite element applications, the values of A_{ij} and b_i are obtained in an incremental fashion. That is, A_{ij} may be equal to VALUE1 + VALUE2, with VALUE1 and VALUE2 being computed at different steps in the user's program, which is utilizing the sparse matrix package. For this reason, INAIJi simply adds VALUE to the appropriate current value of A_{ij} in storage rather than making an assignment; we can then handle such incremental calculation of numerical values. The same remarks apply to treatment of the right hand side. Note that this strategy implies that the storage used for L and b must be initially set to zero before numerical values of A and b are supplied. This initialization is performed automatically by the interface during the first calls to INAIJi and INRHS, through the use of a "state variable" called STAGE, discussed in section 4.24.

Our package has no provisions for explicitly storing the nonzero components of A in compact form, which implies that the position of step 5 in the sequence of tasks in section 3.0 is significant. The nonzero components of A supplied by the user are placed directly into the data structure for L, and are overwritten by L during the factorization of the matrix. Thus, the numerical values of A and b cannot be accepted by the package until the determination of the structure of L has been completed. The advantage of this approach is that it conserves storage; storage used for the ordering and data structure set-up can be reused to store the numerical values. Moreover, once the permutation P and the data structures are determined, the matrix structure of A is not needed by the package, and can be (and is) discarded. Finally, our experience is that in many applications the structure of A is known much earlier than its numerical values anyway.

This decision to put the numerical values of A directly into the space to be occupied by L has a disadvantage as well. Obviously, if the (i,j) pairs for which $A_{ij} \neq 0$ and the numerical values A_{ij} are naturally available at the same time, the user must save the numerical values until after ORDERi has been successfully executed. Our advice is to write out the (I,J,A_{ij}) triples on an auxiliary file at the same time INIJ(I,J,S) is called, and then later read the file and insert the numerical values using

INAIJi. The right hand side b can be handled similarly if it is inconvenient to compute it when it is needed.

§3.5 The numerical computation

The algorithm used to perform the numerical computation is the standard Cholesky method. The actual implementations obviously vary across the methods, since different data structures are involved. However, this again is a fact that should not concern the user.

Up to this point we have distinguished between the factorization and solution steps, but in the actual interface both steps are initiated by the single Fortran statement

```
CALL SOLVEi(S)
```

where S is the working storage array for the package, provided by the user.

It turns out that enough information can be retained by the interface to allow the user to handle the various possible situations discussed in section 2 (multiple problems having a common structure, multiple right hand sides etc.). Again through the use of the state variable STAGE, discussed in the next section, the interface can detect upon entry to SOLVEi whether the factorization of the matrix has already been performed, and bypass executing that part of the module.

§4 Implementation and Features

The crucial feature of this sparse matrix package is the layer of user interface routines, which relieves the user from the tedious and error prone storage management task. As far as the user is concerned, he needs only to know the interface modules, which are extremely easy to use. The skeleton program below illustrates how simple it is to use the package. The labelled COMMON block USER is discussed in section 4.1.

```
      COMMON /USER/ MSGLVL, IERR, MAXS
      REAL  S(10000)
      MAXS = 10000
      CALL INIT(S)
    { Input of adjacency pairs by repeated use of }
    { CALL INIJ(I,J,S)                            }

      CALL IJEND(3,S)
      CALL ORDERi(S)

    { Input of the matrix nonzeros by repeated use of }
    { CALL INAIJi(I,J, AIJ, S)                        }
    { Input of the right hand side by repeated use of }
    { CALL INRHS(I, BI, S)                            }

      CALL SOLVEi(S)
    {Solution is now in the first N locations of S}
      STOP
      END
```

In the previous two sections, we have discussed the functional capabilities and the design objectives of the package. We now examine the implementation details of the interface. Figure 4.1 shows that the user is completely insulated from all the sparse matrix routines; communication is made possible via a handful of interface modules. In other words, the interface serves as a bridge between the user and the set of sparse matrix routines and at the same time it provides communication among the matrix routines. Communication is done through common blocks, most of which need not even be considered by the user. We shall discuss them in detail below.

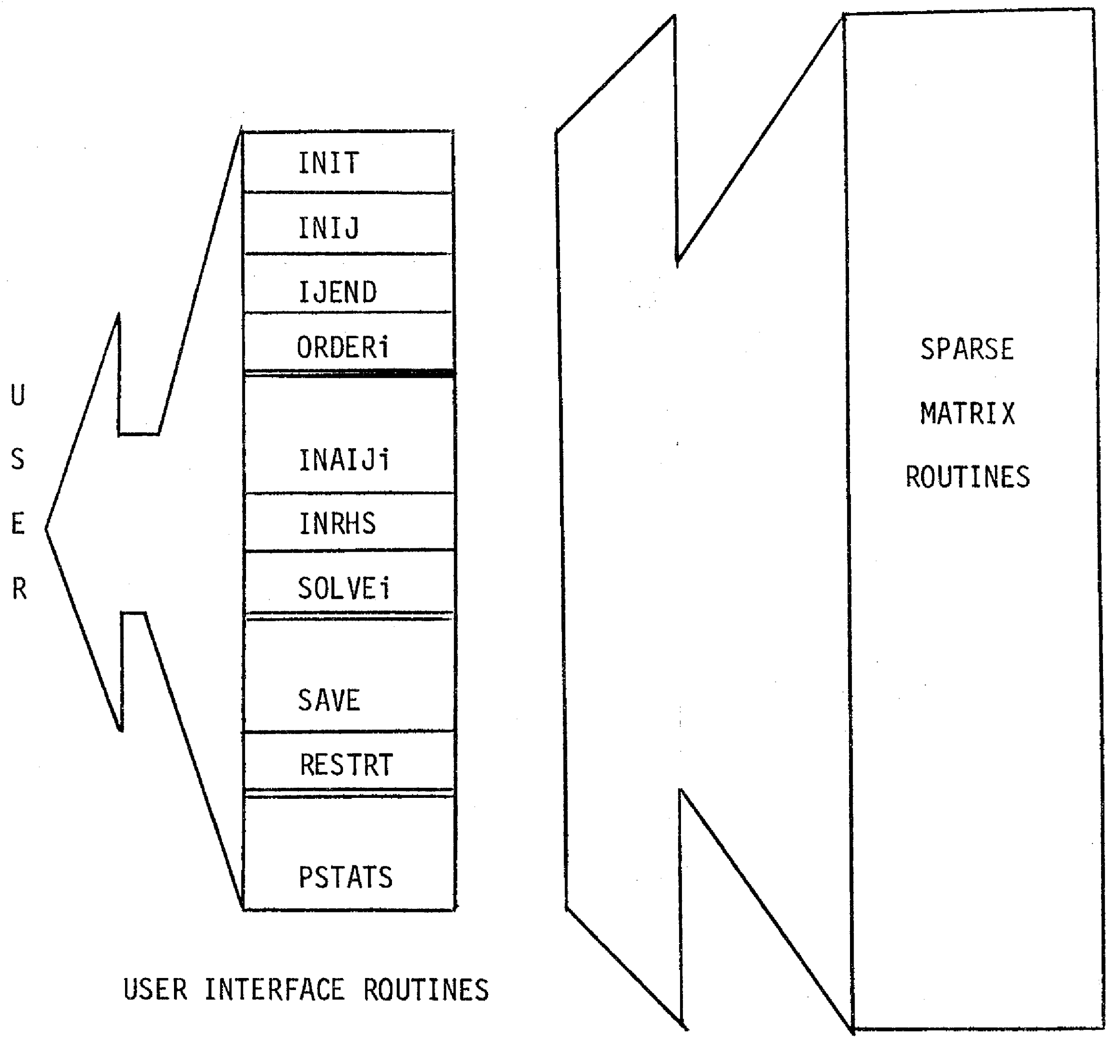

Figure 4.1

§4.1 User/module Communication

As noted earlier in section 3, the user supplies a one dimensional real array S, from which all array storage is allocated. In particular, the interface allocates the first N storage locations in S for the solution vector of the linear system. After all the interface modules for a particular method have been successfully executed, the user can retrieve the solution from these N locations.

There is one labelled COMMON block that the user must provide, having three variables:

COMMON /USER/ MSGLVL, IERR, MAXS.

The variable MAXS is the declared size of the one dimensional array S and it must be set by the user at the beginning of his program. For each module in the interface that allocates storage (e.g. INIJ, IJEND, ORDERi), MAXS is used to make sure there is enough storage to carry out the particular phase.

When a fatal error is detected, so that the computation cannot proceed, a positive code is assigned to IERR. The user can simply check the value of IERR to see if the execution of an interface module has been successful. This error flag can be used in conjunction with the save/restart feature to retain the results of successfully completed parts of the computation, as shown by the program fragment below.

```
                .
                .
                .
        CALL ØRDERi(S)
        IF (IERR.EQ.0) GØ TØ 100
        CALL SAVE(3,S)
        STØP
100     CØNTINUE
                .
                .
                .
```

In case an error is found in ORDERi, unit 3 will be used to save the relevant data in the storage array. The contents of the data saved could be the adjacency structure of the matrix, the ordering, or the ordering together with the data structure (depending on what went wrong, as discussed in section 3.3).

The first variable MSGLVL in /USER/ stands for "message level", and governs the amount of information printed by the interface modules. Its default value is two, and for this value a relatively small amount of summary information is printed, indicating the completion of each phase and the values of some important numbers, such as the amount of storage used by each module. When MSGLVL is set to one by the user, only fatal error messages are printed; this option could be useful if the package is being used in the "inner loop" of a large computation, where even summary information would generate excessive output. Increasing the value of MSGLVL (up to 4) provides increasingly detailed information about the computation.

In many circumstances, our package will be imbedded in still another "super" package which models phenomena which produce sparse matrix problems. Messages printed by our package may be useless or even confusing to the ultimate users of this super package, or the super package may wish to field the error conditions and perhaps take some corrective action which makes the error messages erroneous. (See section 4.2.2 for an example.) Thus, <u>all</u> printing by the package can be unhibited by setting MSGLVL to zero.

If all phases of a method execute successfully, in addition to the solution vector, the user may want to obtain statistics of the particular run. In view of this, the package provides a COMMON block for statistics:

```
COMMON /STATS/ ORDTIM, ALOCTM, FCTIME, SLVTIM, FCTOPS, SLVOPS,
                     ORDSTR, ALOSTR, SLVSTR, OVERHD
```

where ORDTIM - the time used to find the ordering.

ALOCTM - the time used for data structure set-up.

FCTIME - the time used for the factorization step.

SLVTIM - the time used for the triangular solution step.

FCTOPS - number of operations required by the factorization step.

SLVOPS - number of operations required by the triangular solution.

ORDSTR - the storage used for the ordering subroutine.

ALOSTR - the storage used for the data structure of the permuted system.

SLVSTR - the storage used by the SOLVEi module.

OVERHD - the overhead storage for the problem.

However, the user does not have to know anything about /STATS/. All he needs to supply is the statement

```
CALL PSTATS
```

at the end of his run to get the required information.

If the package is used as described in section 2 to select a method, PSTATS could be called after executing ORDERi for each i, thus providing storage information for each method. Of course, the user could also obtain the storage information during execution by including the STATS common declaration in his program and examining the appropriate variables.

In order to supply timing information, the package assumes the existence of a real function DTIME which returns the processor execution time that has elapsed since DTIME was last referenced. Thus, a DTIME function must be supplied for each installation of the package.

§4.2 Module/module Communication

There are two labelled COMMON blocks used for communication among modules within the interface. They are the control block and the storage map block:

```
/CNTROL/  STAGE, MXUSED, MXREQD, NEQNS, NEDGES, METHOD,
          {other method-related control variables}

/SMAP/    PERM, INVP, RHS,
          {data structure pointers}.
```

The /CNTROL/ block has ten integer variables and contains control information about the specific run. There are fifteen variables in the /SMAP/ block and they form a storage map of the array S.

4.2.1 <u>Locations of Storage Arrays</u>

Since storage management is the responsibility of the interface, it must be able to tell the various modules where data should be stored or has been stored. The fifteen variables in the /SMAP/ block are used to keep the locations (origins in S) of the various arrays used in the particular storage scheme. These storage schemes differ in complexity across the methods, so the same /SMAP/ block must be used in the corresponding routines ORDERi, INAIJi and SOLVEi.

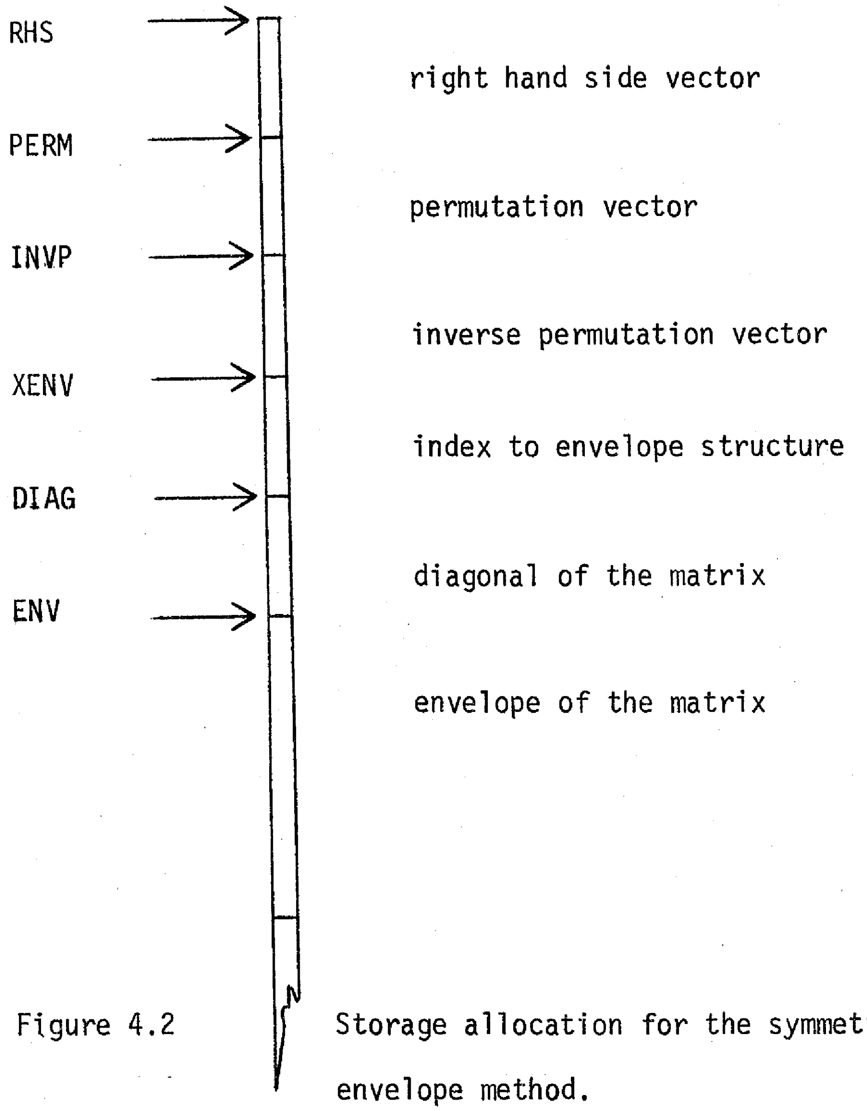

Figure 4.2 Storage allocation for the symmetric envelope method.

In figure 4.2, there is an example of the storage allocation for the symmetric envelope method [3]. Since three vectors are sufficient for the matrix structure, the corresponding /SMAP/ could be:

```
/SMAP/  PERM, INVP, RHS, DIAG, XENV, ENV, IPAD(9)
```

4.2.2 Save/restart Implementation

The SAVE routine saves the control information in/CNTROL/, the storage pointers in /SMAP/, as well as the storage vector S. In this way, the state of the computation can be re-established by executing RESTRT, which restores the /CNTROL/ and /SMAP/ blocks, and the vector S.

The variable MXUSED in/CNTROL/is used to avoid saving irrelevant data from S. After the successful completion of each phase, MXUSED is set to the maximum number of storage locations used thusfar. It is then only necessary to save the first MXUSED locations of S whenever the routine SAVE is called.

Some operating systems allow a program to change the space it occupies in main storage during execution. Thus, in some installations the user of our package might be able to dynamically increase or decrease the size of the working storage S. He can determine what the value of MAXS should be by declaring the common block CNTROL in his mainline program, and examining the value of MXREQD. At the end of each successfully executed phase of the computation, MXREQD is set to the minimum value of MAXS required to successfully execute the next phase of the computation.

It is often the case that when this dynamic growing of program space is provided, the effect is to increase the space allocated to unlabelled COMMON, which is usually assigned the highest memory locations in the user's program area. In such a circumstance the array S in the user's program would have to be declared in blank common.

In this connection, we might have asked the user of our package to always declare S in a blank or labelled common block, and consequently avoided having the parameter S appearing in all our interface modules. While there were advantages and disadvantages to this, we felt that in balance our current decision allowed the user somewhat more flexibility. In some applications the array S which the user passes to the sparse matrix package may actually be a segment of the user's own working storage arrays, and MAXS is simply the amount of that array left over by his own program's computation. It is sometimes inconvenient to arrange that the storage made available to the package be in blank common, or an appropriately labelled common block.

4.2.3 Method Checking

As we discussed in section 2, using a particular "method" means calling the appropriate interface routines ORDERi, INAIJi, and SOLVEi, where the last character is a numerical digit denoting the method. These ordering, input, and solve modules cannot be mixed since they in general involve different data structures for L. In order to ensure that these modules are not inadvertently mixed by the user, ORDERi sets the variable METHOD equal to i, and this variable is checked by subsequently executed modules INAIJi and SOLVEi.

4.2.4 Stage(Sequence)Checking

Another control variable that deserves special consideration is the STAGE code. As its name implies, it is used to keep track of the current step or stage of the execution. This variable is particularly important in connection with the save/restart feature. In restarting the system using the RESTRT routine, the STAGE code in/CNTROL/is restored, and it indicates the last successfully completed stage before the routine SAVE was called. In this way, the execution can be restarted without repeating already successfully completed steps.

Another function of this variable is to enforce the correct execution sequence of the various interface routines. Before the actual execution of each routine, the STAGE code is used to check that all previous modules have been successfully completed. This avoids producing erroneous results due to improper processing sequences, or accidental omission of steps.

The content of the variable STAGE is only changed after a phase has been successfully executed. When an error occurs during the execution of the phase, the STAGE code remains unchanged. This prevents the execution of all the subsequent phases, even if they are invoked by the user. As mentioned in sections 3.4 and 3.5, STAGE is also used by the modules to determine whether some initialization is necessary in a module, or whether part of the module has already successfully executed during a previous call to it.

4.2.5 Storage Allocation of Integers and Floating Point Arrays

The ANSI Fortran standard specifies that the numbers of bits used to represent integers and floating point numbers are the same. However, many vendors provide the user with the option of specifying "short" integers, either explicitly in declarations such as "INTEGER*2", or via a parameter to the Fortran processor which automatically represents all integers using fewer bits than used for floating point numbers. Since a significant portion of the storage used in sparse matrix computation involves integer data for pointers, subscripts etc., it is desirable to try to exploit these short integer features when it makes sense to do so.

Our interface contains a variable RATIO, set in the module INIT, which specifies the ratio of the number of bits used for floating point numbers to the number used for integers. In our package floating point arrays are declared REAL, integer arrays are declared INTEGER, and RATIO is set to 1. However, if the size of the integer representation is halved, (either through changing the integer array declarations, or specifying a system parameter, or by some other mechanism), then the only change required is to set RATIO to 2 in INIT. The interface then uses RATIO to allocate only $\lceil p/\text{RATIO} \rceil$ elements of S for integer arrays of length p. Since a good portion of the real storage vector S is allocated for integer arrays, the provision for different lengths of integer and floating point numbers may lead to word boundary problems, when the integer arrays are passed to the sparse matrix routines. This is overcome by always starting each array at a boundary of a floating point number in S. In other words, the storage pointers in /SMAP/ are defined with respect to the storage vector S.

The variable RATIO would also be set to 2 if the floating point arrays were declared to be double precision. However, we assume that the declaration of S that the user makes in his program is of the same type as that used for the floating point computation, so the user's declaration of the working storage array would also have to be double precision. We also make the reasonable assumption that RATIO $\geq$ 1.

§5 Concluding Remarks

The numerous examples supplied in the Appendix serve to illustrate that the interface meets the functional requirements outlined in section 2. The error messages printed also illustrate how the use of the internal parameters METHOD and STAGE discussed in section 4 serve to protect the user from many of the potential blunders.

It is a relatively simple task to include an additional layer of software around the various ORDERi, INAIJi and SOLVEi modules, to provide an automatic method-selection feature in the package. The user would then call SELECT, INAIJ, and SOLVE, where SELECT would determine the value of the internal variable METHOD by calling each ORDERi, and INAIJ and SOLVE would call the appropriate INAIJi and SOLVEi modules according to the value of METHOD. Of course,the selection of the method could depend on various criteria, so SELECT might have some parameters.

There are advantages and disadvantages associated with this second layer of software. One advantage is that the user must remember even less about the package in order to use it, and will use the most efficient method (according to the criterion SELECT uses). There is a danger with the package as it stands in that a user may simply choose a method and not bother investigating other possibly more efficient ones included in the package. Obviously there is a tradeoff here that is difficult to quantify, since SELECT exacts a price which might offset any gains realized through choosing the best method. However, in cases where many problems having the same structure must be solved, it would appear that a strong case could be made for having a SELECT module in the package.

We regard the case for having general INAIJ and SOLVE modules as much weaker than that for having a SELECT module. The main disadvantage is that under most operating system environments, all the INAIJi and SOLVEi modules of the package would be loaded during execution of SOLVE, even though only one of each would be actually executed. This problem was solved in the EISPACK system [5] on IBM computer systems by using the execution time linking and loading features provided by OS/360. However, we reject this approach because not all operating systems provide such facilities. To summarize this point, we regard a SELECT module as desirable, and plan to include one in a future version of the package. However, we do not think the advantages of having a general INAIJ and SOLVE warrant their inclusion.

The use of the interface routines INIJ and INAIJi provides great flexibility in the input of the matrix structure and matrix components. However, for systems with large overhead in subroutine calls, the repeated use of INIJ and INAIJi can be expensive. In view of this, the package includes interface routines which allow the input of an entire row or subarray of nonzero subscripts and nonzero components. This can be useful when the structure and nonzeros are available in a more structured manner.

At the moment, all our sparse matrix software lies within the portable subset of Fortran specified by the PFORT verifier [4]. The same applies to the interface routines, with only one exception: array types are allowed to change across subroutines. We do not regard this as a serious violation, since it is tolerated and handled uniformly by most systems. As described in section 4.2.5, we have been careful to allocate storage for arrays in such a way as to avoid the alignment problems which sometimes occur, particularly on IBM 360/370 systems, when array types are mixed in this way.

One of our objectives in creating the interface was to reduce the time and effort required to use our sparse matrix software. We feel this has been achieved. The parameter lists for the various interface modules are short, and apart from the storage array S, they all mean something specific to the user and his problem. Furthermore, except for the last character in the module names (which distinguishes the method), the names of the modules are the same across the various methods. Thus, there are relatively few things for the user to remember.

§6 References

[1] W.S. Brown, "An operating environment for dynamic-recursive computer programming systems", Comm. A.C.M. 8 (1965), pp. 371-377.

[2] W.M. Gentleman and Alan George, "Sparse matrix software", in Sparse Matrix Computations, edited by J.R. Bunch and D.J. Rose, Academic Press, New York, 1976, pp. 243-261.

[3] Joseph W.H. Liu, "On reducing the profile of sparse symmetric matrices", Report CS-76-07, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada (February 1976).

[4] B.G. Ryder, "The PFORT verifier", Software Practise and Experience 4 (1974), pp. 359-377.

[5] B.T. Smith, J.M. Boyle, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler, Lecture Notes in Computer Science, Matrix Eigensystem Routines Eispack Guide, Springer Verlag, New York, 1974.

[6] USA Standard Fortran. ANS X3.9-1966, Amer. Nat. Standards Inst., New York, 1966.

APPENDIX

In this appendix we provide several mainline programs which illustrate how the package can be used in the various modes discussed in section 2. In order to keep the programs short, we utilize the subroutines GRID, GRIDA, GRIDB1, GRIDB2, GRIDE1 and GRIDE2, whose Fortran listings are given below. An approximate solution to the Poisson equation $u_{xx} + u_{yy} = f$ is computed on the domain $R = (0,1) \times (0,1)$, using the standard 5-point difference operator on a regular n by n grid, with grid points (ih,jh), $1 \le i \le n$, $h = 1/(n+1)$. The subroutine GRID uses the sparse matrix package routines INIT, INIJ, and IJEND to set up the structure of the coefficient matrix A. The subroutine GRIDA uses the sparse matrix module INAIJ1 (since we use ORDER1 in all our examples) to insert the numbers for A into the package. The subroutine GRIDB1 uses INRHS to insert the appropriate right hand side b so that the solution to the continuous problem is $u(x,y) = e^{(x+y)}$, and GRIDE1 computes the error in the computed (discrete) solution. The subroutines GRIDB2 and GRIDE2 are analogous, except the solution to the problem they treat is $u(x,y) = \sin(x-y)$.

The programs were run under the WATFIV debugging compiler, developed at the University of Waterloo, on an IBM 360/75 computer. Since the code generated by the compiler has extensive error checking overhead, the execution times are quite large. All times reported are in seconds.

```
C***********************************************************
C***********************************************************
C****** GRID ... GENERATES AN N BY N GRID STRUCTURE ********
C******          BY REPEATED CALLS TO INIJ.
C***********************************************************
C***********************************************************
C
C       INPUT PARAMETERS-
C                N - SIZE OF THE GRID (IMPLIES THAT THE
C                    NUMBER OF EQUATIONS IS N**2 )
C                S - WORKING STORAGE VECTOR
C
C***********************************************************
C
      SUBROUTINE GRID( N, S )
C
C***********************************************************
      REAL    S(1)
      INTEGER SUB, I, J, IJ, N
C***********************************************************
C
C         SUB MAPS THE COORDINATES (I,J) OF THE GRID ONTO
C         THE INTEGERS 1,2,...,N**2. THE SUBSCRIPTS
C         GENERATED BY SUB CORRESPOND TO THE STANDARD ROW
C         BY ROW ORDERING OF THE GRID.
C         ---------------------------------------------------
          SUB(II, JJ) = (II - 1)*N + JJ
C-----------------------------------------------------------
C
          CALL INIT ( S )
C
          DO 200 I = 1, N
             DO 100 J = 1, N
                IJ = SUB(I,J)
                IF ( I .NE. 1 ) CALL INIJ( IJ, SUB(I-1, J), S)
                IF ( J .NE. 1 ) CALL INIJ( IJ, SUB(I, J-1), S)
  100        CONTINUE
  200     CONTINUE
C
C         NOW CONVERT ADJACENCY STRUCTURE TO INTERNAL FORMAT.
C
          CALL IJEND (S)
C
          RETURN
      END
```

```
C************************************************************
C************************************************************
C****** GRIDA .. GENERATES A COEFFICIENT MATRIX FOR THE   ***
C****** 5 - POINT DIFFERENCE OPERATOR ON AN N BY N GRID.  ***
C************************************************************
C************************************************************
C
C       INPUT PARAMETERS-
C                N - SIZE OF THE GRID (IMPLIES THAT THE
C                    NUMBER OF EQUATIONS IS N**2 )
C                S - WORKING STORAGE VECTOR
C
C************************************************************
C
      SUBROUTINE GRIDA ( N, S )
C
C************************************************************
      REAL    S(1)
      INTEGER SUB, I, J, IJ, N
C************************************************************
C
C         SUB MAPS THE COORDINATES (I,J) OF THE GRID ONTO
C         THE INTEGERS 1,2,...,N**2. THE SUBSCRIPTS
C         GENERATED BY SUB CORRESPOND TO THE STANDARD ROW
C         BY ROW ORDERING OF THE GRID.
C         ----------------------------------------------------
          SUB(I, J) = (I - 1)*N + J
C------------------------------------------------------------
          DO 200 I = 1, N
             DO 100 J = 1, N
                IJ = SUB(I,J)
                CALL INAIJ1( IJ, IJ, 4.0, S)
                IF(I .NE. 1) CALL INAIJ1( IJ, SUB(I-1, J), -1.0, S)
                IF(J .NE. 1) CALL INAIJ1( IJ, SUB(I, J-1), -1.0, S)
  100        CONTINUE
  200     CONTINUE
C
          RETURN
      END
```

```
C*************************************************************
C*************************************************************
C****** GRIDB1. GENERATES A RIGHT HAND SIDE FOR A    ********
C****** FIVE POINT DIFFERENCE OPERATOR ON AN N BY N ********
C****** GRID. THE DOMAIN OF THE PROBLEM IS ASSUMED TO ******
C****** BE (0,1) X (0,1), SO THE MESH WIDTH H IS 1/(N+1) ***
C****** THE BOUNDARY CONDITIONS  AND THE TRUE SOLUTION *****
C****** ARE PROVIDED BY THE STATEMENT FUNCTION F(I,J)  ******
C****** WHICH EVALUATES THE FUNCTION F AT (X,Y), WHERE *****
C****** X = I*H AND Y = J*H. ********************************
C*************************************************************
C*************************************************************
C
C       INPUT PARAMETERS-
C              N - SIZE OF THE GRID (IMPLIES NEQNS = N**2)
C              S - WORKING STORAGE VECTOR
C
C*************************************************************
C
      SUBROUTINE GRIDB1( N, S )
C
C*************************************************************
      REAL    S(1), H, H2
      INTEGER SUB, I, J, IJ, N
C*************************************************************
C
C         SUB MAPS THE COORDINATES (I,J) OF THE GRID ONTO
C         THE INTEGERS 1,2,...,N**2. THE SUBSCRIPTS
C         GENERATED BY SUB CORRESPOND TO THE STANDARD ROW
C         BY ROW ORDERING OF THE GRID.
C         ----------------------------------------------
          SUB(I, J) = (I - 1)*N + J
C         ----------------------------------------------
C         F(I,J) = EXP(X + Y), WHERE X=I*H AND Y=J*H. THE
C         RIGHT HAND SIDE IS ADJUSTED SO THAT F IS ALSO THE
C         TRUE SOLUTION TO THE PROBLEM, AS H GOES TO 0.
C
          F(I, J) = EXP(FLOAT(I)*H + FLOAT(J)*H )
C*************************************************************
C
          H = 1.0/FLOAT(N + 1)
          H2 = H*H
C
          DO 200 I = 1, N
C
             CALL INRHS( SUB(I,1), F(I,0), S)
             CALL INRHS( SUB(I,N), F(I,N+1), S)
             CALL INRHS( SUB(1,I), F(0,I), S)
             CALL INRHS( SUB(N,I), F(N+1,I), S)
C
             DO 100 J = 1, N
                CALL INRHS( SUB(I,J), -2.0*H2*F(I,J), S )
  100        CONTINUE
C
  200     CONTINUE
C
          RETURN
      END
```

```
C*************************************************************
C*************************************************************
C****** GRIDE1..COMPUTES THE ERROR FOR THE N BY N GRID  ****
C****** PROBLEM GENERATED BY GRIDA AND GRIDB1.    **********
C*************************************************************
C*************************************************************
C
C       INPUT PARAMETERS-
C              N - SIZE OF THE GRID (IMPLIES NEQNS = N**2)
C              S - WORKING STORAGE VECTOR
C       OUTPUT PARAMETER -
C              ERROR - THE MAXIMUM ERROR IN THE COMPUTED
C                 SOLUTION.
C
C*************************************************************
C
      SUBROUTINE GRIDE1( N, S, ERROR )
C
C*************************************************************
      REAL    S(1), H, H2, ERROR
      INTEGER SUB, I, J, IJ, N
C*************************************************************
C
C         SEE COMMENTS IN GRIDB1 ON SUB AND F.
C         -----------------------------------
          SUB(I, J) = (I - 1)*N + J
          F(I, J) = EXP(FLOAT(I)*H + FLOAT(J)*H )
C*************************************************************
C
          H = 1.0/FLOAT(N + 1)
          H2 = H*H
          ERROR = 0.0
C
          DO 200 I = 1, N
             DO 100 J = 1, N
                IJ = SUB(I, J)
                ERROR = AMAX1( ERROR, ABS(F(I,J) - S(IJ)) )
  100        CONTINUE
  200     CONTINUE
C
          RETURN
      END
```

Example 1

This is an example of the simplest use of the package, with each of the modules of method 1 used in sequence. The structure of A is input using GRID, which uses INIT, INIJ and IJEND. After ORDER1 is executed, GRIDA and GRIDE1 are used to input the numerical values of A and b respectively, using the interface modules INAIJ1 and INRHS. The module SOLVE1 is called to do the numerical solution, and then PSTATS is called to print out the statistics gathered by the interface during execution. Finally GRIDE1 is called to compute the error in the computed approximate solution.

```
$JOB    WATFIV   *************
        COMMON /USER/  MSGLVL, IERR, MAXS
        REAL     S(250)
           MAXS = 250
           N = 5
           CALL GRID(N, S)
           CALL ORDER1( S )
           CALL GRIDA( N, S)
           CALL GRIDB1( N, S )
           CALL SOLVE1 ( S )
           CALL PSTATS
           IF ( IERR .GT. 0 ) STOP
           CALL GRIDE1( N, S, ERROR  )
           WRITE(6, 11 ) ERROR
    11     FORMAT(/ 6X, 31HMAXIMUM ERROR IN THE SOLUTION    , E14.6)
           STOP
        END

$ENTRY

IJEND-  END OF ADJACENCY PAIRS

ORDER1- RCM ORDERING
         NUMBER OF EQUATIONS          25
         NUMBER OF EDGES IN GRAPH     40

         SIZE OF THE ENVELOPE         90
         BANDWIDTH                     5

SOLVE1- ENVELOPE SOLVE

PSTATS: STATISTICS
         TIME FOR ORDERING                      0.120
         STORAGE FOR ORDERING                 207.
         TIME FOR ALLOCATION                    0.030
         STORAGE FOR ALLOCATION               101.
         STORAGE FOR SOLUTION                 216.
         OVERHEAD STORAGE                      76.
         TIME FOR FACTORIZATION                 0.130
         TIME FOR SOLVING                       0.080
         OPERATIONS IN FACTORIZATION          320.
         OPERATIONS IN SOLVING                230.

 MAXIMUM ERROR IN THE SOLUTION    0.945091E-03
```

Example 2

This is similar to example 1, except SOLVE1 is called immediately after GRIDA is called. The interface detects that no right side b has been supplied, so after computing the factorization it by-passes the triangular solution and sets the solution to zero. After GRIDB1 is called, SOLVE1 is again called. The interface detects that the factorization has already been performed, and only the triangular solution is performed.

```
$JOB    WATFIV   *************
        COMMON /USER/  MSGLVL, IERR, MAXS
        REAL     S(250)
           MAXS = 250
           N = 5
           CALL GRID(N, S)
           CALL ORDER1( S )
           CALL GRIDA( N, S)
           CALL SOLVE1 ( S )
           CALL GRIDB1( N, S )
           CALL SOLVE1 ( S )
           CALL PSTATS
           IF ( IERR .GT. 0 ) STOP
           CALL GRIDE1( N, S, ERROR  )
           WRITE(6, 11 ) ERROR
    11     FORMAT(/ 6X, 31HMAXIMUM ERROR IN THE SOLUTION    , E14.6)
           STOP
        END

$ENTRY

IJEND-  END OF ADJACENCY PAIRS

ORDER1- RCM ORDERING
           NUMBER OF EQUATIONS            25
           NUMBER OF EDGES IN GRAPH       40

           SIZE OF THE ENVELOPE           90
           BANDWIDTH                       5

SOLVE1- ENVELOPE SOLVE
                NO RIGHT HAND SIDE PROVIDED,
                SOLUTION WILL BE ALL ZEROS.

SOLVE1- ENVELOPE SOLVE
                FACTORIZATION  ALREADY DONE.

PSTATS: STATISTICS
           TIME FOR ORDERING                        0.120
           STORAGE FOR ORDERING                   207.
           TIME FOR ALLOCATION                      0.030
           STORAGE FOR ALLOCATION                 101.
           STORAGE FOR SOLUTION                   216.
           OVERHEAD STORAGE                        76.
           TIME FOR FACTORIZATION                   0.130
           TIME FOR SOLVING                         0.080
           OPERATIONS IN FACTORIZATION            320.
           OPERATIONS IN SOLVING                  230.

 MAXIMUM ERROR IN THE SOLUTION    0.945091E-03
```

Example 3

This is the same as example 1, except after solving the problem corresponding to GRIDB1, a new right hand side is input using GRIDB2, corresponding to a different problem. The module SOLVE1 is called a second time, and just as in example **2**, the interface detects that the factorization has already been done, and only the triangular solution is performed.

```
$JOB    WATFIV   *************
        COMMON /USER/  MSGLVL, IERR, MAXS
        REAL     S(250)
           MAXS  = 250
           N  =  5
           CALL  GRID(N, S)
           CALL  ORDER1( S )
           CALL  GRIDA( N, S)
           CALL  GRIDB1( N, S )
           CALL  SOLVE1  ( S )
           CALL  PSTATS
           IF  ( IERR .GT. 0 )  STOP
           CALL  GRIDE1( N, S, ERROR  )
           WRITE(6,  11 )  ERROR
    11     FORMAT(/ 6X, 31HMAXIMUM ERROR IN THE SOLUTION    , E14.6)
           CALL  GRIDB2( N, S )
           CALL  SOLVE1  ( S )
           CALL  GRIDE2( N, S, ERROR  )
           WRITE(6,  11 )  ERROR
           STOP
        END

 $ENTRY

IJEND-  END OF ADJACENCY PAIRS

ORDER1- RCM ORDERING
           NUMBER OF EQUATIONS              25
           NUMBER OF EDGES IN GRAPH         40

           SIZE OF THE ENVELOPE             90
           BANDWIDTH                         5

SOLVE1- ENVELOPE SOLVE

PSTATS: STATISTICS
           TIME FOR ORDERING                           0.120
           STORAGE FOR ORDERING                      207.
           TIME FOR ALLOCATION                         0.030
           STORAGE FOR ALLOCATION                    101.
           STORAGE FOR SOLUTION                      216.
           OVERHEAD STORAGE                           76.
           TIME FOR FACTORIZATION                      0.130
           TIME FOR SOLVING                            0.080
           OPERATIONS IN FACTORIZATION               320.
           OPERATIONS IN SOLVING                     230.

 MAXIMUM ERROR IN THE SOLUTION    0.945091E-03

SOLVE1- ENVELOPE SOLVE
                FACTORIZATION  ALREADY DONE.

 MAXIMUM ERROR IN THE SOLUTION    0.454187E-04
```

Example 4

This example is almost identical to example 3, except it illustrates how problems having the same structure, but differing in both A and b can be solved. After solving the first problem both GRIDB2 and GRIDA are called, thus simulating a completely new numerical problem to be processed.

```
$JOB     WATFIV   *************
         COMMON /USER/  MSGLVL, IERR, MAXS
         REAL    S(250)
            MAXS = 250
            N = 5
            CALL GRID(N, S)
            CALL ORDER1( S )
            CALL GRIDA( N, S)
            CALL GRIDB1( N, S )
            CALL SOLVE1 ( S )
            CALL PSTATS
            IF ( IERR .GT. 0 ) STOP
            CALL GRIDE1( N, S, ERROR  )
            WRITE(6, 11 ) ERROR
     11     FORMAT(/ 6X, 31HMAXIMUM ERROR IN THE SOLUTION    , E14.6)
            CALL GRIDA( N, S)
            CALL GRIDB2( N, S )
            CALL SOLVE1 ( S )
            CALL GRIDE2( N, S, ERROR  )
            WRITE(6, 11 ) ERROR
            STOP
         END

     $ENTRY
```

```
IJEND-  END OF ADJACENCY PAIRS

ORDER1- RCM ORDERING
           NUMBER OF EQUATIONS              25
           NUMBER OF EDGES IN GRAPH         40

           SIZE OF THE ENVELOPE             90
           BANDWIDTH                         5

SOLVE1- ENVELOPE SOLVE

PSTATS: STATISTICS
           TIME FOR ORDERING                      0.120
           STORAGE FOR ORDERING                 207.
           TIME FOR ALLOCATION                    0.020
           STORAGE FOR ALLOCATION               101.
           STORAGE FOR SOLUTION                 216.
           OVERHEAD STORAGE                      76.
           TIME FOR FACTORIZATION                 0.130
           TIME FOR SOLVING                       0.080
           OPERATIONS IN FACTORIZATION          320.
           OPERATIONS IN SOLVING                230.

 MAXIMUM ERROR IN THE SOLUTION    0.945091E-03

SOLVE1- ENVELOPE SOLVE

 MAXIMUM ERROR IN THE SOLUTION    0.454187E-04
```

<u>Example 5</u>

This example is a modification of example 2 illustrating the use of the checkpoint/restart feature of the package. After the factorization is computed, SAVE is executed, which writes the current state of the computation on Fortran logical unit 3. The modules INIT and RESTRT are then executed to read the information from unit 3 and the computation resumes at the point at which SAVE was invoked.

```
$JOB    WATFIV  *************
        COMMON /USER/  MSGLVL, IERR, MAXS
        REAL     S(250)
           MAXS = 250
           N = 5
           CALL GRID(N, S)
           CALL ORDER1( S )
           CALL GRIDA( N, S)
           CALL SOLVE1 ( S )
           CALL SAVE ( 3, S )
C
C          THE NEXT DAY ...........
C
           CALL INIT ( S )
           CALL RESTRT ( 3, S )
           CALL GRIDB1( N, S )
           CALL SOLVE1 ( S )
           CALL PSTATS
           IF ( IERR .GT. 0 ) STOP
           CALL GRIDE1( N, S, ERROR  )
           WRITE(6, 11 ) ERROR
   11      FORMAT(/ 6X, 31HMAXIMUM ERROR IN THE SOLUTION    , E14.6)
           STOP
        END

$ENTRY

IJEND-   END OF ADJACENCY PAIRS

ORDER1-  RCM ORDERING
            NUMBER OF EQUATIONS              25
            NUMBER OF EDGES IN GRAPH         40

            SIZE OF THE ENVELOPE             90
            BANDWIDTH                         5

SOLVE1-  ENVELOPE SOLVE
                 NO RIGHT HAND SIDE PROVIDED,
                 SOLUTION WILL BE ALL ZEROS.

SAVE-    STORAGE VECTOR SAVED

RESTRT-  RESTART SYSTEM

SOLVE1-  ENVELOPE SOLVE
                 FACTORIZATION  ALREADY DONE.
```

```
PSTATS: STATISTICS
           TIME FOR ORDERING                        0.110
           STORAGE FOR ORDERING                   207.
           TIME FOR ALLOCATION                      0.020
           STORAGE FOR ALLOCATION                 101.
           STORAGE FOR SOLUTION                   216.
           OVERHEAD STORAGE                        76.
           TIME FOR FACTORIZATION                   0.120
           TIME FOR SOLVING                         0.080
           OPERATIONS IN FACTORIZATION            320.
           OPERATIONS IN SOLVING                  230.

 MAXIMUM ERROR IN THE SOLUTION    0.945091E-03
```

Example 6

This example illustrates a situation where there was sufficient storage provided to input the adjacency structure, but insufficient storage to execute the ordering algorithm.

```
$JOB    WATFIV   *************
        COMMON /USER/  MSGLVL, IERR, MAXS
        REAL     S(185)
           MAXS = 185
           N = 5
           CALL GRID(N, S)
           CALL ORDER1( S )
           STOP
        END

$ENTRY

IJEND-  END OF ADJACENCY PAIRS

ORDER1- RCM ORDERING
           NUMBER OF EQUATIONS              25
           NUMBER OF EDGES IN GRAPH         40

ORDER1- INSUFFICIENT STORAGE FOR
ORDERING, MAXS MUST BE AT LEAST       207
```

Example 7

This example is identical to example 1, except MAXS is not large enough for SOLVE1 to successfully execute. This situation is detected by ORDER1, which sets IERR positive. The modules INAIJ and INRHS detect the error condition, and do not increment STAGE, hence the error message from SOLVE1.

```
$JOB    WATFIV   *************
        COMMON /USER/  MSGLVL, IERR, MAXS
        REAL    S(210)
           MAXS = 210
           N = 5
           CALL GRID(N, S)
           CALL ORDER1( S )
           CALL GRIDA( N, S)
           CALL GRIDB1( N, S )
           CALL SOLVE1 ( S )
           CALL PSTATS
           IF ( IERR .GT. 0 ) STOP
           CALL GRIDE1( N, S, ERROR  )
           WRITE(6, 11 ) ERROR
     11    FORMAT(/ 6X, 31HMAXIMUM ERROR IN THE SOLUTION    , E14.6)
           STOP
        END

$ENTRY

IJEND-  END OF ADJACENCY PAIRS

ORDER1- RCM ORDERING
          NUMBER OF  EQUATIONS            25
          NUMBER OF  EDGES IN GRAPH       40

          SIZE OF THE ENVELOPE            90
          BANDWIDTH                        5

ORDER1- INSUFFICIENT STORAGE
FOR SOLVE1, MAXS MUST BE AT LEAST     216

SOLVE1- ENVELOPE SOLVE

SOLVE1- INCORRECT EXECUTION SEQUENCE.
ROUTINE INAIJ1 AND/OR INRHS MUST
BE CALLED BEFORE SOLVE1.

PSTATS: STATISTICS
          TIME FOR ORDERING                      0.120
          STORAGE FOR ORDERING                 207.
          TIME FOR ALLOCATION                    0.030
          STORAGE FOR ALLOCATION               101.
          STORAGE FOR SOLUTION                 216.
          OVERHEAD STORAGE                      76.
```

Example 8

This example is again identical to example 1, except SOLVE2 is erroneously called instead of SOLVE1. The interface detects that METHOD is 1, and sets IERR positive. (Method 2 is the unsymmetric analog of method 1, and handles the case where $A \neq A^T$.)

```
$JOB    WATFIV   *************
          COMMON /USER/  MSGLVL, IERR, MAXS
          REAL     S(250)
             MAXS = 250
             N = 5
             CALL GRID(N, S)
             CALL ORDER1( S )
             CALL GRIDA( N, S)
             CALL GRIDB1( N, S )
             CALL SOLVE2 ( S )
             CALL PSTATS
             IF ( IERR .GT. 0 ) STOP
             CALL GRIDE1( N, S, ERROR  )
             WRITE(6, 11 ) ERROR
      11     FORMAT(/ 6X, 31HMAXIMUM ERROR IN THE SOLUTION   , E14.6)
             STOP
          END

$ENTRY
```

```
IJEND-  END OF ADJACENCY PAIRS

ORDER1- RCM ORDERING
          NUMBER OF EQUATIONS            25
          NUMBER OF EDGES IN GRAPH       40

          SIZE OF THE ENVELOPE           90
          BANDWIDTH                       5

SOLVE2- ENVELOPE SOLVE

SOLVE2- INCOMPATIBLE ORDERING AND
SOLUTION ROUTINES,     METHOD =       1

PSTATS: STATISTICS
          TIME FOR ORDERING                      0.120
          STORAGE FOR ORDERING                 207.
          TIME FOR ALLOCATION                    0.030
          STORAGE FOR ALLOCATION               101.
          STORAGE FOR SOLUTION                 216.
          OVERHEAD STORAGE                      76.
          TIME FOR FACTORIZATION                 0.000
          TIME FOR SOLVING                       0.000
          OPERATIONS IN FACTORIZATION            0.
          OPERATIONS IN SOLVING                  0.
```