User Guide for **Sparspak90** A Fortran 90 Version of **Sparspak** : The Waterloo Sparse Matrix Package*

Alan George†

November 6, 2001

Abstract

Sparspak is a sparse matrix package that was implemented in the late 1970's. One of the important features of that package is an interface which shields the user from the complicated calling sequences common to most sparse matrix software. Modern programming languages such as Fortran 90 have important features that facilitate the design of flexible and "user-friendly" interfaces for software packages. These features include dynamic storage allocation, function name overloading, user-defined data types, and the ability to hide functions and data from the user. A new version of **Sparspak** called **Sparspak90** has been implemented in Fortran 90 . This guide described the interface and features of the new package and explains how to use it.

*This work was supported by the Natural Sciences and Engineering Research Council of Canada grant OGP 000811.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

Contents

List of Figures

1 Introduction

It is assumed that the reader has a working knowledge of Fortran 90 . The book by Metcalf and Reid [13] is a good reference for the language and its use.

For definiteness, the problem to be solved will be denoted by $\boldsymbol{Ax} = \boldsymbol{b}$, where $\boldsymbol{A}$ is an $n \times n$ sparse coefficient matrix, and the method to be used is Gaussian elimination.

Given a sparse linear system, reordering the matrix $\boldsymbol{A}$ may effect a substantial reduction in the cost of factorization and forward and backward solution. Thus given $\boldsymbol{A}$, one normally computes a factorization of $\boldsymbol{PAQ}$, where $\boldsymbol{P}$ and $\boldsymbol{Q}$ are row and column permutation matrices of the appropriate sizes. So the new system becomes $(\boldsymbol{PAQ})(\boldsymbol{Q}^{-1}\boldsymbol{x}) = \boldsymbol{Pb}$.

For purposes of explaining the structure and features of **Sparspak90** so that the user can understand how to use it, it is sufficient to restrict the problems to be solved to sparse systems whose coefficient matrices are positive definite and structurally symmetric, although they may not be numerically symmetric. Once an understanding of the basic structure and features of the package is established, it would be easy to extend this understanding to include other classes of problems which **Sparspak90** can handle. These will be described in Section13 in this guide.

With this restriction, it is possible to do symmetric reordering of $\boldsymbol{A}$ (i.e., $\boldsymbol{Q} = \boldsymbol{P}^T$) without regard to numerical stability and before the numerical factorization begins. In this way, the locations where fill will occur during the factorization can be determined before any numerical factorization is performed, and the data structures used to store the lower triangular factor $\boldsymbol{L}$ and/or upper triangular factor $\boldsymbol{U}$ can be set up prior to the numerical factorization. This process is called *symbolic factorization.*

Hence, the process of solving such sparse linear systems consists of a number of steps:

1. Reordering of the matrix $\boldsymbol{A}$
2. Symbolic factorization of the (reordered) matrix $\boldsymbol{A}$ and the creation of the data structures for the factorization and forward and backward substitution
3. Putting numerical values of $\boldsymbol{A}$ into the data structures
4. Numerical factorization of $\boldsymbol{A}$
5. Forward and backward substitution (triangular solution)

There is no general "best method" for solving sparse systems of equations. Even if one restricts the basic algorithm to Gaussian elimination, the way it is best implemented often depends on the characteristics of the given sparse linear system. As a consequence, this sparse matrix package is designed to accommodate a variety of methods, and allow for convenient inclusion of methods yet to be developed.

Sparse systems arise in a variety of contexts. Sometimes many problems having the same nonzero structure must be solved, and sometimes many problems differing only in their right hand sides must be solved. Also, the way in which their structure and numerical values become available is highly variable. The package is able to handle these situations in a natural way.

2 Overall Structure and Basic Use of Sparspak90

This section describes the high-level structure of the package, with particular emphasis on its user interface. Only the basic features of the various components of the package will be discussed in this section. Discussion of more sophisticated use of the package and additional features of it is postponed until later sections.

Use of the package is "object-oriented". Users can adopt the view that the package provides two basic types of "objects" (user-defined data types and subroutines which act on them): "problem" objects that contain a problem ($\boldsymbol{A}$, $\boldsymbol{b}$ and eventually the solution $\boldsymbol{x}$), and "solver" objects that accept as input a problem (in a problem object) and produce a solution to $\boldsymbol{Ax} = \boldsymbol{b}$. In the sequel, "object" means an instance of a user-defined data type together with subroutines that act upon it. An example appears in Figure 1 to provide some concreteness to this somewhat abstract description.

In the example, the subroutine **MakeTriDiagProblem** generates a 5×5 symmetric positive definite tridiagonal system of equations. The subroutine **MakeTriDiagProblem** is given in Figure 2 in the next subsection.

2.1 The Problem type

Regardless of the level of user sophistication, one task is fundamental and ubiquitous; the user must communicate the sparse matrix problem to the package. The task is complicated by the variety of ways in which the problem may materialize, as well as by transformations that the user might want to apply to the input. The different ways include:

Figure 1 Example of simple use of the Sparspak90 package.

```
program SimpleExample
    use Sparspak90                    ! make package available
    type (Problem) :: p               ! declare problem object
    type (SparseSpdSolver) :: s       ! declare solver object

    call MakeTriDiagProblem(p, 5)     ! create test problem

    call Construct(s, p)              ! create solver object
    call Solve(s, p)                  ! solve problem

    call PrintSolution(p)             ! print solution

    call Destruct(p)                  ! release storage used by p
    call Destruct(s)                  ! release storage used by solver

end program SimpleExample
```

- the structure of the problem and its numerical values may become available simultaneously or at differing times.
- there may be many systems to be solved, differing only in their numerical values.
- there may be numerous problems that differ only in their right hand sides; these may be available all at once, or each may be the result of computations involving previous right hand side(s) in a sequence. The latter circumstance arises naturally if the package is being used in connection with solving a system of nonlinear equations.
- given the matrix $\boldsymbol{A}$, for testing purposes it may be desirable to generate a right hand side corresponding to a given solution, or given the structure of $\boldsymbol{A}$, it may be desirable to assign numerical values giving $\boldsymbol{A}$ certain properties (random, symmetric, positive definite, diagonally dominant, etc.). Such capabilities can be useful in developing and testing software.

The list above is far from exhaustive, but serves to illustrate the variety of situations which the package can handle.

With these considerations in mind, it is natural that the package provides a type that can be viewed as the "problem repository." This type is given

the name **Problem** since instances of this type contain the numerical values and structure of the coefficient matrix, its right hand side, corresponding solution vector, and related information pertaining to the problem.

The type **Problem** has associated with it various subroutines which operate on its data. Roughly speaking, these routines fall into four categories:

1. procedures which provide for input of structural and numerical values, such as **InAij**, **InRow**, **InColumn** and **InRhs**.

2. procedures which adjust the input such as **ReplaceAij**, **ZeroMatrix**, **ZeroRhs**, **MakeStructureSymmetric**, and **MakeSymmetric**, (The last two procedures make the problem structurally and numerically symmetric, respectively.)

3. procedures which retrieve and/or display information, such as **GetRhs**, **GetSolution**, and **PrintSolution**.

4. procedures which provide information about the problem, such as **IsSymmetric**, **IsStructureSymmetric**, **IsAijPresent**. The first two procedures determine whether the matrix is numerically and structurally symmetric, respectively, while the latter determines whether the ij-th element of the matrix is present.

The **Problem** type can represent sparse matrix problems with square or rectangular matrices, with symmetric, unsymmetric or triangular structures, and with symmetric or unsymmetric numerical values. However, it is important to understand that the **Problem** type does not have any built-in "intelligence". It is simply a repository for whatever the user presents to it. It does not attempt to identify or exploit special features of the problem that might be present, and does not maintain any such information.

Figure 2 contains a Fortran 90 subroutine which generates an $n \times n$ tri-diagonal matrix problem. The example provides an opportunity to elaborate on several design features of **Sparspak90** . Extensive use is made of the programming language feature known as *function name overloading*. That is, different routines are allowed to have the same names, provided that their parameter lists ("signatures") differ. The compiler detects which routines should be called by matching up the types and number of parameters in the routines. Thus, routines which perform essentially the same role, even though they may employ different internal data structures or operate on different types, can still have the same name. This name overloading capability helps to reduce the intellectual overhead in learning to use the package.

Figure 2 Tridiagonal test problem generator.

```
subroutine MakeTriDiagProblem(p, n)
    use Sparspak90
    integer :: n, i
    type (Problem) :: p

    call Construct(p)

    do i = 1, n-1
        call InAij(p, i+1, i, -1D0);    call InAij(p, i, i, 4D0)
        call InAij(p, i, i+1, -1D0);    call InRhs(p, i, 1D0)
    end do

    call InAij(p, n, n, 4D0);    call InRhs(p, n, 1D0)

end subroutine MakeTriDiagProblem
```

In the the example in Figure 2, an instance of the **Problem** type is initialized by calling the function **Construct**. *All* user-defined types in the package are initialized by calling a function whose name is **Construct**. Of course the routine that is actually executed will depend on the type of the first argument of the parameter list (which the compiler can detect). The key point is that the user has to remember only one function name in connection with creating new instances of a type (objects).

Another example demonstrating the convenience of overloading is illustrated by the function **InAij** in the example in Figure 2. As noted in the previous section, the structure and the numerical values of the system of equations can arrive at different times and in different aggregations. If the user does not know the value of a_{ij} but wishes to communicate the fact that the (i,j)-th element of $\boldsymbol{A}$ is present, the function **InAij** is still used, but the last parameter is omitted. Analogously overloaded input routines, e.g., **InRow**, **InColumn** are available in the event that the nonzeros (or perhaps only their positions in the matrix) become available by rows or columns (or parts thereof). Similar remarks apply to **InRhs**. The right hand side $\boldsymbol{b}$ can be input one element at a time, as shown in the example in Figure 2, or as a sub-array with an accompanying list of subscripts, or all at once. In all cases a function with the same name is called. Thus, the user must remember only a small number of function names to use **Problem** objects.

2.2 Solver objects

The second major type of object that the typical user of the package will employ is a "solver" object. Loosely speaking, a **Solver** object accepts a **Problem** object as input and produces a solution to the problem. The package contains numerous different "solvers" for sparse systems of equations. That is, in addition to a type **Problem** in the package, there are solver types, each consisting of a particular data structure (or structures) with a default ordering algorithm and numerical factorization and triangular solution routines. Instances of these types can be regarded as "solver objects". The solver types implement a particular overall approach to solving a sparse system. For example, for symmetric positive definite systems, there are several effective algorithms for finding a low fill ordering, there are several efficient methods for storing Cholesky factors, and there are several efficient ways of implementing the factorization using the same data structure (left-looking, right-looking, multifrontal) [1, 9, 14]. Various solver objects result from selecting different combinations of these options.

The multitude of solvers is necessary because problems vary in several dimensions. They may or may not be square; if square, they may or may not be structurally symmetric. If they are structurally symmetric, they may or may not be numerically symmetric. Regardless of either shape or symmetry, row and/or column interchanges may be necessary to ensure numerical stability. In addition, for any particular combination of problem attributes above, there may be more than one approach that will solve the problem. Of course a solver that assumes no special features will cope with them all, but generally not as efficiently as one that exploits special features that a problem may possess.

2.3 Coarse structure of Sparspak90

At a basic level of resolution, the package can be regarded as providing just two fundamental types of objects, namely problem objects and xxxSolver objects, where xxx denotes one of the numerous possibilities mentioned in the previous subsection. For example, **Sparspak90** contains a solver for symmetric positive definite problems that reorders the problem to reduce fill. This solver type has the name **SparseSpdSolver**, standing for (Sparse Symmetric positive-definite Solver). A simple example showing its use is displayed in Figure 1, where the subroutine in Figure 2 was used to create a small symmetric positive definite tri-diagonal problem.

The package also contains a solver for symmetric positive definite prob-

Figure 3 Example of simple use of the package involving a different solver type.

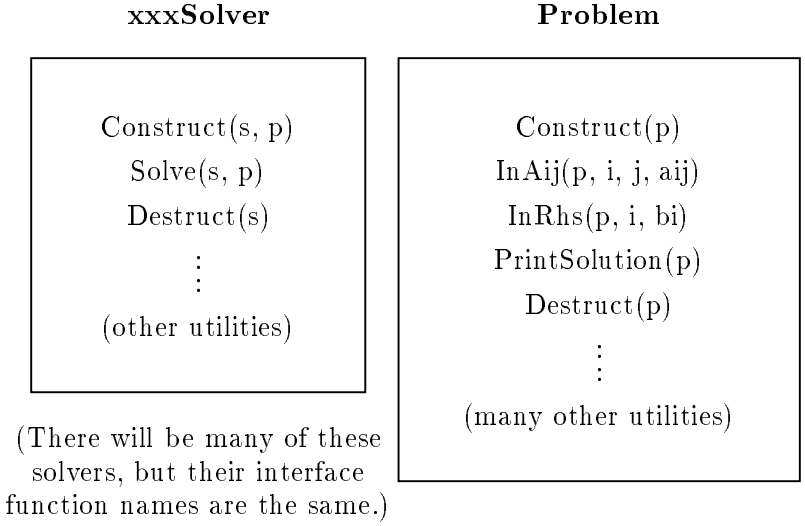
```
program SimpleExample
    use Sparspak90
    type (Problem) :: p              ! declare problem
    type (EnvSpdSolver) :: s         ! declare solver

    call MakeTriDiagProblem(p, 5)    ! create test problem

    call Construct(s, p)             ! create solver object

    call Solve (s, p)                ! instruct s to solve p

    call PrintSolution(p)            ! print the solution

    call Destruct(p)                 ! release storage for p
    call Destruct(s)                 ! release storage for s

end program SimpleExample
```

lems that orders the problem so that it has a small envelope [9]. This solver type has the name **EnvSpdSolver**, which stands for (Envelope-reducing Symmetric positive-definite Solver). The *only* change necessary in the program in Figure 1 in order to use this solver would be in line 4, where **SparseSpdSolver** would be changed to **EnvSpdSolver**. The use of function name overloading for **Solve** and **Destruct** means that no other changes are required; of course different procedures will be invoked. The compiler detects which procedures should be called by matching up the types and numbers of parameters in the procedures. Thus, the new program would be as shown in Figure 3.

Thus, from the perspective of a typical user, **Sparspak90** can be viewed as shown in Figure 4. Two types of objects are involved in its use, namely **Problem** objects and solver objects.

Typically, a user would have a problem at hand. The first step is to create a **Problem** object **p** by making a subroutine call, **Construct(p)**, which sets up and initializes the data structures for storing the problem matrix, the right hand side and the solution. The **Problem** module provides subroutine **InAij(p, i, j, aij)** for input of the entries in the matrix $\boldsymbol{A}$ to the **Problem** object. The routine communicates to **p** that there is a nonzero at row **i** and

Figure 4 Coarse structure of Sparspak90



column **j** in $\boldsymbol{A}$ with the numerical value **aij**. The subroutine **InRhs(p, i, bi)** allows the user to input a value **bi** to the i^{th} entry in the right hand side vector $\boldsymbol{b}$.

Important Notes If **InAij(p, i, j, aij)** is called multiple times with the same i and j, then the effect is additive, meaning that each time it is called the value aij is added to the value at position (i, j) if one already exists.

To solve the problem, a call to subroutine **Construct(s, p)** would create a solver object, **s**, of a type specified by the user. Then a call to subroutine **Solve(s, p)** would solve the system and the solution is stored in the **Problem** object **p**. To see the solution, the user would call **PrintSolution(p)**. Then calling **Destruct(s)** and **Destruct(p)** would release the storage used by **s** and **p**.

3 Additional Features

3.1 Other ways to input a problem

There are subroutines in the **Problem** module which provide other ways of inputting entries into either the matrix $\boldsymbol{A}$ or the right hand side $\boldsymbol{b}$.

Important Notes As mentioned before, with all input routines which input a value, the effect of inputting to a location multiple times is cumulative. This means that the new value is added to the existing value. Therefore, if a user wants to input a new matrix $\boldsymbol{A}$ or a new right hand side $\boldsymbol{b}$, then the user must call **ZeroMatrix(p)** or **ZeroRhs(p)** first which sets the relevant data structure of the **Problem** object **p** to zero. Another point to note is that if a nonzero does not already exist at a specified location then it is inserted.

InRow (p, rNum, nElements, colSubs, values) Execution of a call to this routine lets the user add to the current values of **nElements** entries in row **rNum** of the matrix $\boldsymbol{A}$ with column subscripts as specified in the array **colSubs**. The array **values** contains the corresponding numerical values for those entries. If the user wants to communicate to the problem object that some locations in row **rNum** of $\boldsymbol{A}$ are nonzero but the numerical values for these locations are not yet available, then the last parameter **values** may be omitted.

InColumn(p, cNum, nElements, rowSubs, values) is a column analogue of **InRow (p, rNum, nElements, colSubs, values)**.

InRhs(p, rhs) adds an array of values **rhs** to the existing values in the right hand side of a **Problem** object.

InRhs(p, nElements, rowSubs, values) adds to the current values of **nElements** entries in the right hand side whose positions are specified in the array **rowSubs**. The values to be added are taken from the corresponding values in the array **values**.

3.2 Retrieval of solution and problem data for use in further computation

GetSolution(p, x) in the **Problem** module allows the user to retrieve the solution vector and store it in an array **x** for use in further computation.

GetRhs(p, rhs) retrieves the right hand side and stores it in an array **rhs**.

3.3 Reinitializing data structures to zero

The **Problem** module offers the user the subroutines

- **ZeroMatrix(p)**
- **ZeroRhs(p)**
- **ZeroSolution(p)**
- **ZeroDiagonal(p)**

These routines set the relevant data structures of the **Problem** object **p** to zero. This is helpful when solving systems with multiple right hand sides or many problems with the same structure but different numerical values as discussed in later sections as well as for testing purposes.

3.4 Solving many problems with the same nonzero structure

In certain applications, many problems which have the same sparsity structure but different numerical values must be solved. In this case, the structure input, ordering, and data structure set-up need only be done once. The control sequence is depicted in Figure 5, where **p** is the **Problem** object and **s** is the solver object.

To reuse the structure and ordering of an existing problem, the user must call the subroutine **ZeroProblem(p)** which sets the values of $\boldsymbol{A}$ and $\boldsymbol{b}$ to zero before inputting the new numerical values for $\boldsymbol{A}$ and $\boldsymbol{b}$. After the new numerical values for $\boldsymbol{A}$ and $\boldsymbol{b}$ have been input to **p**, the user need only call **Solve(s, p)** again, reusing the same solver **s**. There is internal information maintained by the solver which makes it aware that this is a new problem with new numerical values so that it would input the new values from $\boldsymbol{A}$ to the existing data structure, factor $\boldsymbol{A}$ and do the triangular solve again. However it would skip the step that finds a reordering for the matrix $\boldsymbol{A}$ and the symbolic factorization part of the solution process. An example of reusing the ordering and data structure of a solver for two problems with the same structure is given in Figure 6.

Note that if such problems must be solved over an extended time period (i.e., in different runs), the user can use the **Save** and **Restore** facility on the problem **p** and **s** as detailed in a later section and thus avoid the input of the structure of $\boldsymbol{A}$ and the ordering part of work in subsequent equation solutions.

Figure 5 Control sequence for solving many problems with the same structure.

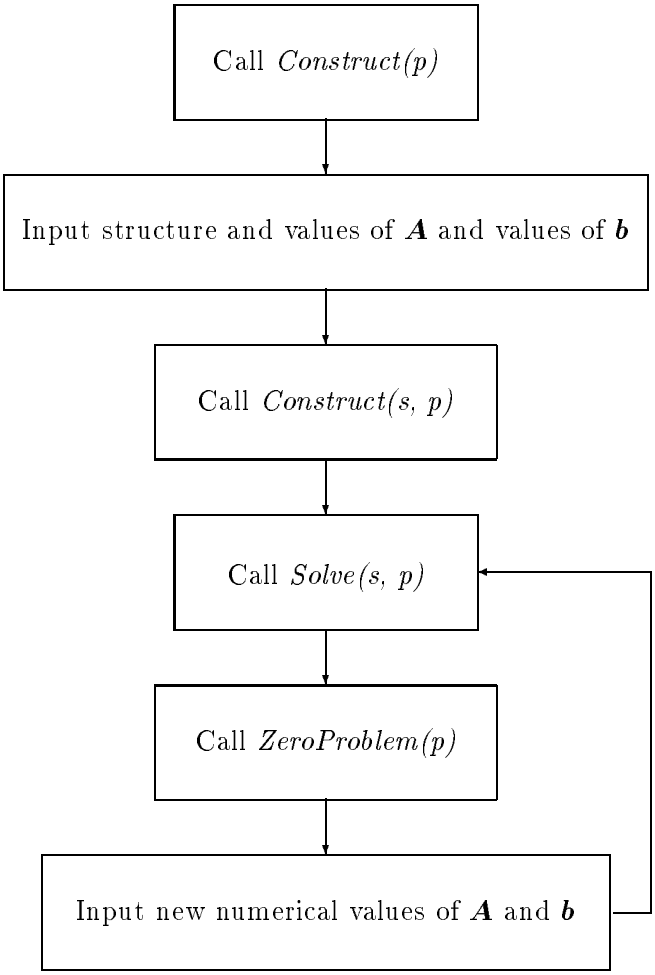


Figure 6 Example of solving two problems with the same structure.

```
program tSameStructure02
    use Sparspak90

    implicit none
    type (Problem) :: p
    type (SparseSpdSolver) :: s
    integer :: size

    size = 5
    call MakeGridProblem(p, size, size, "5pt")
                                          ! create a grid problem p
    call MakeProblemRandom(p)             ! make p numerically random
    call MakeSymmetric(p)                 ! make p symmetric
    call MakeDiagDominant(p)              ! make p diagonally dominant
    call MakeRhs(p)                       ! set the right hand side of p
                                          ! such that the solution vector
                                          ! will be (1, 2, 3, ...)
    call Construct(s, p)                  ! create solver object s
    call Solve(s, p)                      ! instruct s to solve p
    call PrintSolution(p, 10)             ! print part of the solution

    call MakeProblemRandom(p)             ! make p numerically random
    call MakeSymmetric(p)                 ! make p symmetric
    call MakeDiagDominant(p)              ! make p diagonally dominant
    call MakeRhs(p)                       ! set the right hand side of p
                                          ! such that the solution vector
                                          ! will be (1, 2, 3, ...)
    call Solve(s, p)                      ! instruct s to solve p
    call PrintSolution(p, 10)             ! print part of the solution

    call Destruct(s)                      ! release storage for s
    call Destruct(p)                      ! release storage for p
end program tSameStructure02
```

3.5 Solving many problems which differ only in their right hand sides

In some applications, numerous problems which differ only in their right hand sides must be solved. In this case, it is necessary to factor $\boldsymbol{A}$ into $\boldsymbol{LU}$ or $\boldsymbol{LL}^T$ only once, and use the factors repeatedly in the calculation of $\boldsymbol{x}$ for each different $\boldsymbol{b}$. Again, **Sparspak90** can handle this situation in a straightforward manner, as illustrated by the flowcharts in Figure 7.

Both factorization and triangular solution are performed during the call to **Solve**, with only the forward and backward substitution part performed in each execution of **TriangularSolve**. There are two versions of the subroutine **TriangularSolve**. For the version which takes a **Problem** object as the second parameter, the user must call **ZeroRhs(p)** first before putting the new values for the right hand side in **p** and then calling **TriangularSolve(s, p)**. For the version which takes an array **rhs** as the second parameter, the user need only put the new values of the right hand side in **rhs** before calling **TriangularSolve(s, rhs)**. The solution is returned in the array **rhs** in this case. Examples demonstrating the two ways of solving problems which differ only in their right hand sides are shown in Figures 8 and 9.

3.6 Symmetric coefficient matrices: saving storage using the parameter mType

In certain contexts the **Problem** object **p** will be used to store a symmetric matrix $\boldsymbol{A}$. In order to conserve storage, the user may choose to input only the *lower triangular part* of $\boldsymbol{A}$. Certain subroutines and functions need to know when a symmetric matrix is represented this way. The parameter used to indicate this is called **mType**. It may have one of two valid values. A value of "L" or "l" indicates to the routine that only the lower triangular part is present. Among others, the subroutines **FindScales** and **ComputeResidual** in the **Problem** module require the use of this parameter when only the lower triangular part of a symmetric matrix $\boldsymbol{A}$ is stored in the **Problem** object **p**.

3.7 Solving $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$

For unsymmetric solvers, **Sparspak90** provides subroutines for solving $\boldsymbol{A}^T\boldsymbol{x} = \boldsymbol{b}$. If the matrix **A** has already been factored, then executing the calling sequence

Call TransposeTriangularSolve (s, p)

Figure 7 Control sequences for solving many problems which differ only in their right hand sides

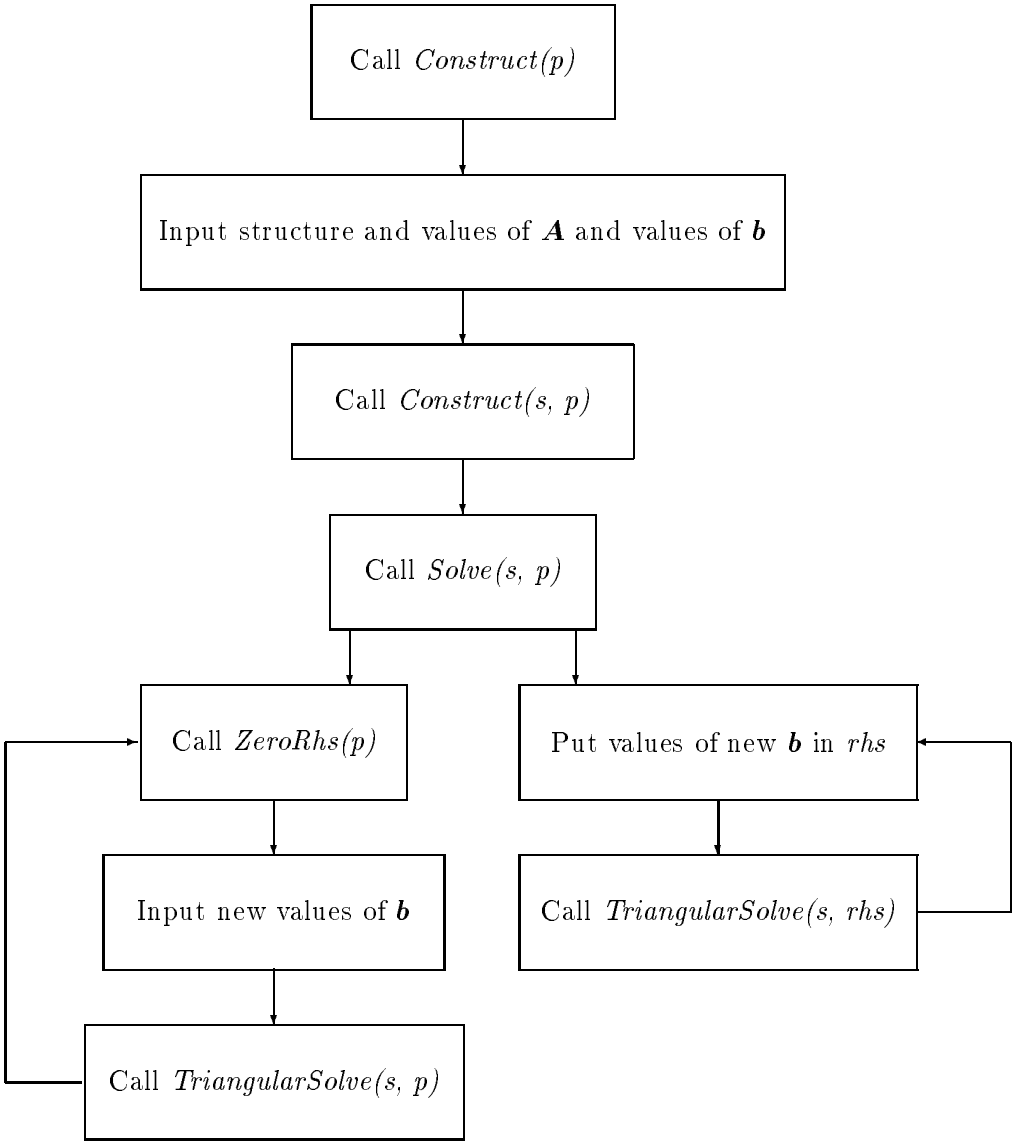


Figure 8 Example of solving three problems which differ only in their right hand sides by putting the new right hand sides into the problem object first.

```
program tDiffRhs02
    use Sparspak90

    implicit none
    type (Problem) :: p
    type (EnvSolver) :: s
    integer :: i
    real(double):: x(25)

    call MakeGridProblem(p, 5, 5, '5pt')     ! create a grid problem p
    call MakeStructureSymmetric(p)           ! make p structurally symmetric
    call MakeProblemRandom(p)                ! make p numerically random
    call MakeDiagDominant(p)                 ! make p diagonally dominant
    call Makerhs(p)                          ! set the right hand side of p
                                             ! such that the solution vector
                                             ! will be (1, 2, 3, ...)

    call Construct(s, p)                     ! create solver s from p
    call Solve( s, p )                       ! instruct s to solve p
    call PrintSolution(p, 10)                ! print part of the solution

    x = (/ ( 1, i = 1, 25 ) /)
    call MakeRhs(p, x)                       ! set the right hand side of p
                                             ! such that the solution vector
                                             ! will be equal to x

    call TriangularSolve(s, p)               ! instruct s to do forward and
                                             ! backward substitution using
                                             ! the factors stored in s

    call PrintSolution(p, 10)                ! print part of the solution

    x = (/ ( i*i, i = 1, 25 ) /)             ! repeat the above with a new
                                             ! set of values for x
    call MakeRhs(p, x)
    call TriangularSolve(s, p)
    call PrintSolution(p, 10)

    call Destruct(s)                         ! release storage for s
    call Destruct(p)                         ! release storage for p
end program tDiffRhs02
```

Figure 9 Example of solving three problems which differ only in their right hand sides using a separate array for the new right hand sides and solutions.

```
program tDiffRhs02a
    use Sparspak90
    implicit none
    type (Problem) :: p
    type (EnvSolver) :: s
    integer :: i
    real(double):: x(25), rhs(25)

    call MakeGridProblem(p, 5, 5, '5pt')  ! create a grid problem p
    call MakeStructureSymmetric(p)        ! make p structurally symmetric
    call MakeProblemRandom(p)             ! make p numerically random
    call MakeDiagDominant(p)              ! make p diagonally dominant
    call Makerhs(p)                       ! set the right hand side of p
                                          ! such that the solution vector
                                          ! will be (1, 2, 3, ...)
    call Construct(s, p)                  ! create solver s from p
    call Solve( s, p )                    ! instruct s to solve p
    call PrintSolution(p, 10)             ! print part of the solution

    x = (/ ( 1, i = 1, 25 ) /)
    call MakeRhs(p, x)                    ! set the right hand side of p
                                          ! such that the solution vector
                                          ! will be equal to x
    call GetRhs(p, rhs)                   ! retrieve the right hand side
                                          ! of p and store in the array rhs
    call TriangularSolve(s, rhs)          ! instruct s to do forward and
                                          ! backward substitution using
                                          ! the factors stored in s and
                                          ! the array rhs as the right
                                          ! hand side and store the
                                          ! solution back in rhs.
    call PrintVector(10, rhs, 'Solution') ! print part of the solution
                                          ! stored in rhs

    x = (/ ( i*i, i = 1, 25 ) /)          ! repeat the above with a new
                                          ! set of values for x.
    call MakeRhs(p, x)
    call GetRhs(p, rhs)
    call TriangularSolve(s, rhs)
    call PrintVector(10, rhs, 'Solution')

    call Destruct(s)                      ! release storage for s
    call Destruct(p)                      ! release storage for p
end program tDiffRhs02a
```

will compute the solution to the system $A^T x = b$ using the factors stored in the solver s. Here **p** is a problem object. There is an another version of this routine - **TransposeTriangularSolve(s, rhs)** where **s** is a solver object and **rhs** is an array in which the right-hand side is stored as input and the solution is returned as output.

Otherwise, executing the statement

Call TransposeSolve (s, p)

will solve the transposed system from scratch.

3.8 Other features

3.8.1 Matrix norms

Matrix norms are often useful in the analysis of matrix algorithms. The subroutines **OneNorm(p, mType)** and **InfinityNorm(p, mType)** in the **Problem** module compute and return the one-norm and infinity-norm respectively of the problem matrix in the **Problem** object **p**. Note that the second parameter **mType** is required when the matrix $\boldsymbol{A}$ is symmetric and only the lower triangular half of it is stored. Refer to subsection 3.6 for details.

3.8.2 Computing the residual

If the user has an approximate solution for the **Problem** object **p** in the array **x**, then the calling sequence

Call ComputeResidual (**p**, **res**, **x**, **mType**)

computes the difference between the right hand side $\boldsymbol{b}$ of the given problem **p** and **"Ax"** and stores it in the array **res**. Again note that **mType** is required when the matrix $\boldsymbol{A}$ is symmetric and only the lower triangular half of it is stored. See subsection 3.6 for details.

3.8.3 Matrix property inquiry

The **Problem** module offers the users the following functions:

- **IsStructureSymmetric**
- **IsLowerTriangular**

- **IsUpperTriangular**
- **IsSymmetric**
- **IsDiagDominant**

Their meaning is obvious from their names. Each of these functions takes a single parameter **p** which is a **Problem** object and returns a logical value. The function **IsDiagDominant** takes a second optional parameter **mType**. See the subsection 3.6 for an explanation for its role.

3.8.4 Modifying a matrix

Sometimes, the user may want to change the numerical value of an entry in the problem matrix $\boldsymbol{A}$. The **Problem** module provides the following subroutines for such purposes.

ReplaceAij(p, i, j, aij) A call to this routine sets the numerical value of the $(i,j)^{th}$ entry in the matrix $\boldsymbol{A}$ to **aij**. If a nonzero did not exist at this location before this routine is called, then the value is inserted.

ReplaceColumn(p, cNum, nElements, rowSubs, values) A call to this subroutine sets the numerical values of **nElements** entries in the column **cNum** of matrix $\boldsymbol{A}$ in the **Problem** object **p**. The user supplies the row subscripts of the locations in the array **rowSubs** and the corresponding values in the array **values**. If a nonzero did not exist at a location (as specified by **cNum** and **rowSubs**) before this routine is called, then the value is inserted at that location.

ReplaceRow(p, rNum, nElements, colSubs, values) is a row counterpart of **ReplaceColumn(p, cNum, nElements, rowSubs, values)**.

3.8.5 Inquiring whether a matrix entry is nonzero, and retrieving its numerical value

AijPresent(p, rNum, cNum) checks to see if the entry (rNum, cNum) is present. If it is, the routine returns .true.; otherwise, .false. is returned.

GetAijProblem(p, rNum, cNum) finds **p**'s matrix entry at (rNum, cNum) and returns it. If none exists, zero is returned.

4 Obtaining Information from Sparspak90

4.1 Displaying information about a problem object

The subroutine **PrintStats(p)** displays the number of rows, columns, edges, diagonal edges in the graph of the matrix, nonzeros, diagonal nonzeros in the matrix and any string of description or information (if any were provided by the user) of the **Problem** object. Note that the edges in the graph of the matrix refers to the structural nonzeros in the matrix $\boldsymbol{A}$; diagonal edges in the graph refers to the structural nonzero diagonal elements in $\boldsymbol{A}$. By contrast, the nonzeros in the matrix refers to the nonzero values in $\boldsymbol{A}$ and the nonzero diagonal elements refers to the diagonal nonzero values in $\boldsymbol{A}$.

4.2 Displaying information about a solver object

At times, it may be useful to see the contents of the data structures of the solver object. Each solver module provides a subroutine for this purpose. To use it, one would call **Print(s)** where s is the relevant solver object.

4.3 Displaying execution times

The subroutine **PrintTimes(s)** in each solver module displays the time used for ordering, symbolic factorization, entering the matrix $\boldsymbol{A}$ from the **Problem** object to the **Solver** object, factorization, triangular solution and iterative refinement.

4.4 Messages produced by Sparspak90

In the package, a variable **msgLevel** which stands for "message level" is provided. It governs the amount of information printed by **Sparspak90** . Its default value is 2, and for this value fatal errors, warnings and summary information are printed. When **msgLevel** is set to 1 by the user, only fatal error messages and summary information are printed. Setting the value of **msgLevel** to 3 provides additional trace information about the computation.

In many circumstances, **Sparspak90** will be embedded as a toolbox in another "super package" which models phenomena involving sparse matrix problems. Messages printed by **Sparspak90** may be useless or even confusing to the ultimate users of the super package, or the super package may wish to field the error conditions and perhaps take some corrective action

which makes the error messages irrelevant. Thus, *all* of the printing by **Sparspak90** can be prevented by setting **msgLevel** to 0.

To summarize, we have

msgLevel	**output messages**
0	no output.
1	fatal errors, summary information.
2	fatal errors, warnings, summary information.
3	fatal errors, warnings, trace and summary information.

Notes By setting **msgLevel** to 3, trace information about major subroutines is printed as they are entered and exited during the computation.

4.5 Creating a log

If the user so chooses, it is possible to save the output printed by **Sparspak90** to a log file. A subroutine **SetLogFile(filename)** is provided for the user to supply a filename for the log file. Once this routine is called, all subsequent output would be printed to the file specified in addition to being displayed on the standard output unit.

5 Displaying Pictures of Objects

5.1 Displaying a picture of the nonzero structure of the matrix A

The subroutine **Picture(p)** displays a picture of the nonzero structure of the problem matrix $\boldsymbol{A}$.

5.2 Displaying a picture of the data structure for the factor L

The subroutine **Picture(s)** prints a "picture" of the data structure for the factor $\boldsymbol{L}$ in the solver object **s** if symbolic factorization has been done.

5.3 Creating Tex files

Sparspak90 provides a facility for the user to create a file containing a LaTeX picture environment of the matrix $\boldsymbol{A}$ contained in a **Problem** object

p. To do that a user would execute the statement

Call MakeTexFile (**p**, **filename**)

where **filename** is the name of the destination LaTeX file.

6 Scaling

Sparspak90 provides scaling facilities to scale the matrix $\boldsymbol{A}$ so that the norms of its columns are all about one. Routines are also provided for applying the appropriate scalings and un-scalings to the right hand side and the solution. Typically a user would first execute the statement

Call FindScales (**p**, **mType**)

which computes the row and column scales. Refer to the subsection 3.6 for the role served by the optional second parameter.

Then executing the statement

Call ScaleProblem (**p**)

would scale the matrix $\boldsymbol{A}$ and the right hand side $\boldsymbol{b}$. After calling **Solve(s, p)** to solve the scaled problem, the user must execute the statement

Call UnscaleSolution (**p**)

in order to obtain the solution to the original problem. If for some reason the user wants to get back the original matrix $\boldsymbol{A}$ and the right hand side $\boldsymbol{b}$, he/she must execute the statement

Call UnscaleProblem (**p**)

which undoes the scaling of $\boldsymbol{A}$ and $\boldsymbol{b}$.

An example showing the use of scaling is given in Figure 10. Refer to section 10 for details on the subroutine HarwellBoeingRead used in the example.

7 Iterative Refinement of the Solution

Sometimes, computed solutions may be improved by doing iterative refinement. The package provides in each solver a subroutine **Refine** which does extended precision iterative improvement. To use it, the user calls **Refine(s,**

Figure 10 Example showing the use of scaling

```
program tScaling01
    use  Sparspak90

    implicit none
    type (GPSolver) :: s
    type (Problem) :: p

    call MakeRandomProblem(p, 100, 100, 0.1_double)
                                ! create a random problem p
    call MakeRHS(p)             ! create a right hand side for p such that
                                ! the solution vector is (1, 2, 3, ...)

    call FindScales(p)          ! find a scaling for p
    call ScaleProblem(p)        ! scale p

    call Construct(s, p);       ! create a solver s from p
    call Solve(s, p)            ! instruct the solver to solve p

    call UnscaleSolution(p)     ! unscale the solution
    call PrintSolution(p, 10)   ! print part of the solution

    call Destruct(s)            ! release storage for the solver
    call Destruct(p)            ! release storage for p
end program tScaling01
```

Figure 11 Example illustrating the use of the subroutine **Refine**.

```
program tRefine01_5pt
    use Sparspak90

    implicit none
    type (OneWaySpdSolver) :: s
    type (Problem) :: p
    integer :: n

    n = 200
    call MakeGridProblem(p, n, n, "5pt")  ! create a grid problem p

    call Construct(s, p)                  ! create a solver s from p
    call Solve(s, p)                      ! instruct s to solve p
    call PrintSolution(p, 20)             ! print part of the solution

    call Refine(s, p, mType="L")          ! do iterative refinement
    call PrintSolution(p, 20)             ! print part of the solution

    call Destruct(p)                      ! release storage for p
    call Destruct(s)                      ! release storage for s
end program tRefine01_5pt
```

p, mType, iter) where **s** is a solver object, **p** is a **Problem** object and, **iter** (optional) is the maximum number of iterations to be performed. The optional parameter **mType** is required when the matrix is symmetric and only the lower triangular part of it is stored. See the subsection 3.6 for details.

An example showing the use of the subroutine **Refine** is given in Figure 11.

8 Condition Number Estimation

For a *square* matrix **A**, **Sparspak90** provides a subroutine **CondEst** which computes an estimate of the one-norm condition number of **A**. It should be noted that factorization *must* be done before **CondEst** is called. To use it, execute the statement

Call CondEst (**s**, **estimate**, **p**, **mType**)

where **s** is a solver object, **p** is a problem object and **estimate** is an estimate of the condition number. See Section 3.6 for the role of **mType**.

Notes For the **KdeltaSolver** type the estimated condition number returned by **CondEst** is that of the augmented matrix

$$\begin{pmatrix} \delta \boldsymbol{I} & \boldsymbol{A} \\ \boldsymbol{A}^T & -\delta \boldsymbol{I} \end{pmatrix}.$$

Similarly, for the **CLLSSolver** type, the estimated condition number returned by **CondEst** is that of the augmented matrix

$$\begin{pmatrix} \boldsymbol{I} & \boldsymbol{O} & \boldsymbol{A} \\ \boldsymbol{O} & \mu \boldsymbol{I} & \boldsymbol{C} \\ \boldsymbol{A}^T & \boldsymbol{C}^T & -\delta \boldsymbol{I} \end{pmatrix}.$$

For these two solver types, the use of the parameter **mType** is not required.

9 Saving and Restoring Problems and Solvers

Sparspak90 provides subroutines **Save** and **Restore** which allow the user to stop the calculation at some point, save the results in an external sequential file, and then resume the calculation at exactly that point some time later. To save the results of the computation done thus far, the user executes the statement

Call Save (**s**, **filename**)

where **filename** is the name of the external file to which the results are to be written, along with other information needed to restart the computation. Here **s** is a solver object. If execution is then terminated, the state of the computation can be re-established by executing the following statement.

Call Restore (**s**, **filename**)

Note that executing **Save** does not destroy any information; the computation can proceed just as if **Save** were not executed.

Similarly, execution of the statements

Call Save (**p**, **filename**)

and

Call Restore (**p**, **filename**)

Figure 12 Example showing how a problem object and an envelope solver are saved.

```
program tSaveEnvSolver01
    use Sparspak90

    implicit none
    type (Problem) :: p
    type (EnvSolver) :: s
    integer :: size
    real(double) :: d

    size = 200
    d = 0.5_double
    call MakeRandomProblem(p, size, size, d)
                                          ! create a random problem p
    call MakeStructureSymmetric(p)        ! make p structurally
                                          ! symmetric
    call MakeProblemRandom(p)             ! make p numerically random
    call MakeDiagDominant(p)              ! make p diagonally dominant
    call MakeRhs(p)                       ! set the right hand side
                                          ! such that the solution
                                          ! vector is (1, 2, 3, ...)
    call Save(p, 'tSaveEnvSolver01.Problem')   ! save p to an external file

    call Construct(s, p)                  ! create solver object s from
                                          ! p
    call Print(s, 'EnvSolver before Save')     ! print the solver object
    call Save(s, 'tSaveEnvSolver01.EnvSolver') ! save s to an external file

    call Destruct(s)                      ! release storage for s
    call Destruct(p)                      ! release storage for p
end program tSaveEnvSolver01
```

save and restore a **Problem** object **p** to and from an external sequential file.

Examples showing how to save and restore a **Problem** object and a **EnvSolver** (envelope solver) object are displayed in Figures 12 and 13.

10 Reading and Writing Harwell-Boeing Files

In **Sparspak90** , a user can read in a problem stored in a file in the Harwell-Boeing format [4] by executing the statement

Call HarwellBoeingRead (**p**, **fileName**)

Figure 13 Example showing how a problem object and an envelope solver are restored.

```
program tRestoreEnvSolver01
    use Sparspak90

    implicit none
    type (Problem) :: p
    type (EnvSolver) :: s

    call Restore(p, 'tSaveEnvSolver01.Problem')   ! restore problem object p
    call Restore(s, 'tSaveEnvSolver01.EnvSolver') ! restore solver object s
    call Print(s, 'EnvSolver after Restore')      ! print data structures of
                                                  ! s

    call Solve(s, p)                              ! instruct s to solve p
    call PrintSolution(p, 10)                     ! print part of the
                                                  ! solution

    call Destruct(s)                              ! release storage for s
    call Destruct(p)                              ! release storage for p
end program tRestoreEnvSolver01
```

where **filename** is the name of the file where the problem data is stored and **p** is the **Problem** object created to store the problem read.

Conversely, if the user has a **Problem** object **p** which he/she wants to save and store in a file in the Harwell-Boeing format, then

Call HarwellBoeingWrite (**p**, **fileName**, **outfile**, **title**)

would be the statement to call. Here **filename** is the name of the output file, **outfile** is an optional unit number and **title** is an optional title to appear at the top of the file.

Important Notes There are restrictions on the type of Harwell-Boeing format files which **Sparspak90** can handle. **Sparspak90** does not handle files which store the problem matrix $\boldsymbol{A}$ in the elemental matrix format. Also if there are multiple right-hand-sides, only the first one is read in.

An example showing the use of the subroutine HarwellBoeingRead is given in Figure 10.

11 Creating Test Problems

11.1 Test problem generation

The **Problem** module provides some facilities for generating test problems.

MakeRandomProblem(p, nRows, nCols, density) generates a random problem with **nRows** rows and **nCols** columns and density **density**.

MakeSPDProblem(p, nRows, density) creates a **Problem** object with an **nRows** $\times$ **nRows** random matrix which is both symmetric and positive-definite.

MakeTridiagProblem(p, n) creates a **Problem** object which is tri-diagonal. The right hand side $\boldsymbol{b}$ is set so that the solution $\boldsymbol{x}$ is all ones.

11.2 Right hand side generation

The **Problem** module has a subroutine **MakeRhs(p, x, mType)** which constructs the right hand side of a problem given an **x** for the equation "**Ax = b**". If **x** is not present, then a right hand side is constructed so that (a,the) solution is 1,2,3,...m. See the subsection 3.6 for details of the role of the optional parameter **mType**. This routine is useful for testing purposes.

11.3 Modification of a matrix so that it has certain properties

The **Problem** module has a number of subroutines that allow the user to modify the matrix $\boldsymbol{A}$ so that it has certain desired properties. The following subroutines are provided:

- **MakeStructureSymmetric**
  This routine adds element positions (no numerical values) so that the structure of the problem is symmetric.
- **MakeSymmetric**
  This routine does the same as **MakeStructureSymmetric**, and in addition, makes the problem numerically symmetric.
- **MakeDiagDominant**
  This routine make the matrix column-diagonally dominant. It takes an optional second parameter **mType**. See subsection 3.6 for details.

- **MakeSpd**
  This routine calls **MakeSymmetric** and then **MakeDiagDominant**.

- **MakeProblemRandom** This routine sets the elements of the matrix to random elements drawn from a uniform random distribution on (0,1).

12 More Sophisticated Use of Sparspak90 : Looking Inside

12.1 One step at a time

The package is designed for ease of use so that once the **Problem** object is constructed and the matrix $\boldsymbol{A}$ and the right hand side $\boldsymbol{b}$ are entered, then all that the user needs to do is to pick an appropriate solver type, call **Construct(s)** and then call **Solve(s, p)** to solve the system of linear equations.

Internally, the call to **Solve(s, p)** involves a sequence of steps. First the matrix $\boldsymbol{A}$ is ordered. Then symbolic factorization is done to find the nonzero structure of the factor(s) and the relevant data structures for the factor(s) are set up. Then the numerical values from the matrix $\boldsymbol{A}$ are input to the data structures. Finally, the matrix $\boldsymbol{A}$ is factored followed by triangular solution to find the solution corresponding to the right hand side.

In certain circumstances, the user may want to call the individual steps directly instead of calling **Solve**. In that case, the user must make sure that these steps are called in the right order although there are safeguards in the package to prevent an improper processing sequence.

The solver type is designed so that it has built-in awareness of the current stage of the solution process. This enables the solver to enforce the correct execution sequence of the various interface routines. Before the actual execution of each interface routine, a check is made to ensure that all previous interface modules have been successfully completed. This avoids producing erroneous results due to an improper processing sequence, or accidental omission of steps.

When an error occurs during the execution of a phase, the execution of all the subsequent phases is skipped, even if they are invoked by the user.

When **Solve** is called, it would find a reordering of the matrix and/or perform symbolic factorization only if they have not been done before. However the other three steps (input of Problem matrix, factorization and substitution) are *always* executed regardless of whether they have been done before.

So if the matrix $\boldsymbol{A}$ remains the same and the factorization has already been done once, it is much more efficient to just call **TriangularSolve** (which only does the forward and backward substitution) instead of calling **Solve** again.

The names of the subroutines to call and the proper calling sequence are as follows:

1. **call FindOrder(s)**
2. **call SymbolicFactor(s)**
3. **call InMatrix(s, p)**
4. **call Factor(s)**
5. **call TriangularSolve(s, p)**

An example demonstrating how to call the individual steps directly to solve a system is given in Figure 14.

12.2 User-supplied ordering functions

Sparspak90 allows a user to provide his/her own ordering function; this is achieved by calling the subroutine

FindOrder (s, **OrderFunction** = **MyOrderRoutine**)

directly with a reference to the ordering function being passed in as the last parameter. A module containing a user-defined ordering function is shown in Fig 15 and an example demonstrating the use of this user-provided ordering method is given in Fig 16

For the solver types **EnvSolver**, **EnvSpdSolver**, **SparseSolver**, **SparseSpdSolver**, **GPSolver**, **KdeltaSolver**, and **CLLSSolver**, a subroutine

FindOrder (s, **perm**)

where **perm** is a permutation vector is also provided to allow a user to supply an ordering permutation. An example showing how to order the problem matrix by supplying a permutation vector is given in Figure 17.

Important Notes The user should note that for the **KdeltaSolver**, and **CLLSSolver** solver types, the ordering the user provides is applied to the augmented matrices as described in subsection 8.

After the ordering is done, the user may choose to simply call **Solve(s, p)** or to call the individual steps as previously described directly to solve the system.

Figure 14 Example showing how a user may call the individual steps for solution directly.

```
program tRQTSolver05
    use Sparspak90

    implicit none
    type (Problem) :: p
    type (RQTSolver) :: s
    integer sze

    sze = 200
    call makeGridProblem(p, sze, sze, "9pt")  ! create a grid problem p

    call MakeStructureSymmetric(p)            ! make p symmetric
    call makeProblemRandom(p)                 ! make p numerically random
    call makeDiagDominant(p)                  ! make p diagonally dominant
    call makeRhs(p)                           ! create a right hand side
                                              ! so that the solution vector
                                              ! is (1, 2, 3, ...)
    call Construct(s, p)                      ! create a solver s from p
    call FindOrder(s)                         ! reorder the matrix
    call SymbolicFactor(s)                    ! do symbolic factorization
    call InMatrix(s, p)                       ! put the numerical values
                                              ! in the data structures.
    call Factor(s)                            ! do numerical factorization
    call TriangularSolve( s, p )              ! do forward and backward
                                              ! substitution
    call printsolution(p, 20)                 ! print part of the solution

    call Destruct(s)                          ! release storage for s
    call Destruct(p)                          ! release storage for p
end program tRQTSolver05
```

Figure 15 Example of a module as a container for a user-defined ordering function

```
!!  The purpose of this module is to demonstrate how to incorporate a user-
!!  supplied ordering when using Sparspak90.
!*****************************************************************************
    module SpkMyOrder
!*****************************************************************************
    use SpkGraph;    use SpkOrdering
!*****************************************************************************
!!  Graph class:
!!  nV  - the number of vertices in the graph.
!!  (xadj, adj) - array pair storing the adjacency lists of the vertices.
!!
!!  The adjacency lists of the graph are stored in consecutive locations
!!  in the array adj. The adjacency list for the i-th vertex in the graph
!!  is stored in positions adj(k), k = xadj(i), .... xadj(i+1)-1.
!!  For convenience in accessing the lists, xadj is of length nV+1, with
!!  xadj(nV+1) = nEdges+1.
!!
!!  When the graph is symmetric, if vertex i is in vertex j's adjacency
!!  then vertex j is in vertex i's list. Using the representation above
!!  each edge in the graph is stored twice. There are no self-loops.
!!  (No "diagonal elements".)
!*****************************************************************************
!!  Ordering class:
!!  nRows is the number of rows in the matrix
!!  nCols is the number of columns in the matrix
!!
!!  Ordering objects contain two permutations and their inverses:
!!  rPerm is a row permutation: rPerm(i)=k means that the new position of
!!      row k is in position i in the new permutation.
!!  rInvp is a row permutation satisfying rInvp(rPerm(i)) = i. Thus,
!!      rInvp(k) provides the position in the new ordering of the original
!!      row k.
!!  cPerm and cInvp are analogous to rPerm and rInvp, except they apply
!!      to column permutations of the matrix.
!*****************************************************************************
    contains
!*****************************************************************************
    subroutine MyOrderRoutine ( g, order )
!*****************************************************************************
        type (Graph) :: g
        type (Ordering) ::  order
!*****************************************************************************
        call Construct(order, g%nV)
                        ! This creates identity permutations of size g%nV.
        ! For purposes of illustration, the reverse ordering is used here.
        order%rperm = (/ (i, i = g%nV, 1, -1) /)
        order%cperm = order%rperm
        order%rInvp(order%rPerm) = (/ (i, i = 1, g%nV) /)
        order%cInvp = order%rInvp
    end subroutine myOrderRoutine
!*****************************************************************************
    end module SpkMyOrder
```

Figure 16 Example showing how a user may use their own ordering method in solving a system.

```
program tMyOrder01
    use Sparspak90
    use SpkMyOrder

    implicit none
    type (SparseSpdSolver) :: s, s1
    type (Problem) :: p

    call MakeGridProblem(p, 5, 5)           ! create a grid problem p
    call Construct(s, p)                    ! create solver object from p
    call FindOrder(s, OrderFunction=MyOrderRoutine)
                                            ! order the problem matrix
                                            ! using a user-suppplied
                                            ! function
    call Print(s%slvr%order)
    call Solve(s, p)                        ! instruct s to solve p
    call PrintSolution(p, 5)                ! print part of the solution

    call Construct(s1, p)                   ! create another solver object
                                            ! s1
    call Solve(s1, p)                       ! instruct s1 to solve p
                                            ! using the default ordering
                                            ! method
    call PrintSolution(p, 5)                ! print part of the solution

    call Destruct(p)                        ! release storage for p
    call Destruct(s)                        ! release storage for s
    call Destruct(s1)                       ! release storage for s1
end program tMyOrder01
```

Figure 17 Example showing the use of a user-supplied permutation vector in ordering a problem.

```
program tFindOrderPerm02
    use Sparspak90

    implicit none
    type (Problem) :: p
    type (EnvSolver) :: s
    integer :: size, i
    integer :: perm(16)

    size = 4
    call MakeGridProblem(p, size, size, "9pt")
                                    ! create a grid problem p
    call MakeStructureSymmetric(p)  ! make p structurally symmetric
    call MakeProblemRandom(p)       ! make p random numerically
    call MakeDiagDominant(p)        ! make p diagonally dominant
    call MakeRhs(p)                 ! create a right hand side so
                                    ! that the solution is (1, 2, 3, ...)

    ! create a random permutation of the right size.
    perm(1:16) = (/ (i, i = 1, 16) /)
    call RandomPermutation(16, perm)

    call Construct(s, p)            ! create a solver s from p
    call FindOrder(s, perm)         ! order using random permutation
    call Solve(s, p)                ! instruct s to solve p
    call PrintSolution(p, 10)       ! print part of the solution

    call Destruct(s)                ! release storage for s
    call Destruct(p)                ! release storage for p

end program tFindOrderPerm02
```

13 Solvers and How to Choose them

13.1 Square Non-Singular Linear Systems

It is mentioned in Section 1 that there are several solver types implementing different methods of solution. For square non-singular linear systems, there are four basic solver types together with their unsymmetric counterparts; the only distinction between the former and the latter being that the former assumes the coefficient matrix **A** is symmetric, and the latter assumes that **A** is unsymmetric. Two of the four basic solver types (**RQTSpdSolver** and **OneWaySpdSolver**) differ only in the ordering method employed. Thus, for this problem class, **Sparspak90** only provides three essentially distinct methods, with each one having a symmetric and unsymmetric version. Hence, in this section, the remarks will largely be confined to the symmetric versions; comparative remarks about them will also apply to their unsymmetric analogues.

The basic solver types are as follows; the remarks comparing them, and the advice provided should be regarded as at best tentative. Characteristics of sparse matrices vary a great deal.

Solver Types	*Basic Strategy and References*
EnvSpdSolver	The objective of this method is to reorder A so it has a small bandwidth or profile[12]. The well-known reverse Cuthill-McKee algorithm is used. For relatively small problems, say $n \leq 200$, it is probably the best overall solver type to use.
OneWaySpdSolver	The objective of this method is to reduce storage requirement, but the factorization time will usually be substantially higher than the EnvSpdSolver or SparseSpdSolver methods. Its storage requirement will usually be substantially less than the SparseSpdSolver methods (unless n is very large). The same remark is true about the relative solution times. Thus, this method is often useful when storage is restricted, and/or when many problems which differ only in the right hand side must be solved (see Section 3.5). This solver uses the one-way dissection ordering method. It is specifically tailored for "finite element problems", typical of those arising in structural analysis and the numerical solution of partial differential equations[6].
RQTSpdSolver	The objective of this method is to reduce storage requirement, but the factorization time will usually be substantially higher than the EnvSpdSolver or SparseSpdSolver methods. Its storage requirement will usually be substantially less than the SparseSpdsolver methods (unless n is very large). The same remark is true about the relative solution times. Thus, this method is often useful when storage is restricted, and/or when many problems which differ only in the right hand side must be solved (see Section 3.5). This solver uses the refined quotient-tree ordering[7] and is effective for problem less specific than the "finite element problems" as mentioned above.

Continued on the next page.

Solver Types	*Basic Strategy and References*
SparseSpdSolver	This method attempts to find orderings which minimize fill-in, and it exploits all zeroes. Its ordering times are almost always greater than those above, but for moderate-to-large problems the reduced factorization times usually are more than compensatory. It uses a variant of the minimum degree algorithm, and is suitable for all sparse problems[11]. For systems arising from "finite element problems" as mentioned above, an alternative ordering methods is the nested dissection ordering [8]. This can be invoked by overriding the default ordering routine by calling **FindOrder(s, OrderFunction=ND)** as mentioned in Subsection 12.2 since this ordering method is provided as part of **Sparspak90** .

To summarize, our tentative advice and guidelines for choosing a solver for square non-singular linear systems are as follows:

1. For small problems, use the EnvSpdSolver.

2. For small to moderate size problems that have to be solved only once, use the EnvSpdSolver if enough storage is available. If not, use the OneWaySpdSolver or RQTSpdSolver. If the problem is quite large, the SparseSpdSolver might be better.

3. For moderate to large problems, use either the OneWaySpdSolver, RQTSpdSolver or SparseSpdSolver. If many problems differing only in the right hand side must be solved, the OneWaySpdSolver or RQT-SpdSolver may be the best. If the problem is quite large, and many problems having the same structure, but different numerical values, must be solved, then the SparseSpdSolver is probably the best. (See Subsections 3.4 and 3.5.)

13.2 Linear Least Squares Problem Solvers

13.2.1 Regularized Least Square Problems

A matrix $\widehat{\boldsymbol{K}}$ is *symmetric quasi-definite* if there exists a permutation matrix $\boldsymbol{P}$ such that $\boldsymbol{P}\widehat{\boldsymbol{K}}\boldsymbol{P}^T$ has the form

$$\boldsymbol{K} = \begin{pmatrix} \boldsymbol{H} & \boldsymbol{A} \\ \boldsymbol{A}^T & -\boldsymbol{G} \end{pmatrix},$$

where $\boldsymbol{H} \in \Re^{m \times m}$, $\boldsymbol{G} \in \Re^{n \times n}$ and both are symmetric and positive definite.

An important property of symmetric quasi-definite matrices $\boldsymbol{K}$ is that $\boldsymbol{PKP}^T$ always has an $\boldsymbol{LDL}^T$ factorization for any permutation matrix $\boldsymbol{P}$. Such a factorization is also called the Cholesky factorization of a quasi-definite system.

The strong factorizability of symmetric quasi-definite matrices makes them attractive in solving a general linear system $\boldsymbol{Ax} = \boldsymbol{b}$, where the matrix $\boldsymbol{A}$ does not have special properties (symmetric, positive definite) or is rectangular. The *"KKT" method, or regularized augmented system method*, is a particular example. The key idea of the KKT method is to handle general systems while enjoying the advantages of symmetric positive definite systems.

For a small scalar $\delta > 0$, consider the augmented system

$$\boldsymbol{K}_\delta \begin{pmatrix} \boldsymbol{s} \\ \boldsymbol{x} \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{0} \end{pmatrix} \quad \text{where} \quad \boldsymbol{K}_\delta = \begin{pmatrix} \delta \boldsymbol{I} & \boldsymbol{A} \\ \boldsymbol{A}^T & -\delta \boldsymbol{I} \end{pmatrix}. \tag{1}$$

The matrix $\boldsymbol{K}_\delta$ is symmetric quasi-definite. For any $\delta > 0$ and any symmetric permutation, $\boldsymbol{PK}_\delta \boldsymbol{P}^T$ always has an $\boldsymbol{LDL}^T$ factorization. A solution to (1) provides a solution to the regularized least squares problem

$$||\boldsymbol{Ax} - \boldsymbol{b}||^2 + ||\delta \boldsymbol{x}||^2.$$

If $\boldsymbol{A}$ is square and nonsingular or the system is rectangular but compatible, the $\boldsymbol{LDL}^T$ factorization of $\boldsymbol{K}_\delta$ can be treated as an approximate factorization of $\boldsymbol{K}_0$, where the matrix

$$\boldsymbol{K}_0 = \begin{pmatrix} \boldsymbol{O} & \boldsymbol{A} \\ \boldsymbol{A}^T & \boldsymbol{O} \end{pmatrix}$$

is obtained from $\boldsymbol{K}_\delta$ by setting $\delta = 0$. Therefore, iterative refinement can be used on the system

$$\boldsymbol{K}_0 \begin{pmatrix} \boldsymbol{s} \\ \boldsymbol{x} \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{0} \end{pmatrix}$$

to remove the effect of δ and get a better approximate solution to $\boldsymbol{Ax} = \boldsymbol{b}$.

Sparspak90 provides the **KdeltaSolver** type which creates an augmented matrix as in (1) from the given matrix $\boldsymbol{A}$ and solves the augmented system for an approximate solution to the system $\boldsymbol{Ax} = \boldsymbol{b}$ where $\boldsymbol{A}$ has no special properties (positive definite, diagonally dominant) or is rectangular. It uses the **SparseSpdSolver** as an underlying solver.

13.2.2 Constrained Least Square Problems

Consider

$$\min_{\boldsymbol{x}} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2$$

where $\boldsymbol{A} \in \Re^{m \times n}$ with $m \geq n$ subject to

$$\boldsymbol{C}\boldsymbol{x} = \boldsymbol{d}$$

where $\boldsymbol{C} \in \Re^{p \times n}$ with $p \leq n$.

For small scalars $\delta > 0$ and $\mu > 0$, consider the augmented system

$$\boldsymbol{K}_{\mu\delta} \begin{pmatrix} \boldsymbol{r} \\ \boldsymbol{s} \\ \boldsymbol{x} \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{d} \\ \boldsymbol{0} \end{pmatrix} \quad \text{where} \quad \boldsymbol{K}_{\mu\delta} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{O} & \boldsymbol{A} \\ \boldsymbol{O} & \mu\boldsymbol{I} & \boldsymbol{C} \\ \boldsymbol{A}^T & \boldsymbol{C}^T & -\delta\boldsymbol{I} \end{pmatrix}. \tag{2}$$

Analogous to the matrix $\boldsymbol{K}_\delta$ in section13.2.1, this matrix $\boldsymbol{K}_{\mu\delta}$ is symmetric quasi-definite. Hence, for any $\delta > 0$, $\mu > 0$ and any symmetric permutation, $\boldsymbol{P}\boldsymbol{K}_{\mu\delta}\boldsymbol{P}^T$ always has an $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$ factorization. Thus, (2) is equivalent to the least squares problem

$$\min_{\boldsymbol{x}} \left\| \begin{pmatrix} \sqrt{\mu}\boldsymbol{A} \\ \boldsymbol{C} \\ \sqrt{\mu\delta}\boldsymbol{I} \end{pmatrix} \boldsymbol{x} - \begin{pmatrix} \sqrt{\mu}\boldsymbol{b} \\ \boldsymbol{d} \\ \boldsymbol{0} \end{pmatrix} \right\|_2^2.$$

A solution to (2) provides a solution to the constrained least squares problem

$$\min_{\boldsymbol{x}} \|\boldsymbol{C}\boldsymbol{x} - \boldsymbol{d}\|_2^2 + \mu\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2 + \mu\delta\|\boldsymbol{x}\|_2^2$$

The $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$ factorization of $\boldsymbol{K}_\delta$ can be treated as an approximate factorization of $\boldsymbol{K}_{\mu 0}$, where the matrix

$$\boldsymbol{K}_{0\mu} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{O} & \boldsymbol{A} \\ \boldsymbol{O} & \mu\boldsymbol{I} & \boldsymbol{C} \\ \boldsymbol{A}^T & \boldsymbol{C}^T & \boldsymbol{O} \end{pmatrix}.$$

is obtained from $\boldsymbol{K}_{\mu\delta}$ by setting $\delta = 0$. Therefore, iterative refinement can be used on the system

$$\boldsymbol{K}_{\mu 0} \begin{pmatrix} \boldsymbol{r} \\ \boldsymbol{s} \\ \boldsymbol{x} \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{d} \\ \boldsymbol{0} \end{pmatrix}$$

to remove the effect of δ and get a better solution to $\boldsymbol{A}\boldsymbol{x} \approx \boldsymbol{b}$ subject to $\boldsymbol{C}\boldsymbol{x} = \boldsymbol{d}$.

Sparspak90 provides the **CLLSSolver** type for solving constrained linear least squares problems using the method outlined above. As in the **KdeltaSolver** type, it uses the **SparseSpdSolver** as an underlying solver.

13.3 Square Non-Singular Linear Systems Which Require Pivoting

Let $\boldsymbol{Ax} = \boldsymbol{b}$ be a large sparse nonsingular system to be solved via Gaussian elimination. If $\boldsymbol{A}$ has no special properties (positive definite, diagonally dominant), some form of row and/or column interchanges are necessary to ensure numerical stability. Given $\boldsymbol{A}$, one normally obtains a factorization of $\boldsymbol{PAQ}$, where $\boldsymbol{P}$ and $\boldsymbol{Q}$ are permutation matrices of the appropriate size. Thus, the process has two stages:

1. factor $\boldsymbol{PAQ}$ into the product of upper and lower triangular matrices $\boldsymbol{LU}$.

2. compute $\boldsymbol{x}$ using $\boldsymbol{L}, \boldsymbol{U}, \boldsymbol{P}, \boldsymbol{Q}$ and $\boldsymbol{b}$: solve $\boldsymbol{Ly} = \boldsymbol{Pb}$ and $\boldsymbol{Uz} = \boldsymbol{y}$, and then set $\boldsymbol{x} = \boldsymbol{Qz}$.

The coefficient matrix $\boldsymbol{A}$ normally suffers some *fill*; its factors will generally have nonzeros in positions that are zero in $\boldsymbol{PAQ}$. The choice of the permutations can dramatically affect the amount of fill that occurs. Thus, when $\boldsymbol{A}$ is sparse, one or both of the permutations above are determined *during the factorization* by a combination of (usually competing) numerical stability and sparsity requirements. Different matrices, though they may have the same nonzero pattern, will usually yield different permutations and therefore have factors with different sparsity patterns. In other words, for general sparse matrices, it normally is not possible to predict where fill will occur before the computation begins. Thus, some form of *dynamic* storage scheme is required which allocates storage for fill as the computation proceeds, which makes efficient implementation of Gaussian elimination for such systems difficult.

The conventional approach to implementation uses both row and column interchanges during the factorization. The interchanges are chosen to minimize the potential fill at each step, subject to the requirement that the pivot element is above a certain threshold value. Thus, $\boldsymbol{P}$ and $\boldsymbol{Q}$ are determined during the numerical factorization. For details on this approach, see [2, 3, 5].

Gilbert and Peierls[10] were the first to provide an implementation of sparse Gaussian elimination whose run time can be shown to be proportional to the amount of floating-point arithmetic done. The implementation uses *partial pivoting* (row interchanges) only; it computes the factorization $\boldsymbol{PAQ} = \boldsymbol{LU}$, for some permutation $\boldsymbol{P}$ that is chosen either in the normal way, or by some form of *threshold* pivoting that selects row interchanges that

limit fill while ensuring that the pivot element is not too small. The column permutation $\boldsymbol{Q}$ is chosen in advance. This approach is referred to as the G-P strategy.

Sparspak90 provides the **GPSolver** type for solving large sparse square non-singular systems that has no special properties. The algorithm used is based on the approach outlined above, which is basically Gaussian elimination with partial pivoting.

References

[1] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1987.

[2] I.S. Duff. MA32 - A package for solving sparse unsymmetric systems using the frontal method. Technical Report AERE R 10079, Harwell, 1981.

[3] I.S. Duff. The design and use of a frontal scheme for solving sparse unsymmetric equations. In J. P. Hennart, editor, *Proceedings of the Third IIMAS Workshop on Numerical Analysis*, pages 240–247. Lecture Notes in Mathematics (909), Springer-Verlag, 1982.

[4] I.S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15:1–14, 1989.

[5] I.S. Duff and J.K. Reid. The multifrontal solution of unsymmetric sets of linear equations. *SIAM J. Sci. Stat. Comput.*, 5:633–641, 1984.

[6] A. George. An automatic one-way dissection algorithm for irregular finite element problems. *SIAM J. Numer. Anal.*, 17:740–751, 1980.

[7] A. George and J. W-H. Liu. Algorithms for matrix partitioning and the numerical solution of finite element systems. *SIAM J. Numer. Anal.*, 15:297–327, 1978.

[8] A. George and J. W-H. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM J. Numer. Anal.*, 15:1053–1069, 1978.

[9] A. George and J. W-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.

[10] J.R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Stat. Comput.*, 9:862–874, 1988.

[11] J. W-H. Liu. On multiple elimination in the minimum degree algorithm. Technical Report 83-03, Dept. of Computer Science, York University, North York, Ontario, 1983.

[12] J. W-H. Liu and A.H. Sherman. Comparative analysis of the Cuthill-McKee and reverse Cuthill-McKee ordering algorithms for sparse matrices. *SIAM J. Numer. Anal.*, 13:198–213, 1976.

[13] M. Metcalf and J. Reid. *Fortran 90 Explained.* Oxford Science Publications, Oxford, England, 1990.

[14] E. Ng and B.W. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor computers. *SIAM J. Sci. Comput.*, 14:1034–1056, 1993.