DAEPACK

Block Solver Manual

Version 1.0

Solution of Large Sparse Sets of Nonlinear Algebraic Equations

John E. Tolsma

December 2, 2000

Contents

This manual describes how to use the block solver of the DAEPACK numerical library for obtaining solutions of systems of nonlinear algebraic equations. The solver is particularly well-suited for large sparse problems. A description of how to use the components of the DAEPACK symbolics library to generate automatically the additional information (e.g., sparsity pattern and analytical derivatives) is provided. This capability reduces the difficulty associated with this often difficult calculation to simply providing a subroutine representing the model equations and a reasonable initial guess for the solution. The special case of consistent initialization of differential/algebraic equations (DAEs) is also described.

1 Overview

Finding a solution to a set of nonlinear algebraic equations is a common and often extremely difficult task. This problem can be stated mathematically as, finding an $x^* \in \mathcal{D} \subset \mathbb{R}^n$ such that

$$F(x^*) = 0 \tag{1}$$

where $F : \mathcal{D} \longrightarrow \mathbb{R}^n$. This problem is complicated by the fact that our initial estimate of the solution, x^o, is often "far" from x^* and F is typically large and highly nonlinear. Further, there may be a specific region, $\mathcal{D} \subset \mathbb{R}^n$, where we seek a solution. (Often, these are simply compact regions defined by upper and lower bounds on the individual components of x.) Several approaches exist for this problem, including the traditional Newton-Raphson method, quasi-Newton methods, the Levenberg-Marquardt method, homotopy continuation, successive linear programming, and many others. There are trade-offs between using methods that are efficient, but unreliable when started far from the solution (e.g., Newton-Raphson and many quasi-Newton methods), and approaches that are more robust yet costly (e.g., Interval Newton/ Generalized bisection and techniques based on the ideas of global optimization). Moreover, all of these approaches typically require additional information other than the model residuals, F, in particular, first and possibly higher order partial derivatives. In addition, if the problem is large and sparse, significant improvements can be realized by exploiting sparsity. However, this requires an explicit representation of the sparsity pattern of the system of equations.

This document describes how structural information can be exploited in order to decompose the original problem (assumed to be large, yet sparse) into a set of smaller problems where appropriate algorithms can be selected based on the size of the subproblem and difficulty to converge. By solving a sequence of smaller problems, we can often apply solution algorithms that are not practical for the full system of equations due to computational complexity or memory limitations.

2 Block Solution of a System of Equations

As stated above, we assume in this document that the system of equations of interest, F, is large but has a sparse Jacobian matrix. Further, we assume that the sparsity pattern (or incidence matrix) is reducible. That is, we can find row and column permutations of the incidence matrix such that the permuted system is block lower triangular and there are several square diagonal blocks, each with dimension much smaller than the original system. This is often true in many practical problems, particularly in chemical process simulation, where the approach described in this manual is used successfully in many modern equation-oriented process modeling environments. The authors hope that by providing a general set of tools, these benefits can be extended to a wide variety of disciplines. The contributions in this document include extending this approach to models coded in a programming language such as Fortran by providing symbolic tools that extract automatically the information required when solving the original system as a sequence of smaller blocks. It is worthwhile to note here that the symbolic tools of DAEPACK have been designed to work with very general Fortran codes for computing systems of equations, including loops, external procedure and function calls, conditional branching, and embedded iterative algorithms.

Suppose we are interested in solving the following system of equations,

$$F(x) = 0, \tag{2}$$

given an initial guess, x^o, where the incidence matrix of F is large and sparse. Using standard structural algorithms, we can find permutation matrices P and Q such that the incidence matrix of the original

system of equations, denoted by A, is in block lower triangular form (provided A is reducible) as shown in Figure 1. As illustrated in Figure 2, the diagonal blocks are solved sequentially, starting with the top left

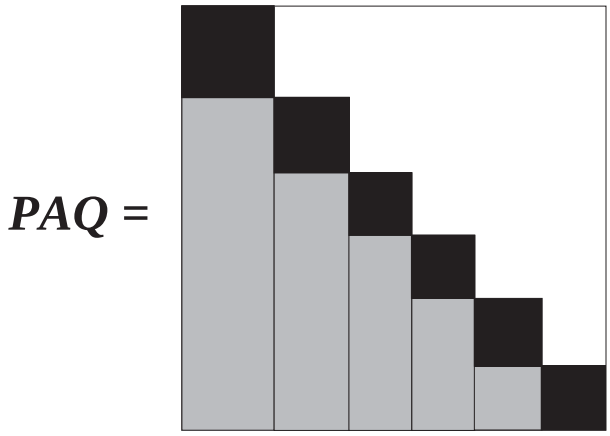

Figure 1: Incidence matrix A permuted to block lower triangular form. The black region denotes the diagonal blocks, the grey denotes areas where there may be nonzero entries, and the white region contains only zeros.

diagonal block, and proceeding down the diagonal until all blocks are solved and the desired solution of the full system is obtained. Since only the smaller diagonal blocks are solved, a more robust algorithm can be employed (depending on the block size and difficulty to converge) that would otherwise be unsuitable (e.g., too costly in terms of temporal and/or spatial complexity) for the full system of equations, or upon fixing the variables determined by an earlier block, some blocks may become linear.

This process has been automated in the block solver component of DAEPACK. Using the block solver, along with the symbolic components of DAEPACK, the user must simply provide the model in the form of a Fortran subroutine and DAEPACK takes care of the rest. The Fortran code of the model may consist of several subroutines and functions, but it is assumed that a single subroutine is called to perform the model evaluation. The following section describes how to call the DAEPACK block solver, assuming the additional information (e.g., sparsity pattern and derivatives) are available. This is followed by a section describing how this additional information is generated automatically with DAEPACK simply given the user's original model.

3 Block Solver Description

The block solver component of DAEPACK, named `BLKSLV`, is currently available as a Fortran subroutine in a pre-compiled library. In future releases of DAEPACK, the block solver will be available in different forms, such as a CORBA or COM component. The block solver interface and comment header is shown below.

```
      SUBROUTINE BLKSLV(ICODE,INFO,RINFO,N,X,LOWER,UPPER,
     1   NE,IRLIST,JCLIST,LRW,RW,LIW,IW,RES,JAC,RPAR,IPAR)
C### 990616 J.E.TOLSMA COPYRIGHT MIT
```

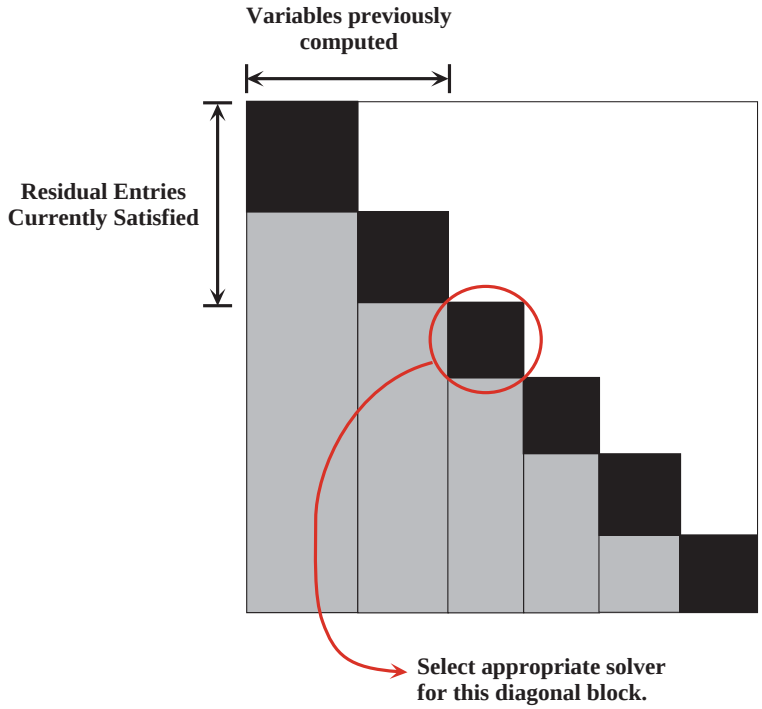

Figure 2: Solving large, sparse system as a sequence of smaller blocks.

```
C### THIS SUBROUTINE SOLVES A SYSTEM OF NONLINEAR EQUATIONS
C### AS A SEQUENCE OF SMALLER BLOCKS DETERMINED BY THE BLOCK
C### LOWER TRIANGULARIZATION OF THE JACOBIAN INCIDENCE MATRIX.
C###
C### ARGUMENTS (# INDICATES ARGUMENT MUST BE SET PRIOR TO CALL)
C### =========
C###     ICODE       - UPON INPUT THIS VALUE IS USED TO DETERMINE WHETHER
C###                   OR NOT BLOCK PERMUTATION IS TO BE PERFORMED:
C###                      ICODE =  0 - PERFORM BLOCK PERMUTATION
C###                            = -1 - DON'T BLOCK PERMUTE, SOLVE AS ONE
C###                                   LARGE BLOCK (SEE BELOW)
C###                   UPON COMPLETION, STATUS VARIABLE (SEE BELOW)
C### (#) INFO(*)      - INTEGER CONTROL PARAMETERS (SEE BELOW)
C### (#) RINFO(*)     - REAL CONTROL PARAMETERS (SEE BELOW)
C### (#) N           - DIMENSION OF ENTIRE SYSTEM OF EQUATIONS
C### (#) X(N)        - INITIAL GUESS FOR SOLUTION
C###                   UPON SUCCESSFUL COMPLETION, CONTAINS SOLUTION
C### (#) LOWER(N)    -  LOWER BOUNDS ON VARIABLES
C### (#) UPPER(N)    -  UPPER BOUNDS ON VARIABLES
C###                   A VALUE OF X OUTSIDE ITS BOUNDS WILL NOT
```

```
C###                   BE PASSED TO THE RESIDUAL ROUTINE (IT WILL BE
C###                   ADJUSTED TO SOME VALUE BETWEEN THE VIOLATED
C###                   BOUNDS).
C### (#) NE          - NUMBER OF NONZEROS IN JACOBIAN MATRIX
C###                   OF ENTIRE SYSTEM
C### (#) IRLIST(NE)  - ROW OCCURRENCE INFORMATION FOR JACOBIAN
C### (#) JCLIST(NE)  - COLUMN OCCURRENCE INFORMATION FOR JACOBIAN
C### (#) LRW         - AMOUNT OF REAL WORKSPACE (SEE BELOW)
C###     RW(LRW)     - REAL WORKSPACE
C### (#) LIW         - AMOUNT OF INTEGER WORKSPACE (SEE BELOW)
C###     IW(LIW)     - INTEGER WORKSPACE
C### (#) RES         - RESIDUAL ROUTINE (SEE BELOW)
C### (#) JAC         - JACOBIAN ROUTINE (SEE BELOW)
C###     RPAR(*)     - WORKSPACE - MUST BE OF DIMENSION AT LEAST N
C###                   FIRST N ENTRIES HOLD CURRENT X, REMAINING
C###                   ENTRIES CAN CONTAIN USER-DEFINED PARAMETERS.
C###                   THIS ARRAY IS PASSED TO THE RESIDUAL AND JACOBIAN
C###                   ROUTINES.
C###     IPAR(*)     - WORKSPACE - MUST BE OF DIMENSION AT LEAST 1
C###                   FIRST ENTRY WILL HOLD N - DIMENSION OF FULL SYSTEM.
C###                   REMAINING ENTRIES CAN CONTAIN USER-DEFINED PARAMETERS.
C###                   THIS ARRAY IS PASSED TO THE RESIDUAL AND JACOBIAN
C###                   ROUTINES.
C###
C### ** STATUS VARIABLE
C### ARGUMENT ICODE SERVES AS BOTH AN INPUT AND AN OUTPUT VARIABLE.  UPON
C### ENTERING BLKSLV, THE VALUE OF ICODE IS USED TO DETERMINE WHETHER OR
C### NOT BLOCK PERMUATION IS TO BE PERFORMED.
C###
C### **SET ICODE EQUAL TO ZERO TO PERFORM BLOCK PERMUTATION**
C### **SET ICODE EQUAL TO ONE TO SOLVE SYSTEM AS ONE BLOCK **
C###
C### SINCE THE BLOCK PERMUTATION CURRENTLY DEPENDS ON HARWELL (HSL) ROUTINES, WHICH
C### MAY NOT BE AVAILABLE TO THE USER, BLKSLV ALLOWS THE BLOCK PERMUTATION TO BE
C### TURNED OFF.  ALTHOUGH THE INDIVIDUAL SOLVERS APPLIED TO THE BLOCKS CAN BE CALLED
C### INDEPENDENTLY OF BLKSLV, THIS FEATURE WAS ADDED SO THAT THE USER WITHOUT THE HSL
C### ROUTINES MAY SET UP A PROGRAM THAT CALLS BLKSLV WITHOUT BLOCK PERMUATION, THEN
C### WHEN HE/SHE OBTAINS THE REQUIRED HSL ROUTINES, THE BLOCK PERMUTATION CAN BE
C### PERFORMED, WITHOUT HAVING TO RE-WRITE THE EXISTING CODE.
C###
C### THE STATUS OF THE SOLUTION PROCEDURE IS RETURNED IN ICODE.
C### CURRENTLY, THE POSSIBLE VALUES ARE
C###    ICODE =  0 - SUCCESSFUL, X CONTAINS SOLUTION
C###            -1 - INVALID ARGUMENTS AND/OR INSUFFCIENT WORKSPACE
C###                 (ISSUED ERROR MESSAGE DESCRIBES VIOLAITON)
C###            -2 - FAILURE WHILE PERMUTING MATRIX INTO BLOCK
```

```
C###                 LOWER TRIANGULAR FORM.  ERROR MESSAGE ISSUED
C###                 WITHIN BLOCK ROUTINE DESCRIBES ERROR.
C###            -3 - FAILURE WHILE CONSTRUCTING OCCURRENCE
C###                 INFORMATION FOR CURRENT DIAGONAL BLOCK.
C###            -4 - FAILURE WHILE ATTEMPTING TO SOLVE CURRENT
C###                 DIAGONAL BLOCK.
C###
C### ** INFORMATION ARRAYS
C### ARRAYS INFO AND RINFO ARE USED TO PASS INTEGER AND REAL
C### INFORMATION TO THE BLOCK SOLVERS. SINCE THE USER MAY WISH
C### TO APPLY DIFFERENT SOLVERS TO EACH OF THE BLOCKS, THE
C### SOLVERS SHOULD BE DESIGNED TO ACCEPT THE SAME INFORMATION
C### IN THESE CONTROL ARRAYS. THE ONLY INFORMAITON USED IN THIS
C### SUBROUTINE ARE
C###
C###    INFO(1) = LP     - PRINT UNIT
C###    INFO(2) = PLEVEL - PRINT LEVEL
C###              PLEVEL = 0 NO MESSAGES
C###                     = 1 ERROR MESSAGES
C###                     = 2 ERROR AND WARNING MESSAGES
C###                     = 3 ERROR, WARNING, AND SOME INFORMATION
C###                     > 3 ERROR, WARNING, AND MORE INFORMATION
C###
C### SOLVERS SHOULD ALSO EXPECT THIS INFORMATION IN THESE LOCATIONS
C### TO AVOID UNNECCESSARY CONFUSION.
C###
C### ** INTEGER AND REAL WORKSPACE
C### INTEGER AND REAL WORKSPACE IS PASSED TO THIS ROUTINE IN TWO
C### ARRAYS, IW AND RW, RESPECTIVELY.  ALL PARTITIONING OF THIS
C### WORKSPACE IS PERFORMED WITHIN THIS ROUTINE.  THE AMOUNT OF
C### INTEGER AND REAL WORKSPACE DEPENDS ON THE SIZE OF THE BLOCKS,
C### THE SOLVER APPLIED TO EACH BLOCK, THE LINEAR ALGEBRA ROUTINES
C### USED (E.G., DENSE, BANDED, SPARSE), ETC. A DESCRIPTION OF THE
C### WORKSPACE REQUIREMENTS FOR DIFFERENT COMBINATIONS OF SOLVERS
C### AND LINEAR ALGEBRA PACKAGES IS SUMMARIZED BELOW.
C###
C### FOR SOLVER *NWTSLV* AND LINEAR ALGEBRA ROUTINES *SPFACT*,
C### *SPSING*, AND *SPBACK*,
C###
C###     LRW >= 11*MAXNC + 4*MAXNZ + N + 5
C###     LIW >= NB + 15*MAXNC + 16*MAXNZ + 2*N + 31
C###
C### WHERE MAXNC IS THE DIMENSION OF THE LARGEST BLOCK, MAXNZ IS
C### THE LARGEST NUMBER OF NONZERO ENTRIES IN ANY OF THE BLOCKS,
C### NB IS THE NUMBER OF BLOCKS, AND N IS THE TOTAL NUMBER OF
C### EQUATIONS IN THE ENTIRE SYSTEM.  DEPENDING ON THE SIZE OF
```

```
C### THE BLOCK, IT MAY BE MORE ADVANTAGEOUS TO APPLY DENSE LINEAR
C### ALGEBRA.  MEMORY AND OVERHEAD ASSOCIATED WITH SPARSE
C### OPERATIONS WILL BE REDUCED IF DENSE LINEAR ALGEBRA IS APPLIED
C### TO BLOCKS WITH FEWER THAN MINNC EQUATIONS (WHERE MINNC IS TO
C### BE DETERMINED THROUGH EXPERIMENTATION).
C###
C### ** RESIDUAL AND JACOBIAN SUBROUTINES
C### THE USER PASSES THE MODEL TO THIS SOLVER IN THE FORM OF A
C### RESIDUAL SUBROUTINE AND A CORRESPONDING JACOBIAN SUBROUTINE.
C### THE INTERFACES OF THESE ROUTINES ARE SHOWN BELOW.
C###
C###     SUBROUTINE RES(ICODE,NC,XC,FC,X,IPAR,IBLK)
C###
C###     SUBROUTINE JAC(ICODE,NC,XC,FC,NZC,DF,IRLIST,JCLIST,
C###                    RPAR,IPAR,IBLK)
C###
C### WHERE ICODE IS A STATUS VARIABLE (A NONZERO RETURN VALUE WILL
C### CAUSE THIS SUBROUTINE TO TERMINATE WITH AN ERROR RETURN), NC
C### IS THE DIMENSION OF THE CURRENT BLOCK, XC(NC) ARE THE VARIABLES
C### ASSOCIATED WITH THE CURRENT BLOCK, FC(NC) ARE THE RESIDUALS
C### ASSOCIATED WITH THE CURRENT BLOCK, (DF(NZC),IRLIST(NZC),
C### JCLIST(NZC)) IS THE SPARSE TRIPLET REPRESENTATION OF THE
C### JACOBIAN MATRIX ASSOCIATED WITH THE CURRENT BLOCK (DF/DXC),
C### RPAR(*) AND IPAR(*) ARE USER-DEFINED PARAMETER WORKSPACES.
C### RPAR MUST CONTAIN AT LEAST N ENTRIES AND IPAR MUST CONTAIN AT
C### LEAST ONE ENTRY.
C###
C### *** THE FIRST N-ENTRIES OF RPAR WILL HOLD  X(N), THE CURRENT
C### *** VALUES OF THE VARIABLES ASSOCIATED WITH THE ENTIRE SYSTEM
C### *** AND IPAR(1)=N. THE REMAINING ENTRIES MAY CONTAIN USER-DEFINED
C### *** PARAMETERS USED DURING RESIDUAL AND JACOBIAN EVALUATION.
C###
C### FINALLY, IBLK CONTAINS THE INFORMATION ASSOCIATED WITH
C### THE BLOCK TRIANGUARIAZTION.  IBLK IS USED TO INFORM THE
C### RESIDUAL AND JACOBIAN ROUTINES WHICH COMPONENTS OF THE ENTIRE
C### SYSTEM ARE REQUIRED.
C###
C###      IBLK(1)                 - CURBLK - CURRENT BLOCK
C###      IBLK(2)                 - NB - NUMBER OF DIAGONAL BLOCKS
C###      IBLK(2+1:2+NB)          - LENB(1:NB) DIMENSION OF BLOCKS
C###      IBLK(2+NB+1:2+NB+N)     - IP(1:N) ROW PERMUTATION INFO
C###      IBLK(2+NB+N+1:2+NB+N+N) - IQ(1:N) COLUMN PERMUTATION INFO
C###
C### THIS INFORMATION IS USED AS FOLLOWS.  LET 1 <= CURBLK <= NB.
C###
C###      N       = IPAR(1)       ! GET DIMENSION OF ENTIRE SYSTEM
```

```
C###      CURBLK  = IBLK(1)       ! GET CURRENT DIAGONAL BLOCK NUMBER
C###      NB      = IBLK(2)       ! GET TOTAL NUMBER OF DIAGONAL BLOCKS
C###      LOCLENB = 3             ! WORKSPACE POINTER FOR LENB(1:NB)
C###      LOCIP   = LOCLENB + NB  ! WORKSPACE POINTER FOR IP(1:N)
C###      LOCIQ   = LOCIP   + N   ! WORKSPACE POINTER FOR IQ(1:N)
C###      OFFSET=0                ! COMPUTE OFFSET FOR CURRENT BLOCK
C###      DO I=1,CURBLK-1
C###         OFFSET=OFFSET+IBLK(LOCLENB+I-1)
C###      END DO
C###
C### FILL FC(1:NC) WITH ENTRIES WITH INDICES
C###
C###      IBLK(LOCIP+OFFSET:LOCIP+OFFSET+NC)
C###
C### IN RESIDUAL VECTOR OF ENTIRE SYSTEM. THE JACOBIAN IS THE PARTIAL
C### DERIVATIVES OF RESIDUALS ABOVE WITH RESPECT TO VARIABLES WITH
C### INDICIES
C###
C###      IBLK(LOCIQ+OFFSET:LOCIQ+OFFSET+NC)
C###
C### ONLY NONZERO JACOBIAN ENTRIES ARE STORED WITHIN DF.  FURTHERMORE,
C### THE OCCURRENCE INFORMATION CONTAINED IN IRLIST AND JCLIST IS
C### LOCAL TO THE CURRENT BLOCK (I.E., ROW AND COLUMN INDICES RANGE
C### FROM 1 TO NC), HOWEVER, THE VALUE FOR OFFSET ALONG WITH IP AND
C### IQ CAN BE USED TO MAP BACK AND FORTH FROM GLOBAL (ENTIRE JACOBIAN)
C### AND LOCAL (CURRENT BLOCK) OCCURRENCE INFORMATION.
C###
```

The code excerpt above describes the arguments of `BLKSLV`. These arguments are summarized in Table 1. Most of the information in Table 1 pertains to the original system of equations before permuted into block lower triangular form: the dimension `n`, vectors `x(n)`, `lower(n)`, and `upper(n)`, and occurrence information `ne`, `irlist(ne)`, and `jclist(ne)` all correspond to the entire system of equations. The only exceptions are the external subroutines `res` and `jac`. These are described in detail in the following sections. Argument `icode` is an input and an output variable. Upon input, the value is used to specify whether or not block permutation is to be performed. If `icode` is set to -1 then block permutation is not performed and the entire systems is solved as a single block. If `icode` is set to 0, then block permuation is performed and the system is solved as a sequence of smaller blocks. This option is described in more detail below. Upon returning from `BLKSLV`, `icode` serves as a status variable (see code section above). The information arrays `info(*)` and `rinfo(*)` are control parameters allowing the user to tailor the solution process and displayed information. The only entries used directly by BLKSLV (as opposed to being used by solvers called by BLKSLV to solve diagonal blocks) are `info(1)` and `info(2)`. Parameter `info(1)` is set by the user to the print unit used for diagnostic, warning, and error messages (e.g, `info(1)` is set to 6 to print to the terminal). Parameter `info(2)` is set by the user to control the amount of printing performed: a zero value turns printing off and larger values increase printing levels. These two entries are also passed to the individual block solvers for the same purpose. Additional entries of `info` and `rinfo` are described in the solver sections below. Arguments of `BLKSLV` that may cause confusion are `rpar` and

ipar. These arrays serve two purposes: 1) they are used to pass information about the entire system to the block residual and Jacobian evaluators (res and jac) and the remaining entries are used to pass user-defined real and integer parameters to these same routines.

Important: *If you need to pass real parameters to* res *and* jac *then these are placed in* rpar(n+1,n+2,...). *Similarly, if you need to pass integer parameters to* res *and* jac *then these are placed in* ipar(2,3,...).

This is described in more detail in Sections 3.1 and 3.2.

As described above, argument icode is initially used to determine whether or not block permuation is to be performed, i.e., setting icode equal to -1 turns off block permutation and the system is solved as a single block, whereas when icode is set equal to 0 the system is block permuted and solved as a sequence of smaller blocks. The reason for this is that BLKSLV requires certain Harwell routines for performing the block permutation and these routines, which require a license, may not be available to all users. If they are not available, then the block permuation feature can be switched off. This feature can easily be turned back on at a later time without modifying the existing code when these routines are available.

3.1 Block Residual Evaluator

As mentioned several times in this manual, the block solver exploits problem structure in order to solve the system of equations of interest in a more robust and efficient manner. This requires the user to provide more detailed information about the problem than what would be needed for less sophisticated algorithms. In particular, the residual evaluator passed to BLKSLV must be able to evaluate specific subsets of the model equations in a particular order (the equations corresponding to the current block being solved). The interface to the user-supplied residual routine is shown below:

```
subroutine res(icode,nc,xc,fc,rpar,ipar,iblk)
```

where icode is a status variable set to zero to indicate a successful residual evaluation and nonzero otherwise (a nonzero return value will result in an immediate termination of the solver), nc is the dimension of the current diagonal block for which residuals are requested, xc(nc) is the current values of the variables associated with the current block, fc(*) must be set to specific residual entries corresponding to the information in iblk (however, fc(*) contains enough room to hold the residuals of the entire system of equations, i.e., it is n entries in length), rpar(*) contains the current values of the entire variable vector in the first n entries and the remaining entries hold user-supplied real parameters (see description of rpar above), ipar(*) contains the dimension of the entire system of equations, n, in the first position and the remaining entries hold user-supplied integer parameters (see description of ipar above), and iblk holds the block information. The last argument, iblk, is used by res to determine which residual entries are desired and in which order and contains the following information:

```
iblk(1)                 - current block (curblk)
iblk(2)                 - number of diagonal blocks (nb)
iblk(2+1:2+nb)          - lenb(1:nb) dimension of diagonal blocks
iblk(2+nb+1:2+nb+n)     - ip(1:n) row permutation information for overall system
iblk(2+nb+n+1:2+nb+n+n) - iq(1:n) column permutation information for overall system
```

The Fortran code segment below shows how the block residual information can be extracted from iblk and ipar(1):

```
n       = ipar(1)       ! get dimension of entire system
curblk  = iblk(1)       ! get current diagonal block number
nb      = iblk(2)       ! get total number of diagonal blocks
loclenb = 3             ! workspace pointer for lenb(1:nb)
locip   = loclenb + nb  ! workspace pointer for ip(1:n)
lociq   = locip   + n   ! workspace pointer for iq(1:n)
offset=0                ! compute offset for current block
do i=1,curblk-1
   offset=offset+iblk(loclenb+i-1)
end do
```

The vector `fc(1:nc)` must contain residual entries in the entire system of equations with the following indices:

```
iblk(locip+offset:locip+offset+nc-1).
```

In other words, suppose in the code segment below `f(1:n)` contains the residual of the full system of equations in their original, un-permuted order:

```
do i=1,nc
   fc(i)=f(iblk(locip+offset+i-1))
end do
```

then `fc(1:nc)` contains the appropriate residual entries. The arguments for `res` are summarized in Table 2 below.

The ability to extract particular residual entries from the entire system of equations may seem like a difficult task. However, the symbolic components of DAEPACK can be used to construct automatically this information from the user-supplied code returning the entire system of equations.

3.2 Block Jacobian Evaluator

Similar to the residual evaluator described above, the Jacobian evaluator passed to `BLKSLV` must be able to evaluate specific subsets of the Jacobian in a particular order (the Jacobian corresponding to the current block being solved). The interface to the user-supplied Jacobian routine is shown below.

```
subroutine jac(icode,nc,xc,fc,nzc,df,irlist,jclist,rpar,ipar,iblk)
```

where `icode` is a status variable set to zero to indicate a successful Jacobian evaluation and nonzero otherwise (as above, a nonzero return value will result in immediate termination of the solver), `nc` is the dimension of the current diagonal block, `xc(nc)` is the current values of the variables associated with the current block, `fc(*)` must be set to specific residual entries corresponding to the information in `iblk` (however, this vector contains enough room to hold the residuals of the entire system of equations), `nzc` contains the number of nonzero entries expected in the Jacobian of the current block, `df(nzc)` must be set to the Jacobian of the current block using the information contained in `iblk`, `irlist(nzc)` and `jclist(nzc)` contain the sparsity pattern of the Jacobian for the current block, `rpar(*)` contains the current values of the entire variable vector in the first `n` entries and the remaining entries hold user-supplied real parameters (see description of `rpar` above), `ipar(*)` contains the dimension of the entire system of equations, `n`, in the first position and the remaining entries hold user-supplied integer parameters (see description of `ipar` above), and `iblk` holds the block information. Using the value of

`offset` computed using the Fortran code segment shown above, the values returned in `df(nzc)` must be the nonzero entries of the Jacobian corresponding to partial derivatives of equations with indices

```
iblk(locip+offset:locip+offset+nc-1)
```

with respect to variables with indices

```
iblk(lociq+offset:lociq+offset+nc-1)
```

This may seem like an even more daunting task than evaluating specific subsets of equations of the original model. However, again, using the symbolic components of DAEPACK, this Jacobian routine can be constructed automatically (returning analytical derivatives) using simply the Fortran code corresponding to the original system of equations! Table 3 contains a summary of the arguments passed to the Jacobian routine.

4 Generating the Additional Information with DAEPACK

After reading the documentation above, the user may question whether or not the additional information required by the block solver (in particular, the block residual and Jacobian evaluators) is too difficult to generate in order to make this DAEPACK component practical. After all, simply coding and validating a Fortran subroutine returning the residuals of the original system of equations is a difficult task by itself. The block solver requires subroutines that return specific subsets of model equations and partial derivatives. Fortunately, the symbolic components of DAEPACK can be used to construct automatically all of the information required by the block solver. The user must simply provide a set of one or more Fortran subroutines and functions returning the complete system of equations. The DAEPACK symbolic library distribution contains a makefile that takes this Fortran source file and calls DAEPACK to construct three new Fortran codes: one for determining the sparsity pattern of the original system of equations (used to obtain `ne`, `irlist(ne)`, and `jclist(ne)` passed to `BLKSLV`), another corresponding to block residual routine `res`, and a third corresponding to block Jacobian routine `jac`. These generated codes are then compiled into a shared library that can be linked into the application using `BLKSLV`. The makefile and supporting wrapper files provided with the symbolic library distribution expect the user to name the Fortran source file corresponding to the original model `res0.f` and the main subroutine called to evaluate the model residuals to have the following interface:

```
subroutine res0(icode,n,x,f,rparams,iparams)
```

where output `icode` is the status code described earlier, input `n` is the dimension of the full system of equations (full, un-permuted system), input `x(n)` contains the current values of the variables, output `f(n)` must be set to the values of the residuals of the model equations, and `rparams(*)` and `iparams(*)` are double precision and integer parameter arrays, respectively. Array `rparams` is set by the user in the code calling `BLKSLV` by specifying values for `rpar(n+1,n+2,...)`, i.e., entry `rparams(k)` in the residual routine is entry `rpar(n+k)` in the code calling `BLKSLV`. Similarly, array `iparams` is set by the user in the code calling `BLKSLV` by specifying values for `ipar(2,3,...)`, i.e., entry `iparams(k)` in the residual routine is entry `ipar(1+k)` in the code calling `BLKSLV`. This is due to the fact that `rpar` and `ipar` serve two purposes: to pass both user-defined parameters and information about the overall system of equations to the block residual and Jacobian routines. The main subroutine returning the residuals of the model equations may contain arbitrarily complex code and call any number of additional subroutines

and/or functions. The name of the file containing the Fortran source code and the name of the residual subroutine may be changed by the user provided the makefile and/or wrapper routines are updated appropriately. The process of generating the additional code is illustrated in Figure 4. The user must

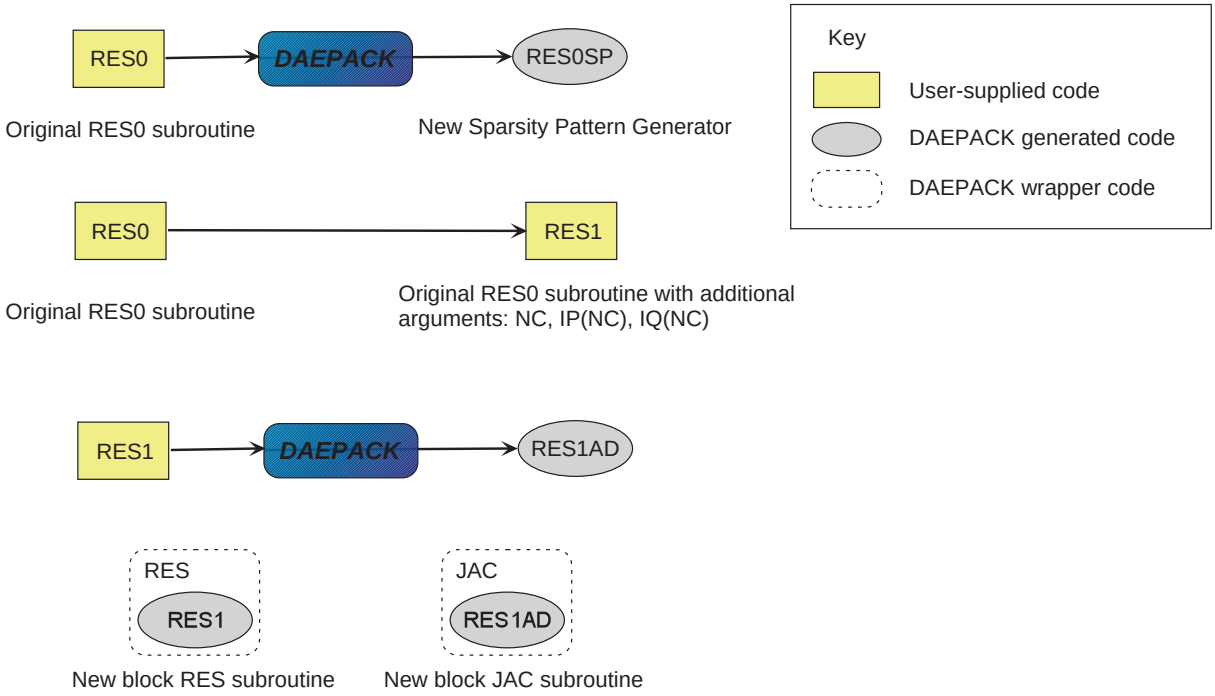

Figure 3: Automatic generation of additional code for BLKSLV using DAEPACK.

simply concentrate on coding and validating the code corresponding to the original system of equations and DAEPACK takes care of the rest. The remainder of this section describes the code generated for a small example problem.

Consider the small system of nonlinear algebraic equations obtained from [3]:

$$
\begin{aligned}
f_1 &= x_1 + x_4 - 10 = 0 && (3)\\
f_2 &= x_2^2 x_4 x_3 - x_5 - 6 = 0 && (4)\\
f_3 &= x_1 x_2^{1.7}(x_4 - 5) - 8 = 0 && (5)\\
f_4 &= x_4 - 3x_1 + 6 = 0 && (6)\\
f_5 &= x_1 x_3 - x_5 + 6 = 0. && (7)
\end{aligned}
$$

This system is coded as a Fortran subroutine shown below and saved to a file name `res0.f`.

```
      subroutine res0(icode,n,x,f,rparams,iparams)
c
c     this is the main residual routine, which computes the
c     full residual vector.
```

```
c
      implicit none
      integer icode,n,iparams(1)
      double precision x(n),f(n),rparams(1)
c
      f(1)=x(1)+x(4)-10.0d0
      f(2)=x(2)**2.0d0*x(3)*x(4)-x(5)-6.0d0
      f(3)=x(1)*x(2)**1.7d0*(x(4)-5.0d0)-8.0d0
      f(4)=x(4)-3.0d0*x(1)+6.0d0
      f(5)=x(1)*x(3)-x(5)+6.0d0
c
      return
      end
```

This file was placed into a directory containing a makefile, provided with the DAEPACK symbolic distribution, for generating the additional code used by the block solver `BLKSLV`. The block residual routine generated is shown below.

```
      subroutine res(icode,nc,xc,fc,x,ipar,iblk)
c###
c###  this routine is called during the block solution of a system
c###  of nonlinear equations to extract the nc entries of the residual
c###  vector corresponding to the current nc-dimensional block.  the
c###  residual routine is treated as a 'black-box', i.e., it is assumed
c###  that the user does not have access to individual residual components
c###  within the subroutine and the full set of residual is computed.
c###  although this is inefficient, it will likely be the case when dealing
c###  with legacy code.
c###
      implicit none
      integer icode,nc,ipar(*),iblk(*)
      double precision xc(nc),fc(*),x(*)
      external res0
      external updvars,adjres
      integer n
c###
c### get dimension of full system from workspace
      n=ipar(1)
c###
c### update elements in full variable vector (stored in x(1:n))
c### with the iteration variable vector for this block.
      call updvars(nc,xc,n,x,iblk)
c###
c### compute full residual vector from user-supplied code
      call res0(icode,n,x,fc,x(n+1),ipar(2))
c###
c### call subroutine to move entries of full residual corresponding to
```

```
c### current block to the first nc entries of fc.
      call adjres(iblk(1),iblk(2),iblk(3),iblk(3+iblk(2)),nc,fc)
c###
      return
      end
```

Currently, DAEPACK evaluates the entire system of equations, then extracts the necessary components for the current block with a call to `adjres`, provided with the DAEPACK library. Future versions of DAEPACK will generate new code which selectively computes the desired components of the residual vector. Fortunately, the residual evaluation is typically not the dominating cost of the calculation (compared to the Jacobian evaluation and factorization), making the current implementation perfectly acceptable. This is not true for the Jacobian evaluation. However, the derivative code generated by DAEPACK will *accumulate* only the nonzero entries associated with the current block. The Jacobian code, created automatically, is shown below.

```
      subroutine jac(icode,nc,xc,fc,nzc,df,irlist,jclist,x,ipar,iblk)
      implicit none
      integer icode,nc,nzc,irlist(nzc),jclist(nzc),ipar(*),iblk(*)
      double precision xc(nc),x(*),fc(*),df(nzc)
c###
c### the generated code requires additional integer workspace not
c### required by the block solver. the integer array (ADIW) must be
c### dimensioned to a number greater than or equal to nc or the number
c### of active variable arguments in any of the generated routines
c### (see the DAEPACK code generation manual for additional information).
c###
c### allocate static array and make sure to adjust the size depending
c### on the size of the particular problem.
c###
      integer LADIW
      parameter(LADIW=100)
      integer ADIW(LADIW)
      external res1ad
c###
      integer n,ioffset,ipoffset,iqoffset,icmpofst
      external icmpofst
      external updvars,adjres
c###
      if(LADIW.lt.nc) then
         write(*,*) '!! insufficient workspace in jac'
         stop
      end if
c###

c### store dimension of full system in n
      n=ipar(1)
```

```
c### get offset in iblk for permutation matrices iq and iq for
c### for current block
      ioffset=icmpofst(iblk(1),iblk(2),iblk(3))
      ipoffset=ioffset+2+iblk(2)+1
      iqoffset=ipoffset+n
c### row  indices  for current block are: iblk(ipoffset:ipoffset+nc-1)
c### column indices for current block are: iblk(iqoffset:iqoffset+nc-1)
c###
c### update elements in full variable vector (stored in rpar(1:n))
c### with the iteration variable vector for this block.
      call updvars(nc,xc,n,x,iblk)
c###
c### call DAEPACK generated derivative code here.
      call res1ad(icode,n,x,fc,x(n+1),ipar(2),nc,iblk(ipoffset),
     1   iblk(iqoffset),df,nzc,irlist,jclist,ADIW)
c###
c### call subroutine to move entries of full residual corresponding to
c### current block to the first nc entries of fc.
      call adjres(iblk(1),iblk(2),iblk(3),iblk(3+iblk(2)),nc,fc)
c###
      return
      end
c###
      integer function icmpofst(curblk,nb,lenb)
c###
c###  computes the offset in iblk (see above) corresponding to the
c###  row and column permutations of the current block.
c###
      implicit none
      integer curblk,nb,lenb(nb)
      integer i,ioffset
c###
c### get offset into block information for permutation information for
c### the current block
      ioffset=0
      do i=1,curblk-1
         ioffset=ioffset+lenb(i)
      end do
c###
      icmpofst=ioffset
c###
      return
      end
```

Subroutine `jac` returns the residual values and nonzero entries of the Jacobian associated with the current block. Function `icmpofst` computes the offset into the block information `iblk` for the row and column

indices corresponding to the current block. The Jacobian subroutine above calls the DAEPACK generated derivative code shown below.

```
C###  - DAEPACK Version 1.0 - Copyright (C) MIT
C###  - DERIVATIVE COMPUTATION -
C###  -   GENERATED: Tue Nov  7 14:37:26 2000
C###  - VALID UNTIL: Fri Jun 01 00:00:00 2001
      subroutine res1ad(icode,n,x,f,rpar,ipar,nc,ip,iq,zzzderiv,
     $ zzzne,zzzirn,zzzjcn,zzziw)
!!independent { x(iq(1:nc)) }
!!dependent   { f(ip(1:nc)) }
      implicit none
      integer n
      integer icode
      integer nc
      double precision f(n)
      integer ip(nc)
      integer iq(nc)
      integer ipar(*)
      double precision x(n)
      double precision rpar(1)
C###  Additional arguments for partial derivative computation
C###     zzzderiv - partial derivative array stored in sparse matrix
C###                format, pattern contained in zzzirn and zzzjcn.
C###     zzzne    - number of entries in zzzderiv.
C###     zzzirn   - row numbers for entries in zzzderiv.
C###     zzzjcn   - column numbers for entries in zzzderiv.
      double precision zzzderiv(*)
      integer zzzne,zzzirn(zzzne),zzzjcn(zzzne)
C###     zzziw    - integer workspace array.
      integer zzziw(*)
C###    zzzimodel - unique key for this model.
      integer zzzimodel
C###  Active variable offsets and loop control variables
      integer foft
      integer xoft
      integer rparoft
      integer zzzn
      integer zzzm
      integer zzzi
      integer zzzindx
C###  Variable offsets for independent, dependent, and
C###  non-common block local active variables.
      foft=0
      xoft=foft+n
      rparoft=xoft+n
```

```
C###  Number of independent and dependent variables.
      zzzn=nc
      zzzm=nc
C###  Create unique identifier for this model.
      call GETMDLID(zzzimodel,'res1ad')
C###  Create SVM for functions/subroutines.
      call CREATESV(zzzimodel,1,zzzn)
      call CREATESV(zzzimodel,2,zzzn)
C###  Construct mapping between independent variable offsets and indices
      zzzindx=0
      do zzzi=1,nc
         zzzindx=zzzindx+1
         zzziw(zzzindx)=xoft+iq(zzzi)
      end do
C###  Initialize independent variable gradients to Cartesian basis vectors.
      call DSETIVCV(1,zzzn,zzziw)
C###
C###  Save SVMs of arguments
      zzziw(1)=1
      zzziw(2)=1
      zzziw(3)=1
C###  Save current offsets for use within embedded subroutine
      zzziw(4)=xoft
      zzziw(5)=foft
      zzziw(6)=rparoft
C###  Call modifed embedded subroutine
      call res0ad(icode,n,x,f,rpar,ipar,zzziw)
C###
C###  Construct derivative matrix and sparsity pattern before returning.
C###  Construct mapping between dependent variable offsets and indices
      zzzindx=0
      do zzzi=1,nc
         zzzindx=zzzindx+1
         zzziw(zzzindx)=foft+ip(zzzi)
      end do
      call DCPRVS(1,zzzm,zzziw,zzzderiv,zzzne,zzzirn,zzzjcn)
C###  Delete SVM for functions/subroutines.
      call DELETESV(zzzimodel,1)
      call DELETESV(zzzimodel,2)
C###  Done
      return
      end
C###
      subroutine res0ad(icode,n,x,f,rpar,ipar,zzziw)
      implicit none
      integer n
```

```
      integer icode
      double precision f(n)
      integer ipar(*)
      double precision x(n)
      double precision rpar(1)
C###  Additional arguments for partial derivative construction
C###     zzziw - integer workspace
      integer zzziw(*)
C###  Define elemenary variables and adjoints
      double precision zzzv1,zzzvbar1
      double precision zzzv2,zzzvbar2
      double precision zzzv3,zzzvbar3
      double precision zzzv4,zzzvbar4
      double precision zzzv5,zzzvbar5
      double precision zzzv6,zzzvbar6
      double precision zzzv7,zzzvbar7
      double precision zzzv8,zzzvbar8
      double precision zzzv9,zzzvbar9
      double precision zzzv10,zzzvbar10
      double precision zzzv11,zzzvbar11
C###
C###  Active variable offsets and loop control variables
      integer xoft
      integer foft
      integer rparoft
      integer zzzsvm1
      integer zzzsvm2
      integer zzzsvm3
C###
      zzzsvm1=zzziw(1)
      zzzsvm2=zzziw(2)
      zzzsvm3=zzziw(3)
C###
C###  Compute offsets for arguments and local non-common
C###  block active variables
      xoft=zzziw(4)
      foft=zzziw(5)
      rparoft=zzziw(6)
C###
C###
C###
C###  Compute elementary variables
      zzzv1=x(1)
      zzzv2=x(4)
      zzzv3=zzzv1+zzzv2
      zzzv4=10.0d0
```

```
      zzzv5=zzzv3-zzzv4
C###  Compute elementary partial derivatives
      zzzvbar5=1.0d0
      zzzvbar3=zzzvbar5
      zzzvbar2=zzzvbar3
      zzzvbar1=zzzvbar3
C###
      f(1)=zzzv5
C###
      call DSVM2(foft+1,zzzsvm2,
     $           zzzvbar2,xoft+4,zzzsvm1,
     $           zzzvbar1,xoft+1,zzzsvm1)
C###
C###  Compute elementary variables
      zzzv1=x(2)
      zzzv2=2.0d0
      zzzv3=zzzv1**zzzv2
      zzzv4=x(3)
      zzzv5=zzzv3*zzzv4
      zzzv6=x(4)
      zzzv7=zzzv5*zzzv6
      zzzv8=x(5)
      zzzv9=zzzv7-zzzv8
      zzzv10=6.0d0
      zzzv11=zzzv9-zzzv10
C###  Compute elementary partial derivatives
      zzzvbar11=1.0d0
      zzzvbar9=zzzvbar11
      zzzvbar8=-zzzvbar9
      zzzvbar7=zzzvbar9
      zzzvbar6=zzzvbar7*zzzv5
      zzzvbar5=zzzvbar7*zzzv6
      zzzvbar4=zzzvbar5*zzzv3
      zzzvbar3=zzzvbar5*zzzv4
      zzzvbar1=zzzvbar3*zzzv2*zzzv1**(zzzv2-1.0d0)
C###
      f(2)=zzzv11
C###
      call DSVM4(foft+2,zzzsvm2,
     $           zzzvbar8,xoft+5,zzzsvm1,
     $           zzzvbar6,xoft+4,zzzsvm1,
     $           zzzvbar4,xoft+3,zzzsvm1,
     $           zzzvbar1,xoft+2,zzzsvm1)
C###
C###  Compute elementary variables
      zzzv1=x(1)
```

```
      zzzv2=x(2)
      zzzv3=1.7d0
      zzzv4=zzzv2**zzzv3
      zzzv5=zzzv1*zzzv4
      zzzv6=x(4)
      zzzv7=5.0d0
      zzzv8=zzzv6-zzzv7
      zzzv9=zzzv5*zzzv8
      zzzv10=8.0d0
      zzzv11=zzzv9-zzzv10
C###  Compute elementary partial derivatives
      zzzvbar11=1.0d0
      zzzvbar9=zzzvbar11
      zzzvbar8=zzzvbar9*zzzv5
      zzzvbar6=zzzvbar8
      zzzvbar5=zzzvbar9*zzzv8
      zzzvbar4=zzzvbar5*zzzv1
      zzzvbar2=zzzvbar4*zzzv3*zzzv2**(zzzv3-1.0d0)
      zzzvbar1=zzzvbar5*zzzv4
C###
      f(3)=zzzv11
C###
      call DSVM3(foft+3,zzzsvm2,
     $           zzzvbar6,xoft+4,zzzsvm1,
     $           zzzvbar2,xoft+2,zzzsvm1,
     $           zzzvbar1,xoft+1,zzzsvm1)
C###
C###  Compute elementary variables
      zzzv1=x(4)
      zzzv2=3.0d0
      zzzv3=x(1)
      zzzv4=zzzv2*zzzv3
      zzzv5=zzzv1-zzzv4
      zzzv6=6.0d0
      zzzv7=zzzv5+zzzv6
C###  Compute elementary partial derivatives
      zzzvbar7=1.0d0
      zzzvbar5=zzzvbar7
      zzzvbar4=-zzzvbar5
      zzzvbar3=zzzvbar4*zzzv2
      zzzvbar1=zzzvbar5
C###
      f(4)=zzzv7
C###
      call DSVM2(foft+4,zzzsvm2,
     $           zzzvbar3,xoft+1,zzzsvm1,
```

```
     $           zzzvbar1,xoft+4,zzzsvm1)
C###
C###  Compute elementary variables
      zzzv1=x(1)
      zzzv2=x(3)
      zzzv3=zzzv1*zzzv2
      zzzv4=x(5)
      zzzv5=zzzv3-zzzv4
      zzzv6=6.0d0
      zzzv7=zzzv5+zzzv6
C###  Compute elementary partial derivatives
      zzzvbar7=1.0d0
      zzzvbar5=zzzvbar7
      zzzvbar4=-zzzvbar5
      zzzvbar3=zzzvbar5
      zzzvbar2=zzzvbar3*zzzv1
      zzzvbar1=zzzvbar3*zzzv2
C###
      f(5)=zzzv7
C###
      call DSVM3(foft+5,zzzsvm2,
     $           zzzvbar4,xoft+5,zzzsvm1,
     $           zzzvbar2,xoft+3,zzzsvm1,
     $           zzzvbar1,xoft+1,zzzsvm1)
C###
C###  Done
      return
      end
```

This generated code constructs the Jacobian using the forward mode of automatic differentiation. Calls are made to several routines contained in the DAEPACK utility library [1] which must be linked into the application using the generated code.

The generated code shown above is compiled into a library that can be linked into the application using the block solver. All the user must provide when using the block solver is the original residual subroutine returning the entire system of equations and a suitable initial guess.

5 The DAE Consistent Initialization Problem

A special, but very important, situation where the solution of a system of nonlinear algebraic equations is needed is for obtaining a consistent set of initial conditions for a system of differential and algebraic equations (DAEs). Failure to provide a consistent set of initial conditions will typically result in immediate failure when numerically integrating the DAE. Often, much of the time spent when performing numerical calculations with DAEs is finding such initial conditions (a user may spend hours or days computing initial conditions when the actual numerical integration may only take seconds). The difficulty of this problem (as well as solving DAEs in general) had inspired the initial development of DAEPACK (hence,

the name), however, it is hoped that the reader of this and other DAEPACK manuals will see applications for DAEPACK far beyond DAEs.

Let $f(\dot{x}, x, y, t) = 0$ denote the DAE of interest where $f : \mathbb{R}^{n_x} \times \mathbb{R}^{n_x} \times \mathbb{R}^{n_y} \times \mathbb{R} \longrightarrow \mathbb{R}^{n_x+n_y}$, $x(t)$ are the differential variables and are a function of time in an n_x-dimensional function space, $\dot{x}(t)$ are the associated time derivatives, $y(t)$ are the algebraic variables and are a function of time in an n_y-dimensional function space, and t is time (the independent variable). The consistent initialization problem can be stated mathematically as finding $\dot{x}(t_0)$, $x(t_0)$, and $y(t_0)$ such that

$$f(\dot{x}(t_0), x(t_0), y(t_0), t_0) = 0 \tag{8}$$

$$c(\dot{x}(t_0), x(t_0), y(t_0), t_0) = 0 \tag{9}$$

where $c(\dot{x}(t_0), x(t_0), y(t_0), t_0)$ are the initial conditions. Furthermore, we require

$$\left[\begin{array}{cc} \frac{\partial f}{\partial \dot{x}} & \frac{\partial f}{\partial y} \end{array} \right] \tag{10}$$

to be nonsingular. This is sufficient to ensure the DAE is index 1 (and thus standard numerical integration techniques can be employed) and exactly n_x additional equations (the dimension of c) are required to determine the consistent initial conditions. In the example above, we consider completely general initial conditions. Special cases would be finding steady-state initial conditions (i.e., $c(\dot{x}(t_0), x(t_0), y(t_0), t_0) = \dot{x}(t_0)$) or finding initial conditions for a specified initial state (i.e., $c(\dot{x}(t_), x(t_0), y(t_0), t_0) = x(t_0) - x_0$). Obviously, c must be selected such that equations (8)-(9) have a solution.

The consistent initialization problem may be solved using the DAEPACK block solver (a separate section in this manual is probably not needed to make the user aware of this). However, this section comments on how the residual routine used by DAEPACK numerical components DSL48E and DSL48S for numerically integrating DAEs can be converted into the form required by BLKSLV.

We assume that the user has coded a Fortran subroutine returning the residuals of the DAEs and this subroutine has the following interface:

```
subroutine dae(n,t,z,zprime,delta,ires,ichvar,rparams,iparams).
```

This is the form required by DSL48E/DSL48S, however, other DAE integrators will require residual routines with roughly the same argument list, thus, the description here is not limited to using DSL48E/DSL48S to numerically integrate the DAE. In the subroutine argument list above, `n` is the dimension of the DAE (i.e., `n`=$n_x + n_y$), `t` is time, `z(n)` is the vector of both differential and algebraic variables (with arbitrary ordering), `zprime(n)` is the vector of time derivatives (only n_x entries of `zprime` appear explicitly in the DAE), `delta(n)` is an output variable returning the DAE residuals, `ires` is a status variable (set to 0 to indicate a successful residual evaluation and nonzero otherwise), `ichvar` is an integer flag (not important for this discussion), and `rparams(*)` and `iparams(*)` are the usual double precision and integer parameter arrays. Next, we assume that the initial conditions are provided by a call to a subroutine with interface:

```
subroutine ics(n,t,z,zprime,delta,ires,ichvar,rparams,iparams).
```

where the arguments are the same as those above except that the initial conditions are returned in `delta`. Two complications exist when using these subroutines with the block solver:

1. The number of variables we are solving for is $2n_x + n_y$ ($\dot{x}(t_0)$, $x(t_0)$, and $y(t_0)$) and

2. The unknowns are `z` and the subset of elements in `zprime` that appear explicitly in the DAE and initial conditions.

We must be able to construct a new residual routine returning equations (8)-(9) in the form required by the block solver. This can be readily achieved by passing information to the residual routine through `iparams` and using `rparams` as workspace. This new subroutine is shown below:

```
      subroutine res0(icode,n,x,f,rparams,iparams)
      implicit none
      integer icode,n,iparams(*)
      double precision x(n),f(n),rparams(12)
      external dae,ics
      integer i,nx,ichvar
      double precision t0
c
c.....Get number of differential variables.
      nx=iparams(1)
c
c.....Get initial value for time
      t0=rparams(1)
c
c.....Map entries of x corresponding to time derivatives into workspace
c     rparams.
      do i=1,nx
         rparams(1+iparams(i+1))=x(n-nx+i)
      end do
c
c.....Evaluate DAE residuals (ichvar equal to 1 indicates variable values
c     have changed since last call to this routine).
      ichvar=1
      call dae(n-nx,t0,x,rparams(2),f,icode,ichvar,rparams(2+n-nx),
     1   iparams(2+nx))
c
c.....Append residual vector with initial conditions.
      call ics(n-nx,t0,x,rparams(2),f(n-nx+1),icode,ichvar,
     1   rparams(2+n-nx),iparams(2+nx))
c
      return
      end
```

The subroutine above requires the user to pass information through `rparams` and `iparams` in order to evaluate the equations defining the consistent initialization problem. The value of `n` passed to `res0` is equal to the dimension of the DAE plus initial conditions (i.e., $2n_x + n_y$, for the index 1 case). The number of differential variables, n_x, is stored in `iparams(1)`. Since the initial time may not always be zero (for example during reinitialization calculation), this value is stored in `rparams(1)`. The unknown vector, `z`, contains the variables x and y in the first $n_x + n_y$ positions and the variables $\dot{x}$ in the remaining n_x positions. We assume that any subset of n_x entries of `zprime` appear explicitly in the DAE (not simply

the first n_x entries of this vector). Consequently, we must map the last n_x entries of `z` into a work vector (in this case, `rparams(2,3,...,n+1)`) that is passed to subroutines `dae` and `ics` to provide the values of the time derivatives that appear explicitly. This mapping is contained in `iparams(2,3,...,nx+1)` and provides the following information: unknown `z(n-nx+k)` is time derivative `zprime(iparams(1+k))` that appears explicitly in the DAE. The additional information passed to this routine can be initialized by hand or extracted automatically from the information returned from the sparsity pattern of `dae` and `ics` where the dependent variables are `delta` and the independent variables are `z,zprime`. The subroutine shown above can be used to generate all of the additional code required to use the block solver using DAEPACK and the process described in the previous section.

The remainder of this section provides the DAE and initial condition subroutines for a small example. The model is a simple mass balance for a tank with a one inlet stream and one outlet stream where flow is restricted by a valve. The model is

$$\frac{dM}{dt} \equiv \dot{M} = F_{in} - F_{out} \tag{11}$$

$$M = \rho A h \tag{12}$$

$$F_{out} = K\sqrt{h} \tag{13}$$

where M is the mass of liquid in the tank, F_{in} and F_{out} are the inlet and outlet mass flowrates, respectively, h is the height of liquid in the tank, ρ is the liquid density, A is the cross-sectional area of the tank, and K is the valve constant. Variables ρ, A, and K are assumed to be constant and are stored in the parameter array `rparams`, allowing the user to change these values without modifying the residual routine. The initial condition for this DAE is simply

$$h(0) = 1. \tag{14}$$

The DAE subroutine is shown below.

```
      subroutine dae(n,t,z,zprime,delta,ires,ichvar,rparams,iparams)
      implicit none
      integer n,ires,ichvar,iparams(1)
      double precision t,z(n),zprime(n),delta(n),rparams(4)
c.....local variables
      double precision mdot,m,h,fout
      double precision a,fin,rho,k
c.....begin
c
c.....unpack workspace and set independent variables to local
c     variables to make model more readable.
      a=rparams(1)
      fin=rparams(2)
      rho=rparams(3)
      k=rparams(4)
c
      mdot=zprime(1)
      m=z(1)
      h=z(2)
```

```
      fout=z(3)
c
c.....model equations
      delta(1)=mdot-fin+fout
      delta(2)=m-rho*a*h
      delta(3)=fout-k*dsqrt(h)
c
c.....done
      return
      end
```

The initial condition subroutine is shown below.

```
      subroutine ics(n,t,z,zprime,delta,ires,ichvar,rparams,iparams)
      implicit none
      integer n,ires,ichvar,iparams(1)
      double precision t,z(n),zprime(n),delta(n),rparams(4)
c.....local variables
      double precision h
c.....begin
c
c.....set independent variables to local variables to make model more readable.
      h=z(2)
c
c.....initial conditions
      delta(1)=h-1.0d0
c
c.....done
      return
      end
```

The simple example above is meant to give the user a feel for what must be provided and can be used as a starting point for larger, more realistic problems.

6 DAEPACK Block Solvers

`BLKSLV` itself does not solve the system of equations, it simply takes the information about the system and determines a permutation such that the equations can be solved as a sequence of smaller problems. It then keeps track of information and calls an appropriate solver to solve the individual blocks. The criteria it currently uses for selecting a solver is the dimension of the block and difficulty to converge (i.e., it first attempts to solve the block using a fast, yet less robust method and if that fails after a certain number of iterations it tries a more robust solver). This section briefly describes some of the solvers provided. Each of the solvers described below can be called independently from the block solver. Future implementations of DAEPACK will allow the user to use his own solver for blocks with certain user-defined characteristics.

6.1 BSCSLV

Often, when a large sparse system of equations is permuted into block lower triangular form many of the diagonal blocks are single equations in terms of a single unknown. Our experience has shown that in some cases, failure to converge the full system of equations can be attributed to failing to converge a single equation. Consequently, the DAEPACK block solver contains a solver, based on *bisection*, specifically for these single equation blocks. The interface and description for the subroutine BSCSLV is shown below.

```
      SUBROUTINE BSCSLV(ICODE,INFO,RINFO,N,X,BOUNDS,F,
     $   NE,LDF,DF,IRLIST,JCLIST,IBLK,RPAR,IPAR,
     $   LRW,RW,LIW,IW,LRLAW,RLAW,LILAW,ILAW,
     $   RES,JAC,DUMMY1,DUMMY2,DUMMY3)
C### 990527 JOHN E. TOLSMA COPYRIGHT MIT
C### BSCSLV - BISECTION NONLINEAR EQUATION SOLVER.
C### DESCRIPTION: SOLVES A SINGLE NONLINEAR EQUATION USING
C### BISECTION.
C###
C### ARGUMENTS (# INDICATES ARGUMENT MUST BE SET PRIOR TO CALL)
C###           (* INDICATES ARGUMENT MAY NOT BE USED AND A SIMPLE
C###              DUMMY ARGUMENT WILL SUFFICE)
C###     ICODE       - SOLUTION STATUS (SEE BELOW)
C### (#) INFO(3)     - INTEGER SOLUTION PARAMETERS (SEE BELOW)
C### (#) RINFO(1)    - REAL SOLUTION PARAMETERS (SEE BELOW)
C### (#) N           - DIMENSION OF SYSTEM, N=1
C### (#) X(N)        - INITIAL GUESS FOR ITERATIVE METHOD.
C###                   UPON SUCCESSFUL COMPLETION, CONTAINS SOLUTION
C### (*) BOUNDS(2*N) - VARIABLE BOUNDS
C###     F(N)        - RESIDUAL VECTOR WORKSPACE
C### (#) NE          - NUMBER OF NONZERO ENTRIES IN JACOBIAN
C###                   MATRIX (LESS THAN N*N IF SPARSITY IS
C###                   EXPLOITED)
C###     DF(LDF)     - JACOBIAN WORKSPACE
C### (*) IRLIST(NE)  - ROW INDICIES (ONLY REQUIRED IF SPARSITY
C###                   IS EXPLOITED - SEE BELOW)
C### (*) JCLIST(NE)  - COLUMN INDICIES (ONLY REQUIRED IF SPARSITY
C###                   IS EXPLOITED - SEE BELOW)
C### (*) IBLK(*)     - BLOCK INFORMATION (SEE BELOW)
C### (*) RPAR(*)     - REAL PARAMETERS PASSED TO RES AND JAC
C### (*) IPAR(*)     - INTEGER PARAMETERS PASSED TO RES AND JAC
C### (#) RW(LRW)     - NEWTON SOLVER REAL WORKSPACE
C###                   LRW >= N + (AMOUNT REQUIRED FOR STEPSIZE ROUTINE)
C###                   FOR ARMIJO RULE STEPSIZE ALGORITHM, 2N IS REQUIRED.
C### (*) IW(LIW)     - NEWTON SOLVER INTEGER WORKSPACE
C###                   ** THIS ARGUMENT IS IGNORED SINCE INTEGER
C###                   ** WORKSPACE IS NOT USED IN CURRENT
C###                   ** IMPLEMENATION
C### (#) RLAW(LRLAW) - LINEAR ALGEBRA REAL WORKSPACE
```

```
C### (#) ILAW(LILAW) - LINEAR ALGEBRA INTEGER WORKSPACE
C### (#) RES         - RESIDUAL ROUTINE (SEE BELOW)
C### (#) JAC         - JACOBIAN ROUTINE (SEE BELOW)
C###     DUMMY1,
C###     DUMMY2,
C###     DUMMY3      - DUMMY ARGUMENTS USED TO KEEP INTERFACE CONSISTENT.
C###
C### ** SOLUTION PARAMETER ARRAYS
C### IN ORDER TO REDUCE ARGUMENT LIST COMPLEXITY AND KEEP THE INTERFACE
C### GENERIC, SOLUTION PARAMETERS ARE PASSED TO THIS ROUTINE THROUGH INTEGER
C### AND REAL PARAMETER ARRAYS.  THE ENTRIES OF THESE ARRAYS ARE DESCRIBED
C### BELOW.
C###
C### INFO - INTEGER PARAMETERS
C###
C### INFO(1) - OUTPUT UNIT FOR INFORMATION/WARNING/ERROR MESSAGES
C###              INFO(1) >= 0
C### INFO(2) - PRINT LEVEL
C###              INFO(2) = 0 - NO PRINTING
C###                        1 - ERROR MESSAGES
C###                        2 - ERROR AND WARNING MESSAGES
C###                        3 - ERROR, WARNING, SOME INFORMATION
C###                      > 3 - ERROR, WARNING, MORE INFORMATION
C### INFO(3) - MAXIMUM NUMBER OF ITERATIONS
C###              INFO(3) = 0 - DEFAULT NUMBER USED
C###
C### RINFO - REAL PARAMETERS
C###
C### RINFO(1) - ABSOLUTE ERROR USED FOR CONVERGENCE CRITERIA
C###            USED FOR BOTH NORM OF RESIDUALS AND NORM OF
C###            STEP
C###               RINFO(1) > EPS (MACHINE TOLERANCE)
C###
C### ** RETURN STATUS VARIABLE
C### THE COMPLETION STATUS OF THE SOLUTION PROCEDURE IS RETURNED
C### IN THE ICODE ARGUMENT.  POSSIBLE RETURN VALUES ARE:
C###
C###    ICODE =  0 - SUCCESSFULLY CONVERGED SYSTEM OF EQUATIONS
C###            -1 - INVALID INPUT - SOLVER PARAMETERS ARRAYS
C###                 INFO AND/OR RINFO
C###            -2 - FAILURE IN SUBROUTINE RES
C###            -3 - FAILURE IN SUBROUTINE JAC
C###            -4 - FAILURE IN LINEAR ALGEBRA ROUTINES
C###            -5 - GENERAL FAILURE - SEE ISSUED ERROR MESSAGE
C###                 FOR MORE DETAIL
C###             K - UNABLE TO CONVERGE AFTER K ITERATIONS
```

```
C###
C### ** EXTERNAL SUBROUTINES
C###
C### RES - RESIDUAL ROUTINE
C###
C###          SUBROUTINE RES(ICODE,N,X,F,RPAR,IPAR,IBLK)
C###
C###       IF SUCCESSFUL, ICODE = 0
C###
C### JAC - JACOBIAN ROUTINE
C###
C###          SUBROUTINE JAC(ICODE,N,X,F,NE,DF,IRLIST,JCLIST,
C###             RPAR,IPAR,IBLK)
C###
C###       IF SUCCESSFUL, ICODE = 0
C###
C### DESIGN PHILOSOPHY
C###
C### THIS SUBROUTINE WAS DESIGNED TO BE MULTPURPOSE AND HIGHLY EXTENDIBLE
C### AND FLEXIBLE. AN ATTEMPT TO MAKE THE INTERFACE AS GENERIC AS POSSIBLE
C### SO THAT THE ROUTINE CALLING BSCSLV MAY READILY EXCHANGE BETWEEN A WIDE
C### VARIETY OF SOLVERS WITH IDENTICAL GENERIC INTERFACES. THEN RETURN VALUES
C### THROUGH ICODE ARE ALSO KEPT GENERIC FOR THE SAME REASON, RELYING ON
C### ISSUED ERROR MESSAGES FOR MORE DETAILED INFORMATION.
C###
C### ** BLOCK DECOMPOSITON
C### BSCSLV HAS THE OPTION TO BE CALLED FROM A BLOCK SOLVER WHERE THE OVERALL
C### SYSTEM OF EQUATIONS HAS BEEN BLOCK LOWER TRIANGULARIZED AND SOLVED AS A
C### SEQUENCE OF SMALLER PROBLEMS. THIS DECOMPOSITION IS TRANSPARENT TO NWTSLV:
C### RES AND JAC RETURN THE PROPER SUBSETS OF THE RESIDUALS AND JACOBIAN MATRIX
C### AND NWTSLV TREATS THESE AS THOUGH THEY WERE A REGULAR SYSTEM OF EQUATIONS
C### AND NOT PART OF A LARGER BLOCK.  IN ORDER FOR RES AND JAC TO KNOW WHICH
C### SUBSETS TO RETURN, ARRAY IBLK(*) IS PASSED AS AN ARGUMENT.  A DESCRIPTION
C### OF THE CONTENTS OF THIS ARRAY IS OMMITTED HERE SINCE BSCSLV NEED NOT KNOW
C### ANYTHING OF THIS DECOMPOSITION.  RES AND JAC TAKE THIS AS ARRAY AS AN
C### ARGUMENT IN ORDER TO RETURN THE PROPER SUBSET OF THE OVERALL SYSTEM (SEE
C### DOCUMENATION OF BLOCK SOLVER FOR MORE INFORMATION) THIS ARGUMENT IS ONLY
C### REQUIRED IF THIS SUBROUTINE IS CALLED FROM A BLOCK SOLVER.
C###
```

6.2 NWTSLV

A solver library would not be complete without the Newton-Raphson method. The implementation in DAEPACK is called `NWTSLV` and contains the following enhancements over the traditional Newton-Raphson method:

1. Flexible incorporation of various linear algebra packages.
2. Bounds on variables.
3. Stepsize selection based on the Armijo Rule.
4. Optional quasi-Newton approach where the Jacobian is not factored when "close" to the solution.
5. Automatic recovery from singularities.

The interface and description of NWTSLV is shown below.

```
      SUBROUTINE NWTSLV(ICODE,INFO,RINFO,N,X,BOUNDS,F,NE,LDF,DF,
     1   IRLIST,JCLIST,IBLK,RPAR,IPAR,LRW,RW,LIW,IW,LRLAW,RLAW,
     2   LILAW,ILAW,RES,JAC,LUFACT,LUSING,LUBACK)
C### 990527 JOHN E. TOLSMA COPYRIGHT MIT
C### NWTSLV - NEWTON/QUASI-NEWTON NONLINEAR EQUATION SOLVER.
C### DESCRIPTION: SOLVES NONLINEAR SYSTEM OF EQUATIONS USING
C### NEWTON'S METHOD OR QUASI-NEWTON METHOD (DEFERRED JACOBIAN
C### UPDATE).  FEATURES: OPTIONAL VARIABLE BOUNDS, STEPSIZE
C### SELECTION STRATEGY (MODIFY SUBOUTINE 'STPSIZ' FOR CUSTOM
C### STEPSIZE STRATEGY), AND SINGULAR JACOBIAN HANDLING.
C###
C### ARGUMENTS (# INDICATES ARGUMENT MUST BE SET PRIOR TO CALL)
C###           (* INDICATES ARGUMENT MAY NOT BE USED AND A SIMPLE
C###              DUMMY ARGUMENT WILL SUFFICE)
C###     ICODE       - SOLUTION STATUS (SEE BELOW)
C### (#) INFO(5)     - INTEGER SOLUTION PARAMETERS (SEE BELOW)
C### (#) RINFO(2)    - REAL SOLUTION PARAMETERS (SEE BELOW)
C### (#) N           - DIMENSION OF SYSTEM
C### (#) X(N)        - INITIAL GUESS FOR ITERATIVE METHOD.
C###                   UPON SUCCESSFUL COMPLETION, CONTAINS SOLUTION
C### (*) BOUNDS(2*N) - (OPITIONAL) VARIABLE BOUNDS - SEE INFO ARRAY
C###     F(N)        - RESIDUAL VECTOR WORKSPACE
C### (#) NE          - NUMBER OF NONZERO ENTRIES IN JACOBIAN
C###                   MATRIX (LESS THAN N*N IF SPARSITY IS
C###                   EXPLOITED)
C###     DF(LDF)     - JACOBIAN WORKSPACE
C###                   USUALLY LDF >= 3*NE WILL SUFFICE
C### (*) IRLIST(NE)  - ROW INDICIES (ONLY REQUIRED IF SPARSITY
C###                   IS EXPLOITED - SEE BELOW)
C### (*) JCLIST(NE)  - COLUMN INDICIES (ONLY REQUIRED IF SPARSITY
C###                   IS EXPLOITED - SEE BELOW)
C### (*) IBLK(*)     - BLOCK INFORMATION (SEE BELOW)
C### (*) RPAR(*)     - REAL PARAMETERS PASSED TO RES AND JAC
C### (*) IPAR(*)     - INTEGER PARAMETERS PASSED TO RES AND JAC
C### (#) RW(LRW)     - NEWTON SOLVER REAL WORKSPACE
C###                   LRW >= N + (AMOUNT REQUIRED FOR STEPSIZE ROUTINE)
```

```
C###                   FOR ARMIJO RULE STEPSIZE ALGORITHM, 2N IS REQUIRED.
C### (*) IW(LIW)     - NEWTON SOLVER INTEGER WORKSPACE
C###                   ** THIS ARGUMENT IS IGNORED SINCE INTEGER
C###                   ** WORKSPACE IS NOT USED IN CURRENT
C###                   ** IMPLEMENATION
C### (#) RLAW(LRLAW) - LINEAR ALGEBRA REAL WORKSPACE
C### (#) ILAW(LILAW) - LINEAR ALGEBRA INTEGER WORKSPACE
C### (#) RES         - RESIDUAL ROUTINE (SEE BELOW)
C### (#) JAC         - JACOBIAN ROUTINE (SEE BELOW)
C### (#) LUFACT      - JACOBIAN FACTORIZATION ROUTINE (SEE BELOW)
C### (#) LUSING      - SINGULARTIY HANDLING ROUTINE (SEE BELOW)
C### (#) LUBACK      - BACKSUBSTITUTION HANDLING ROUTINE (SEE BELOW)
C###
C### ** SOLUTION PARAMETER ARRAYS
C### IN ORDER TO REDUCE ARGUMENT LIST COMPLEXITY AND KEEP THE INTERFACE
C### GENERIC, SOLUTION PARAMETERS ARE PASSED TO THIS ROUTINE THROUGH INTEGER
C### AND REAL PARAMETER ARRAYS.  THE ENTRIES OF THESE ARRAYS ARE DESCRIBED
C### BELOW.
C###
C### INFO - INTEGER PARAMETERS
C###
C### INFO(1) - OUTPUT UNIT FOR INFORMATION/WARNING/ERROR MESSAGES
C###              INFO(1) >= 0
C### INFO(2) - PRINT LEVEL
C###              INFO(2) = 0 - NO PRINTING
C###                        1 - ERROR MESSAGES
C###                        2 - ERROR AND WARNING MESSAGES
C###                        3 - ERROR, WARNING, SOME INFORMATION
C###                      > 3 - ERROR, WARNING, MORE INFORMATION
C### INFO(3) - MAXIMUM NUMBER OF ITERATIONS
C###              INFO(3) = 0 - DEFAULT NUMBER USED
C### INFO(4) - VARIABLE BOUNDS FLAG
C###              INFO(4) = 0   - VARIABLES ARE BOUNDED,
C###              INFO(4) NOT 0 - UNBOUNDED PROBLEM
C### INFO(5) - QUASI-NEWTON METHOD (DEFERRED JACOBIAN UPDATE)
C###              INFO(5) = 0   - STANDARD NEWTON'S METHOD
C###              INFO(5) NOT 0 - QUASI-NEWTON METHOD
C### INFO(6) - STEPSIZE SELECTION ALGORITHM.
C###              INFO(6) = -1 - FULL NEWTON STEP TAKEN
C###              INFO(6) =  0 - STEPSIZE SELECTION USING ARMIJO'S RULE
C###
C### RINFO - REAL PARAMETERS
C###
C### RINFO(1) - ABSOLUTE ERROR USED FOR CONVERGENCE CRITERIA
C###           USED FOR BOTH NORM OF RESIDUALS AND NORM OF
C###           STEP
```

```
C###               RINFO(1) > EPS (MACHINE TOLERANCE)
C### RINFO(2) - CRITERIA TO DETERMINE WHETHER OR NOT QUASI-NEWTON
C###            METHOD SHOULD BE APPLIED (WHEN INFO(5) DOES NOT
C###            EQUAL ZERO)
C###               EPS < RINFO(2) < ONE
C###
C### ** RETURN STATUS VARIABLE
C### THE COMPLETION STATUS OF THE SOLUTION PROCEDURE IS RETURNED
C### IN THE ICODE ARGUMENT.  POSSIBLE RETURN VALUES ARE:
C###
C###    ICODE =  0 - SUCCESSFULLY CONVERGED SYSTEM OF EQUATIONS
C###            -1 - INVALID INPUT - SOLVER PARAMETERS ARRAYS
C###                 INFO AND/OR RINFO
C###            -2 - FAILURE IN SUBROUTINE RES
C###            -3 - FAILURE IN SUBROUTINE JAC
C###            -4 - FAILURE IN LINEAR ALGEBRA ROUTINES
C###            -5 - GENERAL FAILURE - SEE ISSUED ERROR MESSAGE
C###                 FOR MORE DETAIL
C###             K - UNABLE TO CONVERGE AFTER K ITERATIONS
C###
C### ** EXTERNAL SUBROUTINES
C###
C### RES - RESIDUAL ROUTINE
C###
C###          SUBROUTINE RES(ICODE,N,X,F,RPAR,IPAR,IBLK)
C###
C###       IF SUCCESSFUL, ICODE = 0
C###
C### JAC - JACOBIAN ROUTINE
C###
C###          SUBROUTINE JAC(ICODE,N,X,F,NE,DF,IRLIST,JCLIST,
C###             RPAR,IPAR,IBLK)
C###
C###       IF SUCCESSFUL, ICODE = 0
C###
C### LUFACT - JACOBIAN FACTORIZATION ROUTINE
C###
C###          SUBROUTINE LUFACT(ICODE,N,NE,A,IRLIST,JCLIST,LP,PLEVEL,
C###             LRLAW,RLAW,LILAW,ILAW)
C###
C###       IF SUCCESSFUL, ICODE = 0
C###
C### LUSING - SINGULARITY HANDLING ROUTINE
C###
C###          SUBROUTINE LUSING(ICODE,N,RANK,NE,X,A,IRLIST,JCLIST,
C###             LP,PLEVEL,LRLAW,RLAW,LILAW,ILAW)
```

```
C###
C###       IF SUCCESSFUL, ICODE = 0
C###
C### LUBACK - BACKSUBSTITUTION ROUTINE
C###
C###          SUBROUTINE LUBACK(ICODE,N,NE,X,Y,F,DF,LP,PLEVEL,
C###             LRLAW,RLAW,LILAW,ILAW)
C###
C###       IF SUCCESSFUL, ICODE = 0
C###
C###
C### DESIGN PHILOSOPHY
C###
C### THIS SUBROUTINE WAS DESIGNED TO BE MULTPURPOSE AND HIGHLY
C### EXTENDIBLE/FLEXIBLE.  AN ATTEMPT TO MAKE THE INTERFACE AS
C### GENERIC AS POSSIBLE SO THAT THE ROUTINE CALLING NWTSLV MAY
C### READILY EXCHANGE BETWEEN A WIDE VARIETY OF SOLVERS WITH
C### IDENTICAL GENERIC INTERFACES. THEN RETURN VALUES THROUGH
C### ICODE ARE ALSO KEPT GENERIC FOR THE SAME REASON, RELYING
C### ON ISSUED ERROR MESSAGES FOR MORE DETAILED INFORMATION.
C### EXTENSIBILTY AND FLEXIBILITY IS ACHIEVED BY ALLOWING THE
C### USER TO FREELY USE ANY NUMBER OF LINEAR ALGEBRA AND
C### SINGULARITY HANDLING ROUTINES (THE INTERFACE TO THESE
C### ROUTINES IS KEPT GENERIC FOR THE SAME REASONS ABOVE).
C### AS A CONSEQUENCE OF THESE DESIGN OBJECTIVES, THE ARGUMENT
C### LIST OF NWTSLV CONTAINS A NUMBER OF ARGUMENTS THAT MAY OR
C### MAY NOT BE REQUIRED. FOR EXAMPLE, IF DENSE LINEAR ALGEBRA
C### IS USED FOR THE FACTORIZATION AND BACKSUBSTITUTION THEN
C### THE OCCURRENCE INFORMATION (IRLIST AND JCLIST) MAY NOT BE
C### REQUIRED, IN WHICH CASE, DUMMY ARGUMENT ARRAYS WITH LENGTH
C### ONE CAN BE PASSED TO NWTSLV INSTEAD OF THE FULL OCCURRENCE
C### INFORMATION. THE ARGUMENTS MARKED BY (*) ABOVE MAY OR MAY
C### NOT BE USED DEPENDING ON THE OPTIONS IN INFO/RINFO AND
C### EXTERNALLY SUPPLIED SUBROUTINES.
C###
C### ** BLOCK DECOMPOSITON
C### NWTSLV HAS THE OPTION TO BE CALLED FROM A BLOCK SOLVER
C### WHERE THE OVERALL SYSTEM OF EQUATIONS HAS BEEN BLOCK LOWER
C### TRIANGULARIZED AND SOLVED AS A SEQUENCE OF SMALLER PROBLEMS.
C### THIS DECOMPOSITION IS TRANSPARENT TO NWTSLV: RES AND JAC
C### RETURN THE PROPER SUBSETS OF THE RESIDUALS AND JACOBIAN
C### MATRIX AND NWTSLV TREATS THESE AS THOUGH THEY WERE A REGULAR
C### SYSTEM OF EQUATIONS AND NOT PART OF A LARGER BLOCK.  IN
C### ORDER FOR RES AND JAC TO KNOW WHICH SUBSETS TO RETURN, ARRAY
C### IBLK(*) IS PASSED AS AN ARGUMENT.  A DESCRIPTION OF THE
C### CONTENTS OF THIS ARRAY IS OMMITTED HERE SINCE NWTSLV NEED
```

```
C### NOT KNOW ANYTHING OF THIS DECOMPOSITION.  RES AND JAC TAKE
C### THIS AS ARRAY AS AN ARGUMENT IN ORDER TO RETURN THE PROPER
C### SUBSET OF THE OVERALL SYSTEM (SEE DOCUMENATION OF BLOCK
C### SOLVER FOR MORE INFORMATION) THIS ARGUMENT IS ONLY REQUIRED
C### IF THIS SUBROUTINE IS CALLED FROM A BLOCK SOLVER.
C###
```

6.3 SLPSLV

Since the Newton-Raphson method has execellent convergence properties when "close" to the solution, NWTSLV is the default solver for the diagonal blocks. If NWTSLV fails to find a solution after a user-specified number of iterations (see NWTSLV description above), a more robust method is employed. If the block is a single equation then BSCSLV is employed, which is guaranteed to converge linearly to a solution if the user provides appropriate bounds on the variables. If the block contains more than one equation then a solver based on sequential linear programming is called. The current implementation, called SLPSLV, uses the Harwell Library routine LA01BD to perform the linear programming (LP) subproblem. Since this is a "dense" implementation (i.e., sparsity is not exploited), it is limited to blocks that are not too large (how large depends on how long you are willing to wait for a solution). Nevertheless, our experience has shown the solver to be quite robust for blocks containing less than a few dozen equations. New versions of this subroutine, exploiting sparse LP codes, are currently under developement. The interface and description of SLPSLV is shown below.

```
      SUBROUTINE SLPSLV(ICODE,INFO,RINFO,N,X,BOUNDS,F,NE,LDF,DF,
     1   IRLIST,JCLIST,IBLK,RPAR,IPAR,LRW,RW,LIW,IW,LRLPW,RLPW,
     2   LILPW,ILPW,RES,JAC,LUFACT,LUSING,LUBACK)
C### 990527 JOHN E. TOLSMA COPYRIGHT MIT
C### SLPSLV - SUCCESSIVE LINEAR PROGRAMMING NONLINEAR EQUATION
C### SOLVER.
C###
C### DESCRIPTION: SOLVES NONLINEAR SYSTEM OF EQUATIONS USING
C### SUCCESSIVE LINEAR PROGRAMMING.
C###
C### ARGUMENTS (# INDICATES ARGUMENT MUST BE SET PRIOR TO CALL)
C###           (* INDICATES ARGUMENT MAY NOT BE USED AND A SIMPLE
C###              DUMMY ARGUMENT WILL SUFFICE)
C###     ICODE       - SOLUTION STATUS (SEE BELOW)
C### (#) INFO(3)     - INTEGER SOLUTION PARAMETERS (SEE BELOW)
C### (#) RINFO(1)    - REAL SOLUTION PARAMETERS (SEE BELOW)
C### (#) N           - DIMENSION OF SYSTEM
C### (#) X(N)        - INITIAL GUESS FOR ITERATIVE METHOD.
C###                   UPON SUCCESSFUL COMPLETION, CONTAINS SOLUTION
C### (*) BOUNDS(2*N) - VARIABLE BOUNDS
C###                   LOWER BOUNDS - BOUNDS(1:N)
C###                   UPPER BOUNDS - BOUNDS(N+1,2N)
C###     F(N)        - RESIDUAL VECTOR WORKSPACE
C### (#) NE          - NUMBER OF NONZERO ENTRIES IN JACOBIAN
```

```
C###                   MATRIX (LESS THAN N*N IF SPARSITY IS
C###                   EXPLOITED)
C###     DF(LDF)     - JACOBIAN WORKSPACE
C### (*) IRLIST(NE)  - ROW INDICIES (ONLY REQUIRED IF SPARSITY
C###                   IS EXPLOITED - SEE BELOW)
C### (*) JCLIST(NE)  - COLUMN INDICIES (ONLY REQUIRED IF SPARSITY
C###                   IS EXPLOITED - SEE BELOW)
C### (*) IBLK(*)     - BLOCK INFORMATION (SEE BELOW)
C### (*) RPAR(*)     - REAL PARAMETERS PASSED TO RES AND JAC
C### (*) IPAR(*)     - INTEGER PARAMETERS PASSED TO RES AND JAC
C### (#) RW(LRW)     - REAL WORKSPACE, LRW >= N
C### (*) IW(LIW)     - INTEGER WORKSPACE, LIW >= 0
C### (#) RLPW(LRLPW) - LP ALGORITHM REAL WORKSPACE
C###                   *** LRLPW = 21N*N + 23N +3
C### (#) ILPW(LILPW) - LP ALGORITHM INTEGER WORKSPACE
C###                   *** LILPW = 3N + 1
C### (#) RES         - RESIDUAL ROUTINE (SEE BELOW)
C### (#) JAC         - JACOBIAN ROUTINE (SEE BELOW)
C###     DUMMY1,
C###     DUMMY2,
C###     DUMMY3      - DUMMY ARGUMENTS USED TO KEEP INTERFACE
C###                   CONSISTENT
C###
C### ** SOLUTION PARAMETER ARRAYS
C### IN ORDER TO REDUCE ARGUMENT LIST COMPLEXITY AND KEEP THE
C### INTERFACE GENERIC, SOLUTION PARAMETERS ARE PASSED TO THIS
C### ROUTINE THROUGH INTEGER AND REAL PARAMETER ARRAYS.  THE
C### ENTRIES OF THESE ARRAYS ARE DESCRIBED BELOW.
C###
C### INFO - INTEGER PARAMETERS
C###
C### INFO(1) - OUTPUT UNIT FOR INFORMATION/WARNING/ERROR MESSAGES
C###              INFO(1) >= 0
C### INFO(2) - PRINT LEVEL
C###              INFO(2) = 0 - NO PRINTING
C###                        1 - ERROR MESSAGES
C###                        2 - ERROR AND WARNING MESSAGES
C###                        3 - ERROR, WARNING, SOME INFORMATION
C###                      > 3 - ERROR, WARNING, MORE INFORMATION
C### INFO(3) - MAXIMUM NUMBER OF ITERATIONS
C###              INFO(3) = 0 - DEFAULT NUMBER USED
C###
C### RINFO - REAL PARAMETERS
C###
C### RINFO(1) - ABSOLUTE ERROR USED FOR CONVERGENCE CRITERIA
C###           USED FOR BOTH NORM OF RESIDUALS AND NORM OF
```

```
C###            STEP
C###               RINFO(1) > EPS (MACHINE TOLERANCE)
C###
C### ** RETURN STATUS VARIABLE
C### THE COMPLETION STATUS OF THE SOLUTION PROCEDURE IS RETURNED
C### IN THE ICODE ARGUMENT.  POSSIBLE RETURN VALUES ARE:
C###
C###    ICODE =  0 - SUCCESSFULLY CONVERGED SYSTEM OF EQUATIONS
C###            -1 - INVALID INPUT - SOLVER PARAMETERS ARRAYS
C###                 INFO AND/OR RINFO
C###            -2 - FAILURE IN SUBROUTINE RES
C###            -3 - FAILURE IN SUBROUTINE JAC
C###            -4 - FAILURE IN LINEAR PROGRAMMING ROUTINES
C###            -5 - GENERAL FAILURE - SEE ISSUED ERROR MESSAGE
C###                 FOR MORE DETAIL
C###             K - UNABLE TO CONVERGE AFTER K ITERATIONS
C###
C### ** EXTERNAL SUBROUTINES
C###
C### RES - RESIDUAL ROUTINE
C###
C###          SUBROUTINE RES(ICODE,N,X,F,RPAR,IPAR,IBLK)
C###
C###       IF SUCCESSFUL, ICODE = 0
C###
C### JAC - JACOBIAN ROUTINE
C###
C###          SUBROUTINE JAC(ICODE,N,X,F,NE,DF,IRLIST,JCLIST,
C###             RPAR,IPAR,IBLK)
C###
C###       IF SUCCESSFUL, ICODE = 0
C###
C### DESIGN PHILOSOPHY
C###
C### THIS SUBROUTINE WAS DESIGNED TO BE MULTPURPOSE AND HIGHLY
C### EXTENDIBLE/FLEXIBLE.  AN ATTEMPT TO MAKE THE INTERFACE AS
C### GENERIC AS POSSIBLE SO THAT THE ROUTINE CALLING SLPSLV MAY
C### READILY EXCHANGE BETWEEN A WIDE VARIETY OF SOLVERS WITH
C### IDENTICAL GENERIC INTERFACES. THEN RETURN VALUES THROUGH
C### ICODE ARE ALSO KEPT GENERIC FOR THE SAME REASON, RELYING
C### ON ISSUED ERROR MESSAGES FOR MORE DETAILED INFORMATION.
C### AS A CONSEQUENCE OF THESE DESIGN OBJECTIVES, THE ARGUMENT
C### LIST OF SLPSLV CONTAINS A NUMBER OF ARGUMENTS THAT MAY OR
C### MAY NOT BE REQUIRED. THE ARGUMENTS MARKED BY (*) ABOVE MAY
C### OR MAY NOT BE USED DEPENDING ON THE OPTIONS IN INFO/RINFO
C### AND EXTERNALLY SUPPLIED SUBROUTINES.
```

```
C###
C### ** BLOCK DECOMPOSITON
C### SLPSLV HAS THE OPTION TO BE CALLED FROM A BLOCK SOLVER
C### WHERE THE OVERALL SYSTEM OF EQUATIONS HAS BEEN BLOCK LOWER
C### TRIANGULARIZED AND SOLVED AS A SEQUENCE OF SMALLER PROBLEMS.
C### THIS DECOMPOSITION IS TRANSPARENT TO SLPSLV: RES AND JAC
C### RETURN THE PROPER SUBSETS OF THE RESIDUALS AND JACOBIAN
C### MATRIX AND SLPSLV TREATS THESE AS THOUGH THEY WERE A REGULAR
C### SYSTEM OF EQUATIONS AND NOT PART OF A LARGER BLOCK.  IN
C### ORDER FOR RES AND JAC TO KNOW WHICH SUBSETS TO RETURN, ARRAY
C### IBLK(*) IS PASSED AS AN ARGUMENT.  A DESCRIPTION OF THE
C### CONTENTS OF THIS ARRAY IS OMMITTED HERE SINCE SLPSLV NEED
C### NOT KNOW ANYTHING OF THIS DECOMPOSITION.  RES AND JAC TAKE
C### THIS AS ARRAY AS AN ARGUMENT IN ORDER TO RETURN THE PROPER
C### SUBSET OF THE OVERALL SYSTEM (SEE DOCUMENATION OF BLOCK
C### SOLVER FOR MORE INFORMATION) THIS ARGUMENT IS ONLY REQUIRED
C### IF THIS SUBROUTINE IS CALLED FROM A BLOCK SOLVER.
C###
```

6.4 User-supplied Solvers

A design objective from the beginning was to make DAEPACK as flexible as possible, allowing the modeler to use DAEPACK for a wide variety of calculations. To meet this objective, DAEPACK should allow the user to incorporate custon-tailored algorithms to solve selected diagonal blocks. However, this feature is not currently implemented but will be avaiable in the near future.

7 Distribution

DAEPACK component `BLKSLV` is currently available as both a shared and static library named:

`libblocksolver_<`*platform*`>_<`*version*`>.sl`

and

`libblocksolver_<`*platform*`>_<`*version*`>.a`,

respecitvely, for UNIX and Linux, where *platform* is the platform for which the library is compiled and *version* is the version number, and

`libblocksolver_win32_<`*version*`>.dll`,

for Windows 98 and Windows NT. See the DAEPACK webpage (http://yoric.mit.edu/daepack/daepack.html) for available platforms and versions. Several additional libraries must be linked with the library above in order to use the block solver. These include certain Harwell library structural and linear algebra routines as well as some of the Linpack linear algebra routines. These libraries are described in [2].

References

[1] J. E. Tolsma, *DAEPACK code generation manual*, Tech. Rep. DAEPACK Documentation. Version 1.0, Massachusetts Institute of Technology, 2000. http://yoric.mit.edu/daepack/daepack.html.

[2] ——, *Supporting libraries for the DAEPACK numerical components*, Tech. Rep. DAEPACK Documentation. Version 1.0, Massachusetts Institute of Technology, 2000. http://yoric.mit.edu/daepack/daepack.html.

[3] A. W. Westerberg, H. P. Hutchison, R. L. Motard, and P. Winter, *Process Flowsheeting*, Cambridge University Press, Cambridge, 1979.

Table 1: Summary of arguments required by BLKSLV.

Status	Type	Name	Description
input/ output	integer	`icode`	Upon input, used to determine whether or not block permutation is to be performed: icode=-1 don't block permute system, icode=0 perform block permutation. Upon output, flag indicating solution status. A zero return value indicates success. Nonzero values indicate an error has occurred.
input	integer	`info(*)`	Integer information array used in BLKSLV and block solvers called for diagonal blocks: info(1) is the print unit, info(2) the print level, and the remaining entries are specific to solvers.
input	real	`rinfo(*)`	Real information array passed to block solvers. See solver descriptions for these parameter values.
input	integer	`n`	Dimension of entire system of equations.
input/ output	real	`x(n)`	Upon input, this vector contains the initial guess for the solution. Upon returning, this contains the solution or values when solver failed at a block.
input	real	`lower(n)`	Lower bounds on `x`.
input	real	`upper(n)`	Upper bounds on `x`.
input	integer	`ne`	Number of nonzero entries in Jacobian of entire system.
input	integer	`irlist(ne)`	Row occurrence information of Jacobian of entire system.
input	integer	`jclist(ne)`	Column occurrence information of Jacobian of entire system.
input	integer	`lrw`	Real workspace length (see code excerpt).
work	integer	`rw(lrw)`	Real workspace (see code excerpt).
input	integer	`liw`	Integer workspace length (see code excerpt).
work	integer	`iw(liw)`	Integer workspace (see code excerpt).
input	external	`res`	Block residual evaluator (described in section 3.1).
input	external	`jac`	Block Jacobian evaluator (described in section 3.2).
input	real	`rpar`	This real array serves two purposes: the first `n` entries are used as workspace (to pass the full vector `x` to `res` and `jac`) and the remaining may be set by the user to provide real parameters for `res` and `jac`.
input	integer	`ipar`	This integer array serves two purposes: the first entry is used as workspace (to pass the `n` to `res` and `jac`) and the remaining may be set by the user to provide integer parameters for `res` and `jac`.

Table 2: Summary of arguments required by user-supplied residual routine `res`.

Status	Type	Name	Description
output	integer	`icode`	Flag indicating solution status. A zero return value indicates success. Nonzero values indicate an error has occurred.
input	integer	`nc`	Dimension of current diagonal block.
input	real	`xc(nc)`	Point at which residual values are desired.
output	real	`fc(*)`	The first `nc` entries of this vector must be filled with residual entries using the information contained in `iblk`. This vector has enough room to hold the residual vector of the entire system of equations.
input	real	`rpar`	This real array serves two purposes: the first `n` entries contain the full vector `x` and the remaining may be set by the user to provide real parameters used in the residual computation.
input	integer	`ipar`	This integer array serves two purposes: the first entry is used to pass the value of `n` and the remaining may be set by the user to provide integer parameters used in residual computation.
input	integer	`iblk(*)`	This array contains block permutation information and used to identify which residual entries to return in `fc`.

Table 3: Summary of arguments required by user-supplied Jacobian routine `res`.

Status	Type	Name	Description
output	integer	`icode`	Flag indicating solution status. A zero return value indicates success. Nonzero values indicate an error has occurred.
input	integer	`nc`	Dimension of current diagonal block.
input	real	`xc(nc)`	Point at which residual values are desired.
output	real	`fc(*)`	The first `nc` entries of this vector must be filled with residual entries using the information contained in `iblk`. This vector has enough room to hold the residual vector of the entire system of equations.
input	integer	`nzc`	Number of nonzero entries in Jacobian of current block.
output	real	`df(nzc)`	Filled with Jacobian of current block. Desired partial derivatives and their ordering can be obtained from `iblk`.
input	integer	`irlist(nzc)`	Row occurrence information for current block.
input	integer	`jclist(nzc)`	Column occurrence information for current block.
input	real	`rpar`	This real array serves two purposes: the first `n` entries contain the full vector `x` and the remaining may be set by the user to provide real parameters used in the Jacobian computation.
input	integer	`ipar`	This integer array serves two purposes: the first entry is used to pass the value of `n` and the remaining may be set by the user to provide integer parameters used in Jacobian computation.
input	integer	`iblk(*)`	This array contains block permutation information and used to identify which residual entries to return in `fc`.