WATSIT-1 and WATSIT-B
Waterloo Sparse, Iterative Matrix Solvers
User's Guide
with
Developer Notes
for
Version 2.0.0

Simon S. Clift, Edward F. D'Azevedo,
Peter A. Forsyth, and James R. Knightly

November 5, 1997

Contents

List of Figures

List of Tables

1 Introduction and Overview

WATSIT-1 and WATSIT-B are packages that make up the WATSIT matrix solver project. They are iterative sparse matrix solvers that operate on the system

$$Ax = b$$

where A is an $N \times N$ sparse, non-singular matrix, and b and x are N-length vectors. The vector b is the source term (or right-hand side), and x is the solution vector. The matrix A must

- be real,
- have non-zero, or non-singular diagonal elements on every row,
- and be "sufficiently" positive definite.

WATSIT-1 is designed to operate on general sparse matrices, without regard to matrix structure. WATSIT-B is designed to take advantage of block structure in matrices, such as arises in solutions of partial differential equation systems, where several variables are being solved at each node. Neither the matrix structure, nor matrix values need be symmetric, in fact, WATSIT-1 and WATSIT-B are designed specifically as a non-symmetric matrix solvers. Symmetric matrices may be solved with WATSIT-1, but the package does not attempt to apply the storage methods which are most efficient for such problems.

The packages are divided into distinct sections.

- **Initialization** — These allow the user to set items such as output unit, and amount of printing to be done by the solver. It makes sure the internal data structures are consistent.
- **Matrix Pre-Processing** — This is a set of operations that might be performed on the matrix or the data structure before attempting the standard precondition and solve. An example of pre-processing would be matrix scaling. This category exists primarily to classify routines that fall outside of the basic matrix solution process.
- **Preconditioning** — This set of routines creates the internal, static, data structure in the work space for the preconditioner and performs the numeric factorization into the work space. Since it is often convenient to retain a previously used data structure, and simply change the matrix values and numerically re-factor the preconditioner, the numeric factorization phase can be used separately from the data structure analysis.
- **Solution** — This routine applies the user's selection of one of a number of current iterative acceleration techniques to solve for the given right hand side of the matrix equation. Available are the Conjugate Gradient (CG), OrthoMin, CGS, CGSTAB-P, and GMRES iteration methods.
- **Utility** — A number of short routines are provided for such operations as interpreting solver errors, returning internal data, and returning pointers to internal data structures.

WATSIT-1 is a derivation of version 12 of the previous Mat1 product from the Scientific Computation Group, and represents a rationalization of its former incarnation. It is written in Fortran-77, with deviations from the standards that are noted later. The releases of Mat1 will be referred to as versions 1.00 to 1.12 of WATSIT-1. WATSIT-B is a rewrite of MatB, a block matrix solver package used at the University of Waterloo, but never generally released. Users of Mat1 and Matb will find many of the concepts familiar, but are advised to note that there are substantial differences between the packages. Users will be pleased to know that the COMMON blocks have been reduced, and that the calling sequences to the main routines have been made more consistent, and in most cases shortened.

1.1 How to Avoid Reading this Manual

If you are already familiar with the concepts behind these matrix solvers, you might be able to avoid reading this manual just by looking at the files `wats1dec.inc` (for WATSIT-1) or `wats2dec.inc` (for WATSIT-B) that accompany this package. They contain template code, with comments, that outlines the basic features of the system, and how to use the subroutines and `COMMON` blocks. You might wish to look at the test harnesses, `w1testpr.f`, and/or `w2testpr.f`, but they contain other routines which don't have to do directly with solving matrices.

1.2 WARNING

Always check the error flag which is returned by each WATSIT subroutine. If an error is detected (i.e. too little workspace), the subroutine will return, with the error flag set. II IS UP TO YOU TO DO SOMETHING ABOUT THIS! If you call the next WATSIT routine, and the errors have not been fixed, unpredictable things may happen.

Note: the subroutine W1ERRP may be used to interpret error codes returned by WATSIT routines.

2 Application Notes

2.1 Nomenclature

For clarity, we shall define the following terms, and use them consistently throughout this guide:

- *scalar matrix* — A matrix without block structure, whose every element is 1×1.
- *block matrix* — A matrix with block structure, whose every element is $m \times n$, where $m \geq 1$ and $n \geq 1$.
- *scalar element* — A 1×1 piece of either a scalar or block matrix.
- *block element* — A $m \times n$ array of non-zero scalar elements within a block matrix.
- *element* — When used by itself, element refers to a scalar element in a scalar matrix, and a block element in a block matrix. This may also be referred to as a *matrix element.*
- *diagonal element* — An element, of dimensions $m \times m$, where $m \geq 1$, on the diagonal of the matrix.

See Figure 1 for a picture.

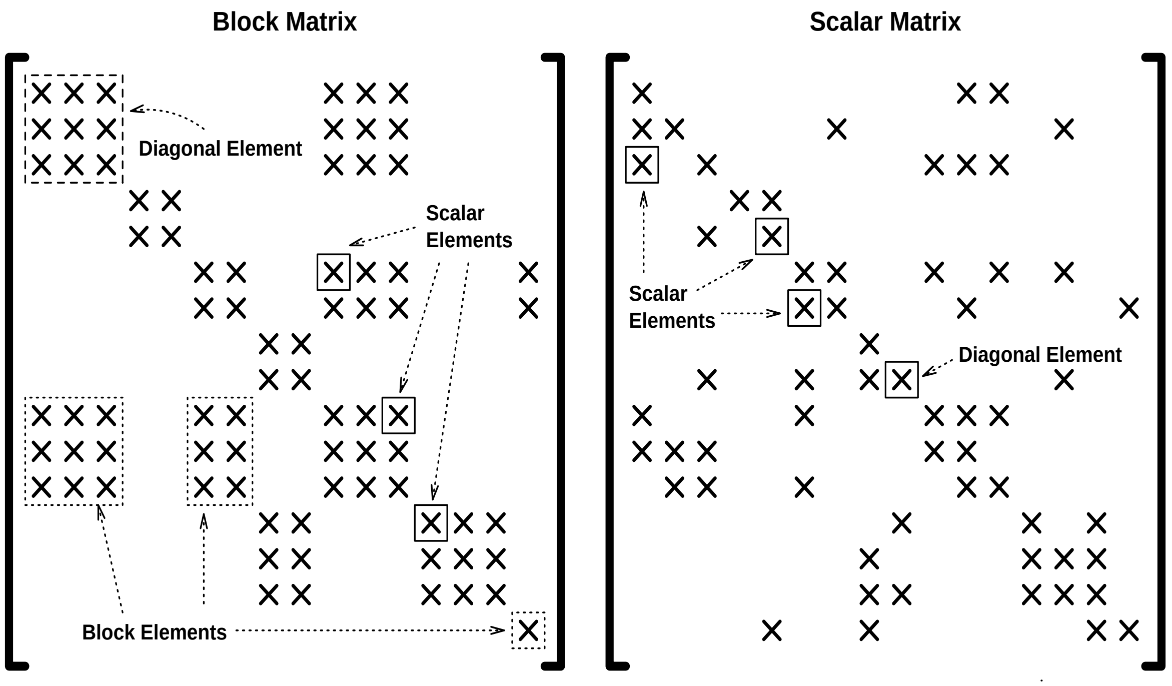

Figure 1: Types of matrices and their parts

2.2 The Sparse Matrix Storage Scheme

At the highest level, either a block or scalar matrix can be considered as rows and columns of elements. The elements are stored in the row pointer/column index form used in the Yale Sparse Matrix Package and (in transposed form) in Sparsepak. It is generally known as *ia,ja* format, and requires two pointer arrays

- ia — the array of row pointers into ja.
- ja — the array of column indices.

Each row's column indices are stored, packed together, in the ja array. Thus there is one ja entry for each non-zero in the matrix. The ia array is a pointer to the first entry of each row. To complete the pattern, and maintain the form of the indexing, ia$(n+1)$, where n is the number of rows of elements in the matrix, points to the position after the last entry of ja.

There are two restrictions on the entries in the *ia,ja* data structure.

1. There must be a non-zero diagonal element for each row. One reason for this is that since the data structure of the preconditioner is static (i.e. non-pivoting), it requires a non-zero element on each diagonal for the incomplete factorization stage.
2. The diagonal element of each row i must be at ia(i). In other words, the first entry of each row in ja must be the diagonal, or ja(ia(i)) $= i$. The off-diagonal entries in the row may be in any order. There is a matrix pre-processing routine provided that puts an *ia,ja* format matrix in the correct row order.

2.2.1 Scalar Matrix Structure

If we are dealing with a **scalar matrix**, then the data structure is completed with the addition of the matrix

- a — the array of all the matrix non-zero scalar elements in turn.

The a array has the same structure as ja, thus the ia entry for a particular row also points to the start of that row's scalar element values in a. Thus for a scalar matrix, a(l) is the $(i,j)^{\mathrm{th}}$ entry of the matrix, where

$$\mathrm{ja}(l) = j \ , \quad \mathrm{ia}(i) \le l \le \mathrm{ia}(i+1) - 1 \ .$$

Refer to Figure 2 for a graphical idea of how the scalar matrix data structure works.

2.2.2 Block Matrix Structure

If we are dealing with a **block matrix**, the data structure is completed with the addition of four more matrices to ia and ja:

- neqa — The number of scalar element rows in each block element row of the matrix.
- kvlst — Each element of this arrays points to the first element of x of $Ax = b$ that corresponds to the given block element row.

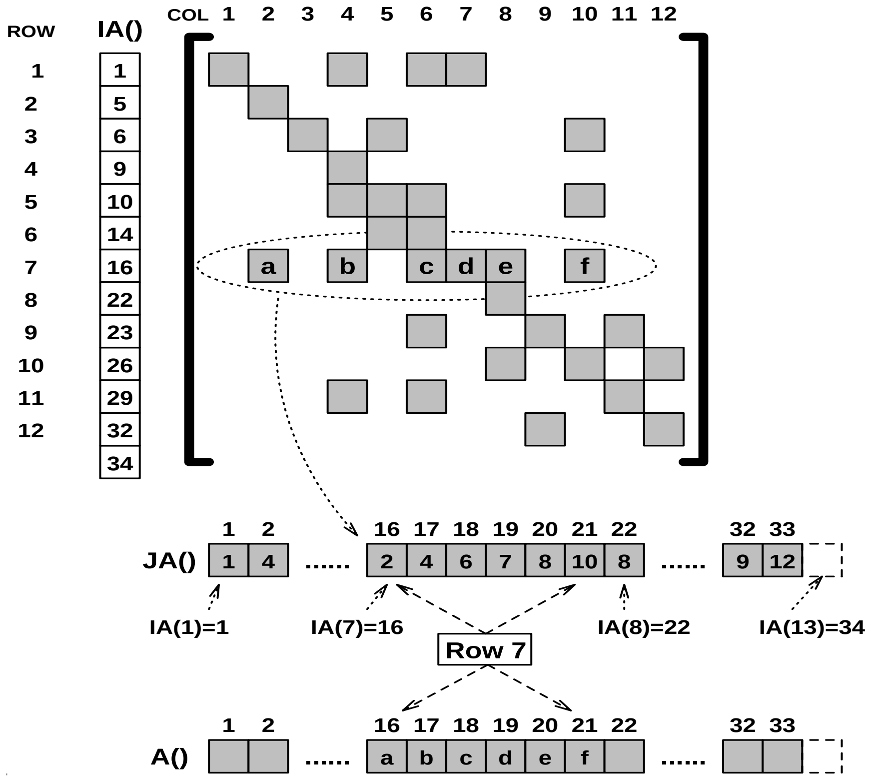

Figure 2: Scalar matrix data structure (*ia,ja,a*)

- ka — Each element of this array points to the portion of A containing the block element in packed form.

- a — Packed storage for each non-zero block element of the matrix in turn.

The vector of unknowns (as well as the right hand side vector) are indexed with the array klvst. For a particular block element there will be a certain number of scalar element rows given by neqa. Thus if x is the array of unknowns, the unknowns corresponding to block row i of the matrix would be

$$\mathrm{x}(\mathrm{kvlst}(i)) \ldots \mathrm{x}(\mathrm{kvlst}(i) + \mathrm{neqa}(i) - 1)\ .$$

Note that the unknowns are **NOT** considered to be stored in

$$\mathrm{x}(\mathrm{kvlst}(i)) \ldots \mathrm{x}(\mathrm{kvlst}(i+1) - 1)\ .$$

This is to allow the user to vary in the number of scalar element rows in the matrix blocks without modifying the main data structure.

The block elements of the matrix, which are stored in a, are pointed to by the ka array. Let a_{ij}^{kl} denote the scalar element on the k^{th} row, and l^{th} column of the non-zero block element located on block row i, block column j of the matrix. The block element a_{ij} is stored in row-wise form, the transpose of the normal Fortran-77. So, if

$$\mathrm{ja}(m) = j$$

such that

$$\mathrm{ia}(i) \leq m \leq \mathrm{ia}(i+1) - 1$$

then if the block elements are stored in the form

$$a_{ij}^{kl} = a(\mathrm{ka}(m) + (l-1) + (k-1) * \mathrm{neqa}(j)) \; .$$

Refer to Figure 3 for a graphical idea of how the block matrix data structure works.

The following code shows the setup of the kvlst() and ka() arrays, given the neqa() array. Note that the form of the kvlst() and ka() arrays used in MatB had the first elements of both array set to zero. This was changed for conceptual consistency with the *ia,ja* format. Now, ia(), ka(), and kvlst() are all pointers into arrays, rather than offsets. Code that utilized MatB, being converted to WATSIT-B, is probably most easily modified by inserting a short loop to add one to each value in the ka() and kvlst() array before WATSIT-B uses it, and if necessary, inserting another loop to reduce all the values by one afterwards.

```
   kvlst( 1 ) = 1
   do 10 i=1, nodes
      kvlst( i+1 ) = kvlst( i ) + neqa( i )
10 continue

   ka( 1 ) = 1
   do 20 inode=1, nodes
      do 30 k=ia(inode), ia(inode+1)-1
           ka( k+1 ) = ka( k ) + neqa( inode ) * neqa( ja(k) )
20    continue
30 continue
```

2.3 Work Arrays

Work space for a particular matrix solution is grouped into three sections

1. a master workspace `MWORK` of a definite length
2. an `INTEGER` workspace
3. and a `DOUBLE PRECISION` workspace.

The master workspace stores pointers into the other two work arrays, and parameters relevant to the solution of the matrix at hand. More than one set of workspaces may be allocated, which allows the solver to be used on more than one matrix in a program. However, work arrays should always be used in sets. Using a work array allocated for one matrix for the solution of another will cause the results to be undefined.

Code	Explanation
-4	Real workspace too small for the iterative solve stage.
-3	Sparsity pattern found to be asymmetric where symmetric expected.
-2	Small diagonal in the preconditioner.
-1	Iterative solve failed to converge.
0	No error.
1	Integer work space too small.
2	Integer work space too small for preconditioning stage.
3	Integer work space too small for solve stage.
4	Double precision work space too small.
5	Double precision work space found to be too small for preconditioning stage.
6	Double precision work space found to be too small for solve stage.
7	Error detected in the sparsity structure (*ia,ja*).
8	Parameter has invalid value.
9	Diagonal is not the first element of the row.
10	Fatally small diagonal in preconditioner.
11	Initialization not done.
12	Preconditioner data structure analysis not completed.
13	Preconditioner numeric factorization not completed.
14	Invalid I/O unit number detected.
15	Number of equations given fewer than 2.
16	Error detected in matrix values (general).
17	Severe internal error, contact user support.

Table 1: Error Code Table

2.4 Explanation of Error Codes

Errors are defined to be conditions which prevent the continuation of the program. Warnings alert the user to conditions where the program may continue, but the results may not be what the user anticipated. Warnings have error code values below zero, errors values above zero, and zero denotes no error. Not all errors can occur in all routines, and the exact meaning of an error might only be clear from its context.

Table 1 lists the basic error codes that the various routines may return, and which may be interpreted with the error interpretation routine. If the user has set the solver to report errors, then a basic message will be printed, displaying one of these categories. Usually another message will be printed, reporting more details, from the routine where the error occurred.

2.5 Convergence Criteria

Determining when to stop the iterative process is not a trivial problem. The WATSIT solvers implement two convergence criteria for each acceleration method, one based on the reduction in the linear residual, the other based on changes in the solution update that iteration. More precisely, if r_0 is the initial linear residual, and r_k the residual at iteration k, then the first criterion requires that

$$\frac{r^k}{r^0} < prec \tag{1}$$

where *prec* is the desired residual reduction. If Δ_i is the change in the ith element of the solution, then the second convergence criterion requires that

$$\Delta_i < toler_i \;\; \forall i \, , \, 1 \leq i \leq NEQ \tag{2}$$

where *toler* is the tolerance vector and *NEQ* the number of equations. Iteration stops if either criterion is satisfied. The first can be disabled by setting $prec = 0$, the second by setting $toler = \mathbf{0}$.

There are advantages and drawbacks to both methods for detecting convergence. The linear residual multiplied by the condition number κ of the matrix, which is the ratio of the highest to the lowest eigenvalue, gives the true accuracy of the solution. Unfortunately those eigenvalues are generally difficult to estimate.

The second method can fail when the acceleration "stalls". A solve is considered stalled if the residual does not change by a certain amount over a specified number of iteration. A small update to the solution may occur even though the solution is not converged. Stall detection is built into the acceleration phase for methods that benefit from being restarted if stalling occurs (at present CGS and CGSTAB-P iteration). The restart is automatic. The stall range, and number of iterations required to flag a stall, are under user control. Stall detection provides a measure of protection against early termination.

What constitutes a suitable Δ_i for a given method is also not generally agreed upon. For CGS, where the update to each element of the solution was a single value, only that value is checked. For CGSTAB-P, where the update was composed of a sum of values, both the absolute value of the sum, and the absolute value of each of the components must to be below the tolerance vector value.

GMRES iteration does not automatically compute the solution except when the maximum number of orthogonal vectors has been reached, or convergence has been obtained. An extra parameter is provided for GMRES iteration that can force the solver to spend the time computing the solution vector at intervals in the iteration process specified by the user. The user must decide whether the trade-off of a possibly earlier detection of convergence is worth the extra work of solving for the solution update.

For GMRES, the stall prevention parameter works by ensuring that, before flagging convergence on the i^{th} iteration, the magnitude of the residual reduction,

$$\| \, reduction \, \| = \left| \frac{\|res\|_2^i - \|res\|_2^{i-1}}{\|res\|_2^{i-1}} \right|$$

is larger than a given value. Thus for convergence, the update to the solution must cause a residual reduction of the given magnitude, and then either meet the *toler* vector criterion (when this is checked), or satisfy the residual reduction criterion.

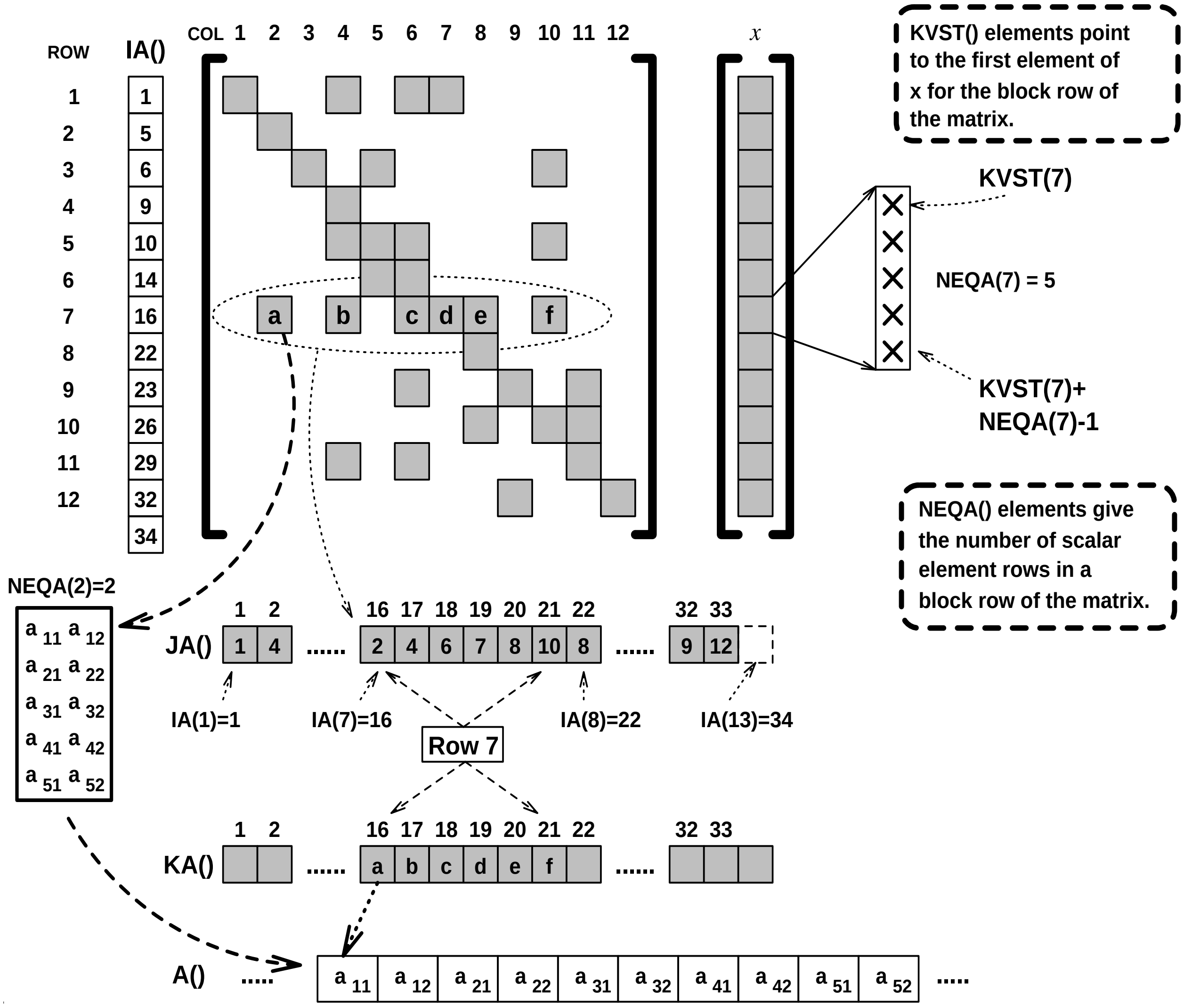

Figure 3: Block matrix data structure (ia, ja, ka, a)

3 Subroutine Structure

This section of the manual gives an overview of the routines in the WATSIT packages. For details of calling parameters and so forth, see the following sections. All user-callable routines for WATSIT-1 begin with the letters `W1`. For WATSIT-B all routine names begin with `W2`. In the individual subprogram descriptions that follow, the two to four letter suffix is given.

There are certain short cuts that might be taken by the user when several matrices must be solved in sequence. Depending on which parts of the previously constructed data structures, or preconditioning the user wishes to retain, it may not be necessary to call the complete set of routines for each solution. Figures 4 and 5 show graphically the possible calling sequences for WATSIT-1 and WATSIT-B respectively. The following information describes the possibilities more precisely.

3.1 Initialization Routines

Whereas in previous versions of Mat1 and MatB, `COMMON` blocks were used to store all current information on the state of the matrix solver, much of this information is now stored in the workspaces. This allows the matrix solver to be applied to more than one matrix in a program. The only parameters stored in `COMMON` or `MODULE` areas are those not directly related to the matrix solution.

In this version, the parameters stored in globally accessable areas are the output parameters, which dictate the level of run-time information, and debugging information to be printed, and the IO unit designated for the output, The initialization and allocation of the *real*, and `INTEGER` workspaces occurs at the data structure analysis phase.

3.1.1 Global Parameter Initialization (`INIT`)

This short routine may be called at any point in the program before the analysis phase routine(s). It sets the output parameters to a reasonable default.

3.2 Matrix Preprocessing Routines

Matrix pre-processing routines are often useful intermediate steps that may affect either the data structure, or the numerical values, of the main matrix being operated upon. It is not recommended that you use these routines on the data stored in the workspaces of WATSIT.

Since matrix preprocessing routines only affect the main matrix, any workspace called for by these routines must not conflict with the workspace provided for the preconditioning and solution routines.

3.2.1 Scale a Matrix (`SCAL`)

This routine multiplies a scalar matrix by an appropriate diagonal matrix so that all the diagonal elements of the main matrix are equal to 1, the identity matrix. The right hand side is also scaled, and the diagonal scaling matrix is returned in a user-supplied vector. In the non-symmetric case, the system is row equivalent, so the result of the iterative solve is also the answer to the scaled system.

Scaling of the matrix is done automatically in the preconditioner numeric factorization phase of WATSIT-B. This is because block matrix solvers rarely function well on unscaled matrices. Since this operation also transforms the original block matrix A into a modified internal storage format, it is not provided as a user-callable routine for WATSIT-B.

Although the WATSIT-1 version of routine does not operate on a specialized, symmetric data structure, but rather the general non-symmetric *ia,ja* data structure, symmetric scaling is an option. The user must remember to rescale the answer after the solution is complete. That is, if D is the diagonal matrix composed of the square roots of the diagonal of symmetric, positive definite matrix A, then if

$$D^{-1}y = x$$

then

$$D^{-1}AD^{-1}y = D^{-1}b.$$

The matrix scaling will return $D^{-1}AD^{-1}$, $D^{-1}b$, and the diagonal of D^{-1} as the scaling vector. The solution from the iterative solver will be y, and the user will have to multiply this by D^{-1} to recover x, the original solution to $Ax = b$. The $D^{-1}y$ multiply is, of course, an element by element multiplication of two vectors.

3.2.2 Matrix Row Re-arrangement (`ROWF`)

The only constraints on the *ia,ja* data structure imposed by this package is that the diagonal of each row must exist, and should appear as the first element of each row (where a row is stored in `ja(ia(r))` to `ja(ia(r+1)-1)`). The matrix row re-arrangement routine places the diagonals at the start of the rows. This routine should be called before running the preconditioner routines, unless your matrix setup satisfies the criteria. If you wish to change the matrix between phases (for example, if you wished to use a different scalar matrix as a preconditioner than the incomplete factorization of the one you are solving) then the new matrix should be processed with this routine before being passed to WATSIT-1 preconditioning or solution phase subprograms.

3.2.3 Matrix Structure Check (`CHEK`)

The matrix structure check is used internally by the preconditioner data structure analysis phase of WATSIT. It verifies that the matrix structure is in the form required by WATSIT, and reports the first five errors found. If an ordering vector is included in the list it is checked also (see the language specific sections for the exact definition of the ordering vector).

This routine is provided for the convenience of WATSIT users for verifying their matrix generation code. Note that a structurally sound matrix may still not be solvable.

3.3 Preconditioning Routines

There are, conceptually, two phases to the preconditioning process. The first is the construction of an internal data structure, called the *analyze* phase in this manual. The second phase, *factor*, actually creates the numeric incomplete factorization. Once this is done, the actual solution of the $Ax = b$ system may be attempted.

3.3.1 Analyse and Create the Static Data Structure (ANLY)

This routine creates the internal static data structure for the incomplete factorization of the matrix. The factorization may be level-based only, with the fill level determined on an element by element basis.

This routine (and the drop-tolerance version below) optionally allows a red/black element system reduction before incomplete factorization. This process groups all matrix element rows into two sets, the red points and the black points, so that each set has no internal connections. So the off-diagonal entries of a black row would all correspond to red rows, and vice versa. The red points are eliminated from the set, and the black points used to construct the preconditioner and solve. The final stage produces the answer for the original system. This process often saves time with matrices that arise from five (in 2D) and seven (in 3D) point operators for partial differential equations. If a valid matrix re-ordering vector is passed to this routine, then the first element in the ordering vector becomes a red point.

This phase also performs a matrix structure check (which can be overridden, as detailed in the parameters section). The main matrix may be changed after this phase, but errors in the structure might not be detected, and may cause unexpected results. Checking can be turned on in later phases.

The WATSIT-B version of this routine can over-ride much of the analysis of the preconditioner for a second or later matrix if only the block sizes have changed. So if the sparsity pattern of a new matrix is the same as the one just analyzed, but the block sizes (defined in the `neqa()` matrix) have changed, the red/black reduction and ILU sparsity pattern analysis phase can be skipped. Internal pointers still need to be set, hence the routine must still be called, but it will run very quickly.

3.3.2 Numeric Incomplete Factorization (FCTR)

Use this routine to perform the numeric phase of the incomplete factorization. This is provided separately so that if several matrices with the same data structure to be are created, the preconditioning stage can, optionally, skip the re-analysis and re-creation of the static data structure.

3.3.3 Analyse and Factor with Drop Tolerance (ANFC)

This routine only exists for scalar matrices. Drop tolerance preconditioning removes elements from the preconditioner factorization based on some measure of how small they are. Since "small" is a concept that is not well defined for block elements, a block drop tolerance has not been implemented.

Drop tolerance preconditioning involves the numeric factorization of the matrix during the process of creating the static data structure, this routine combines the analyze and factor phases. After this routine, the numeric incomplete factorization routine may be called on its own to refactor the matrix, but since the data structure will not be changed, there is no guarantee that the drop tolerance ILU is being done consistently.

See Section 3.3.1 for a note about red/black ordering, and matrix structure checking.

3.4 Solution Routines

At present, several CG-like acceleration techniques are combined into a single solution routine. Currently available are CG [HS52], OrthoMin, CGS [Son89], GMRES [SS86], and CGSTAB [vdV92].

3.4.1 Solve Using CG-like Acceleration (`SOLV`)

Once the initialization and preconditiong phases are complete, the actual matrix may be solved for a given source vector using this routine. Although the actual acceleration technique is not specified until this phase, some workspace allocations specified in the analyze phase affect which acceleration may be used later on since some acceleration techniques require more space than others. This routine will return with an error if the workspace allocations are insufficient for the specified acceleration method.

For scalar matrices, a new matrix and/or right hand side may be solved, retaining the old preconditioner, by constructing the changes and calling the solve routine again. Because of certain modifications made to block main matrices for speed during the numeric factorization phase of WATSIT-B, any new right hand side or matrix must be numerically factored before being handed to the solution phase.

If the `wginfo` variable of the `COMMON` block `WATSI1` (see Section 4.1) is set to 1 or higher then convergence information is printed during the solution. Note that the residual given in this information is the preconditioned residual over the black points in the solution, not the residual of the final solution.

3.5 Information Routines

Since it is inadvisable for users to attempt to access the internal data structures of WATSIT directly, the following information routines are provided.

3.5.1 Pointers to Internal Data Structure (`DATA`)

This routine returns the pointers to the *ia,ja* storage of the internal preconditioner matrix. Also, information about the current allocation of the work arrays is provided, from which the user may determine what area of each main work array is unused. Use this information at your own risk.

3.5.2 Error Printing and Explanation Routine (`ERRP`)

Use this routine to generate a text explanation of an error, either to a specified I/O unit, or to a text string parameter you provide. This routine is used internally when error reporting is specified in the output parameters in the global variable area.

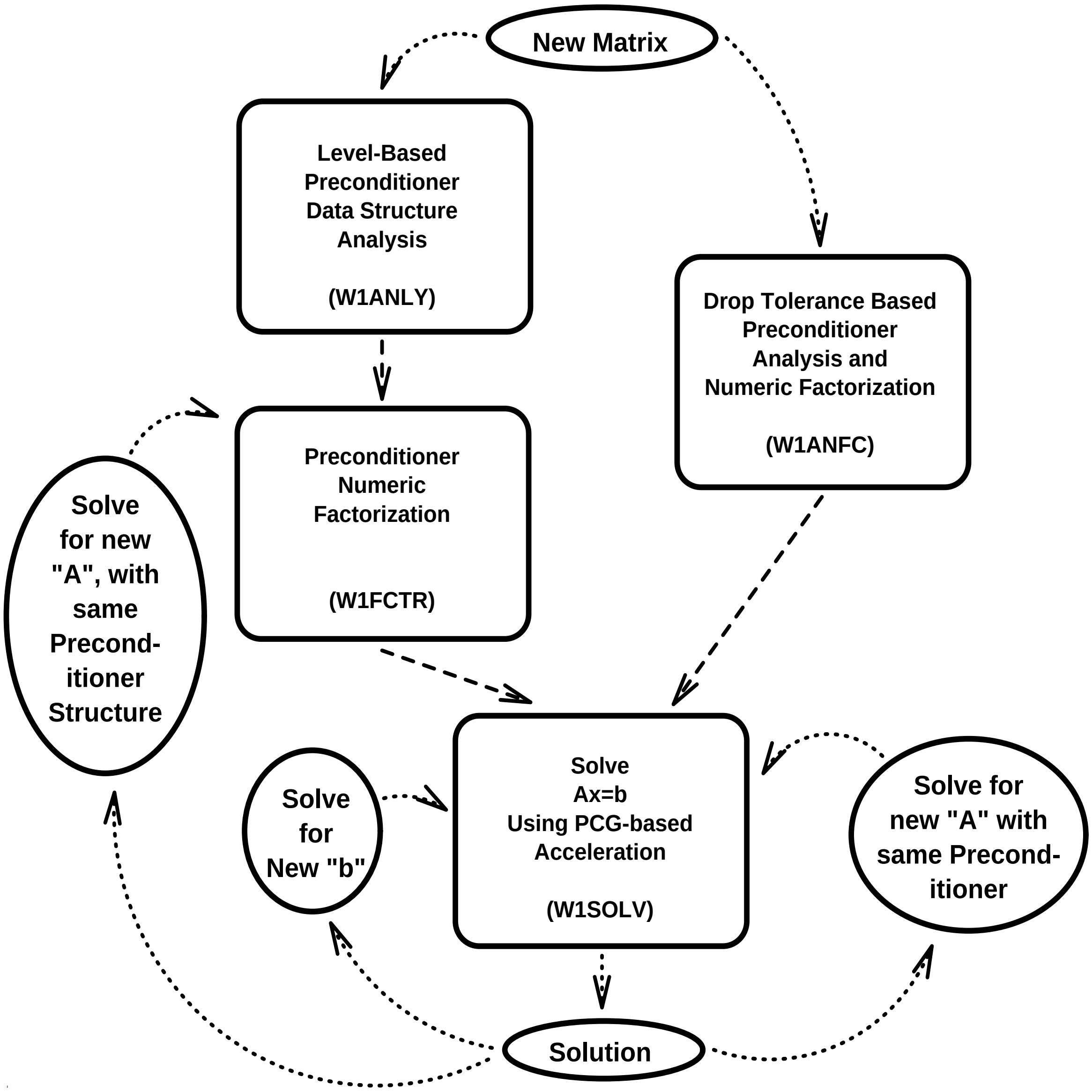

Figure 4: WATSIT-1 scalar matrix solution allowed sequence of operations.

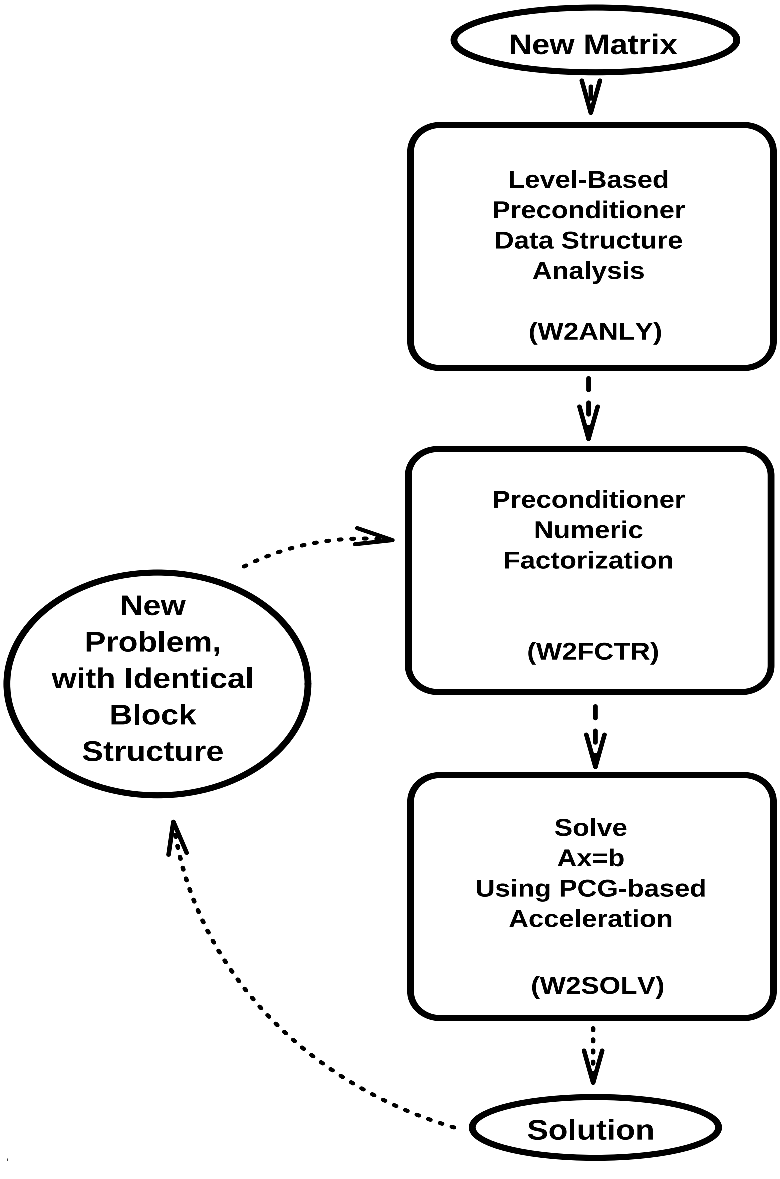

Figure 5: WATSIT-B block matrix solution allowed sequence of operations.

4 WATSIT-1 and WATSIT-B Common Subroutines

Every attempt has been made to keep WATSIT-1 and WATSIT-B conceptually similar. The practical upshot of this is that they share a `COMMON` blocks, and share a set of routines in common. These are documented in the following sections.

4.1 Common Blocks `WATSI1` and `WATSI2`

There are two common blocks used together in both WATSIT-1 and WATSIT-B. Both of these are fully defined in the file `watsit12.com`, which should be included in user programs. The first block contains variables relating to the output of information, errors, and debugging details from the various routines. The block is defined as follows.

```
      INTEGER wgunit,wginfo,wgdebg
C
      COMMON /WATSI1/ wgunit,wginfo,wgdebg
```

Variable Definitions:

`wgunit` Unit number for output. Must be between 1 and 99 inclusive, default is 6 because Unix uses this for standard output.

`wginfo` Level of infomation messages printed
- $\leq$ -2 No printing
- = -1 Error messages only
- = 0 Warning and error messages (default)
- = 1 Convergence history
- = 2 Convergence history, fill, and important scalars
- = 3 User-called subprograms print entry messages plus #2
- = 4 All subprograms print entry messages plus #3

`wgdebg` Debug information level
- = 0 No printing (default)
- = 1 Scalars printed, W1CHEK run at each stage. Most entry messages.
- = 2 Vectors printed, all entry messages, plus #1.

The second common block is strictly informational. Its contents are updated at various phases of the solve, but never used by any WATSIT routine. Thus if the user changes any of them the solve will not be affected. The common block is structured as follows.

```
      INTEGER wgrow,wgiwku,wgrwku,wgfill,wgblak,wgnitr,wgrest,wgcnvc
      DOUBLE PRECISION wgdiag,wgrmsi,wgrmsf,wgohm
C
      COMMON /WATSI2/ wgdiag,wgrmsi,wgrmsf,wgohm
      COMMON /WATSI2/ wgrow,wgiwku,wgrwku,wgfill,wgblak
      COMMON /WATSI2/ wgnitr,wgrest,wgcnvc
```

Variable Definitions:

`wgdiag`	Smallest diagonal in the preconditioner (normalized by max. in row). This is available for WATSIT-1 only.
`wgrow`	Row where the small diagonal is (orig. ordering). This is available for WATSIT-1 only.
`wgrmsi`	Initial l_2 norm of linear residual
`wgrmsf`	Final l_2 norm of linear residual
`wgohm`	Coefficient in last update of solution
`wgiwku`	Amount of IWork actually used.
`wgrwku`	Amount of RWork actually used.
`wgfill`	Number of elements in the preconditioning matrix. In both versions of the solver this is the length of the `ja()` array. Thus in WATSIT-B this is the number of block elements in the preconditioner, and in WATSIT-1 this is equal to the size of the `a()` array.
`wgasiz`	Size of the a() array, or the number of scalar elements in the preconditioner matrix. This is equal to `wgfill` for WATSIT-1.
`wgblak`	Number of black points allocated in the sparsity pattern.
`wgbleq`	Number of scalar equation rows in the preconditioner. Equal to `wgblak` in WATSIT-1.
`wgnitr`	Number of iterations take in the solve
`wgrest`	Number of restarts required
`wgcnvc`	Convergence criteria satisfied at convg. May be summed if more than one is satisfied. = 1 Residual reduction = 10 Change < Tolerance vector

4.2 `W1INIT` — Global Parameter Initialization

```
SUBROUTINE w1init
```

Parameters

None

4.3 `W1DATA` — Pointers to Internal Data Structure

```
      SUBROUTINE w1data(rliwkt,rliwks,rliwke,rlrwkt,rlrwks,rlrwke,
     1                  rnblak,rnbleq,rnjaf,rnaf,rpiaf,rpjaf,rpkaf,
     2                  rpdiag,rpaf,rpkbls,mwork)
C
      INTEGER rliwkt,rliwks,rliwke
      INTEGER rlrwkt,rlrwks,rlrwke
C
      INTEGER rnblak,rnbleq,rnjaf,rnaf,rpiaf,rpjaf,rpkaf
      INTEGER rpdiag,rpaf,rpkbls
C
      INTEGER mwork(50)
```

<u>Parameters</u>

`rliwkt`	Total size of IWORK
`rliwks`	First available element of IWORK
`rliwke`	Last available element of IWORK
`rlrwkt`	Total size of RWORK
`rlrwks`	First available element of RWORK
`rlrwke`	Last available element of RWORK
`rnblak`	Number of black points in preconditioner sparsity pattern.
`rnbleq`	Number of scalar equations in preconditioner.
`rnjaf`	Length of preconditioner ja() and a() arrays
`rnaf`	Length of af() array
`rpiaf`	Pointer to ia() array of preconditioner in IWORK
`rpjaf`	Pointer to ja() array of preconditioner in IWORK
`rpkaf`	Pointer to ka() array of preconditioner in IWORK (for WATSIT-B only, zero for WATSIT-1)
`rpdiag`	Pointer to diag() array of preconditioner in IWORK
`rpaf`	Pointer to a() array of preconditioner in RWORK
`rpkbls`	Pointer to kvlst() array of preconditioner in IWORK (for WATSIT-B only, zero for WATSIT-1)
`mwork`	Master work array of the matrix about which you wish information.

4.4 W1ERRP — Error Printing and Explanation Routine

```
      SUBROUTINE W1ERRP( ierrin, idest, cexpl, ierror )
C
      INTEGER ierrin, idest
C
      CHARACTER cexpl*80
```

<u>Parameters</u>

`ierrin` Error code to be interpreted

`idest` Where to send the error message. Value must be between -2 and 99 inclusive
- = 0 Just return a character string
- = -1 Use WRITE (wgunit,*)
- = -2 Use PRINT *
- > 0 Use WRITE (idest,*)

`cexpl` CHARACTER*80 string for error explanation

5 Using WATSIT-1 Subroutines

For the general details of the function of each user-callable routine, refer to Section 3. The sections that follow give the exact calling parameters for each of the user-callable routines in WATSIT-1. For the convenience of the user, the complete set of parameter declarations, and calling templates, is provided in the file `wats1dec.inc`. This file may be included in programs to speed up the process of programming with WATSIT-1. For access to the common blocks, users may include `watsit12.com` in their programs.

5.1 Parameters Common to Many Routines

There are a number of parameters that will be used in several of the following routines. For the sake of being concise, these are listed below, and not in the details of the routine parameters that follow. Both their declarations, and an explanation of the purpose and valid values of the variables follow.

```
      INTEGER neq,nja
      INTEGER ia(neq+1)
      INTEGER ja(nja)
C
      INTEGER lorder(neq)
C
      DOUBLE PRECISION a(nja)
      DOUBLE PRECISION b(neq)
C
      INTEGER liwork
      INTEGER lrwork
C
      INTEGER iwork(liwork)
      DOUBLE PRECISION rwork(lrwork)
      INTEGER mwork(50)
C
      INTEGER ierror
```

Definitions:

`neq`	Number of equations in the main matrix.
`nja`	Number of elements in the a() and ja() matrices.
`ia()`	*ia* array of *ia,ja* pair (see Section 2.2).
`ja()`	*ja* array of *ia,ja* pair.
`a()`	Matrix coefficient array.
`b()`	Source term, or right hand side of the $Ax = b$ equation.
`lorder()`	Matrix reordering vector such that `lorder(`*new ordering*`) =` *old ordering*. This is a symmetric permutation of the matix. Note that it is a common mistake to pass the inverse of this. If the first element of the array is zero (i.e. `lorder(1) = 0`) then the ordering array is ignored.
`liwork`	Length of the `INTEGER` work array `iwork`.
`lrwork`	Length of the `DOUBLE PRECISION` work array `rwork`.
`iwork()`	`INTEGER` work array.
`rwork()`	`DOUBLE PRECISION` work array.
`mwork()`	Master work array.
`ierror`	Integer error/status code. For a full explanation see Section 2.4, or decode the error with the error printing routine `W1ERRP`. $= 0$ Routine completed successfully < 0 Routine finished with a warning > 0 Routine finished with an error

Notes

The values `liwork` and `lrwork` are not directly calculable; they only reach their maximum values after the analyze and factor phases have constructed the preconditioner. Their upper bounds are, however, given by the expressions

$$\texttt{liwork} = 3\,neq + 3\,nblack + \mathit{fill} + 2$$

and

$$\texttt{lrwork} = 3\,nblack + 2(north + 1)(nblack) + north + ischek + \left[2\,north^2 + 2\,north + 2\,nblack\right] + \mathit{fill}$$

where

- *neq* is the total number of equations,
- *nblack* is the number of black points in the system,
- *fill* is the number of elements in the preconditioner,

- *north* is the number of orthogonal vectors for GMRES and OrthoMin, which is equal to 2 for the other acceleration methods,
- *ischek* is the number of iterations for the stall checking (see the `W1SOLV` routine and Section 2.5), and
- the amounts in brackets ([]) only apply when using GMRES.

Note that the `WATSI2 COMMON` block will return how much of the space was actually required. In the current version, the allocated space runs from the start of the workspaces to the point indicated in the common block, however, to be safe, one may use the W1DATA routine to determine exactly what portion of the space is un-allocated.

5.2 `W1SCAL` — Scale a Matrix

```
      SUBROUTINE w1scal(neq,nja,ia,ja,symm,a,b,dscal,ierror)
C
      INTEGER symm
      INTEGER dscal(neq)
```

Parameters not listed in Section 5.1

`symm`	Indicates the type of scaling to be performed. = 0 for non-symmetric scaling. = 1 for symmetric scaling.
`dscal(neq)`	Vector of length (neq) to return scaling factors. Note that these will be equal to the square root of the diagonal if the matrix is to be symmetrically scaled, equal to the diagonal for non-symmetric scaling.

5.3 `W1ROWF` — Matrix Row Re-arrangement

```
      SUBROUTINE w1rowf(neq,nja,ia,ja,a,ierror)
```

Parameters not listed in Section 5.1

None

5.4 `W1CHEK` — Matrix Structure Check

```
      SUBROUTINE w1chek(neq,nja,ia,ja,lorder,iwktmp,ierror)
C
      INTEGER iwktmp(neq)
```

Parameters not listed in Section 5.1

`iwktmp()` Integer workspace of length `neq`

5.5 `W1ANLY` — Analyse Static Data Structure

```
      SUBROUTINE w1anly(neq,nja,ia,ja,redblk,level,lorder,liwork,lrwork,
     1                  mwork,iwork,rwork,ierror)
C
      INTEGER redblk,level
```

Parameters not listed in Section 5.1

`redblk` Red/black reduction parameter.
 =0 No red/black reduction.
 =1 Operate on reduced system.
`level` Level of fill to be preserved in ILU factorization.
`ierror` Set this parameter equal to -10 on entry to suppress a matrix structure check.

5.6 `W1FCTR` — Numeric Incomplete Factorization

```
      SUBROUTINE w1fctr(ia,ja,a,imilu,mindia,mwork,iwork,rwork,ierror)
C
      INTEGER imilu
      DOUBLE PRECISION mindia
```

Parameters not listed in Section 5.1

`imilu` Row sum preservation parameter (Modified ILU).
 = 0 Drop terms as usual.
 = 1 Add sum of dropped quantities to diagonal.
`mindia` Minimum row-maximum scaled diagonal permitted in the ILU factorization. In other words, if the preconditioner numerical factorization produces a diagonal element that is smaller than `mindia` times the maximum absolute value in the row, the routine will return an error.

5.7 W1ANFC — Analyse and Factor with Drop Tolerance

```
      SUBROUTINE w1anfc(neq,nja,ia,ja,a,redblk,imilu,drtype,mindia,
     1                  thresh,lorder,liwork,lrwork,mwork,iwork,rwork,
     2                  ierror)
C
      INTEGER redblk,imilu,drtype
      DOUBLE PRECISION mindia,thresh
```

Parameters not listed in Section 5.1

redblk Red/black reduction parameter.
=0 No red/black reduction.
=1 Operate on reduced system.

imilu Row sum preservation parameter (Modified ILU).
= 0 Drop terms as usual.
= 1 Add sum of dropped quantities to diagonal.

drtype Type of drop tolerance to perform. The superscript k refers to the k^{th}, or curren, stage of the ILU factorization where rows up to $(k-1)$ have been eliminated. Note that while factoring row i, that $a_{jj}^{(k)}$ where $j > i$ is not available, (using the ordering of the preconditioner).
= 1 Drop $a_{ij} <$ thresh $\times |\text{RowMax}^{(1)}|$
= 2 Drop $a_{ij} <$ thresh $\times |a_{ii}^{(1)}|$
= 3 Drop $a_{ij} <$ thresh $\times |a_{jj}^{(1)}|$
= 4 Drop $a_{ij} <$ thresh $\times \sqrt{|a_{ii}^{(1)}||a_{jj}^{(1)}|}$
= 5 Drop $a_{ij} <$ thresh $\times |\text{RowMax}^{(k)}|$
= 6 Drop $a_{ij} <$ thresh $\times |a_{ii}^{(k)}|$
= 7 Drop $a_{ij} <$ thresh $\times |a_{jj}|$ where $a_{jj} = a_{jj}^{(k)}$ if $j < i$ and $a_{jj} = a_{jj}^{(1)}$ if $j > i$
= 8 Drop $a_{ij} <$ thresh $\times \sqrt{|a_{ii}^{(k)}||a_{jj}|}$ where $a_{jj} = a_{jj}^{(k)}$ if $j < i$ and $a_{jj} = a_{jj}^{(1)}$ if $j > i$

mindia Minimum row-maximum scaled diagonal permitted in the ILU factorization. In other words, if the preconditioner numerical factorization produces a diagonal element that is smaller than mindia times the maximum absolute value in the row, the routine will return an error.

thresh Drop tolerance threshhold.

5.8 `W1SOLV` — Solve Using CG-like Acceleration

```
      SUBROUTINE w1solv(ia,ja,a,b,iaccel,north,itmax,ischek,rstall,
     1                  rprec,rtoler,x,mwork,iwork,rwork,ierror)
C
      INTEGER iaccel,north,itmax,ischek
      DOUBLE PRECISION rstall,rprec,rtoler(neq)
C
      DOUBLE PRECISION x(neq)
```

<u>Parameters not listed in Section 5.1</u>

- `iaccel` Type of acceleration to use.
  - = 0 CG (for symmetric matrices only)
  - = 1 OrthoMin
  - = 2 CGS
  - = 3 CGSTAB-P
  - = 4 GMRES
- `north` Number of orthogonal vectors for GMRES and OrthoMin. For CGS and CGSTAB-P it is the maximum number of stall/restart cycles permitted.
- `itmax` Maximum number of iterations allowed.
- `ischek` Stall check parameter, except for GMRES, where it is the extra residual check parameters. The stall check is done by comparing the current residual, to the residual `ischek` iterations ago. For GMRES, a residual reduction check is done every `north`, and `ischek` iterations. See Section 2.5.
- `rstall` Range for stall. If the relative change in the residual is less than `rstall` then a stall, not convergence, is indicated. For GMRES, the check is done on the l_2 norm of the update to the solution. See Section 2.5.
- `rprec` Residual reduction precision required.
- `rtoler` Absolute tolerance array. Convergence is indicated when the relative update to the solution is less than rtoler_i.
- `x` Solution to the problem $Ax = b$.

6 Using WATSIT-B Subroutines

For the general details of the function of each user-callable routine, refer to Section 3. The sections that follow give the exact calling parameters for each of the user-callable routines in WATSIT-B. For the convenience of the user, the complete set of parameter declarations, and calling templates, is provided in the file `wats2dec.inc`. This file may be included in programs to speed up the process of programming with WATSIT-B. For access to the common blocks, users should include `watsit12.com` in their programs.

6.1 Parameters Common to Many Routines

There are a number of parameters that will be used in several of the following routines. For the sake of being concise, these are listed below, and not in the details of the routine parameters that follow. Both their declarations, and an explanation of the purpose and valid values of the variables follow.

```
      INTEGER neq,nelem
      INTEGER nja,ndima
      INTEGER ia(nelem+1)
      INTEGER ja(nja)
      INTEGER ka(nja+1)
C
      INTEGER neqa(nelem)
      INTEGER kvlst(nelem+1)
C
      INTEGER lorder(nelem)
C
      DOUBLE PRECISION a(ndima)
      DOUBLE PRECISION b(neq)
C
      INTEGER liwork
      INTEGER lrwork
C
      INTEGER iwork(liwork)
      DOUBLE PRECISION rwork(lrwork)
      INTEGER mwork(50)
C
      INTEGER ierror
```

Definitions:

`neq`	Total number of scalar element rows in the main matrix. Must equal $\texttt{kvlst(nelem)} + \texttt{neqa(nelem)} - 1$.
`nelem`	Number of block element rows in the main matrix.
`nja`	Number of elements in the a() and ja() matrices. Must be $\geq \texttt{ia(nelem} + 1)$
`ndima`	Length of the a() array. Must be $\geq \texttt{ka(nja} + 1)$
`ia()`	*ia* array of *ia,ja,ka* structure (see Section 2.2).
`ja()`	*ja* array of *ia,ja,ka* structure.
`ka()`	*ka* array of *ia,ja,ka* structure.
`neqa()`	*neqa* array of the number of scalar element rows per block element row.
`kvlst()`	*kvlst* array of pointers to the start of the unknowns corresponding to each block element row.
`lorder()`	Matrix reordering vector such that `lorder(`*new ordering*`) =` *old ordering.* This is a symmetric permutation of the matix. Note that it is a common mistake to pass the inverse of this. If the first element of the array is zero (i.e. `lorder(1) = 0`) then the ordering array is ignored.
`a()`	Matrix coefficient array.
`b()`	Source term, or right hand side of the $Ax = b$ equation.
`liwork`	Length of the `INTEGER` work array `iwork`.
`lrwork`	Length of the `DOUBLE PRECISION` work array `rwork`.
`iwork()`	`INTEGER` work array.
`rwork()`	`DOUBLE PRECISION` work array.
`mwork()`	Master work array.
`ierror`	Integer error/status code. For a full explanation see Section 2.4, or decode the error with the error printing routine `W1ERRP`. $= 0$ Routine completed successfully < 0 Routine finished with a warning > 0 Routine finished with an error

Notes

The values `liwork` and `lrwork` are not directly calculable; they only reach their maximum values after the analyze and factor phases have constructed the preconditioner. Their upper bounds are, however, given by the expressions

$$\texttt{liwork} = 3\,neq + 3\,nblunk + \mathit{fill} + 2$$

and

$$\texttt{lrwork} = 3\,nblunk + 2(north + 1)(nblunk) + north + ischek + \left[2\,north^2 + 2\,north + 2\,nblunk\right] + \mathit{fill}$$

where

- *neq* is the total number of scalar equations, black or red,
- *nblunk* is the number of black scalar equations in the system,
- *fill* is the number of elements in the preconditioner,
- *north* is the number of orthogonal vectors for GMRES and OrthoMin, which is equal to 2 for the other acceleration methods,
- *ischek* is the number of iterations for the stall checking (see the W2SOLV routine and Section 2.5), and
- the amounts in brackets ([]) only apply when using GMRES.

Note that the WATSI2 COMMON block will return how much of the space was actually required. In the current version, the allocated space runs from the start of the workspaces to the point indicated in the common block, however, to be safe, one may use the W1DATA routine to determine exactly what portion of the space is un-allocated.

6.2 W2ROWF — Matrix Row Re-arrangement

```
      SUBROUTINE w2rowf(nelem,nja,ndima,neqa,ia,ja,ka,a,rsize,rswap,
     1                  ierror)
C
      INTEGER rsize
      DOUBLE PRECISION rswap(rsize)
```

Parameters not listed in Section 6.1

rsize	Size of swapping work space.
rswap	Swap space at least $\texttt{neqa}_{max}{}^2$ in size, where $\texttt{neqa}_{max}$ is the largest value in the neqa matrix.

6.3 W2CHEK — Matrix Structure Check

```
      SUBROUTINE w2chek(nelem,nja,ndima,neqa,ia,ja,ka,lorder,iwktmp,
     1                  ierror)
C
      INTEGER iwktmp(nelem)
```

Parameters not listed in Section 6.1

iwktmp()	Integer workspace of length nelem

6.4 W2ANLY — Analyse Static Data Structure

```
      SUBROUTINE w2anly(nelem,nja,ndima,neqa,ia,ja,ka,redblk,level,
     1                  blkupd,lorder,liwork,lrwork,mwork,iwork,rwork,
     2                  ierror)
C
      INTEGER redblk,level,blkupd
C
      INTEGER ierror
```

<u>Parameters not listed in Section 6.1</u>

`redblk` Red/black reduction parameter.
=0 No red/black reduction.
=1 Operate on reduced system.

`level` Level of fill to be preserved in ILU factorization.

`blkupd` Block size update parameter. If the *ia,ja* structure is unchanged, but `neqa()` values have changed, this allows the skipping of some of the analysis phase.
= 0 Do full analysis.
= 1 Update block parameters only.

`ierror` Set this parameter equal to -10 on entry to suppress a matrix structure check.

6.5 W2FCTR — Numeric Incomplete Factorization

```
      SUBROUTINE w2fctr(neqa,ia,ja,ka,a,kvlst,b,mwork,iwork,rwork,
     1                  ierror)
```

<u>Parameters not listed in Section 6.1</u>

None

6.6 W2SOLV — Solve Using CG-like Acceleration

```
      SUBROUTINE w2solv(neqa,ia,ja,ka,a,kvlst,b,iaccel,north,itmax,
     1                  ischek,rstall,rprec,rtoler,x,mwork,iwork,rwork,
     2                  ierror)
C
C The term mwork(mneq) is the total number of scalar equations as
C stored in the mwork workspace.
C
      INTEGER iaccel,north,itmax,ischek
      DOUBLE PRECISION rstall,rprec,rtoler(mwork(mneq))
C
      DOUBLE PRECISION x(mwork(mneq))
```

Parameters not listed in Section 6.1

`iaccel`	Type of acceleration to use. = 1 OrthoMin = 2 CGS = 3 CGSTAB-P = 4 GMRES
`north`	Number of orthogonal vectors for GMRES and OrthoMin. For CGS and CGSTAB-P it is the maximum number of stall/restart cycles permitted.
`itmax`	Maximum number of iterations allowed.
`ischek`	Stall check parameter, except for GMRES, where it is the extra residual check parameters. The stall check is done by comparing the current residual, to the residual `ischek` iterations ago. For GMRES, a residual reduction check is done every `north`, and `ischek` iterations. See Section 2.5.
`rstall`	Range for stall. If the relative change in the residual is less than `rstall` then a stall, not convergence, is indicated. For GMRES, the check is done on the l_2 norm of the update to the solution. See Section 2.5.
`rprec`	Convergence criterion precision required.
`rtoler`	Absolute tolerance array. Convergence is indicated when the relative update to the solution is less than rtoler_i.
`x`	Solution to the problem $Ax = b$.

7 Advanced User's and Developer's Information

The following information is provided for users wishing to make more advanced or more flexible use of the WATSIT solvers.

7.1 Adherance to Standards and Conventions

There are three divergences from standard Fortran-77 in the following programs.

1. Lower-case letters are used in variable names.

2. Many of the routines use an `INCLUDE` statement that makes a set of parameters available from the files `watsit12.com` and `watsit12.com`. If your Fortran-77 compiler does not support the use of the `INCLUDE` statement, then replace all occurences of the lines

   ```
   INCLUDE 'watsit12.com'
   INCLUDE 'watsitmw.com'
   ```

   with the contents of the files.

3. Almost all current machine architectures store `DOUBLE PRECISION` variables in 16 bytes, and `INTEGER` variables in 8 bytes. The data structure analysis routines make the assumption that an `INTEGER` variable requires *less* storage than a `DOUBLE PRECISION` variable, however, no assumptions are made to the ratios of required storage. In other words, the solver will use, at one point, a certain amount of the real work space as integer work space, but will only assume that a `DOUBLE PRECISION` array of length N will allocate enough space to store an `INTEGER` array of length N. Similarly, `REAL` variables are assumed to take less space than `DOUBLE PRECISION` variables.

All of the routines provided are `SUBROUTINE` type subprograms. For efficiency, these routines make use of the `Level 1 BLAS` (Basic Linear Algebra Subprograms), which are provided with the package. If your machine has a set of BLAS optimized for the particular architecture, then some benefit will be derived from compiling your program with these, rather than the generic routines provided.

7.2 Modifying the Code for `REAL` Matrix or Preconditioner Elements

It is quite conceivable that `DOUBLE PRECISION` matrix, right-hand side, or preconditioner elements might not be required for many applications. The code for the factor phase has been arranged to allow an easy switch of these items to single precision `REAL`. To perform this switch, the following changes must be made to the code, and the code recompiled. Note that only the matrix coefficient array `a`, the right-hand side aray `b`, and the preconditioner coefficient array `af` can change size. It is our experience that whilst the matrix values may be single precision, the remainder of the acceleration and computation routines are best left in double precision.

To change the preconditioner perform the following steps.

1. In file `watsitmw.com`, in the last line, there is a `PARAMETER` named `zrealr`. Set this equal to the number of `REAL` variables that will fit in the space of one `DOUBLE PRECISION` variable. This will be equal to 2.0 on most architectures. If you aren't sure, 2.0 is probably a safe choice.

2. In the source files, lines appear that begin with `CRP`. These are "commented out" source lines that cause the solver to work with a `REAL` preconditioner. Comment out the corresponding original lines (which are always grouped between comment lines with the `CRP` lines) by putting `CDP` in the first three columns (this way you can find the lines again easily). Remove the `CRP` letters from the start of the source lines where they appear.

It might be useful to first find all the source lines you have to modify using a pattern searching tool (such as the Unix `grep`) command. Most of the lines, you will notice, involve either dividing or multiplying by the `zrealr` parameter, or involve changing a `DOUBLE PRECISION` key word to `REAL` when defining the size of the `af` array internally.

The changeover of the `a` and `b` arrays is somewhat easier. All declarations of these arrays are on lines by themselves, hence, to make the switchover you need to change all occurences of

```
DOUBLE PRECISION a(
```

or

```
DOUBLE PRECISION b(
```

to

```
REAL a(
```

or

```
REAL b(
```

respectively. If you are using a global search and replace, it is important that you specify the entire string shown above, since there are other variables that begin with "a", or end with "a(", which are important not to change. With the strings above you will only change the declarations for the main array and right-hand side. As with the preconditioner, it is important that internal computations remain in double precision arithmetic.

Note that even if the preconditioner, matrix coefficients, and right hand side (or any combination) are in single precision, the `rwork` `DOUBLE PRECISION` workspace must remain `DOUBLE PRECISION`. In other words, it must actually be an array of `lrwork` `DOUBLE PRECISION` variables, **not** `REAL` variables, so that internal space allocation remains consistent.

If these instructions are not clear enough to allow you to make the change successfully, you should probably consult someone more familiar with Fortran, and the system you are using.

7.3 Internal Coding Conventions

These programming conventions are noted for the use of anyone who wishes to upgrade or modify the code. These have been adhered to in the supported code of the version as distributed, in the interest of producing the most robust and easy-to-read code possible.

Conventions

1. The following naming conventions will be followed internally in WATSIT-1.

   (a) All user-callable routines will be named `W1xxxx` where `xxxx` is a four letter/number designation. The order of subprogram calling parameters in WATSIT-1 will adhere to the conventions listed in Section 7.3.

   (b) All internal routines will be named `W0xxxx` .

   (c) Common blocks will be designated `/WATSIn/`, where `n` is replaced by a number. All common block variables will be designated `WGxxxx`.

   (d) Parameters to `W1xxxx` routines as described in the calling sequence details will follow no conventions, since users may rename these at their discretion.

2. Function and subroutine dummy parameters will be classified in five groups, and set in the following order:

   (a) the objects being operated on (such as the main matrix or preconditioner), which are not modified during the call,

   (b) input options controlling the operation performed by the subprogram, which may or may not be modified,

   (c) resulting output values, except for error or status codes,

   (d) work arrays, defined as such because their contents are undefined upon exit,

   (e) and error and status codes.

   Within each of these groups, parameters will be listed in the order of:

   (a) scalars,

   (b) then arrays, ordered by the number of dimensions.

   Within each of these classifications, parameters will be ordered by type. The order will be

   (a) `LOGICAL`,

   (b) `INTEGER`,

   (c) `REAL`,

   (d) `DOUBLE PRECISION`,

   (e) `COMPLEX`,

   (f) then `CHARACTER`,

   Finally, parameters of the same scalar or array type, will be listed by order of application to the subroutine, order of return, or any other conceptually logical order.

3. Each program file and subprogram is to be preceded by a header. Template headers are given in the files `module_header.f` and `subprogram_header.f`. These should be filled in in detail.

4. The first items following a subprogram header are to be

   (a) The inclusion of the `watsitmw.com` parameter and `watsit12.com` `COMMON` block definitions. Note all parameters declared in the first file start with 'm'. It would be wise to avoid the use of internal variables starting with 'm' in subprograms.

   (b) The definition of subprogram dummy parameters, in the order they appear in the calling sequence. Each parameter classification group is to be defined in lines separated by a comment line from the other groups. Note the file `watsitmw.com` contains `MWORK` work space declarations for inclusion in the code as required.
   Note: To facilitate the easy switchover from `DOUBLE PRECISION` matrix coefficients and right-hand side coefficients, only the matrix coefficients may have the array name `a()`, and only the source array may have the array name `b()`. Furthermore, the declaration of these arrays must be on a line by itself, so the user can perform the switchover operation described in Section 7.2.

   (c) The local variables, with commented explanations as required. All variables must be declared.

   (d) Subprogram internal parameters, i.e. those defined with a `PARAMETER` statement.

   (e) External subprogram calls must be stated. Not all compilers let you get away with not doing this.

5. The first step of the subprogram should check parameters. The `WOICHK` routine validates the `wgunit` variable and returns a non-zero error code if it is invalid. All `W1xxxx` (i.e. user-callable) routines should check to see if `wgunit` is invalid, and return the `WOICHK` error without further printing if it is not. All other parameters may then be checked as far as possible. If an error is detected, set `ierror`. If `wginfo.GE.-1` then print an explanation to unit `wgunit` which begins with the subprogram name, then call `WOERRP` with the error code.

6. If the subprogram is user-callable, and `wginfo.GE.3` then an entry message should be printed. Relevant parameters may be printed with the message. If the routine is internal, then either `wginfo.GE.4` or `wgdebg.GE.1` should cause and entry message.

7. Other error should be handled as they arise, with a message and call to `WOERRP` as with the parameter errors. Errors usually cause a `RETURN` at the point where they occurred. Warnings should be printed if `wginfo.GE.0` and should not cause the subprogram to exit.

8. Pay attention to points where debugging and other such info may be printed. If the user has set `wginfo` or `wgdebg` to print this information, do so in a message that is clear, and begins with the subprogram name.

9. No control parameters may be placed in `COMMON /WATSI2/` block. The contents of this global area is undefined, as far as the WATSIT-1 routines are concerned. User changes this to global information must not affect the solver in any way.

10. Comment as if your life depended on it. It might; you never know if your code is going to be used in the next Airbus or Audi.

11. When the coding is complete, use a Fortran 77 code polisher such as Toolpack/1's `pol` to touch up the code. Make variables lower case, and key words upper case for code readability. The final distributed code is a concatenation of all the various source files. The `watsit12.com` and `watsitmw.com` common block files should remain separate. The BLAS and `d1mach` machine constants routines should also remain separate, since the users may wish to provide versions optimized for their various systems.

References

[HS52] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.

[Son89] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric systems. *SIAM J. Sci. Statist. Comput.*, 10:36–52, 1989.

[SS86] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, July 1986.

[vdV92] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, March 1992.