DEVELOPMENT OF SOFTWARE FOR SOLVING SYSTEMS OF LINEAR EQUATIONS

Karl Westerberg

Engineering Design Research Center

It should be noted that the software described in this document has

been rewritten in C, and has been copiled and tested on a number

of UNIX hardware platforms including Apollo:Motorola, Cray, Dec:Mips, Sun:Sparc, HP:PA.

Contact ascend+ftp@cs.cmu.edu for further information.

1. Abstract

This report discusses the design and implementation of software for solving systems of sparse linear equations. The motivation for this work came from the needs of the ASCEND (Piela, 1989) computer environment for building and solving complex mathematical models.

The original intent was to use an existing software package such as MA28 (from the United Kingdom Atomic Energy Authority, Harwell) (Duff, 1977); however, an initial survey convinced us that a package to suit our requirements for an extremely flexible research tool did not exist. We were looking for software that had been written in a modular fashion, allowing us to change all aspects of the solving process such as matrix representation, and solving algorithm. In response to these requirements, we created our own software package which is described in this report.

The report is divided into four sections, the first of which describes in detail the software structure and implementation of the solver. The second section describes the methods used for testing both the correctness and efficiency of the solver. The third section discusses the development of the solving algorithm, and in the fourth section the solver is compared with two other packages: MA28, and a solver developed by Amoco (Cummings, 1985).

2. Introduction

The linear equation solver is a stand alone software package, designed primarily for solving large sparse systems of linear equations. It was designed with *readability, modularity, flexibility, and portability* being the most important concerns; and speed being of lesser importance. An important requirement was that the solver should support multiple solving strategies. The solver is written in Pascal on an Apollo workstation. To date the code has not been ported to any other machines.

3. Description of the software structure

The module dependency tree is shown in 3-1. A brief description of the function of each module is given below:

1. **linear:** user interface to the linear equation solver.
2. **mio:** input and output of matrices, either to standard input/output or to disk files.
3. **sio:** output of the solution of a system of equations to standard output.
4. **linsol:** solution of the system of linear equations.
5. **gauss:** factorization of a matrix. The factors are then applied to the righthand side (RHS) to

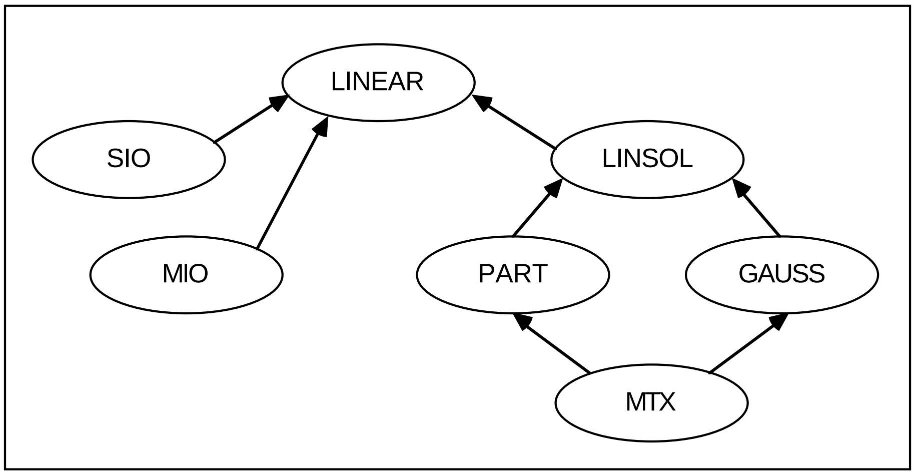

Figure 3-1: Module dependency tree

obtain the solution.

6. **part:** production of an output assignment, which is used to partition a matrix into lower block triangular form.
7. **mtx:** storage and management of matrix information including RHS, equation and variable labels, and row/column permutation.

Each module will be described in fuller detail later. Also incorporated (but not shown in the dependency tree) are the following utility modules:

1. **str:** string manipulation
2. **fmt:** input and output of numbers (real or integer) from or into formatted text.
3. **msg:** management and display of messages.
4. **utl:** graphics primitives.
5. **list:** management of "linked lists." Linked lists combine the advantages of arrays, such as indexed addressing, with the advantages of linked lists, such as flexible size, insertion, and deletion.

Possibly the most important aspect of developing structured software is deciding on how best to decompose the overall task. It seemed natural to us that the task of equation solving should be completely separated from the task of matrix storage. This would allow us to change the matrix representation without having to rewrite the solving procedures. However, this division does not exist in most packages where the goal of better performance has fused matrix representation and solving. As a result, any change to the storage philosophy requires modification of the solver. Additional benefits of separation in our task included improved debugging, readability, and the use of the matrix storage software in other applications.

The solving procedure was decomposed into several modules. For example the reordering algorithm and factorization algorithm were implemented in separate modules. This separation could have been achieved by making them separate procedures within the same module; however, our approach allowed any reordering technique to be paired with any factorization technique, a gain in flexibility.

Furthermore, **linsol** had to manage the structure associated with solving linear equations as well as the translation of the factors and the solution. Therefore, the factorization algorithm was separated out of **linsol** and occupies its own module, **gauss**.

The bulk of the reordering algorithm remains in **linsol**, because the reordering algorithm may affect the particular use of factorization. The procedures used for output assignment and partitioning the matrix (also a part of the reordering algorithm) are stored in their own module. This was done because output assignment and partitioning are generic operations which are used in other applications, e.g., non-linear equation solving depends on output assigning for the purpose of determining structural singularity.

4. Description of each module

In this section the primitive procedures will be discussed, and then the implementation method(s) employed will be described. It should be noted that all identifiers (procedure names, constants, types and variables) associated with a module begin with "<name>$_", where <name> is the name of the module. This has the advantages of enabling the reader to use a particular identifier's name to determine which module it came from, as well as preventing potential naming clashes.

The algorithms described below are those which are currently implemented. The reader is referred to the section on development for a discussion on the evolution of these algorithms.

4.1. Module mtx

4.1.1. Procedures

The following is a list of exported procedures:

mtx$_create_matrix	**Create a zero matrix of zero order**
mtx$_destroy_matrix	**Destroy the matrix and returns the space to the system**
mtx$_set_current_matrix	**Set the current matrix**
mtx$_push_current_matrix	**Set the current matrix, placing the old one on a stack**
mtx$_pop_current_matrix	**Restore previous current matrix from the stack**
mtx$_set_region	**Set the region which restricts the scope of many calls**
mtx$_get_order	**Return the order of the matrix**
mtx$_set_order	**Set the order of the matrix**
mtx$_clear_matrix	**Clear the entire matrix**
mtx$_clear_coef	**Clear the coefficient matrix**

`mtx$_clear_rhs`	Clear the rhs
`mtx$_clear_label`	Clear the labels
`mtx$_clear_perm`	Reset the row/column permutations
`mtx$_copy_matrix`	Copy the entire matrix to another
`mtx$_copy_coef`	Copy the coefficient matrix to another
`mtx$_copy_rhs`	Copy the rhs to another matrix
`mtx$_copy_label`	Copy the labels to another matrix
`mtx$_copy_perm`	Copy the row/column permutations to another matrix
`mtx$_del_zr_in_row`	Remove numerical zeros from a row
`mtx$_del_zr_in_col`	Remove numerical zeros from a column
`mtx$_get_element`	Return an element from the matrix
`mtx$_set_element`	Set the value of an element in the matrix
`mtx$_get_next_col`	Return the next entry (by row) which is non-zero
`mtx$_get_next_row`	Return the next entry (by col) which is non-zero
`mtx$_get_row_max`	Return the maximum entry of a given row
`mtx$_get_row_min`	Return the minimum non-zero entry of the given row
`mtx$_get_col_max`	Return the maximum entry of a given column
`mtx$_get_col_min`	Return the minimum non-zero entry of the given column
`mtx$_row_count`	Return the number of non-zeros in the given row
`mtx$_col_count`	Return the number of non-zeros in the given column
`mtx$_get_rhs`	Return an element from the matrix rhs
`mtx$_set_rhs`	Set the value of an element in the matrix rhs
`mtx$_eqn_to_row`	Return the row corresponding to the equation
`mtx$_row_to_eqn`	Return the equation corresponding to the row
`mtx$_var_to_col`	Return the column corresponding to the variable
`mtx$_col_to_var`	Return the variable corresponding to the column
`mtx$_get_eqn_label`	Return the label of an equation
`mtx$_get_var_label`	Return the label of a variable
`mtx$_set_eqn_label`	Set the label of an equation
`mtx$_set_var_label`	Set the label of a variable
`mtx$_increment_diagonal`	Increment the diagonal of the matrix by a given amount
`mtx$_transpose`	Transpose the matrix
`mtx$_swap_row`	Swap the two given rows
`mtx$_row_parity`	Return the parity of the row permutation
`mtx$_swap_col`	Swap the two given columns
`mtx$_col_parity`	Return the parity of the column permutation
`mtx$_mult_row`	Multiply the given row by a given amount
`mtx$_mult_col`	Multiply the given column by a given amount
`mtx$_add_row`	Add a multiple of one row to another
`mtx$_add_mult_row`	Add a linear combination of many rows to one row
`mtx$_eliminate_row`	Perform row operations which zero the left of the diagonal
`mtx$_add_col`	Add a multiple of one column to another
`mtx$_view`	View current matrix in bitmap format

4.1.2. Introduction

The MTX module is responsible for the storage and maintenance of a matrix. The matrix is currently resticted to a maximum order of 5000, (this upper limit can be changed by changing the constant mtx$_max_order_c) and must be square (although zero rows or zero columns are permitted). The structure of a matrix also includes a RHS vector (same length as the order of the matrix), equation and variable labels, and row/column permutation information.

Before a matrix can be used, it must be created using mtx$_create_matrix. This procedure will create the internal structure of a matrix and will return a handle (pointer) which is used to reference the matrix in subsequent procedure calls. The internal structure of the matrix can be destroyed by calling mtx$_destroy_matrix with the handle. This procedure returns a handle value of NIL to mark the fact that it no longer exists.

Most of the routines described below operate on the "current matrix," rather than taking a matrix handle as input. This can improve readability since most of the time the user wants to perform many actions on one matrix at a time. Before operating on a particular matrix, the user must declare it to be the current matrix using mtx$_set_current_matrix. For situations in which the user wants to work simultaneously with several matrices, we have implemented the concept of a stack of matrices. In this case the current matrix is set by pushing it to the top of the stack using mtx$_push_current_matrix. Lower matrices in the stack can be made current by repeatedly calling mtx$_pop_current_matrix.

Many routines act on a limited region of a matrix. A region is any rectangular portion of a matrix defined by a row and column range. Rather than specifying a region with every procedure call, a "current region" is set once using mtx$_set_region. When the current matrix is changed, the current region is reset to include the entire matrix. When the current matrix is saved by mtx$_push_current_matrix, the current region is saved along with it.

4.1.3. Implementation

A description of the data structures in MTX is given in appendix I. Notice that the row/column headers, RHS vector, and equation/variable labels are referenced through pointers in the main record. Initially, these pointers are set to NIL, so space need not be allocated for these large arrays until they are actually needed. Since most matrix applications access the permutation arrays and most operations require access to them, these arrays are built into the main record. When these arrays are created, they are initialized only up to the current matrix order. If the order of the matrix is increased, the size of the permutation arrays is correspondingly increased.

The matrix is stored using a fishnet structure as diagrammed in appendix I. It should be noted that a non-zero element is indexed by its equation/variable numbers and not its row/column numbers. Header arrays are indexed in a similar manner. The reason for this is that the fishnet structure does not change when two rows or two columns are swapped. A disadvantage is that the row/column numbers passed in from the outside world must be translated into equation/variable numbers before using the permutation arrays. However, in general, the cost of translation is negligible compared to the cost of swapping row header pointers, and then changing the row index for all non-zero elements in the row. Since our favored solution algorithm (called RANKI, see module **gauss** for details) uses many row and column swaps, the equation/variable indexing is more efficient.

The RHS vector is stored as a full vector. It is also indexed by equation number instead of row number, for the same reason as above.

The equation and variable labels are stored as arrays of strings, and indexed by equation and variable numbers.

Because users reference the matrix by row and column numbers, the permutation arrays are heavily used and are built into the main record to save time and enhance readability. Since all of the other data structures are indexed by equation/variable numbers, only the permutation arrays are modified when a row or column swap takes place, making row and column swaps very efficient. Four permutation arrays are maintained : equation $\rightarrow$ row, variable $\rightarrow$ column, row $\rightarrow$ equation, and column $\rightarrow$ variable. Although all of the permutation information is contained in the first two arrays, four arrays are maintained; although the last two can be calculated from the first two, the calculation is costly. Row and column parities are stored with the permutation arrays. The parity of a permutation is defined to be the parity (evenness or oddness) of the number of elementary swaps required to convert the identity permutation to the current permutation. Parity is uniquely determined by the permutation, and row and column parities are used to calculate the sign of the determinant of the matrix.

The mtx$_add_row procedure has received the most attention during development, because it consumes the most time during solving. The original implementation scanned the source row and for every non-zero element in the row attempted to locate the corresponding element in the target row. If the element existed its value was adjusted, otherwise, the element would have to be created and assigned a value. This algorithm was slow (of order $z^2$2, where z is the number of non-zeros per row). In the second implementation column indices were maintained (actually the variable indices) in increasing order so that both source row and target rows could be scanned simultaneously. The drawback with maintaining indices in increasing order was that element creation became slower because the new element had to be correctly positioned. However, this change was still an improvement, since in many cases the fill-in was relatively low. In our next approach, the target row was first expanded into an array of pointers, where NIL indicates the element does not exist. Then, for each non-zero in the source row, the corresponding non-zero element in the target row was searched for, by using the variable number as an index into the array of pointers. If the pointer was NIL, then the target element did not exist and was created.

Otherwise, the element the pointer was pointing to would be updated. The problem of initializing a very large array to NIL was taken care of by making the array static, and resetting only those pointers in the array which corresponded to target row elements. This implementation eliminated the need to preserve the indices in increasing order. When implemented, this ran about as fast as the second attempt; however, we still felt that there was room for improvement.

With the RANKI algorithm a row is eliminated by adding multiples of many rows to a single target row. Therefore, the target row is only expanded once and the array of pointers cleared once. The primitive mtx$_mult_row was created to perform this task. Finally, the entire row elimination process was enclosed in a matrix primitive (mtx$_eliminate_row).

4.2. Module part

4.2.1. Procedures

The following is a list of exported procedures:

```
part$_find_chain            Find an output assignment and place it on the diagonal
part$_partition             Permute the matrix into BTF
part$_destroy_block_list    Destroy partition data
```

4.2.2. Introduction

This module is used to form an output assignment and to partition a matrix into lower block triangular form. The algorithm used for forming the output assignment is a depth first search (Duff, 1981). Each row is assigned in order by first scanning the row for a non-zero in an unassigned column. If this non-zero is found, it is moved via column permutation to the diagonal. Otherwise, the algorithm checks each assigned column incident to the given row, and attempts to reassign the corresponding row to a new column (using the same algorithm), so that the non-zero can be moved to an unassigned column. If this depth first search terminates without assigning the row, then the matrix is structurally singular up to that point and the row is permuted to the bottom.

The algorithm used for partitioning a matrix into lower block triangular form is the work of Tarjan (Tarjan, 1972), and is based on finding the strong components of the directed graph corresponding to the non-zero structure of the matrix.

4.3. Module gauss

4.3.1. Procedures

The following is a list of exported procedures:

`gauss$_set_pivot_calc`	**Set the address of the pivot determining procedure**
`gauss$_find_inverse`	**Find the inverse of the given region**
`gauss$_apply_inverse`	**Apply the inverse to the RHS**
`gauss$_solve_wo_inverse`	**Solve the region without keeping an inverse**
`gauss$_why_singular`	**Determine why a system of equations is singular**

4.3.2. Introduction

This module is responsible for transforming a problem matrix into a solution matrix. Each procedure operates on a given region of a matrix, so the user could solve only part of the matrix. The procedures generally act directly on the matrix, and destroy its original structure, that is, the resulting factors (equivalent to finding an inverse) overwrite the coefficient matrix. Solving is completed by applying these factors to the RHS vector whose original values are overwritten with the solution. In the case of pivoted variables, the slot in the solution vector contains the corresponding value. In the case of unpivoted equations the corresponding slot contains the residual of the equation. Unpivoted variables are assigned a value of zero. On completion of solving (either with or without keeping the factors), the rank, determinant, and smallest pivot used are stored in three global variables.

The why-singular procedure is used to analyze singular problems. Why-singular will determine the linear combination of the independent rows, which equals each of the dependent rows. These linear combinations are stored in the matrix itself. The why-singular analysis requires the matrix factors to be calculated, and will destroy them on completion of the analysis.

The solving algorithm that has been implemented is RANKI (Stadtherr and Wood, 1984). RANKI systematically eliminates spiked columns by first eliminating the left of every spike row with the rows above it, and then performing a number of row and column swaps to remove the spike. A spiked column is a column with non-zeros above the diagonal. A spiked row is a row whose corresponding column is a spiked column. (A demonstration of the RANKI algorithm is given in appendix II.) We decided to use RANKI instead of LU factorization because in problems with a relatively small numbers of spikes RANKI creates significantly less fill-in than LU factorization, resulting in better performance. RANKI is very good for sparse matrices (which have been reordered), but is not very good with full matrices or sparse matrices which have not been reordered.

When saving the "inverse," one need only save the multipliers used for elimination of each spike row in that same row. It turns out that when the "inverse" is saved, the original matrix is factored into UL, where U is upper triangular (for the independent rows) and L is lower triangular (for the independent rows).

4.4. Module linsol

4.4.1. Procedures

The following is a list of procedures:

`linsol$_init_matrix_system`	**`Initialize the matrix system`**
`linsol$_destroy_matrix_system`	**`Destroy the matrix system`**
`linsol$_get_input_matrix`	**`Return the input matrix handle`**
`linsol$_get_coef`	**`Return a coefficient of the input matrix`**
`linsol$_set_coef`	**`Change a coefficient of the input matrix`**
`linsol$_get_rhs`	**`Return an RHS element`**
`linsol$_set_rhs`	**`Change an RHS element`**
`linsol$_coef_modified`	**`Signal the solver that the coef matrix was modified`**
`linsol$_rhs_modified`	**`Signal the solver that the rhs matrix was modified`**
`linsol$_sacrifice`	**`Set the sacrifice flag`**
`linsol$_reorder`	**`Permute the input matrix based on structure`**
`linsol$_copy_reorder`	**`Copy reordering information`**
`linsol$_solve`	**`Solve the matrix system`**
`linsol$_solve_wo_inverse`	**`Solve, but does not keep an inverse`**
`linsol$_get_rank`	**`Return the rank of the matrix`**
`linsol$_get_determinant`	**`Return the determinant of the matrix`**
`linsol$_eqn_pivoted`	**`Return whether or not an equation was pivoted`**
`linsol$_get_smallest_pivot`	**`Return the smallest valid pivot used`**
`linsol$_get_var_value`	**`Return the value of a variable`**
`linsol$_get_eqn_residual`	**`Return the residual of an equation`**
`linsol$_eqn_to_var`	**`Return variable corresponding to the equation`**
`linsol$_var_to_eqn`	**`Return equation corresponding to the variable`**
`linsol$_determine_why_singular`	**`Compute why singular`**
`linsol$_get_linear_combination`	**`Return why-singular coefficients`**
`linsol$_forget_why_singular`	**`Release memory required to remember why singular`**

4.4.2. Introduction

This module is responsible for solving systems of linear equations. It relies on module **gauss** to perform elimination, and module **part** to output assign and partition the coefficient matrix. Having partitioned the coefficient matrix **linsol** is responsible for reordering it prior to solving. The algorithm used for reordering is SPK1 (Mark A. Stadtherr and E. Stephen Wood, 1984). SPK1 uses row and column counting techniques to permute each irreducible block with the goal of minimizing the number of spikes, thereby reducing the factorization time. An example of a reordered matrix is shown in figure 4-1.

To isolate the data structures representing the linear system of equations, the module exports a handle to the user. The handle is of type linsol$_matrix_system_t, which is defined inside the module, and hidden from the user. Procedures are provided to initialize and destroy the structure of the matrix system.

Figure 4-1: A matrix that has been reordered using the SPK1 algorithm

Linsol gives the user the ability to perform input/query requests on the matrix coefficients and RHS; however, further information can be obtained by requesting the matrix handle by calling linsol$_get_input_matrix, and using the procedures contained in the **mtx** module. If the input matrix is modified using **mtx** procedures, then **linsol** must be informed by using procedures linsol$_coef_modified, and linsol$_rhs_modified.

Once the coefficients and RHS have been set, the matrix system can be solved, and the solution analyzed. This module supports the solution of the why-singular problem (see module **gauss** for details). Unlike **gauss**, when the system is solved, the coefficient matrix is preserved (unless the sacrifice flag is set by linsol$_sacrifice). When why-singular is calculated, the inverse is preserved (unless the sacrifice flag is set).

4.4.3. Implementation

The matrix system is stored as three matrices (the input matrix, solution matrix, and why-singular matrix); solution parameters (rank, determinant, and smallest pivot used); flags (coefficient matrix modified, RHS modified, and sacrifice); and reordering information (partition data, and reordering validity). The flags are used to determine whether or not to make a backup copy of the input matrix when solving, to make a backup copy of the solution matrix when calculating why-singular, and to determine whether the inverse needs to be recalculated or whether the coefficient matrix actually needs to be reordered.

5. Methods of testing: validation and efficiency

In order to test each module independently, a utility program called **general tester** was written. General tester allows the user to create matrices, load matrices from file, save matrices to file, input or display a matrix in a number of formats. General tester will also allow the user to perform a number of high and low level operations ranging from solving a system of linear equations to swapping two specified rows. During development, this program allowed us to methodically check each procedure for errors.

Although we have not been willing to sacrifice modularity for speed, the large problems that we intend to solve dictate that speed be an important consideration. To evaluate the efficiency of a program it is important to be able to determine its execution time (especially CPU time). General tester has options for displaying the elapsed time (both actual and CPU) for some procedures. These facilities enable a crude analysis to be performed; however, in order to revise the code most efficiently, information is required at the procedure and statement level.

Apollo provides a software package called DPAK (Apollo Computer Inc., 1988) which is ideal for this sort of analysis. In particular, this package includes a program DPAT which, when run in parallel with a given program, will interrupt the program at regular time intervals, and record the traceback information. Afterwards, DPAT uses this information to determine how much time was spent in or under any procedure (time spent "under the procedure" refers to time spent in procedures which were called by this procedure), and will display this information in one of two formats: ranked or hierarchical. In ranked format, each procedure is listed in descending order of the amount of time spent in or under that procedure. In hierarchical format, the top-level procedures are listed, and each procedure is followed by the procedures it calls. Times listed are the time spent in the given procedure relative to its parent procedure, and the time spent relative to the total time. The hierarchical format is useful for understanding how procedures relate to one another. Note that in both cases, the time spent under the procedure is included in the time listed for that procedure. This is appropriate, since the time required for the procedure to perform its task includes the time spent under it. We made extensive use of DPAK in the development of our solving software.

5.1. Creation of example matrices

To test the solver adequately, we needed to generate large systems of sparse linear equations. To this end, a random matrix generator was written which was designed to generate a matrix randomly and write it to a specified file. The order of the matrix and the total number of non-zeros is specified by the user. The row counts are first determined by randomly assigning each non-zero to a row. Then, within each row, each of the non-zeros are assigned a column number at random. Finally, each non-zero in the matrix and every RHS is assigned a value at random. Matrices created this way are generally structurally singular; structurally non-singular matrices are created by first specifying the diagonal, and then determining the remaining non-zeros as before. At the end, the rows are randomly permuted to break up the diagonal. These structurally non-singular matrices turn out almost always to be numerically non-singular as well.

Seventeen examples matrices were generated (see appendix III). Matrix 1 was created by an early version of the random matrix generator which only produced 0 and 1 entries, and was used mainly for testing the reordering techniques. Matrices 2-4 were created by the random singular matrix generator. Matrices 5-17 were created by the random non-singular matrix generator. Matrices 2-6 were used mainly during development. Matrices 8-17 were created specifically for the final comparison among the solvers.

We are not certain to what extent these matrices are representative, but they did provide a common basis for comparison.

6. Development

We began by using "conventional" algorithms for matrix storage and equation solving. The matrix was stored as an **n**x**n** array of values, and the linear equations were solved using LU factorization with partial pivoting. No reordering algorithm appeared in version 1 of **linsol**.

The intention was to have a working modular structure. A front end was quickly written and we had a working version of a linear equation solver in about 2 weeks. It was adequate for small problems.

The next step was to rewrite the matrix software. Version 2 of **mtx** used a sparse matrix fishnet structure to store the matrix. The RHS and the fishnet headers were both indexed by row/column. Then **gauss** and **linsol** were rewritten. Version 2 of **gauss** used RANKI instead of LU factorization, and version 2 of **linsol** incorporated the SPK1 reordering algorithm. The first timing tests were performed on example matrix 1, which was already reordered. The two versions of **gauss** were compared. RANKI was quicker than LU by a factor of about 2.5 when RANKI kept its inverse, and by a factor of 6 when RANKI didn’t keep its inverse. Furthermore, RANKI created half the fill-in of LU when RANKI kept its inverse, and by less than a tenth when it didn’t keep its inverse. From then on, we adopted RANKI as the solving algorithm of choice, and efficiency was measured in comparison to RANKI, particularly, when RANKI did not keep its inverse, since this had the fastest time.

Although RANKI ran faster than LU factorization, its performance was still too slow. It took 42 seconds to solve a 400x400 matrix. DPAT was used to obtain a more detailed analysis of where the solver was spending its time. The results showed that more than one third of the time was spent swapping rows and columns. This was because the coefficient matrix and RHS were indexed using row/column numbers. As a result, when two rows were exchanged, in addition to updating the permutation arrays, the RHS values and header pointers had to be exchanged, and the row indices of all of the non-zeros in both rows had to be changed. It became apparent that indexing by equations and variables would be more efficient, because in that case swapping would only involve changing the permutation arrays.

The DPAT results also showed that almost one fifth of the time was spent adding a multiple of one row to another. The then current version of mtx$_add_row scanned the target row for a non-zero corresponding to each non-zero in the source row, which required time of order z^2 (z is the number of non-zeros per row). In order to reduce the time order, the user could maintain the fishnet structure so that the variable/equation numbers appeared in increasing order. Then, time required to add rows would reduce to order z; however, the time order of creating an element would then increase from order 1 to order z. Because RANKI creates a small amount of fill-in, the time associated with creating new elements should be correspondingly small.

We incorporated both of the above ideas into version 3 of **mtx**, and decreased the solution time to 35 seconds. The time required for row and column swaps was reduced considerably. However, the cost of translation from row/column numbers to equation/variable numbers lessened the benefits. Mtx$_get_next_col increased from 9% to 17% and mtx$_get_element increased from 6% to 10%.

At this point, we began to investigate different compiler options. In particular, we examined the effect of the following options: optimization level, CPU type, and the inclusion of subscript checking for arrays. All of these options had a noticeable effect on execution time. Specification of CPU reduced solution times by 15-30%. Specification of full optimization, rather than none, reduced solution times by 8-20%. Elimination of subscript checking reduced solutions times by 8-15%. The reduction for an option was dependent on the values chosen for the other two. Using the best set of options, matrix 3, a 400x400 matrix, was solved in 14.3 seconds, and Matrix 4 (a singular 1000x1000 matrix) was solved in 94.9 seconds.

When the non-singular matrix generator was written, we immediately created matrix 5, supposedly a 1000x1000 structurally non-singular matrix; however, the output assigner claimed that the matrix was structurally singular. Furthermore, the solver (correctly) claimed that the matrix was numerically non-singular. Apparently there was a bug in the output assigner. This bug was corrected, but another was detected immediately afterwards in the reordering procedure, causing the program to crash only if the output assignment was complete. The bugs together had caused the reordering algorithm to perform poorly, and after they were corrected, solution times decreased considerably. Matrix 3 was solved in 4.8 seconds, matrix 4 was solved in 10.6 seconds, and matrix 5 was solved in 37.7 seconds. The reason for the discrepancy between the 1000x1000 matrices has to do with the size of the largest irreducible block.

We obtained a DPAT profile of the solver and determined that about 60% of the time was spent eliminating rows. This led us to wonder whether the cost of maintaining the linked lists in order of increasing equation/variable number was too high. So we searched for alternatives to mtx$_add_row, which did not require the lists to be ordered, and devised the following algorithm. First, the target row would be expanded into an array indexed by variable number of pointers to the non-zero elements or NIL. Then, for every non-zero in the source row, the corresponding element in the target row would be located using the array. If the array element was NIL, then the corresponding non-zero in the target row did not exist yet, and would be created. If the array element was not NIL, then it would point to the appropriate non-zero in the target row and the non-zero would be updated. This left the problem of resetting the array to NIL every time mtx$_add_row was called. That problem was solved by making the array static, initializing it to NIL at the beginning of program execution, and then resetting only the array elements corresponding to non-zeros in the target row on completion of mtx$_add_row, since all of the other elements were already NIL. This procedure was still of order z (z is the number of non-zeros per row), but it was no longer necessary to maintain order in the linked lists. Version 4 of **mtx** used the array version of mtx$_add_row and no longer required the linked lists to be in order. At this point, the solver with version 4 of **mtx** ran slightly slower than with version 3 of **mtx**.

However, we felt that there was room for improvement. In order to eliminate a row, multiples of many rows have to be added to one row. Therefore, the target row only needed to be expanded into the array once when eliminating the row. Also, the array needed to be cleared only once. This resulted in the creation of a new matrix primitive, mtx$_add_mult_row, which added multiples of many rows to one specific row. This primitive only expanded the target row and cleared the array once, making it more efficient than calling mtx$_add_row multiple times. With this new primitive, **gauss** ran slightly faster using version 4 of **mtx** than it did using version 3.

The elimination of a row using the rows above it in the matrix created temporary fill to the left of the diagonal. This row tended to get full during the elimination, and would have been more efficiently handled if it were expanded into a full vector of reals. Expanding the row into an array also removed the problem of eliminating the row in descending order. So **gauss** had to handle adding multiples of the previous rows to the given row for all columns to the left of the pivot, and mtx$_add_row handled the remaining columns. This required communicating with **mtx** using primitive scanning and fetching routines, rather than the more powerful routines, such as mtx$_add_row. Therefore, in the interest of saving time, the entire action of eliminating a row became a matrix primitive. The solution time the reduced by slightly more than half. Except for minor stylistic modifications, this is the way the linear equation solver is currently implemented.

7. Comparison with other solvers

During the course of development, we compared our solver to two others. One of them, MA28, is a Harwell routine written in Fortran. The other one was developed at Amoco, and is written in Apollo Pascal with a modular structure not unlike our own. Both of these solvers use LU factorization with Markowitz threshold pivoting. Our solver uses the RANKI algorithm. Both the Amoco solver and our solver use a linked list representation for storage of matrices, although the exact implementations differ somewhat. MA28 uses a row pointer/column index storage technique where the non-zeros, row pointers, and column indices are stored in three arrays. All three solvers provide the option of partitioning the matrix into lower block triangular form. Only our solver offers further reordering using the SPK1 algorithm which is a requirement of the RANKI algorithm.

In order to ensure a valid comparison, all programs were compiled with the same (or equivalent) options, run on the same machines, and tested with the same matrices. The total time required for solution was counted for each solver including the time required to partition, reorder, factor, and apply the factors. For the final comparison all of the modules of each solver were compiled using the most efficient options possible for each compiler, and the actual timing was carried out on one machine. All programs were provided with enough workspace to perform without hinder. (This is particularly important for MA28, which will use time-consuming garbage collections to deal with a reduced memory situation.) The time for the comparison was measured in CPU time. A summary of the results is given in appendix IV.

The actual timing was carried out using DPAT. Proper setup programs were required for each of the solvers. In particular, the program had to provide the option of choosing a file from which the matrix was loaded, and then load that matrix and solve it.

Our first comparison with MA28 was made with version 3 of **mtx**. The ratio of our time to MA28's time was 2.3 for a matrix of order 100, 16.8 for a matrix of order 400, and 30.1 for a matrix of order 1000. It appeared as though the execution time for our solver ran on a higher order of magnitude than MA28. However, after the two bugs in the reordering procedure (as described above) were corrected, the increasing ratio disappeared. The ratio ranged from 1.01 to 4.4 depending on the problem (but not the order). As we continued through development, we periodically made comparisons to MA28. With the current implementation the ratio ranges between 0.56 and 2.26. It should be noted that for these tests, our solver did not keep its inverse, and so we are being slightly unfair to MA28. In the final comparison, we make the comparison for both cases, and keeping the inverse just about doubles the solution time.

The results of the final comparison are shown in appendix IV, and can be summarized as follows: MA28 and our solver (no inverse) had similar times which were the lowest. Our solver (with inverse) took between 1.5 and 2 times as long as the lowest; and the Amoco solver took between 2 and 3 times as long as the lowest.

Given that both our solver and the Amoco solver were structured in a similar fashion, written in the same language for the same machine, we wondered why our software seemed to perform consistently better on the test problems. A difference in solving algorithms seemed to be the obvious reason, but why did RANKI outperform LU factorization with the Markowitz threshold pivot selection?

With LU factorization, the coefficient matrix is factored into LU, where L is a lower triangular matrix, and U is an upper triangular matrix. At step k in gaussian elimination, the first k-1 rows of U and the first k-1 columns of L have been computed. A pivot is selected from the remainder of the coefficient matrix (called the active part of the matrix) and moved to (k,k) [row,column]. The kth column below the kth row is then zeroed by adding multiples of the kth row to the rows below it. The original values of the kth column are preserved and are part of L. The kth row becomes the kth row of U. Processing continues until there are no more pivots. There are many schemes for pivot selection; but some are good for numerical stability, while others are good for reducing fill-in.

The pivoting scheme used by Amoco, and probably one of the best pivoting schemes used in conjunction with gaussian elimination on large sparse linear equations, is the Markowitz criterion with threshold. The Markowitz criterion selects as a pivot the element (i,j), which minimizes the product r(i)c(j), where r(i) is the number of non-zeros in row i whose column index is greater than k, and c(j) is the number of non-zeros in column j whose row index is greater than k. The Markowitz criterion minimizes the potential fill-in at each step, and does a reasonable job of reducing total fill-in. The threshold condition requires that selected pivots be a given multiple of the maximum value within its row. The threshold condition is a guard against numerical instability which can result from choosing an extremely small pivot. If the threshold tolerance is made too small, then serious numerical instability can occur. If the threshold tolerance is made too large, then only a few pivot candidates would exist, so the value of r(i)c(j) for the selected pivot would be high, resulting in large fill-in. A typical value of the threshold is 0.1.

With RANKI, spike columns are systematically eliminated resulting in a lower triangular matrix which can be solved by forward substitution. Appendix II contains a demonstration of the RANKI algorithm for a 5x5 matrix. A spike (spike column) is a column with non-zeros above the diagonal. A spike row is a row whose corresponding column is a spike column. (Row j is a spike row if and only if column j is a spike column.) At step k of the elimination, columns 1 to k-1 are not spikes, and the diagonal elements of these columns hold valid pivots. If column k is not a spike and element (k,k) is greater than a given multiple of the largest non-zero in (k, k..n), processing continues to row k+1. Otherwise, elements (k,1..k-1) (i..j is the set of integers between i and j) are eliminated by adding multiples of rows 1..k-1 to row k. Then the largest non-zero in row k is selected as a pivot and is moved into position (k,k). Let t denote the top of the spike in column k. Row k is then moved to row t, and all of the rows between t and k are shifted down one (This can be accomplished by swapping rows j and j-1 for j = k down to t+1.) Column k is permuted in the same way. The symmetric permutation will preserve the diagonal, and hence the pivots. Furthermore, as the example demonstrates, the spike in column k disappears. Processing then continues to row k+1. After all of the spikes are eliminated, the resulting lower triangular matrix is solved by forward substitution.

The low fill-in and low solution times characteristic of RANKI depend on there being relatively few spikes, since only spike rows and rows with bad pivots on the diagonal need to be processed. Therefore, it is imperative that the matrix be reordered before applying the RANKI algorithm into a form with relatively few spikes. We used the SPK1 reordering algorithm to perform this task and it seemed to perform reasonably well.

For RANKI, fill-in can only occur at the intersection of spike rows and spike columns, whereas for LU factorization, fill-in can occur anywhere. Even though with RANKI, virtually all potential fill-in is actually filled in, its fill-in count is much lower than that of LU factorization. For LU factorization, fill-in occurs in the active part of the matrix, and so the density of the active part will increase as elimination proceeds. This causes the rate of fill-in to increase, which in turn causes the density to increase at a faster rate. The proper pivoting scheme will attempt to reduce fill-in as much as possible and maintain the sparsity of the active part of the matrix. However, even with the Markowitz criterion, a point will come in the elimination where the active part of the matrix is almost completely dense. This problem does not occur with RANKI since fill-in is restricted to certain parts of the matrix, so that the rate of fill-in never increases. This is true even in the case where RANKI keeps its inverse. Keeping the inverse requires the storage of dense rows of multipliers; however, these multipliers play no further role in elimination. (The reason our implementation of RANKI is much slower when saving its inverse is because scanning an already eliminated row requires traversing through the saved multipliers; this can be eliminated, but it would require major changes in how the matrix is stored.) Since RANKI creates less fill-in, the matrix remains sparse, and all row operations require less time. Consequently, RANKI requires less time.

Based on DPAT profiles of the Amoco solver, pivot selection required about 50% of the solution time. Theoretically, the product r(i)c(j) must be evaluated for each non-zero (i,j) in the active region. In practice, there are techniques which reduce the actual amount of searching and still yield the least value of r(i)c(j), such as searching rows and columns in increasing order of count and stopping as soon as it is determined that a better value of r(i)c(j) cannot be attained. However, evaluation of the Markowitz criterion required a substantial amount of time. This time was further amplified by loss of sparsity as elimination proceeded.

At first glance it appears that RANKI should be at least an order of magnitude faster than LU factorization. However, testing showed our implementation of RANKI was only 2-3 times faster than LU factorization. The reason was that elimination of a row was an expensive operation for RANKI. Since none of the rows above the row to be eliminated have effectively been eliminated (some of the rows may have been eliminated, but in those cases, the permutations which eliminate the spike effectively restore the non-zeros in that row), each time a row is used to eliminate one non-zero, temporary fill-in occurs, and eventually, the entire row remaining to be eliminated fills in completely. As a result, at a given step many more row operations are required to eliminate the row than is required by LU factorization (at least in the beginning). This can be made clear by looking at the multipliers which result from saving the inverse. Each multiplier in the U matrix corresponds to a row operation. Therefore, beyond a certain number of spikes, RANKI could actually run slower than LU factorization. Fortunately, the problems that our solver is primarily designed to solve generally have very few spikes, and can therefore be solved efficiently by RANKI.

8. Conclusions

Our linear equation solver appears to have achieved an adequate compromise between style and efficiency. The solver is almost completely modular, the sole exception being the **mtx** procedure for eliminating rows. Writing the solver in Pascal gave it a highly structured form which can be easily validated. The solver is also reasonably efficient; we are able to solve a system of equations about as fast as MA28. Therefore we believe that our solver is the best choice for our development of a non-linear equation solver.

I. Representation of matrix structure

Non-zero’s are stored individually in a fishnet structure (linked list.) They are referenced by their equation and variable number which correspond to their row and column number when the matrix is created. In addition, there is a vector of equation headers and a vector of variable headers which point into the matrix. Following equation pointers across a particular equation, will return all non-zeros in that equation. The analogous is true for variables. The pointers are in one direction only (to save space).

```
                            variable headers
                         1         2         3

                         |         |         |
                         |         |         |
                         V         |         V
                                   |
             1 --------> X --------+-------> X
                                   |
                         |         |         |
                         |         |         |
                         V         V         V

equation     2 --------> X ------> X ------> X
headers
                         |
                         |
                         V

             3 --------> X
```

The X’s represent non-zeros. The pointers do NOT have to be in any particular order. The non-zeros are created when they are needed. A non-zero is destroyed whenever both its equation and its variable no longer point to it.

The RHS, the labels, and the coefficient matrix itself are all independently created. Each of the structures is created only when it is used, and is promptly destroyed when the structure is cleared using the clear procedures. The permutation vectors are created when the matrix handle is created.

II. A demonstration of the RANKI algorithm

The following is an example of elimination using the RANKI algorithm on a 5x5 system. The initial matrix has been previously reordered.

```
In the matrices shown below:

         X   Non-zero
        spc  Zero
         0   Zero created by elimination


        1 2 3 4 5      Assuming pivots (1,1), (2,2), and (3,3) are
      1 X              suitable, no action is taken on rows 1-3.  We start
      2   X     X      with k=4.
      3 X   X X
 k--> 4 X X   X
      5   X X X X

        1 2 3 4 5      Row 4 is eliminated using rows 1-3 (actually, only
      1 X              rows 1 and 2 are needed).  Note the fill-in at
      2   X     X      (4,5).  In fact, (4,5) is the only position that
      3 X   X X        could have filled in this stage.
 k--> 4 0 0 0 X X
      5   X X X X

        1 2 3 5 4      Assuming that element (4,5) is larger than (4,4)
      1 X              (after elimination), columns 4 and 5 are swapped in
      2   X   X        order to place (4,5) on the diagonal.  It will be
      3 X   X   X      used as a pivot in the future.
 k--> 4 0 0 0 X X
      5   X X X X

        1 5 2 3 4      The top of the spike now in column 4 is 2.  So row
      1 X              4 is moved to row 2 and rows 2 and 3 are moved down
      4 0 X 0 0 X      1.  The columns are permuted the same way.  The
      2   X X          fourth column is no longer a spike.
      3 X     X X
      5   X X X X

        1 2 3 4 5      Processing now continues to row 5.  For clarity, the
      1 X              row/column indices were renumbered.
      2   X     X
      3   X X
      4 X     X X
 k--> 5   X X X X

        1 2 3 4 5      Row 5 is now eliminated using rows 1-4.  At first
      1 X              glance, it appears that row 1 is not needed.
      2   X     X      However, when row 4 was used to eliminate (5,4),
      3   X X          (5,1) filled in.  This kind of fill-in occurs often,
      4 X     X X      but it is only temporary.  (5,5) is accepted as the
 k--> 5 0 0 0 0 X      pivot because it is the only non-zero left in row 5.

        1 5 2 3 4      The top of the spike in column 5 is 2.  So row 5
      1 X              is moved to row 2 and rows 2-4 move down one.  The
      5 0 X 0 0 0      columns are permuted the same way.  The spike in
      2   X X          column 5 no longer exists.  The resulting lower
      3     X X        triangular matrix can be solved by forward
      4 X X     X      substitution.  Note that only one fill-in occurred.
```

III. Example matrices

matrix	order	nonzeros	rank	largest block
1	400	1400	386	291
2	100	350	98	72
3	400	1400	380	201
4	1000	3500	931	515
5	1000	3500	1000	764
6	1000	3500	1000	568
7	5000	17500	5000	2678
8	50	175	50	40
9	50	175	50	41
10	100	350	100	80
11	100	350	100	77
12	250	875	250	190
13	250	875	250	201
14	500	1750	500	380
15	500	1750	500	385
16	1000	3500	1000	544
17	1000	3500	1000	723

IV. Final comparison

		MA28			Solver/ without inverse (RANKI)			Solver/ with inverse (RANKI)				Amoco solver			
Matrix	Order	partition/ factor	apply factors	total	reorder	solve	total	reorder	factor	apply factors	total	partition	factor	apply factors	total
8	50	0.24	0.03	0.27	0.07	0.07	0.14	0.08	0.08	0.04	0.20	0.26	0.16	0.03	0.45
9		0.15	0.00	0.15	0.04	0.07	0.11	0.09	0.09	0.04	0.22	0.12	0.20	0.00	0.32
10	100	0.40	0.05	0.45	0.19	0.23	0.42	0.23	0.37	0.14	0.74	0.13	0.57	0.04	0.74
11		0.37	0.05	0.42	0.19	0.24	0.43	0.19	0.19	0.14	0.52	0.22	0.61	0.00	0.83
12	250	1.81	0.00	1.81	1.23	0.88	2.11	1.15	1.48	0.43	3.06	0.34	3.88	0.10	4.32
13		2.43	0.10	2.53	1.09	1.04	2.13	1.13	1.57	0.49	3.19	0.24	5.37	0.10	5.71
14	500	6.39	0.20	6.59	3.36	3.36	6.72	3.17	6.77	1.44	11.38	0.44	16.44	0.20	17.08
15		9.59	0.21	9.80	3.42	3.18	6.60	3.21	6.28	1.34	10.83	0.63	20.62	0.21	21.46
16	1000	16.73	0.36	17.09	7.98	8.13	16.11	7.55	19.86	3.65	31.06	1.17	40.23	0.46	41.86
17		23.65	0.51	24.16	11.71	14.18	25.89	11.23	34.47	5.03	50.73	1.09	72.01	0.40	73.50

References

Apollo Computer Inc. (Jun 1988). *Analyzing Program Performance with Domain/PAK* (Tech. Rep.)). Apollo Computer Inc.

Daniel L. Cummings. (Mar 1985). *A Sparse Matrix Abstraction* (Tech. Rep.)). Amoco Production Company, Tulsa Research Center.

I. S. Duff. (1977). *MA28 - a set of Fortran subroutines for sparse unsymmetric linear equations* (Tech. Rep.)). AERE, Harwell, England.

I. S. Duff. (Sept 1981). On Algorithms for obtaining Maximum Transversal. *ACM Trans. Math Software*, *7*(3), 315-330.

Mark A. Stadtherr and E. Stephen Wood. (1984). Sparse matrix methods for equation-based chemical process flowsheeting-I. *Computers and Chemical Engineering*, *8*(1), 9-18.

Peter Piela. (1989). *An object-oriented computer environment for modeling and analysis*. Doctoral dissertation, Dept. of Chemical Engineering Carnegie Mellon University.

Mark A. Stadtherr and E. Stephen Wood. (1984). Sparse matrix methods for equation-based chemical process flowsheeting-II. *Computers and Chemical Engineering*, *8*(1), 19-33.

R.E. Tarjan. (1972). Depth-first search and linear graph algorithms. *SIAM J. Computing*, *1*, 146-160.

Table of Contents

List of Figures