Impact Project

Impact Centre
The University of Melbourne
153 Barry Street, Carlton
Vic. 3053 Australia
Phones: (03) 341 7417/8
Telex: AA 35185 UNIMEL
Telegrams: UNIMELB, Parkville

IMPACT is an economic and demographic research project conducted by Commonwealth Government agencies in association with the Faculty of Economics and Commerce at The University of Melbourne and the School of Economics at La Trobe University.

SPARSE MATRIX METHODS FOR COMPUTABLE GENERAL EQUILIBRIUM MODELS OF THE JOHANSEN CLASS

by

K.R. Pearson and Russell J. Rimmer
La Trobe University

Preliminary Working Paper No. OP-43 Melbourne November 1983

The views expressed in this paper do not necessarily reflect the opinions of the participating agencies, nor of the Commonwealth Government

ISBN 0 642 52458 0

CONTENTS

SPARSE MATRIX METHODS FOR COMPUTABLE GENERAL EQUILIBRIUM MODELS OF THE JOHANSEN CLASS

by

K.R. Pearson and Russell J. Rimmer*

1. INTRODUCTION

The equations of a computable general equilibrium (CGE) model may be written as

$$F(Z) = 0 , \qquad (1.1)$$

where F is, in general, a nonlinear function. CGE models of the Johansen class are solved by first linearizing F, near a known solution of (1.1), in terms of percentage changes in the variables Z. (1.1) is then replaced by the matrix equation

$$Dz = 0 , \qquad (1.2)$$

where z is a vector of percentage changes and D is an $m \times n$ matrix. In general

$$n > m ,$$

so that values for $(n - m)$ components of z must be set exogenously, and (1.2) may be solved to obtain values for the remaining m quantities. When z has been partitioned into a vector z_2, consisting of exogenous

components, and a vector z_1 of components to be determined from (1.2), then (1.2) may be written as

$$Az_1 = -Bz_2 \quad , \tag{1.3}$$

where A is an m × m matrix, z_1 is m × 1, B is m × p (p = n - m) and z_2 is p × 1. In the case of the ORANI[1] model of the Australian economy a system of equations (1.3) with

$$m \approx 300 \quad \text{and} \quad p \approx 300$$

must currently be solved.[2] For models of this size computation of solutions absorbs large amounts of computer time, and our aim is to make significant reductions in these times. In this paper we discuss ways of accomplishing this.

Our first recommendation is to avoid matrix inversion when solving large systems of linear equations. Instead, techniques known as "LU methods" should be preferred. Section 2 is devoted to a discussion of these techniques which only take about 25 per cent of the CPU times used by inversion methods. Further CPU savings may be possible by using sparse matrix methods. We outline how this is done in section 3.

Sparse matrix packages save CPU time by ignoring zeros when performing matrix operations, and so the fewer nonzeros in A then the greater will be the savings. As the packages operate on nonzeros only, then only the nonzero entries of A need to be stored. Consequently, if the proportion of nonzeros in A is low then significant savings in time and storage will be accomplished. To some extent these savings are offset by the need to store the row and column position in A of each nonzero

element, and the need to check the positions of these entries often throughout a computation. For a matrix with a high proportion of nonzeros these overheads will erode the savings made by ignoring the zeros in A. As we report in section 4, this was found to be the case when the Harwell Laboratories sparse matrix package[3] was applied to randomly generated matrices of the size and sparsity of ORANI, where the percentage of nonzeros is around 50 per cent. However, randomly generated matrices are unlikely to have the special structure that ORANI displays.

In section 4 we compare the performance of the Harwell code with that of two other algorithms - neither of which distinguishes between zero and nonzero elements - in solving (1.3) for:

(a) randomly generated matrices A of different dimensions, with the density of nonzeros being either 10 per cent or 50 per cent;

and

(b) matrices generated by the stylized Johansen,[4] MO[5] and ORANI models.

A significant feature of our results is that the Harwell code solved ORANI more quickly than either of the other algorithms - a result which could not have been anticipated from the results for randomly generated matrices. This suggests that by moving to a version of the matrix A for ORANI which contains less than 10 per cent nonzeros, and has an even more structured pattern of zeros, it may be possible to obtain further savings of computer resources. But more importantly it may be possible to save considerable research and computing time in the development of Johansen style models by using sparse matrix methods.

To see this, consider for a moment the evolution of the ORANI model from the system of many millions of equations, as described in chapter 3 of DPSV, into a collection of about 300 equations[6] to be solved numerically. To obtain a system of this size, the approach of the builders of ORANI was to find a sub-matrix of $A^{-1}B$ giving the response of selected endogenous variables (i.e., components of z_1 in (1.3)) with respect to those variables in z_2 which are to be varied by a particular user. This involved eliminating a large number of quantities and prices from the original ORANI equations, and absorbing many others into composite variables. Because there was a large amount of elementary algebraic manipulation to be performed, the reduction to a final system with $m \approx 300$ was performed in two stages[7]. First, the ORANI equations were reduced to a condensed system with $m = 2635$ and $n = 6087$.[8] Each coefficient that appears in the condensed system has a clear economic interpretation, and so the formation of equations in this system can be checked easily. Even with $m = 2635$ this condensed form of ORANI was still too large for (1.3) to be solved conveniently using standard routines, so a second round of eliminations was undertaken to form the final system with $m \approx 300$. Unfortunately, the coefficients in the system resulting from this second stage are not so easily interpreted, and considerable effort was expended in getting the computer implementation of this round of eliminations correct. Indeed this absorbed a great deal of research time and also considerable computing resources, all of which could have been avoided if a sparse matrix package had been used to solve the condensed system, for which the matrix A in (1.3) has less than 10% of its entries nonzero. This is currently being attempted by the authors using a VAX 11/780 computer. Our plans for further work in this area are discussed in section 5.

The rest of the paper is organized as follows. Section 2 is devoted to a discussion of the number of arithmetic operations required to solve systems of linear equations using different methods. The implications of sparsity in the matrix A of these systems is taken up in section 3. In section 4 we compare the performance of the Harwell sparse matrix package and two other routines in trials on the systems of equations obtained from three economic models, as well as using systems where the coefficients appearing in the equations were generated randomly.

2. SOLUTIONS OF SYSTEMS OF LINEAR EQUATIONS

Once the choice of exogenous variables has been made, the equations of a computable general equilibirum model in the Johansen class can be represented by equation (1.3). From this equation it follows that the matrix of elasticities of the model satisfies

$$AX = -B, \qquad (2.1)$$

where the $(i,j)^{th}$ entry of X is the elasticity of the i^{th} endogenous variable with respect to the j^{th} exogenous variable. When X has been computed, the solution z_1 of (1.3) for a given set of exogenous shocks z_2 is obtained from

$$z_1 = Xz_2 \ . \qquad (2.2)$$

In (2.2) X is $m \times p$ and z_2 is $p \times 1$ so that the product of X with z_2 involves mp multiplications and $m(p - 1) = mp - m$ additions. When m and p are roughly equal, and are large, the number of operations required to obtain z_1 from (2.2) is about $2m^2$. For ORANI computations, we can assume that $m \approx p \approx 300$. However, the number of operations involved in solving (2.1) for X is likely to be an order of magnitude greater than $2m^2$, and so will use a significantly greater slice of computer time. In this section we investigate three approaches to calculating X in (2.1), or, more accurately, to calculating the columns X_j of X as the solution of each of the p problems

$$AX_j = -B_j \qquad j=1,2,\ldots,p, \qquad (2.3)$$

where B_j is the j^{th} column of B. Use of (2.3) exploits the fact that for the numerical algorithms discussed here, there is no need to repeat the entire algorithm whenever the right hand side of (2.3) changes. For example, suppose that (2.3) is solved by

(1) inversion of A ,

and

(2) formation of $-A^{-1}B_j$.

Clearly A need only be inverted once, as solutions for all vectors $B_1,\ldots,B_p$ can be obtained by repeating step (2) p times.

2.1 Solution of (2.3) by first inverting A

A^{-1} might be computed from the rule

$$A^{-1} = \text{adj}(A)/\det(A) , \qquad (2.4)$$

where adj(A) is the adjoint matrix of A, and det(A) is the determinant of A (assumed nonzero). Each of the m^2 entries in adj(A) is obtained by evaluating the determinant of an $(m - 1) \times (m - 1)$ matrix. Therefore, evaluation of A^{-1} from (2.4) involves around $2m^2(m!)$ operations when m is large. The number of operations, and therefore the time, required to invert A can be reduced by employing other methods, as we will see in section 2.5. (Indeed we will see that explicit calculation of A^{-1} is unnecessary when solving (2.3).)

2.2 Gaussian elimination and back substitution

Gaussian elimination is just an orderly process for the

elimination of unknowns from a system of equations, and this technique is used to transform A into an upper triangular matrix with nonzero diagonal elements. To illustrate this process consider the following system of linear equations:

$$2x_1 + 4x_2 - 2x_3 = 6 \qquad (2.5)$$

$$x_1 + 2x_2 + 5x_3 = 2 \qquad (2.6)$$

$$4x_1 + x_2 - 2x_3 = 2 \qquad (2.7)$$

This system may be re-written in tabular form as

x_1	x_2	x_3	B_1	perm
2	4	-2	6	1
1	2	5	2	2
4	1	-2	2	3

Tableau 1 ,

where column perm is used to record the location in the tableau of each of equations (2.5) - (2.7). In this first tableau (2.5) is located in row 1, (2.6) is located in row 2, and (2.7) is located in row 3. Now use equation (2.5) to eliminate the variable x_1 from equations (2.6) and (2.7), that is, use row 1 in the tableau to produce zeros in rows 2 and 3 of column 1. To this end multiply row 1 (i.e., equation (2.5) by $\frac{1}{2}$ and subtract it from row 2 (i.e., equation (2.6)). Similarly, multiply row 1 by 2 and subtract the result from row 3. When these operations have been performed the tableau becomes:

x_1	x_2	x_3	B_1	perm
2	4	-2	6	1
0	0	6	-1	2
0	-7	2	-10	3

Tableau 2 .

Two features of this second tableau should be noticed.

(i) First, the cells below position 1 in column 1 no longer contain useful information. So we might as well use them to store the numbers used to multiply row 1 when eliminating x_1 from rows 2 and 3. These constants are called the multipliers, and they will be used again later.

(ii) The entry in row 2, column 2 is now zero. As our aim is to produce an upper triangular matrix with nonzero entries along its diagonal, this is a problem. The solution is to interchange rows 2 and 3. That this interchange has taken place is recorded in column "perm", where the entries in rows 2 and 3 of this column have also been interchanged. Now the tableau is:

x_1	x_2	x_3	B_1	perm
2	4	-2	6	1
2	-7	2	-10	3
$\frac{1}{2}$	0	6	-1	2

Tableau 3 .

Usually the next step is to reduce the entries in column 2 below row 2 to zero. In the example this is the same as eliminating x_2 from the equation corresponding to row 3 - but this has already occurred. So for this example the Gaussian elimination is complete, and tableau 3 may be translated into the three equations:

$$2x_1 + 4x_2 - 2x_3 = 6 \qquad (2.8)$$

$$- 7x_2 + 2x_3 = -10 \qquad (2.9)$$

$$6x_3 = -1 \ , \qquad (2.10)$$

To obtain a solution of the original system of equations we appeal to the following theorem:

Repeated application of the operations used above, namely,

(i) interchage of rows,

and

(ii) addition of a multiple of one row to another,

will transform the system of equations (2.3) into

$$TX_j = C_j \ , \qquad (2.11)$$

where T is upper triangular, and where any solution of (2.11) is a solution of (2.3) and vice versa. Therefore in the example the components of the solution vector may be obtained in the order x_3, x_2 and x_1 from (2.10), (2.9) and (2.8) respectively. This calculation is known as "back substitution".

Suppose now that (2.3) has been solved by Gaussian elimination and back substitution in the case j = 1. To solve (2.3) when j=2,3,...,p there is no need to go through the entire elimination again. Rather, knowing the multipliers (stored in the lower diagonal entries as they were calculated in the example) and the pivoting strategy (i.e., the contents of the column vector perm in the example), the operations applied to B_1 may be re-constructed and performed on B_j to obtain a vector C_j, where

$$TX_j = C_j \qquad j=2,3,\ldots,p \quad .$$

The algorithm which computes C_j is called "forward substitution". Hence, for j=2,3,...,p (2.3) can be solved by applying the forward substitution algorithm followed by back substitution.

To illustrate the method of "forward substitution" suppose that we want to solve (2.5) - (2.7) when the right hand side of this equation is

$$B_2 = \begin{pmatrix} 3 \\ 1 \\ 4 \end{pmatrix} .$$

In producing tableau 3 the first row of the table was not altered, and so the initial entry in column B_2 of tableau 3 is just the first entry in the new right hand side C_2 . Next, recall that rows 2 and 3 were permuted to obtain the final tableau. This information is given by column "perm" of tableau 3 where position $\underline{2}$ of perm contains the value $\underline{3}$, and position 3 contains the value 2. Now re-order the entries B_2 according to this scheme so that row $\underline{2}$ of B_2 contains the value 4. To find the "multiplier"

(namely, 2) applied to row 1 to obtain row 2 of the final tableau from row 3 of the initial tableau, look in column 1 of row 2 of the final tableau. Therefore the new entry in position 2 of column B_2 in tableau 3 is

$$4 - 2(3) = -2 \quad .$$

Similarly, the new entry in position 3 of column B_2 in tableau 3 is now

$$1 - \tfrac{1}{2}(3) - 0(4) = -\tfrac{1}{2} \quad ,$$

where the coefficient 0 in the third term on the left hand side is the multiplier which was implicitly applied to row 3 of tableau 2 to obtain row 3 of the final tableau from row 2 of tableau 2. The zero multiplier is located in column 2 of row 3 of the final tableau. (This zero appeared in column 2 of row 3 as a result of interchanging rows 2 and 3 in moving from tableau 2 to tableau 3.)

2.3 Operation counts for Gaussian elimination, and forward and backward substitution.

The results in this section are well known; we rely on page 129 of Conte and de Boor (1972).

The number of operations required to calculate the multipliers used in the Gaussian elimination and to obtain the upper triangular matrix T is

$$\frac{1}{3}m^3 + \frac{1}{2}m^2 - m \quad \text{multiplications and divisions}$$

and

$$m(m - 1) \quad \text{additions.}$$

It takes m divisions, m(m - 1)/2 multiplications and m(m - 1)/2 additions to perform the back substitution algorithm for (2.3). The forward substitution technique takes the same number of operations except that no divisions are required. So together forward substitution and back substitution involve approximately

m^2 multiplications and divisions

and

$m(m - 1)$ additions .

Therefore the combined forward and backward substitution methods involve as many additions as does Gaussian elimination. More important though is the fact that forward and backward substitutions involve only m^2 multiplications and divisions while Gaussian elimination requires more than $\frac{1}{3}m^3$ multiplications and divisions. That is, the combined forward and backward substitutions take an order of magnitude less operations than Gaussian elimination.

2.4 LU decomposition of A

An LU decomposition is an algorithm which factors a square matrix into the product of a lower (L) and an upper (U) triangular matrix. If A can be decomposed as

$$A = LU$$

then the solution of AX = B can be obtained by first solving the problem

$$LY = B \qquad (2.12)$$

and secondly, with Y computed, solving

$$UX = Y \quad . \tag{2.13}$$

Solution of (2.13) corresponds to the "back substitution" process used to solve (2.3) after A had been reduced to an upper triangular matrix. Also, solution of (2.12) corresponds to the "forward substitution" algorithm used to solve (2.3) for j=2,3,...,p.

The triangular matrices L and U can be generated by performing Gaussian elimination on A. For example, consider the matrix A generated from the system of equations (2.5) - (2.7) with rows 2 and 3 interchanged:

$$A = \begin{pmatrix} 2 & 4 & -2 \\ 4 & 1 & -2 \\ 1 & 2 & 5 \end{pmatrix} \quad . \tag{2.14}$$

Now Gaussian elimination produces tableau 3 above. So define L to be the matrix whose lower diagonal elements are the multipliers used to produce tableau 3, and whose diagonal elements are unity. That is,

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{pmatrix} \quad .$$

The matrix U is just the remaining entries in the first three columns of tableau 3, which are the coefficients of the upper triangular system (2.8) - (2.10). So,

$$U = \begin{pmatrix} 2 & 4 & -2 \\ 0 & -7 & 2 \\ 0 & 0 & 6 \end{pmatrix} \quad .$$

Calculation of the product LU verifies that

$$LU = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & -2 \\ 0 & -7 & 2 \\ 0 & 0 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 1 & -2 \\ 1 & 2 & 5 \end{bmatrix} = A .$$

It should be clear that this LU factorization, known as the Doolittle decomposition, involves as many operations in solving (2.3) as does Gaussian elimination with backward and forward substitution.

Another LU decomposition which is frequently used is the Crout decomposition. The only difference from the Doolittle is that now U has all diagonal entries equal to 1 but L has arbitrary diagonal entries. For the matrix A given above in (2.14),

$$LU = \begin{bmatrix} \ell_{11} & 0 & 0 \\ \ell_{21} & \ell_{22} & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \begin{bmatrix} 1 & U_{12} & U_{13} \\ 0 & 1 & U_{23} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 1 & -2 \\ 1 & 2 & 5 \end{bmatrix} .$$

Hence

$$\begin{bmatrix} \ell_{11} & \ell_{11}U_{12} & \ell_{11}U_{13} \\ \ell_{21} & \ell_{21}U_{12}+\ell_{22} & \ell_{21}U_{13}+\ell_{22}U_{23} \\ \ell_{31} & \ell_{31}U_{12}+\ell_{32} & \ell_{31}U_{13}+\ell_{32}U_{23}+\ell_{33} \end{bmatrix} = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 1 & -2 \\ 1 & 2 & 5 \end{bmatrix} \quad (2.15)$$

The entries in L and U can be determined by equating coefficients on each side of (2.15). This can be done in an orderly way by first finding one column of L and then the corresponding row of U; then the next column of L and row of U, etc. For our example,

Col 1 of L : $\ell_{11}=2$, $\ell_{21}=4$ and $\ell_{31}=1$

Row 1 of U : $U_{12}=4/\ell_{11}=4/2=2$ and $U_{13}=-2/\ell_{11}=-1$ (requires $\ell_{11}\neq 0$)

Col 2 of L : $\ell_{22}=1-\ell_{21}U_{12}=1-8=-7$ and $\ell_{32}=2-\ell_{31}U_{12}=2-2=0$

Row 2 of U : $U_{23}=(-2-\ell_{21}U_{13})/\ell_{22}=(-2+4)/(-7)=-2/7$ (requires $\ell_{22}\neq 0$)

Col 3 of L : $\ell_{33}=5-\ell_{31}U_{13}-\ell_{32}U_{23}=5+1-0=6$.

Therefore,

$$A = \begin{bmatrix} 2 & 4 & -2 \\ 4 & 1 & -2 \\ 1 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 4 & -7 & 0 \\ 1 & 0 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ 0 & 1 & -2/7 \\ 0 & 0 & 1 \end{bmatrix} .$$

Algorithms for the Crout decomposition use essentially the same number of operations as the Doolittle decomposition.[9] We use a Crout algorithm in section 4.

2.5 Theoretical comparison of methods

Suppose we wish to solve p systems

$$AX_j = -B_j \qquad j=1,2,\ldots,p$$

for a fixed m × m matrix A, with p different right hand sides $B_1,\ldots,B_p$ (each m ×1). We have two basic alternatives.

(1) Form an LU decomposition of A and then, for each $j=1,2,\ldots,p$, solve for X_j by using forward and backward substitution. It doesn't really matter whether we use the Doolittle decomposition (obtained from Gaussian elimination) or the Crout decomposition as they involve essentially the same number of operations.

(2) Calculate A^{-1} in the quickest possible way and then for each $j=1,2,\ldots,p$, calculate

$$X_j = -A^{-1}B_j \quad .$$

Method 1 (LU method)

As we have seen in section 2.3,

(i) the LU decomposition takes about $\frac{1}{3}m^3$ multiplications and divisions, and

(ii) each of the p forward and backwards substitutions takes m^2 multiplications and $m(m-1)$ additions.

Thus the total number of multiplications and divisions for this LU method is about

$$\frac{1}{3}m^3 + pm^2 \quad .$$

Method 2 (Invert method)

(i) Calculation of A^{-1}. One of the best ways to do this is

to first use Gaussian elimination to make A upper triangular (about $\frac{1}{3}\,m^3$ multiplications and divisions) and then, for each k=1,2,...,m, find the k^{th} column Y_k of A^{-1} by solving $AY_k = e_k$ where e_k is the $m \times 1$ column vector with 1 in the k^{th} row and zeros elsewhere. (This is in effect solving the matrix equation AY = I where I is the $m \times m$ indentity matrix.) Solution of each problem $AY_k = e_k$ requires m^2 multiplications (see section 2.3) and so this method of calculating A^{-1} uses about

$$\frac{1}{3}\,m^3 + m.m^2 = \frac{4}{3}\,m^3$$

multiplications and divisions. (Clearly this is vastly better than the $2m^2m!$ operations using (2.4) in section 2.1).

(ii) Calculation of $-A^{-1}B_j$. It is easy to see that, for each j=1,2,...,p this involves m^2 multiplications and m(m - 1) additions, which is exactly that required for the forward and backward substitution in (ii) of Method 1.

Thus the total number of multiplications and divisions for Method 2 is about

$$\frac{4}{3}\,m^3 + pm^2 \quad .$$

Comparison Clearly an LU method is superior to the Invert method since the latter requires about m^3 extra multiplications and divisions (which is just the number required to calculate A^{-1} explicitly).

This comparison is seen most starkly when p is very small compared to m. Then we can ignore pm^2 and so

LU uses about $\frac{1}{3} m^3$,

while

Invert uses about $\frac{4}{3} m^3$.

In this case we would expect Invert to take about 4 times as long as LU. This is borne out by our tests in section 4.

The moral of section 2 is:

> Use an LU method when solving linear equations.

3. WHEN A IS SPARSE

Matrices with many zero entries are said to be sparse. When A is sparse the best method for solving

$$AX = B \qquad (3.1)$$

is still one of the LU methods (Doolittle or Crout) discussed in section 2. But now it is possible to reorganize the code to make two kinds of savings, firstly in storage requirements and secondly in CPU time, by eliminating unnecessary operations such as multiplying 0 by 0.

3.1 Storage savings

When an $m \times m$ matrix A is stored in the usual way it takes an amount of memory equal to that occupied by m^2 real numbers in the computer. This is called storing A in full mode.

When A is sparse, however, there is no point in using up space with the many entries equal to zero. It is only necessary to store the nonzeros. The simplest way of storing A in sparse mode is to store the nonzero entries in a one dimensional array containing

$$A(1),\ A(2),\ldots,A(NZ) \qquad (3.2)$$

and, because we no longer have the double array A(I,J) and so don't know yet in which row and column each entry occurs, we need also to store two integer arrays

$$IRN(1),\ IRN(2),\ldots,IRN(NZ)$$

$$ICN(1),\ ICN(2),\ldots,ICN(NZ)\ ,$$

where entry A(I) is in row IRN(I) and column ICN(I). This requires only storage for 3NZ numbers (where NZ is the total number of nonzeros in A). If, for example, A is 100 x 100 and has 90 per cent of its entries zero then NZ = 1000. Thus sparse mode requires only 3000 places of store while full mode requires 100^2 = 10000 places.

In practice, in code for sparse matrices, there are other storage overheads besides the arrays IRN and ICN above. But they are generally of small size compared to 3NZ.

3.2 Savings in operations and CPU time

Consider Gaussian elimination applied to a 10 × 10 sparse matrix A whose first two rows are

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 2 \\ 2 & 0 & 0 & -1 & 6 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} .$$

The first operation is to eliminate x_1 from the second equation by replacing R_2 (i.e., row 2) by $R_2 - 2R_1$. this requires 10 calculations (one for each column), 6 of which are the calculation of

$$0 - 2(0) ,$$

which is rather wasteful.

In sparse mode, there would only be 3 entries in row 1, namely in columns 1, 5 and 10. The code would be organized so that the only changes to row 2 would be made in these columns. Thus only 3 calculations would be done.

This example shows one other feature of sparse code. In A the entry in row 2, column 10 is zero and so it is not held in the array which contains

$$A(1),\ldots,A(NZ) \quad .$$

But after the elimination of x_1 from the second equation, this entry becomes nonzero (equal to -4). This is handled in sparse code by increasing NZ by 1, and then setting

$$A(NZ+1) = -4, \quad IRN(NZ+1) = 2, \quad ICN(NZ+1) = 10 \quad .$$

Accordingly sparse code must allow for the number NZ of nonzeros to increase (or, perhaps, decrease). The storage space allocated for the arrays A, IRN and ICN must be big enough initially to provide room for the greatest number of nonzeros occurring at any stage in the decomposition of A.

3.3 Examples of savings with sparse code

We quote below two examples from Table 5.1 of Duff (1981) which show the differences between sparse and full code in storage, number of multiplications, decomposition time and time for computing the solution once the LU decomposition is known:

Size of A Number of nonzeros (initially)			199 × 199 701	363 × 363 3279
Storage	:	Full Sparse	39601 6485	131769 13353
Multiplications	:	Full Sparse	2626800 3455	15943928 4769
Decomposition time	:	Full Sparse	5033 250	29580 613
Solution Time	:	Full Sparse	77 10	457 23

Clearly very big savings can be made when the matrix A is as sparse as those in the examples above.

3.4 Keeping the number of nonzeros to a minimum

The storage and CPU savings for sparse matrices are only preserved if the matrix remains sparse as it is decomposed.

Just because a matrix A is sparse doesn't mean that other matrices associated with it (e.g. its inverse) will be sparse. For example[10]

$$A = \begin{pmatrix} 2 & 5 & 0 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 0 & 3 & 7 \\ 4 & 1 & 2 & 3 \end{pmatrix}$$

is fairly sparse but its inverse

$$A^{-1} = \begin{pmatrix} -0.238 & 0.205 & -0.246 & 0.369 \\ 0.295 & -0.082 & 0.098 & -0.148 \\ 0.918 & -1.033 & 0.639 & -0.459 \\ -0.393 & 0.443 & -0.131 & 0.197 \end{pmatrix}$$

is full.

Sparse codes must be careful to ensure that as much sparsity as possible is retained during decomposition. Each step of the decomposition involves looking at one equation, say equation i, and using it to express one variable, say variable x_j, in terms of the others and then making this substitution in the other equations. This is called pivoting at position (i,j). (In section 2.2, tableau 2 is obtained from tableau 1 by pivoting at position (1,1).)

In practice the pivots need not be taken in a fixed order. Sparse code usually chooses pivot order so as to keep the number of new nonzero entries as small as possible subject to obtaining a solution of the desired accuracy.

The example below shows how varying the order of pivots can change the number of nonzeros. Consider

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 \end{pmatrix} .$$

If we pivot at position (1,1) we need to replace R_2 (i.e., row 2) by $R_2 - 2R_1$ (i.e., multiply each entry in row 1 by 2 and subtract element-wise from row 2), R_3 by $R_3 - 3R_1$, and R_4 by $R_4 - 4R_1$. This leads to

$$\left(\begin{array}{c|ccc} 1 & 2 & 3 & 4 \\ \hline 0 & -3 & -6 & -8 \\ 0 & -6 & -8 & -12 \\ 0 & -8 & -12 & -15 \end{array}\right)$$

Since the multipliers 2,3,4 would be stored where the zeros are in column 1, this leads from a sparse A to a full matrix in one hit.

But look at what happens if we pivot in a different order. First pivot at (2,2), giving

$$\begin{pmatrix} -3 & 0 & 3 & 4 \\ 2 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 \end{pmatrix} \qquad R_1 - 2R_2$$

and then pivot at (3,3) and (4,4) giving

$$\begin{pmatrix} -12 & 0 & 0 & 4 \\ 2 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 \end{pmatrix} \qquad R_1 - 3R_3$$

$$\begin{pmatrix} -28 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 \end{pmatrix} \qquad R_1 - 4R_4$$

With this pivoting order, A remains sparse to the end.

Choosing pivots in various orders leads to an LU decomposition not of the original A but of the matrix obtained from A by permuting the order of rows and columns accordingly. In the example above with the second pivoting order, row 1 and column 1 would be permuted to row 4 and column 4 so that we end up with an LU decomposition of A permuted to

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \\ 2 & 3 & 4 & 1 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 3 & 4 & 1 \end{bmatrix}}_{L_1} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & -28 \end{bmatrix}}_{U_1} .$$

A rough estimate of the number of nonzeros introduced by pivoting at (i,j) is the product of r_i and s_j, where r_i is the number of nonzeros in row i and s_j is the number of nonzeros in column j. In many sparse codes (including the Harwell code used in section 4), of all the possible pivots, the one chosen is one for which this product (called the Markowitz count) is smallest (subject to a numerical stability criterion). We refer the interested reader to Duff (1977) for details.

4. COMPARISON OF THREE METHODS FOR SOLVING AX = B

4.1 Preliminaries

We have carried out tests on three methods for solving AX = B, where A is a sparse m × m matrix, B is m × p and X, the solution matrix, is also m × p. The methods used are described in sections 2 and 3 of this paper, so here we give them abbreviated names which will be used in the rest of the paper, and describe briefly the code used in each case:

(a) Sparse This consists of applying subroutine MA28[11] from the Harwell Laboratories library. MA28 was written specifically for the purpose of solving sparse linear systems. It forms the Doolittle decomposition of A;

(b) Crout A code based on the routine UNSYMSOL[12] was used in this phase to produce the Crout LU decomposition of A. This routine stores A in full form;

and

(c) Invert Here the matrix A was inverted (using Gaussian elimination as described in section 2.5) with the code currently employed to solve the ORANI model. This code stores A in full form.

Each method involves two stages:

(1) A is transformed into another matrix. In the case of the Sparse and Crout methods this is an LU decomposition of A; for Invert, A^{-1} is computed. In tables 4.1 and 4.2 this stage is labelled "Transform A";

and

(2) Using this transformation of A, the system AX = B is solved. For the Sparse and Crout methods this is done by forward and backward substitution; and for Invert it is done by calculating $A^{-1}B$.

In constructing Tables 4.1 and 4.2 we were interested in matrices which were sparse. To distinguish between very sparse matrices (that is, those having only a few nonzero entries) and less sparse matrices (i.e. matrices with more nonzeros) we use the term density.

The density of a matrix is the ratio of the number of nonzero entries to the total number of entries of the matrix.

Thus, if A is m × m and has T nonzeros then the density of A is

$$\frac{T}{m^2} .$$

For example:

$$\begin{pmatrix} 0 & 2 \\ 3 & 0 \end{pmatrix}$$ is 50% dense ;

and $\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 4 \\ 5 & 0 & 0 & 0 & 0 \end{bmatrix}$ is 20% dense.

The results reported in Tables 4.1 and 4.2 for the Sparse, Crout and Invert algorithms are CPU times measured in seconds on a VAX 11/780 which was running under version 3.2 of the VMS operating system. This is a paged virtual memory machine, and when the results for Tables 4.1 and 4.2 were produced, the system allocated us up to 120 pages of physical memory.

4.2 Results for randomly generated matrices

A random number generator was used to obtain floating point numbers whose signs and magnitudes were random, but were within bounds imposed by us, and to locate them randomly in square matrices A of either 10% or 50% density. The latter density was chosen because this is about the density of the matrix A given by ORANI; the former was chosen because this is approximately the density of the matrix A obtained from the ORANI condensed system. In Table 4.1 we give computer execution times associated with the solution of (2.1) for these random A matrices using each of the three methods. Separate execution times are given for use of the method Sparse with matrices having distinct densities, as Sparse decomposes a very sparse A much more quickly than a less sparse A. For the Crout and Invert methods, we would not expect the times to be different for

TABLE 4.1 : Comparison of execution times required by three algorithms used to solve $AX = B$[1,2], where A is $m \times m$, the elements of A are randomly generated, and the density of A is either 10% or 50%. Each algorithm consists of a transformation of A followed by a solution phase.

Order of the randomly generated matrix A	Percentage Density of Nonzeros in A	Sparse[3]		Crout[4,5]		Invert[4,6]	
		Transform A	Solve AX=B	Transform A	Solve AX=B	Transform A	Solve AX=B
100	10	3.2	1.0	4.5	3.1	25.3	3.7
	50	12.2	2.2				
150	10	13.7	2.6	15.8	7.4	94.2	8.8
	50	41.0	4.6				
250	10	93.5	8.7	76.1	21.5	452	25.1
	50	291	16.4				

1. In the results reported here B has 23 columns. The transform time is independent of the number of columns in B, but the solve time is proportional to the number of columns.

2. All times are CPU times measured in seconds on a VAX 11/780, running under version 3.2 of the VMS operating system with a working set of 120 pages.

3. The sparse matrix package used was the Harwell Laboratories subroutine library MA28 (see Duff (1977)).

4. For the Crout and Invert packages differences in the density of nonzeros produce only insignificant differences in CPU times. All times are the average for a number of trials with matrices having both densities.

5. The code for the Crout decomposition was derived from a routine written by Bowdler, Martin, Peters and Wilkinson (1971).

6. *The matrix inversion routine from the ORANI* programme library was used to generate the results in this column.

different densities (since they use full mode). This was confirmed by our tests and so for each size of matrix we give only one figure for these techniques in Table 4.1.

The conclusions that may be drawn from Table 4.1 are:

(i) Invert is by far the worst of the three algorithms, especially for the "Transform A": stage. This is consistent with the discussion of inversion in section 2.5.

(ii) Sparse performs much better at 10% density than at 50% density.

(iii) For randomly generated matrices having 10% density, Sparse is comparable with Crout. Indeed, in this case, Sparse is slightly worse than Crout in the transform phase, but is substantially quicker in the solve phase. The latter saving would be more significant if the number of columns in B were greater because the solve time is proportional to the number of columns of B. When solving ORANI, B matrices with more than 100 columns are often encountered.

(iv) For randomly generated matrices with 50% density, Crout performs significantly better in the transform stage than does Sparse.

4.3 Results for computable general equilibrium models

Table 4.2 gives the execution times required to solve three CGE models of the Johansen class via the Sparse, Crout and Invert methods. The models solved by these methods were:

(1) The ORANI model of the Australian economy. ORANI is a disaggregated model which distinguishes 230 commodities (115 domestically produced and 115 imported), 113 industries, 9 types of labour, 113 types of capital, 7 types of agricultural land, and includes detailed modelling of margins industries. ORANI allows for multi-product industries and multi-industry products. Substitution is allowed between domestic and imported goods in response to relative price changes, between factors of production in response to changes in relative factor rewards, and between the products of the industries within the multi-product group in response to relative price changes. The prime reference for this model is DPSV (1982). Here we consider the final system for ORANI with a choice of exogenous variables close to that given in Table 23.3 of DPSV.[13]

(2) The miniature version (MO) of ORANI. MO is a two sector model designed to clarify the main ideas of ORANI. To quote its inventors[14]: "It recognises only one type of labour ... [and it] fails to model margins ... It uses a fictional data base and overly restrictive specifications of various substitution possibilities. Nevertheless [its inventors] feel that it is a useful model of a model". We solved this model for the economic environment given in DPSV, Section 6, pp. 32-37.

(3) The stylized Johansen model. This is a small pedogogical device consisting of seventeen equations. (See Dixon (1978)).

TABLE 4.2 : Comparison of execution times (in seconds) required by three algorithms used to solve three computable general equilibrium models. Each algorithm consists of a transformation of A in AX=B[1], followed by a solution phase.

Model	Order of A	Percentage Density of A	Number of columns of B	Sparse[2]		Crout[3,4]		Invert[3,5]	
				Transform A	Solve AX=B	Transform A	Solve AX=B	Transform A	Solve AX=B
Stylized Johansen	17	26	2	0.08	0.01[6]	0.04	0.02[6]	0.12	0.01[6]
MO	39	10	13	0.12	0.09	0.31	0.25	1.35	0.27
ORANI	273	51	23	88.9	9.0	102	25.4	591	30.1

1. All times are CPU times measured in seconds on a VAX 11/780, running under version 3.2 of the VMS operating system with a working set of 120 pages.

2. The sparse matrix package used was the Harwell Laboratories subroutine library MA28 (see Duff (1977)).

3. For the Crout and Invert packages differences in the density of nonzeros produces only insignificant differences in CPU times. All times are the averages for a number of trials with matrices having both densities.

4. The code for the Crout decomposition was derived from a routine written by Bowdler, Martin, Peters and Wilkinson (1971)).

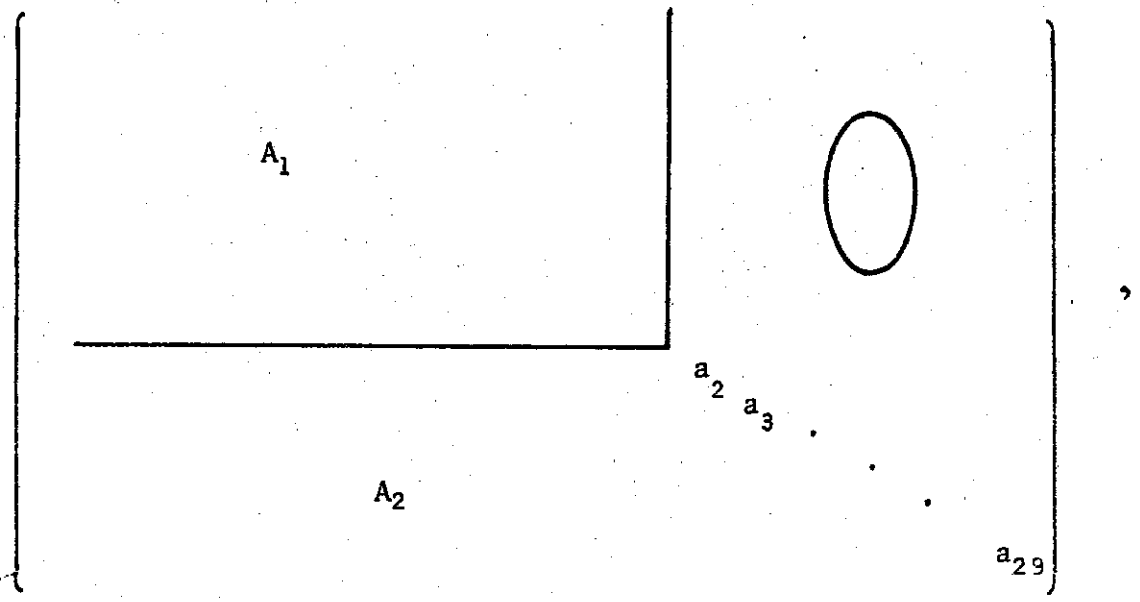
5. The matrix inversion routine from the ORANI programme library was used to generate the results in this column.

6. These solution times are so small as to be unreliable.

The conclusions which can be drawn from Table 4.2 are:

(i) The figures for the stylized Johansen and MO models are consistent with the conclusions reached in section 4.2. Notice that MO is less dense than the stylized model, and accordingly, Sparse compares more favourably with Crout for MO than for the stylized Johansen system.

(ii) The surprise (after the figures given in Table 4.1 for a 250×250 matrix A) is that, even though ORANI is over 50% dense, Sparse is a little quicker than Crout for this model. Obviously the randomly generated matrices of section 4.1 are not good predictors of the execution times which can be obtained with Sparse and Crout for the system AX = B generated by ORANI. The feature of the Harwell routine MA28 responsible for this is a preliminary step in which the rows and columns of A are re-arranged to produce the matrix:



where A_1 is 245×245, $a_2, a_3, \ldots, a_{29}$ are scalars and the top right hand corner is a zero matrix.

By operating on this matrix Sparse is able to produce a Doolittle decomposition more quickly than Crout, as the Harwell code only needs to decompose the partition A_1.

(iii) As for the randomly generated matrices Crout takes nearly three times as long for the solve stage. With, say 300 exogenous variables (which is a typical number for ORANI), the solve time would be about 120 seconds for Sparse and around 330 seconds for Crout. In this case Sparse significantly outperforms Crout.

4.4 Comparison of VAX 11/780 times with those given in Table 34.1 of DPSV

Table 34.1 of DPSV gives execution times for each step in the solution procedure for ORANI implemented on the CSIRONET Cyber 76 machine. As essentially the same code was used to invert ORANI on this Cyber 76 as on the VAX 11/780, we may compare the execution time given for this step in Table 34.1 with the corresponding transformation of A reported in Table 4.2. On the Cyber 76 formation of the inverse of A requires 35 seconds. This is about 17 (i.e., 591/35) times faster than for the VAX. This difference may presumably be attributed to differences in the two environments.

We would expect to obtain ratios of around the same magnitude for the CPU times involved in execution of the Sparse and Crout methods. Based on the times in row 3 of Table 4.2, the Sparse method would require

about 5 (i.e., 88.9/17) seconds to transform the matrix of the ORANI final system on the Cyber 76, while the Crout technique would take approximately 6 (i.e., 102/17) seconds. This is about one sixth of the time taken to transform A on the Cyber 76 using Invert.

5. PLANS FOR FURTHER EXPLORATION

The conclusions reached above must be regarded as preliminary in nature, and will possibly need to be modified after further work. In what follows, we briefly touch on several topics which, as yet, we have investigated only a little, or not at all. These seem to us to be relevant to the construction of efficient and practical procedures for the solution of Johansen type models. We propose four areas for future research.

(1) We need to make other comparisons between Sparse and Crout for systems arising from CGE models of the Johnasen type. We will do tests on other 273 × 273 matrices obtained from ORANI for different economic environments.

(2) The final ORANI system is obtained by elimination of variables from the condensed system (see DPSV, section 34) for which the A matrix is over 2000 × 2000. We expect this matrix to be less dense than the 273 × 273 version and that, accordingly, Sparse should be more efficient. For the reasons indicated earlier in this paper, we regard this as the most important area for future work.

(3) The relative accuracy and stability of Sparse and Crout are obviously important. In Sparse, one chooses a parameter u with $0 < u < 1$.[15] The closer is u to 1, the more accurate the solution becomes but the decomposition time increases, since the number of new nonzero entries is likely to increase. Our results in section 3 were computed with u = 0.1 (recommended by Duff). With u = 0.1 we have found Sparse gives

less accurate solutions than Crout. We will experiment with increasing u to see how much this improves accuracy (a good thing), and how much it increases CPU time (a bad thing).

(4) A feature of ORANI is the flexibility that users have in being able to select different sets of exogenous variables. At present the model is completely re-solved for each new exogenous selection. Certainly when only a small number of variables are interchanged on the lists of endogenous and exogenous variables, it seems likely that, instead of re-solving the system, the new solution can be obtained directly from the old solution (by using a simple linear transformation).[16] We plan to investigate the efficiency and stability of this variable swapping technique, and hope to incorporate it as a further user option in ORANI.

REFERENCES

Bowdler, H.J., R.S. Martin, G. Peters and J.H. Wilkinson (1971) "Solution of Real and Complex Systems of Linear Equations", pp. 93-110, in Linear Algebra, (Handbook of Automatic Computation, Vol. 2), J.H. Wilkinson and C. Reinsch (eds), Springer-Verlag, Berlin.

Conte, S.D. and C. de Boor (1972), Elementary Numerical Analysis - An Algorithmic Approach, McGraw-Hill Kogakusha Ltd, Tokyo.

Daniels, R.W. (1978), An Introduction to Numerical Methods and Optimization Techniques, North Holland, New York.

Dixon, P.B., B.R. Parmenter, J. Sutton and D.P. Vincent (1982) ORANI: A Multisectoral Model of the Australian Economy, Contributions to Economic Analysis, Vol. 142, North Holland Publishing Co, Amsterdam.

Dixon, P.B., (1978) "A Stylized Johansen Model", IMPACT Research Memorandum, mimeo., 16 pp. (Available from IMPACT Research Centre, University of Melbourne.)

Duff, I.S. (1977), "MA28 - A set of Fortran Subroutines for Sparse Unsymmetric Linear Equations", Harwell Report R. 8730, HMSO, London.

Duff, I.S. (1981) "A Sparse Future", in Sparse Matrices and Their Uses, Iain S. Duff (ed.), Academic Press, London.

Stewart, G.W. (1973), Introduction to Matrix Computations, Academic Press, London.

NOTES

* The authors are indebted to Brian Parmenter and Peter Dixon for their careful appraisals of an earlier version of this paper.

1. The ORANI model is described in Dixon, Parmenter, Sutton and Vincent (1982) (hereafter, referred to as DPSV).

2. The number of exogenous variables in a partition of the final ORANI system is much larger than 300 (see DPSV, section 44). However, in the typical ORANI computation many of these are set to zero. Only the columns of D corresponding to the non-zero exogenous variables need to be included explicitly in B in the computations.

3. See Duff (1977).

4. See Dixon (1978).

5. See DPSV, Chapter 2.

6. The actual number of equations can be reduced below 300. See footnote 22, DPSV, p. 228.

7. The reduction to the final system is described in DPSV, sections 31 and 32.

8. See DPSV, Tables 32.1 and 32.2, pp. 211-221.

9. See Stewart (1973), p. 136.

10. Daniels (1978), pp. 35-36.

11. Duff (1977) describes the operation of MA28.

12. UNSYMSOL was written by Bowdler, Martin, Peters and Wilkinson (1971).

13. The differences are that c_R and i_R were not exogenous in our tests, but ΔB and f_R were. See DPSV, Tables 23.2, 23.3 pp. 136-141 and pp. 143-144

14. See DPSV, p. 9.

15. Duff (1981), Section 2.

16. DPSV, Section 36.