Chapter 5

Linear Solvers for Dense Systems

This chapter describes the CM Fortran interface to the CMSSL general linear system solver routines. One section is devoted to each of the following topics:

- introduction
- Gaussian elimination (*LU* decomposition)
- solving linear systems using Householder transformations (*QR* decomposition)
- matrix inversion and the Gauss-Jordan solver
- Gaussian elimination with external storage
- *QR* factorization and least squares solution with external storage
- references

5.1 Introduction

Listed below are the CMSSL routines for solving dense linear systems. All routines accept either real or complex data.

Copyright © 1994 Thinking Machines Corporation

- In-core solvers:
  - Routines that use Gaussian elimination (with or without pivoting) to decompose one or more matrices A into their LU factors; use those factors to solve the linear systems $AX = B$ or $A^{T}X = B$ (where B consists of one or more right-hand-side vectors); and perform related operations:

    gen_lu_factor
    save_gen_lu
    save_gen_lu_fms
    restore_gen_lu
    restore_gen_lu_fms
    gen_lu_solve
    gen_lu_solve_tra
    gen_lu_apply_l_inv
    gen_lu_apply_u_inv
    gen_lu_apply_l_inv_tra
    gen_lu_apply_u_inv_tra
    gen_lu_apply_l
    gen_lu_apply_l_tra
    gen_lu_get_l
    gen_lu_get_u
    gen_lu_zero_rows
    gen_lu_infinity_norm_inv
    deallocate_gen_lu

  - Routines that use Householder transformations (with or without column pivoting) to decompose one or more matrices A into their QR factors; use those factors to solve the linear systems $AX = B$ or $A^{T}X = B$ (where B consists of one or more right-hand-side vectors); and perform related operations:

    gen_qr_factor
    save_gen_qr
    save_gen_qr_fms
    restore_gen_qr
    restore_gen_qr_fms
    gen_qr_solve
    gen_qr_solve_tra
    gen_qr_apply_q
    gen_qr_apply_q_tra
    gen_qr_apply_r_inv
    gen_qr_apply_r_inv_tra
    gen_qr_get_r
    gen_qr_apply_p
    gen_qr_apply_p_inv
    gen_qr_zero_rows
    gen_qr_extract_diag
    gen_qr_deposit_diag
    gen_qr_infinity_norm_inv
    gen_qr_r_infinity_norm_inv
    deallocate_gen_qr

  - A routine, **gen_gj_invert**, that inverts a square matrix in place, using the Gauss-Jordan routine.

  - A routine, **gen_gj_solve**, that solves (with partial or total pivoting) a system of equations of the form $AX = B$ using a version of Gauss-Jordan elimination. B contains one or more right-hand-side vectors.

  The save and restore routines do not work with the nodal CMSSL library.

Copyright © 1994 Thinking Machines Corporation

- External solvers:
  - Routines that use block Gaussian elimination with full column pivoting to solve the linear system of equations $AX=B$ where A is a real or complex square matrix that is too large to fit into core memory:

    gen_lu_setup_ext
    gen_lu_factor_ext
    gen_lu_solve_ext

  - Routines that use block Householder reflections to perform the factorization $A = QR$, where A is a matrix that is too large to fit into core memory, and use the QR factors to solve the linear system $AX = B$:

    gen_qr_factor_ext
    gen_qr_solve_ext

The external solvers do not work with the nodal CMSSL library.

5.1.1 Embedding Coefficient Matrices within Larger Matrices

All of the in-core CMSSL general linear system solver routines allow you to embed the systems to be solved within larger matrices that have more rows and columns than the number of equations to be solved or the number of unknowns, respectively. You must specify, in the calling sequences, the axis lengths of the systems to be solved. For example, if the systems to be solved have dimensions m x n, with rows and columns counted by axes *row_axis* and *col_axis* respectively, you may declare axes *row_axis* and *col_axis* to have extents greater than m and n, respectively; but the routine will work only with the upper left-hand m x n elements of the matrix defined by *row_axis* and *col_axis*. The man pages for the individual solvers provide detailed information about axis extents.

5.1.2 Choosing an Algorithm

Use these guidelines when deciding whether to use the *QR* or *LU* routines, and whether to use pivoting:

Copyright © 1994 Thinking Machines Corporation

- In most cases, the *QR* routines without pivoting or the *LU* routines with partial pivoting suffice. These two options are both stable for almost all practical purposes.
- For rank-deficient matrices, use the *QR* routines with pivoting. (Section 5.3.8 discusses working with ill-conditioned matrices.) It is unnecessary and wasteful of time to use the *QR* routines with pivoting for well-conditioned matrices.
- For matrices that are diagonally dominant, or where *LU* decomposition without pivoting is known to work, use the *LU* routines without pivoting. This option will not yield correct results if the matrix is ill-conditioned or if zeros appear on the diagonal during the elimination process.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

5.2 Gaussian Elimination ("*LU*" Routines)

This section describes the CMSSL Gaussian elimination routines routines (commonly referred to as the "*LU*" routines). The following topics are included:

- the *LU* routines
- blocking and load balancing
- the *LU* factors defined
- results of the solver routines
- numerical stability
- saving and restoring the *LU* state

For detailed descriptions of the *LU* routines (including calling sequences, argument definitions, and information about usage), refer to the man page at the end of this section.

Throughout this section, the following conventions are used:

- $M^{-\mathrm{T}}$ denotes $(M^{-1})^{\mathrm{T}} = (M^{\mathrm{T}})^{-1}$.
- "*A*" refers to a matrix being factored and "*B*" refers to the right-hand side(s). One or more instances of A and B are embedded (possibly within larger matrices) in the CM arrays ***A*** and ***B***, respectively; the operations described are performed on all instances concurrently. The man page provides details about ***A*** and ***B***.

5.2.1 The *LU* Routines

Given a CM array ***A*** containing one or more instances of a dense m x n matrix A (with $m \geq n$), and a CM array ***B*** containing corresponding right-hand sides B, the *LU* routines perform the operations listed below for each instance.

Factorization routine:

gen_lu_factor — Uses Gaussian elimination, with or without partial pivoting, to perform the factorization $A = LU$. When you specify pivoting, the effects of pivoting are included in L.

Copyright © 1994 Thinking Machines Corporation

In the non-square case ($m > n$), the factorization is defined so that L is $m \times m$ and U is the $m \times n$ matrix $[U]_{mm}[I]_{mn}$, where $[U]_{mm}$ is an $m \times m$ matrix defined in Section 5.2.3 and $[I]_{mn}$ is the $m \times n$ matrix consisting of the $n \times n$ identity matrix extended by $m-n$ rows of zeroes. However, no $m \times m$ matrices are actually stored; L and U are stored together in an $m \times n$ matrix.

Save and restore routines:

save_gen_lu — Saves internal information about the LU factors in a file using serial-order I/O.

save_gen_lu_fms — Saves internal information about the LU factors in a file using fixed-machine-size I/O.

restore_gen_lu — Restores internal information about the LU factors from a file using serial-order I/O.

restore_gen_lu_fms — Restores internal information about the LU factors from a file using fixed-machine-size I/O.

Solver routines:

gen_lu_solve — Uses the LU factors returned by **gen_lu_factor** to solve $AX = B$. In the non-square case ($m > n$), the solver routine solves $L[U]_{mm}X = B$.

gen_lu_solve_tra — In the square case, **gen_lu_solve_tra** uses the LU factors returned by **gen_lu_factor** to solve $A^{\mathrm{T}}X = B$. In the non-square case ($m > n$), conceptually, **gen_lu_solve_tra** produces a solution to $A^{\mathrm{T}}X = B_n$; B_n is the first n block cyclic rows of B. To perform this operation, **gen_lu_solve_tra** solves the system $(L[U]_{mm})^{\mathrm{T}}X = B_0$, where B_0 is B with the last $m - n$ block cyclic rows zeroed.

Factor application routines: These routines use the factors produced by the LU factorization routine to solve "triangular" systems of the form $UX=B$, $U^{\mathrm{T}}X=B$, $LX=B$, $L^{\mathrm{T}}X=B$, $L^{-1}X=B$, $L^{-\mathrm{T}}X=B$. (Because of implementation

Copyright © 1994 Thinking Machines Corporation

details, L and U are not actually triangular; see Section 5.2.3.) In the non-square case, replace U with $[U]_{mm}$.

gen_lu_apply_l_inv Given the LU factors returned by **gen_lu_factor**, applies L^{-1} to B.

gen_lu_apply_u_inv Given the LU factors returned by **gen_lu_factor**, applies U^{-1} to B. In the non-square case, this routine applies $[U]_{mm}{}^{-1}$ to B.

gen_lu_apply_l_inv_tra Given the LU factors returned by **gen_lu_factor**, applies $L^{-\mathrm{T}}$ to B.

gen_lu_apply_u_inv_tra Given the LU factors returned by **gen_lu_factor**, applies $U^{-\mathrm{T}}$ to B. In the non-square case, this routine applies $[U]_{mm}{}^{-\mathrm{T}}$ to B.

gen_lu_apply_l Given the LU factors returned by **gen_lu_factor**, applies L to B.

gen_lu_apply_l_tra Given the LU factors returned by **gen_lu_factor**, applies L^{T} to B.

Get-factor routines:

gen_lu_get_l Given the LU factors returned by **gen_lu_factor**, produces L separately. (Available only in the square case.)

gen_lu_get_u Given the LU factors returned by **gen_lu_factor**, produces U separately.

Zeroing routine:

gen_lu_zero_rows Zeroes the final rows of B. The rows are counted in block cyclic order.

Infinity norm routine:

gen_lu_infinity_norm_inv Given the LU factors returned by **gen_lu_factor**, estimates the infinity norm of A^{-1}. Uses the method developed by Hager; see reference 5 listed in Section 5.7. (Available only in the square case.)

Copyright © 1994 Thinking Machines Corporation

Deallocation routine:

deallocate_gen_lu	Deallocates the memory required by the above routines.

5.2.2 Blocking and Load Balancing

The *LU* routines use blocking and load balancing (block cyclic ordering). For information about block cyclic ordering and about the blocking factor (which is an input argument), see the section on computation of block cyclic permutations in Chapter 15. For details about how the *LU* routines implement blocking, see reference 4 listed in Section 5.7. The man page at the end of this section provides guidelines for choosing a blocking factor. The definitions of the *LU* factors presented in Section 5.2.3 include the effects of blocking and load balancing.

5.2.3 The *LU* Factors Defined

One common representation for the *LU* factorization is

$$PA = LU$$

where P is a permutation matrix resulting from the pivoting process, and L and U are lower triangular and upper triangular, respectively. However, because of details of the implementation of the CMSSL *LU* routines, we represent the factorization as $A = LU$, where the effects of pivoting are included in L, as indicated in the definitions below.

In the non-square case ($m > n$), the factorization is defined so that L is m x m and U is the m x n matrix defined below. However, no m x m matrices are actually stored; L and U are stored together in an m x n matrix.

The **gen_lu_factor** routine actually factors a block cyclic permutation, A_c, of each matrix A that you supply in $\boldsymbol{A}$. The matrix A_c is defined by

$$A_c = {P_1}^{-1} A P_2$$

where P_1 and P_2 are the permutations giving the correspondence between standard and block cyclic row order and column order, respectively, for A. (These matrices depend on the array size and layout, the partition size, and the blocking factor you supply.) The matrix A_c is factored into

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

$A_c = {P_c}^{-1} L_c U_c$ (factorization with pivoting)
$A_c = L_c U_c$ (factorization without pivoting).

L_c is lower triangular and U_c is upper triangular. In the non-square case, L_c is m x m; U_c is m x n and is zero except for (i, j) where $i \leq j \leq n$. P_c is the permutation matrix resulting from the pivoting process. The LU factors of A are defined in terms of A_c and its factors as described below.

- *Case 1: Factorization with pivoting; $m \geq n$*

  The definition

  $$A_c = {P_1}^{-1} A P_2$$

  implies that

  $$\begin{aligned} A = LU = & \ P_1 A_c {P_2}^{-1} \\ & = P_1 \left({P_c}^{-1} L_c U_c\right) {P_2}^{-1} \\ & = P_1 {P_c}^{-1} L_c \left({P_1}^{-1} P_1\right) U_c {P_2}^{-1} \end{aligned}$$

  from which we choose to define

  $$\begin{aligned} L &= P_1 {P_c}^{-1} L_c {P_1}^{-1} \\ U &= P_1 U_c {P_2}^{-1}. \end{aligned}$$

  Note that these definitions make sense in both the square case and the non-square case, since P_1 and P_c are m x m and P_2 is n x n.

- *Case 2: Factorization without pivoting; $m = n$*

  The no-pivoting case requires that no small pivots be encountered during the elimination process. Therefore, the factorization routine pre-permutes the matrix A to assure that the diagonal elements of A also appear on the diagonal of A_c.

  In the square case, A is pre-permuted internally to obtain

  $$A^* = A P_1 {P_2}^{-1}$$

  By definition, we have

  $$A_c = {P_1}^{-1} A^* P_2$$

  from which it follows that

  $$A_c = {P_1}^{-1} A P_1.$$

  Therefore,

  $$\begin{aligned} A = LU = & \ P_1 A_c {P_1}^{-1} \\ & = P_1 L_c U_c {P_1}^{-1} \\ & = P_1 L_c \left({P_1}^{-1} P_1\right) U_c {P_1}^{-1} \end{aligned}$$

Copyright © 1994 Thinking Machines Corporation

from which we choose to define

$$L = P_1 L_c {P_1}^{-1}$$
$$U = P_1 U_c {P_1}^{-1}.$$

- *Case 2: Factorization without pivoting; $m > n$*

  In the non-square case, A is pre-permuted internally to obtain

  $$A^* = P_1 [{P_2}^{-1}]_{mm} A$$

  where $[{P_2}^{-1}]_{mm}$ is the inverse of the $m \times m$ permutation matrix $[P_2]_{mm}$ formed by replacing the upper left $n \times n$ elements of the $m \times m$ identity matrix with P_2, as follows:

$$\begin{array}{c} \\ n \\ m-n \end{array} \begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \begin{bmatrix} \boxed{P_2} & \\ & \mathrm{I} \end{bmatrix} \end{array}$$

  By the definition of block cyclic ordering, we have

  $$A_c = {P_1}^{-1} A^* P_2$$

  from which it follows that

  $$A_c = [{P_2}^{-1}]_{mm} A P_2.$$

  Therefore,

  $$\begin{aligned} A = LU &= [P_2]_{mm} A_c {P_2}^{-1} \\ &= [P_2]_{mm} L_c U_c {P_2}^{-1} \\ &= [P_2]_{mm} L_c ({P_1}^{-1} P_1) U_c {P_2}^{-1} \end{aligned}$$

  from which we choose to define

  $$L = [P_2]_{mm} L_c {P_1}^{-1}$$
  $$U = P_1 U_c {P_2}^{-1}.$$

Summary of Factor Definitions

In summary, the LU factors are defined as follows:

$m \geq n$, pivoting	$L = P_1 {P_c}^{-1} L_c {P_1}^{-1}$ $U = P_1 U_c {P_2}^{-1}$
$m = n$, no pivoting	$L = P_1 L_c {P_1}^{-1}$ $U = P_1 U_c {P_1}^{-1}$

Copyright © 1994 Thinking Machines Corporation

$m > n$, no pivoting $\quad L = [P_2]_{mm} L_c P_1^{-1}$
$\qquad\qquad\qquad\qquad\quad U = P_1 U_c P_2^{-1}$

Factors Returned by the Get-Factor Routines

The **gen_lu_get_l** and **gen_lu_get_u** routines return the L and U factors defined above. In the non-square case, **gen_lu_get_l** is not available; to obtain the m x m matrix L, you must apply **gen_lu_apply_l** to the m x m identity matrix.

For readers who want a detailed explanation of why **gen_lu_get_l** is not available for the non-square case: in order to return the essential m x n portion of L in B, we would need to compute $L[I]_{mn}$, which is equal to $P_1 P_c^{-1} L_c P_1^{-1}[I]_{mn}$ (in the pivoting case) or $[P_2]_{mm} L_c P_1^{-1}[I]_{mn}$ (in the no-pivoting case). But P_1^{-1} would permute some of the columns with essential information out of range of the first n columns; the information in those columns would not appear in B, and would be lost.

Factors Applied by the Factor Application Routines

The matrices L^{-1}, L^{-T}, L, and L^T applied by the factor application routines are given by the definitions of L above. The inverses are true inverses, in the sense that $L^{-1}L = LL^{-1} = I$. The inverses are also true in the block cyclic space; that is, $L_c^{-1} L_c = L_c L_c^{-1} = I$.

In the square case, the **gen_lu_apply_u_inv** and **gen_lu_apply_u_inv_tra** routines apply U^{-1} and U^{-T} as derived from the definitions of U above; the inverses are true inverses in both the original space and the block cyclic space.

In the non-square case, we define $[U_c]_{mm}$ to be the m x m extension of U_c shown below.

$$\begin{array}{c} \\ m \end{array} \begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \left[\begin{array}{c|c} U_c & \begin{array}{c} 0 \\ \hline \mathrm{I} \end{array} \end{array}\right] \end{array} \begin{array}{c} \\ \begin{array}{c} n \\ m-n \end{array} \end{array}$$

Note that by the structure of this matrix, we have $U_c = [U_c]_{mm}[I]_{mn}$, where $[I]_{mn}$ is the m x n matrix consisting of the n x n identity matrix extended by $m-n$ rows of zeroes. We now define the m x m matrix

Copyright © 1994 Thinking Machines Corporation

$$[U]_{mm} = P_1[U_c]_{mm}[P_2{}^{-1}]_{mm}.$$

The **gen_lu_apply_u_inv** and **gen_lu_apply_u_inv_tra** routines apply $[U]_{mm}{}^{-1}$ and $[U]_{mm}{}^{-\mathrm{T}}$, respectively, to the arbitrary matrices you supply (although no m x m matrices are actually stored).

Alternate View of the Non-Square Factorization

Finally, we see that in the non-square case, with or without pivoting,

$$\begin{aligned} U &= P_1 U_c P_2{}^{-1} \\ &= P_1[U_c]_{mm}[I]_{mn}P_2{}^{-1} \\ &= P_1[U_c]_{mm}[P_2{}^{-1}]_{mm}[I]_{mn} \\ &= [U]_{mm}[I]_{mn}. \end{aligned}$$

Thus, in the non-square case, we can view the factorization of A as

$$A = L[U]_{mm}[I]_{mn}.$$

5.2.4 Results of the Solver Routines

In the square case ($m = n$),

- The **gen_lu_solve** routine solves the system $AX = B$. Upon successful completion, each B is overwritten with the solution.
- The **gen_lu_solve_tra** routine solves the system $A^{\mathrm{T}}X = B$. Upon successful completion, each B is overwritten with the solution.

In the non-square case ($m > n$),

- The **gen_lu_solve** routine solves the system $L[U]_{mm}X = B$. Upon successful completion, each B is overwritten with the solution to $L[U]_{mm}X = B$.
- Conceptually, the **gen_lu_solve_tra** routine produces a solution to the system $A^{\mathrm{T}}X = B_n$, where B_n is the first n block cyclic rows of B. To perform this operation, the transpose solver routine solves the system $(L[U]_{mm})^{\mathrm{T}}X = B_0$, where B_0 is B with the last $m - n$ block cyclic rows zeroed. Upon successful completion, each B is overwritten with the solution to $(L[U]_{mm})^{\mathrm{T}}X = B_0$.

Copyright © 1994 Thinking Machines Corporation

5.2.5 Numerical Stability

The stability of Gaussian elimination is a function of the size of the linear system and the growth factor (see references 1 and 3). For extreme cases, the growth factor may be very large. For most systems, it is highly unlikely that the growth factor for Gaussian elimination with partial pivoting will be very large, and for all practical purposes Gaussian elimination with partial pivoting is stable.

5.2.6 Saving and Restoring the *LU* State

The *LU* factorization routine generates internal state variables required for computing the solution. These variables are not made available as arrays to user applications because their sizes and contents are CM configuration-dependent. However, it is sometimes desirable to save the internal state to a file for future use. The **save_gen_lu**, **save_gen_lu_fms**, **restore_gen_lu**, and **restore_gen_lu_fms** routines allow you to save and restore the internal *LU* state.

The *LU* routines allow you to have more than one factorization "active" at a time; for example, the sequence of calls

```
setup_X = gen_lu_factor(X, ...)
setup_Y = gen_lu_factor(Y, ...)
call gen_lu_solve(B_X, X, setup_X, ...)
call gen_lu_solve(B_Y, Y, setup_Y, ...)
```

is valid. You may, however, want to use the save and restore routines to carry the internal state over between program runs.

It is not intended that the save and restore routines be used to conserve memory. The state variables are very small compared to the size of the typical matrix A.

Copyright © 1994 Thinking Machines Corporation

NOTE

The save and restore routines do not work with the nodal CMSSL library.

Copyright © 1994 Thinking Machines Corporation

Gaussian Elimination

Given a CM array ***A*** containing one or more instances of a dense m x n matrix A, with $m \geq n$, and a CM array ***B*** containing corresponding right-hand sides B, the **gen_lu_factor** routine uses Gaussian elimination (with or without partial pivoting) to perform the factorization $A = LU$. When you specify pivoting, the effects of pivoting are included in L. The other routines described below use the LU factors returned by **gen_lu_factor** to solve the linear systems $AX = B$ or $A^{\mathrm{T}}X = B$; apply matrices derived from L and U to arbitrary matrices; provide access to the L and U factors; and estimate the infinity norm of each A^{-1}.

SYNTAX

Factorization routine:

setup = **gen_lu_factor**	(***A***, *m, n, row_axis, col_axis, nblock, pivoting_strategy, ier*)

Save and restore routines:

save_gen_lu	(*setup, unit, iostat, ier*)
save_gen_lu_fms	(*setup, unit, iostat, ier*)
setup = **restore_gen_lu**	(***A***, *m, n, row_axis, col_axis, nblock, pivoting_strategy, unit, iostat, ier*)
setup = **restore_gen_lu_fms**	(***A***, *m, n, row_axis, col_axis, nblock, pivoting_strategy, unit, iostat, ier*)

Solver routines:

gen_lu_solve	(***B***, ***A***, *setup, nrhs, ier*)
gen_lu_solve_tra	(***B***, ***A***, *setup, nrhs, ier*)

Factor application routines:

gen_lu_apply_l_inv	(***B***, ***A***, *setup, nrhs, ier*)
gen_lu_apply_u_inv	(***B***, ***A***, *setup, nrhs, ier*)
gen_lu_apply_l_inv_tra	(***B***, ***A***, *setup, nrhs, ier*)

Copyright © 1994 Thinking Machines Corporation

gen_lu_apply_u_inv_tra	(***B***, ***A***, *setup, nrhs, ier*)
gen_lu_apply_l	(***B***, ***A***, *setup, nrhs, ier*)
gen_lu_apply_l_tra	(***B***, ***A***, *setup, nrhs, ier*)

Get-factor routines:

gen_lu_get_l	(***B***, ***A***, *setup, ier*)
gen_lu_get_u	(***B***, ***A***, *setup, ier*)

Zeroing routine:

gen_lu_zero_rows	(***B***, ***A***, *setup, limit, nrhs, ier*)

Infinity norm routine:

gen_lu_infinity_norm_inv	(***A_norm***, ***A***, *setup, ier*)

Deallocation routine:

deallocate_gen_lu	(*setup*)

ARGUMENTS

In the descriptions below, A and B refer to the active matrices with which the routines work. These matrices may be contained (as the upper left-hand submatrices) in larger matrices within ***A*** and ***B***, respectively. Details are provided below.

Also, throughout these descriptions, M^{-T} denotes $(M^{-1})^T = (M^T)^{-1}$.

setup — Scalar integer variable. Setup ID returned by **gen_lu_factor**, **restore_gen_lu**, and **restore_gen_lu_fms**. When you call any of the other *LU* routines, you must supply the value returned by the corresponding **gen_lu_factor**, **restore_gen_lu**, or **restore_gen_lu_fms** call.

B — CM array of the same type (real or complex) and precision as ***A***. Must be distinct from ***A***. The instance axes of ***B*** must match those of ***A*** in order of declaration and extents. When you call **gen_lu_get_l** or **gen_lu_get_u**, ***A*** and ***B*** must have the same rank, axis extents, and layout directives.

Copyright © 1994 Thinking Machines Corporation

Solver, Factor Application, and Zeroing Routines. When you call an *LU* solver, factor application, or zeroing routine, ***B*** must contain one or more instances of *B*, where each *B* consists of one or more right-hand-side vectors. The following restrictions hold:

- If each instance *B* within ***B*** consists of only one right-hand-side vector (*nrhs* = 1), you may represent *B* in either of the following ways:
    - It may have rank 2 with number of columns = 1. In this case, each *B* has dimensions *m* x 1 (and may consist of the upper left-hand *m* x 1 elements of a larger matrix). The rows of each *B* must be counted by axis *row_axis* (from the **gen_lu_factor** call); the single column must be counted by axis *col_axis*.
    - It may have rank 1. In this case, each *B* has dimension *m* (and may consist of the first *m* elements of a larger vector). The elements of each *B* must be counted by axis *row_axis* (if *row_axis* < *col_axis*) or by axis (*row_axis* − 1) (if *col_axis* < *row_axis*). For an example, see the Notes section.
- If each *B* within ***B*** consists of multiple right-hand-side vectors (*nrhs* > 1), then each *B* has dimensions *m* x *nrhs*, and may consist of the upper left-hand *m* x *nrhs* elements of a larger matrix. The rows and columns of *B* must be counted by axes *row_axis* and *col_axis*, respectively.

Upon successful completion of **gen_lu_solve** in the square case (*m* = *n*), each matrix *B* is overwritten with the solution to $AX = B$.

Upon successful completion of **gen_lu_solve_tra** in the square case, each *B* is overwritten with the solution to $A^{T}X=B$.

Upon successful completion of **gen_lu_solve** in the non-square case (*m* > *n*), each *B* is overwritten with the solution to the system $L[U]_{mm}X = B$, where $[U]_{mm}$ is an *m* x *m* matrix defined in the Description section, below. (However, no *m* x *m* matrices are actually stored.)

Copyright © 1994 Thinking Machines Corporation

Upon successful completion of **gen_lu_solve_tra** in the non-square case, each B is overwritten with the solution to $(L[U]_{mm})^{\mathrm{T}}X = B_0$, where B_0 is B with the last $m - n$ block cyclic rows zeroed. (However, no m x m matrices are actually stored.)

Upon return from a factor application routine, each B is overwritten by the specified product: $L^{-1}B$, $L^{-\mathrm{T}}B$, LB, or $L^{\mathrm{T}}B$ ($m \geq n$); $U^{-1}B$ or $U^{-\mathrm{T}}B$ ($m = n$); $[U]_{mm}{}^{-1}B$ or $[U]_{mm}{}^{-\mathrm{T}}B$ ($m > n$). For the exact definitions of the factors that are applied to B, see the Description section.

The **gen_lu_zero_rows** routine zeroes the last $m-limit$ block cyclic rows of each B within ***B***.

Get-Factor Routines. When you call **gen_lu_get_l** or **gen_lu_get_u**, ***B*** must have the same rank, axis extents, layout directives, data type, and precision as ***A***. ***B*** contains an m x n matrix B corresponding to each A in ***A***. The rows and columns of each B are represented by *row_axis* and *col_axis*, respectively. These axes may have extents greater than m and n**,** respectively; that is, each B may be contained (as the upper left-hand m x *nrhs* elements) in a larger matrix within ***B***.

In the square case ($m = n$), upon completion of **gen_lu_get_l**, each B is overwritten with the factor L of the corresponding A within ***A***. The **gen_lu_get_l** routine is available in the square case only; to obtain L in the non-square case, you must apply **gen_lu_apply_l** to the m x m identity matrix.

Upon completion of **gen_lu_get_u**, each B is overwritten with the factor U of the corresponding A.

The L factor produced by **gen_lu_get_l** contains the effects of pivoting. Furthermore, the L and U factors produced by **gen_lu_get_l** and **gen_lu_get_u** are in block cyclic form. To obtain the factors in elementwise consecutive order, you may use the **compute_fe_block_cyclic_perms** and **permute_cm_matrix_axis_from_fe** routines. For the exact definitions of the L and U factors returned by the get-factor routines, see the Description section.

Do not supply the arrays obtained from the get-factors routines in the input array ***A*** when you call the solver or factor application routines.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

A_norm Real CM array with the same rank and precision as ***A***. The axes identified by *row_axis* and *col_axis* in the **gen_lu_factor** call must have extent 1. The remaining axes of ***A_norm*** must match the instance axes of ***A*** in order of declaration and extents. Thus, each matrix *A* embedded in the array ***A*** that you originally supplied to **gen_lu_factor** corresponds to a real number in ***A_norm***.

Upon successful completion of **gen_lu_infinity_norm_inv**, the estimated infinity norm of the *inverse* of each matrix *A* within the original input array ***A*** is placed in the corresponding position of ***A_norm***.

A Real or complex CM array of rank greater than or equal to 2. Must be distinct from ***B***.

Factor Routine. When you call **gen_lu_factor**, ***A*** should contain one or more instances of a coefficient matrix *A* to be factored. Each *A* is assumed to be dense with dimensions *m* x *n*, where *m* is greater than or equal to *n*, with rows counted by axis *row_axis* and columns counted by axis *col_axis*. These axes may have extents greater than *m* and *n*, respectively; that is, each *A* may be contained (as the upper left-hand *m* x *n* elements) in a larger matrix within ***A***. Upon completion of **gen_lu_factor**, each *A* in ***A*** is overwritten with data giving the corresponding *LU* factors, in a form used by the other *LU* routines.

All Other Routines. When you call any of the other *LU* routines, ***A*** must have the same data type, precision, rank, and shape (axis extents and layout directives, including orderings and weights) as the original ***A*** that was factored. You must also be using the same partition size as when you originally factored ***A***. Supply in ***A*** the values returned in ***A*** by **gen_lu_factor**.

m Scalar integer variable. The number of rows in each coefficient matrix *A* within ***A***. Also, the number of rows in each right-hand side *B* (or, if each *B* is a single vector, the number of elements in *B*). Must be greater than or equal to *n*.

If you intend to call **gen_lu_infinity_norm_inv**, *m* must equal *n*, since each matrix *A* within ***A*** must be invertible, and therefore square.

n Scalar integer variable. The number of columns in each coefficient matrix *A* within ***A***. Must be less than or equal to *m*.

Copyright © 1994 Thinking Machines Corporation

If you intend to call **gen_lu_infinity_norm_inv**, m must equal n, since each matrix A within $\boldsymbol{A}$ must be invertible, and therefore square.

row_axis Scalar integer variable. Identifies the axis of $\boldsymbol{A}$ that counts the rows of each matrix A. The extent of this axis must be at least m; *row_axis* must be in the range 1 through the rank of $\boldsymbol{A}$, inclusive; and *row_axis* and *col_axis* must not be equal. (For definitions of row and column axes, see Chapter 1.)

col_axis Scalar integer variable. Identifies the axis of $\boldsymbol{A}$ that counts the columns of each matrix A. The extent of this axis must be at least n; *col_axis* must be in the range 1 through the rank of $\boldsymbol{A}$, inclusive; and *row_axis* and *col_axis* must not be equal. (For definitions of row and column axes, see Chapter 1.)

nblock Scalar integer variable. Blocking factor. The blocking factor you specify when you call **gen_lu_factor** is also used in any subsequent *LU* solver, factor application, get-factor, or infinity norm calls in which you supply the setup ID returned by **gen_lu_factor**. Use these guidelines when choosing an *nblock* value:

- If each instance of A and B is completely local to a vector unit, and you are either not using pivoting, or using pivoting with real (single- or double-precision) data, setting *nblock* to 1 will decrease memory requirements and improve performance for small m and n. For an additional performance improvement in this special case, see the Notes section, below.
- For typical applications, *nblock* = 16 is a good choice. An *nblock* value of 8 or 32 may yield faster factorization in some cases.
- *nblock* should always be $\leq n$; *nblock* values $> n$ use excess time and especially memory.
- For a single right-hand-side vector, the solver routines will most likely be faster with a larger value of *nblock*. On the other hand, the amount of auxiliary storage used is proportional to *nblock*, so if memory is tight, a smaller *nblock* may be a better choice.

Copyright © 1994 Thinking Machines Corporation

- Choose an *nblock* value that is less than or equal to the subgrid lengths in both dimensions. For optimal performance, ensure that the subgrid length in each dimension is a multiple of *nblock*.

pivoting_strategy Scalar integer variable specifying the pivoting strategy to be used. Specify one of the following values:

CMSSL_partial_pivoting
Selects partial pivoting. The pivoting process searches for the largest element in the pivot column; rows are, in effect, permuted. Note that this implementation does not use block or parallel pivoting; it finds one pivot row at a time.

CMSSL_no_pivoting
Selects no pivoting. The pivot is taken from the block cyclic diagonal.

nrhs Scalar integer variable. The number of columns in each instance B within $\boldsymbol{B}$. Must be greater than or equal to 1. If each B is a single vector, supply 1 for the value of *nrhs*.

limit Scalar integer variable. Must be in the range from 0 through m. The **gen_lu_zero_rows** routine zeroes the last $m-limit$ block cyclic rows of each two-dimensional matrix defined by axes *row_axis* and *col_axis* of $\boldsymbol{B}$.

unit Scalar integer. Valid unit number associated with the file to or from which the *LU* state is to be written or read. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a device or socket with a unit number). The **save_gen_lu** and **restore_gen_lu** calls write and read data using **CMF_CM_ARRAY_TO_FILE_SO** and **CMF_CM_ARRAY_FROM_FILE_SO**, respectively. The **save_gen_lu_fms** and **restore_gen_lu_fms** calls write and read data using **CMF_CM_ARRAY_TO_FILE_FMS** and **CMF_CM_ARRAY_FROM_FILE_FMS**, respectively. You must rewind the file before calling **restore_gen_lu** or **restore_gen_lu_fms**.

Copyright © 1994 Thinking Machines Corporation

iostat Scalar integer variable. Upon return, contains the status of the I/O operation. If *ier* = 0, *iostat* contains the number of bytes written or read. For the meanings of other *iostat* codes (returned if *ier* = −200), refer to the descriptions of **CMF_CM_ARRAY_TO_FILE_SO** (for **save_gen_lu**), **CMF_CM_ARRAY_TO_FILE_FMS** (for **save_gen_lu_fms**), **CMF_CM_ARRAY_FROM_FILE_SO** (for **restore_gen_lu**), and **CM_ARRAY_FROM_FILE_FMS** (for **restore_gen_lu_fms**) in the CM Fortran documentation set.

ier Scalar integer variable. Return code; set to 0 upon successful return.

Values between −1 and −9, inclusive, indicate problems with one or more of the CM arrays containing matrices in any of the *LU* calls:

−1	Invalid array home. The array must be a CM array.
−2	Invalid rank; must be ≥ 2.
−3	Invalid column extent; must be $\geq m$.
−4	Invalid row extent; must be $\geq n$.
−9	Invalid data type; must be real or complex (single- or double-precision).

Values that are multiples of −10 indicate problems with non-array arguments:

−10	System failed to allocate the setup object, *setup*.
−20	*m*, *n*, or *nrhs* is invalid; all must be > 0 and *m* must be greater than or equal to *n*.
−30	*row_axis* or *col_axis* is invalid. $1 \leq row_axis, col_axis \leq$ rank ($\boldsymbol{A}$) must be true, and *row_axis* and *col_axis* must not be equal.
−40	*nblock* is invalid; it must be greater than or equal to 1.
−50	*pivoting_strategy* is invalid; must be **CMSSL_partial_pivoting** or **CMSSL_no_pivoting**.
−60	*nrhs* is invalid.

Copyright © 1994 Thinking Machines Corporation

−80 You specified *m* not equal to *n* in the factorization call associated with this call to **gen_lu_infinity_norm_inv** or **gen_lu_get_l**.

−100 *setup* is invalid. (You did not supply the value returned by **gen_lu_factor**.)

Values between −102 and −108, inclusive, indicate problems with the consistency of ***A*** or ***B*** in a solver, factor application, or get-factor routine:

−102 The rank of ***A*** or ***B*** is invalid (must be $\geq$ 2 for ***A*** or $\geq$ 1 for ***B***), or is inconsistent with the rank of ***A*** in the factorization call.

−105 The extents of the instance axes of ***A*** or ***B*** are inconsistent with those of ***A*** in the factorization call.

−106 ***B*** must have the same layout directives as ***A*** when you call **gen_lu_get_u** or **gen_lu_get_l**.

−108 The data type or precision of ***A*** or ***B*** is inconsistent with that of ***A*** in the factorization call.

The save and restore routines return the following value if they encounter an I/O error:

−200 I/O error. See the value of *iostat* for more information.

DESCRIPTION

Given a CM array ***A*** containing one or more instances of a dense *m* x *n* matrix *A* (with $m \geq n$), and a CM array ***B*** containing corresponding right-hand sides *B*, the *LU* routines perform the operations listed below for each instance.

Factorization routine:

gen_lu_factor Uses Gaussian elimination, with or without partial pivoting, to perform the factorization $A = LU$. When you specify pivoting, the effects of pivoting are included in *L*.

In the non-square case ($m > n$), the factorization is defined so that *L* is *m* x *m* and

Copyright © 1994 Thinking Machines Corporation

U is the m x n matrix $[U]_{mm}[I]_{mn}$, where $[U]_{mm}$ is an m x m matrix defined "Alternate View of the Non-Square Factorization," below, and $[I]_{mn}$ is the m x n matrix consisting of the n x n identity matrix extended by $m-n$ rows of zeroes. However, no m x m matrices are actually stored; L and U are stored together in an m x n matrix.

Save and restore routines:

save_gen_lu Saves internal information about the LU factors in a file using serial-order I/O.

save_gen_lu_fms Saves internal information about the LU factors in a file using fixed-machine-size I/O.

restore_gen_lu Restores internal information about the LU factors from a file using serial-order I/O.

restore_gen_lu_fms Restores internal information about the LU factors from a file using fixed-machine-size I/O.

Solver routines:

gen_lu_solve Uses the LU factors returned by **gen_lu_factor** to solve $AX = B$. In the non-square case ($m > n$), the solver routine solves $L[U]_{mm}X = B$.

gen_lu_solve_tra In the square case, **gen_lu_solve_tra** uses the LU factors returned by **gen_lu_factor** to solve $A^{\mathrm{T}}X = B$. In the non-square case ($m > n$), conceptually, **gen_lu_solve_tra** produces a solution to $A^{\mathrm{T}}X = B_n$; B_n is the first n block cyclic rows of B. To perform this operation, **gen_lu_solve_tra** solves the system $(L[U]_{mm})^{\mathrm{T}}X = B_0$, where B_0 is B with the last $m - n$ block cyclic rows zeroed.

Factor application routines: These routines use the factors produced by the LU factorization routine to solve "triangular" systems of the form $UX=B$, $U^{\mathrm{T}}X=B$, $LX=B$, $L^{\mathrm{T}}X=B$, $L^{-1}X=B$, $L^{-\mathrm{T}}X=B$. (Because of implementation details, L and U

Copyright © 1994 Thinking Machines Corporation

are not actually triangular; see "The *LU* Factors Defined," below.) In the non-square case, replace U with $[U]_{mm}$.

gen_lu_apply_l_inv Given the *LU* factors returned by **gen_lu_factor**, applies L^{-1} to B.

gen_lu_apply_u_inv Given the *LU* factors returned by **gen_lu_factor**, applies U^{-1} to B. In the non-square case, this routine applies $[U]_{mm}{}^{-1}$ to B.

gen_lu_apply_l_inv_tra Given the *LU* factors returned by **gen_lu_factor**, applies $L^{-\mathrm{T}}$ to B.

gen_lu_apply_u_inv_tra Given the *LU* factors returned by **gen_lu_factor**, applies $U^{-\mathrm{T}}$ to B. In the non-square case, this routine applies $[U]_{mm}{}^{-\mathrm{T}}$ to B.

gen_lu_apply_l Given the *LU* factors returned by **gen_lu_factor**, applies L to B.

gen_lu_apply_l_tra Given the *LU* factors returned by **gen_lu_factor**, applies L^{T} to B.

Get-factor routines:

gen_lu_get_l Given the *LU* factors returned by **gen_lu_factor**, produces L separately. (Available only in the square case.)

gen_lu_get_u Given the *LU* factors returned by **gen_lu_factor**, produces U separately.

Zeroing routine:

gen_lu_zero_rows Zeroes the final rows of B. The rows are counted in block cyclic order.

Infinity norm routine:

gen_lu_infinity_norm_inv Estimates the infinity norm of A^{-1}, given the *LU* factors of each A as computed by the **gen_lu_factor** routine.

Copyright © 1994 Thinking Machines Corporation

Deallocation routine:

deallocate_gen_lu	Deallocates the memory required by the above routines.

Memory Allocation and Deallocation. You must call **gen_lu_factor**, **restore_gen_lu**, or **restore_gen_lu_fms** before calling any of the other *LU* routines. The **gen_lu_factor**, **restore_gen_lu**, and **restore_gen_lu_fms** routines allocate storage space and return a setup ID. You must supply this setup ID in subsequent *LU* calls, as long as you are working with the same set of factors. You can follow one call to **gen_lu_factor**, **restore_gen_lu**, or **restore_gen_lu_fms** with multiple calls to the other *LU* routines, thus avoiding the overhead of factoring or restoring the same matrix or matrices repeatedly.

The **deallocate_gen_lu** routine deallocates the memory allocated by the factorization or restore routine and required by the other *LU* routines. Be sure to call **deallocate_gen_lu** when you have finished working with a set of *LU* factors. The **deallocate_gen_lu** routine invalidates the setup ID you specify. Attempts to use a deallocated setup ID result in errors.

You can work with more than one set of *LU* factors at a time by calling **gen_lu_factor**, **restore_gen_lu**, or **restore_gen_lu_fms** more than once without calling **deallocate_gen_lu**. Be sure to supply the correct setup ID in each subsequent *LU* call. When you have finished working with a set of factors, be sure to use **deallocate_gen_lu** to deallocate the associated memory. Repeated calls to **gen_lu_factor**, **restore_gen_lu**, or **restore_gen_lu_fms** without deallocation can cause you to run out of memory.

Factorization Routine. The **gen_lu_factor** routine uses Gaussian elimination, with or without partial pivoting, to perform the factorization $A = LU$. When you specify pivoting, the effects of pivoting are included in L.

In the non-square case ($m > n$), the factorization is defined so that L is $m \times m$ and U is the $m \times n$ matrix $[U]_{mm}[I]_{mn}$, where $[U]_{mm}$ is an $m \times m$ matrix defined "Alternate View of the Non-Square Factorization," below, and $[I]_{mn}$ is the $m \times n$ matrix consisting of the $n \times n$ identity matrix extended by $m-n$ rows of zeroes. Thus, we can view the factorization as $A = L[U]_{mm}[I]_{mn}$; however, no $m \times m$ matrices are actually stored.

Upon completion of **gen_lu_factor**, each instance of A within $\boldsymbol{A}$ is overwritten with data giving the corresponding *LU* factors. When you call an *LU* solver, factor application, get-factor, zeroing, or infinity norm routine, you must supply the same

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

A that was returned by **gen_lu_factor**. To obtain the L and U factors separately, use the get-factor routines (or, to get L in the non-square case, apply **gen_lu_apply_l** to the identity matrix).

Save and Restore Routines. You may save internal information about the LU factors in a file for use in later calls to the other LU routines. To save the LU information, call **save_gen_lu** or **save_gen_lu_fms** after the factorization is complete but before deallocating the storage space. To restore the LU information, rewind the file and call **restore_gen_lu** or **restore_gen_lu_fms**; this call is typically followed by calls to the other LU routines.

Solver Routine. In the square case ($m = n$), given the values returned in A by **gen_lu_factor**, the **gen_lu_solve** routine solves one or more instances of the system

$$AX = B$$

where A and B are corresponding instances within ***A*** and ***B***, respectively. Upon successful completion of **gen_lu_solve**, each B is overwritten with the solution.

In the non-square case ($m > n$), the **gen_lu_solve** routine solves one or more instances of the system

$$L[U]_{mm}X = B$$

where $[U]_{mm}$ is an m x m matrix defined under "Alternate View of the Non-Square Factorization," below. (However, no m x m matrices are actually stored.) Upon successful completion of **gen_lu_solve**, each B is overwritten with the solution to $L[U]_{mm}X = B$.

Steps Performed by Solver Routine: Square Case. In the square case, to solve $AX = B$, **gen_lu_solve** performs forward elimination:

Let $UX = C$;

$A = LU$ implies that $AX = B$ is equivalent to $C = L^{-1}B$

followed by back substitution:

$$X = U^{-1}C = U^{-1}(L^{-1}B)$$

To obtain this solution, the **gen_lu_solve** routine performs the following steps:

1. Applies L^{-1} to B.

Copyright © 1994 Thinking Machines Corporation

2. Applies U^{-1} to $L^{-1}B$.

To perform these steps yourself, you would

1. Call **gen_lu_apply_l_inv** to apply each L^{-1} to the corresponding right-hand side, B.
2. Call **gen_lu_apply_u_inv** to apply U^{-1} to the result from Step 1.

Steps Performed by Solver Routine: Non-Square Case. In the non-square case, to solve $L[U]_{mm}X = B$, **gen_lu_solve** performs forward elimination:

Let $[U]_{mm}X = C$;

$L[U]_{mm}X = B$ is equivalent to $C = L^{-1}B$

followed by back substitution:

$X = [U]_{mm}{}^{-1}C = [U]_{mm}{}^{-1}(L^{-1}B)$

To obtain this solution, the **gen_lu_solve** routine performs the following steps:

1. Applies L^{-1} to B.
2. Applies $[U]_{mm}{}^{-1}$ to $L^{-1}B$.

To perform these steps yourself, you would

1. Call **gen_lu_apply_l_inv** to apply each L^{-1} to the corresponding right-hand side, B.
2. Call **gen_lu_apply_u_inv** to apply $[U]_{mm}{}^{-1}$ to the result from Step 1.

Transpose Solver Routine. In the square case ($m = n$), given the values returned in A by **gen_lu_factor**, the **gen_lu_solve_tra** routine solves one or more instances of the system

$A^{\mathrm{T}}X = B$

where A and B are corresponding instances within $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. Upon successful completion of **gen_lu_solve_tra**, each B is overwritten with the solution.

In the non-square case, conceptually, the **gen_lu_solve_tra** routine produces a solution to the system $A^{\mathrm{T}}X = B_n$, where B_n is the first n block cyclic rows of B. To perform this operation, **gen_lu_solve_tra** solves the system $(L[U]_{mm})^{\mathrm{T}}X = B_0$,

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

where B_0 is B with the last $m - n$ block cyclic rows zeroed. Upon successful completion, each B is overwritten with the solution to $(L[U]_{mm})^{\mathrm{T}}X = B_0$.

Steps Performed by Transpose Solver Routine: Square Case. In the square case, to solve $A^{\mathrm{T}}X = B$, **gen_lu_solve_tra** performs forward elimination:

$A^{\mathrm{T}} = U^{\mathrm{T}}L^{\mathrm{T}}$; let $L^{\mathrm{T}}X = C$

$C = U^{-\mathrm{T}}B$

followed by back substitution:

$X = L^{-\mathrm{T}}C = L^{-\mathrm{T}}(U^{-\mathrm{T}}B)$

To obtain this solution, the **gen_lu_solve_tra** routine performs the following steps:

1. Applies $U^{-\mathrm{T}}$ to B.
2. Applies $L^{-\mathrm{T}}$ to $U^{-\mathrm{T}}B$.

To perform these steps yourself, you would

1. Call **gen_lu_apply_u_inv_tra** to apply each $U^{-\mathrm{T}}$ to the corresponding right-hand side, B.
2. Call **gen_lu_apply_l_inv_tra** to apply $L^{-\mathrm{T}}$ to $U^{-\mathrm{T}}B$.

Steps Performed by Transpose Solver Routine: Non-Square Case. In the non-square case, to solve $(L[U]_{mm})^{\mathrm{T}}X = B_0$, the **gen_lu_solve_tra** routine performs forward elimination:

Let $L^{\mathrm{T}}X = C$;

$(L[U]_{mm})^{\mathrm{T}}X = B_0$ is equivalent to $C = [U]_{mm}{}^{-\mathrm{T}}B_0$

followed by back-substitution:

$X = L^{-\mathrm{T}}C = L^{-\mathrm{T}}[U]_{mm}{}^{-\mathrm{T}}B_0$

To obtain this solution, the **gen_lu_solve_tra** routine performs the following steps:

1. Applies $[U]_{mm}{}^{-\mathrm{T}}$ to B.
2. Zeroes the last $m-n$ block cyclic rows of $[U]_{mm}{}^{-\mathrm{T}}B$.
3. Applies $L^{-\mathrm{T}}$ to the result from Step 2.

Copyright © 1994 Thinking Machines Corporation

To perform these steps yourself, you would

1. Call **gen_lu_apply_u_inv_tra** to apply each $[U]_{mm}{}^{-\mathrm{T}}$ to the corresponding right-hand side, B.

2. Call **gen_lu_zero_rows** to zero the last $m - n$ block cyclic rows of each two-dimensional matrix defined by *row_axis* and *col_axis* of the result from Step 1. (Note: Step 2 is not required if the last $m - n$ block cyclic rows of each B were set to zero prior to Step 1.)

3. Call **gen_lu_apply_l_inv_tra** to apply $L^{-\mathrm{T}}$ to the result from Step 2.

Step 2 is required so that inactive data in the last $m - n$ rows of the right-hand-sides B does not affect the solution. This zeroing is required only when you are solving $A^{\mathrm{T}}X = B$.

Factor Application Routines. The **gen_lu_apply_l_inv**, **gen_lu_apply_u_inv**, **gen_lu_apply_l_inv_tra**, **gen_lu_apply_u_inv_tra**, **gen_lu_apply_l**, and **gen_lu_apply_l_tra** routines allow you to apply matrices derived from the LU factors to arbitrary matrices or vectors B contained in $\boldsymbol{B}$. Upon completion of the routine, each B in $\boldsymbol{B}$ is overwritten with the specified product.

The matrices L^{-1}, $L^{-\mathrm{T}}$, L, and L^{T} applied by the **gen_lu_apply_l_inv**, **gen_lu_apply_l_inv_tra**, **gen_lu_apply_l**, and **gen_lu_apply_l_tra** routines are given by the definitions of L in "The LU Factors Defined," below. The inverses are true inverses, in the sense that $L^{-1}L = LL^{-1} = I$. The inverses are also true in the block cyclic space; that is, $L_c{}^{-1}L_c = L_cL_c{}^{-1} = I$.

In the square case, the **gen_lu_apply_u_inv** and **gen_lu_apply_u_inv_tra** routines apply U^{-1} and $U^{-\mathrm{T}}$ as derived from the definitions of U in "The LU Factors Defined," below; the inverses are true inverses in both the original space and the block cyclic space.

In the non-square case, the **gen_lu_apply_u_inv** and **gen_lu_apply_u_inv_tra** routines apply $[U]_{mm}{}^{-1}$ and $[U]_{mm}{}^{-\mathrm{T}}$, respectively, to the arbitrary matrices you supply (although no m x m matrices are actually stored). The inverses are true inverses in both the original space and the block cyclic space.

In most cases, you should use a solver routine, rather than using the factor application routines separately, to solve $AX = B$ or $A^{\mathrm{T}}X = B$. Using the factor application calls may require extra communication because of alignment and permutations required internally for B and X.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Get-Factor Routines. The **gen_lu_get_l** and **gen_lu_get_u** routines return the L and U factors defined under "The LU Factors Defined," below. Upon completion, each instance B within ***B*** is overwritten by the requested factor of the corresponding A within ***A***. In the non-square case, **gen_lu_get_l** is not available; to obtain the m x m matrix L, you must apply **gen_lu_apply_l** to the m x m identity matrix.

The L factor produced by **gen_lu_get_l** contains the effects of pivoting. Furthermore, the L and U factors produced by **gen_lu_get_l** and **gen_lu_get_u** are in block cyclic form. To obtain the factors in elementwise consecutive order, you may use the **compute_fe_block_cyclic_perms** and **permute_cm_matrix_axis_from_fe** routines. (For an example, see the on-line sample code in the subdirectory **`block-cyclic/cmf/`** of the CMSSL examples directory.) For the exact definitions of the L and U factors, see "The LU Factors Defined," below.

Zeroing Routine. The **gen_lu_zero_rows** routine zeroes the last $m - limit$ block cyclic rows of each two-dimensional matrix defined by *row_axis* and *col_axis* of ***B***. This routine is useful if you want to perform separately the zeroing that the transpose solver routine performs, as described above.

Infinity Norm Routine. Given the LU factors returned by **gen_lu_factor**, the **gen_lu_infinity_norm_inv** routine estimates the infinity norm of each matrix A^{-1}. Upon successful completion, the infinity norm of each A^{-1} is placed in the position of ***A_norm*** corresponding to the position of A in the original array ***A*** you supplied to **gen_lu_factor**.

The infinity norm of a matrix M, denoted here by $||\, M \,||_\infty$, is defined by

$$||\, M \,||_\infty = \max_{||\, x \,||_\infty = 1} ||\, Mx \,||_\infty$$

where the infinity norm of a vector, $||\, x \,||_\infty$, is defined as the maximum of the absolute values of the vector components:

$$||\, x \,||_\infty = \max_i |\, x_i \,|$$

The infinity-norm condition number of a matrix M is equal to the product of $||M||_\infty$ and $||M^{-1}||_\infty$.

Copyright © 1994 Thinking Machines Corporation

The *LU* Factors Defined. The **gen_lu_factor** routine actually factors a block cyclic permutation, A_c, of each matrix A that you supply in $\boldsymbol{A}$. The matrix A_c is defined by

$$A_c = P_1{}^{-1} A P_2$$

where P_1 and P_2 are the permutations giving the correspondence between standard and block cyclic row order and column order, respectively, for A. (These matrices depend on the array size and layout, the partition size, and the blocking factor you supply.) The matrix A_c is factored into

$$A_c = P_c{}^{-1} L_c U_c \quad \text{(factorization with pivoting)}$$
$$A_c = L_c U_c \quad \text{(factorization without pivoting).}$$

L_c is lower triangular and U_c is upper triangular. In the non-square case, U_c is m x n and is zero except for (i, j) where $i \leq j \leq n$. P_c is the permutation matrix resulting from the pivoting process. We define the L and U factors of A as follows:

$m \geq n$, pivoting
$$L = P_1 P_c{}^{-1} L_c P_1{}^{-1}$$
$$U = P_1 U_c P_2{}^{-1}$$

$m = n$, no pivoting
$$L = P_1 L_c P_1{}^{-1}$$
$$U = P_1 U_c P_1{}^{-1}$$

$m > n$, no pivoting
$$L = [P_2]_{mm}\, L_c\, P_1{}^{-1}$$
$$U = P_1 U_c P_2{}^{-1}$$

where $[P_2]_{mm}$ is the m x m permutation matrix formed by replacing the upper left n x n elements of the m x m identity matrix with P_2.

For the derivations of the definitions of L and U, see Section 5.2.3.

Alternate View of the Non-Square Factorization. In the non-square case, we define $[U_c]_{mm}$ to be the m x m extension of U_c shown below.

$$\begin{array}{c} \\ m \end{array} \begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \left[\begin{array}{c|c} U_c & \begin{array}{c} 0 \\ \hline \mathrm{I} \end{array} \end{array}\right] \end{array} \begin{array}{c} \\ \begin{array}{c} n \\ m-n \end{array} \end{array}$$

Note that by the structure of this matrix, we have $U_c = [U_c]_{mm}[I]_{mn}$. We now define the m x m matrix

$$[U]_{mm} = P_1 [U_c]_{mm} [P_2{}^{-1}]_{mm}.$$

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

In the non-square case, with or without pivoting, we now have

$$\begin{aligned} U &= P_1 U_c {P_2}^{-1} \\ &= P_1 [U_c]_{mm} [I]_{mn} {P_2}^{-1} \\ &= P_1 [U_c]_{mm} [{P_2}^{-1}]_{mm} [I]_{mn} \\ &= [U]_{mm} [I]_{mn}. \end{aligned}$$

Thus, in the non-square case, we can view the factorization of A as

$$A = L[U]_{mm}[I]_{mn}.$$

NOTES

No Nodal Library Support for Save and Restore Routines. The save and restore routines do not work with the nodal CMSSL library.

NaNs and Infinities. As mentioned above, the matrices A and B may be contained (as the upper left-hand submatrices) in larger matrices within the arrays ***A*** and ***B***, respectively. In this case, if there are NaNs or infinities in the larger matrix outside of A or B, it is possible that other locations outside of A or B could become NaNs or infinities as well.

Distinct Variables. The input CM arrays ***A*** and ***B*** must be distinct variables.

Include the CMSSL Header File. The **gen_lu_factor** routine is a function and uses symbolic constants. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls these routines. This file declares the types of the CMSSL functions and symbolic constants.

Setup ID. The internal variable *setup* is required for communicating information between the factorization routine and the other *LU* routines. The application must not modify the contents of this variable.

Saving and Restoring the *LU* State. If you want to save the internal state in one program run and restore it in a different run, you must save the array of factored matrices in a file in addition to saving the internal state using **save_gen_lu** or **save_gen_lu_fms**. When you read the array back into memory prior to restoring the inter-

Copyright © 1994 Thinking Machines Corporation

nal state, you must use the same partition size as when you originally performed the factorization; and the restored array must have exactly the same shape (axis extents and layout directives, including orderings and weights) as when you saved it.

Nondegeneracy Required. In the current release, each matrix A within $\boldsymbol{A}$ must have a column space of rank n when you call one of the solver routines.

Rank of *B*. The following example illustrates the options for defining the rank of $\boldsymbol{B}$. Suppose $\boldsymbol{A}$, n, m, *row_axis*, and *col_axis* are defined as follows:

$\boldsymbol{A}$ (5, 10, 5)
$m = n = 5$
row_axis = 1
col_axis = 3

and each B in $\boldsymbol{B}$ is a single vector. You may define $\boldsymbol{B}$ in either of the two following (equivalent) ways:

$\boldsymbol{B}$ (5, 10, 1)
$\boldsymbol{B}$ (5, 10)

On the other hand, if you define

$\boldsymbol{A}$ (5, 10, 5)
$m = n = 5$
row_axis = 3
col_axis = 1

then the possibilities for $\boldsymbol{B}$ are as follows:

$\boldsymbol{B}$ (1, 10, 5)
$\boldsymbol{B}$ (10, 5)

Performance. Performance improves for larger subgrid sizes (and therefore depends upon the layout of $\boldsymbol{A}$). For information on subgrids, refer to the CM Fortran documentation set.

To optimize performance, follow these guidelines:

- Lay out $\boldsymbol{A}$ so that the subgrid sizes along axes *row_axis* and *col_axis* differ from one another by no more than a factor of 4 or 5.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- Use axis extents exactly equal to m x n for the matrices A and m x *nrhs* for the matrices B. Use the same processing element layout for the arrays ***A*** and ***B***.
- If each instance of A and B is completely local to a vector unit, and you are either not using pivoting, or using pivoting with real (single- or double-precision) data, setting *nblock* to 1 will decrease memory requirements and improve performance for small m and n. For an additional performance improvement in this special case (assuming $nblock = 1$),
  - Ensure that *row_axis* and *col_axis* have the highest possible strides. To achieve this end, if you are using the CM Fortran **-axisreorder** switch, either declare these axes as the two leftmost axes, and/or declare them as **:serial**. If you are using the **-noaxisreorder** switch, declare these axes as the two rightmost axes.
  - In addition, if you have real single-precision data and you have satisfied the point above, ensure that the product of the subgrid lengths of the instance axes is an even number.
- For typical applications, $nblock = 16$ is a good choice. An *nblock* value of 8 or 32 may yield faster factorization in some cases.
- *nblock* should always be $\leq n$; *nblock* values $> n$ use excess time and especially memory.
- For a single right-hand-side vector, the solver routines will most likely be faster with a larger value of *nblock*. On the other hand, the amount of auxiliary storage used is proportional to *nblock*, so if memory is tight, a smaller *nblock* may be a better choice.
- Choose an *nblock* value that is less than or equal to the subgrid lengths in both dimensions. For optimal performance, ensure that the subgrid length in each dimension is a multiple of *nblock*.

Numerical Complexity. If the matrices A have dimensions (m x n), the matrices B have *nrhs* right-hand sides, and I is the number of instances (the product of all axis extents except axes *row_axis* and *col_axis*), then:

- The LU factorization routine requires approximately $[m-(n/3)]n^2I$ floating-point operations for real operands and $4[m-(n/3)]n^2I$ floating-point operations for complex operands.

Copyright © 1994 Thinking Machines Corporation

- The *LU* solver routine requires approximately 2**nrhs*mnI* floating-point operations for real operands and 8**nrhs*mnI* floating-point operations for complex operands.

Performance Cost of Pivoting. The cost of pivoting is very much dependent on size and layout. The extra cost of pivoting is greatest for relatively small matrices. For very large matrices (using nearly all processing element memory), the performance of the factorization with pivoting is comparable to the performance without pivoting, whereas the solver remains about 50% slower for the pivoting version.

EXAMPLES

Sample CM Fortran code that uses the *LU* routines can be found on-line in the subdirectories

```
lu/cmf/
```

and

```
infinity-norm/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

5.3 Routines for Solving Linear Systems Using Householder Transformations ("*QR*" Routines)

This section describes the CMSSL routines for solving linear systems using Householder transformations (commonly referred to as the "*QR*" routines). The following topics are included:

- the *QR* routines
- Householder algorithm
- blocking and load balancing
- the *QR* factors defined
- choosing a backsolve strategy
- results of the solver routines
- numerical stability
- the pivoting option: working with ill-conditioned systems
- scaling
- saving and restoring the *QR* state

For detailed descriptions of the *QR* routines (including calling sequences, argument definitions, and information about usage), refer to the man page at the end of this section.

Throughout this section, the following conventions are used:

- $\overline{M}$ denotes the conjugate of M.
- $M^{-\mathrm{T}}$ denotes $(M^{-1})^{\mathrm{T}} = (M^{\mathrm{T}})^{-1}$.
- "*A*" refers to a matrix being factored and "*B*" refers to the right-hand side(s). One or more instances of A and B are embedded (possibly within larger matrices) in the CM arrays $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively; the operations described are performed on all instances concurrently. The man page provides details about $\boldsymbol{A}$ and $\boldsymbol{B}$.

Copyright © 1994 Thinking Machines Corporation

5.3.1 The *QR* Routines

Given a CM array ***A*** containing one or more instances of a dense $m \times n$ matrix A, and a CM array ***B*** containing corresponding right-hand sides B, the *QR* routines perform the operations listed below for each instance.

Factorization routine:

gen_qr_factor — Uses Householder transformations, with or without column pivoting, to perform a *QR* factorization of A. In the no-pivoting case, the factorization is $A = QR$. If you specify pivoting, **gen_qr_factor** computes $A = QRP^{-1}$, where P is the permutation matrix that corresponds to the pivoting process.

In the non-square case ($m > n$), the factorization is defined so that Q is $m \times m$ and R is the $m \times n$ matrix $[R]_{mm}[I]_{mn}$, where $[R]_{mm}$ is an $m \times m$ matrix defined in Section 5.3.4 and $[I]_{mn}$ is the $m \times n$ matrix consisting of the $n \times n$ identity matrix extended by $m-n$ rows of zeroes. However, no $m \times m$ matrices are actually stored; Q and R are stored together in an $m \times n$ matrix.

Save and restore routines:

save_gen_qr — Saves internal information about the *QR* factors in a file using serial-order I/O.

save_gen_qr_fms — Saves internal information about the *QR* factors in a file using fixed-machine-size I/O.

restore_gen_qr — Restores internal information about the *QR* factors from a file using serial-order I/O.

restore_gen_qr_fms — Restores internal information about the *QR* factors from a file using fixed-machine-size I/O.

Solver routines:

gen_qr_solve — Uses the *QR* factors returned by **gen_qr_factor** to solve $AX = B$. In the non-square case, the

Copyright © 1994 Thinking Machines Corporation

first n rows of the B are overwritten with the least squares solution to $AX = B$; the solver routine solves $Q[R]_{mm}X = B$ (in the no-pivoting case) or $Q[R]_{mm}[P^{-1}]_{mm}X = B$ (in the pivoting case), where $[P^{-1}]_{mm}$ is an m x m extension of P^{-1} defined in Section 5.3.4.

gen_qr_solve_tra In the square case, **gen_qr_solve_tra** uses the QR factors returned by **gen_qr_factor** to solve $A^{\mathrm{T}}X = B$. In the non-square case, conceptually, **gen_qr_solve_tra** provides the minimal 2-norm solution to $A^{\mathrm{T}}X = B_n$; B_n is the first n block cyclic rows of B. To perform this operation, **gen_qr_solve_tra** solves $(Q[R]_{mm})^{\mathrm{T}}X = B_0$ (in the no-pivoting case) or $(Q[R]_{mm}[P^{-1}]_{mm})^{\mathrm{T}}X = B_0$ (in the pivoting case), where B_0 is B with the last $m - n$ block cyclic rows zeroed.

Factor application routines: These routines use the factors produced by the QR factorization routine to solve "triangular" systems of the form $RX{=}B$ and $R^{\mathrm{T}}X{=}B$ and "trapezoidal" systems of the form $QX{=}B$ or $Q^{\mathrm{T}}X{=}B$. Because of implementation details, R and Q are not actually triangular and trapezoidal, respectively; see Section 5.3.4. In the non-square case, replace R with $[R]_{mm}$.

gen_qr_apply_q Given the QR factors returned by **gen_qr_factor**, applies Q (or $\overline{Q}$, in the case of complex data) to B.

gen_qr_apply_q_tra Given the QR factors returned by **gen_qr_factor**, applies Q^{T} (or Q^{H}, in the case of complex data) to B. Note that since Q is orthogonal (or unitary, in the complex case), $Q^{\mathrm{T}} = Q^{-1}$ (or, in the complex case, $Q^{\mathrm{H}} = Q^{-1}$).

gen_qr_apply_r_inv Given the QR factors returned by **gen_qr_factor**, applies R^{-1} to B. In the non-square case, this routine applies $[R]_{mm}{}^{-1}$ to B.

gen_qr_apply_r_inv_tra Given the QR factors returned by **gen_qr_factor**, applies $R^{-\mathrm{T}}$ to B. In the non-square case, this routine applies $[R]_{mm}{}^{-\mathrm{T}}$ to B.

Copyright © 1994 Thinking Machines Corporation

Get-*R* routine:

gen_qr_get_r Given the *QR* factors returned by **gen_qr_factor**, produces the factor *R* separately.

Pivot application routines:

gen_qr_apply_p Given the *QR* factors returned by **gen_qr_factor**, applies *P* to *B*. In the non-square case, this routine applies $[P]_{mm}$ to *B*; $[P]_{mm}$ is an *m* x *m* extension of *P* defined in Section 5.3.4. Use this routine only if you specified pivoting in the associated call to **gen_qr_factor**.

gen_qr_apply_p_inv Given the *QR* factors returned by **gen_qr_factor**, applies $P^{-1} = P^{T}$ to *B*. In the non-square case, this routine applies $[P^{-1}]_{mm} = [P^{-T}]_{mm} = [P]_{mm}{}^{-1}$ to *B*. Use this routine only if you specified pivoting in the associated call to **gen_qr_factor**.

Zeroing routine:

gen_qr_zero_rows Zeroes the final rows of *B*. The rows are counted in block cyclic order.

Diagonal manipulation routines:

gen_qr_extract_diag Given the *QR* factors returned by **gen_qr_factor**, returns the block cyclic diagonal entries of *R*.

gen_qr_deposit_diag Given the *QR* factors returned by **gen_qr_factor**, overwrites the block cyclic diagonal entries of *R* with values you supply.

Infinity norm routines:

gen_qr_infinity_norm_inv Given the *QR* factors returned by **gen_qr_factor**, estimates the infinity norm of A^{-1}. Uses the method developed by Hager; see reference 5 listed in Section 5.7. (Available only in the square case.)

Copyright © 1994 Thinking Machines Corporation

gen_qr_r_infinity_norm_inv Given the *QR* factors returned by **gen_qr_factor**, estimates the infinity norm of $(R^*)^{-1}$, where R^* is the block cyclic upper left corner formed by discarding any trailing columns of R that contain zeros on the block cyclic diagonal.

Deallocation routine:

deallocate_gen_qr Deallocates the memory required by the above routines.

NOTE

The sections that follow contain detailed information about how the *QR* routines are implemented. This information may help you use "auxiliary" *QR* routines (the routines other than the factorization and solver routines). If you only need to use the factorization and solver routines, you may not need all the details in these sections.

5.3.2 Householder Algorithm

This section provides a standard mathematical description of the Householder algorithm used by the *QR* routines. For simplicity, this description does not take into account implementation details such as blocking and load balancing. In subsequent sections (beginning with Section 5.3.4), we describe the *QR* operations and factors in a manner that does take these implementation details into account. Therefore, the matrix and vector names used in this section do not correspond exactly to those used in subsequent sections. For example, the *QR* factorization is actually performed on a block cyclic permutation, A_c, of each matrix A you supply. The matrices A, Q, and R in this section actually correspond to the matrices A_c, Q_c, and R_c described later.

Copyright © 1994 Thinking Machines Corporation

Also for the sake of simplicity, we describe the Householder algorithm here for the no-pivoting case. Details about the pivoting case are provided in later sections.

NOTE

This section assumes that A and B are real. For complex matrices, replace "orthogonal" with "unitary" and replace "Q^{T}" with "Q^{H}."

The Householder reduction algorithm computes a series of n Householder matrices that, when successively multiplied by the coefficient matrix A, yield an upper triangular matrix R. As an example, suppose A is 4 x 3. First, a Householder transformation H_1 is computed, such that H_1 applied to the first column of A yields a vector that is 0 except for its first component. H_1 is applied to all the columns of A. Next, a Householder transformation H_2 is computed such that H_2 preserves the new first column of A and H_2 applied to the new second column of A yields a vector that is 0 except for its first and second components. H_2 is applied to all the columns of A. Finally, a Householder transformation H_3 is calculated such that H_3 preserves the first two columns of A and H_3 applied to the last column of A yields a vector that is 0 except for its first, second, and third components. (See Figure 15.)

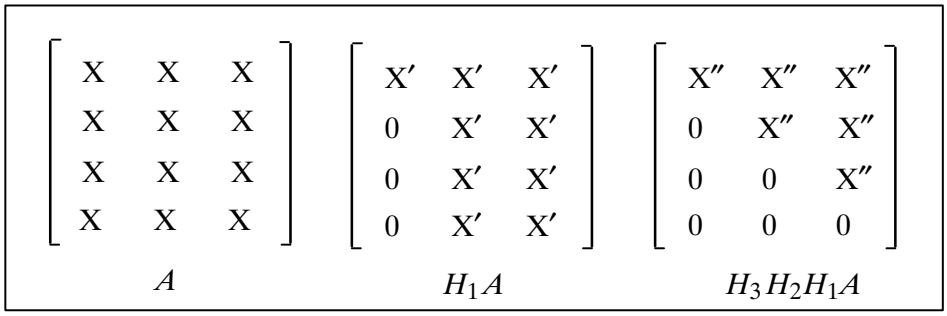


Figure 15. Successive Householder transformations (discounting blocking and load balancing).

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Each Householder transformation is defined by a corresponding vector. The transformation associated with the vector v is given by the following formula:

$$Hw = w - \left(\frac{2 < w, v >}{< v, v >} \right) v$$

Here the notation $< x,y >$ means the scalar product of the vectors x and y. If x and y have n components, then

$$<x,y> = \sum_{i=1}^{n} x_i y_i$$

(If x and y are complex, then

$$<x,y> = \sum_{i=1}^{n} x_i \ \overline{y_i}.)$$

After n Householder transformations, the upper triangular matrix R has been computed as $R = (H_n \ H_{n-1} \ ... \ H_1)A$. Since each H is orthogonal, the product $H_n \ ... \ H_1$ is also orthogonal, and is defined as Q^{T}. The nontrivial portion of Q is a series of Householder vectors.

Upon return from the factorization routine, the upper triangular portion of A is overwritten by R, and the lower triangular portion of A is overwritten by the Householder vectors that form the non-trivial portion of Q, as shown in Figure 16. Specifically, components $j+1{:}m$ of the j^{th} Householder vector are stored in $A(j+1{:}m, j)$ for all j less than m. (Bear in mind that this description does not take blocking and load balancing into account.)

Copyright © 1994 Thinking Machines Corporation

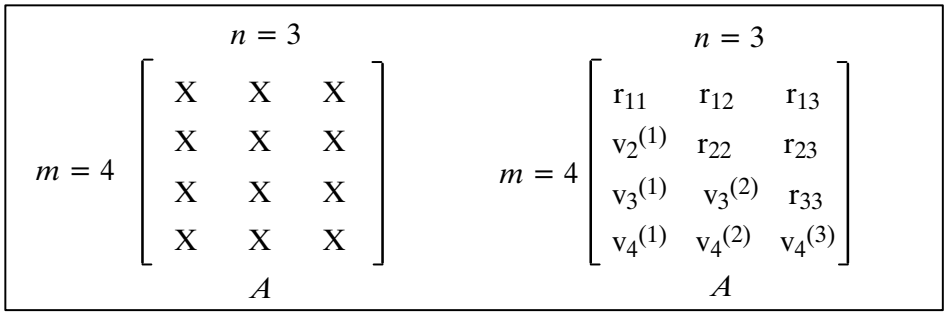


Figure 16. The upper triangular portion of A is overwritten by R and the lower triangular portion is overwritten by Householder vectors. (This figure does not take blocking and load balancing into account.)

The stored Householder vectors are normalized so that $v(1{:}j-1) = 0$ and $v(j) = 1$ for the j^{th} Householder vector. This normalization makes it possible to store only the essential part of each Householder vector in the strictly lower triangular portion of A, leaving room in the upper triangular portion (including the diagonal) for R.

The QR solver routine applies Q^{T} (the n Householder transformations corresponding to the n vectors stored in the strictly lower triangular part of A) to the right-hand sides that comprise the matrix B. In this way, the linear system $AX = B$ is transformed to $RX = Q^{\mathrm{T}}B$. This upper triangular system is solved by r concurrent back substitutions, where r is the number of right-hand-side vectors. For each right-hand-side vector b in B, the system $Ax = b$ becomes

$$x = R^{-1}Q^{\mathrm{T}}b.$$

5.3.3 Blocking and Load Balancing

The QR routines use blocking and load balancing (block cyclic ordering) to optimize performance. For information about block cyclic ordering and about the blocking factor (which is an input argument), see the section on computation of block cyclic permutations in Chapter 15, and also reference 11 listed in Section 5.7. The man page at the end of this section provides guidelines for choosing a blocking factor. The definitions of the QR factors presented in Section 5.3.4 include the effects of blocking and load balancing.

Copyright © 1994 Thinking Machines Corporation

5.3.4 The *QR* Factors Defined

The *QR* factorization routine performs the factorizaiton $A = QR$ (in the no-pivoting case) or $A = QRP^{-1}$ (in the pivoting case); P is the permutation matrix that corresponds to the pivoting process. In the non-square case ($m > n$), the factorization is defined so that Q is m x m and R is the m x n matrix defined below. However, no m x m matrices are actually stored; Q and R are stored together in an m x n matrix.

The **gen_qr_factor** routine actually factors a block cyclic permutation, A_c, of each matrix A that you supply in A. The matrix A_c is defined by

$$A_c = {P_1}^{-1} A P_2$$

where P_1 and P_2 are the permutations giving the correspondence between standard and block cyclic row order and column order, respectively, for A. (These matrices depend on the array size and layout, the partition size, and the blocking factor you supply.) The matrix A_c is factored into

$$A_c = Q_c R_c {P_c}^{-1} \quad \text{(factorization with pivoting)}$$
$$A_c = Q_c R_c \quad \text{(factorization without pivoting)}$$

Q_c is orthogonal (or unitary, in the complex case); R_c is upper triangular. In the non-square case, R_c is m x n and is zero except for (i, j) where $i \leq j \leq n$. P_c is the permutation matrix resulting from the pivoting process.

The definitions of Q and R in terms of A_c and its factors depend on the value you supply in the **gen_qr_factor** *back_solve_strategy* argument. The two possible values are **CMSSL_qr_post_permute** and **CMSSL_qr_pre_permute**. Details about the two backsolve strategies are provided in Section 5.3.5. The **CMSSL_qr_pre_permute** strategy exists only in the no-pivoting case. Therefore, we have three cases to consider: post-permutation with no pivoting, post-permutation with pivoting, and pre-permutation with no pivoting.

- *Case 1: Post-permutation, no pivoting; $m \geq n$*

  The definition

  $$A_c = {P_1}^{-1} A P_2$$

  implies that

  $$\begin{aligned} A = QR &= P_1 A_c {P_2}^{-1} \\ &= P_1 (Q_c R_c) {P_2}^{-1} \\ &= P_1 Q_c ({P_1}^{-1} P_1) R_c {P_2}^{-1} \end{aligned}$$

  from which we choose to define

Copyright © 1994 Thinking Machines Corporation

$$Q = P_1 Q_c {P_1}^{-1}$$
$$R = P_1 R_c {P_2}^{-1}.$$

Note that these definitions make sense in both the square case and the non-square case, since P_1 is m x m and P_2 is n x n.

- *Case 2: Post-permutation with pivoting; $m \geq n$*

  This case is just like the Case 1 except that we include P, the permutation matrix that corresponds to the pivoting process. We have

$$\begin{aligned} A = QRP^{-1} &= P_1 A_c {P_2}^{-1} \\ &= P_1 (Q_c R_c {P_c}^{-1}) {P_2}^{-1} \\ &= P_1 Q_c ({P_1}^{-1} P_1) R_c ({P_2}^{-1} P_2) {P_c}^{-1} {P_2}^{-1} \\ &= (P_1 Q_c {P_1}^{-1}) (P_1 R_c {P_2}^{-1}) (P_2 {P_c}^{-1} {P_2}^{-1}) \end{aligned}$$

  from which we choose to define

$$Q = P_1 Q_c {P_1}^{-1}$$
$$R = P_1 R_c {P_2}^{-1}$$
$$P^{-1} = P_2 {P_c}^{-1} {P_2}^{-1}.$$

  These definitions also make sense in both the square case and the non-square case, since P_c is n x n.

- *Case 3: Pre-permutation, no pivoting ($m = n$)*

  The **CMSSL_qr_pre_permute** strategy requires $m = n$. In this case, the factorization routine pre-permutes the matrix A to obtain

$$A^* = A P_1 {P_2}^{-1}.$$

  By the definition of block cyclic ordering, we have

$$A_c = {P_1}^{-1} A^* P_2$$

  from which it follows that

$$A_c = {P_1}^{-1} A P_1.$$

  Therefore,

$$\begin{aligned} A = QR &= P_1 A_c {P_1}^{-1} \\ &= P_1 Q_c R_c {P_1}^{-1} \\ &= P_1 Q_c ({P_1}^{-1} P_1) R_c {P_1}^{-1} \end{aligned}$$

  from which we choose to define

$$Q = P_1 Q_c {P_1}^{-1}$$
$$R = P_1 R_c {P_1}^{-1}.$$

Copyright © 1994 Thinking Machines Corporation

This definition of R makes sense only when $m = n$, since R_c is m x n and P_1 is m x m.

Summary of Factor Definitions

The factorization routine operates on A with a sequence of block Householder transformations that result in the matrix R. The sequence of transformations, which is orthogonal (unitary, in the complex case) by construction, is defined as Q^T (Q^H, in the complex case). Thus, the factorization yields the results listed below, where $Q = P_1 Q_c {P_1}^{-1}$ and $P^{-1} = P_2 {P_c}^{-1} {P_2}^{-1}$. Replace Q^T by Q^H in the complex case.

$m \geq n$, post-permutation, no pivoting
$$Q^T A = R = (P_1 \, R_c \, {P_2}^{-1})$$
$$A = Q \, (P_1 \, R_c \, {P_2}^{-1})$$

$m \geq n$, post-permutation, pivoting
$$Q^T A = RP^{-1} = (P_1 \, R_c \, {P_2}^{-1})P^{-1}$$
$$A = Q \, (P_1 \, R_c \, {P_2}^{-1})P^{-1}$$

$m = n$, pre-permutation, no pivoting
$$Q^T A = R = (P_1 \, R_c \, {P_1}^{-1})$$
$$A = Q \, (P_1 \, R_c \, {P_1}^{-1})$$

When the factorization routine returns, the block cyclic upper triangle of A is overwritten with the elements of R_c. The remaining elements of A are used internally to reconstruct Q as described in Section 5.3.2.

Factors Returned by the Get-*R* Routine

The **gen_qr_get_r** routine returns the R factors defined above.

Factors Applied by the Factor and Pivot Application Routines

The matrices Q and Q^T (or Q^H, in the complex case) applied by **gen_qr_apply_q** and **gen_qr_apply_q_tra** are given by the definitions of Q above. The inverses are true inverses; that is (in the real case), $Q^T Q = QQ^T = I$. The inverses are also true in the block cyclic space; that is, ${Q_c}^T Q_c = Q_c {Q_c}^T = I$. In the complex case, replace Q^T by Q^H in these equations.

In the square case, the **gen_qr_apply_r_inv**, **gen_qr_apply_r_inv_tra**, **gen_qr_apply_p**, and **gen_qr_apply_p_inv** routines apply R^{-1}, R^{-T}, P, and P^{-1} as derived from the definitions of R and P above. The inverses are true inverses in both the original space and the block cyclic space.

Copyright © 1994 Thinking Machines Corporation

In the non-square case, we define $[R_c]_{mm}$ to be the $m \times m$ extension of R_c shown below.

$$\begin{array}{c} \\ m \end{array} \begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \left[\begin{array}{c|c} R_c & \begin{array}{c} 0 \\ \hline \mathrm{I} \end{array} \end{array}\right] \end{array} \begin{array}{c} \\ \begin{array}{c} n \\ m-n \end{array} \end{array}$$

Note that by the structure of this matrix, we have $R_c = [R_c]_{mm}[I]_{mn}$, where $[I]_{mn}$ is the $m \times n$ matrix consisting of the $n \times n$ identity matrix extended by $m-n$ rows of zeroes.

We also define the $m \times m$ matrix $[P_2]_{mm}$ to be the $m \times m$ permutation matrix formed by replacing the upper left $n \times n$ elements of the $m \times m$ identity matrix with P_2, as follows:

$$\begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \left[\begin{array}{cc} \boxed{P_2} & \\ & \mathrm{I} \end{array}\right] \end{array} \begin{array}{c} \begin{array}{c} n \\ m-n \end{array} \end{array}.$$

Note that $[P_2{}^{-1}]_{mm} = [P_2]_{mm}{}^{-1}$. Finally, we define the $m \times m$ matrix

$$[R]_{mm} = P_1[R_c]_{mm}[P_2{}^{-1}]_{mm}.$$

The **gen_qr_apply_r_inv** and **gen_qr_apply_r_inv_tra** routines apply $[R]_{mm}{}^{-1}$ and $[R]_{mm}{}^{-\mathrm{T}}$, respectively, to the arbitrary matrices you supply (although no $m \times m$ matrices are actually stored).

Alternate View of the Non-Square Factorization

In the non-square case, we have

$$\begin{aligned} R &= P_1 R_c P_2{}^{-1} \\ &= P_1[R_c]_{mm}[I]_{mn}P_2{}^{-1} \\ &= P_1[R_c]_{mm}[P_2{}^{-1}]_{mm}[I]_{mn} \\ &= [R]_{mm}[I]_{mn}. \end{aligned}$$

Thus, in the non-square no-pivoting case, we can view the factorization of A as

Copyright © 1994 Thinking Machines Corporation

$$A = Q[R]_{mm}[I]_{mn}.$$

To obtain a similar expression of the factorization in the non-square pivoting case, we define the m x m matrix $[P_c]_{mm}$ as

$$\begin{array}{c} \\ n \\ m-n \end{array} \begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \begin{bmatrix} \boxed{Pc} & \\ & \mathrm{I} \end{bmatrix} \end{array}$$

and define $[P]_{mm} = [P_2]_{mm}[P_c]_{mm}[P_2{}^{-1}]_{mm}$. To see that $[P]_{mm}$ is what we expect it to be, we note that by the structures of its three factors, $[P]_{mm}$ is equal to

$$\begin{array}{c} \\ n \\ m-n \end{array} \begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \begin{bmatrix} \boxed{T} & \\ & \mathrm{I} \end{bmatrix} \end{array}$$

where $T = P_2P_cP_2{}^{-1}$. But since we have defined $P^{-1} = P_2P_c{}^{-1}P_2{}^{-1}$, we have $T = P$. We also note that by the structures of $[P]_{mm}$ and $[I]_{mn}$, $[P]_{mm}{}^{-1} = [P^{-1}]_{mm}$ and

$$[P^{-1}]_{mm}[I]_{mn} = [I]_{mn}P^{-1}.$$

Using this fact, we can express the factorization of A as

$$\begin{aligned} A &= QRP^{-1} \\ &= Q[R]_{mm}[I]_{mn}P^{-1} \\ &= Q[R]_{mm}[P^{-1}]_{mm}[I]_{mn}. \end{aligned}$$

Example of Layout of *R*

Figure 17 is a simple example showing the shape of the non-zero entries of (P_1 R_c $P_2{}^{-1}$) when A is a (32 x 32) matrix laid out as an (8 x 8) subgrid on each processing element of a (4 x 4)-processing-element layout. The block size in this example is 2.

Copyright © 1994 Thinking Machines Corporation

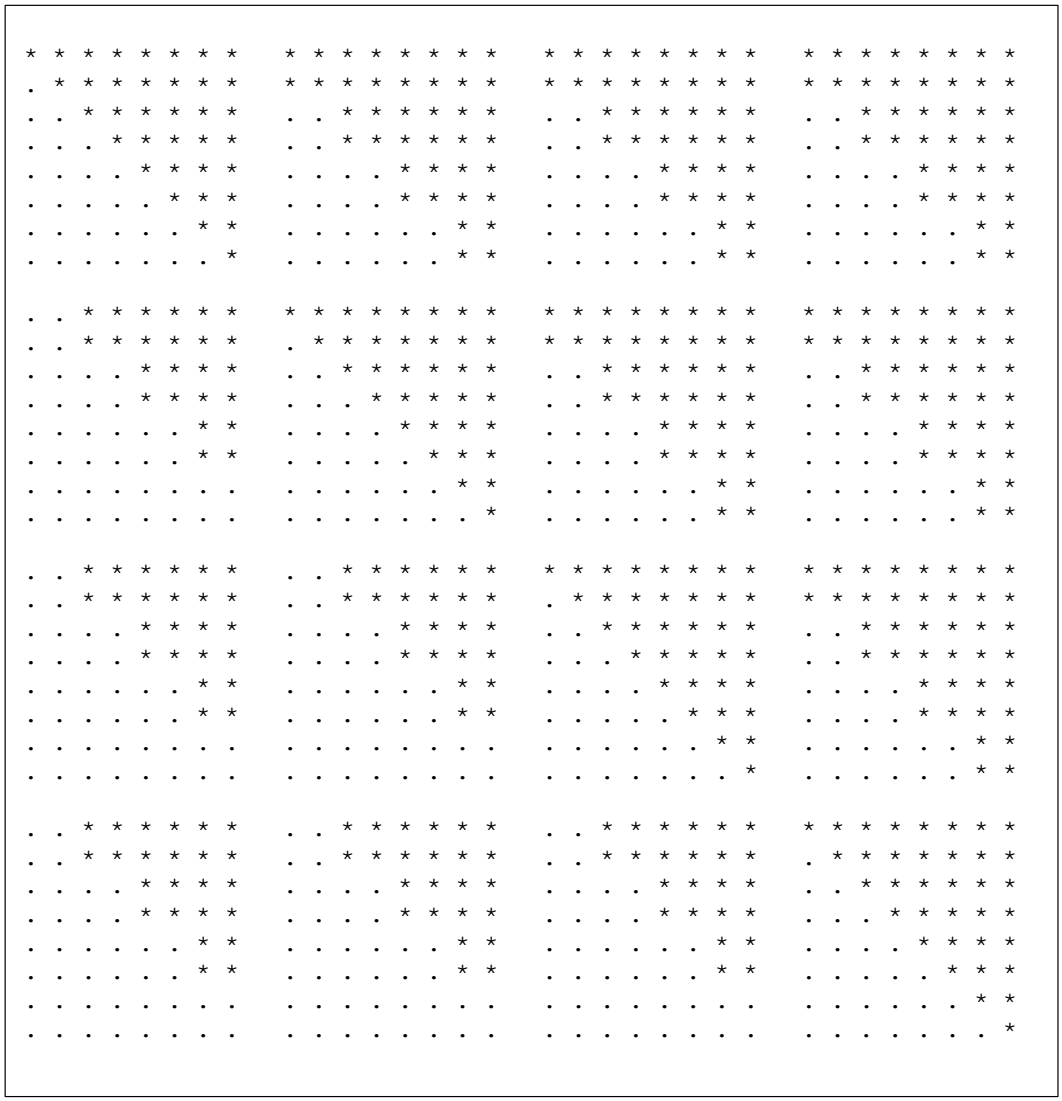


Figure 17. Non-zero entries in ($P_1 R_c P_2^{-1}$) for a (32 x 32) matrix laid out as an (8 x 8) subgrid on each processing element of a (4 x 4)-processing-element layout. The block size is 2.

Copyright © 1994 Thinking Machines Corporation

5.3.5 Choosing a Backsolve Strategy

This section provides background information about the two backsolve strategies and guidelines for choosing a strategy.

Because R_c is permuted by P_1 and P_2, the back substitution process may require further permutations in order to arrive at the solution to the original linear system. The *back_solve_strategy* argument allows you to determine when these further permutations occur:

- The **CMSSL_qr_pre_permute** strategy causes the factor routine to permute the columns of A prior to the factorization.
- The **CMSSL_qr_post_permute** strategy causes the solver routine to permute the rows of the solution after the back substitution is complete.

The **CMSSL_qr_post_permute** always works; however, your choice of backsolve strategy may affect performance. Follow these guidelines:

- If the matrices A are not square, you *must* choose **CMSSL_qr_post_permute**. Specifying **CMSSL_qr_pre_permute** with non-square matrices yields an error.
- If you are specifying pivoting, you *must* choose **CMSSL_qr_post_permute**. Specifying **CMSSL_qr_pre_permute** with pivoting yields an error.
- If the matrices A are square, each A has square subgrids, and you are not pivoting, the permutations are not required and your choice of backsolve strategy has no effect on performance.
- If the matrices A are square but do not have square subgrids, and you are not pivoting, then use these guidelines:
  - If the layouts of $\boldsymbol{A}$ and $\boldsymbol{B}$ coincide (most typically in this context, this means that the matrix axis extents are exactly n x n for each A and n x *nrhs* for each B, and that the layout of processing elements is the same for $\boldsymbol{A}$ and $\boldsymbol{B}$), then to optimize performance, choose the backsolve strategy that moves less data. The two strategies move the following amounts of data for each instance:
    - **CMSSL_qr_pre_permute** moves n^2 elements (the number of elements in each matrix A).

Copyright © 1994 Thinking Machines Corporation

- **CMSSL_qr_post_permute** moves $n\Sigma r_i$ elements, where the sum is over all calls made to the solve routine after one call to the factor routine, and r_i is the number of right-hand-side vectors in the ith call to the solve routine.

Therefore, if $n^2 < n\Sigma r_i$ or $n < \Sigma r_i$, choose **CMSSL_qr_pre_permute**; if $n > \Sigma r_i$, choose **CMSSL_qr_post_permute.** If $n = \Sigma r_i$, the two strategies are likely to yield approximately the same performance.

- If the layouts of $\boldsymbol{A}$ and $\boldsymbol{B}$ do not coincide, choose **CMSSL_qr_post_permute**, which does not move any elements in this case (as compared with **CMSSL_qr_pre_permute**, which moves n^2 elements).

Backsolve Strategy Details

The following descriptions of the two backsolve strategies are for readers who need more details about the solution process. For simplicity, this discussion describes the operations performed by **gen_qr_solve** in the square, no-pivoting case. For information about the operations performed by the transpose solver routine, see the man page at the end of this section.

NOTE

This section assumes that A and B are real. For complex matrices, replace "orthogonal" with "unitary" and replace "Q^{T}" with "Q^{H}."

In the following descriptions, bear in mind that

$Q = P_1 Q_c \, {P_1}^{-1}$ (both backsolve strategies)
$Q^{\mathrm{T}} = P_1 {Q_c}^{\mathrm{T}} {P_1}^{-1}$ (both backsolve strategies)
$R = P_1 \, R_c \, {P_2}^{-1}$ (post-permutation)
$R = P_1 \, R_c \, {P_1}^{-1}$ (pre-permutation)

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

In the **CMSSL_qr_post_permute** strategy, the solver routine backsolves the system

$$(P_1 R_c P_1^{-1}) y = (P_1 Q_c^{\mathrm{T}} P_1^{-1}) b$$

to yield

$$y = (P_1 R_c P_1^{-1})^{-1} (P_1 Q_c^{\mathrm{T}} P_1^{-1}) b$$

for each right-hand-side vector b in the matrix B. The original system Ax = b then becomes

$$(P_1 Q_c P_1^{-1}) (P_1 R_c P_2^{-1}) x = b$$

$$(P_1 R_c P_2^{-1}) x = (P_1 Q_c^{\mathrm{T}} P_1^{-1}) b$$

$$P_1 R_c (P_1^{-1} P_1) P_2^{-1} x = (P_1 Q_c^{\mathrm{T}} P_1^{-1}) b$$

$$(P_1 P_2^{-1}) x = (P_1 R_c P_1^{-1})^{-1} (P_1 Q_c^{\mathrm{T}} P_1^{-1}) b$$

$$x = (P_2 P_1^{-1}) y$$

If you choose **CMSSL_qr_pre_permute** when you call **gen_qr_factor**, the factor routine multiplies each A on the right with $P_1 P_2^{-1}$ by doing a send to rearrange the columns before performing the factorization. (If A has square subgrids, then $P_1 = P_2$, so this permutation is the identity and no send is performed.) The factorization yields

$$Q^{\mathrm{T}} A (P_1 P_2^{-1}) = (P_1 Q_c^{\mathrm{T}} P_1^{-1}) A (P_1 P_2^{-1}) = (P_1 R_c P_2^{-1})$$

Thus, for each right-hand-side vector b in the matrix B, the original system

$$Ax = b$$

or

$$(P_1 Q_c^{\mathrm{T}} P_1^{-1}) A \ (P_1 P_2^{-1} P_2 P_1^{-1}) x = (P_1 Q_c^{\mathrm{T}} P_1^{-1}) b$$

is equivalent to

$$(P_1 R_c P_2^{-1}) (P_2 P_1^{-1}) x = (P_1 Q_c^{\mathrm{T}} P_1^{-1}) b$$

$$(P_1 R_c P_1^{-1}) x = (P_1 Q_c^{\mathrm{T}} P_1^{-1}) b$$

$$x = (P_1 R_c P_1^{-1})^{-1} (P_1 Q_c^{\mathrm{T}} P_1^{-1}) b$$

Copyright © 1994 Thinking Machines Corporation

In this case, the solver routine produces the desired result without a post-permutation. Finally, note that if A is (m x n) with $m > n$, then P_1 is (m x m), and multiplying A by P_1 on the right does not make sense. This is why the **CMSSL_qr_pre_permute** strategy requires A to be square.

5.3.6 Results of the Solver Routines

In the square case ($m = n$),

- The **gen_qr_solve** routine solves the system $AX = B$. Upon successful completion, each B is overwritten with the solution.
- The **gen_qr_solve_tra** routine solves the system $A^{\mathrm{T}}X = B$. Upon successful completion, each B is overwritten with the solution.

In the non-square case ($m > n$),

- The **gen_qr_solve** routine solves the system $Q[R]_{mm}X = B$ (in the no-pivoting case) or $Q[R]_{mm}[P^{-1}]_{mm}X = B$ (in the pivoting case). Upon successful completion, each B is overwritten with the solution to $Q[R]_{mm}X = B$ or $Q[R]_{mm}[P^{-1}]_{mm}X = B$. Depending on your application, you may be interested in only the first n rows of the solution (which contain the least squares solution to $AX = B$), or you may be interested in the entire solution to the larger square system.
- Conceptually, the **gen_qr_solve_tra** routine provides the minimal 2-norm solution to $A^{\mathrm{T}}X = B_n$, where B_n is the first n block cyclic rows of B. To perform this operation, **gen_qr_solve_tra** solves $(Q[R]_{mm})^{\mathrm{T}}X = B_0$ (in the no-pivoting case) or $(Q[R]_{mm}[P^{-1}]_{mm})^{\mathrm{T}}X = B_0$ (in the pivoting case), where B_0 is B with the last $m - n$ block cyclic rows zeroed. Upon successful completion, each B is overwritten with the solution (with minimal 2-norm) to the underdetermined system $A^{\mathrm{T}}X = B_n$.

5.3.7 Numerical Stability

The orthogonalization methods used in the QR factorization have guaranteed stability; there is no "growth factor" as with Gaussian elimination. Even for extremely poorly conditioned matrices, the QR factorization routine with no pivoting produces small residuals.

Copyright © 1994 Thinking Machines Corporation

However, if the matrix to be factored is truly singular, the pivoting option is recommended (see Section 5.3.8).

The *QR* solver performs both a forward solve and a backsolve. The forward solve is the application of a sequence of (block) Householder transformations, and is stable (see reference 1 listed in Section 5.7). The backsolve is triangular; for infomation on its stability, see reference 8.

5.3.8 The Pivoting Option: Working with Ill-Conditioned Systems

To use the *QR* pivoting option, supply the value **CMSSL_column_pivoting** (or **CMSSL_column_pivoting_scale**; see Section 5.3.9) for the *pivoting_strategy* argument when you call **gen_qr_factor**. (See the man page at the end of this section for details about the calling sequence.)

Why Use Pivoting?

Pivoting is useful in the following ways:

- It allows you to determine the column rank of the matrix A accurately. In contrast, when you perform the factorization without pivoting, it is relatively easy to misjudge the column rank of A.
- It gives you more options for working with ill-conditioned matrices.

In the current release, the *QR* factor and solve routines with pivoting and the singular value decomposition routines (described in Chapter 9) are the recommended methods for working with ill-conditioned matrices.

Determining the Column Rank of the Matrix *A*

This section describes the advantage of using pivoting in determining the column rank of the matrix A. Throughout this discussion, a *tiny* number is a number that is tiny *relative to the norm of the matrix A*.

A column of A that is dependent or close to dependent on the previous columns (indicating that A is ill-conditioned) will appear, during one of the elimination steps in the factorization process, as a column of zeros or tiny numbers. If **gen_qr_factor** encounters such a column and you have specified no pivoting, the routine either fails or places a zero or tiny number on the

Copyright © 1994 Thinking Machines Corporation

diagonal of the corresponding column of R. In fact, a zero or tiny number on the diagonal of R *always* means that the corresponding column of A was dependent (or almost dependent, respectively) on the previous columns. Thus, if R contains columns with zeros or tiny numbers on the diagonal, you can assume that A is singular or ill-conditioned.

Suppose one wants to determine the column rank of R (which equals the column rank of A, since Q is orthogonal). When counting the linearly independent columns of R, one strategy might be to discount any column with a zero or tiny number on the diagonal. But this strategy can be misleading. For example, consider the matrix

$$R = \begin{matrix} 1 & 1 & 1 & 1 & 2^{-1/2} \\ & e & e & e & 2^{-1/2} \\ & & e & e & e \\ & & & e & e \\ & & & & e \end{matrix}$$

where e is tiny. The values of e on the diagonal indicate that A (and R) are ill-conditioned. However, if you ignore the columns with e on the diagonal, you conclude that the matrix has column rank 1, whereas in fact, it has column rank 2 (the first and last columns are linearly independent).

In contrast, when you specify pivoting, each time **gen_qr_factor** processes a column, it examines the remaining columns and moves the one with the greatest vector 2-norm forward (to a lower column position) in the matrix. Therefore, columns with zeros or tiny numbers end up in the last column positions of R. In the above example, if you had specified pivoting, R would be

$$R = \begin{matrix} 1 & 2^{-1/2} & 1 & 1 & 1 \\ & 2^{-1/2} & e & e & e \\ & & e & e & e \\ & & & e & e \\ & & & & e \end{matrix}$$

This time, you would discount the last three columns and correctly conclude that the column rank is 2.

Copyright © 1994 Thinking Machines Corporation

It is important to note that if R has no zeros or tiny numbers on the diagonal, you *cannot* safely conclude that A is well-conditioned. For example, consider the matrix

$$R = \begin{matrix} 1 & 1 & 0 & 0 \\ & u & 1 & 0 \\ & & u & 1 \\ & & & u \end{matrix}$$

where e is tiny and $u = e^{1/2}$ (which is "large"). This matrix has no zeros or tiny numbers on the diagonal. However, its condition number is on the order of $1/u^3 = 1/(e^{3/2})$, which is large; thus, the matrix is ill-conditioned.

Strategies for Working with Ill-Conditioned Matrices

In most ill-conditioned problems, the dependent columns occur at the end of the matrix, so that pivoting gains you no special advantage. However, in most cases you do not know ahead of time whether this condition is true for a given matrix. Furthermore, in extremely ill-conditioned cases, **gen_qr_factor** without pivoting may fail altogether because of underflow when processing a dependent column. Therefore, pivoting is a safer strategy when working with matrices that may be ill-conditioned. However, since pivoting also exacts a performance cost, you may want to call the factorization routine without pivoting first, as in the following strategy:

1. Factor without pivoting:
    a. Call **gen_qr_factor** without pivoting.
    b. Call **gen_qr_infinity_norm_inv** to estimate the infinity norm of A^{-1}; call **gen_infinity_norm** to obtain the infinity norm of A; and thus find the condition number of A.
    c. If A is well-conditioned, proceed with the solve routine. If A is ill-conditioned, you may still wish to call the solve routine, bearing in mind that your relative error will be large. Alternatively, try Step 2.
2. Factor with pivoting, if necessary.
    a. Call **gen_qr_factor** with pivoting.

Copyright © 1994 Thinking Machines Corporation

b. Call **gen_qr_extract_diag** to extract the block cyclic diagonal entries of *R*. If there are zeros or tiny numbers at the end of the block cyclic diagonal, *A* is ill-conditioned. (Remember that if there are no zeros or tiny entries at the end of the block cyclic diagonal, you cannot be sure the matrix is well-conditioned.)

c. Change any tiny entries at the end of the block cyclic diagonal to zeros.

d. Call **gen_qr_deposit_diag** to deposit the modified block cyclic diagonal entries back into *R*.

e. Call **gen_qr_r_infinity_norm_inv**. This routine estimates the infinity norm of $(R^*)^{-1}$, where R^* is the upper left corner of *R* formed by discarding any trailing columns of *R* that contain zeros on the block cyclic diagonal. Find the condition number of *R**. If *R** is ill-conditioned, you may wish to use **gen_qr_extract_diag** and **gen_qr_deposit_diag** to change the last block cyclic diagonal entries of *R** to zeros and then repeat this step. When you have finally discarded enough columns to obtain an *R** that is well-conditioned, you will know that you can solve the corresponding portion of your original problem (by discarding some portions of the right-hand side) with confidence.

5.3.9 Scaling

The *pivoting_strategy* argument of **gen_qr_factor** allows you to select scaling as well as pivoting. The values

CMSSL_column_pivoting_scale
CMSSL_no_pivoting_scale

have the same effects as

CMSSL_column_pivoting
CMSSL_no_pivoting

respectively, except that the first two select scaling while the second two do not.

If you select scaling, the **gen_qr_factor** routine uses a scaling factor to eliminate the possibility that $||col||^2$ yields underflow or overflow, where *col* is

Copyright © 1994 Thinking Machines Corporation

a column of A used in the elimination process. In particular, **gen_qr_factor** replaces $(\Sigma a_i^2)^{1/2}$ with $S(\Sigma(a_i/S)^2)^{1/2}$, where S is the scaling factor and a_i are the elements of *col*. The scaling factor S is defined by $(||col||_\infty)^{1/2} = (\max(col))^{1/2}$.

Scaling is not usually necessary; it is required only when $||A||^2$ is close to underflow or overflow, for any matrix A within $\boldsymbol{A}$. (Note that underflow of $||col||^2$ does not cause a problem if *col* is a column with zeros or tiny numbers at the end of the block cyclic diagonal.) Because scaling involves a significant performance cost, you should use it only when necessary.

5.3.10 Saving and Restoring the *QR* State

The *QR* factorization routine generates internal state variables required for computing the solution. These variables are not made available as arrays to user applications because their sizes and contents are CM configuration-dependent. However, it is sometimes desirable to save the internal state to a file for future use. The **save_gen_qr**, **save_gen_qr_fms**, **restore_gen_qr**, and **restore_gen_qr_fms** routines allow you to save and restore the internal *QR* state.

The *QR* routines allow you to have more than one factorization "active" at a time; for example, the sequence of calls

```
setup_X = gen_qr_factor(X, ...)
setup_Y = gen_qr_factor(Y, ...)
call gen_qr_solve(B_X, X, setup_X, ...)
call gen_qr_solve(B_Y, Y, setup_Y, ...)
```

is valid. You may, however, want to use the save and restore routines to carry the internal state over between program runs.

It is not intended that the save and restore routines be used to conserve memory. The state variables are very small compared to the size of the typical matrix A.

Copyright © 1994 Thinking Machines Corporation

NOTE

The save and restore routines do not work with the nodal CMSSL library.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Solving Linear Systems Using Householder Transformations

Given a CM array ***A*** containing one or more instances of a dense m x n matrix A, with $m \geq n$, and a CM array ***B*** containing corresponding right-hand sides B, the **gen_qr_factor** routine uses Householder transformations (with or without column pivoting) to to perform the factorization $A = QR$ (in the no-pivoting case) or $A = QRP^{-1}$ (in the pivoting case). The other routines described below use the QR factors returned by **gen_qr_factor** to solve the linear systems $AX = B$ or $A^{T}X = B$; apply matrices derived from Q, R, and P to arbitrary matrices; provide access to the R factor; estimate the infinity norm of each A^{-1}; and perform related operations.

SYNTAX

Factorization routine:

setup = **gen_qr_factor**	(***A***, *m, n, row_axis, col_axis, nblock, pivoting_strategy, back_solve_strategy, ier*)

Save and restore routines:

save_gen_qr	(*setup, unit, iostat, ier*)
save_gen_qr_fms	(*setup, unit, iostat, ier*)
setup = **restore_gen_qr**	(***A***, *m, n, row_axis, col_axis, nblock, pivoting_strategy, back_solve_strategy, unit, iostat, ier*)
setup = **restore_gen_qr_fms**	(***A***, *m, n, row_axis, col_axis, nblock, pivoting_strategy, back_solve_strategy, unit, iostat, ier*)

Solver routines:

gen_qr_solve	(***B***, ***A***, *setup, nrhs, ier*)
gen_qr_solve_tra	(***B***, ***A***, *setup, nrhs, ier*)

Factor application routines:

gen_qr_apply_q	(***B***, ***A***, *setup, nrhs, ier*)

Copyright © 1994 Thinking Machines Corporation

gen_qr_apply_q_tra (***B***, ***A***, *setup, nrhs, ier*)

gen_qr_apply_r_inv (***B***, ***A***, *setup, nrhs, ier*)

gen_qr_apply_r_inv_tra (***B***, ***A***, *setup, nrhs, ier*)

Get-*R* routine:

gen_qr_get_r (***B***, ***A***, *setup, ier*)

Pivot application routines:

gen_qr_apply_p (***B***, ***A***, *setup, nrhs, ier*)

gen_qr_apply_p_inv (***B***, ***A***, *setup, nrhs, ier*)

Zeroing routine:

gen_qr_zero_rows (***B***, ***A***, *setup, limit, nrhs, ier*)

Diagonal manipulation routines:

gen_qr_extract_diag (***d***, ***A***, *setup, ier*)

gen_qr_deposit_diag (***A***, ***d***, *setup, ier*)

Infinity norm routines:

gen_qr_infinity_norm_inv (***A_norm***, ***A***, *setup, ier*)

gen_qr_r_infinity_norm_inv (***A_norm***, ***A***, *setup, ier*)

Deallocation routine:

deallocate_gen_qr (*setup*)

ARGUMENTS

In the descriptions below, *A* and *B* refer to the active matrices with which the routines work. These matrices may be contained (as the upper left-hand submatrices) in larger matrices within the arrays ***A*** and ***B***, respectively. Details are provided below.

Copyright © 1994 Thinking Machines Corporation

Also, throughout these descriptions, $\overline{Q}$ denotes the conjugate of Q, and the notation $M^{-\mathrm{T}}$ is used for $(M^{-1})^{\mathrm{T}} = (M^{\mathrm{T}})^{-1}$.

setup Scalar integer variable. Setup ID returned by **gen_qr_factor**, **restore_gen_qr**, and **restore_gen_qr_fms**. When you call any of the other *QR* routines, you must supply the value returned by the corresponding **gen_qr_factor**, **restore_gen_qr**, or **restore_gen_qr_fms** call.

B CM array of the same type (real or complex) and precision as ***A***. The instance axes of ***B*** must match those of ***A*** in order of declaration and extents. When you call **gen_qr_get_r**, ***A*** and ***B*** must have the same rank, axis extents, and layout directives. ***B*** must be distinct from ***A***.

Solver, Factor Application, Pivot Application, and Zeroing Routines. When you call one of the *QR* solver, factor application, pivot application, or zeroing routines, ***B*** must contain one or more instances of *B*, where each *B* consists of one or more right-hand-side vectors. The following restrictions hold:

- If each instance *B* within ***B*** consists of only one right-hand-side vector (*nrhs* = 1), you may represent *B* in either of the following ways:
  - It may have rank 2 with number of columns = 1. In this case, each *B* has dimensions *m* x 1 (and may consist of the upper left-hand *m* x 1 elements of a larger matrix). The rows of each *B* must be counted by axis *row_axis* (from the **gen_qr_factor** call); the single column must be counted by axis *col_axis*.
  - It may have rank 1. In this case, each *B* has dimension *m* (and may consist of the first *m* elements of a larger vector). The elements of each *B* must be counted by axis *row_axis* (if *row_axis* < *col_axis*) or by axis (*row_axis* − 1) (if *col_axis* < *row_axis*). For an example, see the Notes section.

Copyright © 1994 Thinking Machines Corporation

- If each B within ***B*** consists of multiple right-hand-side vectors ($nrhs > 1$), then each B has dimensions m x $nrhs$, and may consist of the upper left-hand m x $nrhs$ elements of a larger matrix. The rows and columns of B must be counted by axes *row_axis* and *col_axis*, respectively.

Upon successful completion of **gen_qr_solve** in the square case ($m = n$), each matrix B is overwritten with the solution to $AX = B$.

Upon successful completion of **gen_qr_solve_tra** in the square case ($m = n$), each B is overwritten with the solution to $A^{\mathrm{T}}X=B$.

Upon successful completion of **gen_qr_solve** in the non-square case ($m > n$), each B is overwritten with the solution to $Q[R]_{mm}X = B$ (in the no-pivoting case) or $Q[R]_{mm}[P^{-1}]_{mm}X = B$ (in the pivoting case), where $[R]_{mm}$ and $[P^{-1}]_{mm}$ are defined in the Description section, below. (However, no m x m matrices are actually stored.) Depending on your application, you may be interested in only the first n rows of the solution (which contain the least squares solution to $AX = B$), or you may be interested in the entire solution to the larger square system.

Upon successful completion of **gen_qr_solve_tra** in the non-square case ($m > n$), each B is overwritten with the solution (with minimal 2-norm) to the underdetermined system $A^{\mathrm{T}}X = B_n$, where B_n is the first n block cyclic rows of B. The transpose solver routine solves $(Q[R]_{mm})^{\mathrm{T}}X = B_0$ (in the no-pivoting case) or $(Q[R]_{mm}[P^{-1}]_{mm})^{\mathrm{T}}X = B_0$ (in the pivoting case), where B_0 is B with the last $m - n$ block cyclic rows zeroed. However, no m x m matrices are actually stored.

Upon completion of a factor application routine, each B is overwritten by the specified product: QB, $\overline{Q}B$, $Q^{\mathrm{T}}B$, or $\overline{Q}^{\mathrm{T}}B$ ($m \geq n$); $R^{-1}B$, or $R^{-\mathrm{T}}B$ ($m = n$); $[R]_{mm}{}^{-1}B$ or $[R]_{mm}{}^{-\mathrm{T}}B$ ($m > n$). For the exact definitions of the factors that are applied to B, see the Description section.

Upon completion of a pivot application routine, each B is overwritten by the product PB or $P^{-1}B$ ($m = n$), or $[P]_{mm}B$ or $[P^{-1}]_{mm}B$ ($m > n$), where P is the permutation matrix that corresponds to the pivoting process.

The **gen_qr_zero_rows** routine zeroes the last $m -$ *limit* block cyclic rows of each B within ***B***.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Get-*R* Routine. When you call **gen_qr_get_r**, ***B*** must have the same rank, axis extents, layout directives, data type, and precision as ***A***. ***B*** contains an m x n matrix B corresponding to each A in ***A***. The rows and columns of each B are represented by *row_axis* and *col_axis*, respectively. These axes may have extents greater than m and n, respectively; that is, each B may be contained (as the upper left-hand m x *nrhs* elements) in a larger matrix within ***B***. Upon completion, each B contains the factor R of the corresponding matrix A within ***A***. (R is defined in the Description section.)

d CM array of the same data type and precision as ***A***, containing a vector corresponding to each matrix A in ***A***. The array ***d*** must have either the same rank as ***A*** or rank one less than that of ***A***, as follows:

- If ***d*** has the same rank as ***A***, then the vector elements are counted by axis *row_axis*, which must have extent greater than or equal to n; axis *col_axis* must have extent 1.
- If ***d*** has rank one less than that of ***A***, then axis *col_axis* is omitted. The vector elements are counted by axis *row_axis* (if *row_axis* < *col_axis*) or by axis *row_axis*−1 (if *col_axis* < *row_axis*).

The remaining axes of ***d*** must be instance axes that match the instance axes of ***A*** in extent and order of declaration.

Upon return from **gen_qr_extract_diag**, the first n elements of each vector within ***d*** are the block cyclic diagonal entries of the R factor of the corresponding A within ***A***.

When you call **gen_qr_deposit_diag**, you must supply in the first n elements of each vector within ***d*** the values you wish to deposit into the block cyclic diagonal of the R factor of the corresponding A within ***A***.

A_norm Real CM array with the same rank and precision as ***A***. Axes *row_axis* and *col_axis* must have extent 1. The remaining axes of ***A_norm*** must match the instance axes of ***A*** in order of declaration and extents. Thus, each matrix A embedded in the array ***A*** that you originally supplied to **gen_qr_factor** corresponds to a real number in ***A_norm***.

Copyright © 1994 Thinking Machines Corporation

Upon successful completion of **gen_qr_infinity_norm_inv**, the estimated infinity norm of the *inverse* of each matrix *A* within the original input array ***A*** is placed in the corresponding position of ***A_norm***.

Upon successful completion of **gen_qr_r_infinity_norm_inv**, the estimated infinity norm of the *inverse* of each *R** within ***A*** is placed in the corresponding position of ***A_norm***. (The array ***A*** you supply to **gen_qr_r_infinity_norm_inv** contains the *R* factors of the matrices *A*, returned by **gen_qr_factor**. *R** is the block cyclic upper left corner of *R* formed by discarding any trailing columns of *R* that contain zeros on the block cyclic diagonal.)

A Real or complex CM array of rank greater than or equal to 2. Must be distinct from ***B***.

Factor Routine. When you call **gen_qr_factor**, ***A*** should contain one or more instances of a coefficient matrix *A* to be factored. Each *A* is assumed to be dense with dimensions *m* x *n*, where *m* is greater than or equal to *n*, with rows counted by axis *row_axis* and columns counted by axis *col_axis*. These axes may have extents greater than *m* and *n*, respectively; that is, each *A* may be contained (as the upper left-hand *m* x *n* elements) in a larger matrix within ***A***. Upon successful completion of **gen_qr_factor**, the block cyclic upper triangle of *A* is overwritten by the elements of *R*. The remaining elements of *A* are used internally to reconstruct *Q*.

All Other Routines. When you call any of the other *QR* routines, ***A*** must have the same data type, precision, rank, and shape (axis extents and layout directives, including orderings and weights) as the original ***A*** that was factored. You must also be using the same partition size as when you originally factored ***A***. Supply in ***A*** the *QR* factors returned in ***A*** by **gen_qr_factor**.

m Scalar integer variable. The number of rows in each matrix *A* embedded in ***A***. Also, the number of rows in each right-hand side *B* (or, if each *B* is a single vector, the number of elements in *B*). Must be greater than or equal to *n*.

If you intend to call **gen_qr_infinity_norm_inv**, *m* must equal *n*, since each matrix *A* within ***A*** must be invertible, and therefore square.

Copyright © 1994 Thinking Machines Corporation

n — Scalar integer variable. The number of columns in each matrix *A* embedded in ***A***. Must be less than or equal to *m*.

If you intend to call **gen_qr_infinity_norm_inv**, *m* must equal *n*, since each matrix *A* within ***A*** must be invertible, and therefore square.

row_axis — Scalar integer variable. Identifies the axis of ***A*** that counts the rows of each matrix *A*. The extent of this axis must be at least *m*; *row_axis* must be in the range 1 through the rank of ***A***, inclusive; and *row_axis* and *col_axis* must not be equal. (For definitions of row and column axes, see Chapter 1.)

col_axis — Scalar integer variable. Identifies the axis of ***A*** that counts the columns of each matrix *A*. The extent of this axis must be at least *n*; *col_axis* must be in the range 1 through the rank of ***A***, inclusive; and *row_axis* and *col_axis* must not be equal. (For definitions of row and column axes, see Chapter 1.)

nblock — Scalar integer variable. Blocking factor. The blocking factor you specify when you call **gen_qr_factor** is also used in any subsequent *QR* calls in which you supply the setup ID returned by **gen_qr_factor**. Use these guidelines when choosing an *nblock* value:

- For smaller matrices, *nblock* = 1 may yield the best performance. For most typical applications, *nblock* = 8 is a good choice. An *nblock* value of 16 may yield faster factorization in some cases.
- *nblock* should always be less than or equal to *n*; *nblock* values > *n* use excess time and especially memory.
- For a single right-hand-side vector, the solver routines will most likely be faster with a larger value of *nblock*. On the other hand, the amount of auxiliary storage used is proportional to *nblock*, so if memory is tight, a smaller *nblock* may be a better choice.
- Choose an *nblock* value that is less than or equal to the subgrid lengths in both dimensions. For optimal performance, ensure that the subgrid length in each dimension is a multiple of *nblock*.

Copyright © 1994 Thinking Machines Corporation

pivoting_strategy Scalar integer variable specifying the pivoting strategy to be used. Specify one of the following values:

CMSSL_no_pivoting	No pivoting, no scaling
CMSSL_no_pivoting_scale	No pivoting, scaling
CMSSL_column_pivoting	Column pivoting, no scaling
CMSSL_column_pivoting_scale	Column pivoting, scaling

If you specify column pivoting, then at each pivoting step, the factorization routine searches for the column with the maximum vector 2-norm and moves it to a lower column position. For information about scaling, see the Notes section below.

back_solve_strategy

Scalar integer variable. Specifies the back substitution strategy. A value of **CMSSL_qr_post_permute** is always acceptable; the value **CMSSL_qr_pre_permute** is used to enhance performance in special cases, as described in Section 5.3.3.

A value of **CMSSL_qr_post_permute** indicates that the rows of the solution are to be permuted by the **gen_qr_solve** routine after the backsolve is completed. A value of **CMSSL_qr_pre_permute** specifies that the columns of the matrices A are to be permuted by **gen_qr_factor** prior to the factorization.

CMSSL_qr_pre_permute requires that $m = n$, and that *pivoting_strategy* = **CMSSL_no_pivoting** or **CMSSL_no_pivoting_scale**.

nrhs Scalar integer variable. The number of columns in each instance B within $\boldsymbol{B}$. Must be greater than or equal to 1. If each B is a single vector, supply 1 for the value of *nrhs*.

limit Scalar integer variable. Must be in the range from 0 through m. The **gen_qr_zero_rows** routine zeroes the last $m-limit$ block cyclic rows of each two-dimensional matrix defined by axes *row_axis* and *col_axis* of $\boldsymbol{B}$.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

unit Scalar integer. Valid unit number associated with the file to or from which the *QR* state is to be written or read. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a device or socket with a unit number). The **save_gen_qr** and **restore_gen_qr** calls write and read data using **CMF_CM_ARRAY_TO_FILE_SO** and **CMF_CM_ARRAY_FROM_FILE_SO**, respectively. The **save_gen_qr_fms** and **restore_gen_qr_fms** calls write and read data using **CMF_CM_ARRAY_TO_FILE_FMS** and **CMF_CM_ARRAY_FROM_FILE_FMS**, respectively. You must rewind the file before calling **restore_gen_qr** or **restore_gen_qr_fms**.

iostat Scalar integer variable. Upon return, contains the status of the I/O operation. If *ier* = 0, *iostat* contains the number of bytes written or read. For the meanings of other *iostat* codes (returned if *ier* = −200), refer to the descriptions of **CMF_CM_ARRAY_TO_FILE_SO** (for **save_gen_qr**), **CMF_CM_ARRAY_TO_FILE_FMS** (for **save_gen_qr_fms**), **CMF_CM_ARRAY_FROM_FILE_SO** (for **restore_gen_qr**), and **CM_ARRAY_FROM_FILE_FMS** (for **restore_gen_qr_fms**) in the CM Fortran documentation set.

ier Scalar integer variable. Return code; set to 0 upon successful return.

Values between −1 and −9, inclusive, indicate problems with one or more of the CM arrays containing matrices in any of the *QR* calls:

−1	Invalid array home. The array must be a CM array.
−2	Invalid rank; must be ≥ 2.
−3	Invalid column extent; must be $\geq m$.
−4	Invalid row extent; must be $\geq n$.
−9	Invalid data type; must be real or complex (single- or double-precision).

Values that are multiples of −10 indicate problems with non-array arguments:

−10	System failed to allocate the setup object, *setup*.

Copyright © 1994 Thinking Machines Corporation

−20 *m*, *n*, or *nrhs* is invalid; all must be > 0 and *m* must be greater than or equal to *n*.

−30 *row_axis* or *col_axis* is invalid. 1 ≤ *row_axis, col_axis* ≤ rank (***A***) must be true, and *row_axis* and *col_axis* must not be equal.

−40 *nblock* is invalid. It must be greater than or equal to 1.

−50 *pivoting_strategy* is invalid; must be **CMSSL_column_pivoting**, **CMSSL_no_pivoting**, **CMSSL_column_pivoting_scale**, or **CMSSL_no_pivoting_scale**.

−60 *nrhs* is invalid.

−70 *back_solve_strategy* is invalid; must be **CMSSL_pre_permute** or **CMSSL_post_permute**.

−80 You specified an invalid combination of *pivoting_strategy*, *back_solve_strategy*, and/or *m* not equal to *n*; or you specified *m* not equal to *n* in the factorization call associated with this call to **gen_qr_infinity_norm_inv**.

−100 *setup* is invalid. (You did not supply the value returned by **gen_qr_factor**.)

Values between −102 and −108, inclusive, indicate problems with the consistency of ***A*** or ***B*** in one of the *QR* routines following a factorization call:

−102 The rank of ***A*** or ***B*** is invalid (must be ≥ 2 for ***A*** or ≥ 1 for ***B***), or is inconsistent with the rank of ***A*** in the factorization call.

−105 The extents of the instance axes of ***A*** or ***B*** are inconsistent with those of ***A*** in the factorization call.

−106 ***B*** must have the same layout directives as ***A*** when you call **gen_qr_get_r**.

−108 The data type or precision of ***A*** or ***B*** is inconsistent with that of ***A*** in the factorization call.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

The save and restore routines return the following value if they encounter an I/O error:

−200 I/O error. See the value of *iostat* for more information.

DESCRIPTION

Given a CM array ***A*** containing one or more instances of a dense m x n matrix A, and a CM array ***B*** containing corresponding right-hand sides B, the QR routines perform the operations listed below for each instance.

Factorization routine:

gen_qr_factor Uses Householder transformations, with or without column pivoting, to perform a QR factorization of A. In the no-pivoting case, the factorization is $A = QR$. If you specify pivoting, **gen_qr_factor** computes $A = QRP^{-1}$, where P is the permutation matrix that corresponds to the pivoting process.

In the non-square case ($m > n$), the factorization is defined so that Q is m x m and R is the m x n matrix $[R]_{mm}[I]_{mn}$, where $[R]_{mm}$ is an m x m matrix defined under "Alternate View of the Non-Square Factorization," below, and $[I]_{mn}$ is the m x n matrix consisting of the n x n identity matrix extended by $m-n$ rows of zeroes. However, no m x m matrices are actually stored; Q and R are stored together in an m x n matrix.

Save and restore routines:

save_gen_qr Saves internal information about the QR factors in a file using serial-order I/O.

save_gen_qr_fms Saves internal information about the QR factors in a file using fixed-machine-size I/O.

restore_gen_qr Restores internal information about the QR factors from a file using serial-order I/O.

Copyright © 1994 Thinking Machines Corporation

restore_gen_qr_fms Restores internal information about the QR factors from a file using fixed-machine-size I/O.

Solver routines:

gen_qr_solve Uses the QR factors returned by **gen_qr_factor** to solve $AX = B$. In the non-square case, the first n rows of the B are overwritten with the least squares solution to $AX = B$; the solver routine solves $Q[R]_{mm}X = B$ (in the no-pivoting case) or $Q[R]_{mm}[P^{-1}]_{mm}X = B$ (in the pivoting case), where $[P^{-1}]_{mm}$ is an extension of P^{-1} formed by replacing the upper left n x n elements of the m x m identity matrix with P^{-1}.

gen_qr_solve_tra In the square case, **gen_qr_solve_tra** uses the QR factors returned by **gen_qr_factor** to solve $A^{\mathrm{T}}X = B$. In the non-square case, conceptually, **gen_qr_solve_tra** provides the minimal 2-norm solution to $A^{\mathrm{T}}X = B_n$; B_n is the first n block cyclic rows of B. To perform this operation, **gen_qr_solve_tra** solves $(Q[R]_{mm})^{\mathrm{T}}X = B_0$ (in the no-pivoting case) or $(Q[R]_{mm}[P^{-1}]_{mm})^{\mathrm{T}}X = B_0$ (in the pivoting case), where B_0 is B with the last $m - n$ block cyclic rows zeroed.

Factor application routines: These routines use the factors produced by the QR factorization routine to solve "triangular" systems of the form $RX{=}B$ and $R^{\mathrm{T}}X{=}B$ and "trapezoidal" systems of the form $QX{=}B$ or $Q^{\mathrm{T}}X{=}B$. Because of implementation details, R and Q are not actually triangular and trapezoidal, respectively; see Section 5.3.4. In the non-square case, replace R with $[R]_{mm}$.

gen_qr_apply_q Given the QR factors returned by **gen_qr_factor**, applies Q (or $\overline{Q}$, in the case of complex data) to B.

gen_qr_apply_q_tra Given the QR factors returned by **gen_qr_factor**, applies Q^{T} (or Q^{H}, in the case of complex data) to B. Note that since Q is orthogonal (or unitary, in the complex case), $Q^{\mathrm{T}} = Q^{-1}$ (or, in the complex case, $Q^{\mathrm{H}} = Q^{-1}$).

Copyright © 1994 Thinking Machines Corporation

gen_qr_apply_r_inv Given the QR factors returned by **gen_qr_factor**, applies R^{-1} to B. In the non-square case, this routine applies $[R]_{mm}{}^{-1}$ to B.

gen_qr_apply_r_inv_tra Given the QR factors returned by **gen_qr_factor**, applies $R^{-\mathrm{T}}$ to B. In the non-square case, this routine applies $[R]_{mm}{}^{-\mathrm{T}}$ to B.

Get-R routine:

gen_qr_get_r Given the QR factors returned by **gen_qr_factor**, produces the factor R separately.

Pivot application routines:

gen_qr_apply_p Given the QR factors returned by **gen_qr_factor**, applies P to B. In the non-square case, this routine applies $[P]_{mm}$ to B; $[P]_{mm}$ is an m x m extension of P defined in Section 5.3.4. Use this routine only if you specified pivoting in the associated call to **gen_qr_factor**.

gen_qr_apply_p_inv Given the QR factors returned by **gen_qr_factor**, applies $P^{-1} = P^{\mathrm{T}}$ to B. In the non-square case, this routine applies $[P^{-1}]_{mm} = [P^{-\mathrm{T}}]_{mm} = [P]_{mm}{}^{-1}$ to B. Use this routine only if you specified pivoting in the associated call to **gen_qr_factor**.

Zeroing routine:

gen_qr_zero_rows Zeroes the final rows of B. The rows are counted in block cyclic order.

Diagonal manipulation routines:

gen_qr_extract_diag Given the QR factors returned by **gen_qr_factor**, returns the block cyclic diagonal entries of R.

gen_qr_deposit_diag Given the QR factors returned by **gen_qr_factor**, overwrites the block cyclic diagonal entries of R with values you supply.

Copyright © 1994 Thinking Machines Corporation

Infinity norm routines:

gen_qr_infinity_norm_inv	Given the *QR* factors returned by **gen_qr_factor**, estimates the infinity norm of A^{-1}. Uses the method developed by Hager; see reference 5 listed in Section 5.7. (Available only in the square case.)
gen_qr_r_infinity_norm_inv	Given the *QR* factors returned by **gen_qr_factor**, estimates the infinity norm of $(R^*)^{-1}$, where R^* is the block cyclic upper left corner formed by discarding any trailing columns of R that contain zeros on the block cyclic diagonal.

Deallocation routine:

deallocate_gen_qr	Deallocates the memory required by the above routines.

Memory Allocation and Deallocation. You must call **gen_qr_factor**, **restore_gen_qr**, or **restore_gen_qr_fms** before calling any of the other *QR* routines. The **gen_qr_factor**, **restore_gen_qr**, and **restore_gen_qr_fms** routines allocate storage space and return a setup ID. You must supply this setup ID in subsequent *QR* calls, as long as you are working with the same set of factors. You can follow one call to **gen_qr_factor**, **restore_gen_qr**, or **restore_gen_qr_fms** with multiple calls to the other *QR* routines, thus avoiding the overhead of factoring or restoring the same matrix or matrices repeatedly.

The **deallocate_gen_qr** routine deallocates the memory allocated by the factorization or restore routine and required by the other *QR* routines. Be sure to call **deallocate_gen_qr** when you have finished working with a set of *QR* factors. The **deallocate_gen_qr** routine invalidates the setup ID you specify. Attempts to use a deallocated setup ID result in errors.

You can work with more than one set of *QR* factors at a time by calling **gen_qr_factor**, **restore_gen_qr**, or **restore_gen_qr_fms** more than once without calling **deallocate_gen_qr**. Be sure to supply the correct setup ID in each subsequent *QR* call. When you have finished working with a set of factors, be sure to use **deallocate_gen_qr** to deallocate the associated memory. Repeated calls to **gen_qr_factor**, **restore_gen_qr**, or **restore_gen_qr_fms** without deallocation can cause you to run out of memory.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Factorization Routine. The **gen_qr_factor** routine uses Householder transformations, with or without column pivoting, to perform the factorization

$A = QR$ (no pivoting)
$A = QRP^{-1}$ (with pivoting)

where P is the permutation matrix that corresponds to the pivoting process.

In the non-square case ($m > n$), the factorization is defined so that Q is $m \times m$ and R is the $m \times n$ matrix $[R]_{mm}[I]_{mn}$, where $[R]_{mm}$ is an $m \times m$ matrix defined "Alternate View of the Non-Square Factorization," below, and $[I]_{mn}$ is the $m \times n$ matrix consisting of the $n \times n$ identity matrix extended by $m-n$ rows of zeroes. Thus, we can view the factorization as

$A = Q[R]_{mm}[I]_{mn}$ (no pivoting; $m > n$)
$A = Q[R]_{mm}[P^{-1}]_{mm}[I]_{mn}$ (pivoting; $m > n$)

and $[P^{-1}]_{mm}$ is defined to be the $m \times m$ permutation matrix formed by replacing the upper left $n \times n$ elements of the $m \times m$ identity matrix with P^{-1}. (However, no $m \times m$ matrices are actually stored.) For the derivations of these expressions, see Section 5.3.4; for exact definitions of the QR factors, see "The QR Factors Defined," below.

Upon completion of **gen_qr_factor**, the block cyclic upper triangle of each A in $\boldsymbol{A}$ is overwritten by the elements of R. The remaining elements of A are used internally to reconstruct Q. When you call the get-R routine or a solver, factor application, pivot application, zeroing, diagonal manipulation, or infinity norm routine, you must supply the same $\boldsymbol{A}$ that was returned by **gen_qr_factor**. To obtain the R factor separately, use **gen_qr_get_r**.

Save and Restore Routines. You may save internal information about the QR factors in a file for use in later calls to the other QR routines. To save the QR information, call **save_gen_qr** or **save_gen_qr_fms** after the factorization is complete but before deallocating the storage space. To restore the QR information, rewind the file and call **restore_gen_qr** or **restore_gen_qr_fms**; this call is typically followed by calls to the other QR routines.

Solver Routine. In the square case ($m = n$), given the values returned in $\boldsymbol{A}$ by **gen_qr_factor**, the **gen_qr_solve** routine solves one or more instances of the system

$AX = B$

Copyright © 1994 Thinking Machines Corporation

where A and B are corresponding instances within $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. Upon successful completion of **gen_qr_solve**, each B is overwritten with the solution to $AX = B$.

In the non-square case ($m > n$), the **gen_qr_solve** routine solves one or more instances of the system

$$Q[R]_{mm}X = B \quad \text{(no pivoting)}$$
$$Q[R]_{mm}[P^{-1}]_{mm}X = B \quad \text{(pivoting)}$$

(However, no m x m matrices are actually stored.) Upon successful completion, each B is overwritten with the solution to $Q[R]_{mm}X = B$ or $Q[R]_{mm}[P^{-1}]_{mm}X = B$. Depending on your application, you may be interested in only the first n rows of the solution (which contain the least squares solution to $AX = B$), or you may be interested in the entire solution to the larger square system.

In the discussion that follows, replace Q^{T} with Q^{H} in the complex case.

Steps Performed by the Solver Routine: No-Pivoting Case. In the square no-pivoting case, the solver routine solves $QRX = B$, which is equivalent to $X = R^{-1}Q^{\mathrm{T}}B$. In the non-square no-pivoting case, the solver routine solves $Q[R]_{mm}X = B$, which is equivalent to $X = [R]_{mm}{}^{-1}Q^{\mathrm{T}}B$. Therefore, **gen_qr_solve** performs the following steps:

1. Applies Q^{T} to B.
2. Applies R^{-1} (square case) or $[R]_{mm}{}^{-1}$ (non-square case) to $Q^{\mathrm{T}}B$.

To perform these steps yourself, you would

1. Call **gen_qr_apply_q_tra** to apply each Q^{T} to the corresponding right-hand side, B.
2. Call **gen_qr_apply_r_inv** to apply R^{-1} or $[R]_{mm}{}^{-1}$ to the result from Step 1.

Steps Performed by the Solver Routine: Pivoting Case. In the square pivoting case, the solver routine solves $QRP^{-1}X = B$, which is equivalent to $X = PR^{-1}Q^{\mathrm{T}}B$. In the non-square no-pivoting case, the solver routine solves $Q[R]_{mm}[P^{-1}]_{mm}X = B$, which is equivalent to $X = [P]_{mm}[R]_{mm}{}^{-1}Q^{\mathrm{T}}B$. Therefore, **gen_qr_solve** performs the following steps:

1. Applies Q^{T} to B.
2. Applies R^{-1} (square case) or $[R]_{mm}{}^{-1}$ (non-square case) to $Q^{\mathrm{T}}B$.

Copyright © 1994 Thinking Machines Corporation

3. Applies P (square case) or $[P]_{mm}$ (non-square case) to the result from Step 2.

To perform these steps yourself, you would

1. Call **gen_qr_apply_q_tra** to apply each Q^{T} to the corresponding right-hand side, B.
2. Call **gen_qr_apply_r_inv** to apply R^{-1} or $[R]_{mm}{}^{-1}$ to the result from Step 1.
3. Call **gen_qr_apply_p** to apply P or $[P]_{mm}$ to the result from Step 2.

Transpose Solver Routine. In the square case ($m = n$), given the values returned in $\boldsymbol{A}$ by **gen_qr_factor**, the **gen_qr_solve_tra** routine solves one or more instances of the system

$$A^{\mathrm{T}}X = B$$

where A and B are corresponding instances within $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. Upon successful completion of **gen_qr_solve_tra**, each B is overwritten with the solution to $A^{\mathrm{T}}X = B$.

Conceptually, in the non-square case, the **gen_qr_solve_tra** routine provides the minimal 2-norm solution to $A^{\mathrm{T}}X = B_n$, where B_n is the first n block cyclic rows of B. To perform this operation, **gen_qr_solve_tra** solves

$$(Q[R]_{mm})^{\mathrm{T}}X = B_0 \qquad \text{(no pivoting)}$$
$$(Q[R]_{mm}[P^{-1}]_{mm})^{\mathrm{T}}X = B_0 \quad \text{(pivoting)}$$

where B_0 is B with the last $m - n$ block cyclic rows zeroed. Upon successful completion, each B is overwritten with the solution (with minimal 2-norm) to the underdetermined system $A^{\mathrm{T}}X = B_n$.

Steps Performed by the Transpose Solver Routine: No-Pivoting Case. In the square no-pivoting case, the transpose solver routine solves $(QR)^{\mathrm{T}}X = B_0$, which is equivalent to $X = QR^{-\mathrm{T}}B_0$. In the non-square no-pivoting case, the transpose solver routine solves $(Q[R]_{mm})^{\mathrm{T}}X = B_0$, which is equivalent to $X = Q[R]_{mm}{}^{-\mathrm{T}}B_0$. Therefore, **gen_qr_solve** performs the following steps:

1. Applies $R^{-\mathrm{T}}$ (square case) or $[R]_{mm}{}^{-\mathrm{T}}$ (non-square case) to B.
2. If $m > n$, zeros the last $m - n$ block cyclic rows of the result from Step 1.
3. Applies Q to the result.

Copyright © 1994 Thinking Machines Corporation

To perform these steps yourself, you would

1. Call **gen_qr_apply_r_inv** to apply each R^{-T} or $[R]_{mm}{}^{-T}$ to the corresponding right-hand side, B.
2. If $m > n$, call **gen_qr_zero_rows** to zero the last $m - n$ block cyclic rows of each two-dimensional matrix defined by *row_axis* and *col_axis* of the result from Step 1. (Note: Step 2 is not required if the last $m - n$ block cyclic rows of each B were set to zero prior to Step 1.)
3. Call **gen_qr_apply_q** to apply Q to the result.

Step 2 is required so that inactive data in the last $m - n$ rows of the right-hand-sides B does not affect the solution. This zeroing is required only when you are solving $A^T X = B$, not when you are solving $AX = B$.

Steps Performed by the Transpose Solver Routine: Pivoting Case. In the square pivoting case, the transpose solver routine solves $(QRP^{-1})^T X = B_0$, which is equivalent to $X = QR^{-T}P^{-1}B_0$. In the non-square pivoting case, the transpose solver routine solves $(Q[R]_{mm}[P^{-1}]_{mm})^T X = B_0$, which is equivalent to $X = Q[R]_{mm}{}^{-T}[P^{-1}]_{mm}B_0$. Therefore, **gen_qr_solve_tra** performs the following steps:

1. Applies P^{-1} (square case) or $[P^{-1}]_{mm}$ (non-square case) to B.
2. Applies R^{-T} (square case) or $[R]_{mm}{}^{-T}$ (non-square case) to the result from Step 1.
3. If $m > n$, zeroes the last $m - n$ rows of the result from Step 2.
4. Applies Q to the result.

To perform these steps yourself, you would

1. Call **gen_qr_apply_p_inv** to apply each P^{-1} or $[P^{-1}]_{mm}$ to the corresponding right-hand side, B.
2. Call **gen_qr_apply_r_inv_tra** to apply R^{-T} or $[R]_{mm}{}^{-T}$ to the result from Step 1.
3. If $m > n$, call **gen_qr_zero_rows** to zero the last $m - n$ block cyclic rows of each two-dimensional matrix defined by *row_axis* and *col_axis* of the result from Step 2. (Note: Step 3 is not required if the last $m - n$ block cyclic rows of each B were set to zero prior to Step 1.)

Copyright © 1994 Thinking Machines Corporation

4. Call **gen_qr_apply_q** to apply Q to the result.

Factor Application Routines. The **gen_qr_apply_q**, **gen_qr_apply_q_tra**, **gen_qr_apply_r_inv**, and **gen_qr_apply_r_inv_tra** routines allow you to apply matrices derived from the QR factors to arbitrary matrices or vectors B contained in $\boldsymbol{B}$. Upon completion of the routine, each B in $\boldsymbol{B}$ is overwritten with the specified product

The matrices Q and Q^{T} (or Q^{H}, in the complex case) applied by **gen_qr_apply_q** and **gen_qr_apply_q_tra** are given by the definitions of Q under "The QR Factors Defined," below. The inverses are true inverses; that is (in the real case), $Q^{\mathrm{T}}Q = QQ^{\mathrm{T}} = I$. The inverses are also true in the block cyclic space; that is, ${Q_c}^{\mathrm{T}}Q_c = Q_c{Q_c}^{\mathrm{T}} = I$. In the complex case, replace Q^{T} by Q^{H} in these equations.

In the square case, the **gen_qr_apply_r_inv** and **gen_qr_apply_r_inv_tra** routines apply R^{-1} and $R^{-\mathrm{T}}$ as derived from the definitions of R under "The QR Factors Defined." The inverses are true inverses in both the original space and the block cyclic space.

In the non-square case, the **gen_qr_apply_r_inv** and **gen_qr_apply_r_inv_tra** routines apply routines apply ${[R]_{mm}}^{-1}$ and ${[R]_{mm}}^{-\mathrm{T}}$, respectively, to the arbitrary matrices you supply (although no m x m matrices are actually stored). The inverses are true inverses in both the original space and the block cyclic space.

To apply R to an arbitrary matrix or vector B, use the **gen_qr_get_r** routine to obtain R, and then perform the multiplication explicitly. To apply R^{T} to an arbitrary matrix or vector B, either transpose R to obtain R^{T}, or use the fact that $R^{\mathrm{T}}B = (B^{\mathrm{T}}R)^{\mathrm{T}}$; thus, apply B^{T} to R and transpose the result. If B is a vector, transposing B and $B^{\mathrm{T}}R$ (which is also a vector) is much less costly than transposing R would be.

In most cases, you should use a solver routine, rather than using the factor application routines separately, to solve $AX = B$ or $A^{\mathrm{T}}X = B$. Using the factor application calls may require extra communication because of alignment and permutations required internally for B and X.

Get-*R* Routine. The **gen_qr_get_r** routine provides access to the R factor of each matrix A. Upon completion, each m x n instance B within $\boldsymbol{B}$ is overwritten by the factor R of the corresponding coefficient matrix A within $\boldsymbol{A}$. (Note that R is a block cyclic upper triangle.)

Pivot Application Routines. The **gen_qr_apply_p** and **gen_qr_apply_p_inv** routines allow you to apply the permutation matrix P that corresponds to the pivoting process, and its transpose $P^{\mathrm{T}} = P^{-1}$, to arbitrary matrices or vectors B contained in $\boldsymbol{B}$.

Copyright © 1994 Thinking Machines Corporation

In the non-square case, **gen_qr_apply_p** and **gen_qr_apply_p_inv** apply $[P]_{mm}$ and $[P^{-1}]_{mm}$, respectively, to the arbitrary matrices you supply (although no m x m matrices are actually stored). Upon successful completion, each B within ***B*** is overwritten by the specified product. These routines are useful if you want to perform separately the permutations that the solver and transpose solver routines perform when pivoting is specified, as described above. Use these routines only if you specified pivoting in the associated call to **gen_qr_factor**.

Zeroing Routine. The **gen_qr_zero_rows** routine zeroes the last $m - limit$ block cyclic rows of each two-dimensional matrix defined by *row_axis* and *col_axis* of ***B***. This routine is useful if you want to perform separately the zeroing that the transpose solver routine performs, as described above.

Extract and Deposit Diagonal Routines. The **gen_qr_extract_diag** routine returns in ***d*** the block cyclic diagonal entries of the factor R of each matrix A within ***A***. The **gen_qr_deposit_diag** routine overwrites the block cyclic diagonal entries of each R with values you supply in ***d***. These routines are useful in working with matrices that may be ill-conditioned.

Infinity Norm Routines. Given the QR factors returned by **gen_qr_factor**, the **gen_qr_infinity_norm_inv** routine estimates the infinity norm of each matrix A^{-1}. Upon successful completion of **gen_qr_infinity_norm_inv**, the infinity norm of each A^{-1} is placed in the position of ***A_norm*** corresponding to the position of A in the original ***A*** you supplied to **gen_qr_factor**.

The **gen_qr_r_infinity_norm_inv** routine estimates the infinity norm of each $(R^*)^{-1}$, where R^* is the block cyclic upper left corner of R formed by discarding any trailing columns of R that contain zeros on the block cyclic diagonal. This routine is useful in working with matrices that may be ill-conditioned. Upon successful completion of **gen_qr_r_infinity_norm_inv**, the infinity norm of each $(R^*)^{-1}$ is placed in the position of ***A_norm*** corresponding to the original matrix A of which R is a factor.

The infinity norm of a matrix M, denoted here by $|| M ||_\infty$, is defined by

$$|| M ||_\infty = \max_{|| x ||_\infty = 1} || Mx ||_\infty$$

Copyright © 1994 Thinking Machines Corporation

where the infinity norm of a vector, $|| x ||_\infty$, is defined as the maximum of the absolute values of the vector components:

$$|| x ||_\infty = \max_i | x_i |$$

The infinity-norm condition number of a matrix M is equal to the product of $||M||_\infty$ and $||M^{-1}||_\infty$.

The *QR* Factors Defined. The **gen_qr_factor** routine actually factors a block cyclic permutation, A_c, of each matrix A that you supply in $\boldsymbol{A}$. The matrix A_c is defined by

$$A_c = {P_1}^{-1} A P_2$$

where P_1 and P_2 are the permutations giving the correspondence between standard and block cyclic row order and column order, respectively, for A. (These matrices depend on the array size and layout, the partition size, and the blocking factor you supply.) The matrix A_c is factored into

$A_c = Q_c R_c {P_c}^{-1}$ (factorization with pivoting)
$A_c = Q_c R_c$ (factorization without pivoting)

Q_c is orthogonal (or unitary, in the complex case); R_c is upper triangular. In the non-square case, R_c is m x n and is zero except for (i, j) where $i \leq j \leq n$. P_c is the permutation matrix resulting from the pivoting process. We define the Q, R, and P^{-1} factors of A as follows:

$$Q = P_1 Q_c {P_1}^{-1}$$

$R = P_1 R_c {P_2}^{-1}$ ($m \geq n$, post-permutation; pivoting or no pivoting)
$R = P_1 R_c {P_1}^{-1}$ ($m = n$, pre-permutation, no pivoting)

$$P^{-1} = P_2 {P_c}^{-1} {P_2}^{-1}$$

For the derivations of the definitions of Q, R, and P^{-1}, see Section 5.3.4.

Alternate View of the Non-Square Factorization. In the non-square case, we define $[R_c]_{mm}$ to be the m x m extension of R_c shown below.

Copyright © 1994 Thinking Machines Corporation

$$\begin{array}{c} \begin{array}{cc} n & m-n \end{array} \\ \begin{array}{c} m \\ \end{array} \left[\begin{array}{c|c} R_c & \begin{array}{c} 0 \\ \hline \mathrm{I} \end{array} \end{array}\right] \begin{array}{l} n \\ m-n \end{array} \end{array}$$

Note that by the structure of this matrix, we have $R_c = [R_c]_{mm}[I]_{mn}$, where $[I]_{mn}$ is the m x n matrix consisting of the n x n identity matrix extended by $m-n$ rows of zeroes.

We also define the m x m matrix $[P_2]_{mm}$ to be the m x m permutation matrix formed by replacing the upper left n x n elements of the m x m identity matrix with P_2. Note that $[P_2{}^{-1}]_{mm} = [P_2]_{mm}{}^{-1}$. Finally, we define the m x m matrix

$$[R]_{mm} = P_1[R_c]_{mm}[P_2{}^{-1}]_{mm}.$$

In the non-square case, we have

$$\begin{aligned} R &= P_1 R_c P_2{}^{-1} \\ &= P_1[R_c]_{mm}[I]_{mn}P_2{}^{-1} \\ &= P_1[R_c]_{mm}[P_2{}^{-1}]_{mm}[I]_{mn} \\ &= [R]_{mm}[I]_{mn}. \end{aligned}$$

Thus, in the non-square no-pivoting case, we can view the factorization of A as

$$A = Q[R]_{mm}[I]_{mn}.$$

In the non-square pivoting case, the factorization can be expressed as

$$A = Q[R]_{mm}[P^{-1}]_{mm}[I]_{mn}.$$

For a derivation of this expression, see Section 5.3.4.

NOTES

No Nodal Library Support for Save and Restore Routines. The save and restore routines do not work with the nodal CMSSL library.

NaNs and Infinities. As mentioned above, the matrices A and B may be contained (as the upper left-hand submatrices) in larger matrices within the arrays $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. In this case, if there are NaNs or infinities in the larger matrix outside of A or B, it is possible that other locations outside of A or B could become NaNs or infinities as well.

Copyright © 1994 Thinking Machines Corporation

Distinct Variables. The input CM arrays ***A*** and ***B*** must be distinct variables.

Include the CMSSL Header File. The **gen_qr_factor** routine uses symbolic constants. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls this routine. This file declares the types of the CMSSL functions and symbolic constants.

Saving and Restoring the *QR* State. If you want to save the internal state in one program run and restore it in a different run, you must save the array of factored matrices in a file in addition to saving the internal state using **save_gen_qr** or **save_gen_qr_fms**. When you read the array back into memory prior to restoring the internal state, you must use the same partition size as when you originally performed the factorization; and the restored array must have exactly the same shape (axis extents and layout directives, including orderings and weights) as when you saved it.

Nondegeneracy Required. Each matrix A within ***A*** must have a column space of rank n when you call one of the *QR* solver routines without pivoting.

Rank of *B*. The following example illustrates the options for defining the rank of ***B***. Suppose ***A***, n, m, *row_axis*, and *col_axis* are defined as follows:

A (5, 10, 5)
$m = n = 5$
row_axis = 1
col_axis = 3

and each B in ***B*** is a single vector. You may define ***B*** in either of the two following (equivalent) ways:

B (5, 10, 1)
B (5, 10)

On the other hand, if you define

A (5, 10, 5)
$m = n = 5$
row_axis = 3
col_axis = 1

Copyright © 1994 Thinking Machines Corporation

then the possibilities for ***B*** are as follows:

B (1, 10, 5)
B (10, 5)

Performance. Performance improves for larger subgrid sizes (and therefore depends upon the layout of ***A***). For information on subgrids, refer to the CM Fortran documentation set.

To optimize performance, follow these guidelines:

- Lay out ***A*** so that the subgrid sizes along axes *row_axis* and *col_axis* differ from one another by no more than a factor of 4 or 5.
- Use axis extents exactly equal to m x n for the matrices A and m x *nrhs* for the matrices B. Use the same processing element layout for the arrays ***A*** and ***B***.
- For smaller matrices, $nblock = 1$ may yield the best performance. For most typical applications, $nblock = 8$ is a good choice. An *nblock* value of 16 may yield faster factorization in some cases.
- *nblock* should always be less than or equal to n; *nblock* values $> n$ use excess time and especially memory.
- For a single right-hand-side vector, the solver routines will most likely be faster with a larger value of *nblock*. On the other hand, the amount of auxiliary storage used is proportional to *nblock*, so if memory is tight, a smaller *nblock* may be a better choice.
- Choose an *nblock* value that is less than or equal to the subgrid lengths in both dimensions. For optimal performance, ensure that the subgrid length in each dimension is a multiple of *nblock*.

Scaling. The *pivoting_strategy* values **CMSSL_column_pivoting_scale** and **CMSSL_no_pivoting_scale** have the same effects as **CMSSL_column_pivoting** and **CMSSL_no_pivoting**, respectively, except that the first two select scaling while the second two do not.

If you select scaling, **gen_lu_factor** uses a scaling factor to eliminate the possibility that $||col||^2$ yields underflow or overflow, where *col* is a column of A used in the elimination process. In particular, **gen_lu_factor** replaces $(\Sigma a_i^2)^{1/2}$ with

Copyright © 1994 Thinking Machines Corporation

$S(\Sigma(a_i/S)^2)^{1/2}$, where S is the scaling factor and a_i are the elements of *col*. The scaling factor S is defined by $(||col||_\infty)^{1/2} = (\max(col))^{1/2}$.

Scaling is not usually necessary; it is required only when $||A||^2$ is close to underflow or overflow, for any matrix A within $\boldsymbol{A}$. (Note that underflow of $||col||^2$ does not cause a problem if *col* is a column with zeros or tiny numbers at the end of the block cyclic diagonal.) Because scaling involves a significant performance cost, you should use it only when necessary.

Numerical Complexity. If the matrices A have dimensions (m x n), the matrices B have *nrhs* right-hand sides, and I is the number of instances (the product of all axis extents except axes *row_axis* and *col_axis*), then:

- The *QR* factorization routine requires approximately $2n^2(m - n/3)I$ floating-point operations for real operands and $8n^2(m - n/3)I$ floating-point operations for complex operands.
- The *QR* solver routines require approximately *nrhs* * $n(4m - n)I$ floating-point operations for real operands and 4*nrhs* * $n(4m - n)I$ floating-point operations for complex operands.

Performance Cost of Pivoting. The performance cost of pivoting is between 10% and 50%, depending on the matrix size; the cost is more for smaller matrices. The cost is approximately the same whether or not you specify scaling. These guidelines are closely tied to the current implementation, and may change in future releases.

EXAMPLES

Sample CM Fortran code that uses the routines described above can be found online in the subdirectories

```
householder/cmf/
```

and

```
infinity-norm/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

5.4 Matrix Inversion and the Gauss-Jordan System Solver

The matrix inversion routine, **gen_gj_invert**, and the Gauss-Jordan solver routine, **gen_gj_solve**, use a variant of the Gauss-Jordan algorithm.

The Gauss-Jordan algorithm requires pivoting if the system is not symmetric positive definite. The **gen_gj_invert** and **gen_gj_solve** routines support two pivoting strategies (partial and total pivoting), described in the context of the inversion routine, below.

5.4.1 Matrix Inversion

Conceptually, the inversion procedure progressively transforms the original matrix A into the identity matrix, I, while progressively transforming the identity matrix into the solution — the inverse of the original matrix, A^{-1}. Figure 18 shows a simplified view of this process. It ignores the details that are introduced by permuting rows and columns and by inverting A in place.

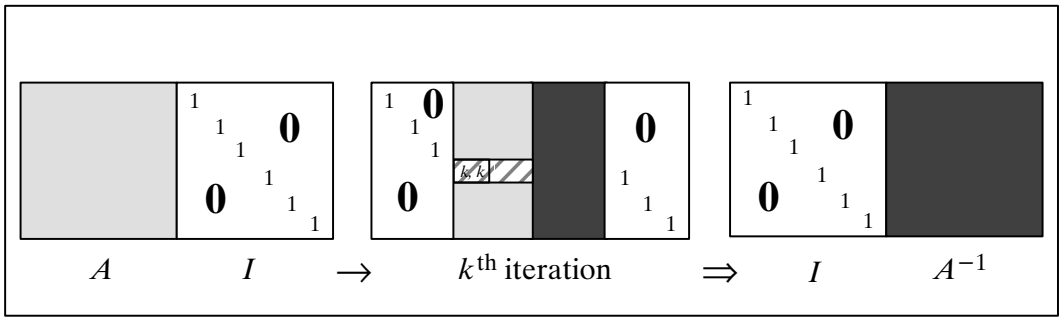


Figure 18. Matrix A becomes I while I becomes A^{-1}.

The pivoting strategy you specify when you call the inversion routine determines the size of the search space for the pivot, as follows:

- If you choose partial pivoting, the pivot element is chosen from the pivot row, and columns are (in effect) permuted. (This is a variant of the conventional partial pivoting method, in which the pivot element is chosen from the pivot column.)

Copyright © 1994 Thinking Machines Corporation

- If you choose total pivoting, the pivot element is chosen from a submatrix and both rows and columns are permuted.

These strategies are illustrated in the next two figures. As in Figure 18, row and column permutations and in-place inversion are ignored.

With partial pivoting, the pivot search is conducted along the pivot row. Figure 19 shows the k^{th} iteration. The pivot search is conducted along the k^{th} row; previous iterations have begun to replace the principal diagonal of A with 1s and successive columns with 0s. The partial pivot search determines the maximum value for row k.

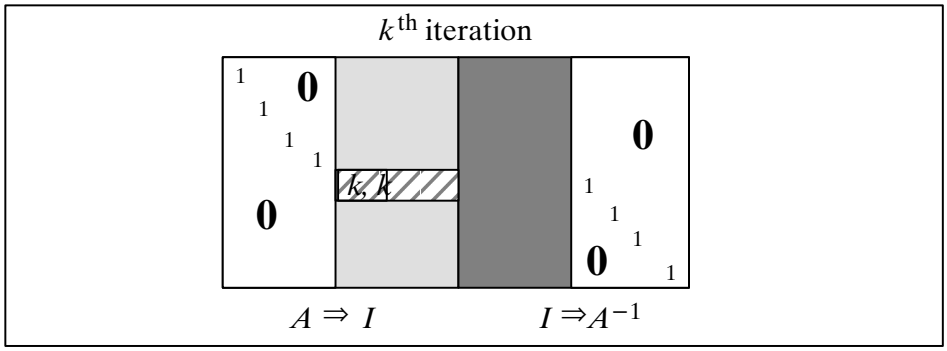


Figure 19. Partial pivoting searches pivot row of A.

With total pivoting, the pivot search is conducted within the submatrix below and to the right of the pivot element, inclusive. Figure 20 illustrates this case.

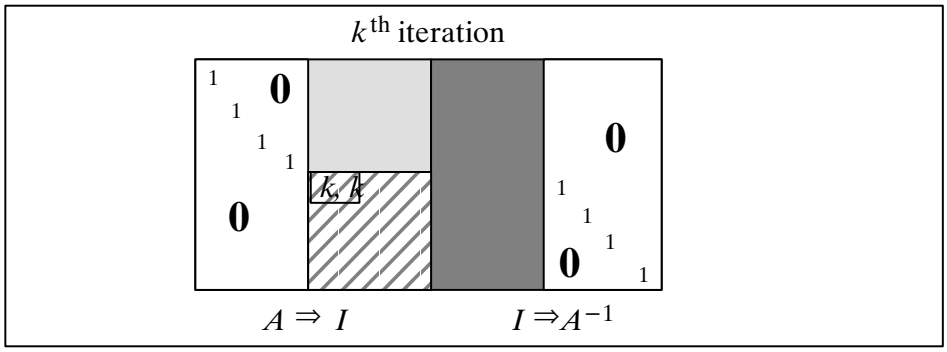


Figure 20. Total pivoting searches lower right-hand submatrix of A.

Copyright © 1994 Thinking Machines Corporation

The total pivoting strategy is numerically more stable but slower than the partial pivoting strategy. For an explanation of the difference in stability, see the works by Golub and Van Loan and by Wilkinson listed in Section 5.7.

At each pivoting iteration, this variant of the algorithm subtracts multiples of the pivot row from the rows *above* as well as from the rows below the pivot row. As a result, the upper triangular matrix is brought to zero along with the lower triangular matrix. This method is different from the Gaussian elimination method, which subtracts multiples of the pivot row from only the rows below it, and thus does not zero the upper triangular matrix. Note that the original matrix is never actually replaced by the identity matrix; the space that would otherwise be "wasted" by 1s and 0s is filled with the accumulated inverse solution. The inversion result is thus efficiently returned in place.

5.4.2 The Gauss-Jordan Solver

The **gen_gj_solve** routine uses the same algorithm and pivoting strategies as the **gen_gj_invert** routine.

5.4.3 Stability and Performance

The variant of the Gauss-Jordan algorithm implemented in CMSSL has been shown to be conditionally stable in the following sense: its residual is about as small as the residual from standard Gaussian elimination with pivoting (where the Gaussian elimination pivot is chosen from the pivot column), as long as the matrix is well conditioned and pivot growth is moderate. For ill-conditioned matrices, this variant fails about as often as Gaussian elimination. For further details, see reference 15 listed in Section 5.7.

If the system of equations is known to be poorly conditioned or the condition of the system is unknown, the *LU* routines are recommended with respect to stability. The *LU* factor and solve routines may yield better performance than **gen_gj_solve**; and using the *LU* factor and solve routines to solve $AX = I$ yields significantly better performance than using **gen_gj_invert** to invert a matrix.

The **CMSSL_total_pivoting** method of pivoting is more numerically stable, but slower than the **CMSSL_partial_pivoting** method.

Copyright © 1994 Thinking Machines Corporation

Matrix Inversion

This function inverts a matrix in place, using a variant of the Gauss-Jordan algorithm.

SYNTAX

pivot_min = **gen_gj_invert** (***A***, *size, pivoting_strategy, ier*)

ARGUMENTS

A	2-dimensional CM array of type real or complex. Contains, and may be larger than, the square matrix to be inverted. Must be of size (*size*, *size*) or larger. Upon successful completion, the data in the upper left-hand (*size*, *size*) area of ***A*** is overwritten with the inverted matrix.
size	Scalar integer greater than 0. The number of rows (or columns) in the matrix to be inverted.
pivoting_strategy	Scalar integer representing the pivoting strategy used. Value must be one of the following symbolic constants (or integer equivalent): **CMSSL_partial_pivoting** (0) Modified partial pivoting. The pivot is chosen from the pivot row; columns are, in effect, permuted. **CMSSL_total_pivoting** (1) Conventional total pivoting, where the pivot is chosen from the submatrix below and to the right of the pivot element; both columns and rows are permuted.
ier	Error code. Scalar integer variable set to 0 if the routine succeeds, and to 1 otherwise. A value of 1 indicates either that one or more arguments were incorrect (for example, *size* = 0), or that a floating-point exception occurred (indicating that the matrix is singular or ill-conditioned).

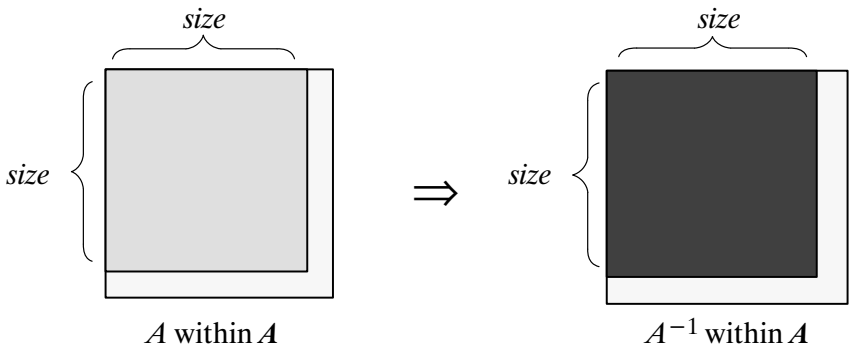
Copyright © 1994 Thinking Machines Corporation

RETURNED VALUE

pivot_min Real double-precision scalar variable. The magnitude of the smallest pivot used. A very small pivot is evidence that the matrix is close to singular (non-invertible). If an error or a floating-point exception occurs, the routine returns a double-precision zero and sets *ier* to 1.

DESCRIPTION

The **gen_gj_invert** routine inverts a (*size* x *size*) real or complex matrix in place, using a rehabilitated Gauss-Jordan algorithm. If the matrix A is smaller than its containing array, $\boldsymbol{A}$, then the remainder of the values in $\boldsymbol{A}$ are left untouched, as shown below.



Matrix A is inverted; the rest of $\boldsymbol{A}$ is unchanged.

NOTES

Include the CMSSL Header File. The matrix inversion routine is a function; it returns the double-precision value *pivot_min*. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of the main program file. This file declares the type of the CMSSL functions and symbolic constants.

Numerical Complexity. Given a matrix with dimensions (n x n), the number of floating-point operations is $2n^3$ for real operands and $8n^3$ for complex operands.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the matrix inversion routine can be found online in the subdirectory

```
invert-and-solve/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

Gauss-Jordan System Solver

Given two matrices, A and B, where B contains one or more right-hand-side vectors, this routine solves a system of linear equations $AX = B$ and overwrites B with the solution $X = A^{-1}B$. A numerically well-behaved variant of the Gauss-Jordan algorithm is used.

SYNTAX

pivot_min = **gen_gj_solve** (***A***, ***B***, *size, nrhs, pivoting_strategy, ier*)

ARGUMENTS

A Two-dimensional CM array of type real or complex. Must be distinct from ***B***. Contains, and may be larger than, the (*size* x *size*) square matrix of coefficients, *A*. Must be of shape (*size*, *size*) or larger.

B One- or two-dimensional CM array of the same type as ***A***. Must be distinct from ***A***. Contains, and may be larger than, the (*size* x *nrhs*) matrix *B* that contains the right-hand-side vectors ($b_1 \ldots b_{\text{nrhs}}$). If there are multiple right-hand sides, ***B*** must be of shape (*size*, *nrhs*) or larger. However, if *nrhs* = 1, then ***B*** can be either a vector of length greater than or equal to *size*, or a matrix of shape (*size*, 1) or larger. Upon successful return, ***B*** is overwritten by the solutions.

size Scalar integer variable. The number of rows (and columns) in the matrix *A*.

nrhs Scalar integer variable. The number of right-hand-side vectors in the matrix *B*. Must be less than or equal to the smaller dimension of ***A***.

pivoting_strategy Scalar integer variable representing the pivoting strategy used. Value must be one of the following symbolic constants (or integer equivalent):

CMSSL_partial_pivoting (0)
Modified partial pivoting. The pivot is chosen from the pivot row; columns are, in effect, permuted.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

CMSSL_total_pivoting (1)
Conventional total pivoting, where the pivot is chosen from the submatrix below and to the right of the pivot element; both columns and rows are permuted.

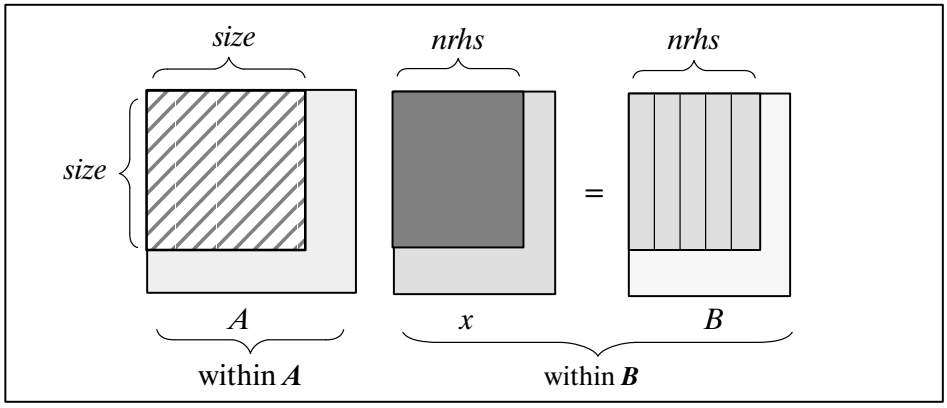
ier Error code. Scalar integer variable set to 0 if the routine succeeds, and to 1 otherwise.

RETURNED VALUE

pivot_min Real double-precision scalar variable. The magnitude of the smallest pivot used. A very small pivot is evidence that the matrix is close to singular. If an error or a floating-point exception occurs, the routine returns a double-precision zero and sets *ier* to 1.

DESCRIPTION

Given a matrix A of shape (*size*, *size*) contained within ***A***, and a second matrix B that contains *nrhs* right-hand-side vectors, $b_1 \ldots b_{nrhs}$ and is contained in ***B***, this function solves for X in $AX = B$ and overwrites B with X, as shown below. Matrix A is left untouched.



This operation is equivalent to performing *nrhs* column solves on B. That is, within B, each column b is replaced by $(A^{-1}\, b)$. Note that while formally $X = A^{-1}B$, as implemented, this routine does not perform an explicit multiplication by A^{-1}.

Copyright © 1994 Thinking Machines Corporation

NOTES

Include the CMSSL Header File. The Gauss-Jordan system solver routine is a function; it returns the double-precision value *pivot_min*. Therefore, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of the main program file. This file declares the type of the CMSSL functions and symbolic constants.

Distinct Variables. The input CM arrays must be distinct variables.

Numerical Complexity. Given an $\boldsymbol{A}$ with dimensions (n x n), and $\boldsymbol{B}$ with dimensions (n x r), the number of floating-point operations is $(2/3)\,n^3 + 2n^2 r$ for real operands and $(8/3)\,n^3 + 8n^2 r$ for complex operands.

As an artifact of the implementation, this linear system solver routine copies $\boldsymbol{A}$ and $\boldsymbol{B}$ into a temporary array with dimensions (*size* x [*size* + *nrhs*]).

EXAMPLES

Sample CM Fortran code that uses the Gauss-Jordan system solver can be found on-line in the subdirectory

invert-and-solve/cmf/

of a CMSSL examples directory whose location is site-specific.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

5.5 Gaussian Elimination with External Storage

The routines described in this section solve a linear system of equations $AX=B$ where A is a square real or complex matrix that is too large to fit into core memory. This operation has three phases:

1. The setup routine, **gen_lu_setup_ext**, assembles the matrix using data you supply through reverse communication, and stores the matrix externally. The routine also assigns values to the setup array, *setup*, and saves *setup* in a file on the partition manager.

2. The factor routine, **gen_lu_factor_ext**, which should be called in a separate program, reads in the setup array and overwrites the matrix with its LU factors.

3. Subsequently, possibly in the same program as the factorization call, the solve routine, **gen_lu_solve_ext**, solves $AX=B$ for a specified number of right-hand sides.

The method used for reducing A to triangular form is block Gaussian elimination with full column pivoting.

Details are provided in the man page that follows.

NOTE

These routines do not work with the nodal CMSSL library.

Copyright © 1994 Thinking Machines Corporation

Gaussian Elimination with External Storage

The routines described below solve the linear system of equations ***AX=B*** where ***A*** is a real or complex square matrix that is too large to fit into core memory. The method used for reducing ***A*** to triangular form is block Gaussian elimination with full column pivoting. The *L* and *U* factors, which overwrite the original matrix, can be used later to solve ***AX=B*** for an arbitrary number of right-hand sides.

SYNTAX

gen_lu_setup_ext (*setup, **C**, n, memory, unit1, unitfe1, ido, col_count, pivoting_strategy, ier*)

gen_lu_factor_ext (*unitfe1, unitfe2, unit1, unit2, restart, ier*)

gen_lu_solve_ext (***B**, nrhs, unitfe1, unitfe2, unit1, unit2, ier*)

ARGUMENTS

setup Output front-end integer array of length $\geq$ 10. Upon completion of **gen_lu_setup_ext**, *setup* contains the following values (see the Description section):

setup(1) The size of the problem actually solved ($N \geq n$).

setup(2) Panel size.

setup(3,4) Processing element layout for the panels.

The other *setup* entries are reserved for internal use. The **gen_lu_setup_ext** routine saves the *setup* array in the file specified by *unitfe1* for later use by **gen_lu_factor_ext**.

C Real or complex CM array of shape (n, p), where n is the number of equations to be solved and $p < n$. ***C*** is used to assemble the n x n coefficient matrix ***A***, p columns at a time; see the Description section.

n Input scalar integer variable. The number of equations to be solved simultaneously.

Copyright © 1994 Thinking Machines Corporation

memory Input scalar integer variable. Specifies the amount of parallel memory per processing element (in bytes) that will determine the size of the blocks of columns (panels) used to partition the coefficient matrix. This much memory must be available when you call **gen_lu_factor_ext**. Note that the amount of memory available during the setup phase may be less than *memory*. If you supply a *memory* value that is ≤ 0, **gen_lu_factor_ext** uses as a default the memory resources available at setup time.

unit1 Input scalar integer variable. Supply the unit number associated with the external storage file (an SFS or parallel CMFS file) indicated below.

Routine	File associated with *unit1*
gen_lu_setup_ext	On return, contains the coefficient matrix A.
gen_lu_factor_ext	On input, must contain the coefficient matrix A. On return, A is overwritten with its LU factors.
gen_lu_solve_ext	On input, must contain the LU factors of A.

You must open the external storage file before calling the setup, factor, or solve routine. The file is not closed at the end of execution.

unitfe1 Input scalar integer variable. Supply the unit number associated with the front-end file (stored on the partition manager) indicated below.

Routine	File associated with *unit1*
gen_lu_setup_ext	On return from the initial setup call, contains the front-end array *setup*.
gen_lu_factor_ext	On input, contains the *setup* array generated by **gen_lu_setup_ext**.
gen_lu_solve_ext	On input, contains the *setup* array generated by **gen_lu_setup_ext**.

Do not modify *unitfe1* between the setup call and the factor or solve call.

You must open the file before calling the setup, factor, or solve routine. The file is not closed at the end of execution.

Copyright © 1994 Thinking Machines Corporation

ido Input/output scalar integer variable. Used in reverse communication. Upon initial input to **gen_lu_setup_ext**, set *ido* = 0. In subsequent iterations, *ido* indicates which operation to perform:

ido = 1 You must fill the CM array ***C*** with appropriate columns of the coefficient matrix, and call **gen_lu_setup_ext** again.

ido = 999 The setup is completed.

col_count Output scalar integer variable. A column index in the interval [1, *n*] returned by **gen_lu_setup_ext** and used in reverse communication. When *ido* = 1, fill array ***C***(*n*, *p*) with columns *col_count* through *col_count*+m−1.

unitfe2 Input scalar integer variable. Supply the unit number associated with the front-end file (stored on the partition manager) indicated below.

Routine	File associated with *unit1*
gen_lu_factor_ext	On return, contains internal information written during program execution. If you call **gen_lu_factor_ext** with *restart* = **CMSSL_lu_restart**, **gen_lu_factor_ext** uses the information stored in this file to restart the factorization from the point of interruption.
gen_lu_solve_ext	On input, contains the internal information written by **gen_lu_factor_ext** during the factorization.

Do not modify *unitfe2* between the factor and solve calls.

You must open the file before calling the factor or solve routine. The file is not closed at the end of execution.

unit2 Input scalar integer variable. Supply the unit number associated with the external storage file (an SFS or parallel CMFS file) indicated below.

Routine	File associated with *unit1*
gen_lu_factor_ext	On return, contains auxiliary internal information.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

gen_lu_solve_ext	On input, must contain the auxiliary internal information written by **gen_lu_factor_ext**.

You must open the external storage file before calling the factor or solve routine. The file is not closed at the end of execution.

restart	Input scalar integer variable. Set to **NULL** if you are starting a new factorization; set to **CMSSL_lu_restart** if you are restarting after an interruption.
B	CM array of rank 2, the same data type as ***A***, and size at least *n* x *nrhs*. On input, must contain the *nrhs* right-hand sides. On return, contains the *nrhs* solutions to $\boldsymbol{AX} = \boldsymbol{B}$.
nrhs	Scalar integer variable. The number of right-hand sides to be solved.
pivoting_strategy	Scalar integer variable specifying the pivoting strategy to be used. In the current release, this argument must be set to **CMSSL_partial_pivoting**.
ier	Scalar integer variable. Return code. Set to 0 upon successful return, or to $-unit$ if an I/O error occurred on file unit *unit*.

DESCRIPTION

Setup Phase. Given *n* equations to be solved simultaneously, and a number specifying the amount of memory that will be available in the subsequent *LU* factorization, the setup routine performs the following tasks:

5. Determines the size of the problem ($N \geq n$) that will actually be solved on the machine.
6. Determines the size of the blocks of columns (panels) used to partition the coefficient matrix.
7. Fills the panels with sub-blocks of columns of the coefficient matrix, and writes the panels to external storage. You supply the sub-blocks using reverse communication.

These tasks are described below.

Copyright © 1994 Thinking Machines Corporation

1. Due to internal constraints, the size of the problem solved by the external *LU* routines, $N = setup(1)$, may be larger than n. The matrix is padded with the bottom $N-n$ rows and rightmost $N-n$ columns of the $N \times N$ identity matrix, as indicated in the figure on page 279. This padding is invisible to the user. As an example, consider solving simultaneously a set of 20,000 double-precision complex equations on a 64-node (256-vector-unit) CM-5 with 32 Mbytes of memory per node (8 Mbytes per vector unit). Assume that 7 Mbytes are available per vector unit; hence, $n = 20{,}000$ and *memory* $= 7{,}000{,}000$. In this case, **gen_lu_setup_ext** determines that $N = 20{,}480$.

2. Internally, the matrix ***A*** is partitioned into blocks of columns, or panels. Factorization requires two panels to be in memory at the same time (in addition to a relatively small amount of extra space for temporary variables). During the setup, the size of the panels is chosen internally to make near optimal use of the memory resources. In the example above, there are 8 panels with panel size 2,560, and the vector unit layout is 32 x 8. Thus, on return from **gen_lu_setup_ext**, $setup(1) = 20{,}480$; $setup(2) = 2{,}560$; $setup(3) = 32$; and $setup(4) = 8$. Each panel occupies 20480*2560*16/256=3,276,800 bytes per vector unit (there are 16 bytes per data element and 64*4 vector units), which is approximately half of 7 Mbytes, the amount of parallel memory per vector unit that will be made available for factorization.

   If $setup(2) = setup(1)$, the problem can be solved in core. In this case, it is more efficient to call the in-core *LU* routines (**gen_lu_factor** and **gen_lu_solve**).

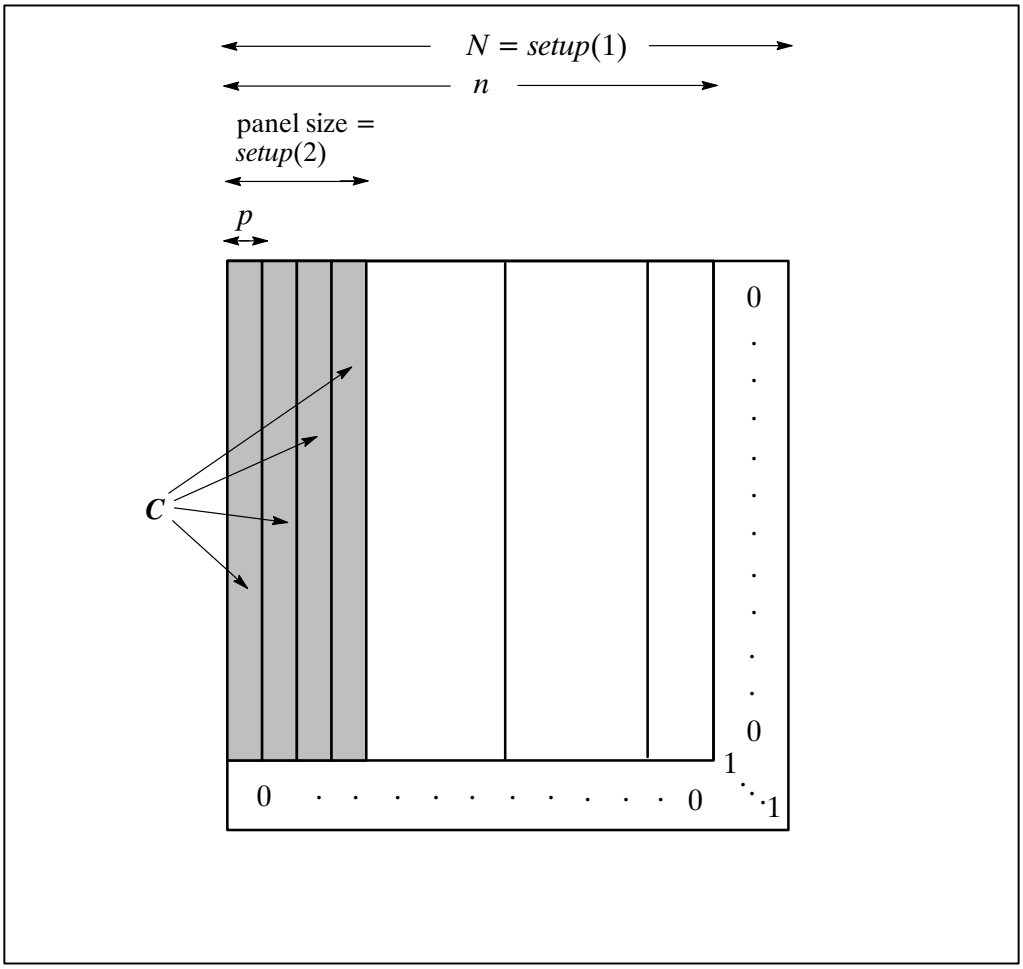
3. The setup routine is called iteratively, and fills the panels with sub-blocks of columns using reverse communication. The purpose of the array ***C***(n, p) is to provide the setup routine with p columns of the coefficient matrix at a time. At each iteration ($ido=1$), you must call your own routine that stores columns *col_count* through $col_count + p - 1$ of the coefficient matrix in ***C***. Internally, these sub-blocks of columns are assembled to make up panels that are written to external storage one at a time.

   If $p=1$, columns of the coefficient matrix are assembled into panels one at a time. A larger value of p allows you to assemble a group of p columns at a time, speeding up the setup phase. In practice, the first time you call the setup routine, you can choose any convenient value of p. When the first setup call returns, you can use the value of $setup(2)$, which is saved on *unitfe1*, to adjust p. Typically, p should be a fraction of $setup(2)$; 1/2, 1/4, 1/8, or 1/16 is expected in most cases, depending on the amount of temporary memory required to build ***C***. Finally, note that if p is not a divisor of n,

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

fewer than p columns are required in the last reverse communication iteration. The content of additional rightmost columns of array ***C*** is ignored in this case.

The figure below illustrates these points.



Setup example. The problem is padded ($N > n$) and has four panels. Four arrays *C* of size (n, p) are required to fill a panel.

Memory Management in the Setup Phase. The amount of data transferred between parallel in-core memory and the external storage device during factorization scales as $O(N^3/b)$, where $b = setup(2)$ is the panel size. It is therefore important to make the panels as large as possible to minimize I/O overhead and obtain good performance. The parallel memory that will be made available for the

Copyright © 1994 Thinking Machines Corporation

factorization (*memory*) determines the size of the panels. The more memory, the bigger and fewer the panels and the better the performance.

Note that it is possible to set up the matrix with less memory than will be available at factorization time. This is because the setup phase requires only one panel in memory at a time (in addition to ***C*** and presumably extra storage required by the user-supplied routine), as opposed to two panels (plus relatively small extra temorary storage) for the factorization. Thus, for example, it is possible to set up the matrix in a timesharing environment (where less than the whole memory is available) and perform the factorization later in dedicated mode with the whole memory available.

The following example shows how to use the reverse communication interface to assemble a double-precision complex coefficient matrix:

```
      complex*16 c(n0,m)
      integer setup(10),n0,unit1,unit2, ido,col_count

      ........

      ido=0

 10   continue
      call
gen_lu_setup_ext(setup,c,n0,memory,unit1,unit2,ido,col_count,ier)
     if(ier.lt.0) then
        print*,'setup error',ier
      stop
     endif
      if (ido.eq.1) then
          call make_slab(c,n0,m,col_count)
      else if(ido.eq.999) then
          print*,'setup done'
          goto 1000
      endif

      goto 10

 1000 continue

   ......

      end
```


Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

```
      subroutine make_slab(x,n0,m,i)
      integer n,m,i
      complex*16 x(n0,m)
cmf$layout x(:,:)

*
* generate columns i to i+m-1 of my coefficient matrix
*
        ........

      end
```

Factorization Phase. The algorithm implemented in **gen_lu_factor_ext** is block Gaussian elimination with full column pivoting.

Since the setup routine saves the setup array in a file, you can factor the matrix at any time after the setup phase. The factorization must be performed on an identical machine configuration as the setup, and each processing element must have at least *memory* free bytes, where *memory* is defined in the setup call.

Memory Management in the Factorization Phase. The factorization phase should be run independently of the setup. Even if as much memory is available for the setup as for the factorization, the region of parallel memory may be fragmented after completion of the setup phase in such a way that no contiguous chunks of memory are available to hold the two panels required by the factorization. It is therefore necessary to clear up the region of parallel memory available before calling **gen_lu_factor_ext**. Currently, the best way to do this is to run a separate program that calls **gen_lu_factor_ext**. In this program, no array should be allocated prior to the **gen_lu_factor_ext** call.

The *LU* factors overwrite the coefficient matrix on *unit1*; *unit2* is used to store the internal information generated for each panel by the in-core *LU* routine. The size of the file associated with *unit2* will be a very small fraction of that associated with *unit1*.

Restart During Factorization. For large problems, runs may take many hours. In case the factor routine is interrupted for any reason, it is desirable to be able to restart factorization where it halted. To this end, information is written on *unitfe2* as the factorization proceeds. This information allows a subsequent factorization call (with **CMSSL_lu_restart** set) to read the history of the previous run and start from wherever the interruption occurred. This information is also used by **gen_lu_solve_ext**.

Copyright © 1994 Thinking Machines Corporation

NOTE: The current restart strategy is not completely secure. In particular, the routine does not handle the case in which the system fails while a factored panel is being written to external storage.

Stability of Factorization. The factorization is stable. In the current release, the only pivoting strategy available is full column pivoting. Future releases may implement other pivoting strategies that will take advantage of diagonal dominance of the coefficient matrix.

Solve Phase. Given a matrix ***A*** factored externally by **gen_lu_factor_ext**, **gen_lu_solve_ext** solves ***AX=B*** for *nrhs* right-hand sides. The machine configuration must be identical to the one used to factor the coefficient matrix. The CM array ***B*** contains the right-hand sides on input, and the solutions after completion of **gen_lu_solve_ext**. ***B*** is thus expected to have dimensions at least (*n*, *nrhs*), where *n* is the number of equations. If the dimensions of ***B*** exceed (*n*, *nrhs*), the upper left submatrix of size (*n*, *nrhs*) should contain the right-hand sides. You may call **gen_lu_solve_ext** any number of times to compute solutions to ***AX=B*** after the matrix ***A*** has been factored by **gen_lu_factor_ext**.

Optimization. Internally, ***B*** is embedded in an array of size (*N*, *M*), where *N* is the size of the padded problem and *M* is the first multiple of *p2* = *setup*(4) that is greater or equal to *nrhs*. This (*N*, *M*) array is laid out on a (*p1*, *p2*) processor grid (*p1*=*setup*(3)). After completion of the solve, the solution matrix of shape (*n*, *nrhs*) is extracted from the (*N*, *M*) array. The extra memory required by this manipulation reduces the number of right-hand sides that can be solved simultaneously. However, if ***B*** already has the shape and layout of the host array described above — that is, shape (*N*, *M*) laid out on a (*p1*, *p2*) processor grid — then no manipulation takes place, reducing memory overhead. In this case, the upper left subarray of ***B*** of shape (*n, nrhs*) should contain the right-hand sides on input, and will contain the respective solutions upon completion of **gen_lu_solve_ext**.

NOTES

No Nodal Library Support. The routines described in this section do not work with the nodal CMSSL library.

Include the CMSSL Header File. Because the **gen_lu_factor_ext** routine uses symbolic constants, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```


Copyright © 1994 Thinking Machines Corporation

at the top of any program module that calls **gen_lu_factor_ext**. This file declares the types of the CMSSL symbolic constants.

File Units. The I/O units *unit1* and *unit2* must be assigned to files before you call the routines that access them. In CM Fortran, file assignment is done with the **CMF_FILE_OPEN** utility (or an equivalent utility for a device or socket). The files associated with *unit1* and *unit2* must be SFS or CMFS files; for example, they may be stored on the Scalable Disk Array (SDA) or a DataVault. For information regarding CM I/O in general, see the *CM-5 I/O System Programming Guide*. For information about CM Fortran I/O routines, see the *CM Fortran Utility Library Reference Manual*.

The file associated with *unit1* will store about as much data as the padded matrix (that is, N^2 data elements), whereas the file associated with *unit2* will contain relatively much less data, the exact amount depending on the machine configuration.

EXAMPLES

Sample CM Fortran code that uses the external *LU* routines can be found on-line in the subdirectory

```
external/lu/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

5.6 *QR* Factorization and Least Squares Solution with External Storage

The routines described in this section perform a *QR* factorization of a real or complex matrix ***A*** of size $m \times n$ (with $m \geq n$) that is too large to fit into core memory. The method uses Householder reflections. The *Q* and *R* factors are stored externally and can later be used to solve ***AX=B*** for an arbitrary number of right-hand sides.

Details are provided in the man page that follows.

NOTE

These routines do not work with the nodal CMSSL library.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

QR Factorization and Least Squares Solution with External Storage

The routines described below perform a *QR* factorization of a real or complex matrix ***A*** of size m x n (with $m \geq n$) that is too large to fit into core memory. The method uses Householder reflections. The *Q* and *R* factors are stored externally and can later be used to solve ***AX=B*** for an arbitrary number of right-hand sides.

SYNTAX

gen_qr_factor_ext (*m, n, blk, type, pivoting_strategy, unit1, unit2, unit3, ier*)

gen_qr_solve_ext (***B***, *nrhs, m, n, blk, type, pivoting_strategy, unit2, unit3, ier*)

ARGUMENTS

m — Scalar integer variable. The number of rows in the matrix ***A*** that is stored on an external device. Also, the number of rows in the matrix ***B***.

n — Scalar integer variable. The number of columns in the matrix ***A*** that is stored on an external device.

blk — Scalar integer variable. Block size. The matrix ***A*** is partitioned into blocks of *blk* columns, or panels. See the Notes section, below, for guidelines for choosing *blk*.

You must use the same block size for both the factor and the solve routine.

type — Scalar integer variable. The data type. Specify one of the following values:

CMSSL_single_real	real*4
CMSSL_double_real	real*8
CMSSL_single_complex	complex*8
CMSSL_double_complex	complex*16

pivoting_strategy — Scalar integer variable specifying the pivoting strategy to be used. The only values currently available are as follows:

Copyright © 1994 Thinking Machines Corporation

CMSSL_no_pivoting	No pivoting, no scaling
CMSSL_no_pivoting_scale	No pivoting, scaling

For a description of scaling, see Section 5.3.9.

B CM array of rank 2, the same data type as ***A***, and size *m* x *nrhs*. On input, must contain the *nrhs* right-hand sides. On return, the first *n* rows of ***B*** contain the *nrhs* solutions to ***AX*** = ***B***.

nrhs Scalar integer variable. The number of columns in ***B***.

unit1 Scalar integer. Valid unit number associated with the file that contains the matrix ***A*** stored in serial order (see the Notes below.) Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number). Data stored in *unit1* is not modified unless *unit1* = *unit2*.

unit2 Scalar integer. Valid unit number associated with the file that will contain the *QR* factors on return from **gen_qr_factor_ext**. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number). If *unit2* = *unit1*, the original matrix ***A*** is overwritten by its *QR* factors.

unit3 Scalar integer. Valid unit number associated with the file that will contain internal information about the *QR* factors on return from **gen_qr_factor_ext**. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a socket or device with a unit number).

ier Scalar integer variable. Return code. Set to 0 upon successful return, or to one of the following error codes:

−1	I/O error on *unit1*.
−2	I/O error on *unit2*.
−3	I/O error on *unit3*.
−4	Invalid type.

Copyright © 1994 Thinking Machines Corporation

DESCRIPTION

The routines described in this man page perform a *QR* factorization of a real or complex matrix ***A*** of size $m \times n$ (with $m \geq n$) that is too large to fit into core memory. The **gen_qr_factor_ext** routine reads blocks of *blk* columns of ***A*** from *unit1*, uses block Householder reflections to factor ***A***, writes the *QR* factors to *unit2*, and writes information about them to *unit3*. The **gen_qr_solve_ext** routine reads the factors from *unit2* and *unit3*, solves $\boldsymbol{AX}=\boldsymbol{B}$ for an arbitrary number of right-hand sides, and returns the *nrhs* solutions in the first *n* rows of ***B***.

The **gen_matrix_mult_ext** routine, described in Chapter 3, can be used to check the accuracy of the result. The best possible accuracy for the solution to $\boldsymbol{A}x = b$ is obtained when $||Ax-b||_\infty / ||A||_\infty ||x||_\infty = \varepsilon$, where ε is the machine accuracy. The quantity $\boldsymbol{AX} - \boldsymbol{B}$ can be computed with **gen_matrix_mult_ext** for all right-hand sides at once.

NOTES

No Nodal Library Support. The routines described in this section do not work with the nodal CMSSL library.

Include the CMSSL Header File. Because the routines described above use symbolic constants, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

at the top of any program module that calls these routines. This file declares the types of the CMSSL symbolic constants.

File Units. The I/O units *unit1*, *unit2*, and *unit3* must be assigned to files before you call the routines that access them. In CM Fortran, file assignment is done with the **CMF_FILE_OPEN** utility (or an equivalent utility for a device or socket). For information regarding parallel I/O in general, see the *CM-5 I/O System Programming Guide*. For information about the CM Fortran interface to parallel I/O, see the *CM Fortran Utility Library Reference Manual*. As described in this manual, there are essentially two modes of external storage: Fixed Machine Size (FMS) and Serial Order (SO). Serial order is the familiar Fortran row-major order and is the one used by the external *QR* routines. Therefore, ***A*** must be stored in serial order in file unit *unit1*. In this order, the data is portable across the CM-5 external storage systems (DataVault, Scalable Disk Array, HIPPI).

Copyright © 1994 Thinking Machines Corporation

The file associated with *unit2* will store as much data as the original matrix (that is, *nm* data elements), whereas the file associated with *unit3* will contain much less data, the exact amount depending on the machine configuration.

If you set *unit2* = *unit1*, the original matrix $\boldsymbol{A}$ is overwritten by its *QR* factors.

Partition Size. The partition size used to solve the system of equations must be identical to the one used previously to factor the matrix.

Choosing the Block Size. The block Householder factorization algorithm partitions the matrix into block columns, or panels, A_i, of size m x *blk*:

$$\boldsymbol{A} = [\, A_1, A_2, ..., A_l \,].$$

The last panel, A_l, contains fewer than *blk* columns if *blk* is not a divisor of n. It is important to choose the block size *blk* as large as possible in order to minimize the I/O cost and optimize machine utilization. The in-core memory requirement for **gen_qr_factor_ext** is approximately $(5v + 16)m^*blk$ bytes, where v is the number of bytes in the data type of $\boldsymbol{A}$.

Choosing the block size *blk* to be a multiple of 16 may also improve performance. This is because the blocking factor, *nblock*, in the in-core *QR* routine **gen_qr_factor** (upon which **gen_qr_factor_ext** is built) is set internally to 16. Given this fact and the memory requirement mentioned above, it is possible to choose a reasonable *blk* value for a given n and a given amount of core memory.

Least Squares Solution. The least squares solution obtained from **gen_qr_solve_ext** is unique only if the problem has full rank (rank $\boldsymbol{A} = n$). Unlike the in-core *QR* routines, the current out-of-core version does not provide a way for you to determine the rank of $\boldsymbol{A}$.

Complexity Analysis. The **gen_qr_factor_ext** routine requires $2n^2[m - (n/3)]$ operations for real operands and $8n^2[m - (n/3)]$ operations for complex operands. The amount of data transferred during the block *QR* factorization is

$$\frac{n(n + blk)}{blk}\left[m + \frac{blk - n}{3}\right].$$

For *blk*/n small, this quantity becomes

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

$$O\left[\frac{n^2}{blk}\left\{m-\frac{n}{3}\right\}\right].$$

Given these numbers, the average floating-point operation (flop) rate for the in-core *QR* routine, and the data transfer rate between the CM and the external storage system, it is possible to make a very rough estimate of the time required for the out-of-core factorization. On the CM-5, for example, a conservative choice for the flop rate for the in-core *QR* routine is 10 Mflops per vector unit, while the data transfer rate on the Scalable Disk Array is roughly 1 Mbyte/sec per disk. With p vector units and q disks, the estimated time for a problem of size m x n is

$$T_{arith} = \frac{un^2}{10p}\left\{m-\frac{n}{3}\right\}10^{-6} \text{ seconds and } T_{transfer} = \frac{n^2}{blk}\left\{m-\frac{n}{3}\right\}\frac{v}{q}\,10^{-6} \text{ seconds}$$

where $u = 2$ for real operands and $u = 8$ for complex operands, and v is the number of bytes in the matrix data type. Hence, the total time is

$$T = n^2\left[m-\frac{n}{3}\right]\left[\frac{u}{10p}+\frac{v}{q*blk}\right]10^{-6} \text{ seconds.}$$

Choosing, for example, $u = 2, v = 8$ (that is, a data type of real*8), $m = 10{,}000, n = 5{,}000, blk = 1200, p = 128$ vector units, and $q = 8$, we have $T_{arith} = 325$ seconds and $T_{transfer} = 174$ seconds, and hence a total time of $T \approx 10$ minutes. Only the order of magnitude of such an estimate should be considered significant.

EXAMPLES

Sample CM Fortran code that uses the *LU* routines can be found on-line in the subdirectory

```
external/qr/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

5.7 References

For information about general linear solvers, see the following references:

1. Golub, G.H., and C. F. Van Loan. *Matrix Computations.* 2d ed. Baltimore: Johns Hopkins University Press, 1989.

2. Johnsson, S. L. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *Journal of Parallel and Distributed Computing* **4** (1987): 133–72.

3. Wilkinson, J. H. Error Analysis of Direct Methods of Matrix Inversion. *J. Assoc. Comput. Mach.* **8** (1961): 281–330.

For information about how the *LU* and *QR* routines implement blocking, and other aspects of the *LU* and *QR* operations, refer to

4. Dongarra, J. J., I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Philadelphia: SIAM, 1991.

5. Hager, W. W. Condition Estimates. *SIAM J. Sci. Stat. Comput.* **5** (1984): 311–16.

6. Higham, N. J. Experience with a Matrix Norm Estimator. *SIAM J. Sci. Stat. Comput.* **11**, no. 4 (1990): 804–9.

7. Higham, N. J. FORTRAN Codes for Estimating the One-Norm of a Real or Complex Matrix, with Applications to Condition Estimation. (Algorithm 674) *ACM Trans. Math. Soft.* **14** (1988): 381–96.

8. Higham, N. J. The Accuracy of Solutions to Triangular Systems. *SIAM J. Numer. Anal.* **26**, no.5 (1989):1252–65.

9. Johnsson, S. L. A Computational Array for the *QR*-method. *Proceedings of the Conference on Advanced Research in VLSI.* Ed. P. Penfield, Jr. Artech House, 1982. Pp. 123–29.

10. Lichtenstein, W. and S. L. Johnsson. Block Cyclic Dense Linear Algebra. *SIAM J. Sci. Comp.* **14**, no. 6 (1993): 1257–86.

11. Schreiber, R. S., and C. F. Van Loan. A Storage Efficient WY Representation for Products of Householder Transformations. *SIAM J. Sci. Stat. Comput.* **10**, no. 1 (1989): 53–57.

Copyright © 1994 Thinking Machines Corporation

12. Vavasis, S. *Implementation of QR factorization on the Connection Machine CM-2*. Personal communication.

13. Wilkinson, J. H. *The Algebraic Eigenvalue Problem*. Oxford: Clarendon Press, 1965.

14. Van de Geijn, R. A. *Massively Parallel LINPACK Benchmark on the Intel Touchstone Delta and iPSC/860 Systems*. Technical Report, University of Texas at Austin, 1991.

For an analysis of the numerical behavior of the algorithm used in the matrix inversion and Gauss-Jordan reduction routines, see the following:

15. Dekker, T. J., and W. Hoffman. Rehabilitation of the Gauss-Jordan Algorithm. *Numerische Mathematik* **54** (1989): 591−99.

Copyright © 1994 Thinking Machines Corporation