Algorithm 9xx, FACTORIZE: an object-oriented linear system solver for MATLAB

TIMOTHY A. DAVIS
University of Florida

The MATLAB^{TM}backslash (`x=A\b`) is an elegant and powerful interface to a suite of high-performance factorization methods for the direct solution of the linear system $Ax = b$ and the least-squares problem $\min_x ||b - Ax||$. It is a meta-algorithm that selects the best factorization method for a particular matrix, whether sparse or dense. However, the simplicity and elegance of its single-character interface prohibits the reuse of its factorization for subsequent systems. Requiring MATLAB users to find the best factorization method on their own can lead to sub-optimal choices; even MATLAB experts can make the wrong choice. Furthermore, naive MATLAB users have a tendency to translate mathematical expressions from linear algebra directly into MATLAB, so that $x = A^{-1}b$ becomes the inferior yet all-to-prevalent `x=inv(A)*b`. To address these issues, an object-oriented `FACTORIZE` method is presented. Via simple-to-use operator overloading, solving two linear systems can be written as `F=factorize(A); x=F\b; y=F\c`, where `A` is factorized only once. The selection of the best factorization method (LU, Cholesky, LDL^T, QR, or a complete orthogonal decomposition for rank-deficient matrices) is hidden from the user. The mathematical expression $x = A^{-1}b$ directly translates into the MATLAB expression `x=inverse(A)*b`, which does not compute the inverse at all, but does the right thing by factorizing `A` and solving the corresponding triangular systems.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*linear systems (direct methods), sparse and very large systems*; G.4 [**Mathematics of Computing**]: Mathematical Software—*algorithm analysis, efficiency*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: linear systems, least-square problems, matrix factorization, object-oriented methods

1. INTRODUCTION

MATLAB provides many ways to solve linear systems and least-squares problems, the most obvious one being `x=A\b`. This method is powerful and simple to use, but its factorization cannot be reused to solve multiple linear systems. An object-oriented programming approach is presented that makes solving systems and

Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, USA. email: davis@cise.ufl.edu or DrTimothyAldenDavis@gmail.com. http://www.cise.ufl.edu/∼davis. Portions of this work were supported by the National Science Foundation, under grants 0620286 and 1115297.

reusing a factorization in MATLAB very easy to do, even for the naive MATLAB user. Section 2 provides a motivation for the `FACTORIZE` package presented in Section 3. Code availability and concluding remarks are given in Section 4.

2. MOTIVATION

The MATLAB backslash is a powerful method, but it has its weaknesses. These are discussed in Section 2.1 below. The factorization methods in MATLAB provide an alternative, but using them efficiently is not trivial. This is illustrated by a sequence of experiments in Section 2.2 that demonstrate the performance of the diverse factorization methods in MATLAB. The section concludes with a list of best-of-breed methods for sparse and dense LU, Cholesky, and QR factorizations. Even MATLAB experts (at The MathWorks, Inc.) find it difficult to select the right method, as illustrated by how built-in MATLAB functions use these methods (Section 2.3). The unfortunate prevalence of "`inv`-abuse" (`x=inv(A)*b`) illustrates yet another motivation for the object-oriented `FACTORIZE` package, as highlighted in Section 2.4. This gap in MATLAB functionality is summarized in Section 2.5, which motivates the `FACTORIZE` package presented in Section 3.

2.1 The strengths and weaknesses of the MATLAB backslash

The backslash operator (or `mldivide`, to use its precise name) is a meta-algorithm that automatically selects an appropriate solver for the matrix A [Gilbert et al. 1992]. If the matrix is diagonal, upper triangular, lower triangular, or a permutation of a triangular matrix, then it is not factorized at all. If the matrix requires factorization, backslash selects an LU, Cholesky, LDL^T, or QR factorization, depending on the matrix. It sometimes attempts multiple factorizations. For example, if the matrix is square and symmetric with a zero-free real diagonal, a Cholesky factorization is attempted. If this fails, an LDL^T factorization is used, and if this fails, backslash falls back to an LU factorization. For rectangular matrices, QR factorization of A is always performed, resulting in a least-squares solution for over-determined systems, and a basic solution when A is under-determined.

Backslash is a powerful function but it has three minor drawbacks and a fourth that is more serious:

(1) There is no way to request a minimum 2-norm solution to an under-determined system.
(2) There is no special handling for square rank-deficient problems. These should be solved with a QR factorization, a complete orthogonal decomposition, or perhaps even a singular value decomposition, so that a least-squares or minimum 2-norm solution could be obtained, depending on the matrix.
(3) Textbook equations from linear algebra often rely on the explicit inverse, A^{-1}. These expressions do not directly translate into a MATLAB expression using backslash.
(4) The factorization computed by backslash cannot be reused. All of the elegant power of backslash's automatic selection of an appropriate solver must be discarded if the user wishes to reuse the factorization of `A`.

In spite of these drawbacks, backslash remains a powerful and general-purpose operator that works well for most users' systems of equations. However, if a MATLAB code needs to reuse a factorization, it must either duplicate the intricacies of the backslash selection, or it must resort to using a potentially sub-optimal factorization technique. Significant expertise on the part of the MATLAB user is required to obtain the fastest and most memory-efficient technique.

2.2 The many factorization methods in MATLAB and their performance profiles

For dense matrices, MATLAB relies on the LU, Cholesky, QR, LDL^T, and SVD factorizations provided by LAPACK [Anderson et al. 1999]. For sparse matrices, MATLAB uses the sparse LU, Cholesky, and QR factorizations in SuiteSparse (UMFPACK, CHOLMOD, and SuiteSparseQR, respectively), a sparse LU by Gilbert and Peierls, MA57 for its sparse LDL^T factorization [Duff 2004], and various specialized solvers (for triangular, tridiagonal, and other special cases), some of which are not available to the MATLAB user except through `x=A\b`.

Selecting between these methods is a daunting task for the basic MATLAB user. Care must be taken in the design of the `FACTORIZE` package to use the best technique for each matrix, considering reliability, speed, and memory requirements.

The first step in determining the best methods is to consider how permutations should be handled (Section 2.2.1), since permutations are used by many factorizations, both sparse and dense. Next, the performance characteristics of alternative sparse and dense LU, Cholesky, and QR factorizations are considered (Sections 2.2.2 through 2.2.8). The best-of-breed methods are summarized in Section 2.2.9.

2.2.1 *Permutations.* Both sparse and dense factorization methods return permutations to represent partial pivoting and/or fill-reducing orderings. During the solve phase, these permutations must be applied to the right-hand side and/or the solution vector. Since permutations can be returned as either index vectors or permutation matrices, the decision on which to use should be based on performance, both time and memory.

Dense case: A dense factorization method in MATLAB can return a dense permutation matrix `P` or a dense vector index `p`. The matrix `P` requires $O(n^2)$ memory in contrast to $O(n)$ memory for the index vector `p`. Likewise, applying the matrix `P` to a right-hand side vector takes $O(n^2)$ time as opposed to $O(n)$ time for the vector `p`. Permutation vectors are thus preferable. Alternatively, a dense permutation vector `p` can be converted into a sparse permutation matrix `P`, via the MATLAB statement `P=sparse(1:n,p,1)`.

Sparse case: Sparse factorization methods return either permutation vectors (requiring $8n$ bytes) or sparse permutation matrices (requiring $24n$ bytes). The difference in memory is slight, since the factorization itself is typically much larger, so the method providing the fastest solve time should be selected.

Table I lists the run time in seconds for the two permutation operations `y=P*x` and `y=P'*x` (where `P` is sparse) and their index-vector equivalents, for various lengths of a dense or sparse vector `x`. The relative run time is the time for the index operation divided by the time for the matrix operation. Unless otherwise specificed, all results in this paper were obtained from a 24-core AMD Opteron 6168 CPU system with

x is dense						
n	y=x(p,:)	y=P*x	relative	y(p,:)=x	y=P'*x	relative
100	3.6×10^{-6}	1.3×10^{-5}	0.26	5.2×10^{-6}	1.4×10^{-5}	0.38
1000	1.7×10^{-5}	2.8×10^{-5}	0.62	2.4×10^{-5}	2.3×10^{-5}	1.04
10000	1.6×10^{-4}	1.7×10^{-4}	0.92	2.3×10^{-4}	1.1×10^{-4}	1.98
100000	2.0×10^{-3}	2.2×10^{-3}	0.92	2.7×10^{-3}	1.7×10^{-3}	1.61
1000000	4.0×10^{-2}	5.1×10^{-2}	0.77	6.0×10^{-2}	6.9×10^{-2}	0.87
x is sparse, but with no zero entries						
n	y=x(p,:)	y=P*x	relative	y(p,:)=x	y=P'*x	relative
100	1.7×10^{-5}	2.1×10^{-5}	0.84	4.0×10^{-5}	2.5×10^{-5}	1.60
1000	9.3×10^{-5}	9.7×10^{-5}	0.96	1.9×10^{-3}	1.1×10^{-4}	17.17
10000	1.2×10^{-3}	1.1×10^{-3}	1.03	1.7×10^{-1}	1.4×10^{-3}	123.15
100000	1.6×10^{-2}	1.5×10^{-2}	1.04	1.8×10^{1}	2.1×10^{-2}	876.97
1000000	2.3×10^{-1}	2.5×10^{-1}	0.90	2.0×10^{3}	4.0×10^{-1}	4995.93
x is sparse, with 1% nonzero entries						
n	y=x(p,:)	y=P*x	relative	y(p,:)=x	y=P'*x	relative
100	1.4×10^{-5}	1.7×10^{-5}	0.93	3.6×10^{-5}	1.9×10^{-5}	1.95
1000	3.8×10^{-5}	2.6×10^{-5}	1.55	1.9×10^{-3}	4.3×10^{-5}	44.35
10000	2.8×10^{-4}	1.1×10^{-4}	2.51	1.9×10^{-1}	4.0×10^{-4}	459.49
100000	3.3×10^{-3}	1.7×10^{-3}	1.90	1.8×10^{1}	9.5×10^{-3}	1933.66
1000000	5.4×10^{-2}	4.6×10^{-2}	1.18	2.0×10^{3}	2.7×10^{-1}	7584.81

Table I. Run time in seconds for sparse permutation matrices and permutation vectors in MATLAB. The matrix `P` is sparse, and is related to `p` via `P=sparse(1:n,p,1)`. These results demonstrate that `P*x` and `P'*x` tend to be faster than the equivalent operation with permutation vectors for large sparse vectors. The difference is extreme for `P'*x`.

128GB of RAM, running MATLAB R2011b and Ubuntu Linux 12.04.

A permutation vector `p` is far too slow for computing `y(p)=x` when `x` and `y` are sparse. One alternative is to exploit an inverse permutation vector `invp` via `invp=1:n; invp(p)=1:n`. With this vector, `y(p)=x` and `y=x(invp)` are equivalent. The latter is much faster when `x` is sparse.

Not all solve phases for the various sparse factorization methods require the computation of `P'*x` or `y(p)=x` for a sparse vector `x`. However, rather than constructing inverse permutation vectors to handle this case, the sparse factorizations in the `FACTORIZE` package rely on sparse permutation matrices instead of vectors. Sparse permutation matrices are faster in MATLAB 2011b for large vectors, even though they do require a modest amount of additional storage.[1]

The results in this section confirm that the `FACTORIZE` package should use permutation vectors for its dense factorizations, and sparse permutation matrices for its sparse factorizations. The next section on dense LU considers alternative options to confirm this selection (including a permuted lower triangular matrix `L`), but subsequent sections will present only one option for handling permutations.

2.2.2 *Dense LU factorization.* There are three alternatives for dense LU factorization that can be used to solve a linear system. Each can be coupled with two methods for backslash (`x=A\b`) and three for slash (`y=c/A`), listed below.

[1]The code behind the built-in `A*B` operation in MATLAB, for the case when either `A` or `B` or both are sparse, can be found at http://www.suitesparse.com.

(1) `[L,U,p]=lu(A,'vector')` is the least-obvious syntax, returning the partial pivoting choices as a permutation vector `p`, where `L*U=A(p,:)`. The five solve steps are listed below, where the `op*` parameters are `struct`'s that describe the properties of `L` and `U` for `linsolve`:
   (a) `x = linsolve (U, linsolve (L, b (p,:), opL), opU)`
   (b) `x = U \ (L \ b (p,:))`
   (c) `y = (linsolve (L, linsolve (U, c', opUT), opLT))' ; y (:,p) = y`
   (d) `y = ((c / U) / L) ; y (:,p) = y`
   (e) `y = (L' \ (U' \ c'))' ; y (:,p) = y`

(2) `[L,U,P]=lu(A)` is the most natural syntax, but it returns `P` as a dense matrix. The factorization is `L*U=P*A`. The five solve steps are:
   (a) `x = linsolve (U, linsolve (L, P*b, opL), opU)`
   (b) `x = U \ (L \ (P*b))`
   (c) `y = (linsolve (L, linsolve (U, c', opUT), opLT))' * P`
   (d) `y = ((c / U) / L) * P`
   (e) `y = (L' \ (U' \ c'))' * P`

(3) `[L,U]=lu(A)` returns `L` as a permuted lower triangular matrix, where `L*U=A`. The permutation is multiplied into `L` so that it does not require a seperate representation, but `linsolve` cannot be used for `L`. The solves are:
   (a) `x = linsolve (U, L \ b, opU)`
   (b) `x = U \ (L \ b)`
   (c) `y = (linsolve (U, c', opUT)') / L`
   (d) `y = ((c / U) / L)`
   (e) `y = (L' \ (U' \ c'))'`

The performance of these three LU factorization methods and their solve phases is illustrated in Figures 1 and 2. Method 1 (`[L,U,p]=lu(A,'vector')`) is slightly faster than the other two alternatives. Its factors also take nearly 50% less memory to store as compared with method 2. For the solves, methods 1a for `x=A\b` and 1c for `y=A/c` are far faster than the alternatives.

To compute `x=L\b`, MATLAB can detect when `L` is lower triangular or a permuted lower triangular matrix, and use a forward solve that does not require `L` to be factorized. However, the time to compute `y=c/A` and `y=c/L` is identical when `L` is a dense permuted lower triangular matrix. MATLAB 2011b does not detect this case and refactorizes `L` instead. As a result, `[L,U]=lu(A)` is not a useful factorization for the subsequent solution of $yA = c$.

The two statements `y=c/L` and `y=(L'\c')'` have identical performance because the MATLAB interpretter immediately translates the first expression into the second. Thus, subsequent sections present results for only one of the two methods.

2.2.3 *Sparse LU factorization.* MATLAB includes two sparse LU factorization methods: GP [Gilbert and Peierls 1988] and UMFPACK [Davis and Duff 1999; Davis 2002; 2004]. GP provides the two- and three-output syntax `[L,U,P]=lu(A)`, with no fill-reducing ordering (`P` arises from partial pivoting). UMFPACK provides the four- and five-output syntax `[L,U,P,Q,R]=lu(A)`, where `P` and `Q` are fill-reducing orderings and `P` also includes partial pivoting permutations.

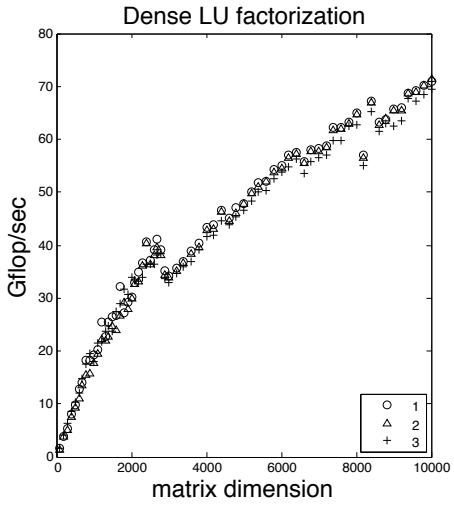

Fig. 1. Performance of three factorization methods for dense matrices.

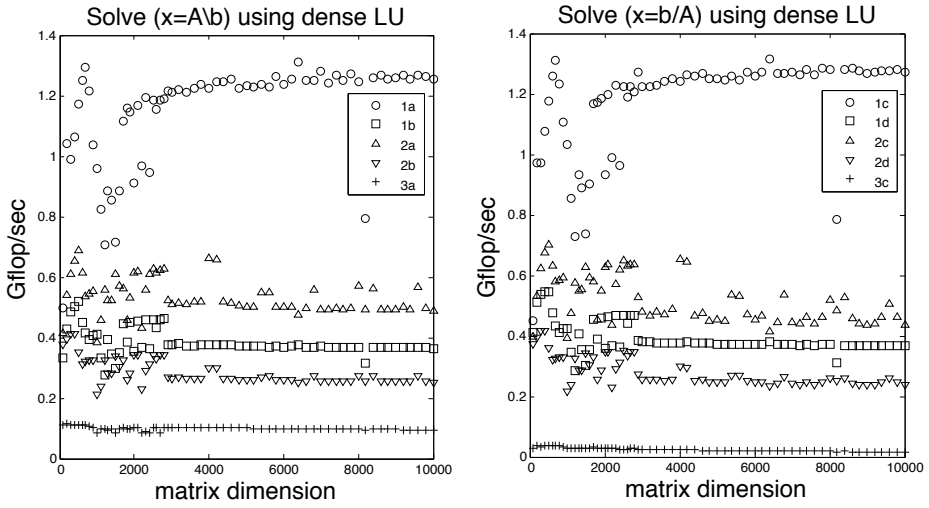

Fig. 2. Performance of solve methods for dense matrices. Solve methods 3a and 3b have nearly identical performance, so 3b is not shown. Likewise, 1e and 2e have nearly identical performance as 1d and 2d, respectively, and are not shown. Methods 3c, 3d, 3e have nearly identical performance, and thus only 3c is shown.

The GP algorithm can be faster than UMFPACK for some matrices (circuit simulation matrices and other very sparse matrices in particular [Davis and Palamadai Natarajan 2010]), but UMFPACK tends to be faster in the general case. UMFPACK requires a fourth output argument since its fill-reducing ordering cannot be disabled. UMFPACK is used by default when `x=A\b` requires a general sparse LU factorization. Four alternatives are considered below:

(1) `[L,U,P,Q,R]=lu(A)` is the method used internally by `x=A\b` when a general sparse LU factorization is required. The diagonal matrix `R` provides row-scaling, which tends to improve accuracy and reduce fill-in in the factorization. The factorization is `L*U=P*(R\A)*Q`, and thus the solves are:
   (a) `x = Q * (U \ (L \ (P * (R \ b))))` for `x=A\b`.
   (b) `y = ((((c * Q) / U) / L) * P) / R` for `y=c/A`.

(2) `[L,U,P,Q]=lu(A)` also uses UMFPACK, but skips the diagonal scaling. The factorization is `L*U=P*A*Q`, and thus the solves are:
   (a) `x = Q * (U \ (L \ (P * b)))` for `x=A\b`.
   (b) `y = (((c * Q) / U) / L) * P` for `y=c/A`.

(3) `Q=sparse(colamd(A),1:n,1) ; [L,U,P]=lu(A*Q)` produces the same factorization as method 2, but using the Gilbert-Peierls method and a different fill-reducing permutation. The solve phases are the same as method 2.

(4) `Q=sparse(amd(A),1:n,1); [L,U,P]=lu(Q'*A*Q,0.1); P=P*Q'` produces the same factorization as methods 2 and 3 (`L*U=P*A*Q`), but with a fill-reducing ordering suitable for matrices with symmetric nonzero pattern (or mostly symmetric). The threshold partial pivoting parameter (`0.1`) attempts to preserve symmetry by giving preference to diagonal entries. UMFPACK (methods 1 and 2) automatically selects between AMD [Amestoy et al. 1996; 2004] and COLAMD [Davis et al. 2004a; 2004b], based on the pattern of the matrix.

The MATLAB `linsolve` function does not work for sparse matrices, so operations such as `x=L\b` must be used instead. The MATLAB backslash quickly determines that `L` is lower triangular in this case, although better performance could be obtained if `linsolve` worked for sparse matrices. This can be seen by the simple function `cs_lsolve` in the CSparse package, for example [Davis 2006].

The hope that a future `linsolve` could handle sparse matrices is yet another motivation for the `FACTORIZE` package, since exploiting a sparse `linsolve` would require a simple one or two-line change to each of the six sparse factorization methods in the package. Other codes that do not attempt to use the MATLAB factorization methods themselves but rely on the `FACTORIZE` package would not have be modified to exploit this possible future upgrade. All that would be needed would be to upgrade to the next version of `FACTORIZE`.

The performance UMFPACK and GP is shown in Figure 3 (methods 1 to 4 from the list above) using test matrices from [Davis and Hu 2011]. In the left plot, each point in the figure is a single matrix. The x axis is the best flop count found by any method for that particular matrix, divided by the best number of nonzeros in the LU factors. The y axis is the relative time: the best run time of GP (methods 3 and 4) divided by the best run time for UMFPACK (method 1). GP and KLU [Davis and Palamadai Natarajan 2010] are faster than UMFPACK for sparse matrices

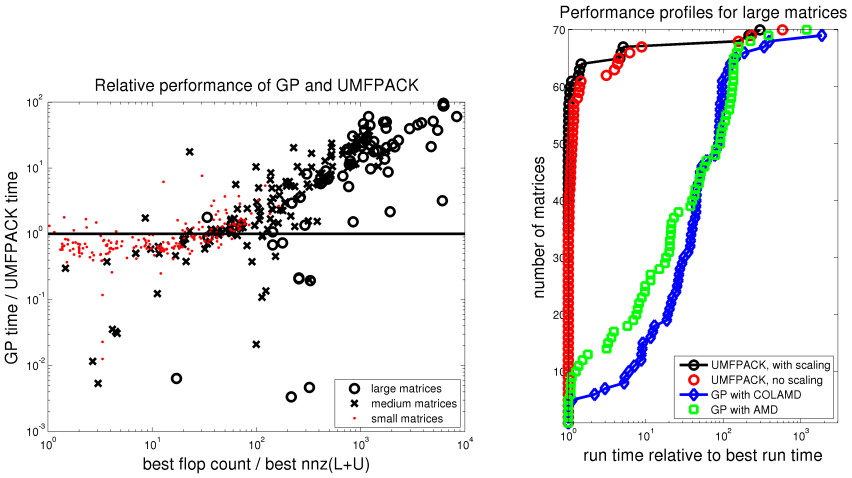

Fig. 3. The left plot shows the relative performance of GP and UMFPACK as a function of the relative number of flops per entry in the factors. The right plot is the performance profile for each method. A matrix is defined as "large" in the plots if the best factorization time was greater than 5 seconds, "medium" if between 0.1 and 5 seconds, and "small" otherwise. Results are from nearly all real non-singular square unsymmetric matrices in the UF Sparse Matrix Collection. The ten largest matrices were excluded because of the excessive run time and/or memory to required to factorize them with all four methods.

arising from circuit simulation; these are the large outliers in the figure. For many matrices, scaling has little effect at all. For those matrices for which scaling has an effect, it almost always improves the accuracy, fill-in, and run time. Thus, method 2 (UMFPACK without scaling) is not shown in the left plot of Figure 3.

GP is faster than UMFPACK for small matrices and for some circuit matrices, but in general the results in Figure 3 show that UMFPACK (`[L,U,P,Q,R]=lu(A)`) is the best choice for most sparse unsymmetric matrices. UMFPACK does poorly for some circuit matrices because its automatic ordering method selection makes the wrong choice (it selects COLAMD, but AMD works much better for those matrices).

2.2.4 *Dense Cholesky factorization.* There are two options for the dense Cholesky factorization: `R=chol(A)` or `L=chol(A,'lower')`, which return the result as an upper or lower triangular matrix, respectively. The performance of the two methods is comparable, but only `R=chol(A)` can be used for a rank-1 update/downdate via the MATLAB `cholupdate` function. The `FACTORIZE` package provides an interface to `cholupdate`, and thus it relies on `R=chol(A)`.

2.2.5 *Sparse Cholesky factorization.* For sparse symmetric positive definite matrices, `x=A\b` relies on CHOLMOD [Chen et al. 2008], which computes an up-looking non-supernodal sparse Cholesky factorization when `A` is very sparse, and a left-looking supernodal sparse Cholesky factorization otherwise. CHOLMOD returns a lower triangular factor `L`, and thus `[L,g,P]=chol(A,'lower')` takes less memory and is slightly faster than `[R,g,P]=chol(A)` since the latter must compute `R=L'`.

2.2.6 *Dense QR factorization.* The decision for which dense QR factorization to use is based on a tradeoff between reliability and performance, with the default being reliability.

For dense rectangular matrices, `x=A\b` relies on QR factorization with column pivoting. The `FACTORIZE` package does the same, but uses a more reliable technique for rank-deficient problems. If the matrix is found to be rank-deficient, the first factorization (QR with column pivoting) is follwed by a second RQ factorization to obtain the complete orthogonal decompostion (COD) $URV^T = A$, where R is r-by-r and upper triangular, U and V have orthonormal columns, and r is the estimated rank of A [Golub and Van Loan 1989].

The COD algorithm for dense matrices is shown below, assuming that $m \geq n$. For the dense case, if A has full rank, V is a permutation matrix arising from column 2-norm pivoting; for the sparse case (not shown), V represents the fill-reducing column ordering. If A has more columns than rows ($m < n$), this algorithm is applied to A^T, and the results are transposed and permuted to obtain $URV^T = A$ with R upper triangular.

```
function [U, R, V, r] = cod (A, tol)
[m, n] = size (A) ;
[U, R, V] = qr (A, 0) ;     % economy U*R = A(V,:) with column pivoting
V = sparse (V, 1:n, 1) ;    % R n-by-n and triu, U m-by-n, V n-by-n
r = sum (abs (diag (R)) > tol) ;         % estimated rank of R
if (r < n)
    [R, Q] = rq (R, r, n) ; % RQ factorization, R now upper triangular
    U = U (:, 1:r) ;        % discard all but the first r columns of U
    V = V * Q' ;            % merge Q and V
end

function [R, Q] = rq (A, r, n)
[Q, R] = qr (A (r:-1:1, n:-1:1)', 0) ;
R = R (end:-1:1, end:-1:1)' ;
Q = Q (end:-1:1, end:-1:1)' ;
```

The RQ factorization can exploit the initial upper trapezoidal r-by-n structure of the matrix R, and thus it takes very little time if r is nearly equal to n.

If the rank is well-defined and accurately detected, the solve phase (not shown) returns the pseudo-inverse solution `x=pinv(A)*b` without computing the more-costly singular value decomposition. The MATLAB `x=A\b` does not use the complete orthogonal decomposition. In the full-rank case, this extra check adds essentially no extra work as compared to `x=A\b`, which uses QR with column pivoting.

Thus, to match the reliability of `x=A\b` the `FACTORIZE` package uses QR with column pivoting for the full-rank case, and exceeds the reliability of `x=A\b` for the rank-deficient case via the COD.

In MATLAB 2012a, QR with column pivoting is based on the `DGEQP3` LAPACK function, which uses the level-3 BLAS [Anderson et al. 1999]. It is about 3 to 4 times slower than QR factorization with no column pivoting (`DGEQRF`), but the increase in reliability is worth the extra work for the default case. The two methods are shown below. On a quad-core Intel i5-2400 CPU with 16GB of RAM and MATLAB R2012a, with a full-rank matrix of size 6000-by-3000, method 1 takes 14.8 seconds whereas method 2 takes 3.5 seconds.

(1) `[Q, R, p] = qr (A,0) ;`
   (a) `x = linsolve (R, Q' * b, opU) ; x (p,:) = x ;`
   (b) `y = (Q * linsolve (R, c (:,p)', opUT))' ;`

(2) `[Q, R] = qr (A,0) ;`
   (a) `x = linsolve (R, Q' * b, opU) ;`
   (b) `y = (Q * linsolve (R, c', opUT))' ;`

For better performance in full-rank case, the user of the `FACTORIZE` package can request a QR factorization without column pivoting (method 2, above), via `F=factorize(A,'qr')`. This option is not available when using `x=A\b`.

For matrices with more columns than rows, `x=A\b` returns a basic solution, but the `FACTORIZE` package returns a better solution via QR factorization of the transposed matrix `A'`. This gives the unique minimum 2-norm solution if `A` has full rank.

2.2.7 *Sparse QR factorization.* The sparse QR factorization in MATLAB (R2009a and later) relies on SuiteSparseQR [Davis 2011a]. Ten different syntax options are listed in `doc qr` in MATLAB R2012a, while the `spqr` mexFunction posted at http://www.suitesparse.com has 15. Selecting the right syntax depends on what the user requires, but different syntaxes have very different performance profiles. This makes for a daunting choice for the basic MATLAB user.

Both `qr` and `spqr` can return `Q` as a sparse matrix, but this is not practical for large problems since `Q` can have many nonzero entries. The `spqr` function can return a representation of `Q` as a set of sparse Householder vectors, which can be as sparse as `L` from a sparse LU factorization [Davis 2006]. This is much more efficient in time and memory, but this feature is not available to MATLAB users (as of R2012a). If `Q` is not needed, the sparse Householder vectors can also be discarded as they are computed, which saves a substantial amount of time and memory.

Thus, the best strategy for the QR factorization is to discard `Q` and use the corrected semi-normal equations [Golub and Van Loan 1989], with one step of iterative refinement. This option is suggested by `doc qr` in MATLAB, except that a fill-reducing ordering should also be used (this is not suggested by `doc qr`).

While `spqr` can return just `R` and the fill-reducing column ordering `p` at the same time as discarding `Q`, MATLAB does not provide this option. However, `qr` in MATLAB does have an option to discard `Q` as it is computed, while at the same time applying it to a second vector or matrix `b`, with the syntax `[C,R,p]=qr(A,b,0)`. This option returns `C=Q'*b`, which is used internally by `x=A\b`.

However, `C` is not useful if the QR factorization needs to be reused with a different right-hand side, `b`. Thus, the best MATLAB syntax for returning `R` and `p` while discarding `Q` is method 1 in Table II, which is very non-obvious. The table also lists the run time for the Pereyra/landmark matrix from [Davis and Hu 2011].

In method 1, `C=Q'*sparse(m,0)` is computed and then discarded. This adds very little time and memory to the computation in SuiteSparseQR. A MATLAB user might be tempted to use method 2, which seems to do the right thing by discarding the first output argument `Q` for the computation `[Q,R,p]=qr(A,0)`, but it is very costly. The tilde argument (`~`) for `[~,R,p]=qr(A,0)`, tells MATLAB to discard the first output argument. However, what happens internally is that SuiteSparseQR is told to compute this argument `Q`, and then MATLAB discards it at the very

method		time (sec)
1	`[~, R, p] = qr (A, sparse (size (A,1), 0), 0) ;`	0.8
2	`[~, R, p] = qr (A, 0) ;`	6057.2
3	`p = amd (A'*A) ; R = qr (A (:,p), 0) ;`	1.0
4	`p = colamd (A) ; R = qr (A (:,p), 0) ;`	7.6

Table II. Sparse QR factorization and results with the Pereyra/landmark matrix of size 71,952-by-2704, on a 24-core AMD Opteron 6168 system with 128GB of RAM. This experiment required a system with a large amount of RAM, since MATLAB required 47GB of space for method 2.

end, just before return its results to the MATLAB user. As shown in Table II, the performance difference between method 2 and the other methods is extreme, because the other three methods never construct `Q` at all.

2.2.8 *Other factorization methods.* The `FACTORIZE` package also relies on LDL^T factorization for matrices that are symmetric indefinite. The sparse `ldl` in MATLAB relies on MA57 [Duff 2004].

MATLAB does not provide a complete orthogonal decomposition (COD), but one can be written that relies on either the dense or sparse QR factorization methods in MATLAB. The former is straight-forward (see Section 2.2.6). The latter requires a QR factorization with an efficient representation of the matrix `Q`. MATLAB uses SuiteSparseQR for its sparse QR factorization, but MATLAB does not provide access to the sparse Householder-vector representation for `Q`, which can easily be an order of magnitude sparser than the explict matrix `Q`, or even sparser. To access this feature, the `FACTORIZE` package uses the mexFunction interface to SuiteSparseQR (`spqr`), available at http://www.suitesparse.com, rather than the built-in SuiteSparseQR. The sparse COD in the `FACTORIZE` package is optional, and the package gracefully skips the sparse COD if `spqr` is not available. For best results with sparse rank-deficient matrices, the user is encouraged to install all of SuiteSparse.

2.2.9 *Best-of-breed methods for LU, Cholesky, and QR.* Table III lists the best techniques for the three primary factorizations for both the dense and sparse cases. Not listed are the LDL^T, SVD, COD factorization methods. None of the methods listed in the table are trivial or obvious, even to the MATLAB expert.

2.3 Factorization methods used by built-in MATLAB functions

A sub-optimal but commonly-used method that uses dense LU factorization is `[L,U,P]=lu(A); x=U\(L\(P*b))`. In Figure 1, this is method 2 for the factorization and method 2b for the solve step.

This sub-optimal technique does not exploit the Cholesky or LDL^T factorizations, which cut the time roughly in half for symmetric matrices. Its solve step relies on a dense permutation matrix, which is very slow to use and takes a lot of memory. In spite of these drawbacks, this sub-optimal method can be found in built-in MATLAB functions (`md2c` and `@idss/ss2ss` in the System Identification Toolbox, and `@umat/inv` in the Robust Control Toolbox, for example).

A better method is used in four of the eight MATLAB ODE solvers, where a permutation vector is used instead (`[L,U,p]=lu(A,'vector'); x=U\(L\b(p))`,

method	sparse?	most efficient code for `x=A\b` and `y=c/A`

LU	no	Assuming `b` and `c` are dense vectors or matrices. If sparse, they must be converted to dense matrices since `linsolve` only operates on dense matrices.
		`[L, U, p] = lu (A, 'vector') ;`
		`opL.LT = true ;`
		`opU.UT = true ;`
		`opUT.UT = true ;`
		`opUT.TRANSA = true ;`
		`opLT.LT = true ;`
		`opLT.TRANSA = true ;`
		`x = linsolve (U, linsolve (L, b (p,:), opL), opU) ;`
		`y = (linsolve (L, linsolve (U, c', opUT), opLT))' ; y (:,p) = y ;`

LU	yes	`b` and `c` can be sparse or dense.
		`[L, U, P, Q, R] = lu (A) ;  % R is a diagonal scaling matrix`
		`x = Q * (U \ (L \ (P * (R \ b)))) ;`
		`y = ((((c * Q) / U) / L) * P) / R ;`

Cholesky	no	Assuming `b` and `c` are dense.
		`[R, g] = chol (A) ;  % R is upper triangular`
		`opU.UT = true ;`
		`opUT.UT = true ;`
		`opUT.TRANSA = true ;`
		`x = linsolve (R, linsolve (R, b, opUT), opU) ;`
		`y = linsolve (R, linsolve (R, c', opUT), opU)' ;`

Cholesky	yes	`b` and `c` can be sparse or dense.
		`[L, g, P] = chol (A, 'lower') ;  % L is lower triangular`
		`x = P * (L' \ (L \ (P' * b))) ;`
		`y = (P * (L' \ (L \ (P' * c'))))' ;`

QR	no	Assuming A has more rows than columns, and both `b` and `c` are dense. This is the faster optional method for full-rank methods; the default is QR with column pivoting via the COD listed in Section 2.2.6.
		`[Q, R] = qr (A,0) ;`
		`opU.UT = true ;`
		`opUT.UT = true ;`
		`opUT.TRANSA = true ;`
		`x = linsolve (R, Q'*b, opU) ;`
		`y = (Q * linsolve (R, c', opUT))' ;`

QR	yes	Assuming A has more rows than columns. `b` and `c` can be sparse or dense. Uses the corrected semi-normal equations with one step of iterative refinement.
		`[m, n] = size (A) :`
		`[~, R, p] = qr (A, sparse (m,0), 0) ;`
		`P = sparse (p, 1:n, 1) ;`
		`x = P * (R \ (R' \ (P' * (A' * b)))) ;`
		`e = P * (R \ (R' \ (P' * (A' * (b - A * x))))) ;`
		`x = x + e ; % computing y to minimize norm (y*A-c) is analogous`

Table III. The most efficient MATLAB code for the three primary factorization methods.

when `A` is dense). This is method 1 for the factorization (the best LU) coupled with method 1b, a sub-optimal solve step. These four functions correctly use `lu` for sparse matrices, allowing for a fill-reducing permutation and a row-scaling matrix `R`. However, they do not use the full power of backslash, such as using `chol` or `ldl`, which are much faster than `lu` for symmetric matrices.

The `condest` function also uses method 1, and a method with similar performace as method 1b (`x=U\(L\b)`, since `condest` can ignore the permutation `p`). Like the ODE functions, `condest` does not exploit symmetry via `chol` or `ldl`, and thus computing `condest(A)` for a sparse symmetric positive definite matrix `A` is many times slower than it could be.

The `sptarn` function in the MATLAB PDE toolbox is slightly more sophisticated, but still limited. It relies on its own backslash mimic, which uses either `chol` or `lu`, depending on the matrix. However, it constrains `lu` with an inferior fill-reducing permutation, and restricts it to use GP, which can be slower than the one relied upon by backslash (UMFPACK). This performance hint appears in `help lu`. This same problem occurs in `fzmult` in the Optimization toolbox.

The ODE solvers `bvp4c` and `bvp5c` use `lu` for a sparse matrix in a way that prohibits `lu` from exploiting any fill-reducing ordering at all. This can be very inefficient if fill-in is excessive. Like `sptarn`, these two methods use `lu` in a style that prohibits the use of UMFPACK.

The `eigs` function comes closest to the efficiency of backslash, but it requires a great deal of code to get it right even though `eigs` only needs to consider the case when the matrix is square.

These built-in MATLAB functions use very different factorization techniques. Some are better than others, but none are as fast or as flexible as backslash. What these functions really need is a backslash whose factorization can be easily reused. Code duplication is also a concern, since the same functionality is duplicated in many places with differing degrees of success.

2.4 Abusing the MATLAB INV

Even the sub-optimal factorizations discussed in the previous section can be difficult for the naive MATLAB user to master or use, which contributes to the prevalent misuse of the MATLAB `inv` function. Using `inv` is trivial in MATLAB: `S=inv(A)` computes the inverse of A, and `x=S*b` is a very simple way to use (or reuse) `S` to compute `x=A\b`.

Textbooks refer to the inverse of A in many formulas: $x = A^{-1}b$ is the solution to $Ax = b$, and $S = D - BA^{-1}C$ is a common way to express the Schur complement S, for example. Although textbook authors do not intend for the reader to compute the explicit inverse (or they shouldn't!) the naive user often simply translates these formulas directly into MATLAB expressions with the `inv` function.

There are many problems with this naive use of `inv`, of course. It can be hopelessly inaccurate, and for sparse matrices it can be impossible to compute since `inv(A)` is typically completely nonzero. MATLAB provides a warning in its M-file editor that flags the use of `x=inv(A)*b`, directing the user to use backslash instead. However, the MATLAB editor does not tell the user how to efficiently reuse a factorization.

Misuse of `x=inv(A)*b` is very common. This author recently conducted a study

to determine the 500 most frequently used functions in MATLAB [Davis 2011b]. Every user-contributed submission on MATLAB Central as of March 2010 was downloaded and parsed to determine which built-in functions were used, and how often they were used in each submission. The `inv` function was found in 554 of the 9,498 submissions (about 6%), and appeared in a total of 2,407 times in those 554 submissions. This places `inv` as the 160th most frequently-used function in MATLAB. There are a few cases where `inv` can be properly used, such as when specific entries of the inverse are required. However, a spot check of about a dozen of the 554 submissions that use `inv` showed that none fell into that category. They were all misuses of `inv`.

For comparison, `sparse` is the 172nd most-used function, and `lu` is merely the 383rd. The `qr` function is slightly more common (ranked 355th), whereas `chol` is ranked 409th. The `inv` function is used more frequently than any of these other functions. Clearly, `inv`-abuse is a serious and common problem for MATLAB users.

2.5 A gap in MATLAB functionality

To summarize, no method is clearly the best for MATLAB users:

(1) **backslash:** simple to use, fast, and accurate, but very slow if you have multiple linear systems to solve. Its syntax is not as as clear as `inv`. Consider the Schur complement, where $D - BA^{-1}C$ translates directly into `D-B*inv(A)*C` or the more esoteric but numerically superior expressions `D-B*(A\C)` or `D-(B/A)*C`.

(2) **LU, QR, Cholesky, and LDL^T:** fast and accurate, but very difficult to use. You will need to pull out your linear algebra textbook and be prepared to do some benchmarking to find the most efficient method. This author wrote the sparse versions of three of these functions (LU, QR, and Cholesky [Chen et al. 2008; Davis 2004; 2011a]) and even he has trouble remembering the best way to use them via MATLAB. What is worse is that new versions of MATLAB often result in different optimal methods for using these factorizations, as new factorization methods appear. This occurred most recently with the introduction of the sparse LDL^T in 2008 (MA57 [Duff 2004]) and the new sparse multifrontal QR factorization [Davis 2011a] in 2009.

(3) **`inv`:** The statement `x=inv(A)*b` is easy to write, but should always be avoided. It is commonly misused by MATLAB users, probably because the syntax is very simple and matches the mathematics, and because it is easy to reuse (*misuse*, to be more precise) for multiple right-hand sides.

3. FACTORIZE: AN OBJECT-ORIENTED LINEAR SYSTEM SOLVER

The solution to this problem is to encapsulate the full suite of linear system solvers in MATLAB into a single object-oriented solver called `FACTORIZE`. This object also extends backslash by improving how rank-deficient systems are handled, incorporating a complete orthogonal decomposition, and the singular value decomposition.

3.1 An overview of the FACTORIZE method

Operator overloading in the object-oriented design of the `FACTORIZE` package makes it extremely easy to use for the MATLAB end-user. Below are a few examples of its use.

```
F = factorize (A) ;  % returns a factorization object
x = F\b ;            % same as x=A\b, but only doing the forward/backsolve
y = F\c ;            % same as y=A\c, reusing the factorization of A
x = b/F ;            % same as x=b/A, but only doing the forward/backsolve
z = F'\d ;           % reuses the factors to solve the transposed system
S = inverse (F) ;    % factorized representation of the inverse (requires no work)
x = S*b ;            % same as x=F\b
c = S (1:3,2:3) ;    % returns entries from inv(A), doesn't compute all of inv(A)
c = condest (F) ;    % same as condest(A), but reuses the factorization
```

Consider the complexity of the best dense LU factorization in the first row of Table III, and the sub-optimal method used in `ode15i` (`[L,U,p]=lu(A,'vector'); x=U\(L\b(p))`). The method `F=factorize(A); x=F\b` is just as fast as the best method in Table III, yet far easier to use than either of these alternatives.

The `inverse` function allows for a direct translation of the many textbook mathematical expressions that use A^{-1}. For example, $x = A^{-1}b$ can be elegantly written as `x=inverse(A)*b`. This statement does *not* compute the inverse, but does the right thing by factorizing `A` and solving the linear system using that factorization. Likewise, the mathematical equation $D - BA^{-1}C$ for the Schur complement translates directly into `D-B*inverse(A)*C`, which is both easy to read and computationally efficient.

3.2 Implementation of the FACTORIZE method

The `factorization` object `F` is constructed via an M-file that mimics the backslash operator, using all of Table III and several other techniques. If the matrix is rectangular, a QR factorization of `A` or `A'` is computed, whichever is tall and thin. This allows `F\b` to return a minimum 2-norm solution for under-determined systems, rather than the basic solution found by `x=A\b`. Both result in a low residual, but a minimum 2-norm is solution unique if `A` has full rank and this is sometimes preferable to a non-unique basic solution.

The next step is the same as backslash. Namely, if the matrix is square, symmetric, with an all-real nonzero diagonal, `chol` is attempted. If this fails, or if the condition on the diagonal does not hold, `ldl` is used. If these conditions do not hold, or if `chol` and `ldl` fail, `lu` is used.

If any of these solvers report that `A` is rank-deficient (or nearly so), backslash simply reports a warning and returns whatever result it found. Backslash uses `qr` factorization with approximate rank-detection for rectangular matrices, and thus it can find a basic solution to under-determined systems. However, backslash does not attempt this if the matrix is square. The `FACTORIZE` method uses a more reliable technique for rank-deficient matrices (both square and rectangular, sparse and dense): a complete orthogonal decomposition (COD). This enables the `FACTORIZE` package to find better solutions to singular problems.

If `A` is a matrix, `S=inverse(A)` is the same as `F=factorize(A)` followed by `S=inverse(F)`. A factorization `F` is computed, and then `S` is merely flagged as being a factorization of the inverse, swapping the roles of the `\` and `*` operators. The factorizations `F` and `S` are otherwise identical. When `S*b` is to be computed in a user's code, the `FACTORIZE` package computes `A\b` using the previously computed factorization of `A`.

3.3 The advantages of operator overloading

Operator overloading is a very useful technique for extending a previous code to handle new kinds of computations. For example, selecting a good factorization should be done in one place, and then reused in other codes that need a factorization. These other codes should not care how it is done or even which factorization is used. A prime example is the MATLAB `normest1` function, which `condest` relies upon to compute an estimate of the 1-norm condition number, $||A||_1||A^{-1}||_1$.

The statement `c=normest1(A)` computes the estimate of $||A||_1$ by relying only on two computations with the matrix `A`: `y=A*x` and `y=A'*x`. If `isa(A,'double')` is true, then `normest1` simply performs `y=A*x` and `y=A'*x`, treating `A` as a matrix. Otherwise, it treats `A` as an operator and calls the function handle `A` to perform these two computations. `condest` uses this to compute an estimate of $||A^{-1}||_1$.

However, consider the computation `S=inverse(A); z=normest1(S)`. The computation in `normest1` uses `S*x` and `S'*x` rather than calling `S` as a function handle, since `isa(S,'double')` is true for the `factorization` object `S`. Since operator overloading for `S*b` computes `A\b` by reusing the factorization, and since `S'*b` becomes `A'\b`, these computations do the right thing, without computing the inverse. The `normest1` function treats `S` just like a matrix, "unaware" that is it being adapted for use by an object-oriented approach to compute $||A^{-1}||_1$. If `condest` were to be rewritten to use the `FACTORIZE` package, the complicated code in `normest1` for working with a function handle could be discarded, and `normest1` would be simpler and faster. It would also automatically adapt to future updates to the factorization methods in MATLAB with no changes to its code.

The MATLAB expression `norm(A,1)*normest1(inverse(A))` uses the `FACTORIZE` package to computes an estimate of $\kappa_1(A) = ||A||_1||A^{-1}||_1$ without any changes to the built-in `normest1`. The expression is faster than `condest(A)` and yet it looks just like the mathematical definition, a strong indication that an object-oriented style of handling factorizations and implicit inverses is a powerful technique for writing code that is fast, accurate, and easy to read.

The new method is yet more efficient if the factorization needs to be reused outside the `condest` computation. Suppose the user wants to compute `x=A\b` followed by `s=condest(A)` (the built-in functions `ode15s` and `ode23t` in MATLAB are two examples). The matrix is factorized twice, which is wasteful. Instead, this can be written as `F=factorize(A); x=F\b; s=condest(F)`, which computes the factorization only once. For large square unsymmetric matrices, the total time is cut in half. If `A` is a dense symmetric positive definite matrix, the time is cut by nearly a factor of 3, because `condest(A)` uses an LU factorization, whereas `condest(F)` reuses the Cholesky factorization computed by `F=factorize(A)`.

3.4 Using alternative factorizations

A second string argument to the `FACTORIZE` function tells it to use a specific method or strategy rather than the default, which is to mimic backslash. The options are `'lu'`, `'qr'`, `'chol'`, `'cod'`, `'ldl'`, and `'svd'`, or a modification of the default backslash strategy (`'symmetric'` and `'unsymmetric'`, which speed up the backslash tests by skipping the test for symmetry). There is no mechanism for providing these hints to backslash.

3.5 Using the singular value decomposition

The built-in functions `norm`, `cond`, `rank`, `null`, `orth`, and `pinv` all rely upon the SVD. They each compute the SVD, use it, and then discard it. The SVD is extremely costly to compute and should not be so lightly discarded. Suppose a user wishes to compute the following. The SVD is computed seven times.

```
[U,S,V] = svd (A) ;
nrm = norm (A) ;
c = cond (A) ;
r = rank (A) ;
Z = null (A) ;
Q = orth (A) ;
C = pinv (A) ;
x = C*b ;
```

Code that relies upon the `FACTORIZE` package is listed below. It is nearly identical and remarkably simple, but it computes the SVD just once. For large matrices, it is close to 6 times faster than the non-object-oriented code listed above.

```
F = factorize (A, 'svd') ;
[U,S,V] = svd (F) ;         % retrieve the factorization from F
nrm = norm (F) ;
c = cond (F) ;
r = rank (F) ;
Z = null (F) ;
Q = orth (F) ;
C = pinv (F) ;
x = C*b ;
```

4. SUMMARY

The `FACTORIZE` package allows the MATLAB user to write simple code that is more elegant than the code it replaces (consider the `normest1` example, or the Schur complement). Code that relies on the `FACTORIZE` package can be faster for large matrices even if the factorization is not reused, since (like backslash) it selects among a wide range of factorization methods, rather than choosing among a few (consider `condest`). Code performance also increases if the factorization can be reused. The `inverse` method based on the `FACTORIZE` package is far superior to the much-abused `inv`, while being just as simple to use.

Judging from how MATLAB experts exploit the many factorization methods in MATLAB (in built-in functions written by The MathWorks), wrapping the best methods into an easy-to-use `factorization` object is a powerful way to encourage the use of most efficient factorization methods in MATLAB.

In addition to appearing as Algorithm 9xx of the ACM, the `FACTORIZE` package is available at http://www.suitesparse.com and at MATLAB Central,[2] where it was selected as a "Pick of the Week" by The MathWorks [Doke 2009]. The code includes a thorough demo that illustrates how to use the object and some of the theory behind solving different kinds of linear systems and least-squares problems via direct factorizations. A complete test suite is included that tests every line of code for accuracy, error-handling, and performance.

[2]http://www.mathworks.com/matlabcentral/fileexchange/24119

REFERENCES

Amestoy, P. R., Davis, T. A., and Duff, I. S. 1996. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl. 17,* 4, 886–905.

Amestoy, P. R., Davis, T. A., and Duff, I. S. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 381–388.

Anderson, E., Bai, Z., Bischof, C. H., Blackford, S., Demmel, J. W., Dongarra, J. J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. C. 1999. *LAPACK Users' Guide*, 3rd ed. SIAM, Philadelphia, PA.

Chen, Y., Davis, T. A., Hager, W. W., and Rajamanickam, S. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Softw. 35,* 3, 1–14.

Davis, T. A. 2002. Algorithm 832: UMFPACK V4.3, an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30,* 2, 196–199.

Davis, T. A. 2004. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30,* 2, 165–195.

Davis, T. A. 2006. *Direct Methods for Sparse Linear Systems.* SIAM, Philadelphia, PA.

Davis, T. A. 2011a. Algorithm 915: SuiteSparseQR, a multifrontal multithreaded sparse QR factorization package. *ACM Trans. Math. Softw. 38,* 1.

Davis, T. A. 2011b. *MATLAB Primer*, 8th ed. Chapman & Hall/CRC Press, Boca Raton.

Davis, T. A. and Duff, I. S. 1999. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Softw. 25,* 1, 1–19.

Davis, T. A., Gilbert, J. R., Larimore, S. I., and Ng, E. G. 2004a. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 377–380.

Davis, T. A., Gilbert, J. R., Larimore, S. I., and Ng, E. G. 2004b. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 353–376.

Davis, T. A. and Hu, Y. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw. 28,* 1.

Davis, T. A. and Palamadai Natarajan, E. 2010. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw. 37*, 36:1–36:17.

Doke, J. 2009. Pick of the week: Don't let that INV go past your eyes; to solve that system FACTORIZE. http://blogs.mathworks.com/pick/2009/06/26/dont-let-that-inv-go-past-your-eyes-to-solve-that-system-factorize/.

Duff, I. S. 2004. MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw. 30,* 2, 118–144.

Gilbert, J. R., Moler, C., and Schreiber, R. 1992. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl. 13,* 1, 333–356.

Gilbert, J. R. and Peierls, T. 1988. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput. 9*, 862–874.

Golub, G. H. and Van Loan, C. 1989. *Matrix Computations*, 2nd ed. Baltimore, Maryland: Johns Hopkins Press.

Received Month Year; revised Month Year; accepted Month Year