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

TIMOTHY A. DAVIS
University of Florida

The MATLAB™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 $Ax = b$ or 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. 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`. The latter 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 reusing a factorization in MATLAB very easy to do, even for the naive MATLAB user. Section 2 provides a strong motivation for the `factorize` method presented in Section 3. Code availability and concluding remarks are given in Section 4.

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

2. MOTIVATION

The MATLAB backslash is a powerful method, but it has its weaknesses. The many factorization methods in MATLAB provide an alternative, but these are difficult to use. The unfortunate prevalence of `x=inv(A)*b` illustrates yet another motivation for the object-oriented `factorize` method.

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`.

2.2 The many factorization methods in MATLAB

In spite of these drawbacks, backslash remains a powerful and general-purpose operator that is adequate 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. Table I lists the best techniques for just three of the primary factorizations. None of these methods are trivial or obvious, even to the MATLAB expert.

A sub-optimal but commonly-used method that uses dense LU factorization is `[L,U,P]=lu(A); x=U\(L\(P*b))`. For dense unsymmetric matrices, the factorization step uses 50% more memory than the best method in Table I, and its

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

LU	no	`[L, U, p] = lu (A, 'vector') ;`
		`P = sparse (1:size(A,1), p, 1) ;`
		`opL.LT = true ;`
		`opU.UT = true ;`
		`opUT.UT = true ;`
		`opUT.TRANSA = true ;`
		`opLT.LT = true ;`
		`opLT.TRANSA = true ;`
		`x = linsolve (U, linsolve (L, full (P*b), opL), opU) ;`
		`y = (P' * linsolve (L, linsolve (U, full (c'), opUT), opLT))' ;`

LU	yes	For some matrices, `[L,U,P,Q,R]=lu(A)` is faster, where `R` is a diagonal scaling matrix.
		`[L, U, P, Q] = lu (A) ;`
		`x = Q * (U \ (L \ (P * b))) ;`
		`y = (P' * (L' \ (U' \ (Q' * c'))))' ;`

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

Cholesky	yes	`[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.
		`[Q R] = qr (A,0) ;`
		`opU.UT = true ;`
		`opUT.UT = true ;`
		`opUT.TRANSA = true ;`
		`x = linsolve (R, Q' * full (b), opU) ;`
		`y = (Q * linsolve (R, full (c'), opUT))' ;`
		`% (where y minimizes norm (y*A-b))`

QR	yes	Assuming A has more rows than columns.
		`n = size (A,2) ;`
		`P = sparse (colamd (A), 1:n, 1) ;`
		`R = qr (A*P, 0) ;`
		`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-b) is analogous)`

Table I. The most efficient MATLAB code for the three primary factorization methods (as of R2011a).

forward/backsolve step is three times slower. This sub-optimal technique does not exploit the Cholesky or LDL^T factorizations. In spite of these drawbacks, this sub-optimal method can be found in built-in MATLAB (R2011a) functions (`md2c` and `@idss/ss2ss` in the System Identification Toolbox, and `@umat/inv` in the Robust Control Toolbox, for example).

A similar method is used in four of the eight MATLAB ODE solvers, except that a permutation vector is used instead (`[L,U,p]=lu(A,'vector'); x=U\(L\b(p))`, when `A` is dense). This is better, but its forward/backward solve phase still takes twice the time as the method in Table I. This is also true of `condest`, which uses `x=U\(L\b)`. These functions correctly use `lu` for sparse matrices, allowing for a fill-reducing permutation. 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. 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 an older sparse LU [Gilbert and Peierls 1988] that can be slower than the one relied upon by backslash (UMFPACK, [Davis and Duff 1999; Davis 2004]). This performance hint appears in `help lu`.

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 wildly different 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.

2.3 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. A study was recently conducted 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.

2.4 A gap in MATLAB functionality

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

—**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`.

—**LU, QR, Cholesky, and LDL^T:** fast and accurate, but very difficult to use. You will need to pull out your linear algebra textbook [Golub and Van Loan 1989] 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.

—`inv`**:** The statement `x=inv(A)*b` is easy to write, but should always be avoided. It is commonly misused by MATLAB users.

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 (optionally, at user discretion) the singular value decomposition.

3.1 An overview of the FACTORIZE method

The object-oriented design of the `factorize` method makes it extremely easy to use for the MATLAB end-user, via operator overloading. 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) ;    % just sets a flag, otherwise the S is the same as F
x = S*b ;            % same as x=F\b
c = S (1:3,2:3) ;    % returns entries from inv(A), doesn't compute all 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 I, 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 I, 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 I 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 unique if `A` has full rank and thus sometimes preferable to a 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. With luck, the approximate rank-detection of `qr` allows it find a basic solution to the under-determined system, but backslash does not attempt this if the matrix is square.

The `factorize` method uses a more reliable technique for rank-deficient matrices, namely, a complete orthogonal decomposition (COD) [Golub and Van Loan 1989]. If A has rank r, the COD is $URV = A$, where R is r-by-r, upper triangular, and well-conditioned if the rank of A is well-defined. The matrices U and V are orthogonal. Suppose A is m-by-n with $m \geq n$. For the dense case, if A has full rank, V is a permutation matrix arising from column 2-norm pivoting. In the sparse case for a full-rank A, V is the fill-reduction permutation. If A is rank-deficient this first QR factorization leaves R in upper trapezoidal form, so it is followed by another factorization that places it in upper triangular form, and the second orthogonal factor is multiplied into V to obtain the decomposition $URV = A$. If A has more columns than rows ($m < n$), this algorithm is applied to A^T, and the results transposed and permuted to obtain $URV = A$ with R upper triangular.

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 computed, the code 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. `normest1` is oblivious that is it being adapted for use by an object-oriented approach. The function handle feature of `normest1` is not needed for an object.

Thus, the MATLAB expression `norm(A,1)*normest1(inverse(A))` 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 both fast 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; 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 `factorization` object 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` method and the `factorization` object it creates allow for simple code that can be more elegant than the code it replaces (consider the `normest1` example, or the Schur complement). Code that relies on the `factorization` object is 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 `factorization` object 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^{TM}), 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 its availability as Algorithm 9xx of the ACM, the `factorize` package is available on MATLAB Central,[1] where it was selected as a "Pick of the Week" by The MathWorks, Inc. [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.

REFERENCES

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

Davis, T. A. 2011a. Algorithm 9xx: SuiteSparseQR, a multifrontal multithreaded sparse QR factorization package. *ACM Trans. Math. Softw.*. to appear.

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.

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

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