Notes for Math 221, Lecture 3, Sep 10, 2024

To summarize our approach to understanding accuracy in the face of
roundoff (or other) error, our goal will be to prove that algorithms
are "backward stable". For example if our algorithm for computing the
scalar function f(x) computes
  alg(x) = f(x + delta) ~ f(x) + f'(x)* delta
where delta is "small", then relative error can be approximated by
  | (alg(x) - f(x)) / f(x) | <= |f'(x)* delta / f(x)|
    = |f'(x)*x/f(x)| * |delta/x|
Def: The factor kappa(x) = |f'(x)*x/f(x)| is the condition_number
of the function f evaluated at x.

Here is a simple geometric interpretation of kappa(x) in this scalar
case: When is kappa(x) infinite? If f is a smooth function with
bounded f', kappa(x) is infinite when f(x)=0. How is kappa(x)
related to the distance from x to the nearest zero xhat of f,
i.e. f(xhat)=0? If f is smooth and f(x) small, one step of Newton
will estimate xhat for us:
  xhat ~ x - f(x)/f'(x) or
  x - xhat ~ f(x)/f'(x) or, computing the relative distance,
  | (x - xhat)/x | ~ | f(x)/(x*f'(x)) | = 1/kappa(x)
I.e. the condition number is close to the reciprocal of the relative
distance to the nearest ``infinitely hard'' problem.
|
We want to use the same approach for the problem of solving A*x=b,
including this geometric interpretation of the condition number,
but where we get (A+Delta)*xhat = b instead, where Delta is "small"
compared to A, or the eigenproblem A*x = lambda*x, but get
   (A+Delta)*xhat = lambdahat*xhat
where again Delta is "small" compared to A.

To formalize the notion of "small", we need to understand vector and
matrix norms. In both cases, we may say
  we want to compute x = f(A), but get xhat = alg(A) = f(A+Delta)
So to bound the error, we write
  error = xhat - x = alg(A) - f(A) = f(A+Delta) - f(A)
Assuming Delta is "small", and taking the first term of the Taylor
expansion, we get
  error ~ J_f(A) * Delta where J_f(A) is the Jacobian of f
If A and x were scalars, we could take absolute values and get an
absolute error bound:
   |error| <~ |J_f(A)| * |Delta|
and proceed as above.

In the cases most relevant to linear algebra, A and x are not scalars but matrices and vectors (with obvious generalizations, depending on the problem). To generalize this error bound, we need to generalize absolute values, which leads us to norms.

Goals: Matrix and vector norms
    Singular Value Decomposition
    Condition numbers for Ax=b

Matrix and vector norms

Def norm: Let B be linear space R^n (or C^n).
    It is normed if there is a function ||.||:B -> R s.t.
     (1) ||x|| >= 0, and ||x|| = 0 iff x=0
        (positive definiteness)
     (2) ||c*x|| = |c|*||x|| (homogeneity)
     (3) ||x+y|| <= ||x|| + ||y|| (triangle inequality)
Examples: p-norm = ||x||_p = (sum_i |x_i|^p)^(1/p), for p >= 1
      Euclidean norm = 2-norm = ||x||_2
         Note: x real => ||x||_2^2 = sum_i x_i^2 = x^T * x
      infinity-norm = ||x||_inf = max_i |x_i|
      C-norm = ||C*x|| where C has full column rank
         (Question 1.5)

Lemma (1.4): all norms are equivalent
    i.e. given any norm_a(x) and norm_b(x), there are positive
       constants alpha and beta such that
        alpha*norm_a(x) <= norm_b(x) <= beta*norm_a(x)
    (proof: compactness)

This lemma is an excuse to use the easiest norm to prove later results.

Def: matrix norm (vector norm on mxn vectors)
   (1) ||A|| >= 0, and ||A|| = 0 iff A=0
      (positive definiteness)
   (2) ||c*A|| = |c|*||A|| (homogeneity)
   (3) ||A+B|| <= ||A|| + ||B|| (triangle inequality)

Ex: max norm = max_ij |A_ij|
  Frobenius norm = (sum_ij |A_ij|^2)^(1/2)

Def: operator norm

norm(A) = max_{nonzero x} norm(A*x)/norm(x)

Lemma (1.6): The operator norm is a matrix norm.
proof: Question 1.15

Lemma (1.7): if norm(A) is an operator norm, there exists x such that
norm(x)=1 and norm(A*x)=norm(A).
proof: norm(A) = max_{nonzero x} norm(A*x)/norm(x)
= max_{nonzero x} norm(A * x/norm(x))
= max_{unit vector y} norm(A * y)
y attaining this maximum exists since norm(A*y) is a continuous
function of y on compact (closed and bounded) set = unit ball

Now we turn to orthogonal and unitary matrices, which we need for
the SVD.

Notation: Q^* = conj(Q^T),
sometimes we write Q^H (H stands for "Hermitian",
since a matrix satisfying A=A^H is called Hermitian)
Def: orthogonal matrix: Q square, real and inv(Q) = Q^T
unitary matrices: Q square, complex, and inv(Q) = Q^*
(For simplicity, we state results for real case, but all extend to
complex.)

Fact: Q orthogonal <=> Q^T*Q = I <=> all columns mutually orthogonal
and are unit vectors, Q*Q^T = I implies same about rows

Fact: norm(Q*x,2)=norm(x,2) - aka Pythagorean theorem
Proof: norm(Q*x,2)^2 = (Q*x)^T*(Q*x) = x^T*Q^T*Q*x = x^T*x
= norm(x,2)^2

Fact: Q and Z orthogonal => Q*Z orthogonal
Proof: (Q*Z)^T * (Q*Z) = Z^T*Q^T * Q*Z = Z^T * I * Z = I

Fact: If Q is m x n, with n<m, and Q^T * Q = I_n, then you
can add m-n columns to Q to make it m x m and orthogonal, so
Q^T*Q=I_m (there may be infinitely many ways to this, we can
give a proof later, but this is a useful fact in some proofs)

Lemma (most proofs in homework, Q1.16)
(1) norm(A*x) <= norm(A)*norm(x) for vector and its operator norm
(2) norm(A*B) <= norm(A)*norm(B) for operator norm
(3) norm(Q*A*Z,2) = norm(A,2) if Q, Z orthogonal

(4) norm(Q,2)=1
(5) norm(A,2) = sqrt(lambda_max(A^T*A))
(6) norm(A^T,2) = norm(A,2)

proof of (5) only:
norm(A,2) = max_{nonzero x} norm(A*x)/norm(x)
= max_{nonzero x} sqrt((A*x)^T*(A*x) / sqrt(x^T*x))
= sqrt (max_{nonzero x} (A*x)^T*(A*x) / (x^T*x))
= sqrt (max_{nonzero x} x^T*A^T*A*x / x^T*x)

Use the fact that A^T*A is symmetric and so has eigenvalue decomposition:
A^T*A*q_i = l_i * q_i
where l_i real, q_i real, unit, orthogonal.
Let Q = [q_1,...,q_n] and Lambda = diag(l_1,...,l_n), so that
A^T*A*Q = Q*Lambda, and A^T*A = Q*Lambda*inv(Q) = Q*Lambda*Q^T
Continuing from above:

norm(A,2)
= sqrt (max_{nonzero x} x^T*Q*Lambda*Q^T*x / x^T*x)
= sqrt (max_{nonzero x} x^T*Q*Lambda*Q^T*x / x^T*Q*Q^T*x)
= sqrt (max_{nonzero y} y^T*Lambda*y / y^T*y)
where y = Q^T*x
= sqrt (max_{nonzero y} sum_i y_i^2 l_i / sum_i y_i^2)
<= sqrt (max_{nonzero y} l_max * sum_i y_i^2 / sum_i y_i^2)
= sqrt (l_max), attained by choosing y_max = 1, rest 0

SVD = Singular Value Decomposition
The SVD is a Swiss Army Knife of numerical linear algebra:
Given the SVD of A, one can easily
solve Ax=b (when A is square),
solve an overdetermined or underdetermined least squares problem
with rectangular A (whether A is full rank or not),
compute the eigenvalues and eigenvectors of A*A^T and A^T*A, or
compute eigenvalues and eigenvectors of A if A is symmetric.
Furthermore, one can use the SVD to write down error bounds for all these problems. It is more expensive to compute than other algorithms specialized for these problems, so it may not be the algorithm of first resort.

History: The first complete statement goes back to Eckart & Young in 1936. The first reliable algorithm was by Golub & Kahan in 1965, with faster ones since then, to be discussed in Chapter 5. Perhaps the

fastest algorithm appears in the prize-winning 2010 PhD thesis by
Paul Willems, which remains to be incorporated into LAPACK.
(We have found some numerical examples on which the algorithm fails
to be accurate enough, and have not managed to fix it yet, despite
conversations with the author. So learning about and testing this
algorithm is a possible (and challenging) class project.)

Thm. Suppose A = m x m, then there is an
orthogonal matrix U = [u(1),...,u(m)]
diagonal matrix Sigma = diag(sigma(1),...,sigma(m))
with sigma(1) >= sigma(2) >= ... >= 0
orthogonal matrix V = [v(1),...,v(m)]
with A = U*Sigma*V^T.
u(i) called left singular vectors
sigma(i) called singular values
v(i) called right singular vectors
More generally, if A is m x n with m > n, then
U m x m and orthogonal as before
V n x n and orthogonal
Sigma is m x n with same diagonal as before
When m > n, we sometimes write this as follows (the "thin SVD")
[u(1),...,u(n)] * diag(sigma(1),...,sigma(n)) * V^T
The same ideas work if A is m x n with m < n.

Geometric interpretation: Thinking of A as a linear mapping from R^n
to R^m, with the right orthogonal choice of bases
of R^n (i.e. columns of V) and R^m (i.e. columns of U)
then A is diagonal (i.e. Sigma):
A = U*Sigma*V^T => A*V = U*Sigma => A*v(i) = sigma(i)*u(i)

Proof: Induction on n:
Two base cases
n=1: Let U have the first column = A/norm(A,2), rest chosen in any
way that makes U orthogonal; Sigma(1,1) = norm(A,2), V = 1
A=0: Let U = I_m, Sigma = 0, V = I_n

Induction step (if A nonzero):
||A||_2 = max_{x: x neq 0} ||A*x||_2 / ||x||_2
= max_{x: ||x||_2 = 1} ||A*x||_2
Let v(1) be x attaining the max, sigma(1) = ||A||_2 = ||A*v(1)||_2
and u(1) = A*v(1) / ||A*v(1)||_2 .
Choose V = [v(1),Vhat] to be square & orthogonal
Choose U = [u(1),Uhat] to be square & orthogonal
Write Ahat = U^T*A*V

= [u(1)^T] * A * [v(1), Vhat]
[Uhat^T]
= [u(1)^T*A*v(1) u(1)^T*A*Vhat]
[Uhat^T*A*v(1) Uhat^T*A*Vhat]
= [sigma(1) A12]
[A21 A22]
Show A21 = 0 by definition of Uhat
Show A12 = 0 by definition of sigma(1) = ||A||_2
(use part (3) of above Lemma)
Apply induction to A22 = U_2*Sigma_2*V_2^T, so
A = U*Ahat*V^T = U * [sigma(1) 0] * V^T
[0 U_2*Sigma_2*V_2^T]
= U * [1 0] * [sigma(1) 0] * [1 0] * V^T
[0 U_2] [0 Sigma_2] [0 V_2^T]
= orthogonal * nonnegative_diagonal * orthogonal, as desired

The SVD has many useful properties; assume A is m x n with m >= n.

Fact 1: In the square nonsingular case,
we can use it to solve A*x=b with just O(n^2) more work:
Proof: X = inv(A)*b = inv(U*Sigma*V^T)*B = (V*inv(Sigma)*U^T)*b
But note: computing the SVD itself is expensive, O(n^3) (see Chap 5)
If all you want to do is solve A*x=b, Gaussian Elimination is
cheaper. On the other hand, we will see that the SVD is more
reliable when A is nearly singular, provides and error bound, and
even lets us "solve" A*x=b when A is exactly singular.

Fact 2: When m>n, we can solve the full rank least squares problem
argmin_x ||A*x-b||_2 as follows:
writing the thin SVD, A = U*Sigma*V^T with U m x n,
then x = V*inv(Sigma)*U^T*b , same formula as the square case
Proof: Write A = Uhat*Sigmahat*V^T where Uhat = [U,U'] is m x m and
Sigmahat = [Sigma; 0] is m x n. Then
|| A*x-b ||_2^2 = || Uhat*Sigmahat*V^T*x - b ||_2^2
= || Uhat^T* (") ||_2^2
= || Sigmahat*V^T*x - Uhat^T*b ||_2^2
= || [Sigma*V^T*x - U^T*b] ||^2
|| [- U'^T*b] ||_2
= || Sigma*V^T*x - U^T*b ||_2^2
+|| -U'^T*b ||_2^2
is clearly minimized by choosing x = V*inv(Sigma)*U^T*b
to zero out the term depending on x

Def: When A = U*Sigma*V^T is m x n, m >= n, and full rank, then
A^+ = V*inv(Sigma)*U^T is n x m, and is called the
Moore-Penrose pseudoinverse of A.

This is the most natural extension of the definition of "inverse"
to rectangular full-rank matrices. We can also use the SVD, and
an appropriately defined Moore-Penrose pseudoinverse to solve the
rank deficient least squares problems,
or underdetermined problem (m < n), as we describe later.
(Q 3.13 has more inverse-like properties of the pseudo-inverse).

Just to solve a least squares problem where you are not
worried about rank deficiency, the QR decomposition is cheaper.
On the other hand, we will see that the SVD is more reliable when
A is nearly singular.

Fact 3: If A symmetric with eigenvalues
Lambda = diag(lambda_1,...,lambda_n)
and orthonormal eigenvectors V = [v(1),...,v(n)],
then its SVD is
A = V*Lambda*V^T (done if all lambda_i >= 0)
= (V*D)*(D*Lambda)*V^T where D = diag(sign(lambda(i)))
= U*Sigma*V^T

Fact 4: Using the thin SVD, the eigenvalue decomposition of
A^T*A = (U*Sigma*V^T)^T*(U*Sigma*V^T) = V*Sigma^2*V^T

Fact 5: Using the thin SVD, the eigenvalue decomposition of
A*A^T = (U*Sigma*V^T)*(U*Sigma*V^T)^T = U*Sigma^2*U^T
(what happens if m>n?)

Fact 6: Let H = [0 A^T] be (m+n) x (m+n), assuming A is m x n
[A 0]
Then H has eigenvalues +- sigma(i)
and eigenvectors 1/sqrt(2)*[v(i) ; +- u(i)]
Proof: plug in A = U*Sigma*V^T

Fact 6 suggests that algorithms for the SVD and the symmetric
eigenproblem will be closely related (see Chap 5)

Fact 7: ||A||_2 = sigma(1), ||inv(A)||_2 = 1/sigma(n) and

Def: kappa(A) = sigma(1)/sigma(n) is called condition number of A
for reasons we will see shortly

Fact 8: Let S be the unit sphere in R^n. Then A*S is an ellipsoid
centered at the origin with principal axes sigma(i)*u(i)

Proof: suppose s = [s_1;...;s_n] where sum_i s(i)^2 = 1, and write
A*s = U*Sigma*V^T*s = U*Sigma*shat = sum_i u(i)*sigma(i)*shat(i)

(matlab demo a = randn(2,2), svddemo2; a = randn(3,3); svddemo3)

Fact 9: Suppose
sigma(1) >= ... >= sigma(r) > 0 = sigma(r+1) = ... = sigma(n).
Then A has rank r; the null-space of A is
span(v(r+1),...,v(n)), of dimension n-r,
and the range space of A is
span(u(1),...,u(r)), of dimension r

Fact 10: Matrix A_k of rank k closest to A in 2-norm is
A_k = sum_{i=1 to k} u_i*sigma(i)*v(i)^T = U*Sigma_k*V^T
where Sigma_k = diag (sigma(1) , ... , sigma(k), 0, ... 0)
and the distance is norm(A - A_k , 2) = sigma(k+1)
In particular, the distance to the nearest singular
(or non-full rank) matrix is sigma(n) = sigma_min.

Proof: easy to see that A_k has right rank, right distance to A;
need to show no closer one:
Suppose B has rank k, so null space has dimension n-k.
The space spanned by {v(1),...,v(k+1)} has dimension k+1.
Since the sum of the dimensions (n-k)+(k+1) > n, these two spaces
must overlap (can't have > n independent vectors in R^n).
Let h be unit vector in their intersection. then
norm(A-B,2) >= norm((A-B)*h,2) = norm(A*h,2)
= norm(U*Sigma*V^T*h,2) = norm(Sigma*V^T*h,2)
= norm(Sigma*[x(1),...,x(k+1),0,...,0]^T,2)
>= sigma(k+1)

(matlab demo: We use this idea that A_k approximates A to
demonstrate image compression, since an image is just a matrix
(of gray values, say).

load clown.mat, [U,S,V]=svd(X);
figure(1), clf, image(X), colormap('gray'), brighten(.5),
figure(2), clf, n = 0; for k=[1:10,20:10:200], n=n+1;
Xk=U(:,1:k)*S(1:k,1:k)*(V(:,1:k))'; ...

```
    err = norm(X-Xk)/norm(X); compr = k*(200+320)/(200*320); ...
    figure(2), image(Xk), colormap('gray'), brighten(.5),...
    title(['k= ',int2str(k),' err= ', num2str(err),' compression= ', ...
      num2str(compr)]), ...
   pause, end
```

To see how the error and compression depend on the rank we can plot:

```
figure(3), s = diag(S);
semilogy(1:200,s/s(1),'r',1:200,(1:200)*(200+320)/(200*320),'b'),
title('Error in red, compression in blue'), xlabel('rank'), grid
```

(Note: jpeg compression algorithm uses a similar idea, on subimages)

Now we start using this material to analyze the condition number
for matrix inversion and solving Ax=b: If A (and b) change
a little bit, how much can inv(A) (and x=inv(A)*b) change?

If |x|<1, recall that 1/(1-x) = 1+x+x^2+x^3+...
Now generalize to matrices:

Lemma: If operator norm(X)<1, then I-X is nonsingular and
    inv(I - X) = sum_{i>=0} X^i and norm(inv(I-X)) <= 1/(1-norm(X))

proof: Claim I + X + X^2 + ... converges:
 norm(X^i) <= norm(X)^i -> 0 as i increases
 so by equivalence of norms there is some C>0 such that
     max_{kj} | (X^i)_kj | <= C* norm(X^i) <= C* norm(X)^i
 and so kj-th entry of I + X + X^2 ... is dominated by
 a convergent geometric series and so converges.
 Claim (I - X)*(I + X + X^2 + ... + X^i)
      = I - X^(i+1) -> I as i increases
 Next norm(I + X + X^2 + ...)
     <= norm(I) + norm(X) + norm(X^2) + ... by triangle inequality
     <= norm(I) + norm(X) + norm(X)^2 + ... by ||A*B||<=||A||*||B||
      = 1   + norm(X) + norm(X)^2 + ... by def of operator norm
      = 1/(1 - norm(X))   ... geometric sum

(Later: generalize to arbitrary Taylor expansions, like
  e^X = sum_i X^i/i!)

Lemma: Suppose A invertible. Then A-E invertible if
      norm(E) < 1/norm(inv(A)) in which case

inv(A-E) = Z + Z(EZ) + Z(EZ)^2 + Z(EZ)^3 + ... where Z=inv(A)
and
norm(inv(A-E)) <= norm(Z)/(1-norm(E)*norm(Z))

proof: inv(A-E) = inv((I-E*Z)*A) = Z*inv(I-E*Z)
exists if I-E*Z invertible i.e. if norm(E*Z) < 1,
i.e. if norm(E)*norm(Z) < 1 in which case
inv(A-E) = Z*(1+EZ + (EZ)^2 + ...)
Then take norms

What does this say for 1x1 matrices?
Why can't we write inv(A-E) = Z + EZ^2 + E^2Z^3 + ... ?

Finally, we can ask how much inv(A) and inv(A-E) differ.

Lemma: Suppose A invertible. If norm(E) < 1/norm(Z), then
norm(inv(A-E)-Z) <= norm(Z)^2*norm(E)/(1-norm(E)*norm(Z))

proof: inv(A-E)-Z = Z(EZ) + Z(EZ)^2 + ...
= ZEZ (I + EZ + (EZ)^2 + ...)
and then take norms

So the relative change in inv(A) is
norm(inv(A-E)-Z)/norm(Z)
<= [norm(A)*norm(Z) * 1/(1 - norm(E)*norm(Z))] *
[norm(E)/norm(A)]
= [norm(A)*norm(Z)] * [norm(E)/norm(A)] + O(norm(E)^2)
= condition_number * relative_change_in_A

What does this say for 1x1 matrices?

This justifies the following definition:

Def: condition number kappa(A) = norm(A)*norm(Z)
Fact: kappa(A) >= 1.
proof: 1 = norm(I) = norm(A*Z) <= norm(A)*norm(Z)

Theorem: min{norm(E)/norm(A): A-E singular}
= relative_distance(A, {singular matrices})
= 1/kappa(A)

proof for 2-norm, using SVD:
min{norm(E)}: A-E singular} = sigma_min(A),
so relative_distance(A,{singular}) = sigma_min(A)/sigma_max(A)

And norm(Z) = norm(inv(A)) = norm(V*inv(Sigma)*U^T)
= norm(inv(Sigma))= 1/sigma_min(A)

We've looked at sensitivity of inv(A), now look at solving A*x=b
Now consider A*x = b vs (A-E)*x' = b+f where x' = x+dx
subtract to get
A*dx - E*x - E*dx = f
(A-E)dx = f + E*x
dx = inv(A-E)*(f + E*x)
norm(dx) = norm(inv(A-E)*(f + E*x))
<= norm(inv(A-E))*(norm(f) + norm(E)*norm(x))
<= norm(Z)/(1-norm(E)*norm(Z))*(norm(f) + norm(E)*norm(x))
norm(dx)/norm(x)
<= norm(Z)*norm(A) * 1/(1-norm(E)*norm(Z))
*(norm(f)/(norm(A)*norm(x)) + norm(E)/norm(A))
<= norm(Z)*norm(A) * 1/(1-norm(E)*norm(Z))
*(norm(f)/norm(b) + norm(E)/norm(A))

relerr in x <= kappa(A)
* something close to 1 unless A nearly singular
* (rel change in b + rel change in A)

Our algorithms will attempt to guarantee that
(*) computed solution of A*x=b is (A-E)*x' = b+f where
norm(f)/norm(b) = O(macheps) and norm(E)/norm(A) = O(macheps)
so relerr in x ~ O(kappa(A)*macheps)
Recall that property (*) is called "backward stability"

Another practical approach: given x', how accurate a solution is it?
Compute residual r = A*x'-b = A*x'-A*x = A*(x'-x) = A*error
so error = inv(A)*r and norm(error) <= norm(inv(A))*norm(r)

norm(r) also determines the backward error in A:

Theorem: The smallest E such that (A+E)x' = b has
norm(E) = norm(r)/norm(x')
proof: r = A*x' - b = -E*x' so norm(r) <= norm(E)*norm(x')
To attain lower bound on norm(E), choose E = -r*x'^T/norm(x')^2,
in 2 norm.

In other words, if Ax'-b is small, then the backwards error is small,
which is probably the most you should ask for when the entries of A
are uncertain. (Later we will see how to use "iterative refinement",
aka Newton's method, to get a tiny error in x as long as kappa(A) is

not about 1/macheps or larger, but this is only sometimes justified.)

All our practical error bounds depend on norm(inv(A)). To actually compute inv(A) costs several times as much as just solving A*x=b (2n^3 versus (2/3)n^3) so we will use cheap estimates of norm(inv(A)) that avoid computing inv(A) explicitly, and work with small probability of large error.

The idea is to use the definition
  || inv(A) || = max_{||x|| = 1} || inv(A)*x ||
          = max_{||x||<= 1} || inv(A)*x ||
and do gradient ascent ("go uphill") on || inv(A)*x ||
on the convex set ||x|| <= 1; one may also start with a random starting vector. For right choice of norm it is easy to figure out ascent direction, and each step requires solving Ax=b for some b, which only costs O(n^2), (assuming we have already done LU factorization). In practice it usually takes at most 5 steps or so to stop ascending so the total costs is O(n^2); see sec 2.4.3 in the text for details.

In fact there is a theorem (Demmel, Diament, Malajovich, 2000) that says that estimating kappa(A) even roughly, but still with some guarantee (say "to within a factor of 10", or within any constant factor) is as about expensive as multiplying matrices, which in turn is about as expensive as doing Gaussian elimination in the first place. Since our goal is to spend just an additional O(n^2) to estimate norm(inv(A)) given that we have already done Gaussian Elimination (LU factorization), this theorem implies that we need to settle for a small probability of getting a large error in our estimate of norm(inv(A)).

Where to find implementations of all this?
  Matlab: A\b or [P,L,U]=lu(A) or condest or normest1
  LAPACK: xGETRF just for GEPP where x = S/D/C/Z
      xGESV to solve A*x=b
      xGESVX for condition estimation, more
      xGECON for condition estimation alone
  ScaLAPACK: PxGESV, etc