Notes for Ma221 Lecture 7, Feb 22, 2022

Goals for today: Least Squares

Example: polynomial fitting
   given sample points (y(i),b(i)), i=1:m, find the "best" polynomial
      p(y) of a fixed degree, i.e. to minimize sum_i (p(y(i))-b(i))^2
   formulate as minimizing norm of A*x-b, A(i,j) = y(i)^(j-1), x are
      coefficients of
        p(y) = x(1) + x(2)*y + x(3)*y^2 + ... + x(j)*y^(j-1)
   run polyfit31 to show results

Standard notation argmin_x norm(A*x-b), A mxn, m>n
    m>n means the system is overdetermined, don't expect A*x=b exactly
Some other variants of Least Squares (LS) problems (all in LAPACK):
   constrained: argmin_x norm(Ax-b) st Bx=y,
      where #rows(B) <= #cols(A) <= #rows(A) + #rows(B)
      so answer unique if matrices involved are also full rank
   weighted: argmin_x norm(y) st b = Ax+By
      weighted because if B = I , it would be usual problem.
      If B is square, nonsingular: argmin_x norm(inv(B)*(b-Ax))
   underdetermined: If #row(A) < #col(A), or more generally if A not
      full column rank, then there is a whole space of solutions
      (add any z st Az=0 to x). To make solution unique, usually
      seek argmin_x norm(x) st Ax=b
In lecture 1, we also mentioned the following variations:
   ridge regression: argmin_x || A*x-b ||_2^2 + lambda*|| x ||^2
        (also called Tikhonov regularization). This guarantees a
        unique solution when lambda > 0.
   total least squares:
        argmin_{x such that (A+E)*x=b+r} || [E,r] ||_2
        useful when there is uncertainty in A as well as b

Algorithms for overdetermined case (all used in various situations)

Solve Normal Equations (NE) - A^T*A*x = A^T*b
   A^T*A is spd, so solve with Cholesky
   fastest in dense case (fewest flops, least communication)
   but not stable if A ill-conditioned
Use QR decomposition A = Q*R, Q has orthonormal columns,
      R is upper triangular (details below)
   Gram-Schmidt - unstable, if A ill-conditioned
   Householder - stable
   blocked Householder - also minimizes data movement
   (also possible to use normal equations approach to get Q
   explicitly, called CholeskyQR, same pros and cons as normal
   equations: fast but can be unstable)
Use SVD - most "complete" answer, shows exactly how small changes in
   A, b could perturb solution x, best when A ill-conditioned, solves
   underdetermined system too.

```
Conversion to a linear system containing A, A^T, but not A^T*A (Q 3.3)
   allows exploiting sparsity, iterative refinement
   (also possible to exploit sparsity with QR, see software survey on
   class web page)

Normal Equations (NE)
Thm: If A full rank, solution of (A^T*A)*x = A^T*b minimizes
norm(A*x-b,2)
Proof: Assume x satisfies NE, multiply out norm(A*(x+e)-b,2)^2:
  norm(A*(x+e)-b,2)^2 = (Ax-b + Ae)^T*(Ax-b + Ae)
                      = (Ax-b)^T*(Ax-b) + 2e^T*A^T*(Ax-b) + (Ae)^T(Ae)
                      = norm(Ax-b,2)^2 + norm(Ae,2)^2
       which is minimized at e=0.
(picture, use of Pythagorean Thm)
#flops: m*n^2 (for A^T*A) + n^3/3 (for Cholesky)
#words moved: We know how to hit lower bound for each phase

QR Decomposition: A = Q*R, where A mxn, Q mxn, R nxn, Q orthogonal,
R upper triangular
Assuming we can compute A = Q*R, how do we solve argmin_x norm(A*x-b)?
    x = inv(R)*Q^T*b = R\(Q'*b) (in Matlab)
proof 1: A = Q*R = [Q,Qhat]*[R;0] where [Q,Qhat] square, orthogonal;
         then compute norm(A*x-b,2)^2 = norm( [Q,Qhat]^T*(A*x-b) ,2)^2
              = norm( [Q,Qhat]^T*(Q*R*x-b) ,2)^2
              = norm( [R*x-Q^T*b ; Qhat^T*b ] ,2)^2
              = norm(  R*x-Q^T*b, 2)^2 + norm( Qhat^T*b, 2)^2
         which is minimized when x = inv(R)*Q^T*b
proof 2: plug into normal equations:
         x = inv(A^T*A)*A^T*b = inv((Q*R)^T*(Q*R))*(Q*R)^T*b
           = inv( R^T*Q^T*Q*R)*R^T*Q^T*b = inv(R^T*R)*R^T*Q^T*b
           = inv(R)*Q^T*b

Algorithms for computing A=Q*R
Classical and Modified Gram-Schmidt
  Equate columns i of A and QR to get
      A(:,i) = sum_{j=1 to i} Q(:,j)*R(j,i)
  The columns of Q are orthogonal so (Q(:,j))^T*A(:,i) = R(j,i)

  for i=1:n
    tmp    = A(:,i)
    for j=1:i-1
       R(j,i) = Q(:,j)^T*A(:,i) ... CGS, costs 2m
       R(j,i) = Q(:,j)^T*tmp    ... MGS, costs 2m
         ... get same R(j,i) either way, in exact arithmetic
       tmp    = tmp    - R(j,i)*Q(:,j) ... costs 2m
    end
    R(i,i) = norm(tmp   ,2) ... costs 2m
    Q(:,i) = tmp   /R(i,i)  ... costs m
  end
total #flops = 4m*n^2/2 +O(mn) = 2mn^2 + O(mn), about 2x NE
```

(We pause the recorded lecture here.)

Two metrics of backward stability:
    If backward stable should get accurate QR decomposition of
      slight perturbed input matrix A+E = QR where
      norm(E)/norm(A) = O(macheps)
    This means norm(Q*R-A)/norm(A) should be O(macheps),
      just like we expect norm(A-PLU)/norm(A) to be O(macheps)
    But it also means norm(Q'*Q-I) should be O(macheps), an extra
       condition.
Plots showing instability:
    run QRStability with cnd = 1e1, 1e2 1e4 1e8 1e16 1e24, m=6, n=4,
          samples = 100
    Notes: MGS2 means running MGS twice, first to factor A = Q*R, and
           second applied to Q to make it even more orthogonal:
           A = Q*R = (Q1*R1)*R = Q1*(R1*R), where R1*R is upper
           triangular. CGS2 is analogous.
           Note that running GS twice helps make Q more orthogonal,
           but failures are possible when A is very ill-conditioned.
           The takeaway is that only Householder QR (explained below)
           is always stable. It will also be a building block for
           eigenvalue and SVD algorithms.

Recall SVD = Singular Value Decomposition

Thm. Suppose A = m x n, m >= n, then A = U*Sigma*V^T with
     mxm orthogonal matrix U = [u(1),...,u(m)] (left singular vectors)
     mxn diagonal matrix Sigma with diag(sigma(1),...,sigma(m)) in
       rows 1:n and m-n zero rows below, with singular values
       sigma(1) >= sigma(2) >= ... >= 0
    nxn orthogonal matrix V = [v(1),...,v(m)] (right singular vectors)
    When m > n, we sometimes write this as
       [u(1),...,u(n)] * diag(sigma(1),...,sigma(n)) * V^T
          == Uhat * Sigmahat * V^T

Recall Fact 2 from Lecture 3:
When m>n, we can use it to solve the full rank least squares problem
argmin_x ||A*x-b||_2 as follows: writing A = Uhat*Sigmahat*V^T as
above, then x = V*inv(Sigmahat)*Uhat^T*b

Def: If A = Uhat*Sigmahat*V^T is full rank, its (Moore-Penrose)
pseudoinverse is
       A^+ = V*inv(Sigmahat)*Uhat^T

Fact: A^+ = inv(A^T*A)*A^T (i.e. SVD and NE give same answer!)
      In matlab, A^+ = pinv(A) (when A not full rank or #cols>#rows,
      defined later)

Perturbation theory for least squares problem:
If x = argmin_x norm(A*x-b,2), and we change A and b a little, how much can x change? Since square case is special case, expect cond(A) to show up, but there is another source of sensitivity (note that A below has cond(A) = 1):

```
   A = [1;0], b = [0;b2],  => x = inv(A^T*A)*A^T*b = 1*0 = 0
   A = [1;0], b = [b1;b2], => x = 1*b1 = b1
```

If b2 very large, b1 could be small compared to b2 but still large, so a big change in x. More generally, if b is (nearly) orthogonal to span(A), then condition number very large.

Expand e = (x+e) - x

```
         = inv[(A+dA)^T(A+dA)]*(A+dA)^T*(b+db) - inv(A^T*A)*A^T*b,
```

using approximation inv(I-X) = I+X + O(||X||^2) as before, only keeping linear terms (one factor of dA and/or db), call eps = max(norm(dA)/norm(A) , norm(db)/norm(b)) to get

```
  norm(e)/norm(x) <= eps*(2*cond(A)/cos(theta)
                     + tan(theta)*cond(A)^2) + O(eps^2)
```

where theta = angle(b,A*x), or sin(theta) = norm(Ax-b,2)/norm(b,2). So condition number can be large if

```
   (1) cond(A) large
   (2) theta near pi/2, i.e. 1/cos(theta) ~ tan(theta) large
   (3) error like cond(A)^2 when theta not near zero
```

Will turn out that using the right QR decomposition, or SVD, keeps eps = O(machine epsilon), so backward stable

Normal equations are not backward stable; since we do Chol(A^T*A) the error bound is always proportional to cond(A)^2, even when theta small.

(We pause the recorded lecture here.)

Stable algorithms for QR decomposition:

Recall that MGS and CGS produced Qs that were far from orthogonal, and running them twice (MGS2 and CGS2) helped but sometime also failed to guarantee orthogonality.

To guarantee stability of solving least squares problems (and later eigenproblems and computing the SVD) we need algorithms that guarantee that Q is (very nearly) orthogonal, i.e. norm(Q^T*Q-I) = O(macheps).
Note: MGS still has its uses in some other algorithms, CGS does not.

Basic idea: express Q as product of simple orthogonal matrices Q=Q(1)*Q(2)*... where each Q(i) accomplishes part of the task (multiplying it by A makes some entries zero, until A turns into R). Since a product of orthogonal matrices is orthogonal, there is no danger of losing orthogonality.

There are two kinds of simple orthogonal matrices:
Householder transformations and Givens rotations.

A Householder transformation (or reflection) is H = I-2*u*u^T,
where u is a unit vector. We confirm orthogonality as follows:
 H*H^T = (I-2*u*u^T)*(I-2*u*u^T) = I - 4*u*u^T + 4*u*u^T*u*u^T = I
It is called a reflection because H*x is a reflection of x in plane
orthogonal to u
(picture).
Given x, we want to find u so that H*x has zeros in particular
locations, in particular we want u so that
  H*x = [c,0,0,...,0]^T = c*e1 for some constant c.
Since the 2-norm of both sides is ||x||_2, then |c| = ||x||_2.
We solve for u as follows: Write
   H*x =(I-2*u*u^T)*x = x - 2*u*(u^T*x) = c*e1, or
   u = (x-c*e1)/(2*u^T*x).
The denominator is just a constant that we can choose so that
||u||_2 = 1, i.e. y = x +- ||x||_2*e1 and u = y/||y||_2.
We write this as u = House(x). We choose the sign of +- to avoid
cancellation (avoids some numerical problems)
  y = [x(1) + sign(x(1))*||x||_2 , x(2), ..., x(n)]

How to do QR decomposition by forming Q as a product of Householder
transformations
(picture of 5 x 4 example)

QR decomposition of m x n matrix A, m >= n
   for i = 1 to min(m-1,n) ... only need to do last column if m > n
      u(i) = House(A(i:m,i))   ... compute Householder vector
      A(i:m,i:n) = (I - 2*u(i)*u(i)^T)*A(i:m,i:n)
                 = A(i:m,i:n) - u(i)*(2*u(i)^T*A(i:m,i:n))

Note: we never actually form each Householder matrix
Q(i) = I-2*u(i)*u(i)^T,
or multiply them to get Q, we only multiply by them.
We only need to store the u(i), which are kept in A in place of the
data they are used to zero out (same idea used to store L in
Gaussian elimination).
(picture)

The cost is
  sum_{i=1 to n} 4*(m-i+1)*(n-i+1) = 2n^2m - (2/3)n^3
      = (4/3)n^3 when m=n, twice the cost of GE

So when we are done we get
  Q(n)*Q(n-1)*...*Q(1)*A = R or
  A = Q(1)^T * ... * Q(n)^T * R = Q(1)*...*Q(n)*R = Q*R
and to solve the least squares problem argmin_x ||Ax-b||_2 we do
  x = R \ (Q^T * b)

or
```
  for i=1 to n
    ... b = Q(i)*b = (I-2*u(i)*u(i)^T)*b = b + (-2*u(i)^T*b)*u(i)
    c = -2*u(i)^T*b(i:m)       ... dot product
    b(i:m) = b(i:m) + c * u(i) ... "saxpy"
  endfor
  x = R \ b(1:n)
```
which costs O(mn), much less than A = QR (again analogous to GE)

In Matlab, x = A\b does GEPP if A is square, and solves the least squares problem using Householder QR when A has more rows than columns.

(We pause the recorded lecture here.)

Optimizing QR decomposition:
So far we have only described the BLAS2 version (analogous to simplest version of LU). We can (and should!) use all the ideas that we used for matmul and LU to minimize how much data is moved by QR decomposition, which shares the same lower bound:
#words moved between fast and slow memory
   = Omega(#flops/sqrt(fast_memory_size)).

The basic idea will be the same as for LU:
(1) do QR on the left part of the matrix
(2) update the right part of the matrix using the factorization of the left part
(3) Do QR on the right part.
As for LU, the left part can be a block of a fixed number of columns, or the whole left half (and then working recursively). Either way, we need to do step (2) above efficiently: apply the Householder factorizations from doing QR of the left part to the right part. In LU this was (mostly) updating the Schur complement using matmul, which is fast. So we need to figure out how to apply a product of Householder transformations Q = Q(b)*Q(b-1)*...*Q(1) with just a few matmuls. Here is the result we need:

Thm (see Q 3.17 for details): If Q(i) = I-2*u(i)*u(i)^T, then
Q = Q(b)*Q(b-1)*...*Q(1) = I - Y*T*Y^T
where Y = [u(1),...,u(b)] is mxb, and T is bxb and upper triangular. So multiplying by Q reduces to 3 matmuls.

There is one other new idea we need to deal with the common case when m >> n, i.e. A is "tall and skinny", or the problem is "very overdetermined". In case n^2 is as small as M = fast memory size, the lower bound becomes
```
   #words moved = Omega(#flops/sqrt(M)) = Omega( m *n^2 / sqrt(M) )
                = Omega(m*n),
```
where m*n is the size of the matrix A. In other words, the lower bound says we should just be able to access all the data once, which is an

obvious lower bound for any algorithm. Using Householder transformations column by column will not work (unless the whole matrix fits in fast memory) since the basic QR algorithm scans over all the columns n times, not once. To access the data just once we instead do "Tall Skinny QR", or TSQR.

Sequential TSQR (picture). To illustrate, Suppose we can only fit a little over 1/3 of the matrix in fast memory at one time. Then here is what we compute:

```
A = [ A1 ]
    [ A2 ]
    [ A3 ]
... just read A1 into fast memory, do QR, yielding A1=Q1*R1
  = [Q1*R1]
    [ A2 ]
    [ A3 ]
  = [ Q1     ] * [ R1 ] = Q1hat * [ R1 ]
    [    I   ]   [ A2 ]           [ A2 ]
    [      I ]   [ A3 ]           [ A3 ]
... where Q1hat is square and orthogonal.
... Q1 is still just represented as a collection of Householder
... vectors, so this requires no more work, just notation

... read A2 into fast memory, pack R1 on top of A2 yielding [R1;A2],
... do QR on [R1;A2], yielding [R1;A2] = Q2*R2
  = Q1hat * [ Q2*R2 ]
            [ A3    ]
  = Q1hat * [ Q2   ] * [ R2 ] = Q1hat * Q2hat * [ R2 ]
            [    I ]   [ A3 ]                   [ A3 ]
... store Q1hat and Q2hat separately

... read A3 into fast memory, do [R2;A3] = Q3*R3
  = Q1hat * Q2hat * [ Q3*R3 ]
  = Q1hat * Q2hat *  Q3hat * R3
```

This is the QR decomposition, because Q1hat * Q2hat * Q3hat is a product of orthogonal matrices, and so orthogonal, and R3 is upper triangular. As described, we only need to read the matrix once (and write the Q factors once).

This may also be called a "streaming algorithm", because it can compute R as A "streams" into memory row-by-row, even if A is too large to store completely.

This algorithm is well suited for parallelism, because it doesn't matter in what order we take pairs of submatrices and do their QR decompositions. Here is an example of parallel TSQR, where each of 4 processors owns 1/4 of the rows of A:

```
... each processor i does QR on its submatrix Ai
A = [ A1 ] = [ Q1*R1 ]  = [ Q1          ]*[R1] = Q1hat * [R1]
    [ A2 ]   [ Q2*R2 ]    [    Q2       ] [R2]           [R2]
    [ A3 ]   [ Q3*R3 ]    [       Q3    ] [R3]           [R3]
    [ A4 ]   [ Q4*R4 ]    [          Q4 ] [R4]           [R4]
  = Q1hat * [ Q12*R12 ] = Q1hat * [Q12   ] * [R12]
            [ Q34*R34 ]           [   Q34]   [R34]
... where [R1;R2]=Q12*R12 and [R3;R4]=Q34*R34

  = Q1hat * Q2hat * [R12] = Q1hat * Q2hat * [Q1234*R1234]
                    [R34]
  = Q1hat*Q2hat*Q3hat*R1234
```

Briefly, one could describe this as MapReduce where the reduction operation is QR.

We note that there is another use of this "reduction" idea for QR:

```
A = [ A1 ] -> [ R1 ]  -> [R1;R2] -> [R12] --> [R12;R34] -> [R1234]
    [ A2 ]    [ R2 ]  /                    /
    [ A3 ]    [ R3 ]  -> [R3;R4] -> [R34] /
    [ A4 ]    [ R4 ]  /
```

to also perform GEPP on an m x n matrix A with one reduction. The goal is to pick the "most independent n rows of A" with one reduction operation; now the output of each step are the n rows selected by GEPP on input rows:

```
A= [ A1 ] -> [ PA1 ]  -> [PA1;PA2] -> [PA12] --> [PA12;PA34] -> PA1234
   [ A2 ]    [ PA2 ]  /                     /
   [ A3 ]    [ PA3 ]  -> [PA3;PA4] -> [PA34] /
   [ A4 ]    [ PA4 ]  /
```

In other words, PA1 are the n rows chosen as pivot rows from GEPP applied to A1, PA12 are the n rows chosen as pivot rows from GEPP applied to [PA1;PA2], etc. This can be much faster than standard GEPP applied to A, which needs to do a reduction n times (once on each column of A) to find the maximum entry. Analogous to calling the QR algorithm TSQR (Tall Skinny QR), we call this TSLU.

TSLU does not choose the same pivot rows that GEPP would, but can still be shown to be backward stable in the following sense: It is possible to construct matrix B, a larger matrix than A, with block entries consisting of multiple copies of the blocks of A, so that the rows chosen by TSLU applied to A are the same as the rows GEPP would choose from B.

(We pause the recorded lecture here.)

Besides Householder transformations, there is one other way to represent Q as a product of simple orthogonal transformations, called Givens rotations. They are useful in other cases than dense least squares problems (for which we use Householder) so we present them here.

A Givens rotation R(theta) = [cos(theta) -sin(theta)]
                             [sin(theta) cos(theta)]
rotates x counterclockwise by theta (picture)
More generally, we will take components i and j of a vector and
rotate them:
R(i,j,theta) = identity except for rows and columns i and j,
 containing entries of R(theta) (picture)
To do QR, we want to create zero entries. So given x(i) and x(j),
we want theta so that R(i,j,theta)*x is 0 in entry j:
    [cos(theta) -sin(theta)] * [x(i)] = [sqrt(x(i)^2+x(j)^2)]
    [sin(theta)   cos(theta)]   [x(j)] = [0   ]
or
    cos(theta) = x(i)/sqrt(x(i)^2+x(j)^2),
    sin(theta) = -x(j)/sqrt(x(i)^2+x(j)^2)
We do QR by repeatedly multiplying A by R(i,j,theta) matrices to
introduce new zeros, until A becomes upper triangular (picture).
When A is sparse, this may cause less fill-in than Householder
transformations.

Stability of applying Orthogonal matrices
Simple summary: Any algorithm that just works by multiplying an input
matrix by orthogonal matrices is always backwards stable. This covers
QR composition using Householder or Givens rotations, as well as later
algorithms for the eigenproblem and SVD.

Here is why this is true:
By using the basic rule fl(a op b) = (a op b)*(1+delta) ,
|delta| <= macheps, one can show that for either multiplying by one
Householder or Givens transformation Q one gets the following, where
Q' is the floating point transformation actually stored in the
machine, eg Q' = I - 2*u'*u'^T where u' is the computed Householder
vector:
   fl(Q'*A) = Q'*A + E where norm(E) = O(macheps)*norm(A)
This follows from our earlier analysis of dot products, etc.
As can be seen from the above formula for u=House(x), every
component u'(i) is nearly equal to the exact value u(i), but roundoff
keeps it from being perfect. Since Q' is nearly equal to the exact
orthogonal Q, then Q' = Q+F with norm(F) = O(macheps), so we can also
write this as
   fl(Q'*A) = (Q+F)*A + E = Q*A + (F*A + E) = Q*A + G
where norm(G) <= norm(F)*norm(A) + norm(E) = O(macheps)*norm(A).
i.e. you get an exact orthogonal transformation Q*A plus a small
error. The same result applies to Givens rotations and block
Householder transformations (used to reduce communication).

We can also write this as fl(Q'*A) = Q*A+G = Q*(A + Q^T*G) = Q*(A+F),
where norm(F)=norm(G)=O(macheps)*norm(A),
i.e. multiplying by Q is backward stable: we get the exact orthogonal
transformation of a slightly different matrix A+F.

Now multiply by a sequence of orthogonal matrices, as in
QR decomposition:

```
   fl(Q3'*(Q2'*(Q1'*A))) = fl(Q3'*(Q2'*(Q1*A+E1)))
                         = fl(Q3'*(Q2*(Q1*A+E1)+E2))
                         = Q3*(Q2*(Q1*A+E1)+E2)+E3
                         = Q3*Q2*Q1*A + Q3*Q2*E1 + Q3*E2 + E3
                         = Q3*Q2*Q1*A + E
                         = Q*A + E
                         = Q*(A + Q^T*E)
                         = Q*(A+F)
```

   where ||F|| = ||E|| <= ||E1|| + ||E2|| + ||E3|| = O(macheps)
so multiplying by many orthogonal matrices is also backwards stable.

This analysis explains not just why QR is stable, but also why
norm(Q^T*Q-I) = O(macheps), unlike Gram-Schmidt.

We close with one more very simple, and fast, QR decomposition
algorithm, called CholeskyQR:

```
   Factor A^T*A = R^T*R using Cholesky
   Q = A*R^(-1)
```

It is clear that Q*R = A, and Q^T*Q = R^(-T)*A^T*A*R^(-1) = I.
But since we form A^T*A, this will clearly be unstable if A is
ill-conditioned, and fails completely if Cholesky fails to complete,
which can happen if we need to take the square root of a nonpositive
number (which also means the trick of running it twice, as used with
Gram-Schmidt, doesn't work either).