Lecture 19. Stability of Least Squares Algorithms

Least squares problems can be solved by various methods, as described in Lecture 11, including the normal equations, Householder triangularization, Gram–Schmidt orthogonalization, and the SVD. Here we compare these methods and show that the use of the normal equations is in general unstable.

Example

To illustrate the behavior of our algorithms, we shall apply them to a numerical example with $m = 100$, $n = 15$. Here is the MATLAB setup:

```
m = 100; n = 15;
t = (0:m-1)'/(m-1);          Set t to a discretization of [0,1].
A = []; for i=1:n,           Construct Vandermonde matrix.
  A = [A t.^(i-1)]; end
b = exp(sin(4*t));           Right-hand side.
b = b/2006.787453080206;     Normalization (see text).
```

The idea behind this example is the least squares fitting of the function $\exp(\sin(4\tau))$ on the interval $[0,1]$ by a polynomial of degree 14. First we discretize $[0,1]$, defining a vector t of 100 points equally spaced from 0 to 1. The matrix A is the 100×15 Vandermonde matrix whose columns are the

powers $1, \tau, \ldots, \tau^{14}$ sampled at the points of t, and the right-hand side b is the function $\exp(\sin(4\tau))$ sampled at these points.

The reason for the bizarre final line of the code is as follows. For simplicity, we are going to compare just the coefficients x_{15} computed by our various algorithms. Without this final line, the correct value of x_{15} would be 2006.787453080206... (this figure was obtained with an extended precision arithmetic package). By dividing by this number, we obtain a problem whose solution has $x_{15} = 1$, making our comparisons easier to follow.

To explain our observations, we shall need the quantities (18.3)–(18.5). One can determine these to sufficient accuracy by solving the least squares problem numerically with the aid of MATLAB's \ operator:

```
x = A\b; y = A*x;                    Solve least squares problem.
kappa = cond(A)
    kappa = 2.2718e+10                   κ(A)
theta = asin(norm(b-y)/norm(b))
    theta = 3.7461e-06                   θ
eta = norm(A)*norm(x)/norm(y)
    eta = 2.1036e+05                     η
```

The result $\kappa(A) \approx 10^{10}$ indicates that the monomials $1, t, \ldots, t^{14}$ form a highly ill-conditioned basis. The result $\theta \approx 10^{-6}$ indicates that $\exp(\sin(4t))$ can be fitted very closely by a polynomial of degree 14. (The fit is so close that we computed θ with the formula $\theta = \sin^{-1}(\|b-y\|/\|b\|)$ instead of (18.4), to avoid cancellation error.) As for η, its value of about 10^5 is about midway between the extremes 1 and $\kappa(A)$ permitted by (18.6).

Inserting these numbers into the formulas of Theorem 18.1, we find that for our example problem, the condition numbers of y and x with respect to perturbations in b and A are approximately

	y	x
b	1.0	1.1×10^5
A	2.3×10^{10}	3.2×10^{10}

Householder Triangularization

As mentioned in Lecture 11, the standard algorithm for solving least squares problems is QR factorization via Householder triangularization (Algorithm 11.2). Here is what we get with a MATLAB experiment:

```
[Q,R] = qr(A,0);                     Householder triang. of A.
x = R\(Q'*b);                        Solve for x.
x(15)
   ans = 1.00000031528723
```

What can we make of this result? Thanks to our normalization, the correct answer would be $x_{15} = 1$. Thus we have a relative error of about 3×10^{-7}. Since the calculation was done in IEEE double precision arithmetic with $\epsilon_{\text{machine}} \approx 10^{-16}$, this means that the rounding errors have been amplified by a factor of order 10^9. At first sight this looks bad, but a glance at the table above reminds us that the condition number of x with respect to perturbations in A is of order 10^{10}. Thus the inaccuracy in x_{15} can be entirely explained by ill-conditioning, not instability. Algorithm 11.2 appears to be backward stable.

Above, we formed $\hat{Q}$ explicitly, but as emphasized in Lectures 10 and 16, this is not necessary. It is enough to store the vectors v_k determined at the kth step of Algorithm 10.1 (equation (10.5)), which can then be utilized to compute $\hat{Q}^*b$ by Algorithm 10.2. In MATLAB, we can achieve this effect by computing a QR factorization not just of A but of the $m \times (n+1)$ "augmented" matrix $[A \ b]$. In the course of this factorization, the n Householder reflectors that make A upper-triangular are applied to b also, leaving the vector $\hat{Q}^*b$ in the first n positions of column $n+1$. An additional $(n+1)$st reflector is then applied to make entries $n+2, \ldots, m$ of column $n+1$ zero, but this does not change the first n entries of that column, which are the ones we care about. Thus:

```
[Q2,R2] = qr([A b],0);          Householder triang. of [A b].
R2 = R2(1:n,1:n);               Extract R̂ ...
Qb = R2(1:n,n+1);                  ... and Q̂*b.
x = R2\Qb;                      Solve for x.
x(15)
   ans = 1.00000031529465
```

The answer is almost the same as before. This indicates that the errors introduced in the QR factorization of A swamp those introduced in the computation of $\hat{Q}^*b$.

There is also a third way to solve the least squares problem via Householder triangularization in MATLAB. We can use the built-in operator \, as we did already in finding $\kappa(A)$, θ, and η:

```
x = A\b;                        Solve for x.
x(15)
   ans = 0.99999994311087
```

This result is distinctly different from the others, and an order of magnitude more accurate. The reason for this is that MATLAB's \ operator makes use of *QR factorization with column pivoting*, based on a factorization $AP = \hat{Q}\hat{R}$, where P is a permutation matrix. In this book we shall not discuss column pivoting.

From the point of view of normwise stability analysis, these three variants of QR factorization are equal. All of them, it can be proved, are backward stable.

Theorem 19.1. *Let the full-rank least squares problem* (11.2) *be solved by Householder triangularization (Algorithm* 11.2*) on a computer satisfying* (13.5) *and* (13.7). *This algorithm is backward stable in the sense that the computed solution* $\tilde{x}$ *has the property*

$$\|(A+\delta A)\tilde{x}-b\| = \min, \qquad \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}}) \tag{19.1}$$

for some $\delta A \in \mathbb{C}^{m\times n}$. *This is true whether* $\hat{Q}^*b$ *is computed via explicit formation of* $\hat{Q}$ *or implicitly by Algorithm* 10.2. *It also holds for Householder triangularization with arbitrary column pivoting.*

Gram–Schmidt Orthogonalization

Another way to solve a least squares problem is by modified Gram–Schmidt orthogonalization (Algorithm 8.1). For $m \approx n$, this takes somewhat more operations than the Householder approach, but for $m \gg n$, the flop counts for both algorithms are asymptotic to $2mn^2$.

The following MATLAB sequence implements this algorithm in the obvious fashion. The function `mgs` is an implementation (not shown) of Algorithm 8.1—the same as in Experiment 2 of Lecture 9.

```
[Q,R] = mgs(A);                    Gram–Schmidt orthog. of A.
x = R\(Q'*b);                      Solve for x.
x(15)
   ans = 1.02926594532672
```

This result is very poor. Rounding errors have been amplified by a factor on the order of 10^{14}, far greater than the condition number of the problem. In fact, this algorithm is unstable, and the reason is easily identified. As mentioned at the end of Lecture 9, Gram–Schmidt orthogonalization produces matrices $\hat{Q}$, in general, whose columns are not accurately orthonormal. Since the algorithm above depends on that orthonormality, it suffers accordingly.

The instability can be avoided by a reformulation of the algorithm. Since the Gram–Schmidt iteration delivers an accurate product $\hat{Q}\hat{R}$, even if $\hat{Q}$ does not have accurately orthogonal columns, one approach is to set up the normal equations $Rx = (\hat{Q}^*\hat{Q})^{-1}\hat{Q}^*b$ for the vector Rx, then get x by back substitution. As long as the computed $\hat{Q}$ is at least well-conditioned, this method will be free of the instabilities described below for the normal equations applied to arbitrary matrices. However, it involves unnecessary extra work and should not be used in practice.

A better method of stabilizing the Gram–Schmidt method is to make use of an augmented system of equations, just as in the second of our two Householder experiments above:

```
[Q2,R2] = mgs([A b]);          Gram–Schmidt orthog. of [A b].
R2 = R2(1:n,1:n);              Extract R̂ ...
Qb = R2(1:n,n+1);                 ... and Q̂*b.
x = R2\Qb;                     Solve for x.
x(15)
   ans = 1.00000005653399
```

Now the result looks as good as with Householder triangularization. It can be proved that this is always the case.

Theorem 19.2. *The solution of the full-rank least squares problem* (11.2) *by Gram–Schmidt orthogonalization is also backward stable, satisfying* (19.1), *provided that $\hat{Q}^*b$ is formed implicitly as indicated in the code segment above.*

Normal Equations

A fundamentally different approach to least squares problems is the solution of the normal equations (Algorithm 11.1), typically by Cholesky factorization (Lecture 23). For $m \gg n$, this method is twice as fast as methods depending on explicit orthogonalization, requiring asymptotically only mn^2 flops (11.14). In the following experiment, the problem is solved in a single line of MATLAB by the \ operator:

```
x = (A'*A)\(A'*b);             Form and solve normal equations.
x(15)
   ans = 0.39339069870283
```

This result is terrible! It is the worst we have obtained, with not even a single digit of accuracy. The use of the normal equations is clearly an unstable method for solving least squares problems. We shall take a moment to explain this phenomenon, for the explanation is a perfect example of the interplay of ideas of conditioning and stability. Also, the normal equations are so often used that an understanding of the risks involved is important.

Suppose we have a backward stable algorithm for the full-rank problem (11.2) that delivers a solution $\tilde{x}$ satisfying $\|(A+\delta A)\tilde{x} - b\| = \min$ for some δA with $\|\delta A\|/\|A\| = O(\epsilon_{\text{machine}})$. (Allowing perturbations in b as well as A, or considering stability instead of backward stability, does not change our main points.) By Theorems 15.1 and 18.1, we have

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O\left(\left(\kappa + \frac{\kappa^2 \tan\theta}{\eta}\right)\epsilon_{\text{machine}}\right), \tag{19.2}$$

where $\kappa = \kappa(A)$. Now suppose A is ill-conditioned, i.e., $\kappa \gg 1$, and θ is bounded away from $\pi/2$. Depending on the values of the various parameters, two very different situations may arise. If $\tan\theta$ is of order 1 (that is, the least squares fit is not especially close) and $\eta \ll \kappa$, the right-hand side (19.2) is $O(\kappa^2\epsilon_{\text{machine}})$. On the other hand, if $\tan\theta$ is close to zero (a very close fit) or η is close to κ, the bound is $O(\kappa\epsilon_{\text{machine}})$. *The condition number of the least squares problem may lie anywhere in the range κ to κ^2.*

Now consider what happens when we solve (11.2) by the normal equations, $(A^*A)x = A^*b$. Cholesky factorization is a stable algorithm for this system of equations in the sense that it produces a solution $\tilde{x}$ satisfying $(A^*A + \delta H)\tilde{x} = A^*b$ for some δH with $\|\delta H\|/\|A^*A\| = O(\epsilon_{\text{machine}})$ (Theorem 23.3). However, the matrix A^*A has condition number κ^2, not κ. Thus the best we can expect from the normal equations is

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\kappa^2\epsilon_{\text{machine}}). \tag{19.3}$$

The behavior of the normal equations is governed by κ^2, not κ.

The conclusion is now clear. If $\tan\theta$ is of order 1 and $\eta \ll \kappa$, or if κ is of order 1, then (19.2) and (19.3) are of the same order and the normal equations are stable. If κ is large and either $\tan\theta$ is close to zero or η is close to κ, however, then (19.3) is much bigger than (19.2) and the normal equations are unstable. *The normal equations are typically unstable for ill-conditioned problems involving close fits.* In our example problem, with $\kappa^2 \approx 10^{20}$, it is hardly surprising that Cholesky factorization yielded no correct digits.

According to our definitions, an algorithm is stable only if it has satisfactory behavior uniformly across all the problems under consideration. The following result is thus a natural formalization of the observations just made.

Theorem 19.3. *The solution of the full-rank least squares problem* (11.2) *via the normal equations (Algorithm* 11.1*) is unstable. Stability can be achieved, however, by restriction to a class of problems in which $\kappa(A)$ is uniformly bounded above or $(\tan\theta)/\eta$ is uniformly bounded below.*

SVD

One further algorithm for least squares problems was mentioned in Lecture 11: the use of the SVD (Algorithm 11.3). Like most computations based on the SVD, this one is stable:

```
[U,S,V] = svd(A,0);          Reduced SVD of A.
x = V*(S\(U'*b));            Solve for x.
x(15)
   ans = 0.99999998230471
```

In fact, this is the most accurate of all the results obtained in our experiments, beating Householder triangularization with column pivoting (MATLAB's \) by a factor of about 3. A theorem in the usual form can be proved.

Theorem 19.4. *The solution of the full-rank least squares problem* (11.2) *by the SVD (Algorithm* 11.3*) is backward stable, satisfying the estimate* (19.1).

Rank-Deficient Least Squares Problems

In this lecture we have identified four backward stable algorithms for linear least squares problems: Householder triangularization, Householder triangularization with column pivoting, modified Gram–Schmidt with implicit calculation of $\hat{Q}^*b$, and the SVD. From the point of view of classical normwise stability analysis of the full-rank problem (11.2), the differences among these algorithms are minor, so one might as well make use of the simplest and cheapest, Householder triangularization without pivoting.

However, there are other kinds of least squares problems where column pivoting and the SVD take on a special importance. These are problems where A has rank $< n$, possibly with $m < n$, so that the system of equations is *underdetermined*. Such problems do not have a unique solution unless one adds an additional condition, typically that x itself should have as small a norm as possible. A further complication is that the correct solution depends on the rank of A, and determining ranks numerically in the presence of rounding errors is never a trivial matter.

Thus rank-deficient least squares problems are not a challenging subclass of least squares problems, but fundamentally different. Since the definition of a solution is new, there is no reason that an algorithm that is stable for full-rank problems must be stable also in the rank-deficient case. In fact, the only fully stable algorithms for rank-deficient problems are those based on the SVD. An alternative is Householder triangularization with column pivoting, which is stable for almost all problems. We shall not give details.

Exercises

19.1. Given $A \in \mathbb{C}^{m\times n}$ of rank n and $b \in \mathbb{C}^m$, consider the block 2×2 system of equations

$$\begin{bmatrix} I & A \\ A^* & 0 \end{bmatrix} \begin{bmatrix} r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}, \tag{19.4}$$

where I is the $m \times m$ identity. Show that this system has a unique solution $(r, x)^T$, and that the vectors r and x are the residual and the solution of the least squares problem (18.1).

19.2. Here is a stripped-down version of one of MATLAB's built-in m-files.

```
[U,S,V] = svd(A);
S = diag(S);
tol = max(size(A))*S(1)*eps;
r = sum(S > tol);
S = diag(ones(r,1)./S(1:r));
X = V(:,1:r)*S*U(:,1:r)';
```

What does this program compute?