Lecture 6: Conditioning and Condition Numbers

NLA Chapters 12

What's next

Two fundamental issues of numerical analysis:

- Conditioning: perturbation behavior of a **mathematical problem** (in general, the difficulty of the problem itself to solve numerically!)
- Stability: perturbation behavior of an **algorithm** to solve that problem on a computer (the quality of the numerical algorithm)

What is perturbation? Why care?

Perturbation behavior is the sensitivity of the output to the small perturbations in the input.

I.e., if I change the input by a certain amount, how much my output is going to change?

Why care about perturbation?

Because reality is harsh ☺

Okay, because pretty much anything we do numerically on a computer involves some kind of error.

Error is Life

- Mathematics vs. Numerical Analysis (Richard W. Hamming):
  - Mathematics routinely work with **infinite** representations for numbers and process (real numbers, limits, etc)
  - Numerical Analysis: digital computer can only model **finite** representations.
    - Finite representation of numbers: **roundoff error**
    - Finite representation of processes: **truncation error**
- The difference is rather fundamental:
  - Numerical computations are “approximate computing”
  - Mathematical equivalent computations lead to very different numerical computations: **numerical analysis**
  - Numerical computations often feel more “art” than science.

Condition of a Problem

Abstractly, we can view a **problem** as a function $f: X \to Y$ where X is the data space and Y is the solution space. This function is usually continuous but not necessarily linear. E.g., linear solve problem $f(A, b) = A^{-1}b$

Typically we shall concern ourselves with the behavior of a problem f at a particular data point $x \in X$ (the behavior may be very different at different x!). The combination of f and x is called a **problem instance.** (Also sometimes confusingly called a problem).

A **well-conditioned** problem (instance) is one that all small perturbation to the x leads to only small changes in $f(x)$. A **ill-conditioned** problem is the opposite.

Quantification of Conditioning

Let δx denote a small perturbation of x, and write $\delta f = f(x + \delta x) - f(x)$.

The **absolute condition number** of a problem $f(x)$ is defined:

$$\hat{k} = \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\|}{\|\delta x\|}$$

This may look scary… but it's conceptually simple. Think about the definition of conditioning in the (infinitesimal) small region around x.

To put this notation more simply, let's just write

$$\hat{\kappa} = \sup_{\delta x} \frac{\|\delta f\|}{\|\delta x\|}$$

where $\delta x, \delta f$ are infinitesimal. (does it look like definition of matrix norm?)

If f is differentiable, then the condition number can be expressed in terms of the Jacobian (the derivative):

$$J(x) = \left[\frac{\partial f_i}{\partial x_j}\right]$$

is a matrix. The definition of derivative gives us

$$\delta f \approx J(x)\delta x$$

Thus the absolute condition number

$$\hat{\kappa} = \|J(x)\|$$

Relative Condition Number

Absolute condition number is **scale dependent.**

Most of the time, we want something that is scale independent. So we develop the relative condition number, which measures perturbation in relative terms (change by x% instead change by an absolute amount).

$$\kappa = \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\| / \|f(x)\|}{\|\delta x\| / \|x\|}$$

Again, assuming infinitesimal $\delta x, \delta f$

$$\kappa = \sup_{\delta x} \frac{\|\delta f\| / \|f(x)\|}{\|\delta x\| / \|x\|}$$

If f is differentiable, then

$$\kappa = \frac{\|J(x)\|}{\|f(x)\| / \|x\|}$$

Relative vs. Absolute

In numerical analysis, we care more about relative condition number than the absolute one.

Why? Because floating point arithmetic introduces relative errors. (It guarantees error in relative terms; e.g. float(x+y) will give us something around (x+y) within a small % of error).

A problem is well conditioned if κ is small (maybe a few hundred); ill conditioned if κ is large (millions). But it all depends (on problem, requirement, precision, algorithms, etc).

All these sound horribly complicated? Let's look at examples…

Examples

Example 12.1. Consider the trivial problem of obtaining the scalar $x/2$ from $x \in \mathbb{C}$. The Jacobian of the function $f : x \mapsto x/2$ is just the derivative $J = f' = 1/2$, so by (12.6),

$$\kappa = \frac{\|J\|}{\|f(x)\|/\|x\|} = \frac{1/2}{(x/2)/x} = 1.$$

This problem is well-conditioned by any standard. □

Example 12.2. Consider the problem of computing $\sqrt{x}$ for $x > 0$. The Jacobian of $f : x \mapsto \sqrt{x}$ is the derivative $J = f' = 1/(2\sqrt{x})$, so we have

$$\kappa = \frac{\|J\|}{\|f(x)\|/\|x\|} = \frac{1/(2\sqrt{x})}{\sqrt{x}/x} = \frac{1}{2}.$$

Again, this is a well-conditioned problem. □

An ill-conditioned example

Example 12.3. Consider the problem of obtaining the scalar $f(x) = x_1 - x_2$ from the vector $x = (x_1, x_2)^* \in \mathbb{C}^2$. For simplicity, we use the ∞-norm on the data space $\mathbb{C}^2$. The Jacobian of f is

$$J = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} & \dfrac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & -1 \end{bmatrix},$$

with $\|J\|_\infty = 2$. The condition number is thus

$$\kappa = \frac{\|J\|_\infty}{\|f(x)\|/\|x\|} = \frac{2}{|x_1 - x_2|/\max\{|x_1|, |x_2|\}}.$$

This quantity is large if $|x_1 - x_2| \approx 0$, so the problem is ill-conditioned when $x_1 \approx x_2$, matching our intuition of the hazards of "cancellation error." □

ill-conditioned problem: Polynomial root finding.

Example 12.5. The determination of the roots of a polynomial, given the coefficients, is a classic example of an ill-conditioned problem. Consider $x^2 - 2x + 1 = (x - 1)^2$, with a double root at $x = 1$. A small perturbation in the coefficients may lead to a larger change in the roots; for example, $x^2 - 2x + 0.9999 = (x - 0.99)(x - 1.01)$. In fact, the roots can change in proportion to the square root of the change in the coefficients, so in this case the Jacobian is infinite (the problem is not differentiable), and $\kappa = \infty$.

Even in case of no multiple roots, polynomial root finding can be quite ill-conditioned.

If the ith coefficient a_i of a polynomial $p(x) = \sum_i a_i x^i$ is perturbed by an infinitesimal δa_i, the jth root is perturbed by $(\delta a_i) x_j^i / p'(x_j)$.

The condition number of x_j with respect to perturbation of a_i is

$$\kappa = \frac{|\delta x_j|/|x_j|}{|\delta a_i|/|a_i|} = \frac{|a_i x_j^{i-1}|}{|p'(x_j)|}$$

Consider Wilkinson's polynomial:

$$p(x) = \prod_{i=1}^{20}(x - i) = a_0 + a_1 x^1 + \cdots + a_{19} x^{19} + a_{20} x^{20}$$

The most sensitive root is $x_j = 15$, and it's most sensitive to coefficient $a_{15} \approx 1.67 \times 10\text{^}19$. The condition number is

$$\kappa \approx 5.1 \times 10^{13}$$

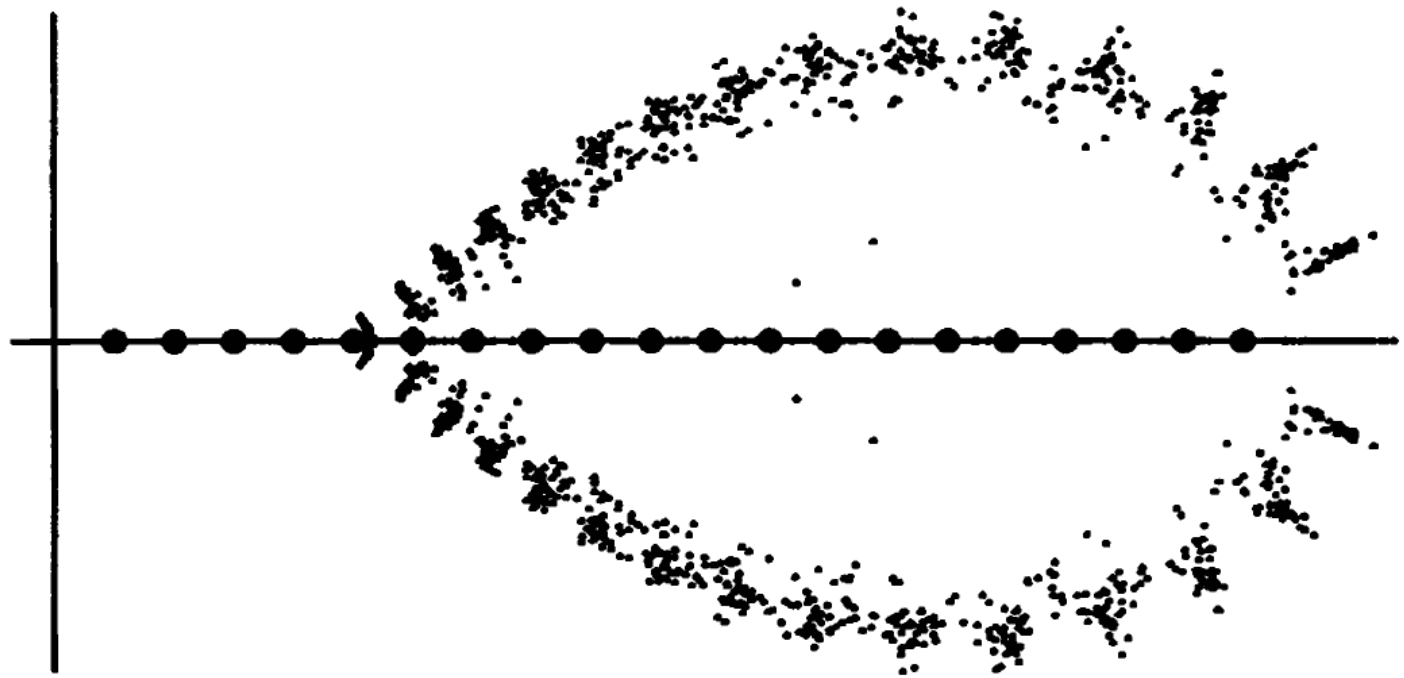


Figure 12.1. *Wilkinson's classic example of ill-conditioning. The large dots are the roots of the unperturbed polynomial* (12.8). *The small dots are the superimposed roots in the complex plane of* 100 *randomly perturbed polynomials with coefficients defined by* $\tilde{a}_k = a_k(1 + 10^{-10} r_k)$, *where* r_k *is a number from the normal distribution of mean* 0 *and variance* 1.

Condition of MatVec Mul

The fundamental condition number in NLA.

We first fix the matrix A, and find the MatVec $f(x) = Ax$. The condition number is

$$\kappa = \sup_{\delta x} \frac{\|A\delta x\|/\|\delta x\|}{\|Ax\|/\|x\|} = \|A\| \frac{\|x\|}{\|Ax\|}$$

We want to get rid of x. If A is invertible, then

$$\|x\|/\|Ax\| \leq \|A^{-1}\|$$

(why?) The equality holds when x is a multiple of minimal right singular vector of A (again, why?)

Now we got an upper bound on condition number:

$$\kappa \leq \|A\|\|A^{-1}\|$$

If A is not-square (tall) but have full column rank, then replace the A^{-1} with A^{+} in the above formula.

Condition number of a matrix

The product $\|A\|\|A^{-1}\|$ appears so often, that it has its own name: the condition number of matrix A.

$$\kappa(A) = \|A\|\|A^{-1}\|$$

If we use 2-norm, the condition number of A is

$$\kappa(A) = \frac{\sigma_1}{\sigma_m}$$

(the eccentricity of the hyperellipse, the image of unit sphere)

Condition of a system of linear equations

In MatVec Mul problem $x = A^{-1}b$ we fix A and see how x changes in response to perturbations in b.

What if we perturb A?

As usual, we perturb A by an infinitesimal δA, and in response x changes by δx, and they satisfy:

$$(A + \delta A)(x + \delta x) = b$$

We have the condition number:

$$\kappa = \frac{\|\delta x\|/\|x\|}{\|\delta A\|/\|A\|} \leq \|A^{-1}\|\|A\| = \kappa(A)$$

and for any A, b there exist some δx for which the equality holds.

Thus, the condition number of system of linear equations is $\kappa(A)$ (exactly the condition number of the matrix)

Floating Point Arithmetic

NLA Chapter 13

Finite representation of real numbers

Digital computer use finite number of bits to represent real numbers. This has two implications:

1. The representable number is cannot be arbitrarily large or small
2. There are gaps between two adjacent representable numbers.

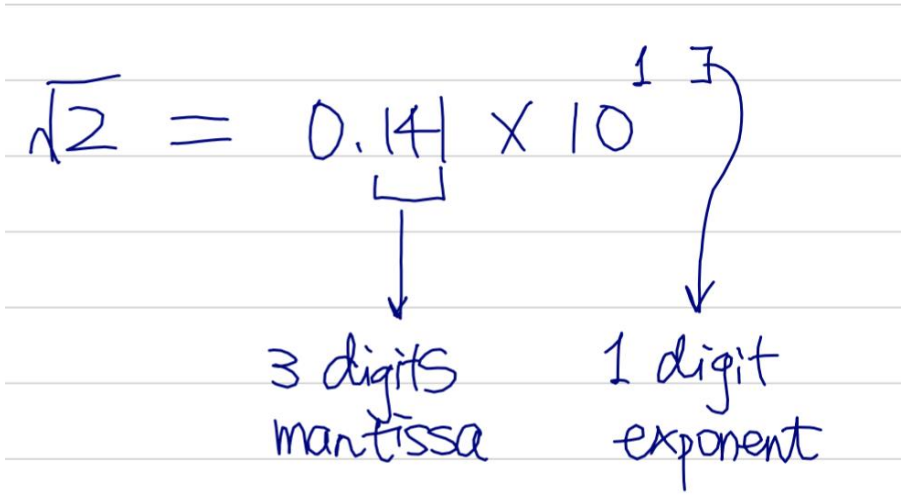
All modern computers (nowadays there are reemergence of customized floating point system or even fixed point system, specialized for AI) use IEEE floating point format to represent numbers and do arithmetic.

IEEE 754

Type	Range	Unit Roundoff Error
FP16	$10^{\pm 5}$	5×10^{-4}
FP32	$10^{\pm 38}$	6×10^{-8}
FP64	$10^{\pm 308}$	1×10^{-16}

- Arithmetic operations $(+, -, \times, \div, \sqrt{})$ performed as if the calculation is done in infinite precision, then rounded
- Half precision used to be a storage only format, but nowadays NVIDIA pioneered implementing arithmetic for FP16

A Toy 3-digit Decimal Floating Point System



0.999×10^{9}

- The number next to zero is 0.100×10^{-9}
- The next number is 0.101×10^{-9}
- The finest space between two numbers (ULP): 0.001×10^{-9}
- Largest number: 0.999×10^{9}
- Surprises:
  - Overflow/underflow 10^{-12}
  - Associativity?
  - Inexact arithmetic: roundoff errors

Rounding

Multiplication:

$$0.318 \times 10^{1}$$
$$\times\ 0.129 \times 10^{4}$$

2862

636

318

$$0.041922 \times 10^{5}$$

$$0.410 \times 10^{2}$$

Addition

$$0.318 \times 10^{1} + 0.129 \times 10^{4}$$

$$0.000318 \times 10^{4}$$
$$+\ 0.129000 \times 10^{4}$$

$$0.129318 \times 10^{4}$$

$$0.129 \times 10^{4}$$

Machine Epsilon

The **resolution** of a floating point system can be summarized by the *machine epsilon* $\epsilon_{\text{machine}}$. This is half the distance between 1 and the next larger floating point number.

In a relative sense, this is the largest gap between floating point numbers. I.e.,$\epsilon_{\text{machine}}$ has the following property:

For all $x \in \mathbb{R}$, there exists $x' \in F$ such that $|x - x'| \leq \epsilon_{\text{machine}}|x|$

In other words, for any real number (not out of range!) x there is a floating point number that is not too far away from it (within machine epsilon relatively)

So if we convert a real number to the (nearest) floating point number, we get a close enough one with relative error bounded by $\epsilon_{\text{machine}}$.

In fact, let's define a function $\text{fl}: \mathbb{R} \to F$ that gives the closest floating point approximation to a real number (its **rounded** equivalent in F). So we can rephrase the property of machine epsilon:

For all $x \in \mathbb{R}$, there exists ϵ with $|\epsilon| \leq \epsilon_{\text{machine}}$

such that $\text{fl}(x) = x(1 + \epsilon)$

Floating Point Arithmetic

In addition to represent real number, one must also compute with them. On a computer, every calculation on floating point are reduced to elementary arithmetic operations, such as $+, -, \times, /$.

We have analogous operations on floating point numbers (F), commonly denoted by $\oplus, \ominus, \otimes, \oslash$

The floating point arithmetic has the following property: given two FP numbers $x, y \in F$, the floating point arithmetic gives:

$$x \otimes y = \mathrm{fl}(x \times y)$$

It works as if you carry out the (real) arithmetic on x, y and then round to the nearest representable FP.

Fundamental Axiom of FP Arithmetic

For all $x, y \in F$, there exists ϵ with $|\epsilon| \leq \epsilon_{\text{machine}}$

$$x \otimes y = (x \times y)(1 + \epsilon)$$

In other words, every floating point arithmetic is exact, up to a certain relative error of size at most machine epsilon.

In our error analysis, we use and only use the two framed properties to get error bounds. The error bounds will hold for any system that satisfies these two conditions, which include the IEEE 754 systems.

Stability

NLA Chapter 14

Stability

It's generally not possible for a digital computer to give exact answer to our problem.

Then, what is possible? What can we expect? What means "right answer" if they are not exact?

Previously, we talked about conditioning of a **mathematical problem** $f: X \to Y$ as a function from input vector space X to output vector space Y.

The conditioning of a problem is the sensitivity of output to input, which measures the (inherent) difficulty of the problem itself. It's regardless of the actual algorithm to compute it.

Numerical Algorithms

Now we introduce *algorithm* as another map $\tilde{f}: X \to Y$ (the same vector spaces as *problem*).

What does $\tilde{f}(x)$ mean? It means:

Let's fix the problem f, an algorithm for f, and a computer program implementing the algorithm.

Given data $x \in X$, first we round it, and supply it as input to the program. The result $\tilde{f}(x)$ is a vector of floating point numbers belonging to Y.

We are generally interested in how far the $\tilde{f}(x)$ (the computed solution) deviates from $f(x)$ (the real solution).

Accuracy

As usual, we measure the accuracy of computed solution using relative error:

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|}$$

Under floating point system, the best we can hope for is that the relative error is at the level of machine epsilon $\epsilon_{\text{machine}}$:

$$\frac{\|\tilde{f}(x) - f(x)\|}{\|f(x)\|} = O(\epsilon_{\text{machine}})$$

The meaning of $O(\epsilon)$ is intuitively like this: the left side is a function of machine epsilon; imagine that we have a bunch of computers that have increasingly small $\epsilon_{\text{machine}}$; as the $\epsilon_{\text{machine}} \to 0$, the left side $\to 0$ at the same rate.

Stability

But this requirement is probably too ambitious. Why? Because if a problem is ill-conditioned, even the tiny roundoff error in the input data is going to change $\tilde{f}(x)$ significantly. Furthermore, each arithmetic could add more errors.

So instead of shooting for accuracy, we step back, and aim for **stability**. We say an algorithm is **stable,** if for each $x \in X$,

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(\tilde{x})\|} = O(\epsilon_{\text{machine}})$$

for some $\tilde{x}$ with

$$\frac{\|\tilde{x} - x\|}{\|x\|} = O(\epsilon_{\text{machine}})$$

Stability

In English, stability means:

A stable algorithm gives *nearly* the right answer,

to *nearly* the right question.

We solve a slightly wrong question, with slightly wrong answer.

This is a counterintuitive way to say stability… but it makes *a lot of sense*. Why? Because it casts error back to the perturbation to the input data, thus solved our problem with conditioning.

Stability is a reasonable requirement for an algorithm.

Backward Stability

Many NLA algorithm satisfy a *stronger* and *simpler* stability called **backward stability:**

$$\tilde{f}(x) = f(\tilde{x}), \text{ for some } \tilde{x} \text{ with } \frac{\|\tilde{x}-x\|}{\|x\|} = O(\epsilon_{\text{machine}})$$

(How is this different from stability?)

In English:

> A backward stable algorithm gives *exactly* the right answer to nearly the right question.

In other words, we are solving slightly wrong question *exactly.*

Confusing? Examples will follow.

Stability Examples

Lets take a look at the simplest “algorithms”, the floating point arithmetic. Are they stable?

In fact, they are not only stable, but backward stable. Take $\ominus$ for example. The input is $(x_1, x_2) \in X$, the output is in space $\mathbb{R}$. The subtraction problem is $f(x_1, x_2) = x_1 - x_2$. The algorithm is

$$\tilde{f}(x_1, x_2) = \mathrm{fl}(x_1) \ominus \mathrm{fl}(\mathrm{x}_2)$$

By the two floating point properties, we have (all the epsilons are bounded by machine epsilon)

$$\mathrm{fl}(x_1) = x_1(1 + \epsilon_1), \mathrm{fl}(x_2) = x_2(1 + \epsilon_2)$$

and

$$\begin{aligned} \mathrm{fl}(x_1) \ominus \mathrm{fl}(\mathrm{x}_2) &= \big(\mathrm{fl}(x_1) - \mathrm{fl}(x_2)\big)(1 + \epsilon_3) \\ &= x_1(1 + \epsilon_1)(1 + \epsilon_3) - x_2(1 + \epsilon_2)(1 + \epsilon_3) \\ &= x_1(1 + \epsilon_4) - x_2(1 + \epsilon_5) \end{aligned}$$

Examples

The inner product of two vectors $f(x, y) = x^T y$: the obvious algorithm is to compute $x_i \otimes y_i$, and then $\oplus$ then together (in whatever order). This algorithm is also backward stable.

How about outer product $f(x, y) = xy^T$? This is not backward stable, but is stable.

Accuracy of Backward Stable Algorithm

So we have a backward stable algorithm. How accurate is its result? Well it depends on the conditioning of the problems.

$$\frac{\left\|\tilde{f}(x) - f(x)\right\|}{\|f(x)\|} = \frac{\|f(\tilde{x}) - f(x)\|}{\|f(x)\|} \leq \kappa(x) \frac{\|\tilde{x} - x\|}{\|x\|}$$
$$= O(\kappa(x)\epsilon_{\text{machine}})$$

This is as good as we can reasonably hope for.

So if you are using a backward stable algorithm, and getting inaccurate result, it must be due to ill-conditioning of the problem itself (the problem is too difficult! Maybe you want to try different problem formulation.)

An unstable algorithm.

Computing eigenvalues using characteristic polynomials. The algorithm works like this:

1. Compute the coefficients of characteristic polynomial: $p(\lambda) = \det(A - \lambda I)$
2. Compute the roots of $p(\lambda) = 0$.

This algorithm is not only backward unstable, but also unstable.

Look at the 2D identity matrix $A = I_2$. The eigenvalues are not sensitive to the perturbations of A (well conditioned). If you use the above algorithm to compute eigenvalues, and you perturb the A by ϵ, your eigenvalues will have error $\sqrt{\epsilon}$ (because root finding is unstable!). You get bad result on well-conditioned problem.

Stability of Householder QR

In the next slides we are going to experimentally explore the remarkable backward stability of Householder QR algorithm, and its use to solve linear systems.

`R = triu(randn(50));`	Set R to a 50×50 upper-triangular matrix with normal random entries.
`[Q,X] = qr(randn(50));`	Set Q to a random orthogonal matrix by orthogonalizing a random matrix.
`A = Q*R;`	Set A to the product QR, up to rounding errors.
`[Q2,R2] = qr(A);`	Compute QR factorization $A \approx Q_2 R_2$ by Householder triangularization.

Householder QR Error

So we constructed a (ill-conditioned) matrix A with known QR factorization $A = QR$, and computed Q2 and R2 from Householder QR algorithm. How good are the Q2 and R2 (forward error?)

Not great… we have about 2~3 correct digits.

```
norm(Q2-Q)                    How accurate is Q2?
    ans = 0.00889
norm(R2-R)/norm(R)            How accurate is R2?
    ans = 0.00071
```

Backward Stability of HouseQR

But, if we measure the backward error…

```
norm(A-Q2*R2)/norm(A)
    ans = 1.432e-15
```

How accurate is Q_2R_2?

It's as good as one can get from double precision ($\epsilon_{\text{machine}} \approx 10^{-16}$)!!
To see how remarkable this is, let's take a random matrix Q3,R3 that are similarly close to Q,R and see their backward error:

“Diabolically Correlation”

Our HouseQR algorithm picks a quite special pair of Q2,R2! Although they look rather different than the real Q,R, their product is the same and $QR \approx Q_2R_2$.

```
Q3 = Q+1e-4*randn(50);      Set Q3 to a random perturbation of Q
                            that is closer to Q than Q2 is.
R3 = R+1e-4*randn(50);      Set R3 to a random perturbation of R
                            that is closer to R than R2 is.
norm(A-Q3*R3)/norm(A)       How accurate is Q3R3?
     ans = 0.00088
```

This shows that **HouseQR is backward stable**; the inaccurate Q2,R2 is entirely due to the ill-condition of the QR problem (with this specific A with large condition number).

HouseQR backward stability

We state here without proof that, HouseQR is backward stable, **only if you compute the implicit Q.**

Precisely what this means is like this:

Let $\tilde{Q}, \tilde{R}$ be the factors computed by HouseQR algorithm, where $\tilde{Q}$ is the **orthogonal** matrix implicitly formed by the Householder vectors: $\tilde{Q} = \widetilde{Q_1}\widetilde{Q_2}\cdots\widetilde{Q_n}$. Then the computed factors satisfy:

$$\tilde{Q}\tilde{R} = A + \delta A, \frac{\|\delta A\|}{\|A\|} = O(\epsilon_{\text{machine}})$$

If you form the explicit $\tilde{Q}$, you lose this backward stability.

HouseQR as Linear Solver

In fact we can prove that HouseQR is backward stable as a theorem (which we don't go into).

Now, because QR factorization is seldom the ends itself, but a means to some ends, we consider the algorithm to use HouseQR to solve (square) linear system:

Algorithm 16.1. Solving $Ax = b$ by QR Factorization

$QR = A$	Factor A into QR by Algorithm 10.1, with Q represented as the product of reflectors.
$y = Q^*b$	Construct Q^*b by Algorithm 10.2.
$x = R^{-1}y$	Solve the triangular system $Rx = y$ by back substitution (Algorithm 17.1).

Is it backward stable, assuming every step is backward stable?

Stability of Least Square Algorithms

There are at least four algorithms to solve least square problems: normal equation, HouseQR, Gram-Schmidt, and the SVD.

Are they (backward) stable? Let's experiment.

```
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).
```

We manufacture a low-degree polynomial least square fitting problem with a know solution $x_{15} = 1$

The conditions of this LLS

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

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

Householder Triangularization

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

How do we make this result? It looks like we get about 7 digits correct; We lost around 9 digits and all of them can be explained by the large numbers ($\sim 10^{10}$) .
This algorithm is therefore backward stable.

Factorize the Augmented System

We can actually avoid explicit computing $Q^T b$, by directly factorizing the augmented matrix $[A\ b]$ instead of A alone.

```
[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
```

This gives us about the same accuracy as before.

How does Matlab solve it

Not bad, actually a little bit more accurate than the last two algorithms.

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

factorization that may work better for ill-conditioned problem.

Gram-Schmidt QR for LLS

How does the modified Gram-Schmidt QR fare in solving LLS?

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

Does not look good. This is **unstable**, because the $Q^T * b$ step is unstable, because Q is not particularly orthogonal. (HouseQR is not affected because if Q is near orthogonal, hence stability)

But mgs() is backward stable, so if we can avoid the second step…

How about factorizing the augmented system $[A\ b]$?

MGS on Augmented System

As good as one can get. This algorithm is backward stable, because all the steps are backward stable.

```
[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
```

Normal Equations

The obvious and simplest way to solve LLS: $A^T Ax = A^T b$

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

This is way off… not a single correct digit. What happened?

More confusingly, every step looks "backward stable"! (The linear solve is via Cholesky factorization which is backward stable).

So the whole algorithm is backward stable; the problem could only be that *we changed the condition number.*

What is the condition number of $A^T A$? (hint, $\kappa(A)^2$)

One more… the SVD

It's time for our heaviest hammer SVD to work.

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

This is the most accurate, about 3x more than Matlab \.

If the matrix A does not have full rank, then it's a completely different problem because we no longer have a unique solution. In that case we must determine the numerical rank. The only stable algorithm is SVD based. Alternatively column pivoted QR may sometimes be usable.