2022-09-06

1 Modeling floating point

The fact that normal floating point results have a relative error bounded by ϵ_{mach} gives us a useful *model* for reasoning about floating point error. We will refer to this as the "$1+\delta$" model. For example, suppose x is an exactly-represented input to the Julia statement

```
z = 1-x*x
```

We can reason about the error in the computed $\hat{z}$ as follows:

$$\begin{aligned}
t_1 &= \text{fl}(x^2) = x^2(1+\delta_1) \\
t_2 &= 1 - t_1 = (1-x^2)\left(1 - \frac{\delta_1 x^2}{1-x^2}\right) \\
\hat{z} &= \text{fl}(1-t_1) = z\left(1 - \frac{\delta_1 x^2}{1-x^2}\right)(1+\delta_2) \\
&\approx z\left(1 - \frac{\delta_1 x^2}{1-x^2} + \delta_2\right),
\end{aligned}$$

where $|\delta_1|, |\delta_2| \leq \epsilon_{\text{mach}}$. As before, we throw away the (tiny) term involving $\delta_1\delta_2$. Note that if z is close to zero (i.e. if there is *cancellation* in the subtraction), then the model shows the result may have a large relative error.

1.1 First-order error analysis

Analysis in the $1+\delta$ model quickly gets to be a sprawling mess of Greek letters unless one is careful. A standard trick to get around this is to use *first-order* error analysis in which we linearize all expressions involving roundoff errors. In particular, we frequently use the approximations

$$\begin{aligned}
(1+\delta_1)(1+\delta_2) &\approx 1 + \delta_1 + \delta_2 \\
1/(1+\delta) &\approx 1 - \delta.
\end{aligned}$$

In general, we will resort to first-order analysis without comment. Those students who think this is a sneaky trick to get around our lack of facility

with algebra[1] may take comfort in the fact that if $|\delta_i| < \epsilon_{\text{mach}}$, then in double precision

$$\left| \prod_{i=1}^{n} (1+\delta_i) \prod_{i=n+1}^{N} (1+\delta_i)^{-1} \right| < (1 + 1.03N\epsilon_{\text{mach}})$$

for $N < 10^{14}$ (and a little further).

1.2 Shortcomings of the model

The $1+\delta$ model has two shortcomings. First, it is only valid for expressions that involve normalized numbers — most notably, gradual underflow breaks the model. Second, the model is sometimes pessimistic. Certain operations, such as taking a difference between two numbers within a factor of 2 of each other, multiplying or dividing by a factor of two[2], or multiplying two single-precision numbers into a double-precision result, are *exact* in floating point. There are useful operations such as simulating extended precision using ordinary floating point that rely on these more detailed properties of the floating point system, and cannot be analyzed using just the $1+\delta$ model.

2 Finding and fixing floating point problems

Floating point arithmetic is not the same as real arithmetic. Even simple properties like associativity or distributivity of addition and multiplication only hold approximately. Thus, some computations that look fine in exact arithmetic can produce bad answers in floating point. What follows is a (very incomplete) list of some of the ways in which programmers can go awry with careless floating point programming.

2.1 Cancellation

If $\hat{x} = x(1+\delta_1)$ and $\hat{y} = y(1+\delta_2)$ are floating point approximations to x and y that are very close, then $\text{fl}(\hat{x} - \hat{y})$ may be a poor approximation to $x - y$ due to *cancellation*. In some ways, the subtraction is blameless in this tail: if x and y are close, then $\text{fl}(\hat{x} - \hat{y}) = \hat{x} - \hat{y}$, and the subtraction causes no

[1] Which it is.

[2] Assuming that the result does not overflow or produce a subnormal.

additional rounding error. Rather, the problem is with the approximation error already present in $\hat{x}$ and $\hat{y}$.

The standard example of loss of accuracy revealed through cancellation is in the computation of the smaller root of a quadratic using the quadratic formula, e.g.

$$x = 1 - \sqrt{1 - z}$$

for z small. Fortunately, some algebraic manipulation gives an equivalent formula that does not suffer cancellation:

$$x = \left(1 - \sqrt{1-z}\right)\left(\frac{1+\sqrt{1-z}}{1+\sqrt{1-z}}\right) = \frac{z}{1+\sqrt{1-z}}.$$

2.2 Sensitive subproblems

We often solve problems by breaking them into simpler subproblems. Unfortunately, it is easy to produce badly-conditioned subproblems as steps to solving a well-conditioned problem. As a simple (if contrived) example, try running the following Julia code:

```
function silly_sqrt(n=100)
  x = 2.0
  for k = 1:n
    x = sqrt(x)
  end
  for k = 1:n
    x = x^2
  end
  x
end
```

In exact arithmetic, this should produce 2, but what does it produce in floating point? In fact, the first loop produces a correctly rounded result, but the second loop represents the function $x^{2^{60}}$, which has a condition number far greater than 10^{16} — and so all accuracy is lost.

2.3 Unstable recurrences

One of my favorite examples of this problem is the recurrence relation for computing the integrals

$$E_n = \int_0^1 x^n e^{x-1}\, dx.$$

Integration by parts yields the recurrence

$$E_0 = 1 - 1/e$$
$$E_n = 1 - nE_{n-1}, \quad n \geq 1.$$

This looks benign enough at first glance: no single step of this recurrence causes the error to explode. But each step amplifies the error somewhat, resulting in an exponential growth in error[3].

2.4 Undetected underflow

In Bayesian statistics, one sometimes computes ratios of long products. These products may underflow individually, even when the final ratio is not far from one. In the best case, the products will grow so tiny that they underflow to zero, and the user may notice an infinity or NaN in the final result. In the worst case, the underflowed results will produce nonzero subnormal numbers with unexpectedly poor relative accuracy, and the final result will be wildly inaccurate with no warning except for the (often ignored) underflow flag.

2.5 Bad branches

A NaN result is often a blessing in disguise: if you see an unexpected NaN, at least you *know* something has gone wrong! But all comparisons involving NaN are false, and so when a floating point result is used to compute a branch condition and an unexpected NaN appears, the result can wreak havoc. As an example, try out the following code in Julia with ‘0.0/0.0‘ as input.

```
function test_negative(x)
  if x < 0.0
    "$(x) is negative"
  elseif x >= 0.0
    "$(x) is non-negative"
  else
    "$(x) is ... uh..."
  end
end
```

[3]Part of the reason that I like this example is that one can run the recurrence *backward* to get very good results, based on the estimate $E_n \approx 1/(n+1)$ for n large.

3 Sums and dots

We already described a couple of floating point examples that involve evaluation of a fixed formula (e.g. computation of the roots of a quadratic). We now turn to the analysis of some of the building blocks for linear algebraic computations: sums and dot products.

3.1 Sums two ways

As an example of first-order error analysis, consider the following code to compute a sum of the entries of a vector v:

```
s = 0
for k = 1:n
  s += v[k]
end
```

Let $\hat{s}_k$ denote the computed sum at step k of the loop; then we have

$$\begin{aligned}\hat{s}_1 &= v_1 \\ \hat{s}_k &= (\hat{s}_{k-1} + v_k)(1+\delta_k), \quad k > 1.\end{aligned}$$

Running this forward gives

$$\begin{aligned}\hat{s}_2 &= (v_1 + v_2)(1+\delta_2) \\ \hat{s}_3 &= ((v_1 + v_2)(1+\delta_2) + v_3)(1+\delta_2)\end{aligned}$$

and so on. Using first-order analysis, we have

$$\hat{s}_k \approx (v_1 + v_2)\left(1 + \sum_{j=2}^{k} \delta_j\right) + \sum_{l=3}^{k} v_l \left(1 + \sum_{j=l}^{k} \delta_j\right),$$

and the difference between $\hat{s}_k$ and the exact partial sum is then

$$\hat{s}_k - s_k \approx \sum_{j=2}^{k} s_j \delta_j.$$

Using $\|v\|_1$ as a uniform bound on all the partial sums, we have

$$|\hat{s}_n - s_n| \lesssim (n-1)\epsilon_{\text{mach}} \|v\|_2.$$

An alternate analysis, which is a useful prelude to analyses to come involves writing an error recurrence. Taking the difference between $\hat{s}_k$ and the true partial sums s_k, we have

$$\begin{aligned} e_1 &= 0 \\ e_k &= \hat{s}_k - s_k \\ &= (\hat{s}_{k-1} + v_k)(1+\delta_k) - (s_{k-1} + v_k) \\ &= e_{k-1} + (\hat{s}_{k-1} + v_k)\delta_k, \end{aligned}$$

and $\hat{s}_{k-1} + v_k = s_k + O(\epsilon_{\text{mach}})$, so that

$$|e_k| \leq |e_{k-1}| + |s_k|\epsilon_{\text{mach}} + O(\epsilon_{\text{mach}}^2).$$

Therefore,

$$|e_n| \lesssim (n-1)\epsilon_{\text{mach}}\|v\|_1,$$

which is the same bound we had before.

3.2 Backward error analysis for sums

In the previous subsection, we showed an error analysis for partial sums leading to the expression:

$$\hat{s}_n \approx (v_1 + v_2)\left(1 + \sum_{j=2}^{n} \delta_j\right) + \sum_{l=3}^{n} v_l \left(1 + \sum_{j=l}^{n} \delta_j\right).$$

We then proceded to aggregate all the rounding error terms in order to estimate the error overall. As an alternative to aggregating the roundoff, we can also treat the rounding errors as perturbations to the input variables (the entries of v); that is, we write the computed sum as

$$\hat{s}_n = \sum_{j=1}^{n} \hat{v}_j$$

where

$$\hat{v}_j = v_j(1+\eta_j), \qquad \text{where } |\eta_j| \lesssim (n+1-j)\epsilon_{\text{mach}}.$$

This gives us a *backward error* formulation of the rounding: we have re-cast the role of rounding error in terms of a perturbation to the input vector v. In terms of the 1-norm, we have the relative error bound

$$\|\hat{v} - v\|_1 \lesssim n\epsilon_{\text{mach}}\|v\|_1;$$

or we can replace n with $n-1$ by being a little more careful. Either way, what we have shown is that the summation algorithm is *backward stable*, i.e. we can ascribe the roundoff to a (normwise) small relative error with a bound of $C\epsilon_{\text{mach}}$ where the constant C depends on the size n like some low-degree polynomial.

Once we have a bound on the backward error, we can bound the forward error via a condition number. That is, suppose we write the true and perturbed sums as

$$s = \sum_{j=1}^{n} v_j \qquad\qquad \hat{s} = \sum_{j=1}^{n} \hat{v}_j.$$

We want to know the relative error in $\hat{s}$ via a normwise relative error bound in $\hat{v}$, which we can write as

$$\frac{|\hat{s}-s|}{|s|} = \frac{|\sum_{j=1}^{n}(\hat{v}_j - v_j)|}{|s|} \leq \frac{\|\hat{v}-v\|_1}{|s|} = \frac{\|v\|_1}{|s|}\frac{\|\hat{v}-v\|_1}{\|v\|_1}.$$

That is, $\|v\|_1/|s|$ is the condition number for the summation problem, and our backward stability analysis implies

$$\frac{|\hat{s}-s|}{|s|} \leq \frac{\|v\|_1}{|s|} n\epsilon_{\text{mach}}.$$

This is the general pattern we will see again in the future: our analysis consists of a backward error computation that depends purely on the algorithm, together with a condition number that depends purely on the problem. Together, these give us forward error bounds.

3.3 Running error bounds for sums

In all the analysis of summation we have done so far, we ultimately simplified our formulas by bounding some quantity in terms of $\|v\|_1$. This is nice for algebra, but we lose some precision in the process. An alternative is to compute a *running error bound*, i.e. augment the original calculation with something that keeps track of the error estimates. We have already seen that the error in the computations looks like

$$\hat{s}_n - s_n = \sum_{j=2}^{n} s_j\delta_j + O(\epsilon_{\text{mach}}^2),$$

and since s_j and $\hat{s}_j$ differ only by $O(\epsilon_{\text{mach}})$ terms,

$$|\hat{s}_n - s_n| \lesssim \sum_{j=2}^{n} |\hat{s}_j| \epsilon_{\text{mach}} + O(\epsilon_{\text{mach}}^2),$$

We are not worried about doing a rounding error analysis of our rounding error analysis — in general, we care more about order of magnitude for rounding error anyhow — so the following routine does an adequate job of computing an (approximate) upper bound on the error in the summation:

```
s = 0.0
e = 0.0
for k = 1:n
  s += v[k]
  e += abs(s) * eps(Float64);
end
```

3.4 Compensated summation

We conclude our discussion of rounding analysis for summation with a comment on the *compensated summation* algorithm of Kahan, which is not amenable to straightforward $1 + \delta$ analysis. The algorithm maintains the partial sums not as a single variable `s`, but as an unevaluated sum of two variables `s` and `c`:

```
s = 0.0
c = 0.0
for k = 1:n
  y = v[i] - c
  t = s + y
  c = (t - s) - y # Key step
  s = t
end
```

Where the error bound for ordinary summation is $(n-1)\epsilon_{\text{mach}}\|v\|_1 + O(\epsilon_{\text{mach}}^2)$, the error bound for compensated summation is $2\epsilon_{\text{mach}}\|v\|_1 + O(\epsilon_{\text{mach}}^2)$. Moreover, compensated summation is exact for adding up to 2^k terms that are within about 2^{p-k} of each other in magnitude.

Nor is Kahan's algorithm the end of the story! Higham's *Accuracy and Stability of Numerical Methods* devotes an entire chapter to summation methods, and there continue to be papers written on the topic. For our purposes, though, we will wrap up here with two observations:

- Our initial analysis in the $1+\delta$ model illustrates the general shape these types of analyses take and how we can re-cast the effect of rounding errors as a "backward error" that perturbs the inputs to an exact problem.
- The existence of algorithms like Kahan's compensated summation method should indicate that the backward-error-and-conditioning approach to rounding analysis is hardly the end of the story. One could argue it is hardly the beginning! But it is the approach we will be using for most of the class.

3.5 Dot products

We now consider another example, this time involving a real dot product computed by a loop of the form

```
dot = 0
for k = 1:n
  dot += x[k]*y[k];
end
```

Unlike the simple summation we analyzed above, the dot product involves two different sources of rounding errors: one from the summation, and one from the product. As in the case of simple summations, it is convenient to re-cast this error in terms of perturbations to the input. We could do this all in one go, but since we have already spent so much time on summation, let us instead do it in two steps. Let $v_k = x_k y_k$; in floating point, we get $\hat{v}_k = v_k(1+\eta_k)$ where $|\eta_k| < \epsilon_{\text{mach}}$. Further, we have already done a backward error analysis of summation to show that the additional error in summation can be cast onto the summands, i.e. the floating point result is $\sum_k \tilde{v}_k$ where

$$\begin{aligned}\tilde{v}_k &= \hat{v}_k(1 + \sum_{j=\min(2,n)}^{n} \delta_j)(1+\eta_k) + O(\epsilon_{\text{mach}}^2) \\ &= v_k(1+\gamma_k) + O(\epsilon_{\text{mach}}^2)\end{aligned}$$

where

$$|\gamma_k| = |\eta_k + \sum_{j=\min(2,n)}^{n} \delta_j| \leq n\epsilon_{\text{mach}}.$$

Rewriting $v_k(1+\gamma_k)$ as $\hat{x}_k y_k$ where $\hat{x}_k = x_k(1+\gamma_k)$, we have that the computed inner product $y^T x$ is equivalent to the exact inner product of $y^T \hat{x}$

where $\hat{x}$ is an elementwise relatively accurate (to within $n\epsilon_{\text{mach}} + O(\epsilon_{\text{mach}}^2)$) approximation to x.

A similar backward error analysis shows that computed matrix-matrix products AB in general can be interpreted as $\hat{A}B$ where

$$|\hat{A} - A| < p\epsilon_{\text{mach}}|A| + O(\epsilon_{\text{mach}}^2)$$

and p is the inner dimension of the product. Exactly what $\hat{A}$ is depends not only on the data, but also the loop order used in the multiply — since, as we recall, the order of accumulation may vary from machine to machine depending on what blocking is best suited to the cache! But the bound on the backward error holds for all the common re-ordering[4] And this backward error characterization, together with the type of sensitivity analysis for matrix multiplication that we have already discussed, gives us a uniform framework for obtaining forward error bounds for matrix-matrix muliplication; and the same type of analysis will continue to dominate our discussion of rounding errors as we move on to more complicated matrix computations.

3.6 Back-substitution

We now consider the floating point analysis of a standard *back-substitution* algorithm for solving an upper triangular system

$$Uy = b.$$

To solve such a linear system, we process each row in turn in reverse order to find the value of the corresponding entry of y. For example, for the 3-by-3 case with

$$U = \begin{bmatrix} 1 & 3 & 5 \\ & 4 & 2 \\ & & 6 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ -12 \\ 12 \end{bmatrix}$$

Back substitution proceeds row-by-row:

Row 3: $6y_3 = 12$ (so $y_3 = 12/2 = 2$)

Row 2: $4y_2 + 2y_3 = -12$ (so $y_2 = (-12 - 2y_3)/4 = -4$)

Row 1: $y_1 + 3y_2 + 5y_3 = 1$ (so $y_1 = (1 - 3y_2 - 5y_3)/1 = 3$)

[4]For those of you who know about Strassen's algorithm — it's not backward stable, alas.

More generally, we have

$$y_i = \left(b_i - \sum_{j>i} u_{ij} y_j \right) / u_{ii}.$$

In code, if we weren't inclined to just write `y=U\b`, we might write this as

```
y = copy(b)
for i = n:-1:1
    # Loop equivalent to y[i] -= dot(U[i,i+1:end], y[i+1:end])
    for j = i+1:n
        y[i] -= U[i,j]*y[j]
    end
    y[i] /= U[i,i]
end
```

If we evaluate this in floating point arithmetic as a dot product, subtraction, and division, we get that

$$\hat{y}_i = \left(b_i - \sum_{j>i} \hat{u}_{ij} \hat{y}_j \right) / u_{ii} \cdot (1+\delta_1)(1+\delta_2)$$

where the $\hat{y}_j$ terms are the previously-computed entries in the y vector, the $\hat{u}_{ij}$ terms are the u_{ij} with a $(n-i-1)\epsilon_{\text{mach}}$ backward error modification from the dot product, the δ_1 error is associated with the subtraction and the δ_2 error is associated with the division. This in turn gives us that

$$\hat{y}_i = \left(b_i - \sum_{j>i} \hat{u}_{ij} \hat{y}_j \right) / \hat{u}_{ii}$$

where

$$\hat{u}_{ii} = \frac{u_{ii}}{(1+\delta_1)(1+\delta_2)} = u_{ii}(1 - \delta_1 - \delta_2 + O(\epsilon_{\text{mach}}^2)).$$

That is, we can recast the final subtraction and division as a relative perturbation of $\lesssim 2\epsilon_{\text{mach}}$ to the diagonal. Putting everything together, we have that

$$\hat{U}\hat{y} = b$$

where $|\hat{U} - U| \lesssim n\epsilon_{\text{mach}}|U|$.

4 Error analysis for linear systems

We now discuss the sensitivity of linear systems to perturbations. This is relevant for two reasons:

1. Our standard recipe for getting an error bound for a computed solution in the presence of roundoff is to combine a backward error analysis (involving only features of the algorithm) with a sensitivity analysis (involving only features of the problem). We saw an example above: we know that the standard back-substitution process results in a backward error like $n\epsilon_{\text{mach}}|U|$, but what does that mean for solutions of the linear system?

2. Even without rounding error, it is important to understand the sensitivity of a problem to the input variables if the inputs are in any way inaccurate (e.g. because they come from measurements).

We describe several different bounds that are useful in different contexts.

4.1 First-order analysis

We begin with a discussion of the first-order sensitivity analysis of the system

$$Ax = b.$$

Using our favored variational notation, we have the following relation between perturbations to A and b and perturbations to x:

$$\delta A\, x + A\, \delta x = \delta b,$$

or, assuming A is invertible,

$$\delta x = A^{-1}(\delta b - \delta A\, x).$$

We are interested in relative error, so we divide through by $\|x\|$:

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\|A^{-1}\delta b\|}{\|x\|} + \frac{\|A^{-1}\delta A\, x\|}{\|x\|}$$

The first term is bounded by

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

and the second term is bounded by

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

Putting everything together, we have

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A)\left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|}\right),$$

That is, the relative error in x is (to first order) bounded by the condition number times the relative errors in A and b.

4.2 Beyond first order

What if we want to go beyond the first-order error analysis? Suppose that

$$Ax = b \quad \text{and} \quad \hat{A}\hat{x} = \hat{b}.$$

Then (analogous to our previous manipulations),

$$(\hat{A} - A)\hat{x} + A(\hat{x} - x) = \hat{b} - b$$

from which we have

$$\hat{x} - x = A^{-1}\left((\hat{b} - b) - E\hat{x}\right),$$

where $E \equiv \hat{A} - A$. Following the same algebra as before, we have

$$\frac{\|\hat{x} - x\|}{\|x\|} \leq \kappa(A)\left(\frac{\|E\|}{\|A\|}\frac{\|\hat{x}\|}{\|x\|} + \frac{\|\hat{b} - b\|}{\|b\|}\right).$$

Assuming $\|A^{-1}\|\|E\| < 1$, a little additional algebra (left as an exercise to the student) yields

$$\frac{\|\hat{x} - x\|}{\|x\|} \leq \frac{\kappa(A)}{1 - \|A^{-1}\|\|E\|}\left(\frac{\|E\|}{\|A\|} + \frac{\|\hat{b} - b\|}{\|b\|}\right).$$

Is this an important improvement on the first order bound? Perhaps not, for two reasons:

- One typically cares about the order of magnitude of possible error, not the exact bound, and
- The first-order bound and the "true" bound only disagree when both are probably pretty bad. When our house is in flames, our first priority is not to gauge whether the garage will catch as well; rather, we want to call the firefighters to put it out!

4.3 Componentwise relative bounds

What if we have more control over the perturbations than a simple bound on the norms? For example, we might have a componentwise perturbation bound

$$|\delta A| < \epsilon_A |A| \quad |\delta b| < \epsilon_b |b|,$$

and neglecting $O(\epsilon^2)$ terms, we obtain

$$|\delta x| \leq |A^{-1}| \left(\epsilon_b |b| + \epsilon_A |A||x|\right) \leq (\epsilon_b + \epsilon_A)|A^{-1}||A||x|.$$

Taking any vector norm such that $\| \, |x| \, \| = \|x\|$, we have

$$\|\delta x\| \leq (\epsilon + \epsilon') \| \, |A^{-1}| \, |A| \, \|.$$

The quantity $\kappa_{\text{rel}}(A) = \| \, |A^{-1}| \, |A| \, \|$ is the componentwise relative condition number (also known as the Skeel condition number).

4.4 Residual-based bounds

The *residual* for an approximate solution $\hat{x}$ to the equation $Ax = b$ is

$$r = A\hat{x} - b.$$

We can express much simpler error bounds in terms of the residual, using the relation

$$\hat{x} - x = A^{-1} r;$$

taking norms immediately gives

$$\|\hat{x} - x\| \leq \|A^{-1}\| \|r\|$$

and for any vector norm such that $\| \, |x| \, \| = \|x\|$, we have

$$\|\hat{x} - x\| \leq \| \, |A^{-1}| |r| \, \|.$$

Note that we can re-cast a residual error as a backward error on A via the relation

$$\left(A - \frac{r\hat{x}^T}{\|\hat{x}\|^2}\right)\hat{x} = b.$$

4.5 Shape of error

So far, we have only really discussed the *magnitude* of errors in a linear solve, but it is worth taking a moment to consider the *shape* of the errors as well. In particular, suppose that we want to solve $Ax = b$, and we have the singular value decomposition

$$A = U\Sigma V^T.$$

If $\sigma_n(A) \ll \sigma_1(A)$, then $\kappa_2 = \sigma_1/\sigma_n \gg q$, and we expect a large error. But is this the end of the story? Suppose that A satisfies

$$1 \geq \sigma_1 \geq \ldots \geq \sigma_k \geq C_1 \quad > \quad C_2 \geq \sigma_{k+1} \geq \ldots \geq \sigma_n > 0.$$

where $C_1 \gg C_2$. Let $r = A\hat{x} - b$, so that $Ae = r$ where $e = \hat{x} - x$. Then

$$e = A^{-1}r = V\Sigma^{-1}U^Tr = V\Sigma^{-1}\tilde{r} = \sum_{j=1}^{n} \frac{\tilde{r}_j}{\sigma_j} v_j.$$

where $\|\tilde{r}\| = \|U^Tr\| = \|r\|$. Split this as

$$e = e_1 + e_2$$

where we have a controlled piece

$$\|e_1\| = \left\| \sum_{j=1}^{k} \frac{\tilde{r}_j}{\sigma_j} v_j \right\| \leq \frac{\|r\|}{C_1}$$

and a piece that may be large,

$$e_2 = \sum_{j=k+1}^{n} \frac{\tilde{r}_j}{\sigma_j} v_j.$$

Hence, backward stability implies that the error consists of a small part and a part that lies in the "nearly-singular subspace" for the matrix.