CS 4220: Numerical Analysis

GEPP, backward error analysis, refinement

David Bindel

2026-02-11

The wonderful backslash

In Julia, we can solve the linear system $Ax = b$ in several different ways, each of them using the backslash operator:

```
# Good if we want to solve just one system
x = A\b

# Good if we want to solve multiple systems with A
F = lu(A)
x = F\b

# Also OK, though verbose
L,U,p = lu(A)
x = U\(L\b[p])
```

The backslash operator is overloaded for different types of objects. For triangular matrices, backslash applies forward or backward substitution; for factorization objects, backslash involves both a forward and backsubstitution pass. When we simply compute `A\b`, Julia factors the matrix A (using Gaussian elimination) and then solves the system using forward and backward substitution.

The following code is generally *not* a good idea:

```
# BAD idea -- don't do this!
x = inv(A)*b   # Code that calls `inv` deserves skepticism
x = U\L\b[p]   # Order of operations means we form U\L!
```

Computing an explicit inverse generally is equivalent to computing `A\I`: first we factor A, then we solve a linear system for each column of the identity matrix. Forming and multiplying by an explicit inverse is thus more expensive than simply using LU factorization from the outset! It is also less numerically stable. There are rare cases where an explicit inverse is appropriate, but in general, if you call `inv` in this class without explicit instructions to do so, we will deduct points.

Perturbation theory

Previously, we described a general error analysis strategy: derive forward error bounds by combining a sensitivity estimate (in terms of a *condition number*) with a *backward* error analysis that explains the computed result as the exact answer to a slightly erroneous problem. To follow that strategy here, we need the sensitivity analysis of solving linear systems.

Suppose that $Ax = b$ and that $\hat{A}\hat{x} = \hat{b}$, where $\hat{A} = A + \delta A$, $\hat{b} = b + \delta b$, and $\hat{x} = x + \delta x$. Then

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

Assuming the delta terms are small, we have the linear approximation

$$\delta A\, x + A\, \delta x \approx \delta b.$$

We can use this to get δx alone:

$$\delta x \approx A^{-1}(\delta b - \delta A\, x);$$

and taking norms gives us

$$\|\delta x\| \lesssim \|A^{-1}\|(\|\delta b\| + \|\delta A\|\|x\|).$$

Now, divide through by $\|x\|$ to get the relative error in x:

$$\frac{\|\delta x\|}{\|x\|} \lesssim \|A\|\|A^{-1}\| \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|A\|\|x\|} \right).$$

Recall that $\|b\| \leq \|A\|\|x\|$ to arrive at

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

where $\kappa(A) = \|A\|\|A^{-1}\|$. That is, the relative error in x is (to first order) bounded by the condition number times the relative errors in A and b. We can go beyond first order using Neumann series bounds – but perhaps not today.

Backward error in Gaussian elimination

Solving $Ax = b$ in finite precision using Gaussian elimination followed by forward and backward substitution yields a computed solution $\hat{x}$ *exactly* satisfies

$$(A + \delta A)\hat{x} = b, \tag{1}$$

where $|\delta A| \lesssim 3n\epsilon_{\text{mach}}|\hat{L}||\hat{U}|$, assuming $\hat{L}$ and $\hat{U}$ are the computed L and U factors.

I will now briefly sketch a part of the error analysis following Demmel's treatment (§2.4.2, *Applied Numerical Linear Algebra*). Here is the idea. Suppose $\hat{L}$ and $\hat{U}$ are the computed L and U factors. We obtain $\hat{u}_{jk}$ by repeatedly subtracting $l_{ji}u_{ik}$ from the original a_{jk}, i.e.

$$\hat{u}_{jk} = \text{fl}\left(a_{jk} - \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}\right).$$

Regardless of the order of the sum, we get an error that looks like

$$\hat{u}_{jk} = a_{jk}(1 + \delta_0) - \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}(1 + \delta_i) + O(\epsilon_{\text{mach}}^2)$$

where $|\delta_i| \leq (j-1)\epsilon_{\text{mach}}$. Turning this around gives

$$\begin{aligned} a_{jk} &= \frac{1}{1+\delta_0}\left(\hat{l}_{jj}\hat{u}_{jk} + \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}(1+\delta_i)\right) + O(\epsilon_{\text{mach}}^2) \\ &= \hat{l}_{jj}\hat{u}_{jk}(1-\delta_0) + \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}(1+\delta_i-\delta_0) + O(\epsilon_{\text{mach}}^2) \\ &= \left(\hat{L}\hat{U}\right)_{jk} + E_{jk}, \end{aligned}$$

where

$$E_{jk} = -\hat{l}_{jj}\hat{u}_{jk}\delta_0 + \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}(\delta_i - \delta_0) + O(\epsilon_{\text{mach}}^2)$$

is bounded in magnitude by $(j-1)\epsilon_{\text{mach}}(|L||U|)_{jk} + O(\epsilon_{\text{mach}}^2)$. A similar argument for the components of $\hat{L}$ yields

$$A = \hat{L}\hat{U} + E, \text{ where } |E| \leq n\epsilon_{\text{mach}}|\hat{L}||\hat{U}| + O(\epsilon_{\text{mach}}^2).$$

In addition to the backward error due to the computation of the LU factors, there is also backward error in the forward and backward substitution phases, which gives the overall bound Equation 1.

Partial pivoting

We saw at the end of the last lecture that even when pivoting is not strictly necessary in exact arithmetic, it may be desirable in floating point to prevent the elements of L and U from becoming much larger in magnitude than the elements of A — particularly given the backward error analysis described in the previous section.

If we wish to control the multipliers (the elements of L), it's natural to choose a permutation P so that each multiplier has magnitude at most one. This standard choice leads to the following algorithm:

```
# Return L, U, p s.t. A[p,:] = L*U and the largest entry of L has magnitude 1
function my_pivoted_lu(A)

    n = size(A)[1]
    A = copy(A)         # Make a local copy to overwrite
    piv = zeros(Int, n) # Space for the pivot vector
    piv[1:n] = 1:n

    for j = 1:n-1

        # Find ipiv >= j to maximize |A(i,j)|
        ipiv = (j-1)+findmax(abs.(A[j:n,j]))[2]

        # Swap row ipiv and row j and record the pivot row
        A[ipiv,:], A[j,:] = A[j,:], A[ipiv,:]
        piv[ipiv], piv[j] = piv[j], piv[ipiv]

        # Compute multipliers and update trailing submatrix
        A[j+1:n,j] /= A[j,j]
        A[j+1:n,j+1:n] -= A[j+1:n,j]*A[j,j+1:n]'

    end

    UnitLowerTriangular(A), UpperTriangular(A), piv
end
```

By design, this algorithm produces an L factor in which all the elements are bounded by one. But what about the U factor? There exist pathological matrices for which the elements of U grow exponentially with n. But these examples are extremely uncommon in practice, and so, in general, Gaussian elimination with partial pivoting does indeed have a small backward error. Of course, the pivot growth is something that we can monitor, so in the unlikely event

that it *does* look like things are blowing up, we can tell there is a problem and try something different.

When problems do occur, it is more frequently the result of ill-conditioning in the problem than of pivot growth during the factorization.

Residuals and iterative refinement

If we know A and b, a reasonable way to evaluate an approximate solution $\hat{x}$ is through the residual $r = b - A\hat{x}$. The approximate solution satisfies

$$A\hat{x} = b + r,$$

so if we subtract off $Ax = b$, we have

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

We can use this to get the error estimate

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

but for a given $\hat{x}$, we also actually have a prayer of *evaluating* $\delta x = A^{-1}r$ with at least some accuracy. It's worth pausing to think how novel this situation is. Generally, we can only *bound* error terms. If I tell you "my answer is off by just about 2.5," you'll look at me much more sceptically than if I tell you "my answer is off by no more than 2.5," and reasonably so. After all, if I knew that my answer was off by nearly 2.5, why wouldn't I add 2.5 to my original answer in order to get something closer to truth? This is exactly the idea behind *iterative refinement*:

1. Get an approximate solution $A\hat{x}_1 \approx b$.
2. Compute the residual $r = b - A\hat{x}_1$ (to good accuracy).
3. Approximately solve $A\,\delta x_1 \approx r$.
4. Get a new approximate solution $\hat{x}_2 = \hat{x}_1 + \delta x_1$; repeat as needed.