Computer System Support for Scientific and Engineering Computation

Lecture 19 - July 5, 1988 (notes revised June 14, 1990)

1 Exception Handling

The IEEE standard defines 5 classes of exceptions. Here is a brief description. For more details see *Why Must* $0^0 = 1$ *?*

Inexact This means that the result had a rounding error. Only IEEE standard conforming machines require that hardware detect this kind of exception and signal it.

Underflow This is what you get when the computed result is smaller than the smallest representable floating point number. The VAX can trap on underflow, but the trap enable bit is reset on every procedure call.

Overflow This is what you should get when you try to compute $\exp(10^{10})$. In the IEEE standard, overflow returns ∞ and sets the inexact flag. IBM/370 FORTRAN can set the result to the largest representable number.

Divide by Zero This includes anything that is exactly equal to ∞, such as $\log 0$ and $\arctan 1$, created from finite operands in one operation.

Invalid Examples are referencing the VAX reserved operand, values outside the domain of a function, indefinite results like 0/0, and invalid control. Invalid control might include things like dereferencing a null pointer, an out of bound array reference, or operating on a matrix with negative dimensions.

There is such a diversity in how the common machines handle these exceptions, that it is probably hopeless for portable software to do anything other than try to avoid exceptions at all costs. The details of this diversity will be presented in a future lecture. For now, consider the role played by exceptions in an important piece of software, namely linear equation solvers.

2 Gaussian Elimination

Despite 30 years of intensive work on algorithms for numerically solving systems of linear equations, we still don't know of an algorithm that works reliably for all problems, yet is efficient. Here are some of the issues.

2.1 Scaling

Consider the matrix

$$\begin{pmatrix} 10^{-98} & 10^{-99} \\ 10^{-98} & 1.5 \times 10^{-99} \end{pmatrix}$$

on a decimal machine without gradual underflow that can represent nonzero magnitudes between $9.99\ldots9 \times 10^{99}$ and 1.0×10^{-99}. Performing Gaussian elimination results in the singular matrix

$$\begin{pmatrix} 10^{-98} & 10^{-99} \\ 0 & 0 \end{pmatrix}$$

where the 0 in the lower right hand corner isn't really 0, but underflowed to 0. This is completely different from what you get if you first scale the matrix by 10^{99} and then perform Gaussian elimination. The result transforms

$$\begin{pmatrix} 10 & 1 \\ 10 & 1.5 \end{pmatrix}$$

into

$$\begin{pmatrix} 10 & 1 \\ 0 & 0.5 \end{pmatrix}$$

which is a perfectly reasonable matrix. Unfortunately, no one knows a scaling algorithm that is both efficient and which always picks reasonable scale factors. This is why LINPACK does not perform scaling, leaving users vulnerable to wrong answers due to avoidable underflow.

When solving a system $A\vec{x} = \vec{b}$, scaling the matrices A and b by α merely scales the data without changing the result. In fact, we can scale each row and column of A separately. If we let

$$\text{diag}(\lambda) = \begin{pmatrix} \lambda_1 & 0 & & 0 \\ 0 & \lambda_2 & & 0 \\ & & \ddots & \\ 0 & 0 & & \lambda_n \end{pmatrix}$$

then $A \cdot \text{diag}(\lambda)$ multiplies the jth column of A by λ_j, and $\text{diag}(\mu) \cdot A$ multiplies the ith row of A by μ_i. So if we rewrite $A\vec{x} = \vec{b}$ as $(\text{diag}(\mu) \cdot A \cdot \text{diag}(\lambda))(\text{diag}(\lambda^{-1}) \cdot \vec{x}) = \text{diag}(\mu) \cdot \vec{b}$, we see that rescaling the matrix A by multiplying the (i,j)th entry by $\mu_i\lambda_j$ simply scales $\vec{x}$ and the solution vector $\vec{b}$ by λ^{-1} and μ respectively. The scale factors should be powers of the radix, to prevent roundoff errors.

Using arbitrary scaling, let us try a more interesting problem. Consider

$$\begin{pmatrix} 1 & -2 & 1 \\ -1 & 10^{-20} & 10^{-20} \\ 1 & 10^{-20} & -10^{-20} \end{pmatrix}$$

Unless the machine precision is so high that it can hold 20 decimal digits, Gaussian elimination will yield the singular matrix

$$\begin{pmatrix} 1 & -2 & 1 \\ 0 & -2 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

If instead we scale the rows by $(10^{-10}, 10^{10}, 10^{10})$ and the columns by $(10^{-10}, 10^{10}, 10^{10})$ we get the matrix

$$\begin{pmatrix} 10^{-20} & -2 & 1 \\ -1 & 1 & 1 \\ 1 & 1 & -1 \end{pmatrix}$$

This matrix has a problem: the upper left hand entry is not a good choice for a pivot. If we rearrange the rows to

$$\begin{pmatrix} -1 & 1 & 1 \\ 1 & 1 & -1 \\ 10^{-20} & -2 & 1 \end{pmatrix}$$

this reduces to the very reasonable matrix

$$\begin{pmatrix} -1 & 1 & 1 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

One of the best scaling algorithms is due to Curtis and Reid (1972), however in order to find the scale factors you need to solve a linear system which is sometimes almost as difficult as the original problem!

2.2 Roundoff Error

When we perform Gaussian elimination, we perform a series of row operations, where a row operation consists of adding a multiple of one row to another row below it. This is the same as multiplying on the left by a matrix. For example, adding λ times the first row to the second is the same as multiplying by

$$\Lambda_1 = \begin{pmatrix} 1 & 0 & & 0 \\ \lambda & 1 & & 0 \\ & & \ddots & \\ 0 & 0 & & 1 \end{pmatrix}$$

When Gaussian elimination is complete, we have an upper triangular matrix U, that is $\Lambda_n \Lambda_{n-1} \ldots \Lambda_1 A = U$. Each Λ_i is lower triangular, with 1's on the diagonal. Since the product of two such matrices is another matrix of the same form, we have $\Lambda A = U$. Also, the inverse of such a lower triangular matrix is another such matrix, so $\Lambda^{-1} = L$ and $A = LU$ decomposes A into the product of a lower triangular matrix with 1's on the diagonal and an upper triangular matrix. A backward error analysis of Gaussian elimination shows that the computed matrices L and U satisfy the equation $LU = A + E$ exactly, where E represents the rounding error, and E is bounded by the roundoff in the maximum element that occurs in place of A during Gaussian elimination. [1] That explains why it is not a good idea to use a small element as a pivot: it will have to be multiplied by a large element in order to cancel other elements in its column, and that will most likely generate large elements

[1] Roughly speaking, the reason is this. As we perform Gaussian elimination, we replace elements of A with other elements, until the upper triangular half of A is filled with the elements of U. If we always choose the largest possible pivot, then the multipliers used to manipulate A are always less than 1. Thus the rounding error committed in computing U is bounded by the rounding error of the largest element that appears in place in A.

elsewhere in the matrix. How large can elements grow to be? The following example was discovered independently by Wilkinson and Kahan around 1959.

$$\Lambda_x = \begin{pmatrix} 1 & 0 & 0 & & 0 & 1 \\ -1 & 1 & 0 & & 0 & 1 \\ -1 & -1 & 1 & & 0 & 1 \\ & & & \ddots & & \\ -1 & -1 & -1 & & 1 & 1 \\ -1 & -1 & -1 & & -1 & x \end{pmatrix}$$

It has 1's on the diagonal (except for the last element), 0's in the upper triangle, -1's in the lower triangle, and 1's along the rightmost column. After the first step of Gaussian elimination, the right hand column will contain 2's from the second row on. After the second step, it will contain 4's from third row on, and so on. When Gaussian elimination is complete, the lower right hand element will be $2^{n-1} - 1 + x$. So error analysis suggests that rounding error will be severe. And if $x = 0$ and n is larger than the floating point precision, the -1 will be lost to roundoff error, so it will be as if $x = 1$. The inverse of this matrix can be computed exactly, it is the difference of two matrices

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 2 & 1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 4 & 2 & 1 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & & & \vdots & \vdots & \vdots \\ 2^{n-3} & 2^{n-4} & 2^{n-5} & \cdots & & 1 & 1 & 0 \\ 0 & 0 & 0 & \cdots & & 0 & 0 & 0 \end{pmatrix} - \frac{1}{\alpha} \begin{pmatrix} 2^{n-2} & 2^{n-3} & 2^{n-4} & \cdots & 2 & 1 & 1 \\ 2^{n-1} & 2^{n-2} & 2^{n-3} & \cdots & 4 & 2 & 2 \\ 2^{n} & 2^{n-1} & 2^{n-2} & \cdots & 8 & 4 & 4 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 2^{2n-4} & 2^{2n-3} & 2^{2n-2} & \cdots & 2^{n-1} & 2^{n-2} & 2^{n-2} \\ 2^{n-2} & 2^{n-3} & 2^{n-4} & \cdots & 2 & 1 & 1 \end{pmatrix}$$

where $\alpha = 2^{n-1} - 1 + x$. Since $1/(2^{n-1} - 1) \approx 1/2^{n-1} + 1/2^{2n-2}$, changing x from 0 to 1 will perturb the inverse by 2^{-2n+2} times the second matrix. For example, the second to last row will change from $(0, 0, 0, \ldots)$ to $(\frac{1}{4}, \frac{1}{8}, \ldots)$. Since the solution to $A\bar{x} = \bar{b}$ is $\bar{x} = A^{-1}\bar{b}$, this change to A^{-1} will result in a dramatic change to the solution.

When performing Gaussian elimination, the usual procedure is at each step to select the row that will give the largest pivot. This is called *partial pivoting*. Scaling the matrix results in selecting different pivots, so finding a good scaling algorithm is equivalent to finding a good pivoting strategy.

Selecting the row that gives the best pivot is equivalent to permuting the rows of the matrix and then picking the pivots in order. This changes the LU decomposition from $LU = A + E$, to $LU = PA + E'$, where P is a permutation matrix. Hopefully if we pick the correct P, the error E' will be negligible. We mentioned earlier that the error E is bounded by the maximum element that occurs in place in A during Gaussian elimination. It turns out that with partial pivoting, the elements of A grow by a factor of at most 2^{n-1}. And the example above shows that the bound can be obtained.

If we also reorder the columns as wells as the rows (*complete pivoting*), then Wilkinson proved that the growth is bounded by roughly $n^{(\log n)/4}$. However, no one knows if this is best possible, and in fact the largest known growth is linear. For $n \leq 4$, it has been proven that the growth is at most n. For random matrices, the growth appears to be $\sqrt{n}$ on average; for real life matrices the maximum often shrinks! It is extremely unusual for the elements to grow by more than a factor of 1000. The code HUPA and LUPA perform partial pivoting, and print out a message if the growth is more than $8n$. No one has ever reported seeing the message except for matrices like Λ_x.

Even if we knew an optimal pivoting strategy, for sparse matrices you would like to choose pivots to retain the sparse structure. There are papers by Skeel (197x) and Arioli, Demmel and Duff (1988) that show that iterative refinement can undo the roundoff damage created by a not too poor choice of pivots, even when the residual is computed in working precision rather than more. Thus iterative refinement, which was out of fashion in the seventies, may come back into fashion. Another way to correct for a poor choice of pivots (or scaling) is to provide extra precision. An extra 10 bits of precision would compensate for a growth of the elements of A by a factor of $2^{10} = 1024$. And in fact, IEEE extended precision provides 10 extra bits of significand over double precision.[2]

2.3 Zero Pivots

Suppose that during Gaussian elimination all the possible pivots are zero. Many programs give up at this point, announcing that the matrix was singular. There are two reasons why this is not a good idea. The first is that sometimes you want to solve an equation with a singular matrix. For example in the *inverse iteration* method for computing eigenvectors, you solve the equation $(A - \lambda I)x = y$, where λ is the eigenvalue, and y is an estimate for the eigenvector. The second reason is that finding a zero pivot is not a good test of whether a matrix is singular, because of roundoff error. The condition of A, $\text{cond}(A) = \|A\| \, \|A^{-1}\|$ is the correct way of estimating how singular a matrix is, and in particular, how close the computed solution is to the true solution. Remember that backward error analysis only says that the computed solution is the exact solution of a slightly perturbed problem. It doesn't say that the computed solution is a slight perturbation of the true solution. If ϵ is the distance between 1 and the next representable number, than the relative error in Gaussian elimination is about $\epsilon \cdot \text{cond}(A)$.

Since the determinant of a matrix is zero exactly when the matrix is singular, you might think that estimating the size of the determinant (which is the product of the diagonal elements of U from the LU decomposition) would be a good estimate of the condition. But it is not. The matrix

$$\begin{pmatrix} 10^{-1} & 0 & & 0 \\ 0 & 10^{-1} & & 0 \\ & & \ddots & \\ 0 & 0 & & 10^{-1} \end{pmatrix}$$

has determinant 10^{-n} but condition 1. The matrix

$$\begin{pmatrix} 1 & -1 & -1 & & -1 & -1 \\ 0 & 1 & -1 & & -1 & -1 \\ 0 & 0 & 1 & & -1 & -1 \\ & & & \ddots & & \\ 0 & 0 & 0 & & 1 & -1 \\ 0 & 0 & 0 & & 0 & 1 \end{pmatrix}$$

has condition $n2^{n-1}$ but determinant 1. So there is no correlation between the condition number and the determinant.[3] Estimating the condition by computing A^{-1} is expensive.

[2] Incidentally, the reason why extended precision has 64 bits of precision, is because that (plus 3 bits for guard, round and sticky) was the widest precision across which carry propagation could be done on the Intel 8087 without increasing the cycle time.

[3] However, when $n = 2$, the condition number and determinant are comparable in size.

LINPACK instead uses a condition estimator based on finding a special vector d and solving $Ax = d$.

This suggests that when Gaussian elimination discovers a small pivot, it should continue and compute the LU decomposition anyway. From the LU decomposition the condition can be estimated, giving an estimate of the accuracy of the solution to the linear system. In addition, the LU decomposition can be used to find eigenvectors. In the HUPA and LUPA codes, if a zero pivot is discovered, the pivot is replaced with ϵ times the largest element in its column. This would be a useful fact to record in a log of retrospective diagnostics.

3 Zero Finding and Quadrature

Linear algebra codes are generally quite robust, but good zero finding routines are more likely to depend on the pecularity of the machine they are running on. The root finder on the HP 28C, 28S and 19C calculators can handle functions with singularities and restricted domains. When the root finder gets a NaN in response to evaluating the function, it refines its guess until the function returns a valid answer. This obviously requires that the hardware support NaNs. It can be done on a Vax, using its reserved operand, but it is painful because reserved operands cause too many traps. On an IBM/370, every bit pattern represents a floating point number, so supporting NaNs is impossible.

Another problem with root finders is knowing when to stop the iteration. Doing this correctly requires knowledge about the distribution of floating point numbers. The recommended IEEE function **nextafter** is useful for this. Having control over rounding is also very helpful. Another example where the distribution of floating point numbers is crucial is in quadrature programs. When integrating a function with a singularity, it is important not to sample the function at a singularity. For functions with vertical tangents, the precise point where the function is evaluated is crucial (since the derivative is ∞). Once again, knowing the distribution of floating point numbers is very important. Another problem with quadrature programs occurs when integrating functions with long tails. It is possible to get underflow far out in the tail, but very long tails can contain a lot of area. Gradual underflow helps here.

To summarize, accurate algorithms for finding zeros and quadrature benefit from features of the IEEE standard. However, rather than create clever algorithms that exploit the IEEE standard, it is also possible to solve these problems using multiple precision.

4 Multiple Precision

In lecture 17, we discussed representing multiple precision floating point numbers as an array of working precision numbers. We gave a distillation algorithm that will collapse the sum of two such arrays. In order to multiply two such arrays, we need to be able to represent the product of two working precision numbers as a sum of working precision numbers. On some machines (IBM/370 and Cyber 17x) there is a single instruction[4] that computes a full product, placing it in two words. The problem is that portable software isn't able to use this instruction, because when multiplying two numbers corresponding to the largest floating point data type, there is no floating point type available to assign the product. John Cocke (1967) has suggested a multiply/add instruction that would compute $A \cdot B + C$ exactly and then round the result. If a machine had such an instruction, and if the compiler recognized

[4] Actually two instructions on the Cyber.

the expression `A*B - C`, then products could be computed portably. First compute `P = A*B` to working precision. Then compute `Q = A*B - P`. The two variables `P` and `Q` contain the two halves of $A \cdot B$.

There is a way to compute a full product portably, which was discovered by Dekker and Veltkamp. It involves spliting a working precision variable into two halves. This will be discussed in lecture 20.