2. Miscellaneous notes on numerical computing

2.1. The condition of a mathematical problem

Broadly speaking, the condition of a mathematical problem is a measure that reflects the worst-case sensitivity of its exact solution to changes in the problem.

Assume that problem $P$ is defined by a set of data, say $d$, and let $s(d)$ denote the exact solution of the problem for the data $d$. If “small” changes in $d$ lead to “small” changes in $s(d)$, the problem is said to be well-conditioned for the data $d$. Otherwise, if small changes in $d$ lead to large changes in $s(d)$, the problem is said to be ill-conditioned.

Suppose that $d_1$ and $d_2$ are two possible sets of data. The condition (or condition number) of problem $P$ is a measure of the maximum possible ratio

$$\max \frac{\|s(d_1) - s(d_2)\|}{\|d_1 - d_2\|}$$

when $\|d_1 - d_2\|$ is small.

The crucial point to remember is that the condition of a problem is a mathematical property, and is independent of computation and rounding error.

As a simple illustration of condition, consider the problem of determining the roots of the polynomial $(x - 1)^4 = 0$, whose four roots are all exactly equal to 1. Suppose that a small perturbation (say, $10^{-8}$) is made in the right-hand side, so that the equation to be solved is now

$$(x - 1)^4 = 10^{-8}. \quad (\star)$$

One exact root of the perturbed problem is $1 + 10^{-2}$, which has changed by $10^{-2}$ from the exact root of the original problem. The perturbed problem is arguably close to the original, yet the change in the exact solution is six orders of magnitude larger than the change in the problem!

A problem is said to be ill-conditioned when small changes in the problem data can lead to large changes in the exact solution. A problem is well-conditioned when small changes in the problem data always lead to small changes in the exact solution.

Obvious questions: how to define “small” and “large”? How to define $d$?
A famous example is a notorious polynomial, first analyzed in this context by J. H. Wilkinson in the late 1940s (the early days of digital computation). Let $W(x)$ (the Wilkinson polynomial) be the following polynomial of degree 20:

$$W(x) = (x - 1)(x - 2) \cdots (x - 19)(x - 20)$$

$$= x^{20} - 210x^{19} + \cdots + 20!,$$

whose exact roots are obviously the first 20 positive integers, 1, 2, \ldots, 20.

Suppose that the coefficient of $x^{19}$ in $W(x)$ (whose value is $-210$) is changed by $2^{-23} \approx 1.19209 \times 10^{-7}$, i.e., the mathematical problem is altered by introducing a relative perturbation of order $10^{-9}$ in one coefficient. The exact roots of the perturbed polynomial $W(x) - 2^{-23}x^{19}$ (rounded to ten digits) are:

$$\begin{array}{ccc}
1.00000000 & 10.095266145 & 0.643500904i \\
2.00000000 & 11.793638381 & 1.652329728i \\
3.00000000 & 13.99258137 & 2.518830070i \\
4.00000000 & 16.730737466 & 2.812624894i \\
4.99999928 & 19.502439400 & 1.940330347i \\
6.00000694 & 10.095266145 & 0.643500904i \\
6.999697234 & 11.793638381 & 1.652329728i \\
8.007267603 & 13.99258137 & 2.518830070i \\
8.917250249 & 16.730737466 & 2.812624894i \\
20.84690810 & 19.502439400 & 1.940330347i \\
\end{array}$$

Not only have some of the real roots changed almost beyond recognition, but ten of the roots have moved into the complex plane. We emphasize that the above are rounded versions of the exact roots of the perturbed polynomial.

The definition of condition seems to assume that we know $s(d)$ (the exact solution), but of course in general we won’t. So we need to think about how to determine whether an $s$ that is handed to us is a “good” solution.

### 2.2. Computable measures of goodness

The ideal computable measure of goodness would be (of course) computable, and it would be exactly zero when we are presented with the exact solution. But it sometimes happens that the obvious computable measure of goodness does not vanish at the exact solution—or that a computable measure of goodness is much smaller at a point far from the solution than close to the solution. Nonetheless, if we can develop an upper bound on the computable measure of goodness at the exact solution, this helps us to set a standard for measuring the quality of a given solution.

An example of the complications: we want to solve $Ax = b$ for a given matrix $A$ and vector $b$. The residual for the linear system $Ax = b$ is the vector $r = b - Ax$, which is exactly zero if $x$ is the exact solution, and we can compute the residual if we have a possible solution to hand.
To illustrate the complications, consider the $2 \times 2$ linear system
\[
\begin{align*}
.550x_1 + .423x_2 &= .127 \\
.484x_1 + .372x_2 &= .112,
\end{align*}
\]
which corresponds to
\[
Ax = b \quad \text{with} \quad A = \begin{pmatrix} .550 & .423 \\ .484 & .372 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} .127 \\ .112 \end{pmatrix}.
\]

The exact solution is easily seen to be $x^* = (1, -1)^T$. Suppose we have two candidate solutions,
\[
\tilde{x} = \begin{pmatrix} 1.7 \\ -1.91 \end{pmatrix} \quad \text{and} \quad \bar{x} = \begin{pmatrix} 1.01 \\ -.99 \end{pmatrix}.
\]

The residuals for $\tilde{x}$ and $\bar{x}$ are
\[
\tilde{r} = b - A\tilde{x} = -\begin{pmatrix} .00007 \\ .00028 \end{pmatrix}, \quad \bar{r} = b - A\bar{x} = -\begin{pmatrix} .00973 \\ .00856 \end{pmatrix}.
\]

Let $\|\cdot\|$ denote the component of the vector with maximum absolute value; this is the infinity norm of a vector. Then
\[
\|b - A\tilde{x}\|_2 = 2.8 \times 10^{-4} \quad \text{and} \quad \|b - A\bar{x}\| = 9.73 \times 10^{-3}.
\]
Comparing the norms suggests that $\tilde{x}$ is a “better” solution $\tilde{x}$ because it has a smaller residual. But $\|\tilde{x} - x\| = .91$ and $\|\bar{x} - x\| = .01$, so that $\tilde{x}$ is much closer to $x^*$, even though its residual is 35 times larger.

The main point is that it is difficult to define computable measures of goodness, especially for ill-conditioned problems, and also difficult to interpret them.

2.3. Stability of an algorithm

Intuitively, a stable algorithm will be well behaved—but what does this mean exactly?

**Forward stability.** Suppose that problem $P(d)$, defined by $d$, has a unique exact solution $s(d)$. Suppose that the result of applying a certain algorithm is a computed solution $\tilde{s}(d)$ (the computed solution wears a crumpled hat). The process of *forward error analysis* involves studying whether $\tilde{s}(d)$ is guaranteed to be “close” to $s$, i.e., we look at
\[
\|\tilde{s}(d) - s(d)\|.
\]
When this norm is “small” for all $d$, the algorithm is said to be *forward stable.*
Forward error analysis seems “natural” because it tells how much the computed solution may differ from the true solution. Unfortunately, forward error analysis often becomes complicated even for simple procedures, and the resulting bounds are frequently disappointing, in large part because the problem to which the algorithm is applied may be ill-conditioned.

For example, an algorithm that carries out every calculation exactly, except for a single tiny error, would certainly be considered “good”. But if this algorithm were applied to a very ill-conditioned problem $P$, then even one tiny error might, in the worst case, produce a huge difference between the computed and exact solutions of $P$. A sad fact: in general one cannot expect an accurate answer to an ill-conditioned problem, where “accurate” means that the computed solution is close to the exact solution.

**Backward stability.** Another form of stability analysis, pioneered and popularized largely by J. H. Wilkinson [of the polynomial], involves backward stability (and backward error analysis). The key insight here is to reflect the solution backward to the data rather than forward to the solution.

Let $\text{alg}(d)$ denote the solution obtained by an algorithm applied to problem $P$ with data $d$. We do not look at $\|s(d) - \text{alg}(d)\|$. Rather, we try to show how $\text{alg}(d)$ is related to the exact solution of $P$ with perturbed data, say $d + \delta d$, where $\|\delta d\|$ is small.

Suppose, for example, that

$$\text{alg}(d) = s(d + \delta d),$$

where $\|\delta d\|$ is small.

When this is so, it is common to say that the algorithm produces the the exact solution of a nearby problem, and the associated algorithm is called backward stable (or numerically stable). Normally, we won’t know $\delta d$, but we can often develop a bound on its norm.

A way to remember the difference between forward and backward stable: a history exam (“Greek” versus “Roman”).

### 2.4. Errors

Given two quantities, we often wish to define a single number that shows how close one is to the other, i.e., to measure the size of their difference. Intuitively, a satisfactory measure of closeness should be zero when the quantities are equal, small if they are “close”, and large when they are “far apart”. In general, we tend to use these terms to mean small or large relative to unity—so that $10^{20}$ is large and $10^{-10}$ is small. Unfortunately, no single definition gives the “correct” measurement for all situations.

It is common practice to use the term “error” in this context, rather than “closeness” or “difference”, even though it may not be true that one quantity is right and the other is wrong.

If $\bar{x}$ is viewed as an approximation to $x$, the absolute error $e_A$ in the approximation is defined as

$$e_A = |x - \bar{x}|.$$
2. Miscellaneous notes on numerical computing

For example, if $x = 10.12$ and $\bar{x} = 10.05$, the absolute error is $e_A = |10.12 - 10.05| = .07$. The idea is that the *actual difference* between $x$ and $\bar{x}$ is meaningful, no matter how large or small $x$ and $\bar{x}$ may be.

To illustrate this situation, consider a long airplane ride. Regardless of the total distance traveled, the difference between the estimated and exact positions may be required to be less than a certain *absolute* quantity—for example, to avoid a collision.

A difficulty with absolute error is that it may be large simply because $|x|$ is large. If $x = 10^{10}$ and $\bar{x} = 10^{10} + 500$, the absolute error is 500, but it’s likely that $\bar{x}$ would be considered close to $x$, since they agree in the first eight figures.

The relative error of $\bar{x}$ (with respect to a nonzero value $x$) is defined as

$$e_R = \frac{|x - \bar{x}|}{|x|}.$$ 

Relative error is useful when $|x|$ itself is a meaningful measure of size.

Which error measure is best? Neither one—it depends on the context. A *mixed error measure* often used is

$$e = \frac{|x - \bar{x}|}{1 + |x|}.$$ 

### 2.5. The condition of a matrix (when solving $Ax = b$)

Suppose that we want to solve $Ax = b$ when $A$ is a nonsingular matrix.

What’s the condition of this problem, i.e., how much can the exact solution change if we perturb the data slightly? Suppose that $x$ is the exact solution of $Ax = b$, so that $x = A^{-1}b$.

What happens to the exact solution if we perturb $b$ to $b + \delta_b$, where $\delta_b$ denotes a perturbation in $b$. Let $x + \delta_x$ be the exact solution of $A(x + \delta_x) = b + \delta_b$, so that $\delta_x$ is the change in $x$. Then, since $Ax + A\delta_x = b + \delta_b$ and $Ax = b$, we have

$$A\delta_x = \delta_b,$$

so that $\delta_x = A^{-1}\delta_b$.

Using norm inequalities (assuming a subordinate matrix norm),

$$||\delta_x|| \leq ||A^{-1}|| ||\delta_b||.$$ 

So the change in the exact solution can undergo a perturbation whose norm is bounded above by $||A^{-1}|| ||\delta_b||$.

Since $b = Ax$, we know that $||b|| \leq ||A|| ||x||$. This means that the maximum *relative* change in the exact solution can be obtained by multiplying these inequalities and dividing through by $||x||$ and $||b||$:

$$\frac{||\delta_x||}{||x||} \leq ||A|| ||A^{-1}|| \frac{||\delta_b||}{||b||}.$$
Thus the maximum possible relative change in the exact solution is the relative change in the right-hand side (RHS) multiplied by \( \|A\| \|A^{-1}\| \).

Now we examine the effect of perturbing the matrix by \( \Delta_A \), assuming that \( A + \Delta_A \) is nonsingular, and seek to bound the size of the associated change \( \delta_x \) in the exact solution:

\[
(A + \Delta_A)(x + \delta_x) = b.
\]

Some simple rearrangement gives

\[
\delta_x = -A^{-1}\Delta_A(x + \delta_x), \quad \text{so that} \quad \|\delta_x\| \leq \|A^{-1}\| \|\Delta_A\| \|x + \delta_x\|.
\]

Multiplying and dividing by \( \|A\| \) and dividing by \( \|x + \delta_x\| \) gives

\[
\frac{\|\delta_x\|}{\|x + \delta_x\|} \leq \|A\| \|A^{-1}\| \frac{\|\Delta_A\|}{\|A\|}.
\]

The same quantity, \( \|A\| \|A^{-1}\| \), appears when we perturb either the right-hand side or the matrix, and is called the condition of \( A \), denoted by \( \text{cond}(A) \):

\[
\text{cond}(A) = \|A\| \|A^{-1}\|.
\]

For any subordinate matrix norm, \( \text{cond}(A) \geq 1 \), since \( I = AA^{-1} \) and we know that \( \|I\| = 1 \) in any subordinate norm:

\[
\|A\| \|A^{-1}\| \geq \|I\| = 1.
\]

A crucial point about \( \text{cond}(A) \): it expresses the worst possible relative change in the exact solution of a linear system following a perturbation in the matrix or the right-hand side. But, even if \( A \) is ill-conditioned, there will be specific cases where the worst does not happen—but it can happen in other cases.

Let’s return to an earlier example, with the matrix

\[
Ax = b \quad \text{with} \quad A = \begin{pmatrix} .550 & .423 \\ .484 & .372 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} .127 \\ .112 \end{pmatrix}.
\]

The exact solution is \( x^* = (1, -1)^T \), and we considered two possible solutions:

\[
\tilde{x} = \begin{pmatrix} 1.7 \\ -1.91 \end{pmatrix} \quad \text{and} \quad \bar{x} = \begin{pmatrix} 1.01 \\ -0.99 \end{pmatrix},
\]

for which

\[
b - A\tilde{x} = -\begin{pmatrix} 7 \times 10^{-5} \\ 2.8 \times 10^{-4} \end{pmatrix}, \quad b - A\bar{x} = -\begin{pmatrix} 9.73 \times 10^{-3} \\ 8.56 \times 10^{-3} \end{pmatrix}.
\]

The point was that, although \( \bar{x} \) is much closer to the exact solution than \( \tilde{x} \), the size of \( \|b - A\tilde{x}\| \) is much larger than \( \|b - A\bar{x}\| \). Here are some calculations showing the extreme possible effects of perturbations in the given \( b \) for this problem.
2. Miscellaneous notes on numerical computing

\[ \text{norm}(A) = 9.2461e-01 \]
\[ A^{-1} = \begin{bmatrix} -2.8182e+03 & 3.2045e+03 \\ 3.6667e+03 & -4.1667e+03 \end{bmatrix} \]
\[ \text{norm}(A^{-1}) = 7.0012e+03 \]
\[ \kappa = \text{cond}(A) = 6.4702e+03 \]
\[ \text{svd} \text{delb}_1 = 0.001*\text{ulast} = \begin{bmatrix} -6.6054e-04 & 7.5079e-04 \end{bmatrix} \]
\[ \text{norm} (\text{svd} \text{delb}_1) = 1.0000e-03 \]
\[ b_{\text{svd} \text{del}1} = b + \text{svd} \text{delb}_1 = \begin{bmatrix} 1.2634e-01 & 1.1275e-01 \end{bmatrix} \]
\[ \text{norm}(b_{\text{svd} \text{del}1} - b) = 1.0000e-03 \]
\[ x_{\text{svd} \text{del}1} = A \backslash b_{\text{svd} \text{del}1} = \begin{bmatrix} 5.2675e+00 & -6.5503e+00 \end{bmatrix} \]
\[ \text{norm}(x_{\text{svd} \text{del}1} - x) = 1.0000e-03 \]
\[ \text{norm}(x_{\text{svd} \text{del}1} - x)/\text{norm}(x) = 4.9506e+00 \]
\[ \text{svd} \text{delb}_2 = -7.5079e-04 \]
\[ \text{norm}(\text{svd} \text{delb}_2) = 1.0000e-03 \]
\[ b_{\text{svd} \text{del}2} = b + \text{svd} \text{delb}_2 = \begin{bmatrix} 1.2625e-01 & 1.1134e-01 \end{bmatrix} \]
\[ x_{\text{svd} \text{del}2} = A \backslash b_{\text{svd} \text{del}2} = \begin{bmatrix} 9.9914e-01 & -1.0007e+00 \end{bmatrix} \]
\[ \text{norm}(x_{\text{svd} \text{del}2} - x) = 1.0821e-03 \]
\[ \text{norm}(x_{\text{svd} \text{del}2} - x)/\text{norm}(x) = 7.6514e-04 \]

**A frequent misunderstanding about** \( \text{cond}(A) \).

Since \( \det(A) = 0 \) when \( A \) is singular, it is sometimes wrongly concluded that \( A \) is ill-conditioned if \( \det(A) \) is small.

But this is not correct; consider
\[
A = \begin{pmatrix} 10^{-10} & 0 \\ 0 & 10^{-10} \end{pmatrix} = 10^{-10}I.
\]
Here, \( \|A\| = 10^{-10} \) and \( \|A^{-1}\| = 10^{10} \), so that \( \text{cond}(A) = 1 \) and \( A \) is perfectly conditioned.

On the other hand, if \( \det(A) \) is “large”, it does not follow that \( A \) is well conditioned. For example, let
\[
A = \begin{pmatrix} 10^4 & 0 \\ 0 & 10^{-3} \end{pmatrix}.
\]

For this matrix \( \det(A) = 10 \), but \( \|A\| = 10^4 \) and \( \|A^{-1}\| = 10^3 \), so that \( \text{cond}(A) = 10^7 \).
Conclusion: without various constraints on the structure and nature of $A$, the size of the determinant contains information about $\text{cond}(A)$ only if $\det(A) = 0$.

If we want to know whether a matrix is ill-conditioned or well-conditioned, we need to find (if possible) alternative ways to determine if $A$ is ill-conditioned. In general, we cannot compute $\text{cond}(A)$ directly, but it can be estimated, often by obtaining bounds.

A bit of “trashing the inverse”.

When we want to solve $Ax = b$, it is almost never a good idea to compute $A^{-1}$ and multiply it by $b$. Using GEPP to calculate $A^{-1}$, one needs to calculate LU and then to solve $LUX = I$, where $X$ is the computed inverse.

Let $x_i$ be the $i$th column of $A^{-1}$, so we have

$$X = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix} \text{ where } LUx_j = e_j.$$ 

Thus we need to solve $L(Ux_j) = Ly_j = e_j$ and $UX_j = y_j$ for $j = 1, \ldots, n$. The systems $Ly_j = e_j$ have a special form because the RHS is a coordinate vector. The total cost (in flops) of computing the inverse is about three times the number of flops needed to solve $Ax = b$.

In terms of backward error analysis, each computed column $x_j$ of $A^{-1}$ is the exact solution of a nearby problem

$$(A + E_j)x_j = e_j,$$

but $E_j$ varies with $j$. We CANNOT say that the computed $X$ is the exact inverse of a nearby matrix. In extreme cases, the computed inverse may be singular, so that it is not the inverse of any matrix (and hence not the exact solution of a nearby problem).

If $X$ is nonsingular, we may be able to find $E$ such that $(A + E)X = I$, but there is no guarantee that $\|E\|$ is small.

A final disadvantage of the inverse, especially when matrices are large, is that the inverse is highly unlikely to be sparse or structured when $A$ is, even though the LU factors are almost always sparse and structured.

2.6. Floating-point calculation

In nearly all scientific computing, calculations with non-integer real numbers are performed using floating-point arithmetic, an analogue of scientific notation. So we say $2.5 \times 10^{-3}$ rather than .0025. Usually, when dealing human to human, we use decimal arithmetic, but computers don’t have fingers and toes—using base 2 (or a power of 2) is easier for them.

Most computers today choose the base $b$ of their floating-point systems as $b = 2$ or $b = 16$ (binary or hexadecimal arithmetic). Given an integer $b \geq 2$ (the base), any scalar $\xi$ may be represented in base $b$ as

$$\xi = \pm m \times b^i,$$
where $z$ (the exponent) is a (signed) integer, and $m$ (the mantissa) is a nonnegative real number. For a general number $\xi$ and base $b$, $m$ may contain an infinite number of figures—e.g., if $b = 10$ and $\xi = 1/3$.

When $\xi$ is represented on a (finite) computer, only a finite number of figures of $m$ can be retained. (NB: We often call these “digits”, even in non-decimal number systems.) A floating-point number $x$ has the form

$$x = \pm .m_1 m_2 \ldots m_t \times b^z,$$

where $t$ is finite and each $m_i$ is an integer satisfying $0 \leq m_i < b$.

In general, $t$ (the number of figures in the mantissa) is fixed for a particular machine and level of precision. On most computers used for numerical computation, floating-point numbers may be represented in single or double precision, where $t$ (the number of figures in the mantissa) is approximately twice as large in the latter case.

On a particular machine, limits are placed on the exponent by specifying positive integers $k_1$ and $k_2$ and requiring that

$$-k_1 \leq z \leq k_2.$$

If the exponent $z$ in the exact expression for $x$ exceeds $k_2$, $x$ is said to overflow; if the exponent $z$ in the exact expression for $x$ is less than $-k_1$, $x$ is said to underflow.

A nonzero number $x$ that can be expressed in the form

$$x = \pm .m_1 m_2 \ldots m_t \times b^z \quad \text{with} \quad -k_1 \leq z \leq k_2$$

and $m_1 \neq 0$ is called a normalized representable number (with respect to particular choices of $b$, $t$, $k_1$ and $k_2$).

A number is representable only if, in its exact representation, it does not overflow or underflow, and all figures of the mantissa $m$ beyond the $t$-th are zero. Some numbers (e.g., $\pi$, $\sqrt{2}$) are not representable on any finite machine.

Other numbers are representable on some machines, but not on all; for example, $1/10$ is representable on a decimal machine (with base $b = 10$), but not on a machine whose base is a power of 2.

When we wish to compute on a computer with any number $x$, it must first be converted to a representable floating-point number (say, $\bar{x}$). This transformation is usually depicted by writing

$$\bar{x} = fl(x),$$

which indicates that $\bar{x}$ is a floating-point version of $x$. (If $x$ itself is representable, $x = fl(x)$.) Please keep in mind that $\bar{x}$ itself is a valid exact number, and not a “fuzzy” or approximate quantity!!

Assuming that overflow and underflow do not occur, $\bar{x}$ is usually obtained by either rounding or truncating (chopping) the exact representation of $x$ in base $b$:

$$x = \pm .m_1 m_2 \cdots \times b^z$$
2. Miscellaneous notes on numerical computing

The generic process of transforming a nonzero number to representable form will sometimes be called simply “rounding” when there is no danger of ambiguity.

The relation between the exact nonzero value $x$ and its representable version $fl(x)$ has the form

$$fl(x) = x(1 + \delta),$$

where $\delta$ may be positive, negative or zero. An obvious question is: how large can $|\delta|$ be?

The rounded floating-point version of $x$, denoted by $x_R$, is the $t$-figure number closest to $x$. (We assume that $x_R$ does not underflow or overflow.) Note that the exponents of $x$ and $x_R$ may differ—for example, when $.99999$ is rounded to three decimal digits.

When the exponents of $x$ and $x_R$ are the same in base $b$, $x_R$ will differ from $x$ by no more than half a unit in the last place of its mantissa.

For example, in three-digit rounded decimal arithmetic, $.3142$ will be rounded to $.314$ (an error of .2 in the last place of the mantissa), and $.3146$ to $.315$ (an error of .4 in the last place of the mantissa).

The maximum error in rounding occurs when representing a number that lies exactly midway between two representable numbers—e.g., in rounding $.3145$ to three digits. (This situation has been called the tablemaker’s dilemma, from the days when people looked things up in tables.) Some systematic rule for resolving ties, such as round to nearest even, must be used to avoid introducing bias into the results. The relative error in a correctly rounded $x_R = fl(x)$ satisfies

$$|x_R - x| \leq \frac{1}{2} b^{1-t} |x|.$$

Thus, with rounding,

$$fl(x) = x_R = x(1 + \delta_R), \quad \text{where} \quad |\delta_R| \leq \frac{1}{2} b^{1-t}.$$

Today, essentially all scientific and engineering calculations are performed with IEEE double precision arithmetic, in which relative precision is about $2.2 \times 10^{-16}$, i.e., approximately 16 correct decimal digits. This remarkable level of precision sometimes leads people to think that there are no worries any more about imprecise computation. But this is not true.

Here is a famous example from 20 years ago: in 1994 the Intel Pentium microprocessor, had been announced with great fanfare as the latest and greatest new chip. A professor at Lynchburg College, Virginia, discovered inconsistencies in his calculations with the Pentium in October 1994, and notified Intel (which later admitted it had been aware of the problem since May 1994). Word spread around the Internet about the bug (in floating-point division), and it was even discussed in a CNN segment.

Intel first tried to claim that this bug was not very important, and that they would replace the chip upon request for users who could prove that they were affected. But the public outcry continued, their stock price dropped and there were lots of Intel jokes. Eventually they offered to replace all the flawed Pentium processors, and took a pre-tax charge against earnings of $475 million in January 1995.
Two jokes:
1. Question: How many Pentium designers does it take to screw in a light bulb?
   Answer: 1.99904, and that’s good enough for us.
2. Intel’s corporate motto: Accuracy is our number 0.99912 priority.
   The main point of this example is that, even though modern computers work with vast
   precision, inaccurate results are possible.

**Arithmetic operations.** When performing even simple arithmetic operations on repre-
sentable numbers), the exact result may be non-representable. Consider four decimal digits,
and the representable numbers 1.6 and .002342. Their exact sum, 1.602342, is not repre-
sentable.

On reasonable machines with reasonable numerical properties, the result of a floating-
point operation on two representable numbers $a$ and $b$ satisfies

$$fl(a \text{ op } b) = (a \text{ op } b)(1 + \delta),$$

where “op” is one of the basic operations “+”, “−”, “×”, or “/”.

The value of $\delta$ should be small, and typically satisfies $|\delta| \leq \epsilon_M$.

A crucial point: the usual laws of arithmetic do **NOT** apply to floating point systems,
even the best ones. For example, if $x$, $y$ and $z$, are representable, the associative law

$$x + (y + z) = (x + y) + z$$

does not necessarily hold, i.e, it can happen that

$$fl(fl(x + y) + z) \neq fl(x + fl(y + z)).$$

Suppose $x = −1$, $y = 1$ and $z = .0001$ and we are using four-digit arithmetic; then

$$fl(x + y) = 0.0, \quad fl(fl(x + y) + z) = .0001,$$

but

$$fl(y + z) = 1.0, \quad fl(x + fl(y + z)) = 0.0.$$  

Similarly, in floating point it may not be true that $(\sqrt{|x|})^2 = |x|$, and other mathematical
properties may not hold. Finite precision can also affect performance of numerical algorithms,
and we need to be careful about relying on any analysis that assumes exact arithmetic.

**IEEE arithmetic.** The default choice for numerical computing today (2015) is IEEE double
precision.

IEEE (**“the world’s largest professional association for the advancement of technology”**),
said “I-triple E”, used to mean the Institute for Electrical and Electronics Engineers. Now
it doesn’t mean anything—it’s just an abbreviation (like AT&T). IEEE is best known for
developing standards for the computing and electronics industries.

IEEE arithmetic is base-2 arithmetic, so $b = 2$. Rounded arithmetic is **mandatory**.

A double-precision nonzero floating-point number $x$ occupies 64 bits:
2. Miscellaneous notes on numerical computing

- 1 bit for sign;
- 11 bits for exponent \((2^{11} = 2048)\);
- 52 bits for mantissa

and has the form

\[ x = \pm.m_1m_2\ldots m_t \times b^z \quad \text{with} \quad -k_1 \leq z \leq k_2 \]

and \(m_1 \neq 0\) is called a *normalized representable number* (with respect to particular choices of \(b, t, k_1\) and \(k_2\)).

“Machine precision” (many definitions; called ‘eps’ in Matlab) is typically used to mean the difference between 1 and the next larger floating-point number.

Because the mantissa contains 52 bits, IEEE \(\epsilon_M\) (machine precision) is

\[ \epsilon_M = 2^{-52} \approx 2.2 \times 10^{-16} \quad \text{(double precision)} \]

The ranges for overflow and underflow are \(k_1 = 1022\) and \(k_2 = 1023\).

The nonzero representable number of smallest absolute value is

\[ 2^{-1022} \approx 2.2 \times 10^{-308} \]

and the representable number of largest absolute value is

\[ (2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308} \].