Chapter 1: Numbers

Section 8: Computer Numbers

This section is about how computers handle numbers. For many people, this is right up there with asking how variable-speed transmissions work in cars: they do their thing, and who cares how? This kind of complacency can lead to trouble. We’ll start at (almost) the beginning: the word ‘computer’ was first used in the 1600’s; it meant a person who does computations professionally. The astronomer Kepler started as a computer for the astronomer Tycho Brahe. By the late 1800’s middle-class women wanted work as calculators; it fit in with social standards of what work a married woman could do, especially if they were not allowed to leave home; see Figure 84.

World War One involved powerful artillery; computing the path of a shell launched at a single angle required about 750 multiplications and about 7 woman-hours of work. The computations had to be done for many angles, and by the end of the second world war an analogue machine reduced the time to 20 minutes. Herman Goldstine and John Mauchley received funding from the US Army to build a digital version, the ENIAC.

The women who performed calculations knew that repeated calculations could lead to errors. Goldstine and Mauchley understood these errors would arise on electronic devices, but without the human understanding the women provided, the machine would not be aware there was a problem. And this lack of intuition and feeling for the numbers can be responsible for all kinds of errors, especially for those who assume computers do their thing and who cares how.

An early error happened on Intel’s Pentium P5 chip, Figure 85. Thomas Nicely discovered that certain inputs of $x$ and $y$ would cause an error in the computation

$$x - \left( \frac{x}{y} \right) \cdot y = 0$$

As an example, for $x = 4195835$ and $y = 3145727$, the chip returned 256 instead of 0 – leading to Figure 86, the autographed chip. We don’t think about a computer making errors like this – after all, it’s just a bunch of wires, which would be either always wrong or always right. That’s why we use computers. In reality, modern computer
chips are very complex circuits, and they have to be programmed to do computations. As the programs are written by humans ... there can be mistakes. For the Pentium bug, see p105.

Even though errors like this are inevitable (in 2018 the problem was SPECTRE and MELTDOWN bugs, with again caused by errors in programming), these aren’t the main issue. Since 1994, computer and chip manufacturers have systems in place to recall and replace defective units (Intel started this in 1994).

But there’s another level of problem – defective software. It’s estimated that the cost of dealing with bugs ran to over one trillion US$ in 2016 alone. Engineers and scientists typically do not have the same kinds of losses, but there can be losses to reputation for scientists, or loss of life in engineering projects. For example, the scientists C.L.Reyes and G. Chang retracted their paper Structure of the ABC transporter MsbA in complex with ADP:vanadate and lipopolysaccharide, Science, May 13, 2005. The structure was analyzed with a computer program, and an error in a ± sign led to a structure where some portions of the molecule were backwards; see Figure 87. The structure in question was a transporter molecule: it allows harmful bacteria to defend themselves against drug treatment, by transporting the drugs out through the bacterium cell wall. Understanding the mechanism used by ABC transporters might lead to techniques to disable it and eliminate resistance to antibiotics; see p105.

What went wrong? The structure of this molecule was determined by

Iterative eightfold noncrystallographic symmetry averaging, solvent flattening/flipping, phase extension, and amplitude sharpening using in-house programs yielded electron density maps of excellent quality for tracing a polypeptide chain.

Structure of MsbA from E. coli: A Homolog of the Multidrug Resistance ATP Binding Cassette(ABC) Transporters Science 7 Sept 2001 vol 293.

This involves intensive computer manipulation of the data, and, as it happened, one of the programs they used had an error.

In lab science, errors in procedure are to be expected. A scientist will spend years or decades learning correct lab procedure; what is different here is that scientists typically do not have decades of experience in computer programming. Here, a program was imported from another lab, but, Chang remarks, "you just trust the code to do the right job". He reports that he now triple checks everything (see Zeeya Merali, Nature 14 October 2010 vol 467, p 775).
Figure 88 gives another example where defective software destroyed (literally) a scientific mission: the $245 million Mars Climate Orbiter. Launched in 1998 to study the climate and atmosphere of Mars, the mission plan was to use onboard thruster rockets to position the spacecraft to enter Mars orbit. Instead, the thrusters sent the craft deep into the Martian atmosphere, where it burned up before crashing.

Here, the problem is more complex: a mission such as this involves thousands of individuals, across multiple corporations and government institutions. On the other hand, the same is true for many modern engineering projects: the construction of highways, skyscrapers, sports arenas, or air-traffic control systems, and even smartphones. Engineering firms taking on these kinds of tasks need to have quality control systems in place; compare the discussion on p47.

For the Mars Orbiter, two studies of the failure were made: the first, NASA’s internal study; a second in Why the Mars Probe went off course, IEEE Spectrum 1 Dec 1999. For details of the reports, see p105.

In these two examples, training and management systems can help reduce errors and losses. But there are mathematical issues.

One of the core issues was inconsistent use of units for the thrust of the orbiter’s rockets (one subsystem used meters, another used feet). The actual numbers generated differed by only a small amount. But two important points emerged:

i) A small initial error, over a million mile trajectory, can result in a large final error. Compared to the million mile trajectory, the final path was off by only a few kilometers – a tiny percentage, but large enough to destroy the orbiter.

ii) Flight engineers made many course corrections; this meant that the small error was made many times. A small error repeated many times can result in a large final error.

We’ll see these two issues again.

There’s a third major issue in using computers: operator error, illustrated in Figure 90. This is an example of a CAC or a computer aided catastrophe. As shown in Figure 89, the oil platform was intended to rest on the sea floor, meaning that the long beams supporting the platform had to carry not only the weight of the platform, but also deal with water pressure and storms. As the supports were lowered to the sea floor, one cracked, causing immediate flooding, dragging the platform underwater, where buoyancy tanks imploded under pressure; see https://wikivisually.com/wiki/Sleipner_A.

The problem was traced to the design of the supports. Large com-
plex structures acted on by many forces are typically designed using computer tools. The standard mathematical technique is called *finite element analysis*; in this technique, a complex structure is analyzed as smaller parts exerting forces on each other, and, in this case, responding to external forces such as water pressure, wind, waves and gravity. It’s then possible to see how a force on one part of the structure flows through the other parts. NASA developed a program, NASTRAN, used in modeling forces on multi-stage rockets, which is famously successful. In the Slepnir CAC, the designer did not fully understand how to implement the elements of the structure, which resulted in a design that underestimated forces by almost 50%.

The Slepnir designer is not alone. Ivo Babuska, a professor of civil engineering at the University of Texas at Austin, tried a small experiment: he sent a problem to a group of engineers. The problem is the Girkman problem, illustrated by the structure in Figure 91: how are the massive concrete walls to be supported under their own weight and wind forces?

Babuska specified that the engineers had to use professional-level computer software to solve the problem; of the 15 licensed professional engineers who submitted a solution, half were wrong. "They gave us numbers which were completely wrong and they believed in them" said Babuska. He commented, "How is it possible that this happened is a good question. There could be many various reasons. Nevertheless, in this case the reason was only one. Some of the analysts did not have sufficient engineering intuition and mathematical and engineering knowledge and possibly used the software incorrectly."

Operator error is a difficult problem to overcome, as it depends not on project management, or program design, but on the integrity of users. Babuska phrases the issue as 'signing the blueprints.' Only humans can sign blueprints and take financial and legal responsibility for structures.

All of the above examples of computer fail are really examples of human failure. Is the myth really true, that computers don’t make errors? As we said at the beginning: repeated computations can lead to errors on paper or on a computer; these kind of errors are a consequence of how numbers are written down or stored. They can only be avoided by understanding the issue, and planning against it – much like Babuska’s “mathematical and engineering knowledge.” This is the take-away: serious users of computers must develop an intuition about computer behavior.
We’ll start with an example:

\[
\lim_{x \to 0^+} \frac{1 - \cos(x)}{x^2}
\]

With a calculator: we let \( x \) approach 0\(^+\), by taking \( x = .1, .01, .001 \) etc, or \( x = 10^{-1}, 10^{-2}, 10^{-3}, \ldots \). Next, recall \( \lim_{x \to c} f(x) = L \) means the more decimal places \( x \) and \( c \) agree, the more decimal places \( f(x) \) and \( L \) agree (compare p74). So, with a Texas Instruments TI-85 calculator:

<table>
<thead>
<tr>
<th>( x )</th>
<th>( (1 - \cos x) / x^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1</td>
<td>0.45970</td>
</tr>
<tr>
<td>.01</td>
<td>0.50000</td>
</tr>
<tr>
<td>.001</td>
<td>0.50000</td>
</tr>
<tr>
<td>( 10^{-5} )</td>
<td>0.50000</td>
</tr>
<tr>
<td>( 10^{-6} )</td>
<td>0.50004</td>
</tr>
<tr>
<td>( 10^{-7} )</td>
<td>0.49960</td>
</tr>
<tr>
<td>( 10^{-8} )</td>
<td>0</td>
</tr>
<tr>
<td>( 10^{-9} )</td>
<td>0</td>
</tr>
</tbody>
</table>

We believe the closer \( x = 10^{-n} \) is to \( 0^+ \), the better answer we’ll get. Therefore, the correct answer is the zero, not the .5.

It’s a kind of a suspicious answer, because of the sudden jump from 0.50000 to zero. Let’s use our mathematical knowledge and intuition to check it in another way, with L’Hospital’s rule:

\[
\lim_{x \to 0^+} \frac{1 - \cos(x)}{x^2} = 0 = \lim_{x \to 0^+} \frac{[1 - \cos(x)]'}{[x^2]'} = \lim_{x \to 0^+} \frac{\sin(x)}{2x} = 0 = \lim_{x \to 0^+} \frac{[\sin(x)]'}{[2x]'} = \lim_{x \to 0^+} \frac{\cos(x)}{2} = \frac{1}{2}
\]

The L’Hospital answer uses a well-know theorem that’s been around for centuries, with no reported errors. But then – that means something very strange must be happening inside the calculator. And to sort this out, we’re going to have to go inside the calculator.

Computers and calculators are electronic devices – they work by shifting electrical charges around. Just as batteries can store and release charge, devices called capacitors can store and release charges quickly enough for modern computers.

We’ll conceptualize a computer number as a row of capacitors, and the charge in each capacitor represents a number; it would look something like the Heng/Zong system from p13, Figure 92. This picture shows the problem: there are only five boxes to store digits, so numbers larger than 999999 or smaller than .000001 can’t be written. We can add more boxes, but in a computer, there’s a limit; this limit is referred to as the \textit{word size} of the computer.

Figure 92: \textsc{The Heng/Zong System}
Recapping the system used in Han Dynasty China for representing numbers, using positional notation.
There are other limitations. First, there’s an issue of making the best use of the small word size. For example, if I wrote \(.000003\) as \(3 \times 10^{-6}\) I could store it as \(3, -, 6\). All I need to know is the third place stores the exponent, the second the sign of the exponent, and the first the actual number.

While we can imagine our capacitors storing ten different levels of charge to represent the numbers \(0, 1, \ldots 9\), in fact subtle variations in charge are very hard to detect. Modern computers use only two levels, which are traditionally thought of as the numbers one and zero (though in fact the two charges are more like \(\pm .5\)). Then a number like \(1001\) could represent \(1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0\), or 9. That is, we have to use binary arithmetic.

We also have to have standards, to know how to write numbers. Usually, to subtract \(5.42\) from \(21.06\), we’d align the numbers:

\[
\begin{array}{c}
  - \quad 2 \quad 1.0 \quad 6 \\
  \hline
  5.4 \quad 2 \\
  \hline
  1 \quad 5.6 \quad 4
\end{array}
\]

This is called fixed point arithmetic, and was used in the earliest computers (see p\textsuperscript{106}). Of course, we could use many other kinds of representation: \(21.06\) would be written as \(.2106 \times 10^2\), or, in scientific notation, \(2.106 \times 10^1\). In the 1960’s, when computers began to be used in business (see Figure 93), standards for storing numbers varied. Each representation had its own problems and associated errors; multiplication by 1.0 could cause loss of the last four decimal places; programmers would use tricks such as replacing \(x\) by \((x+x)-x\) to fool the computer into getting the answer right. Programming each individual computer was a craft in its own right, but as long as manufacturers like IBM kept one standard, programmers could adjust, and companies paid for programmer-craftworkers. See Charles Severance’s interview with “The Old Man of Floating-Point’, William Kahane, at https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html.

To see how we can get into trouble, we’ll invent a silly machine, the Kathytron 5, Figure 94. The machine uses floating point arithmetic: for the number 21.06 the decimal point floats to the front, to give \(.2106 \times 10^2\). With the bit structure described, the Kathytron has machine numbers of the form \(\pm b_1b_2 \times 2^{\pm e_1}\), where \(b_1, b_2, e_1\) can be 0, 1. These numbers are:

\begin{tabular}{|c|c|c|c|}
\hline
\(+/\) & 0, 1 & 0, 1 & \(+/\) & 0, 1 \\
\hline
\end{tabular}
$b_1 b_2 \times 2^{-1} \times 2^0 \times 2^1$

<table>
<thead>
<tr>
<th></th>
<th>.00</th>
<th>.00</th>
<th>.00</th>
<th>.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>.00</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
<td>.00</td>
</tr>
<tr>
<td>.01</td>
<td>.01</td>
<td>.01</td>
<td>.01</td>
<td>.01</td>
</tr>
<tr>
<td>.10</td>
<td>.10</td>
<td>.10</td>
<td>.10</td>
<td>.10</td>
</tr>
<tr>
<td>.11</td>
<td>.11</td>
<td>.11</td>
<td>.11</td>
<td>.11</td>
</tr>
</tbody>
</table>

Converting binary to a fraction, $.11 = \frac{1}{2^1} + \frac{1}{2^2} = \frac{1}{2} + \frac{1}{4} = \frac{3}{4}$ and then to a decimal:

<table>
<thead>
<tr>
<th></th>
<th>.125</th>
<th>.25</th>
<th>.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>.125</td>
<td>.25</td>
<td>.5</td>
<td></td>
</tr>
<tr>
<td>.25</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>.375</td>
<td>.75</td>
<td>1.5</td>
<td></td>
</tr>
</tbody>
</table>

On a number line, the positive machine numbers look like this:

These numbers are not equally spaced, and, although the Kathytron 5 is a small machine, this un-evenness is typical for this standard; we’ll talk about that when we discuss errors in approximating actual numbers by machine numbers.

There’s also a smallest non-zero number the machine can represent, called machine epsilon. In many programming languages, you can access machine epsilon by running the command $\text{eps}$; on the Kathytron, you’d get $\text{eps} = .125$ This book is being written on a MacBook Air running the public domain language Octave; and the $\text{eps}$ command gives $\text{eps} = 2.2204 \times 10^{-16}$, which will give rather more accurate computations than the Kathytron 5.

The Kathytron rounds a number smaller than machine epsilon to zero; this is called underflow, and a given computer may or may not send a message that this has occurred. Any number larger than the largest machine number generates overflow. For many machines, the processing unit will generate an error message; depending on the machine, this can terminate the computation.
Now, we’re not finished. The Kathytron saves one bit by using normalized floating point notation: the initial number can’t be a zero. In doing a computation, the machine will take the $b_1b_2$ from memory, and send it to the processing unit as $.1b_1b_2$; our Kathytron 5 numbers are now $\pm .1b_1b_2 \times 2^{\pm e_1}$. This gives some smaller numbers; our table is now

<table>
<thead>
<tr>
<th>$b_1b_2$</th>
<th>$2^{-1}$</th>
<th>$2^0$</th>
<th>$2^1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>.100</td>
<td>.100</td>
<td>.100</td>
<td>.100</td>
</tr>
<tr>
<td>.101</td>
<td>.101</td>
<td>.101</td>
<td>.101</td>
</tr>
<tr>
<td>.110</td>
<td>.110</td>
<td>.110</td>
<td>.110</td>
</tr>
<tr>
<td>.111</td>
<td>.111</td>
<td>.111</td>
<td>.111</td>
</tr>
</tbody>
</table>

In decimals

<table>
<thead>
<tr>
<th>.25</th>
<th>.5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>.3125</td>
<td>.625</td>
<td>1.25</td>
</tr>
<tr>
<td>.375</td>
<td>.75</td>
<td>1.5</td>
</tr>
<tr>
<td>.4375</td>
<td>.875</td>
<td>1.75</td>
</tr>
</tbody>
</table>

The numbers are still not equally spaced, especially near zero and machine epsilon, which is now twice as large. This is called the gap at zero, and occurs in machines that use any similar standard. And it is the cause of the original problem: when I subtract the machine numbers .375 and .3125, I get .0625, which is smaller than machine epsilon. This machine rounds this down to zero.

This is what went wrong when I computed $\lim_{x \to 0^+} \frac{1 - \cos(x)}{x^2}$. When $x$ was very close to zero, the subtraction $1 - \cos(x)$ caused the leading terms in the decimal to cancel, and the decimal places that remained were less than machine epsilon. Without notice, the machine rounded down to zero, and my table had a sudden jump from .5 to 0, where $1 - \cos(x)$ becomes less than machine epsilon.

We mentioned ‘intuitions’ and ‘craftworker tricks’ earlier; here’s one, for $1 - \cos(x)$:

$$\frac{1 - \cos(x)}{x^2} = \frac{1 - \cos(x)}{x^2} \frac{1 + \cos(x)}{1 + \cos(x)} = \frac{\sin^2(x)}{x^2} \left( \frac{1}{1 + \cos(x)} \right)$$
The trick gives us what calculator couldn’t:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\frac{\sin^2 x \left(\frac{1}{1 + \cos(x)}\right)}{x^2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>.1</td>
<td>0.49958</td>
</tr>
<tr>
<td>.01</td>
<td>0.49999</td>
</tr>
<tr>
<td>.001</td>
<td>0.50000</td>
</tr>
<tr>
<td>$10^{-5}$</td>
<td>0.50000</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>0.50000</td>
</tr>
<tr>
<td>$10^{-7}$</td>
<td>0.50000</td>
</tr>
<tr>
<td>$10^{-8}$</td>
<td>0.50000</td>
</tr>
<tr>
<td>$10^{-9}$</td>
<td>0.50000</td>
</tr>
</tbody>
</table>

But: the user has to know underflow can happen, and plan against it. This is why we need programmers-craftworkers, or, in the old days, women calculators. The trick is to eliminate possible subtractions where many leading terms in a decimal will cancel. There’s a rule of thumb: "Don’t subtract nearly equal numbers." That’s one of our ‘mathematical intuitions’.

Cancellation is one issue; underflow is another. If we try using the Pythagorean Theorem to compute the hypotenuse of a right triangle (sides $x$, $y$ and hypotenuse $h$) we have $h = \sqrt{x^2 + y^2}$. If one or both of $x$, $y$ are small, their square will underflow to zero. For example, on the Kathytron with normalized floating point numbers, choosing $x = y = \frac{1}{4}$ gives $x^2 = y^2 = \frac{1}{16}$, which underflows to zero.

One way to avoid the problem is to make $x$ and $y$ appear larger. If we let $s = x + y$, then

$$h = s \sqrt{\left(\frac{x}{s}\right)^2 + \left(\frac{y}{s}\right)^2}$$

Now $\frac{x}{x+y} = \frac{y}{x+y} = \frac{1}{2}$, so

$$h = \left(\frac{1}{4} + \frac{1}{4}\right) \sqrt{\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2} = \frac{1}{2} \sqrt{\frac{1}{4} + \frac{1}{4}}$$

Each of the squares, and their sum, is a machine number, so the formula produces the correct $\frac{1}{2}\sqrt{2}$. This will have to be rounded to a machine number, in this case $\frac{1}{2}$.

There is another way to deal with underflow: allow numbers in the gap near zero. You can do this by allowing non-normalized numbers, $\pm 0.0 \times 2^{\pm e_1}$. This brings in $.00$ again, but also $\pm 0.1 \times 2^{-1}$ and also $\pm 0.1 \times 2^{0}, \pm 0.1 \times 2^{1}$, which only gives us the new numbers $\pm 0.01$, inserted into the gap near zero.
When computers became available for medical, scientific and engineering research, the lack of a single standard became an issue. Software written for one computer would not run properly on other computers. Estimates of the accuracy of a computation were only good on a specific computer, and could be unreliable on others.

But the real motivation for setting a standard came from the development of microcomputers. IBM might sell a few hundred machines, Cray a dozen, but a chip designer like Intel expected to sell millions. Moreover, the users would not be large businesses with expensive craftworker-programmers, but small businesses with perhaps no programmers, who still expected correct answers.

A group of scientists began meeting to resolve the problem. Intel and Motorola wanted not just a standard, but the best possible standard. Dealing with underflow, overflow, the gap near zero – all of these inserted extra steps into computations. The desired standard would be one that didn’t slow down the chips too much. In 1974, the IEEE 794 standard for representing numbers on a computer came out. In reality it’s a set of standards; see again p. 106. Initially, more like a set of suggestions, but, as all the large chip-makers followed it, it became a ‘standard’ by default. IEEE 794 includes the idea of using non-normalized numbers to fill the gap near zero, as well as sending notifications for overflow and underflow.

Unfortunately, this doesn’t finish the job: every computation must result in a machine number, so almost every computation has to be rounded to a machine number, which results in a ‘wrong’ answer. If we’re going to use computers, we have to understand how much wrong those answers are.

We’ll start with one example of how a roundoff-error can go badly wrong: the 1991 SCUD missile attack on a US Army base at Dhahran, Saudi Arabia. The base was protected from missile attacks by the Patriot anti-missile defense (see Figure 96). In the 1991 incident, the system failed to intercept a SCUD attack (Figure 97), resulting in major loss of life. Investigation showed it was due to a software error caused by rounding.

The Patriot system was designed to run for short lengths of time, to avoid detection – and because the internal clock rounded absolute time to machine numbers, causing an error of $9.5 \times 10^{-8}$. This particular system was protecting soldiers 24/7, so it had been left running for over 100 hours. The accumulated round-off errors caused the system to generate an estimate of the SCUD position that was off by 572 meters. In short, the Patriot missile missed. For a short discussion see
As if this weren’t enough, all kinds of odd algebra can happen with machine numbers. For example, the machine number .3125 squares to 0.09765625 < \text{eps}, so the square becomes zero. We add numbers in any order; this is the associative law: \((a + b) + c = a + (b + c)\).

It doesn’t work in a computer, even using just machine numbers: 
\[
1.75 + .25 - .5 \text{ can be } (1.75 + .25) - .5 = \text{overflow.}
\]
But we could add in a different order to get 
\[
1.75 + (.25 - .5) = 1.75 - .25 = 1.5
\]

It almost seems as we’re back in the days of hand-made arithmetic, with craftwomen calculators checking that every step works. In reality, it’s not that bad: we can predict errors.

We’ll start with the error caused by roundoff. Let’s look at an example, again from IEEE 794 on the Kathytron. Our number \(x\) will be .6. The Kathytron follows IEEE, which requires rounding to the closest machine number. That’s .625. There are two different ways to look at error:

a) Absolute Error = \(|\text{True value} - \text{Approximate value}|\), and
b) Relative Error = (Absolute Error)/(True value).

In this case, the relative error is
\[
\frac{.6 - .625}{.6} = \frac{1 - 1 + .025}{6} = \frac{1.25}{3} \leq \frac{1}{2} \text{eps}
\]

– since machine epsilon is \(\text{eps} = .25\). In general:

\[
\text{Relative Error} \leq \frac{1}{2} \text{eps}
\]

This is an easy result to understand: the number you round is between two machine numbers, so the error is at most half the length between the machine numbers. The gap between machine numbers increases when the numbers get large (see p100). But, while the absolute error can increase as the numbers increase, the relative error gets divided by the size of the number, which makes the gap \(\text{eps}\), and relative error \(\leq \frac{1}{2} \text{eps}\).

The next thing we have to understand is how errors change when we do basic arithmetic and round. We’ll use a special notation: if we take a number \(a\), the closest machine number is denoted by \(f\ell(a)\) (floating point of \(a\)). What we want to understand, then, is how 
\[
f\ell(a + b), \ f\ell(a \cdot b), \ f\ell\left(\frac{a}{b}\right),
\]
relate to \(f\ell(a)\) and \(f\ell(b)\).
We’ll follow the discussion from Ward Cheney and David Kincaid, *Numerical Mathematics and Computing*, Brooks/Cole 1994. The assumption is that the machine takes \( a, b \), computes the arithmetic operation correctly, then rounds. This is partly true; many machines use double word lengths to do the computation, and then round to machine numbers. The assumption fails for underflow or overflow, but, aside from this, is reasonable. So we still have Relative Error \( \leq \frac{1}{2} \varepsilon \), or, for example,

\[
\left| \frac{fl(a + b) - (a + b)}{a + b} \right| \leq \frac{1}{2} \varepsilon
\]

or,

\[
\frac{fl(a + b) - (a + b)}{a + b} = \delta
\]

where \( |\delta| \leq \frac{1}{2} \varepsilon \). Rewriting,

\[
fl(a + b) - (a + b) = (a + b) \times \delta
\]

and similarly for multiplication and division. Let’s check this with the special case of \( a = \frac{3}{4} = .75 \) a machine number, and \( b = \frac{3}{5} = .6 \) not a machine number; we chose these numbers to avoid underflow or overflow.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Result</td>
<td>fl(B)</td>
<td></td>
<td>B − C</td>
</tr>
<tr>
<td>( a + b )</td>
<td>1.35</td>
<td>1.25</td>
<td>.1</td>
<td>.16875</td>
</tr>
<tr>
<td>( a \times b )</td>
<td>.45</td>
<td>.5</td>
<td>.05</td>
<td>.05625</td>
</tr>
<tr>
<td>( \frac{b}{a} )</td>
<td>.8</td>
<td>.75</td>
<td>.05</td>
<td>.1</td>
</tr>
</tbody>
</table>

Column D should always be less than Column E, which works out, though rather closely for multiplication. So, we can in fact control the errors in arithmetic operations, which addresses early concerns about using inaccurate floating point numbers rather than accurate fixed point numbers; see p106.
Notes for Chapter 1 Section 8: Computer Numbers

p94 The Pentium bug arose because of the way computers do division. In a way, it’s very much like we do division. Try dividing 376 by 7. We have a technique – an algorithm. We’d first divide 7 into 37. We remember that $7 \times 5 = 35$ but $7 \times 6 = 42$. So the divisor is 5, with a remainder of 26. Now divide 7 into 26. Again we remember that $7 \times 3 = 21$ but $7 \times 4 = 28$. So the divisor is 3, with a remainder of 5.

When I was younger – a lot younger – I’d look up some of those multiplications, in a multiplication table. The Pentium used a division algorithm called the SRT algorithm, named after the inventors, Sweeney, Robertson, and Tocher. It also uses a lookup table, though everything is binary, so it seems rather strange; it’s similar to Figure 98. See Stuart F Oberman and Michael J Flynn, An analysis of division and implementations at http://i.stanford.edu/pub/cstr/reports/csl/tr/95/675/CSL-TR-95-675.pdf

p94 For the importance of the ABC transporters, see Christopher F. Higgins and Kenneth J. Linton, The xyz of ABC Transporters, Science 7 Sept 2001 vol 293 p1782. They remark,

A cell must selectively translocate molecules across its plasma membrane to maintain the composition of its cytoplasm distinct from that of the surrounding milieu. The most intriguing, and, arguably, the most important membrane proteins for this purpose are the ABC (ATP-binding cassette) transporters. These proteins, found in all species, use the energy of ATP hydrolysis to translocate specific substrates across cellular membranes.

Overexpression of certain ABC transporters is the most frequent cause of resistance to cytotoxic agents including antibiotics, antifungals, herbicides, and anticancer drugs.

p95 In the analyses of the failure of the Mars Orbiter Mission, NASA identified the problem as a software issue: a contractor wrote a software module to determine the thrust of the orbiter’s rockets. The module gave thrust in $ft/sec^2$; the software to which it reported expected force to be reported in Newtons. NASA blamed its own internal procedures for not finding the units mismatch.

The IEEE Spectrum analysis asserts that the problems in NASA went much deeper: flight controllers believed that something was wrong with the trajectory, but were over-ruled by higher management. The article claims that the engineers were told to “stop thinking like engineers and think like managers.” They claim that the expectation was that all systems worked perfectly, and if problems were suspected, it was up to the engineers to prove there was a problem.
For a brief history, see Michael L. Overton *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM Press 2001. Overton notes: "Von Neumann […] promoted the use of fixed point representation. He was well aware that the range limitations of fixed point would be too severe to be practical, but he believed that the necessary scaling by a power of 2 should be done by the programmer, not the machine; he argued that bits were too precious to be wasted on storing an exponent when they could be used to extend the precision of the significand." See Figure 99, von Neuman with the 1949 EDVAC computer, using both binary arithmetic and stored programs.