Floating Point Arithmetic, Errors, and Flops

2.1 The Floating Point Number System

Floating point numbers have the form

\[ \pm m_0.m_1m_2 \ldots m_{t-1} \times b^e \]

\( m = m_0m_1m_2 \ldots m_{t-1} \) is called the mantissa, \( b \) is the base, \( e \) is the exponent, and \( t \) is the precision. Each \( m_i \) is between 0 and \( b-1 \) (inclusive,) except \( m_0 \neq 0 \). The exponents range between two numbers \( e_{\text{min}} \leq e \leq e_{\text{max}} \). The system is inherently finite: there are only \( (b-1)^t \) many mantissas, \( e_{\text{max}} - e_{\text{min}} + 1 \) many exponents, and two signs, for a total of \( 2(b-1)b^{t-1}(e_{\text{max}} - e_{\text{min}} + 1) \) floating point numbers.

Let \( F \) represent the set of floating point numbers. The elements of \( F \) are unevenly distributed. Within each interval \([b^e, b^{e+1})\) there are \((b-1)b^t\) uniformly distributed numbers (one for each mantissa.) But the interval \([b^{e+1}, b^{e+2})\) immediately to the right of \([b^e, b^{e+1})\) is \( b \) times longer than \([b^e, b^{e+1})\). Hence the gaps between the numbers are getting larger as the exponents increase.

IEEE Standard Floating Point Arithmetic

The standard for floating point arithmetic is IEEE 754. The primary aims of the standard are consistent and sensible conventions for handling exceptions, leading to portability and predictability of code. The format standard is

<table>
<thead>
<tr>
<th>Precision</th>
<th>Base</th>
<th>Exponent Bits</th>
<th>Min Exponent</th>
<th>Max Exponent</th>
<th>Unique Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>2</td>
<td>23</td>
<td>-126</td>
<td>127</td>
<td>( 2^{-23} \approx 6 \times 10^{-8} )</td>
</tr>
<tr>
<td>Double</td>
<td>2</td>
<td>53</td>
<td>-1022</td>
<td>1023</td>
<td>( 2^{-53} \approx 1 \times 10^{-16} )</td>
</tr>
</tbody>
</table>

MATLAB performs all operations in double precision binary, \( b = 2 \). MATLAB displays in decimal, \( b = 10 \). MATLAB does support two display modes, format long and format short. Format short is the default mode when a MATLAB session is opened, displaying 5 decimal places. By typing format long the display mode will be switched to show 15 decimal places. This is only the display, the additional digits are available, just not displayed.

Let \( \text{fl}(x) \) be the floating point approximation to \( x \). There are several rules for computing \( \text{fl}(x) \), which we will discuss below. For now notice that we require the following properties

\[
\begin{align*}
  x &\leq y \Rightarrow \text{fl}(x) \leq \text{fl}(y) \\
  x \in F &\Rightarrow \text{fl}(x) = x \\
  f_1, f_2 \in F &\text{ are adjacent and } f_1 \leq x \leq f_2 \Rightarrow \text{fl}(x) = f_1 \text{ or } \text{fl}(x) = f_2
\end{align*}
\]

Some Base Conversion Issues

We will focus on binary and decimal bases, since \( b = 2 \) is typically the base prescribed by IEEE 754 and \( b = 10 \) is the base of the display and many data acquisition devices.

A binary integer is written in the form \((b_k \ldots b_0)_2\) where each \( b_i \) is either 0 or 1. This representation is the number

\[ b_k2^k + \ldots + b_12^1 + b_02^0 \]

and by expanding this sum the conversion back to decimal can be made. In MATLAB the function polyval allows us to easily expand this, as polyval takes the vector \( b = [b_k \ldots b_0] \) and evaluates the polynomial

\[ b_kx^k + \ldots + b_1x^1 + b_0x^0 \]
Thus, \( \text{polyval}(b,2) \) will find \( b_2 2^k + \ldots + b_1 2^1 + b_0 2^0 \).

A binary fraction is written in the form \( .b_1 b_2 \ldots b_k \), which represents the fraction

\[
\frac{b_1}{2^1} + \frac{b_2}{2^2} + \ldots + \frac{b_k}{2^k} = b_k x^k + b_{k-1} x^{k-1} + \ldots + b_1 x
\]

where \( x = 1/2 \) and conversion to decimal can be made by expansion of this expression. In MATLAB we can do this with \( \text{polyval([b_k, \ldots, b_1, 0], 1/2)} \). The reversal of \( b \) is obtained by \( b=b(k:-1:1) \).

The conversion from a decimal representation to binary is by iterated application of division by 2. Generally division of an integer \( n \) by \( d \) has the form

\[
n = dq + r \quad \text{where } 0 \leq r < d
\]

In MATLAB the functions \( r=\text{mod}(n,d) \) and \( q=\text{floor}(n/d) \) provide values for \( q \) and \( r \). The binary representation is obtained as follows:

\[
b=\text{mod}(n,2) \\
q=\text{floor}(n/2) \\
\text{while } q > 0 \quad \\
\quad b=[\text{mod}(q,2), b]; \\
\quad q=\text{floor}(q/2);
\]

The conversion from a decimal fraction representation to a binary fraction is by iterated multiplication by two. The primary observation is the following:

If \( n = .b_1 b_2 \ldots b_k \), then \( 2n = b_1 b_2 \ldots b_k \) and \( b_1 = \text{floor}(2n) \). To find \( b_2 \) repeat for \( n_1 = b_2 \ldots b_k \). Not every fraction has a finite binary (or decimal) expansion, but they will have an eventually repeating expansion.

We are familiar with the fact that the fraction \( 1/3 \) expands as a repeating decimal, \( .333 \ldots \). Similarly, using the technique described above \( 1/3 \) expands as a repeating binary expansion as \( .010101 \ldots \). We can convert back to fractional form in the usual way: \( s = .010101 \ldots \), then \( 4s = 01.010101 \ldots \) and \( 4s - s = 3s = 1.0101 \ldots - .0101 \ldots = 1 \), so that \( s = 1/3 \).

The floating point representation for \( 1/3 \) where \( b = 2 \) and \( t = 10 \) is

\[
\text{fl}(1/3) = 1.010101010 \times 2^{-2}
\]

In decimal this is .3330. The absolute error is approximately

\[
|1/3 - \text{fl}(1/3)| = .00032552
\]

and the relative error is approximately

\[
\frac{|1/3 - \text{fl}(1/3)|}{|1/3|} = .00097656
\]

ulp and eps

Suppose that \( f_1, f_2 \in \mathcal{F} \) are adjacent and \( f_1 \leq x \leq f_2 \), where \( f_1 = m2^e \) and \( f_2 = m'2^e \) and \( m \) and \( m' \) are the (adjacent) mantissas, then the absolute error in the floating point approximation is

\[
|x - \text{fl}(x)| = 2^e (m - m') \leq 2^e 2^{-t+1} = 2^{e-t+1} = 2^e/2^{t-1}.
\]

2
since the distance between adjacent mantissas is $2^{-t+1}$. Since this depends on the exponent $e$, the absolute error can be quite large, reflecting the large gaps between the floating point numbers for large $e$ or it can be quite small for small (that is, negative) $e$. The relative error is

$$\frac{|x - \text{fl}(x)|}{|x|} < \frac{2^{e-t+1}}{2^e} = 2^{-t+1}$$

since $2^e < x$. The number $2^{-t+1} = \epsilon$, is called the *machine precision* or *machine epsilon*. $\epsilon$ is the distance to the next floating point number after 1.0. The next floating point number after 1.0 is $1.\overline{0...1}$ and so

$$\epsilon = \frac{0...1}{t-1-\text{places}} = \frac{1}{2^{t-1}} = 2^{-t+1}.$$

Since, by definition, $1 + \epsilon$ is the next floating point number after 1.0, $1 + \epsilon > 1$. We will call $\text{ulp}$, for *unit in the last place*,

$$\text{ulp} = \epsilon/2 = 1/2^t = 2^{-t}.$$

Note that since $\text{ulp} < \epsilon, 1 + \text{ulp} = 1$.

Most modern commodity floating point processors conform to IEEE 754, and, in particular, MATLAB uses the double precision format. Thus, in MATLAB

$$\text{ulp} = 2^{-53} \approx 1.110223024625157e-016.$$

MATLAB has a variable for the machine precision, `eps`. The value for `eps` for on any machine is given in decimal by

$$2.220446049250313e-16$$

which is 2·ulp. Even though the last digit is not correct, this is a display matter as

```plaintext
abs(eps - 2*ulp)
```

or

```plaintext
eps == 2*ulp
```

shows.

**Note** that the use of $x == y$ when $x$ and $y$ are floating point numbers is not appropriate. It is actually very rare when two floating point numbers which would mathematically be equal turn out to be equal, for example,

```plaintext
sqrt(5)*sqrt(5)-5 == 0
```

returns 0 when it should return 1. But

```plaintext
abs(sqrt(5)*sqrt(5)-5)
```

returns

$$8.881784197001252e-16$$

which is a number very close to 0. For this reason an equality check in floating point arithmetic should be of the form

```plaintext
abs(x-y) < tol
```

where $tol$ is a (usually small) quantity chosen to reflect the accuracy of the data. Since we may not know the scaling of the data it is even better to do a relative check like

```plaintext
abs(x-y)/abs(y) < tol.
```

**IEEE Arithmetic**

We will use the usual notations $x + y, x - y, x \times y$, and $x/y$ for operations in true arithmetic and $x \oplus y, x \ominus y, x \otimes y$ and $x \oslash y$ for the analogous floating point operations of addition, subtraction, multiplication
and division (respectively.) The basic model of computation should satisfy these relative error bounds for \(x, y \in \mathbb{F}\)

\[
\begin{align*}
\frac{|(x \oplus y) - (x + y)|}{|x + y|} & \leq \text{ulp} \\
\frac{|(x \ominus y) - (x - y)|}{|x - y|} & \leq \text{ulp} \\
\frac{|(x \otimes y) - (x \times y)|}{|x \times y|} & \leq \text{ulp} \\
\frac{|(x \oslash y) - (x/y)|}{|x/y|} & \leq \text{ulp}
\end{align*}
\]

Suppose that \(f_1 \leq x \leq f_2\) where \(f_1\) and \(f_2\) are adjacent floating point numbers, then we can express differing round-off modes as

1. round towards \(+\infty\) : always round up, or \(\text{fl}(x) = f_2\).
2. round towards \(-\infty\) : always round down, or \(\text{fl}(x) = f_1\).
3. round towards 0: round negative numbers up and positive numbers down, or \(\text{fl}(x) = f_2\), when \(x < 0\), and \(\text{fl}(x) = f_1\), when \(x > 0\).
4. round towards the nearest integer: if \(|f_1 - x| < |f_2 - x|\), then \(\text{fl}(x) = f_1\) else \(\text{fl}(x) = f_2\).
5. round to nearest even: use (4) but in case \(|f_1 - x| = |f_2 - x|\), \(\text{fl}(x) = f_i\) where \(f_i\) is the choice whose least significant bit is 0.

IEEE 754 will allow the user to choose from four rounding schemes, but the default mode is (5), round to nearest even. The standard also includes two additional conventions \(\text{Inf}\), \(\text{infinity}\) and \(\text{NaN}\), (not a number.)

\(\text{Inf}\) results from either division by 0 or an overflow condition which occurs when a computation leads into an exponent \(e > e_{\text{max}}\).

\(\text{NaN}\) occurs when \(\infty/\infty\) or \(-\infty - \infty\) or \(0 \cdot \infty\) or if the computation leads to the square root of a negative number. This latter is not an issue in MATLAB, as MATLAB will automatically switch into complex mode. When a computation encounters \(\text{NaN}\), any operation on \(\text{NaN}\) will return \(\text{NaN}\).

It is possible to get 0.0 from an underflow condition, that is, when \(e < e_{\text{min}}\). If the underflow occurs with negative numbers, then we can get \(-0.0\).

### 2.2 Flops and Processor Elapsed Time

We are going to consider two ways of measuring the amount of work or running time of an algorithm. The most obvious is to measure processor time using a clock. MATLAB provides a way to access the clock in your machine with a variety of commands like commands \(\text{etime}, \text{tic}, \text{toc}, \text{clock}\), and \(\text{cputime}\). Suppose that you want to time a program or a command,

\[
a = \text{rand}(10); \quad b = \text{rand}(10,1) \\
\text{tic; a}\backslash b; \quad \text{toc}
\]

Try repeating the same line several times. You will most likely get some variation in the time. Try repeating the operation 20 times and averaging.

\[
n = 20; \\
\text{tic; for i=1:n, a}\backslash b; \text{end, toc/n}
\]

If you repeat this several times you should notice less variation in the answer. Try it for other values of \(n\). When you are working in MATLAB this is a very useful way to measure the speed of a program, unfortunately this will tend to measure time to access the processor and gain operating system priority, which creates the variation in the measurement of run times.
To get some measure of efficiency which is independent of the machine or the language that it is written in, you should count the operations performed by the algorithm. A *flop* or floating point operation is either an addition, subtraction, multiplication, or division operation of floating point numbers. In floating point arithmetic these operations are counted as one flop. In complex number arithmetic, addition and subtraction are counted as 2 flops; multiplication and division are counted as 6 flops. We are going to concentrate on real arithmetic.

Prior to MATLAB Release 12, MATLAB had a function, `flops`, for counting flops. Release 12 was a major change in code which did not include the flop counter. MathWorks promises to bring the flop counter back in some future release. We will use the old MATLAB conventions for counting flops. There are other conventions, for example, one convention counts the two operations of an addition and a multiplication together as a flop. Other operations such as \( \sin \), \( \log \), etc. were also counted as a single flop by MATLAB.

**Counting Flops**

The dot product of vectors \( x = (x_1, \ldots, x_n) \) and \( y = (y_1, \ldots, x_n) \) is

\[
x \cdot y = x_1y_1 + \cdots + x_ny_n
\]

has \( n \) multiplications and \( n - 1 \) additions. *We will follow the standard convention of counting this dot product as \( 2n \) flops.* The reason is that it can be coded in the following way:

```matlab
s=0
for i=1:n
    s=s+x(i)*y(i);
end
```

This code uses \( n \) additions and \( n \) multiplications for \( 2n \) flops. This is the old MATLAB convention, which simplifies flop counting.

Let’s count the flops for matrix multiplication. Suppose that \( A \) is an \( m \times n \) matrix and \( B \) is an \( n \times p \) matrix, then \( C = AB \) is a \( m \times p \) matrix and each entry is the dot product of a row of \( A \) with a column of \( B \). By the dot product computation given above we have \( 2n \) flops for each entry and since there are \( mp \) entries we have a total of \( 2nmp \) flops for matrix multiplication. If \( A \) and \( B \) are both \( n \times n \) matrices, then this formula becomes \( 2n^3 \) flops. Here is a general principle. Suppose we have a loop

```matlab
for i=1:k,
    P(i),
end
```

where each execution of the program \( P(i) \) requires \( F(i) \) flops. Then the loop requires

\[
\sum_{i=1}^{k} F(i) \text{ flops}
\]
We need to review some rules of summation to progress further.

\[
\sum_{k=1}^{n} 1 = n
\]
\[
\sum_{k=1}^{n} k = n(n + 1)/2
\]
\[
\sum_{k=1}^{n} k^2 = n(n + 1)(2n + 1)/6
\]
\[
\sum_{k=1}^{n} r^k = r \sum_{k=1}^{n} a_n
\]
\[
\sum_{k=1}^{n} a_n + b_n = \sum_{k=1}^{n} a_n + \sum_{k=1}^{n} b_n
\]

Now look at a program for multiplication:

```matlab
[m,n]=size(A); [n,p]=size(B); C=zeros(m,p);
for i=1:m
    for j=1:p
        for k=1:n
            C(i,j)=C(i,j)+A(i,k)*B(k,j);
        end
    end
end
```

We can now compute the flops in the following way:

\[
\sum_{i=1}^{m} \sum_{j=1}^{p} \sum_{k=1}^{n} 2 = \sum_{i=1}^{m} \sum_{j=1}^{p} 2n = \sum_{i=1}^{m} 2np = 2npm
\]

**Forward Substitution**

We are now going to estimate the flops for *forward substitution*. Forward substitution is the means for solving a linear system \(Lx = b\) where \(L\) is lower triangular and \(L(i, i) \neq 0\).

```matlab
function x=forsub(A,b)
    n=size(L,1);
    x=zeros(n,1);
    x(1)=b(1)/L(1,1);
    for i=2:n
        x(i)=b(i);
        for j=1:i-1
            x(i)=x(i)-L(i,j)*x(j);
        end
        x(i)=x(i)/L(i,i);
    end
end
```
The flop count is given by 1 flop for $x(1)=b(1)/L(1,1)$ and then the loops

$$1 + \sum_{i=2}^{n} (1 + \sum_{j=1}^{i-1} 2) = 1 + \sum_{i=2}^{n} (1 + 2(i-1))$$

$$= 1 + (n-1) + 2 \sum_{i=2}^{n} (i-1)$$

$$= n + \sum_{i=1}^{n-1} i$$

$$= n + 2(n-1)n/2$$

$$= n^2$$

**Gaussian Elimination**

The code below is `badgauss`, similar to that given at the end of Chapter 1. We assume that $A$ is $m \times n$.

```matlab
function B=badgauss(A)
    m=size(A,1);
    B=A;
    for i=1:m-1
        for j=i+1:m
            B(j,:)=B(j,:)-(B(j,i)/B(i,i))*B(i,:);
        end
    end
end
```

To count the flops we notice that the innermost assignment is an elementary row operation of type 3, $B(j,:) = B(j,:) - (B(j,i)/B(i,i)) \ast B(i,:)$ which has a flop count of $2n + 1$, given in this form. Thus the flop count for the whole code is

$$\sum_{i=1}^{m-1} \sum_{j=i+1}^{m} (1 + 2n) = (2n + 1) \sum_{i=1}^{m-1} (m - i)$$

$$= (2n + 1)m(m - 1)/2$$

If we assume that $m = n$, then this simplifies $n^3 - n^2/2 - n/2$.

We will give a short example of how to estimate flops for a recursive program. Consider the following program for computing $2^n$.

```matlab
function y=twoexp(n)
    if n == 0
        y=1;
    else
        y=2*twoexp(n-1)
    end
end
```

The first step is to find a recurrence relation which counts the flops. Let $F(n)$ represent the flops for computing `twoexp(n)`, then we see that $F(0) = 0$ and $F(k + 1) = F(k) + 1$, the $+1$ comes from the multiplication by 2 and the $F(k)$ comes from the recursive call to `twoexp(n-1)`. In this example we can evaluate $F(n)$ easily

$$F(n) = F(n-1) + 1 = F(n-2) + 2 = \ldots = F(n-n) + n = n.$$
This recurrence is particularly easy to evaluate. It is usually much more difficult to find an expression for a recurrence relation.

2.3 Errors

Broadly speaking there are four sources of error.

I. Data Error
Data error is any data that is inaccurate before we attempt to compute with the data. Data error can occur from the limitations of the measuring instrument or simply from data that is passed in as the result of a prior computation.

II. Representational Error
There is almost always error in the approximation of a real number by a floating point number. We saw above that 1/3 has a repeating infinite binary expansion, and so there must be error in the finite representation of 1/3 as a floating point number. This type of error is essentially part of the floating point number system and little can be done about it.

III. Error Approximating Functions
With most functions we must resort to an approximation to the function by some finite means, frequently some type of polynomial approximation. The approximation of the function will introduce error.

IV. Rounding Error
These are the rounding errors that emerge during the course of a computation. Analyzing these errors is one of our primary tasks.

We will focus on the last two sources of error, as the first two are beyond our control. The edge of rounding error is visible with the value ulp from section 2.1, where

\[ 1 + \text{ulp} = 1 \]

This leads immediately to breakdowns like

\[(x \oplus y) \oplus z \neq x \oplus (y \oplus z)\]

by taking \( x = \text{ulp}, y = 1 \) and \( z = -1 \). Similarly,

\[ x \oslash (x \oplus y) \neq (x \oslash y) \oplus (x \oslash z) \]

by taking \( x = 2, y = 1 \) and \( z = \text{ulp} \). Most other axioms for arithmetic remain true, though the associative law for multiplication also fails.

In view of the failure of associativity for addition we are left with the dilemma for how to evaluate a sum like

\[ x_1 + x_2 + \ldots + x_n \]

or for that matter more complex expressions like the dot product of two vectors, or the multiplication of two matrices. For example, a sum like

\[ 1 + \text{ulp} + (-1) + 1 + \text{ulp} + (-1) \]

should be eps, but will return 0. It is possible to keep track of the losses in a sum like this and greatly improve the accuracy. The addition would normally proceed from left to right, so the parenthesis would be \(((1 + \text{ulp}) + (-1)) + 1 + \text{ulp} + (-1)\).
The following code (due to Kahan) tracks round-off losses and reintroduces them. This is a fairly safe method of addition of a sequence of numbers given by a vector $x$.

```text
sm=0; dif=0;
for k=1:n
    y=x(k)-dif;
    temp=sm;
    sm=temp+y;
    dif=(sm-temp)-y;
end
```

The code is theoretically correct, in that, in true arithmetic, $sm$ is the sum and $dif$ is 0.

If $x_{true}$ is the correct solution to a problem and $x_{comp}$ is the computer solution, then the absolute error is $|x_{true} - x_{comp}|$ and the relative error is $\frac{|x_{true} - x_{comp}|}{|x_{true}|}$. Compute the absolute and relative error for the expression

$$x = (ulp + 1) - 1$$

$x_{true} = ulp$ while $x_{comp} = 0$. The absolute error is

$$|x_{true} - x_{comp}| = ulp$$

which is small. On the other hand the relative error, which leads to the more accurate notion of error in terms of the relative sizes of the quantities involved, is

$$\frac{|x_{true} - x_{comp}|}{|x_{true}|} = \frac{ulp}{ulp} = 1$$

The error is as large as the answer. This is bad.

**Condition Numbers**

Consider this simple system of equations:

$$x + y = 100$$

$$x + (1 + \epsilon)y = 100$$

Notice that the unique solution to this is $x = 100, y = 0$. Now modify the system, very slightly

$$x + y = 100$$

$$x + (1 + \epsilon)y = 100(1 + \epsilon)$$

and notice that the unique solution to this is $x = 0, y = 100$. The wild swing in the solution to this system has nothing to do with round-off error or even floating point numbers, it is in the structure of the problem. The problem is inherent to the data and there is really nothing we can do about it.

A problem where a small change in the data produces a large change in the solution is called *ill-conditioned*. The *condition number* is a way of measuring the change in the value of the solution. So, conditioning has to do with the relationship between the output and the input. The notion of a condition number changes with the type of problem. Later we will talk about the condition number for a linear system.

MATLAB is on its guard for ill-conditioned problems and will provide a message like

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 5.551115e-17
```
As a simple example of a condition number we will consider the condition number to compute the value of a function $F : R \to R$. Suppose we know that the input has relative error for $x_0 = \text{fl}(x)$ given by

$$\frac{|x_0 - x|}{|x|}$$

At best, we can compute $y_0 = F(x_0)$ and the relative error would be

$$\frac{|y_0 - y|}{|y|}$$

where $y = F(x)$. Look at

$$\frac{|y_0 - y|}{|y|} = \frac{|y_0 - y|}{|x_0 - x|} \frac{|x_0 - x|}{|x|} \frac{|x|}{|y|}$$

Since $\frac{|y_0 - y|}{|x_0 - x|}$ is an approximation to the derivative of $F(x)$, we will replace it with $F'(x)$ to get

$$\frac{|y_0 - y|}{|y|} \approx |F'(x)| \frac{|x_0 - x|}{|x|} \frac{|x|}{|F(x)|}$$

Let $\kappa(x) = \frac{|x|F'(x)|}{|F(x)|}$, then

$$\frac{|y_0 - y|}{|y|} \approx \kappa(x) \frac{|x_0 - x|}{|x|}$$

$\kappa(x)$ is the condition number of $F(x)$ at $x$.

We can use $\kappa(x)$ in the following way:

$$-\log_{10} \frac{|x_0 - x|}{|x|}$$

is the number of accurate digits in floating point representation $x_0$ of $x$. Since we know

$$\frac{|x_0 - x|}{|x|} \leq \epsilon$$

the number of accurate digits in $\text{fl}(x)$ is approximately $-\log_{10}(\epsilon) \approx 16$.

Similarly, $-\log_{10} \frac{|y_0 - y|}{|y|}$ is the number of accurate digits in the computation $y_0$ of $F(x_0)$. Thus we can estimate the number of accurate digits of $y_0$ with

$$-\log_{10} \frac{|y_0 - y|}{|y|} \approx -\log(\kappa(x)) - \log_{10} \frac{|x_0 - x|}{|x|} \approx 16 - \log_{10}(\kappa(x))$$

As an example consider the function $F(x) = \exp(x)$. For this $F(x)$, $\kappa(x) = |x|$. For $x = 100$, we can expect 14 accurate digits. For $x = 10^{10}$, we can expect 4 accurate digits, for $x = 1$, 16 digits.

Similarly, consider $F(x) = \log(x)$. Here $\kappa(x) = 1/\log(x)$. Since $\kappa(x)$ has an asymptote at $x = 1$, we can expect problems here. Let $x = 1 + 10^{-5}$, then $\kappa(x) \approx 100,000$ and $\log_{10}(\kappa(x)) \approx 5$, so we expect $16 - 5 = 11$ accurate digits. If we take $x = 1 + 10^{-10}$, $\log_{10}(\kappa(x)) \approx 10$, so we expect $16 - 10 = 6$ accurate digits. On the other hand, if we take $x = 10^{5}$, then $\kappa(x) \approx .00868$ and $\log_{10}(\kappa(x)) \approx -1.06$, so we expect $16 - (-1) = 17$!!!

Notice how this discussion makes no mention of the method used to compute. The nature of conditioning is that the function can be intrinsically difficult to compute. The function $\kappa(x)$ gives us a means of measuring that difficulty.

When the solution method produces an accumulation of round-off error. The method is called an \textit{unstable} In this case there is hope that another method might be stable and yield an accurate result for the problem.