COMPUTING ANOMALIES

These examples are meant to help motivate the study of machine arithmetic.

1. *Calculator example*: Use an HP-15C calculator, which contains 10 digits in its display. Let

$$x_1 = x_2 = x_3 = 98765$$

There are keys on the calculator for the mean

$$\xi = \frac{1}{n}\sum_{j=1}^{n} x_j$$

and the standard deviation s where

$$s^2 = \frac{1}{n-1}\sum_{j=1}^{n}\left(\xi - x_j\right)^2$$

In our case, what should these equal? In fact, the calculator gives

$$\xi = 98765 \qquad\qquad s \doteq 1.58$$

Why?

2. *A Fortran program example*: Consider two programs run on a now extinct computer.

Program A:

$A = 1.0 + 2.0 * *(-23)$
$B = A - 1.0$
$PRINT *, A, B$
END
Output: 1.0 $\qquad$ $1.19E-7 \quad (\doteq 2^{-23})$

Program B:

$A = 1.0 + 2.0 * *(-23)$
$SILLY = 0.0$
$B = A - 1.0$
$PRINT *, A, B$
END
Output: 1.0 $\qquad$ 0.0

Why the change, since presumably the variable $SILLY$ does not have any connection to B.

DECIMAL FLOATING-POINT NUMBERS

Floating point notation is akin to what is called *scientific notation* in high school algebra. For a nonzero number x, we can write it in the form

$$x = \sigma \cdot \xi \cdot 10^e$$

with e an integer, $1 \leq \xi < 10$, and $\sigma = +1$ or -1. Thus

$$\frac{50}{3} = (1.66666 \cdots)_{10} \cdot 10^1, \qquad \text{with } \sigma = +1$$

On a decimal computer or calculator, we store x by instead storing σ, ξ, and e. We must restrict the number of digits in ξ and the size of the exponent e. For example, on an HP-15C calculator, the number of digits kept in ξ is 10, and the exponent is restricted to $-99 \leq e \leq 99$.

BINARY FLOATING-POINT NUMBERS

We now do something similar with the binary representation of a number x. Write

$$x = \sigma \cdot \xi \cdot 2^e$$

with

$$1 \leq \xi < (10)_2 = 2$$

and e an integer. For example,

$$(.1)_{10} = (1.10011001100 \cdots)_2 \cdot 2^{-4}, \qquad \sigma = +1$$

The number x is stored in the computer by storing the σ, ξ, and e. On all computers, there are restrictions on the number of digits in ξ and the size of e.

FLOATING POINT NUMBERS

When a number x outside a computer or calculator is converted into a machine number, we denote it by $fl(x)$. On an HP-calculator,

$$fl(.3333\cdots) = (3.333333333)_{10} \cdot 10^{-1}$$

The decimal fraction of infinite length will not fit in the registers of the calculator, but the latter 10-digit number will fit. Some calculators actually carry more digits internally than they allow to be displayed.

On a binary computer, we use a similar notation. I will concentrate on a particular form of computer floating point number, that called the IEEE floating point standard.

In single precision, we write such a number as

$$fl(x) = \sigma \cdot (1.a_1 a_2 \cdots a_{23})_2 \cdot 2^e$$

$$fl(x) = \sigma \cdot (1.a_1a_2 \cdots a_{23})_2 \cdot 2^e$$

The *significand* $\xi = (1.a_1a_2 \cdots a_{23})_2$ immediately satisfies $1 \le \xi < 2$. What are the limits on e.

To understand the limits on e and the number of binary digits chosen for ξ, we must look roughly at how the number x will be stored in the computer. Basically, we store σ as a single *bit*, the significand ξ as 24 bits (only 23 need be stored), and the exponent fills out 32 bits (=4 bytes). Thus the exponent e occupies 8 bits, including both negative and positive integers. Roughly speaking, we have that e must satisfy

$$-(1111111)_2 \le e \le (1111111)_2$$

$$-127 \le e \le 127$$

In actuality, the limits are

$$-126 \le e \le 127$$

for reasons related to the storage of 0 and other numbers such as $\pm\infty$.

What is the connection of the 24 bits in the significand ξ to the number of decimal digits in the storage of a number x into floating point form. One way of answering this is to find the integer M for which
1. $0 < x \leq M$ and x an integer implies $fl(x) = x$; and
2. $fl(M + 1) \neq M + 1$
This integer M is at least as big as

$$\left(1.\underbrace{11\cdots1}_{23\ 1's}\right)_2 \cdot 2^{23} = 2^{23} + \cdots + 2^0$$

This sums to $2^{24} - 1$. In addition, $2^{24} = (1.0\cdots0)_2 \cdot 2^{24}$ also stores exactly. What about $2^{24} + 1$? It does not store exactly, as

$$\left(1.\underbrace{0\cdots0}_{23\ 0's}1\right)_2 \cdot 2^{24}$$

Storing this would require 25 bits, one more than allowed. Thus

$$M = 2^{24} = 16777216$$

This means that all 7 digit decimal integers store exactly, along with a few 8 digit integers.

THE MACHINE EPSILON

Let y be the smallest number representable in the machine arithmetic that is greater than 1 in the machine. The *machine epsilon* is $\eta = y - 1$. It is a widely used measure of the accuracy possible in representing numbers in the machine.

The number 1 has the simple floating point representation

$$1 = (1.00 \cdots 0)_2 \cdot 2^0$$

What is the smallest number that is greater than 1? It is

$$1 + 2^{-23} = (1.0 \cdots 01)_2 \cdot 2^0 > 1$$

and the machine epsilon in IEEE single precision floating point format is $\eta = 2^{-23} \doteq 1.19 \times 10^{-7}$.

THE UNIT ROUND

Consider the smallest number $\delta > 0$ that is representable in the machine and for which

$$1 + \delta > 1$$

in the arithmetic of the machine.

For any number $0 < \alpha < \delta$, the result of $1 + \alpha$ is exactly 1 in the machines arithmetic. Thus α ‘drops off the end’ of the floating point representation in the machine. The size of δ is another way of describing the accuracy attainable in the floating point representation of the machine. The machine epsilon.has been replacing it in recent years.

It is not too difficult to derive δ. The number 1 has the simple floating point representation

$$1 = (1.00 \cdots 0)_2 \cdot 2^0$$

What is the smallest number which can be added to this without disappearing? Certainly we can write

$$1 + 2^{-23} = (1.0 \cdots 01)_2 \cdot 2^0 > 1$$

Past this point, we need to know whether we are using *chopped* arithmetic or *rounded* arithmetic. We will shortly look at both of these. With chopped arithmetic, $\delta = 2^{-23}$; and with rounded arithmetic, $\delta = 2^{-24}$.

ROUNDING AND CHOPPING

Let us first consider these concepts with decimal arithmetic. We write a computer floating point number z as

$$z = \sigma \cdot \zeta \cdot 10^e \equiv \sigma \cdot (a_1.a_2 \cdots a_n)_{10} \cdot 10^e$$

with $a_1 \neq 0$, so that there are n decimal digits in the significand $(a_1.a_2 \cdots a_n)_{10}$.

Given a general number

$$x = \sigma \cdot (a_1.a_2 \cdots a_n \cdots)_{10} \cdot 10^e, \quad a_1 \neq 0$$

we must shorten it to fit within the computer. This is done by either *chopping* or *rounding*. The floating point chopped version of x is given by

$$fl(x) = \sigma \cdot (a_1.a_2 \cdots a_n)_{10} \cdot 10^e$$

where we assume that e fits within the bounds required by the computer or calculator.

For the rounded version, we must decide whether to round up or round down. A simplified formula is

$$fl(x) = \begin{cases} \sigma \cdot (a_1.a_2 \cdots a_n)_{10} \cdot 10^e & a_{n+1} < 5 \\ \sigma \cdot [(a_1.a_2 \cdots a_n)_{10} + (0.0 \cdots 1)_{10}] \cdot 10^e & a_{n+1} \geq 5 \end{cases}$$

The term $(0.0 \cdots 1)_{10}$ denotes 10^{-n+1}, giving the ordinary sense of rounding with which you are familiar. In the single case

$$(0.0 \cdots 0a_{n+1}a_{n+2} \cdots)_{10} = (0.0 \cdots 0500 \cdots)_{10}$$

a more elaborate procedure is used so as to assure an unbiased rounding.

CHOPPING/ROUNDING IN BINARY

Let

$$x = \sigma \cdot (1.a_2 \cdots a_n \cdots)_2 \cdot 2^e$$

with all a_i equal to 0 or 1. Then for a *chopped* floating point representation, we have

$$fl(x) = \sigma \cdot (1.a_2 \cdots a_n)_2 \cdot 2^e$$

For a *rounded* floating point representation, we have

$$fl(x) = \begin{cases} \sigma \cdot (1.a_2 \cdots a_n)_2 \cdot 10^e & a_{n+1} = 0 \\ \sigma \cdot [(1.a_2 \cdots a_n)_2 + (0.0 \cdots 1)_2] \cdot 10^e & a_{n+1} = 1 \end{cases}$$

ERRORS

The error $x - fl(x) = 0$ when x needs no change to be put into the computer or calculator. Of more interest is the case when the error is nonzero. Consider first the case $x > 0$ (meaning $\sigma = +1$). The case with $x < 0$ is the same, except for the sign being opposite.

With $x \neq fl(x)$, and using chopping, we have

$$fl(x) < x$$

and the error $x - fl(x)$ is always positive. This later has major consequences in extended numerical computations. With $x \neq fl(x)$ and rounding, the error $x - fl(x)$ is negative for half the values of x, and it is positive for the other half of possible values of x.

We often write the *relative error* as

$$\frac{x - fl(x)}{x} = -\varepsilon$$

This can be expanded to obtain

$$fl(x) = (1 + \varepsilon)x$$

Thus $fl(x)$ can be considered as a *perturbed* value of x. This is used in many analyses of the effects of chopping and rounding errors in numerical computations.

For bounds on ε, we have

$$\begin{array}{llll} -2^{-n} & \leq \varepsilon & \leq 2^{-n}, & \text{rounding} \\ -2^{-n+1} & \leq \varepsilon & \leq 0, & \text{chopping} \end{array}$$

IEEE ARITHMETIC

We are only giving the minimal characteristics of IEEE arithmetic. There are many options available on the types of arithmetic and the chopping/rounding. The default arithmetic uses rounding.

Single precision arithmetic:

$$n = 24, \qquad -126 \le e \le 127$$

This results in

$$M = 2^{24} = 16777216$$

$$\eta = 2^{-23} = 1.19 \times 10^{-7}$$

Double precision arithmetic:

$$n = 53, \qquad -1022 \le e \le 1023$$

What are M and η?

There is also an extended representation, having $n = 69$ digits in its significand.

NUMERICAL PRECISION IN MATLAB

MATLAB can be used to generate the binary floating point representation of a number. Execute the command

```
format hex
```

This will cause all subsequent numerical output to the screen to be given in hexadecimal format (base 16). For example, listing the number 7 results in an output of

```
401c000000000000
```

The 16 hexadecimal digits are $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$. To obtain the binary representation, convert each hexadecimal digit to a four digit binary number. For the above number, we obtain the binary expansion

$$0100\,0000\,0001\,1100\,0000\ \ldots\ 0000$$

for the number 7 in IEEE double precision floating-point format.

NUMERICAL PRECISION IN FORTRAN

In Fortran, variables take on default *types* if no explicit typing is given. If a variable begins with I, J, K, L, M, or N, then the default type is *INTEGER*. Otherwise, the default type is *REAL*, or "*SINGLE PRECISION*". We have other variable types, including *DOUBLE PRECISION*.

Redefining the default typing: Use the statement

```
IMPLICIT DOUBLE PRECISION(A-H,O-Z)
```

to change the original default, of REAL, to DOUBLE PRECISION. You can always override the default typing with explicit typing. For example

```
DOUBLE PRECISION INTEGRAL, MEAN
INTEGER P, Q, TML_OUT
```

FORTRAN CONSTANTS

If you want to have a constant be DOUBLE PRECISION, you should make a habit of ending it with $D0$. For example, consider

```
DOUBLE PRECISION PI
⋮
PI=3.14159265358979
```

This will be compiled in way you did not intend. The number will be rounded to single precision length and then stored in a constant table. At run time, it will be retrieved, zeros will be appended to extend it to double precision length, and then it will be stored in PI. Instead, write

```
PI=3.14159265358979D0
```