Representations of Numbers

Mathematical models almost always involve numbers of some sort. When we implement a model, we need to represent these numbers on the computer.

We will look at representations for *integers* and for *real numbers*.

We can judge a representation in several ways:

- What *range* of numbers can be represented?
- How *accurate* is the representation?
- How *efficient* is the representation in terms of time and space?

Maple can represent numbers in many ways. Most other languages (eg, C or Turing) are restricted to using the more efficient ways.

Representing Integers

Both people and computers usually represent positive integers in positional notation, using some base, b (often $b = 2$). The sequence of integer digits $d_n d_{n-1} \cdots d_1 d_0$, with $0 \le d_i < b$, represents

$$d_n b^n \;+\; d_{n-1} b^{n-1} \;+\; \cdots \;+\; d_1 b \;+\; d_0$$

The scheme can be extended to negative integers in several ways — we won't go into the details.

Other schemes are certainly possible (eg, Roman numerals), but are not used very often.

Fixed vs. Indefinite Size Integers

In languages such as C or Turing, there is a limit on how big an integer can be represented — ie, on how many digits (bits) are allowed. This allows for some efficiencies.

Trying to compute a bigger integer gives an error, or just the wrong answer.

In Maple, integers can be arbitrarily large, limited only by the available memory.

```
> 2^1000;

 1071508607186267320948425049060001810561\
    4048117055336074437503883703510511249\
    3612249319837881569585812759467291755\
    3146825187145285692314043598457757469\
    8574803934567774824230985421074605062\
    3711418779541821530464749835819412673\
    9876755916554394607706291457119647768\
    6542167660429831652624386837205668069\
    376
```

How About Fractions?

What if you want to represent a number that isn't an integer — eg, 1/3?

In languages such as C or Turing, such numbers may not be exactly representable, but Maple can do arithmetic on fractions, just like you did in primary school:

```
> 5/15;

              1/3

> 1/3+1/4;

              7/12

> 643432432 * (1/4324234 + 3/9567435834
>                 + 434324/123432434234);

              513440646811720847717711223424
              ------------------------------
                21277607705464438287603227 1
```

Other Symbolic Representations

Maple can represent numbers symbolically in many other ways as well:

```
> sqrt(4*5);
```

$$2\sqrt{5}$$

```
> 1/sqrt(2);
```

$$\frac{1}{2}\sqrt{2}$$

```
> sqrt(2)*sqrt(50);
```

$$10$$

```
> sin(Pi/8);
```

$$\frac{1}{2}\sqrt{2-\sqrt{2}}$$

All these representations are *exact*. The price of exactness is that these representations can become complex and cumbersome.

Approximate Representations

Most numerical calculations are done with *approximate* representations of numbers.

These representations can remain simple even through long, complex calculations — but they may also become less and less exact.

Two schemes for approximate representations are commonly used, distinguished by the type of error they permit:

- *Fixed-point* representations have errors of a certain *absolute* magnitude (eg, ± 0.001).
- *Floating-point* representations have errors of a certain *relative* magnitude (eg, $\pm 1\%$).

With either representation, numbers can be stored in memory slots of fixed size, and special hardware can be used to do arithmetic quickly.

Fixed-Point Representations

A *fixed-point* representation allows only numbers that can be written with some fixed number of digits (decimal or binary) in the fractional part.

For instance, we might allow three decimal digits in the fraction. We can then exactly represent numbers such as

$$0.138,\ \ -12.901,\ \ 8.900,\ \ 12387.003$$

We can't exactly represent numbers such as 3.1878 and −8.87309; we would have to round them to 3.188 and −8.873.

As well, there is usually an upper limit on the allowed magnitude of a fixed-point number.

Fixed-point representations can also be seen as *scaled-integer* representations. We use the integer i to represent the number $i \times S$, for some scaling factor S (above, S is 0.001).

Floating-Point Representations

A *floating-point* representation can exactly represent numbers with a certain number of digits (decimal or binary) in total, without regard to where these digits are in relation to the decimal point.

With three digits of accuracy, we can exactly represent numbers such as

$$1.45,\ 0.0000781,\ 90400,\ -77.0$$

We can't exactly represent numbers such as 12.81, 0.09217, and 79210; we would have to round them to 12.8, 0.0922, and 79200.

A number in a floating-point representation is like one in "scientific notation". The above numbers can be written as

0.145×10^1, 0.781×10^{-4}, 0.904×10^5, -0.770×10^2

Here, the fraction part (*mantissa*) is in $(0, 1)$; the *exponent* gives the scale of the number.

Overflow and Underflow

Numbers represented in floating-point have limited *accuracy*, given by the size of the mantissa. Usually, they also have a limited *range*, because the exponent can be only so big.

Overflow occurs when a number with too big an exponent arises. Depending on the machine and language used, overflow might be an error — causing the program to crash — or the result might be a special number, “Infinity”.

Underflow occurs when a number with too small an exponent (ie, a large negative exponent) arises. This is less serious than overflow. Because such a number is very close to zero, it may be OK to replace it with zero.

IEEE Standard Floating-Point

Old computers used many different floating-point representations, but nowadays, most follow a standard developed by the IEEE. This standard specifies two binary representations.

In *single precision* floating-point, the binary exponent ranges from -126 to $+127$, giving a range for numbers of about $10^{\pm 38}$. The binary mantissa has 24 bits (one invisible), giving accuracy to about 6 decimal digits.

In *double precision* floating-point, the binary exponent ranges from -1022 to $+1023$, for a range of about $10^{\pm 308}$. The mantissa has 53 bits (one invisible), for about 15 decimal digits of accuracy.

Maple will use the computer's built-in floating-point if you ask (using `evalhf`), but usually it uses its own scheme.

Maple Floating-Point

Maple uses decimal floating-point, with arithmetic implemented in software. This is much slower than using the built-in hardware floating-point, but has some advantages.

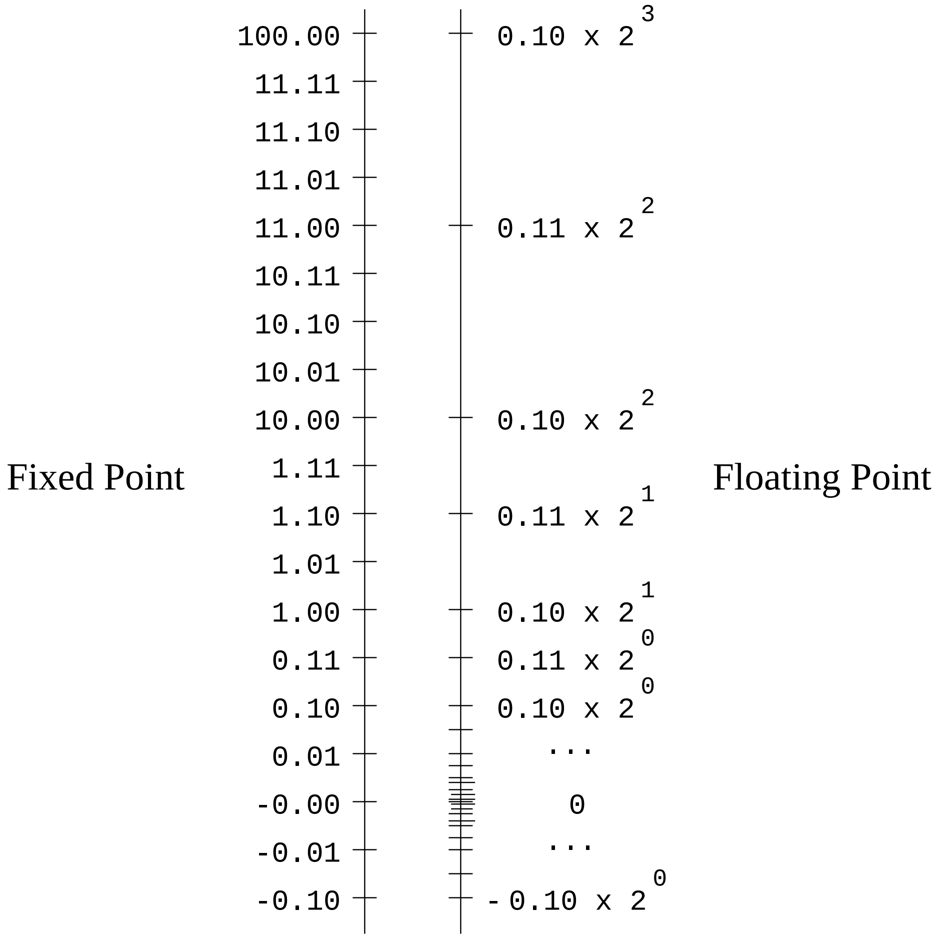
Maple lets the exponent grow to almost any size, so overflow and underflow don't occur.

You can specify the accuracy of Maple's floating-point by setting the `Digits` variable. For example:

```
> Digits:=5;

                              Digits := 5

> evalf(Pi);

                                3.1416

> Digits:=30;

                             Digits := 30

> evalf(Pi);

                    3.14159265358979323846264338328
```

Representable Numbers

Here are pictures of the numbers representable in fixed-point, with a two-bit fraction, and floating point, with two bits of accuracy:



Which representation is better depends on the expected magnitude of the numbers, and the allowable sorts of errors.

Which Representation of Numbers Would be Most Appropriate for...

- The number of customer complaints a company receives in some time period.
- The heights of various children.
- World population at various times in history.
- The concentration of some pollutant that is observed or predicted to be present in the water of Lake Ontario.
- The direction (degrees westward from north) at which migratory birds were observed to fly away from a given location.

Arithmetic with Inexact Numbers

What happens when we try to do arithmetic with numbers kept in inexact fixed-point or floating-point representations?

Often, the true result will not be representable. At best, we will get the *rounded* result — the number *closest* to the correct answer that can be represented.

Such *round-off error* can accumulate in a long computation, and sometimes make the final answer be nonsense.

Round-off error can also invalidate the mathematical properties that hold for exact computations — eg, associativity of addition.

Round-off Error with Fixed-Point Arithmetic

Addition and *subtraction* are exact with fixed-point arithmetic (as long as there is no overflow).

With a two-digit fraction, for example: $3.82 + 1.09 = 4.91$, with no round-off error.

Multiplication and *division* may not be exact.

For example: $1.01 \times 2.22 = 2.2422$, which is rounded to 2.24.

Not all the properties you might expect hold. For example:

$$\begin{aligned} 0.01 \times (0.01 \times 2000.00) &= 0.20 \\ \text{but}\quad (0.01 \times 0.01) \times 2000.00 &= 0.00 \end{aligned}$$

$$\begin{aligned} 0.01 \times (0.4 + 0.3) &= 0.01 \\ \text{but}\quad 0.01 \times 0.4 + 0.01 \times 0.3 &= 0.00 \end{aligned}$$

Round-off Error with Floating-Point Arithmetic

Addition and *Subtraction* may not be exact with floating-point arithmetic. For example, with three-digit accuracy:

$$\begin{aligned} 1.36 + 52.1 &= 53.47 \quad \text{rounds to} \quad 53.5 \\ 3300 - 259 &= 3041 \quad \text{rounds to} \quad 3040 \\ 380 + 1.2 &= 381.2 \quad \text{rounds to} \quad 381 \end{aligned}$$

Multiplication and *division* may not be exact either.

For example, with two-digit accuracy:
$1.1 \times 2200 = 2320$ rounds to 2300.

Mathematical Properties of Floating-Point Arithmetic

Floating-point point arithmetic has some properties of real arithmetic:

$$\begin{aligned} x + y &= y + x \\ x \times y &= y \times x \end{aligned}$$

if $x \le y$ and $c \ge 0$, then $x + c \le y + c$

But it doesn't have some other properties, as seen below for two-digit accuracy:

$(10 + 0.4) + 0.4 = 10$ but $10 + (0.4 + 0.4) = 11$

$$(99 + 10) - 10 = 100$$

$(49 \times 0.5) \times 0.5 = 13$ but $49 \times (0.5 \times 0.5) = 12$

$$(49/2) \times 2 = 50$$

$0.1 < 0.2$ and $10 > 0$, but $0.1 + 10 \not< 0.2 + 10$

Two Ways Round-Off Error Can Cause Big Problems

Cancellation can occur when two quantities are subtracted. The result then has many fewer digits of accuracy that the operands.

For example, with nine digits of accurracy:

$$12345.6789 - 12345.6721 \;=\; 0.0068$$

If these operands were previously rounded, the answer will be accurate to only two digits.

Saturation occurs when adding a non-zero quantity causes no change.

For example, with nine digits of accuracy:

$$1234567.89 + 0.004 = 1234567.89$$

This is not too bad if it happens once, but what if we add 0.004 a million times? The true answer is 123867.89, but the result we get is 1234567.89, wrong in the fourth digit.

An Example of Disastrous Cancellation

We want to calculate the profit of a company over the last year, from monthly revenues and expenses (all amounts in millions of dollars).

We decide to use floating-point arithmetic with three decimal digits of accuracy...

```
> Digits := 3:

> revenue := [ 81.9, 28.1, 13.9, 30.7, 22.1, 79.8,
>              43.2, 31.0, 34.9, 88.1, 52.3, 67.3 ]:

> expenses:= [ 82.1, 27.8, 13.9, 30.2, 22.2, 79.5,
>              43.0, 30.7, 35.2, 87.8, 52.1, 67.2 ]:

> total_revenue := sum(revenue[i],i=1..12);

                       total_revenue := 573.

> total_expenses:= sum(expenses[i],i=1..12);

                       total_expenses := 572.

> total_revenue - total_expenses;

                               1.
```

The answer: A total profit of 1 million dollars. But this is very wrong, as seen below:

```
> sum(revenue[i]-expenses[i],i=1..12);

                               1.6
```

How Can We Avoid Problems?

- Figure out how big round-off error could get, and make sure this is small enough.

  Example: The error resulting from floating-point addition with d digits of N numbers in the range $(-1, 1)$ must be less than $\pm N \times 10^{-(d-k)}$, where $k = 1 + \log N$.

- Use a scheme called *interval arithmetic*, which gets the computer to check how big the error could possibly be.

- A pragmatic approach:

  Be aware of the possible pitfalls.

  Be alert to signs that the answer may be wrong.

  Use plenty of precision, unless you're really concerned with speed.