Representable numbers

floating-point format numbers
are rational numbers with terminating expansion in base

Irrational numbers, such as π or √2, or non-terminating rational numbers
==> must be approximated.

The number of digits (or bits) of precision also limits the set of rational numbers that can be represented exactly.

For example, the number 123456789 clearly cannot be exactly represented if only eight decimal digits of precision are available.

Representable numbers cont'd

Whether or not a rational number has a terminating expansion depends on the base.

In base-10:

the number 1/2 has a terminating expansion (0.5) while the number 1/3 does not (0.333...).

In base-2:

only rationals with denominators that are powers of 2 (such as 1/2 or 3/16) are terminating.

Any rational with a denominator that has a prime factor other than 2 will have an infinite binary expansion.

This means that numbers which appear to be short and exact when written in decimal format may need to be approximated when converted to binary floating-point.

Representable numbers cont'd

For example, the decimal number 0.1 is not representable in binary floating-point of any finite precision;

the exact binary representation would have a "1100" sequence continuing endlessly:

$e = -??; s = 1100110011001100110011001100110011...,$

When rounded to 24 bits this becomes

$e = -27; s = 110011001100110011001101$

which is actually 0.100000001490116119384765625 in decimal.

Representable numbers cont'd

π, represented in binary as an infinite series of bits is:
11.0010010000111111011010101000100010000101101000110000100011010011...

approximated by 24 bits: 11.0010010000111111011011

binary single-precision floating-point,
s=110010010000111111011011 with e = −22.

Decimal: 3.1415927410125732421875,

Accurate approximation of the true value of π is:
3.14159265358979323846264338327950...

Floating-point arithmetic addition and subtraction

A simple method to add floating-point numbers is to represent them with the same exponent:

123456.7 = 1.234567 * 10^5
101.7654 = 1.017654 * 10^2 = 0.001017654 * 10^5

Hence:
123456.7 + 101.7654 = (1.234567 * 10^5) + (1.017654 * 10^2)
= (1.234567 * 10^5) + (0.001017654 * 10^5)
= (1.234567 + 0.001017654) * 10^5
= 1.235584654 * 10^5

Floating-point arithmetic addition and subtraction cont'd

```
  e=5;  s=1.234567     (123456.7)
+ e=2;  s=1.017654     (101.7654)

  e=5;  s=1.234567
+ e=5;  s=0.001017654  (after shifting)
--------------------
  e=5;  s=1.235584654  (true sum: 123558.4654)
```

It will be rounded to seven digits and then normalized if necessary. The final result is

```
  e=5;  s=1.235585    (final sum: 123558.5)
```

The low 3 digits of the second operand (654) are essentially lost. This is round-off error.

Floating-point arithmetic addition and subtraction cont'd

In extreme cases, the sum of two non-zero numbers may be equal to one of them:

```
  e=5;  s=1.234567
+ e=-3; s=9.876543

  e=5;  s=1.234567
+ e=5;  s=0.00000009876543 (after shifting)
----------------------
  e=5;  s=1.23456709876543 (true sum)
  e=5;  s=1.234567         (after rounding/normalization)
```

Our Example

```
x = 0.0;
h = 0.1;
for i = 1:10
x = x + h;
end;
y = 1.0 - x;
x, y,
```

Our Example cont'd

```
>> clear
>> x = 0.0;
h = 0.1;
for i = 1:10
x = x + h;
end;
y = 1.0 - x;
>> x, y,
x = 1.0000
y = 1.1102e-16
```

Our Example cont'd

Roundoff error in each step of

```
for i = 1:10  x = x + h;
```

Since h = 0.1 is approximated itself the addition in each step has some errors

==> The final x is not exactly 1.000000000000000

(y - x) can not be exact

it is also rounded.

==>

```
y = 1.0 - x;
>> x, y,
x = 1.0000
y = 1.1102e-16
```