Absolute and relative error

Let $\hat{z} =$ exact answer to some problem,
$z^* =$ computed answer using some algorithm.

Absolute error: $|z^* - \hat{z}|$

Relative error: $\dfrac{|z^* - \hat{z}|}{|\hat{z}|}$

If $|\hat{z}| \approx 1$ these are roughly the same.

But in general relative error is a better measure of
how many correct digits in the answer:

$$\text{Relative error } \approx 10^{-k} \quad \Longrightarrow \quad \approx k \text{ correct digits.}$$

Absolute and relative error

Example:

Compute length of diagonal of 1 meter $\times$ 1 meter square.

True value: $\hat{z} = \sqrt{2} = 1.4142135623730951\ldots$ meters
We compute $z^* = 1.413$ meters

Absolute error: $|z^* - \hat{z}| \approx 0.0012135 \approx 10^{-3}$ meters

Relative error: $\dfrac{|z^* - \hat{z}|}{|\hat{z}|} \approx \dfrac{0.0012135}{1.414} \approx 0.000858 \approx 10^{-3}$

Note: Relative error is dimensionless.

The absolute and relative errors are both $\approx 10^{-3}$.

Roughly 3 correct digitis in solution.

Absolute and relative error

Exactly same problem but now measure in kilometers.

Compute length of diagonal of 0.001 km $\times$ 0.001 km square.

True value: $\hat{z} = \sqrt{2} \times 0.001 = 0.0014142135623730951\ldots$ km
We compute $z^* = 0.001413$ km

Absolute error: $|z^* - \hat{z}| \approx 0.0000012135 \approx 10^{-6}$ km

Relative error: $\dfrac{|z^* - \hat{z}|}{|\hat{z}|} \approx \dfrac{0.0012135}{1.414} \approx 0.000858 \approx 10^{-3}$

The absolute error is much smaller than before
but there are still only 3 correct digits!

Absolute and relative error

Exactly same problem but now measure in nanometers.

Compute length of diagonal of 10^9 nm $\times 10^9$ nm square.

True value: $\hat{z} = \sqrt{2} \times 10^9 = 1414213562.3730951\ldots$ nm
We compute $z^* = 1413000000$ nm

Absolute error: $|z^* - \hat{z}| \approx 1213562.373 \approx 10^6$ nm

Relative error: $\dfrac{|z^* - \hat{z}|}{|\hat{z}|} \approx \dfrac{0.0012135}{1.414} \approx 0.000858 \approx 10^{-3}$

The absolute error is much larger than before
but there are still 3 correct digits!

Computer memory

Memory is subdivided into bytes, consisting of 8 bits each.

One byte can hold $2^8 = 256$ distinct numbers:

```
00000000    = 0
00000001    = 1
00000010    = 2
...
11111111    = 255
```

Might represent integers, characters, colors, etc.

Usually programs involve integers and real numbers that require more than 1 byte to store.

Often 4 bytes (32 bits) or 8 bytes (64 bits) used for each.

Integers

To store integers, need one bit for the sign ($+$ or $-$)
In one byte this would leave 7 bits for binary digits.

Two-complements representation used:

```
10000000    = -128
10000001    = -127
10000010    = -126
...
11111110    = -2
11111111    = -1
00000000    = 0
00000001    = 1
00000010    = 2
...
01111111    = 127
```

Advantage: Binary addition works directly.

Integers

Integers are typically stored in 4 bytes (32 bits). Values between roughly -2^{31} and 2^{31} can be stored.

In Python, larger integers can be stored and will automatically be stored using more bytes.

Note: special software for arithmetic, may be slower!

```
>>> 2**30
1073741824

>>> 2**100
1267650600228229401496703205376L
```

Note L on end!

32-bit vs. 64-bit architecture

Each byte in memory has an address, which is an integer.
On 32-bit machines, registers can only store

$$2^{32} = 4294967296 \approx 4 \text{ billion}$$

distinct addresses $\Longrightarrow$ at most 4GB of memory can be addressed.

Newer machines often have more, leading to the need for 64-bit architectures (8 bytes for addresses).

Note: Integers might still be stored in 4 bytes, for example.

Fixed point notation

Use, e.g. 64 bits for a real number but always assume N bits in integer part and M bits in fractional part.

Analog in decimal arithmetic, e.g.:

5 digits for integer part and
6 digits in fractional part

Could represent, e.g.:

```
00003.141592
00000.000314
31415.926535
```

Disadvantages:

- Precision depends on size of number
- Often many wasted bits (leading 0's)
- Limited range; often scientific problems involve very large or small numbers.

Floating point real numbers

Base 10 scientific notation:

$$\texttt{0.2345e-18} = 0.2345 \times 10^{-18} = 0.000000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Floating point real numbers

Base 10 scientific notation:

$$\texttt{0.2345e-18} = 0.2345 \times 10^{-18} = 0.000000000000000002345$$

Mantissa: 0.2345, Exponent: -18

Binary floating point numbers:

Example: Mantissa: 0.101101, Exponent: -11011 means:

$$\begin{aligned} 0.101101 &= 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) + 1(2^{-4}) + 0(2^{-5}) + 1(2^{-6}) \\ &= 0.703125 \ \text{(base 10)} \\ -11011 &= -1(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0) = -27 \ \text{(base 10)} \end{aligned}$$

So the number is

$$0.703125 \times 2^{-27} \approx 5.2386894822120667 \times 10^{-9}$$

Floating point real numbers

Fortran:

real (kind=4): 4 bytes
    This used to be standard single precision real

real (kind=8): 8 bytes
    This used to be called double precision real

Python float datatype is 8 bytes.

8 bytes = 64 bits,

53 bits for mantissa and 11 bits for exponent (64 bits $=$ 8 bytes).

We can store 52 binary bits of precision.

$2^{-52} \approx 2.2 \times 10^{-16} \implies$ roughly 15 digits of precision.

Floating point real numbers

Since $2^{-52} \approx 2.2 \times 10^{-16}$ this corresponds to roughly 15 digits of precision.

For example:

```
>>> from numpy import pi
>>> pi
3.1415926535897931

>>> 1000 * pi
3141.5926535897929
```

Note: storage and arithmetic is done in base 2
Converted to base 10 only when printed!

Overflow

8 bytes floats: 64 bits for each real number with
    53 bits for mantissa and
    11 bits for exponent.

Exponents range between -1022 and 1023, so magnitude of real number must be less than $N_{max} \approx 2^{1023} \approx 1.8 \times 10^{308}$.

If an operation gives a number outside this range we get an overflow exception.

Or perhaps a special value representing “infinity”.

Underflow

Exponents range between -1022 and 1023.
Smallest nonzero real number is about
$N_{min} = 2^{-1022} \approx 2.2 \times 10^{-308}$ if we insist it be normalized (i.e no leading zeros).

Can represent even smaller numbers by using gradual underflow, and subnormal numbers e.g.,

$$0.000005 \times 10^{-308} = 5.0 \times 10^{-314}$$

With 16 digits, can go down to about 10^{-324} in this manner.

Not-a-Number (NaN)

Some arithmetic operations give undefined results.

The result of such an operation is often replaced by a special value representing NaN.

Examples:

$0/0 =$ NaN

$0*$Infinity$=$ NaN

Machine epsilon (for 8 byte reals)

```
>>> y = 1. + 3.e-16
>>> y
1.0000000000000002

>>> y - 1.
2.2204460492503131e-16
```

Machine epsilon is the distance between 1.0 and the next largest number that can be represented: $2^{-52} \approx 2.2204 \times 10^{-16}$

```
>>> y = 1 + 1e-16
>>> y
1.0

>>> y == 1
True
```

Cancellation

We generally don't need 16 digits in our solutions

But often need that many digits to get reliable results.

```
>>> from numpy import pi
>>> pi
3.1415926535897931

>>> y = pi * 1.e-10
>>> y
3.1415926535897934e-10

>>> z = 1. + y
>>> z
1.0000000003141594   # lost several digits!

>>> z - 1.
3.1415936518897070e-10  # only 6 or 7 digits right!
```

Rounding errors can cause big errors!

Example: Solve $Ax = b$ using Matlab, for

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 4 - 10^{-12} \end{bmatrix}, \quad b = \begin{bmatrix} 5 \\ 10 - 2 \times 10^{-12} \end{bmatrix}. \quad \text{Solution:} \quad \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

```
>> format long e
>> A
A =
     1.000000000000000e+000    2.000000000000000e+000
     2.000000000000000e+000    3.999999999999000e+000

>> b
b =
     5.000000000000000e+000
     9.999999999998000e+000

>> x = A\b

x =
     9.982238010657194e-001    rel. error 0.00178
     2.000888099467140e+000    rel. error 0.00044
```

Rounding errors can cause big errors!

Example: Solve $Ax = b$ using Matlab, for

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 4 - 10^{-12} \end{bmatrix}, \quad b = \begin{bmatrix} 5 \\ 10 - 2 \times 10^{-12} \end{bmatrix}. \quad \text{Solution:} \quad \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

```
>> format long e
>> A
A =
     1.000000000000000e+000    2.000000000000000e+000
     2.000000000000000e+000    3.999999999999000e+000

>> b
b =
     5.000000000000000e+000
     9.999999999998000e+000

>> x = A\b

x =
     9.982238010657194e-001    rel. error 0.00178
     2.000888099467140e+000    rel. error 0.00044
```

Note: This matrix is nearly singular (ill-conditioned).
Second column is almost a scalar multiple of the first.