Computers have finite memory & processing speed
- can't express \( x \in \mathbb{R} - \mathbb{Q} \) (irrational #s)

Floating Point #:

\[ x = \pm \text{sign} \times \text{mantissa} \times 10^\text{exponent} \]

\[ x \in \mathbb{R} = \overline{x} + \varepsilon(x) \]

(FP#) absolute FP error

"round off error"

Representation in the computer:

<table>
<thead>
<tr>
<th>64-bit float double</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 sign</td>
</tr>
<tr>
<td>11 exponent</td>
</tr>
<tr>
<td>52 mantissa (also called significant)</td>
</tr>
</tbody>
</table>

Part of the #: sign exponent mantissa

# of bits: 11 + 8 + 23

This is of course binary (base 2) and of course the computer stores all #s in binary.

So, for example, the number close to 10) 43.625, which corresponds to binary 101011.101 can be represented also as 1.01011101 · 25 in binary scientific notation.

Check: \[ 1.01011101 \overline{2} = 2^5 + 2^3 + 2^1 + 2^0 + 2^-1 + 2^-3 \]

\[ 32 + 8 + 2 + 1 + 0.5 + 0.125 = 43.625 \]
so exponent: $2^5$ values, but must also handle
negative exponents: $0, 128, 127, 1, ... 2^5$
so for $43.625 \equiv 1.01011101 \cdot 2^5$ the exponent would be $5+127$
so $132$, which in binary is $10000100$

\[ \begin{align*}
2^7 & \uparrow \\
2^2 & \uparrow \\
-128 + 4 &= 132
\end{align*} \]


t mantissa: 23 bit

assumption: binary scientific notation always has a 1
in front of the decimal so drop it and gain accuracy!

so mantissa of $43.625$ is just $0101101000000000000000$

0 to 24 significant digits in binary

\[ \frac{15}{8} \]

Conversion from binary FP to decimal

so our number $43.625$ in all its glory is

\[ \begin{align*}
&\underline{0} \quad 10000100 \quad 01011101 \quad 0000000000000000 \\
&\downarrow \text{sign} \quad \text{exponent} \\
&5+127=132 \quad \text{mantissa}
\end{align*} \]

\[ 2^5 \times 1.01011101 = 2^5 \times (2^0 + 2^{-2} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-8}) = 43.625 \]
Conversion from decimal to binary FP:

Example: 329.370625

(1) stuff before the dot: \( \overline{296} \overline{43} \) \( 2 \overline{10} \)

\[ \begin{array}{c}
229 \text{ corresponds to } 101001001 \\
256 + 64 + 8 + 1 = 329
\end{array} \]

(2) stuff after the dot: 0.370625

Algorithm:

\[
\begin{array}{c}
0.370625 \times 2 = 0.74125 \\
0.74125 \times 2 = 1.4825 \\
1.4825 \div 2 = 0.74125 \\
0.74125 \div 2 = 0.370625
\end{array}
\]

A simple example:

59 = first binary digit's largest power of 2 that fits into 59

\[
\begin{array}{c}
2^5 = 32 \\
1, \text{ remainder: } 59 - 32 = 27
\end{array}
\]

2^4 = 16 \quad 1, \text{ remainder: } 27 - 16 = 11

2^3 = 8 \quad 1, \text{ remainder: } 11 - 8 = 3

2^2 = 4 \quad 0

2^1 = 2 \quad 1, \text{ remainder: } 3 - 2 = 1

2^0 = 1 \quad 1 - 1 = 0

So, \( 101001001.011001 = 1.01001001011001 \cdot 2^8 \)

Sign: 0

Exponent: \( 8 + 127 = 135 \quad 10000111 \)

Mantissa: 01001001.01100100000000000

\[ \uparrow \quad \text{14 \ zeros} \]

Largest exponent that can be represented: \( 2^{12} = 3.4 \times 10^{38} \)

in single precision

Largest # of significant digits: \( \approx \) (16777216 is largest value of

so really 7-8 mantissa in case 10)
11.2. FP #5s and round of error

Basic ops: $x + y$, $x - y$, $x \cdot y$, $x/y$

\[
\text{not } x + y \neq \bar{x} + \bar{y}
\]

\[
\bar{x} - \bar{y} = \bar{x} - \bar{y} + e(x) - e(y)
\]

\[
\text{can be problematic if } x \text{ and } y \text{ are nearly equal}
\]

\[
e.g.,\ m = 3 \quad x = 42.102 \quad y = 42.043
\]

\[
x - y = 0.059
\]

\[
\bar{x} - \bar{y} = 42.1 - 42.0 = 0.1
\]

\[
\bar{x} - \bar{y} = \bar{x} - \bar{y} + e(x) + e(y) + \frac{e(x) \cdot e(y)}{\bar{y}}
\]

\[
\text{at most } 2m \text{ dig. long; result rounded to } m \text{ digits.}
\]

\[
\frac{x}{y} = \frac{x + e(x)}{y + e(y)} = \frac{x + e(x)}{y} = \frac{x}{y} + \frac{e(x)}{y} - \frac{x \cdot e(y)}{y^2}
\]

\[
\text{error will grow drastically as } y \to 0
\]

\[
\text{avoid dividing by small numbers}
\]

\[
f(x + e(x), \bar{y} + e(y)) = f(\bar{x}, \bar{y}) + e(x) \frac{\partial f}{\partial x} (\bar{x}, \bar{y}) + e(y) \frac{\partial f}{\partial y} (\bar{x}, \bar{y}) + \ldots
\]
III.1.3 Stability of a Calculation

Def: A numerical calculation is called **unstable** if small errors made at one stage of the process are magnified in subsequent steps or seriously degrade the accuracy of the overall solution.

**Example:** WS 2

III.2 Finite Differences (FD)

III.2.1 Basic FD

will often need to compute derivatives of functions:

\[ f'(x) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} \]

Taylor expansion:

\[ f(x_0 + h) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!} h^n \]

\[ = f(x_0) + h f'(x_0) + \frac{h^2}{2} f''(x_0) + \ldots \]

Truncating at \( h^2 \) term:

\[ f(x_0 + h) = f(x_0) + h f'(x_0) + \sigma(h^2) \]

and solving for \( f' \):

\[ f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \sigma(h) \]

\( \uparrow \) error is first order in \( h \)!
The kind of error we are making here is a **truncation error**, because we truncate a series expansion!

→ so two kinds of errors: (round off, **truncation**) error

→ we have derived a forward derivative in the above.

Also: backward can be used (see notes)

### III. 2.3 Convergence

\[
\lim_{h \to 0} y(x; h) = y(x) \quad \rightarrow \text{numerical method is convergent}
\]

\[
\text{numerical solution} \rightarrow \text{true solution}
\]

→ for small \( h \), \( y(x; h) = y(x) \) → num. solution approaches true solution

Convergence order: if true solution is known,

**convergence factor:** \( Q = \frac{|y(x; h_2) - y(x; h_1)|}{|y(x; h_1) - y(x; h_2)|} = \left( \frac{h_2}{h_1} \right)^{n \text{ order}} \)

\( h_2 < h_1 \),

→ if \( n = 2 \), factor of 2 in res. would reduce error by factor of 4

→ self-convergence: often true solution not known, in this case, for three discretization steps

\( h_3 < h_2 < h_1 \)
\[ Q_s = \frac{y(x_1h_2) - y(x_1h_2)}{y(x_1h_2) - y(x_1h_1)} \]

which, at convergence

\[ n \text{ must equal } Q = \frac{h_2^n - h_1^n}{h_2^n - h_1^n} \]

### 3. Interpolation

Formal interpolation problem: Know value of function \( f \) at discrete points \( \{x_i\} \), but would like to know it at general points \( x \).

[Show graph]

#### 3.2 Lagrange Interpolation

- Can construct polynomials of degree \( n \) using \( n+1 \) points at which \( f \) is known.

- These polynomials have interpolating property in that \( p(x_i) = f(x_i) \) for \( \{x_i\} \), \( i = 1, \ldots, n+1 \) points at which \( f \) is known.

- General Lagrange formula:

\[
p(x) = \sum_{j=0}^{n} f(x_j) L_{nj}(x) + O(h^{n+1})
\]

\[
L_{nj} = \frac{1}{n!} \frac{r-x}{x_j-x_j}
\]
- This is a global interpolation method, constructing a high order interpolating polynomial.

- But high order does not always mean more accurate:
  - Runge's phenomenon
  [Show Runge Plot]

Global vs. Piecewise Interpolation

Lagrange's global method gets very oscillatory for large.

Alternatively, piecewise interpolation → use only local subset of k points \( x_i \) if enclosing point \( x \) where want to evaluate \( f(x) \).

→ use first, second, third, forth order interpolation
→ can give much better results in many cases!

Recall linear interpolation:
\[ p(x) = \sum_{j=0}^{k-1} f(x_j) L_j(x) + \sigma (Ch^2) \]

\[ = f(x_0) L_0(x) + f(x_1) L_1(x) + \ldots \]

\[ = \frac{f(x_0)}{x-x_1} \frac{x-x_1}{x_0-x_1} + \frac{f(x_1)}{x_1-x_0} \frac{x-x_0}{x_1-x_0} \]

Which can be brought into
\[ p(x) = a \cdot x + b \text{ form.} \]