Example 1 (1):

\[
\begin{align*}
\text{b} &= 2 \\
\text{p} &= 3 \\
\text{L} &= -1 \\
\text{U} &= 1
\end{align*}
\]

start enumerating possibilities:

\[
\begin{align*}
\pm & \quad m & \quad E \\
+ & \quad 0.00 & -1 & \rightarrow 0 \\
+ & \quad 0.00 & 0 & \rightarrow 0 \\
+ & \quad 0.00 & +1 & \rightarrow 0 \\
+ & \quad 0.01 & -1 & \rightarrow 0.001 \\
+ & \quad 0.01 & 0 & \rightarrow 0.01 \\
+ & \quad 0.01 & +1 & \rightarrow 0.1 \\
+ & \quad 0.10 & -1 & \rightarrow 0.01 \\
+ & \quad 0.10 & 0 & \rightarrow 0.1 \\
+ & \quad 0.10 & +1 & \rightarrow 1.0 \\
\end{align*}
\]

In general, number of possibilities:

\[2^* b^p * (U - L + 1)\]

but

- lots of duplicates
- non-unique representation

Normalization

- require the leading digit to be non-zero
- so mantissa, \( m \)
  \[1 \leq m < b\]
- nice because:
  - representation is now *unique*
  - don't waste digits on any leading 0's
  - for binary base, leading digit must be 1
  - so don't need to store it, just assume number is 1.d1d2..dp
  - gain an extra bit of precision!
Properties
- Finite and discrete system
- Finite: how many (normalized) numbers can be represented? Count them:
  \[ 2 \times (b - 1) \times b^{-p+1} \times (U - L + 1) + 1 \]
  \[ (Ex. 1) \quad 2 \times 1 \times 2^{3+1} + 1 = 25 \]
- What's the smallest (positive) normalized number? or "underflow level (UFL)"
  \[ 1.0 \ldots 0 \times b^{-L} = b^{-L} \quad (Ex. 1) \quad UFL = 0.5 \]
- What's the biggest normalized number? or "overflow level (OFL)"
  \[
  (b-1) \times (b-1) \ldots (b-1) \times b^{U+1} \\
  = ((b - b^{-p+1})) \times b^{U+1} \\
  = (1 - b^{-p}) \times b^{U+1}
  \]
  \[ (Ex. 4) \quad OFL = (1 - 2^{-3}) \times 2^{2} \\
  = (1 - \frac{1}{8}) \times 4 = 4 - \frac{1}{2} = 3.5 \]

Example 1 (2):

- Number of normalized numbers:
  \[
  2 \times (b - 1) \times b^{p-1} \times (U - L + 1) + 1 \\
  = 2 \times (2 - 1) \times 2^{3-1} \times (1 - 1 + 1) + 1 \\
  = 2 \times 1 \times 4 \times 3 + 1 \\
  = 25 \quad \checkmark
  \]
- UFL:
  \[
  b^{-L} = 2^{-1} \quad \checkmark
  \]
- OFL:
  \[
  (1 - b^{-(p)}) \times b^{U+1} = (1 - 2^{-(3)}) \times 2^{2} \]
  \[ = 3.5 \]
PICTURE of representable numbers
- note: evenly spaced only for a given exponent

Rounding
- floating point system is discrete!
  - not all real numbers representable
  - those that are called "machine numbers"
  - others must be "rounded"

\[ x \leftarrow \text{fl}(x) \]

- leads to "rounding error" or "roundoff error"

- How to round?
  1. Chop - truncate digits - "round to zero"
  2. Round to nearest
    - in case of tie go to even

Example: Rounding

<table>
<thead>
<tr>
<th>Number</th>
<th>Chop</th>
<th>Round to nearest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.649</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>1.650</td>
<td>1.6</td>
<td>1.6 (tie - round to even)</td>
</tr>
<tr>
<td>1.651</td>
<td>1.6</td>
<td>1.7</td>
</tr>
<tr>
<td>1.699</td>
<td>1.6</td>
<td>1.7</td>
</tr>
<tr>
<td>1.749</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>1.750</td>
<td>1.7</td>
<td>1.8 (tie - round to even)</td>
</tr>
<tr>
<td>1.751</td>
<td>1.7</td>
<td>1.8</td>
</tr>
<tr>
<td>1.799</td>
<td>1.7</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Machine Precision
Subnormals
- Normalized numbers: gap between 0 and $b^\infty$
- Fill in by allowing denormalized or subnormal numbers
- Can make use of capacity for non-normalized numbers by allowing leading 0's
- Though precision won't be full precision, since have leading 0's

Example 1(3):

- Allows 6 new numbers around 0
- New smallest number is $(0.01)_2 \times 2^{-1} = (0.125)_10$
- Called "gradual underflow" because we gradually lose precision

- Implementation: reserved value of exponent field
- Leading bit not stored

Exceptional values
- $\infty$
  - Dividing finite number by 0
  - Exceeding OFL
- NaN
  - Undefined operation $0/0$, $\text{Inf}/\text{Inf}$, $0*\text{Inf}$
  - Implemented through reserved values of exponent field

Floating Point Math
- Adding or subtracting
  - Match exponents first
  - Must shift smaller number
Rounding
- floating point system is discrete!
  - not all real numbers representable
  - those that are called "machine numbers"
  - others must be *rounded*

\[ x \leftarrow \text{fl}(x) \]

- leads to "rounding error" or "roundoff error"

- How to round?
  1. Chop - truncate digits - "round to zero"
  2. Round to nearest
     - in case of tie go to even

**Example: Rounding**

<table>
<thead>
<tr>
<th>Number</th>
<th>Chop</th>
<th>Round to nearest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.649</td>
<td>1.6</td>
<td>1.6</td>
</tr>
<tr>
<td>1.650</td>
<td>1.6</td>
<td>1.6 (tie - round to even)</td>
</tr>
<tr>
<td>1.651</td>
<td>1.6</td>
<td>1.7</td>
</tr>
<tr>
<td>1.699</td>
<td>1.6</td>
<td>1.7</td>
</tr>
<tr>
<td>1.749</td>
<td>1.7</td>
<td>1.7</td>
</tr>
<tr>
<td>1.750</td>
<td>1.7</td>
<td>1.8 (tie - round to even)</td>
</tr>
<tr>
<td>1.751</td>
<td>1.7</td>
<td>1.8</td>
</tr>
<tr>
<td>1.799</td>
<td>1.7</td>
<td>1.8</td>
</tr>
</tbody>
</table>

**EXAMPLE**  !!!!! warning: don't compare fp numbers with == !!!!!

```octave
>> 4/3-1 == 1/3
ans = 0

>> single((4/3-1)) == single(1/3)
ans = 1
```
\[
\begin{align*}
&= (4/3-1)/3 \\
\text{ans} = -5.5511e-17 \\
\text{right way to compare:} \\
&= \text{abs}((4/3 - 1) - 1/3) \leq 1e-16 \\
\text{ans} = 1 \\
\end{align*}
\]

---

**Machine Precision**

\[ \epsilon_{\text{mach}} \]

- characterizes accuracy
- "machine epsilon", "machine precision", "unit roundoff"
- depends on rounding rule

\[
\begin{array}{cccc}
\vdots & \vdots & \vdots & \vdots \\
0 & 1 & 2 & 3 \\
\end{array}
\begin{array}{c}
\text{x}
\text{x}
\text{x}
\text{x}
\end{array}
\begin{array}{c}
\text{...}
\text{...}
\text{...}
\text{...}
\end{array}
\begin{array}{c}
\text{p}
\text{p}
\end{array}
\]

- chop: (chop everything at and after \( b^{p} \) position)
  \[ b^{-(p-1)} = b^{(1-p)} \] lose up to this amount
- round: (lose up to half of chop)
  \[ \frac{1}{2} b^{(1-p)} \]
- tells us the max possible relative error in representation

\[
\left| \frac{f(x) - x}{x} \right| \leq \epsilon_{\text{mach}} \quad (\text{for normalized } x) \\
\]

- check:
  \[ \leq \epsilon_{\text{mach}} \cdot b^{e} \quad \left| \frac{x}{|x|} \right| \\
  = \epsilon_{\text{mach}} \cdot b^{e} \quad \left| \frac{m \cdot b^{e}}{m \cdot b^{e}} \right| \\
  = \epsilon_{\text{mach}} \quad \left| \frac{m}{m} \right| \\
  \leq \epsilon_{\text{mach}} \quad \text{because } m \text{ is at least } 1 \\
\]

\[ m \geq 1 \]
- alternative characterization, $\varepsilon_{\text{mach}}$ smaller # s.t.,

$\text{fl}(1 + \varepsilon_{\text{mach}}) > 1$

**Examples:**
- (Ex. 1) $\varepsilon_{\text{mach}}$ (chop, nearest) = .25, .125
- IEEE SP $\varepsilon_{\text{mach}}$ (nearest) = $2^{-24} \approx 10^{-7}$ (about 7 decimal digits of precision)
- IEEE DP $\varepsilon_{\text{mach}}$ (nearest) = $2^{-53} \approx 10^{-16}$ (about 16 decimal digits of precision)

**Floating Point Math**

- adding or subtracting $\pm$, $\mp$
  - match **exponents** first
  - must **shift** smaller number
  - if the sum (or diff) contains more than $p$ digits, then the ones smaller than $p$ will be lost
  - **smallest number may be lost completely**

- multiplication ok
  - mult **mantissas** and sum **exponents**
  - still need to round though, because product will generally have more digits (up to $2p$)

**Example**

$1.23 \times 10^5$

$+ 1.00 \times 10^4$ ($10^3$, $10^2$)

- can also get **overflow** or **underflow**
  - **underflow** often ok - 0 is good approximation
  - **overflow** more serious problem - can't approximate the number in question
- IEEE standard gives us
  \[ \text{op} = +, -, \times, \div \]
as long as overflow doesn't occur.
- + and * commutative but *not* associative.
- Example: for \( \text{eps} < \text{eps}_\text{mach} \), and \( 2 \text{eps} > \text{eps}_\text{mach} \)
  \[
  (1 + \text{eps}) + \text{eps} = 1
  \\
  1 + (\text{eps} + \text{eps}) = 1 + 2 \text{eps} > 1
  
  \]

**Rounding Error Analysis**

Basic idea is:
\[
\text{fl}(x \text{ op } y) = (x \text{ op } y)(1 + \text{delta}),
\]
\[|\text{delta}| \leq \text{eps}_\text{mach}, \text{ and op} = +, -, \times, /\]
rearranging, get bound on relative *forward error*:
\[
|\text{fl}(x \text{ op } y) - (x \text{ op } y)|
\]
\[= |\text{delta}| \leq \text{eps}_\text{mach} \]

or, can interpret in terms of *backward error* (with \( \text{op} = + \)):
\[
\text{fl}(x + y) = (x + y)(1 + \text{delta}) = x(1+\text{delta}) + y(1+\text{delta})
\]

**Example:** Compute \( x(y+z) \)

\[
\text{fl}(y+z) = (y+z)(1+d_1), \ |d_1| \leq \text{eps}_\text{mach}
\]
\[
\text{and}
\]
\[
\text{fl}(x(y+z)) = (x(y+z)(1+d_1))(1+d_2), \ |d_2| \leq \text{eps}_\text{mach}
\]
\[= x(y+z)(1+d_1+d_2+d_1d_2)
\]
\[\approx x(y+z)(1+d_1)
\]
\[= x(y+z)(1+d), \ |d| = |d_1 + d_2| \leq 2 \text{ eps}_\text{mach}
\]

- pessimistic bound
- typical, multiples of \( \text{eps}_\text{mach} \) accumulate
- but in practice this is generally ok

**Cancellation**
problems can arise when subtracting two very close numbers
- result is exactly representable, but
- e.g., if the numbers differ by rounding error, this can basically
  leave rounding error only after subtracting

**Examples**

\[
\begin{align*}
x &= 1.92403 \times 10^2 \\
y &= 1.92275 \times 10^2 \\
\underline{\text{------------------------}} \\
0.00128 \times 10^2 &= .128 = 1.28 \times 10^{-1}
\end{align*}
\]

- only 3 significant digits in the result

BAD: computing *small quantity* as a difference of *large quantities*
\[
e^x = 1 + x + x^2/2 + x^3/3! + \ldots, \text{ for } \|x\| < 0
\]

**Example:** Quadratic formula

\[
ax^2 + bx + c = 0
\]

\[
-b \pm \sqrt{b^2 - 4ac} \\
b = \frac{\text{------------------------}}{2a}
\]

\[
\begin{align*}
0.05010 &\times 2 - 98.78 \times + 5.015 \\
\text{roots} &\approx 1971.605916, \quad \text{answer to 10 digits} \\
&\quad 0.05077069387 \\
\end{align*}
\]

\[
\begin{align*}
b^2 - 4ac &= 9757-1.005 = 9756 \quad \text{answer to 4 digits} \\
\sqrt{\text{"}} &= 98.77 \\
\text{roots:} &\quad (98.78 + 98.77) / 0.1002 = 1972, 0.09980
\end{align*}
\]

subtraction of two *close* numbers (cancellation error), followed
by division by *small* number (amplification)
Examples of Floating Point Issues

Finite precision

*Calculating $e$ in floating point*. The limit converges very slowly. By the time we hit machine epsilon for $1/n$, we still don’t have a very accurate estimate. On the other hand, the series formula converges up to full precision fairly quickly.

1. As a limit:

$$e = \lim_{n \to \infty} \left( 1 + \frac{1}{n} \right)^n.$$ 

2. Series formula for $e$:

$$e^x = 1 + x + x^2/2 + \ldots + x^n/n!$$

$$e^1 = 1 + 1/1 + 1/2 + \ldots + 1/n!$$

Overflow

Calculating

$$s = \frac{x}{\sqrt{1 + x^2}}.$$ 

Note that $s = \left(\sqrt{1 + \frac{1}{x^2}}\right)^{-1}$, so $s \to 1$ as $x \to \infty$. Computing $s$ directly in the first form given is problematic. As $x$ gets too large, the intermediate computation of $x^2$ will overflow, even though the final value we are interested in computing is close to 1. See below. Instead, the second form is better suited to computation.

(same precision)

```matlab
res = 0;
for i=1:4
    res(i) = single(10^i)/sqrt(1+single(10^i)^2)
end
```
for i=0:10:1000
    res(i) = 10^i/sqrt(1+(10^i)^2)
end

Cancellation

Example 1: Variance

Subtracting two large numbers to get at a small difference is a bad idea. There won’t be much if any precision to represent the small number we are interested in computing. As an example, let’s consider two ways of calculating the variance of a set of numbers.

Method 1: Calculate the mean first. Then calculate variances as sum of squares of distance to mean.

\[
\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \quad [\text{mean}]
\]

\[
\sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \quad [\text{variance}]
\]

Method 2: Use this (mathematically equivalent) formula:

\[
\sigma^2 = \frac{1}{n-1} \left( \sum_{i=1}^{n} x_i^2 - n\bar{x}^2 \right) \quad [\text{variance}]
\]

Method 2 requires only one pass over the data, whereas Method 1 requires two passes. Which method is better numerically? If \( \bar{x} \) is very large, but \( \sigma^2 \) is small, then Method 2 is very bad. The precision we have for \( \sigma^2 \) will depend on \( \sigma^2/(n\bar{x}^2) \). In the first formula the dependence is on \( \sigma^2/\bar{x} \).

Example 2: Series with alternating sign

Consider the series

\[e^{-x} = 1 - x + \frac{x^2}{2} - \frac{x^3}{3!} + \ldots + (-1)^n \frac{x^n}{n!} + \ldots\]

Computing this directly involves subtracting values of relatively large magnitude to compute a relatively small answer. This will result in loss of precision.

A better approach to computing this would be

\[
e^{-x} = \frac{1}{e^x} = \frac{1}{1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots}
\]