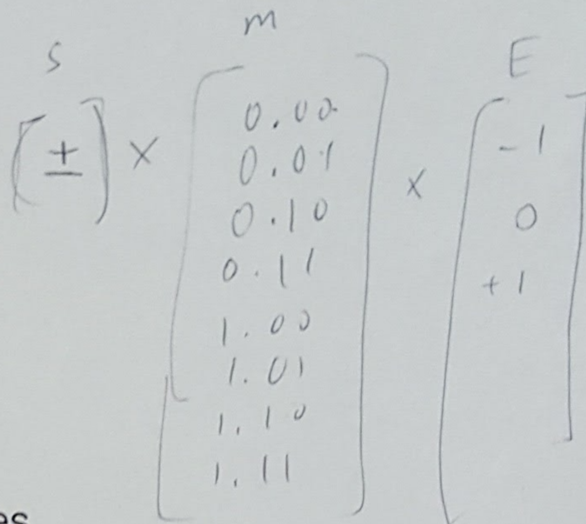
exponent: E

Example 1 (1):

```
b =  2
p =  3
L = -1
U =  1
```

start enumerating possibilities:

```
+-  m      E
+-  0.00  -1   -> 0
+-  0.00   0   -> 0
+-  0.00  +1   -> 0
+-  0.01  -1   -> 0.001
+-  0.01   0   -> 0.01
+-  0.01  +1   -> 0.1
+-  0.10  -1   -> 0.01
+-  0.10   0   -> 0.1
+-  0.10  +1   -> 1.0
```



duplicates!

In general, number of possibilities

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

$2 \times 8 \times 3 = 48$

but

- lots of duplicates
- non-unique representation

Normalization

- require the leading digit to be non-zero
- so mantissa, m

  1 <= 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!

$b=10 \quad (10.00)_{10} = (10)_{10}$

$b=2 \quad (10.00)_2 = (2)_{10}$

$b=3 \quad (10.00)_3 = (3)_{10}$

Properties

- finite and discrete system
- finite: how many (normalized) numbers can be represented?

count them:

```
2 * (b - 1) * b^(p-1) * (U - L + 1) + 1
S   d0        d1..dp-1    E            0
```

(Ex. 1 $2 \cdot 1 \cdot 2^2 \cdot 3 + 1 = 25$)

- what's the smallest (positive) normalized number? or "underflow level (UFL)"

(Ex. 1 UFL = .5)

```
1.0 ... 0 * b^L = b^L
```

- what's the biggest normalized number? or "overflow level (OFL)"

```
(b-1).(b-1) ... (b-1) * b^U
    = ( b - b^(-(p-1)) ) * b^U
    = ( 1 - b^(-p) ) * b^(U+1)
```

(Ex 1 $OFL = (1 - 2^{-3}) \cdot 2^2 = (1 - \frac{1}{8}) \cdot 4 = 4 - \frac{1}{2} = 3.5$)

```
Example 1 (2):
--------------
        b =  2
        p =  3
        L = -1
        U =  1
```

- number of normalized

```
2 * (b - 1) * b^(p-1) * (U - L + 1) + 1
    = 2 * (2 - 1) * 2^(3-1) * (1 - -1 + 1) + 1
    = 2 * 1 * 4 * 3 + 1
    = 25
```

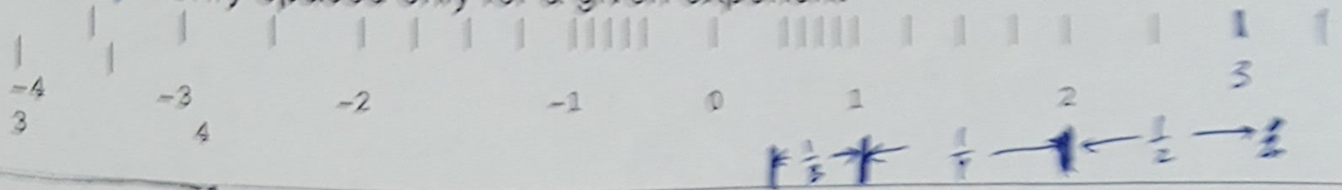
- UFL

```
b^L
    = 2^-1
    = .5
```

- OFL

(1 - b^(-p)) * b^(U+1)
  = (1 - 2^(-3)) * 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  <-- 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

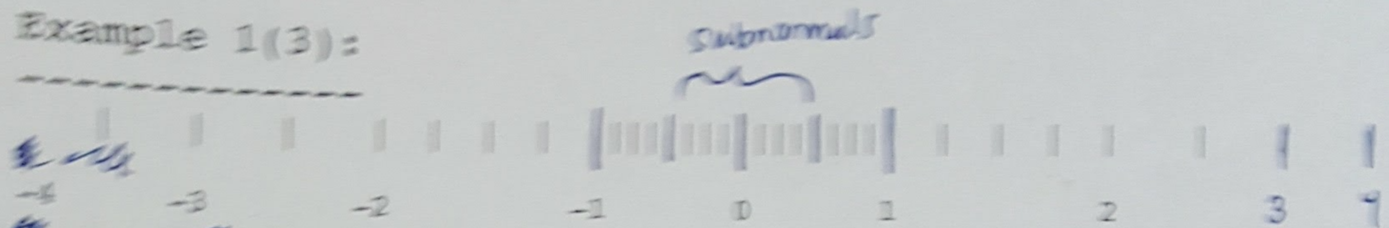
```
Example: Rounding
-----------------

Number     Chop    Round to nearest
------    -----    ----------------
1.649       1.6    1.6
1.650       1.6    1.6 (tie - round to even)
1.651       1.6    1.7
1.699       1.6    1.7
1.749       1.7    1.7
1.750       1.7    1.8 (tie - round to even)
1.751       1.7    1.8
1.799       1.7    1.8
```

Machine Precision

Subnormals
- normalized numbers: gap between 0 and b^L
- 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 * 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
- Inf
  - dividing finite number by 0
  - exceeding OFL
- NaN
  - undefined operation 0/0, Inf/Inf, 0*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   <-- 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
-----------------

Number    Chop   Round to nearest
------   -----   ----------------
1.649      1.6   1.6
1.650      1.6   1.6 (tie - round to even)
1.651      1.6   1.7
1.699      1.6   1.7
1.749      1.7   1.7
1.750      1.7   1.8 (tie - round to even)
1.751      1.7   1.8
1.799      1.7   1.8
```

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

octave-online.net

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

>> single((4/3-1))==single(1/3)
ans = 1
```

>> (4/3-1)-1/3
ans = -5.5511e-17

right way to compare:
>> abs((4/3 - 1) - 1/3) <= 1e-16
ans = 1

Machine Precision

```
    eps_mach
```
ε_{mach}

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

```
 _   .  _ _ _  ...    _      x x x ... x
 0 . 1 2 3 ... (p-1) p ...
```

- 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)
```
    1/2 b^(1-p)
```

- ~~tells us~~ bound the max possible relative error in representation

```
 | fl(x) - x |
 ------------- <=  eps_mach
     | x |
```
(for normalized #'s)

- check:
```
   <= eps_mach * b^e / | x |
    = eps_mach * b^e / (m * b^e)
    = eps_mach / m
   <= eps_mach
```
, because mantissa is at least 1

$m \geq 1$

- alternative characterization, ε_{mach} smallest # s.t.

fl(1 + eps_mach) > 1

Examples:
- (Ex. 1) eps_mach (chop, nearest) = .25, .125
- IEEE SP eps_mach (nearest) = 2^-24 ~= 10^-7 (about 7 decimal digits of precision)
- IEEE DP eps_mach (nearest) = 2^-53 ~= 10^-16 (about 16 decimal digits of precision)

Floating Point Math

+, −

- adding or subtracting +, −
  - 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)
- division (also need to round)

Example

at this point smaller # totally lost

```
    1.23 * 10^5
+   1.00 * 10^4 (10^3, 10^2)
```

$\begin{pmatrix} 1.23 \times 10^5 \\ + 1.00 \times 10^4 \end{pmatrix} \Rightarrow \begin{pmatrix} 1.23 \times 10^5 \\ .10 \times 10^5 \end{pmatrix}$ 1.33×10^5

$\begin{pmatrix} 1.23 \times 10^5 \\ 1.00 \times 10^2 \end{pmatrix} \Rightarrow \begin{pmatrix} 1.23 \times 10^5 \\ 0.00 \times 10^5 \end{pmatrix}$ (1.23×10^5)

- 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

```
x flop y = fl(x op y)
```

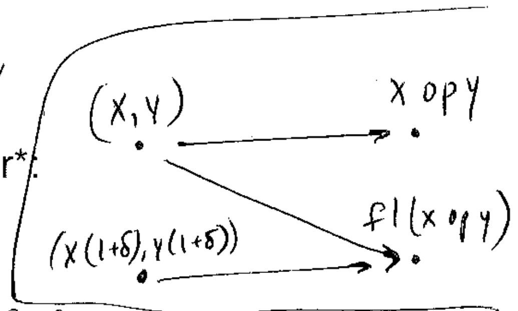
as long as overflow doesn't occur
- \+ and * commutative but *not* associative
- Ex: for eps < eps_mach, and 2 eps > eps_mach

```
( 1 + eps ) + eps  = 1
  1 + ( eps + eps )  = 1 + 2 eps > 1
```

Rounding Error Analysis

Basic idea is:

```
fl(x op y) = (x op y)(1 + delta),
    |delta| <= eps_mach, and op = +, -, *, /
```

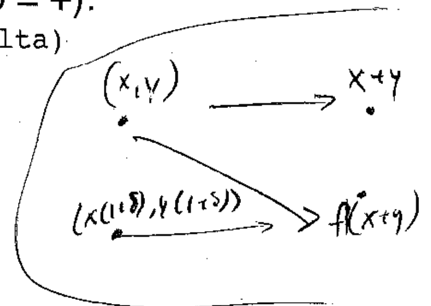


rearranging, get bound on relative *forward error*:

```
|fl(x op y) - (x op y)|
-----------------------  = |delta| <= eps_mach
     |(x op y)|
```

or, can interpret in terms of *backward error* (with op = +):

```
fl(x + y) = (x + y)(1 + delta) = x(1+delta) + y(1+delta)
```




```
Example: Compute x(y+z)
--------

fl(y+z) = (y+z)(1+d1), |d1|<=eps_mach
and
fl(x(y+z)) = (x(y+z)(1+d1))(1+d2), |d2|<=eps_mach
           = x(y+z)(1+d1+d2+d1d2)
          ~= x(y+z)(1+d1+d2)
           = x(y+z)(1+d), |d| = |d1 + d2| <= 2 eps_mach
```

- pessimistic bound
- typical, multiples of eps_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

```
  x = 1.92403 * 10^2
- y = 1.92275 * 10^2
  -----------------
      0.00128 * 10^2  = .128 = 1.28 * 10^-1
```

- 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! + ..., for x < 0
```

Example: Quadratic formula

```
ax^2 + bx + c = 0

     -b +- sqrt(b^2-4ac)
b = --------------------
             2a

0.05010 x^2 - 98.78 x + 5.015
roots ~= 1971.605916,    answer to 10 digits
         0.05077069387
```

```
b^2 - 4ac = 9757-1.005 = 9756  answer to 4 digits
sqrt( " ) = 98.77
roots: (98.78 +- 98.77) / 0.1002 = 1972, 0.09980
```

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:

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

```
n=20;
for i=1:n
  e_lim_bad(i) = (1+1/10^i)^(10^i);
end
```

2. Series formula for e:

$$e^x = 1 + x + x^2/2 + ... + x^n/n!$$
$$e^1 = 1 + 1 + 1/2 + ... + 1/n!$$

```
n=20;
e_series(1) = 1;
for i=2:n
  e_series(i) = e_series(i-1) + 1 / factorial(i-1);
end
```

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.

(single precision)

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

(double precision)

```
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 calcuate variances as sum of squares of distance to mean.

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

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

Method 2: Use this (mathematically equivalent) formula:

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

Method 2 requires only one pass over the data, where as Method 1 requires two passes. Which method is better numerically? If $\bar{x}$ is very large, but σ^2 is small, then Method 2 is very bad. The precision we have for σ^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

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