Floating-point operations

- The science of floating-point arithmetics
- IEEE standard
- Reference

*What every computer scientist should know about floating-point arithmetic, ACM computing survey, 1991*
Why learn more about floating-point operations I

Example:
- A one-variable problem

\[ \min_x f(x) \]

\[ x \geq 0 \]

- In your program, should you set an upper bound of \( x \)?
- \( x \) in your program may be wrongly increased to \( \infty \)
Why learn more about floating-point operations II

- What is the largest representable number in the computer?
- Is there anything called infinity?

Example:
- A ten-variable problem

\[
\min f(x) \\
0 \leq x_i, \ i = 1, \ldots, 10
\]
Why learn more about floating-point operations III

- After the problem is solved, want to know how many are zeros?
- Should you use
  
  ```c
  for (i=0; i < 10; i++)
    if (x[i] == 0) count++ ;
  ```

- People said: don’t do floating-point comparisons
  
  ```c
  epsilon = 1.0e-12 ;
  for (i=0; i < 10; i++)
    if (x[i] <= epsilon) count++ ;
  ```

How do you choose $\epsilon$?
Why learn more about floating-point operations IV

Is this true?
Floating-point Formats I

- We know float (single): 4 bytes, double: 8 bytes
  Why?
- A floating-point system
  base $\beta$, precision $p$, significand (mantissa) $d.d \ldots d$
- Example

$$0.1 = 1.00 \times 10^{-1} \quad (\beta = 10, p = 3)$$
$$\approx 1.1001 \times 2^{-4} \quad (\beta = 2, p = 5)$$

exponent: $-1$ and $-4$
- Largest exponent $e_{\text{max}}$, smallest $e_{\text{min}}$
Floating-point Formats II

- \( \beta^p \) possible significands, \( e_{\text{max}} - e_{\text{min}} + 1 \) possible exponents

\[
\left\lceil \log_2(e_{\text{max}} - e_{\text{min}} + 1) \right\rceil + \left\lceil \log_2(\beta^p) \right\rceil + 1
\]

bits for storing a number
1 bit for \( \pm \)

- But the practical setting is more complicated
- See the discussion of IEEE standard later
- Normalized: 1.00 \( \times 10^{-1} \) (yes), 0.01 \( \times 10^{1} \) (no)
- Now most used normalized representation

\( \Rightarrow \) cannot represent zero
A natural way for 0: $1.0 \times \beta^{e_{\min}-1}$

preserve the ordering

Will use $p = 3, \beta = 10$ for most later explanation
Relative Errors and Ulps I

- When $\beta = 10$, $p = 3$, $3.14159$ represented as $3.14 \times 10^0$

  ⇒ error = $0.00159 = 0.159 \times 10^{-2}$, i.e. 0.159 units in the last place

- $10^{-2}$: unit of the last place

- ulps: unit in the last place

- relative error $0.00159/3.14159 \approx 0.0005$

- For a number $d.d \ldots d \times \beta^e$, the largest error is

  $0.0\ldots0\beta' \times \beta^e, \beta' = \beta/2$
Relative Errors and Ulps II

Error = \frac{\beta}{2} \times \beta^{-p} \times \beta^e

1 \times \beta^e \leq \text{original value} < \beta \times \beta^e

relative error between

\frac{\beta}{2} \times \beta^{-p} \times \beta^e \quad \text{and} \quad \frac{\beta}{2} \times \beta^{-p} \times \beta^{e+1}

so

\text{relative error} \leq \frac{\beta}{2} \beta^{-p} \quad (1)

\frac{\beta}{2} \beta^{-p} = \beta^{-p+1}/2: \text{ machine epsilon}
The bound in (1)

- When a number is rounded to the closest, relative error bounded by $\epsilon$
ulps and $\epsilon$:

- $p = 3$, $\beta = 10$
- Example: $x = 12.35 \Rightarrow \tilde{x} = 1.24 \times 10^1$
  error $= 0.05 = 0.005 \times 10^1$
- ulps $= 0.01 \times 10^1$, $\epsilon = \frac{1}{2} \times 10^{-2} = 0.005$
- error 0.5 ulps
  relative error $0.05/12.35 \approx 0.004 = 0.8\epsilon$
- $8x = 98.8$, $8\tilde{x} = 9.92 \times 10^1$
  error $= 4.0$ ulps
  relative error $= 0.4/98.8 = 0.8\epsilon$.
- ulps and $\epsilon$ may be used interchangeably
Guard Digits I

- $p = 3, \beta = 10$
- Calculate $2.15 \times 10^{12} - 1.25 \times 10^{-5}$:
- Compute and then round

$$x = 2.15 \times 10^{12}$$
$$y = 0.0000000000000000125 \times 10^{12}$$

$$x - y = 2.1499999999999999875 \times 10^{12}$$

round to $2.15 \times 10^{12}$

Here we assume that computation is exactly done.
Guard Digits II

- Round and then compute

\[ x = 2.15 \times 10^{12} \]
\[ y = 0.00 \times 10^{12} \]
\[ x - y = 2.15 \times 10^{12} \]

Answer is the same
Reasonable as \( x \approx x - y \)

- Another example: \( 10.1 - 9.93 = 0.17 \)
Round and then compute

\[ 10.1 - 9.93 = 1.01 \times 10^1 - 0.99 \times 10^1 = 0.02 \times 10^1 \]
\[ = 2.00 \times 10^{-1} \]
error \[= 2.00 \times 10^{-1} - 0.17 = 0.03 \]
ulps \[= 0.01 \times 10^{-1} = 10^{-3} \]
error \[= 0.03 = 30\text{ulps} \]

Relative error

\[ = 0.03/0.17 = 3/17 \]
The error is quite large

- Compute and round

\[ 10.1 - 9.93 = 0.17 = 1.7 \times 10^{-1} \]

error = 0
The problem: cannot compute and then round
Guard Digits V

- How big can the error be? (if round and then compute)

**Theorem**

*Using $p$ digits with base $\beta$ for $x - y$, the relative error can be as large as $\beta - 1$*

**Proof:**

$x = 1.0\ldots0, y = 0.\underbrace{\eta\ldots\eta}_p, \eta = \beta - 1$

Correct solution $x - y = \beta^{-p}$

Computed solution $= 1.0\ldots0 - 0.\underbrace{\eta\ldots\eta}_{p-1} = \beta^{-p+1}$
Guard Digits VI

Relative error

\[ |\beta^{-p} - \beta^{-p+1}| = \beta - 1 \]

Example: \( p = 3, \beta = 10 \)

\( x = 1.00, y = 0.999, x - y = 0.001 = 10^{-3} \)

Computed solution = \( 1.00 \times 10^0 - 0.99 \times 10^0 = 0.01 \times 10^0 = 0.01 \)

Relative error

\[ \frac{|0.01 - 0.001|}{0.001} = 9 \]
Guard Digits VII

Such large errors occur if $x$ and $y$ are close

- Single guard digit

$p$ increased by 1 in the device for addition and subtraction

round and then compute

$1.010 \times 10^1 - 0.993 \times 10^1 = 0.017 \times 10^1$

Note $0.017 \times 10^1 = 1.70 \times 10^{-1}$ can be stored as $p = 3$

- That is, one additional digit in the process of subtraction. All values are still stored using $p = 3$
So in the device for subtraction, we should put additional digits

Another example:

\[
110 - 8.59 = 1.100 \times 10^2 - 0.085 \times 10^2 = 1.015 \times 10^2 \approx 1.02 \times 10^2
\]

Correct answer 101.41

Relative error around 0.006
Guard Digits IX

$$\epsilon = \frac{1}{2}\beta^{-p+1} = \frac{1}{2}10^{-2} = 0.005$$

**Theorem**

*Using $p + 1$ digits for $x - y \Rightarrow$ relative rounding error $< 2\epsilon$ ($\epsilon$: machine epsilon)*

**Proof:**

- Assume $x > y$
- Assume $x = x_0.x_1 \cdots x_{p-1} \times \beta^0$
  - The proof is similar if it’s not $\beta^0$
- If $y = y_0.y_1 \cdots y_{p-1}$ no error
Guard Digits X

- If $y = 0.y_1 \cdots y_p \Rightarrow 1$ guard digit, exact $x - y$ rounded to a closest number $\Rightarrow$ relative error $\leq \epsilon$

- In general $y = 0.0 \cdots 0 y_{k+1} \cdots y_{k+p}$
  $\bar{y}$: $y$ truncated to $p + 1$ digits

$$|y - \bar{y}| < (\beta - 1)(\beta^{-p-1} + \beta^{-p-2} + \cdots + \beta^{-p-k})$$

$$\leq \beta^{-p}$$

(2)

$-p - 1$: we have $p + 1$ digits now

(Think about $p = 3, \beta = 10$, first digit truncated $\leq 9 \times 0.0001 = 9 \times 10^{-4}$)
After $y$ is truncated, we need to calculate

$$x - \bar{y}$$

It’s rounded to

$$x - \bar{y} + \delta$$

$$|\delta| \leq \beta^{-p} \leq (\beta/2)\beta^{-p} = \epsilon$$

The first inequality comes from that $x$ and $\bar{y}$ both have $p + 1$ digits

error: $$(x - y) - (x - \bar{y} + \delta) = \bar{y} - y - \delta$$
case 1: if $x - y \geq 1$, 

$$
\text{relative error } \frac{|\bar{y} - y - \delta|}{x - y} \leq \frac{|\bar{y} - y - \delta|}{1} \\
\leq \beta^{-p}[(\beta - 1)(\beta^{-1} + \cdots + \beta^{-k}) + \beta/2] \\
= \beta^{-p}[(\beta - 1)\beta^{-k}(1 + \cdots + \beta^{k-1}) + \beta/2] \\
= \beta^{-p}[(\beta - 1)\beta^{-k} \frac{1 - \beta^k}{1 - \beta} + \beta/2] \\
= \beta^{-p}[(1 - \beta^{-k}) + \beta/2] \\
< \beta^{-p}(1 + \beta/2) \leq 2\epsilon
$$
case 2: $x - \bar{y} \leq 1$: enough digits $\delta = 0$

the smallest $x - y$: (smallest $x - \text{largest } y$)

$$1.0 - 0.0 \ldots 0 \rho \ldots \rho > (\beta - 1)(\beta^{-1} + \cdots + \beta^{-k})$$

$k$ zeros, $p \rho$'s, $\rho = \beta - 1$, from (2) the relative error

$$\leq \frac{|\bar{y} - y - \delta|}{(\beta - 1)(\beta^{-1} + \cdots + \beta^{-k})}$$

$$< \frac{(\beta - 1)\beta^{-p}(\beta^{-1} + \cdots + \beta^{-k})}{(\beta - 1)(\beta^{-1} + \cdots + \beta^{-k})} = \beta^{-p} < 2\epsilon$$

case 3: $x - y < 1$ but $x - \bar{y} > 1$
Guard Digits XIV

We show that this situation is impossible
If \( x - \bar{y} = 1.0\cdots1 \Rightarrow x - y \geq 1 \): a contradiction

Why \( x - y \) must be \( \geq 1 \):

\[
|y - \bar{y}| < \beta^{-p} = 0.0\cdots1
\]

- Conclusion: adding some guard digits can reduce the error
  Especially when subtracting two nearby numbers
- Cost: the adder is one bit wider (cheap)

Most modern computers have guard digits
Catastrophic cancellation and benign cancellation

Catastrophic cancellation:

\[ b = 3.34, \ a = 1.22, \ c = 2.28, \ b^2 - 4ac = 0.0292 \]

\[ b^2 \approx 11.2, \ 4ac \approx 11.1 \implies \text{answer} = 0.1 \]

error = 0.1 - 0.0292 = 0.0708

answer = 0.0292 = 2.92 \times 10^{-2}

ulps = 0.01 \times 10^{-2} = 10^{-4}

0.0708 \approx 708 \text{ ulps}

Happens when subtracting two close numbers
Benign cancellation: subtracting exactly known numbers, by guard digits

\[ \Rightarrow \text{small relative error} \]

In the example, \( b^2 \) and \( 4ac \) already contain errors
Avoid Catastrophic Cancellation

- By rearranging the formula
- Example

\[-b + \sqrt{b^2 - 4ac} \over 2a\]  

\(3\)

- If \(b^2 \gg 4ac\) ⇒ no cancellation when calculating \(b^2 - 4ac\) and \(\sqrt{b^2 - 4ac} \approx |b|\)
- Then \(-b + \sqrt{b^2 - 4ac}\) has a catastrophic cancellation if \(b > 0\)
Avoid Catastrophic Cancellation II

- Multiplying $-b - \sqrt{b^2 - 4ac}$, if $b > 0$

$$\frac{2c}{-b - \sqrt{b^2 - 4ac}}$$ (4)

- Use (3) if $b < 0$, (4) if $b > 0$

- Difficult to remove all catastrophic cancellations, but possible to remove most by reformulations

- Another example: $x^2 - y^2$

  Assume $x \approx y$

  $(x - y)(x + y)$ is better than $x^2 - y^2$
Avoid Catastrophic Cancellation III

$x^2, y^2$ may be rounded $\Rightarrow x^2 - y^2$ may be a catastrophic cancellation

$x - y$ by guard digit

- A catastrophic cancellation is replaced by a benign cancellation

Of course $x, y$ may have been rounded and $x - y$ is still a catastrophic cancellation.

Again, difficult to remove all catastrophic cancellations, but possible to remove some
Avoid Catastrophic Cancellation IV

- Calculating area of a triangle

\[ A = \sqrt{s(s - a)(s - b)(s - c)}, \quad s = \frac{a + b + c}{2} \] (5)

\( a, b, c: \) length of three edges

If \( a \approx b + c, \) \( s = \frac{a + b + c}{2} \approx a, s - a \) may have a catastrophic error

Example: \( a = 9.00, b = c = 4.53 \)

\( s = 9.03, A = 2.342 \)

Computed solution: \( A = 3.04, \) error \( \approx 0.7 \) ulps \( = 0.01, \) error \( = 70 \) ulps
A new formulation by Kahan [1986], \( a \geq b \geq c \)

\[
A = \frac{\sqrt{(a + (b + c))(c - (a - b))(c + (a - b))(a + (b - c))}}{4}
\]  
(6)

\( A \approx 2.35 \), close to 2.342

HW 1-1: Calculate \( A = 3.04 \) using (5) and \( A = 2.35 \) using (6)
Avoid Catastrophic Cancellation VI

Note: to get $A = 3.04$ you need to calculate $s$ by

$$s = \frac{a + (b + c)}{2}$$

Note that for multiplication and square root we assume that exact calculation can be done and results are rounded.

- Conclusion: sometimes a formula can be rewritten to have higher accuracy using benign cancellation
- Only works if guard digit is used; most computers use guard digits now
Avoid Catastrophic Cancellation VII

- But reformulation is difficult!!
  You may think that you will never need to do this
- Two real cases:
- Line 213-216 of tron.cpp of LIBLINEAR version 2.11
  http://www.csie.ntu.edu.tw/~cjlin/liblinear/oldfiles

HW1-2: Check Eq. (13) of the paper
and explain how we avoid catastrophic cancellations
We do not consider the latest version of LIBLINEAR because some more complicated settings have been used.

- Probability outputs of LIBSVM

HW1-3: Repeat the experiment on page 5, line 12 of the paper
Discuss what you found
Exactly Rounded Operations I

- Round then calculate $\Rightarrow$ may not be very accurate
- Exactly rounded: compute exactly then rounded to the nearest $\Rightarrow$ usually more accurate
- The definition of rounding
- $12.5 \Rightarrow 12$ or $13$ ?
- Rounding up: $0, 1, 2, 3, 4 \Rightarrow$ down, $5, 6, 7, 8, 9 \Rightarrow$ up
  - Why called “rounding up”? Always up for $5$
- Rounding even:
  - the closest value with even least significant digit
50% probability up, 50% down
example: 12.5 ⇒ 12; 11.5 ⇒ 12

- Reiser and Knuth [1975] show rounding even may be better

**Theorem**

Let $x_0 = x$, $x_1 = (x_0 ⊕ y) ⊕ y$, ..., $x_n = (x_{n-1} ⊕ y) ⊕ y$, if $⊕$ and $⊖$ are exactly rounded using rounding even, then $x_n = x$, $∀n$ or $x_n = x_1$, $∀n ≥ 1$.

$x ⊕ y$: computed solution

- Consider rounding up,
Exactly Rounded Operations III

\[ \beta = 10,\ p = 3,\ x = 1.00,\ y = -0.555 \]

\[ x - y = 1.555,\ x \oplus y = 1.56,\ (x \ominus y) + y = 1.56 - 0.555 = 1.005,\ x_1 = (x \ominus y) \oplus y = 1.01 \]

\[ x_1 - y = 1.565,\ x_1 \ominus y = 1.57,\ (x_1 \ominus y) + y = 1.57 - 0.555 = 1.015,\ x_2 = (x_1 \ominus y) \oplus y = 1.02 \]

Increased by 0.01 until \( x_n = 9.45 \)

- **Rounding even:**

\[ x - y = 1.555,\ x \ominus y = 1.56,\ (x \ominus y) + y = 1.56 - 0.555 = 1.005,\ x_1 = (x \ominus y) \oplus y = 1.00 \]

\[ x_1 - y = 1.555,\ x_1 \ominus y = 1.56,\ (x_1 \ominus y) + y = 1.56 - 0.555 = 1.005,\ x_2 = (x_1 \ominus y) \oplus y = 1.00 \]
Exactly Rounded Operations IV

- How to implement “exactly rounded operations”? We can use an array of words or floating-points. But you don’t have an infinite amount of spaces.
- Goldberg [1990] showed that using 3 guard digits, the result is the same as using exactly rounded operations.
IEEE standard I

- IEEE 754 during 80s, now standard everywhere
- Two IEEE standards:
  - 754: specify $\beta = 2$, $p = 24$ for single, $\beta = 2$, $p = 53$ for double
  - 854 ($\beta = 2$ or 10, does not specify how floating-point numbers are encoded into bits)
- Why IEEE 854 allows $\beta = 2$ or 10 but not other numbers:
  - 10 is the base we use
  - smaller $\beta$ causes smaller relative error
smaller $\beta$: more precision. For example,

$$\beta = 16, \ p = 1 \text{ versus } \beta = 2, \ p = 4$$

- 4 bits for significand

$$\epsilon = \frac{16}{2} 16^{-1} = 1/2, \ \epsilon = \frac{2}{2} 2^{-4} = 1/16$$

We can see that $\epsilon$ of $\beta = 2, \ p = 4$ is smaller.

- However, IBM/370 uses $\beta = 16$. Why? Two possible reasons:
  First,
IEEE standard III

a number: 4 bytes = 32 bits

\( \beta = 16, p = 6, \) significand: \( 4 \times 6 = 24 \) bits,

exponents: \( 32 - 24 - 1 = 7 \) bits (1 bit for sign),

\( 16^{-2^6} \) to \( 16^{2^6} = 2^{2^8} \)

for \( \beta = 2 \) \( \Rightarrow \) 9 bits \((-2^8 \text{ to } 2^8 = 2^9)\) for exponents,

\( \Rightarrow 32 - 9 - 1 = 22 \) for significand

Same exponents, less significand for \( \beta = 2 \) (24 vs. 22)

Second,

Shifting: \( \beta = 16, \) less frequently to adjust exponents when adding or subtracting two numbers
IEEE standard IV

For modern computers, this saving is not important

- Single precision: $\beta = 2, p = 24$ (23 bits as normalized), exponent 8, 1 bit for sign ($32 = 23 + 8 + 1$)
- An example: $176.625 = 1.0101100101 \times 2^7$

0 10000110 010110010100000000000000

1 of 1.\cdots is not stored (normalized)

- Biased exponent (described later in detail)
  $10000110 = 128 + 4 + 2 = 134, 134 - 127 = 7$

Note that we have negative exponent
IEEE standard V

- Use rounding even

<table>
<thead>
<tr>
<th>Binary</th>
<th>Rounded</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.00011</td>
<td>10.00</td>
<td>(&lt; 1/2, down)</td>
</tr>
<tr>
<td>10.00110</td>
<td>10.01</td>
<td>(&gt; 1/2, up)</td>
</tr>
<tr>
<td>10.11100</td>
<td>11.00</td>
<td>(1/2, up)</td>
</tr>
<tr>
<td>10.10100</td>
<td>10.10</td>
<td>(1/2, down)</td>
</tr>
</tbody>
</table>

This example is from http://www.cs.cmu.edu/afs/cs/academic/class/15213-s12/www/lectures/04-float-4up.pdf

- A summary
### IEEE standard VI

<table>
<thead>
<tr>
<th>IEEE</th>
<th>Fortran</th>
<th>C</th>
<th>Bits</th>
<th>Exp.</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>REAL*4</td>
<td>float</td>
<td>32</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>Single-extended</td>
<td></td>
<td></td>
<td>44</td>
<td>≤11</td>
<td>32</td>
</tr>
<tr>
<td>Double</td>
<td>REAL*8</td>
<td>double</td>
<td>64</td>
<td>11</td>
<td>53</td>
</tr>
<tr>
<td>Double-extended</td>
<td>REAL*10</td>
<td>long double</td>
<td>≥80</td>
<td>≥15</td>
<td>≥64</td>
</tr>
</tbody>
</table>

32 = 8 + 24 but 44 ≠ 11 + 32

- 44 ≠ 11 + 32:
  Hardware implementation of extended precision normal don’t use a hidden bit
  (Remember we normalized each number so 1 is not stored)
IEEE standard VII

- It seems everyone is using double now
  But single is still needed sometime (if memory is not enough)
- Minimal normalized positive number

\[ 1 \times 2^{-126} \approx 1.17 \times 10^{-38} \]

\[ e_{\text{min}} = -126 \]

- 8 bits for exponent: 0 to 255. IEEE uses biased approach exponent

\[ (0 \text{ to } 255) - 127 = -127 \text{ to } 128 \]
IEEE standard VIII

- Why $e_{\text{min}} = -126$ but $e_{\text{max}} = 127$?
  
  reasons: $1/2^{e_{\text{min}}}$ not overflow, $1/2^{e_{\text{max}}}$ underflow, but less serious

- Thus, $-127$ for $0$ and denormalized numbers (discussed later), $-126$ to $127$ for exponents, $128$ for special quantity

- Motivation for extended precision: from calculator, display 10 digits but 13 internally

  Some operations benefit from using more digits internally
Example: binary-decimal conversion (Details not discussed here)

- Operations: IEEE standard requires results of addition, subtraction, multiplication and division exactly rounded.

- Exactly rounded: an array of words or floating-point numbers, expensive

- Goldberg [1990] showed using 3 guard digits the result is the same as using exactly rounded operations

Only little more cost
IEEE standard X

- Reasons to specify operations run on different machines $\Rightarrow$ results the same
- HW 2-1: write the binary format of $-250$ as a double floating-point number
- IEEE: square root, remainder, conversion between integer and floating-point, internal formats and decimal are correctly rounded (i.e. exactly rounded operations)
- Binary to decimal conversion
  Think about reading numbers from files
When writing a binary number to a decimal number and read it back, can we get the same binary number?

- Writing 9 digits is enough for short
  Though $10^8 > 2^{24}$, 8 digits are not enough
- 17 for double precision (proof not provided).

Example:

numbers in a data set from Matrix market:
IEEE standard XII

> tail s1rmq4m1.dat

8.2511736085618438E+01  2.5134528659924950E+01
-6.0042951255041466E+00  8.6599442206615524E+04
1.0026197619563723E+01  -1.3136837661844502E+04
-1.5108331040361231E+01  5.1423173996955084E+04
-1.1690286345961363E+03  1.6250726655807816E+03
8.2511736074473220E+01  1.5108331040361227E+01

Matrix market:

http://math.nist.gov/MatrixMarket/
A collection of matrix data
IEEE standard XIII

- Transcendental numbers:
  e.g., exp, log

- IEEE does not require transcendental functions to be exactly rounded
  Cannot specify the precision because they are arbitrarily long
Special quantities I

- On some computers (e.g., IBM 370) every bit pattern is a valid floating-point number.
- For IBM 370, $\sqrt{-4} = 2$ and it prints an error message.
  IEEE: $\text{NaN}$, not a number

  why $\sqrt{-4} = 2$ on IBM 370 $\Rightarrow$ every pattern is a number

- Special value of IEEE:
  $+0$, $-0$, denormalized numbers, $+\infty$, $-\infty$, NaNs (more than one NaN)
A summary

<table>
<thead>
<tr>
<th>Exponent</th>
<th>significand</th>
<th>represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e = e_{\text{min}} - 1$</td>
<td>$f = 0$</td>
<td>$+0, -0$</td>
</tr>
<tr>
<td>$e = e_{\text{min}} - 1$</td>
<td>$f \neq 0$</td>
<td>$0.f \times 2^{e_{\text{min}}}$</td>
</tr>
<tr>
<td>$e_{\text{min}} \leq e \leq e_{\text{max}}$</td>
<td></td>
<td>$1.f \times 2^{e}$</td>
</tr>
<tr>
<td>$e = e_{\text{max}} + 1$</td>
<td>$f = 0$</td>
<td>$\pm \infty$</td>
</tr>
<tr>
<td>$e = e_{\text{max}} + 1$</td>
<td>$f \neq 0$</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Why IEEE has NaN

Sometimes even $0/0$ occurs, the program can continue
Example: find \( f(x) = 0 \), try different \( x \)'s, even \( 0/0 \) happens, other values may be ok.

If \( b^2 - 4ac < 0 \)

\[
\frac{-b + \sqrt{b^2 - 4ac}}{2a}
\]

returns NaN

\(-b+\) NaN should be NaN

In general when a NaN is in an operation, result is NaN

Examples producing NaN:
### Special quantities IV

<table>
<thead>
<tr>
<th>Operation</th>
<th>NaN by</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>$\infty + (\infty)$</td>
</tr>
<tr>
<td>$\times$</td>
<td>$0 \times \infty$</td>
</tr>
<tr>
<td>/</td>
<td>$0/0, \infty/\infty$</td>
</tr>
<tr>
<td>REM</td>
<td>$x$ REM 0, $\infty$ REM $y$</td>
</tr>
<tr>
<td>$\sqrt{}$</td>
<td>$\sqrt{x}$ when $x &lt; 0$</td>
</tr>
</tbody>
</table>
\[ \beta = 10, \; p = 3, \; e_{\text{max}} = 98, \; x = 3 \times 10^{70}, \]
\[ x^2 \text{ overflow and replaced by } 9.99 \times 10^{98}?? \]
In IEEE, the result is \( \infty \)

- Note \( 0/0 = \text{NaN}, \; 1/0 = \infty, \; -1/0 = -\infty \)
  \[ \Rightarrow \text{nonzero divided by 0 is } \infty \text{ or } -\infty \]
  Similarly, \( -10/0 = -\infty \), and \( -10/-0 = +\infty \)
  (\( \pm0 \) will be explained later)

- \( 3/\infty = 0, \; 4 - \infty = -\infty, \; \sqrt{\infty} = \infty \)

- How to know the result?
  replace \( \infty \) with \( x \), let \( x \to \infty \)
Infinity II

Example:

\[
\frac{3}{\infty} : \lim_{x \to \infty} \frac{3}{x} = 0
\]

If limit does not exist \( \Rightarrow \) NaN

- \( \frac{x}{x^2 + 1} \) vs \( \frac{1}{x + x^{-1}} \)
  \( \frac{x}{x^2 + 1} \): if \( x \) is large, \( x^2 \) overflow, \( \frac{x}{\infty} = 0 \) but not \( \frac{1}{x} \).
  \( \frac{1}{x + x^{-1}} \): \( x \) large, \( \frac{1}{x} \) ok
  \( \frac{1}{x + x^{-1}} \) looks better but what about \( x = 0 \)?
  \( x = 0, \frac{1}{0 + 0^{-1}} = \frac{1}{0 + \infty} = \frac{1}{\infty} = 0 \)

- If no infinity arithmetic, an extra instruction needed to test if \( x = 0 \), may interrupt the pipeline
Signed zero I

- Why do we have +0 and −0?
  First, it is available (1 bit for sign)
  if no sign, \(1/(1/x)) = x\) fails when \(x = \pm \infty\)
  \(x = \infty, 1/x = 0, 1/0 = +\infty\)
  \(x = -\infty, 1/x = 0, 1/0 = +\infty\)
- Compare +0 and −0: if (\(x == 0\))
  IEEE defines +0 = −0
- IEEE: \(3 \times (+0) = +0, +0/(-3) = -0\)
Signed zero II

• ±0 useful in the following situations:

\[
\log x \equiv \begin{cases} 
-\infty & x = 0 \\
\text{NaN} & x < 0 
\end{cases}
\]

A small underflow negative number ⇒ \(\log x\) should be NaN

\(x\) underflow ⇒ round to 0, if no sign, \(\log x\) is \(-\infty\) but not NaN
Signed zero III

- With $\pm 0$, we have

$$\log x = \begin{cases} 
-\infty & x = +0 \\
\text{NaN} & x = -0 \\
\text{NaN} & x < 0 
\end{cases}$$

Positive underflow $\Rightarrow$ round to $+0$

- Very useful in complex arithmetic

$$\sqrt{1/z} \text{ and } 1/\sqrt{z}$$

$z = -1$, $\sqrt{1/(-1)} = \sqrt{-1} = i$, $1/\sqrt{-1} = 1/i = -i$

$\Rightarrow \sqrt{1/z} \neq 1/\sqrt{z}$
Signed zero IV

- This happens because square root is multi-valued.
  \[ i^2 = (-i)^2 = -1 \]

- However, by some restrictions (or ways of calculation), they can be equal
  \[ z = -1 = -1 + 0i, \]
  \[ 1/z = 1/(-1 + 0i) = -1 + (-0)i \]
  so \[ \sqrt{1/z} = \sqrt{-1 + (-0)i} = -i \]
  \[ \Rightarrow -0 \text{ is useful} \]

- Disadvantage of \( +0 \) and \( -0 \):
  \[ x = y \iff 1/x = 1/y \text{ is destroyed} \]
  \[ x = 0, y = -0 \Rightarrow x = y \text{ under IEEE} \]
Signed zero V

\[ \frac{1}{x} = +\infty, \quad \frac{1}{y} = -\infty, \quad +\infty \neq -\infty \]

- There are always pros and cons for floating-point design
If

\[ \text{if (a < 0)} \]

always holds and \( b \) is neither too large nor too small, how do we guarantee

\[ \text{if a/max(b, 0.0) < 0} \]

always holds

- If \( \max(b, 0.0) \) returns \(-0.0\), then it may not hold
- For the max function, should we use

\[ (x>y)? \ x:y \]

or
(x < y) ? y : x

- Your max need to return +0.0 but not −0.0
- How to specifically assign +0.0 and −0.0?
- How to use subroutines to get the sign of a number?
- In a regular program, if you write 0.0, is it +0.0 or −0.0?

Find the statement in the manual saying that 0.0 means +0.0

- Do some experiments to check your arguments
- Use Java but not other systems
Denormalized number I

- $\beta = 10$, $p = 3$, $e_{\min} = -98$, $x = 6.87 \times 10^{-97}$,
  $y = 6.81 \times 10^{-97}$
- $x, y$ are ok but $x - y = 0.6 \times 10^{-98}$ rounded to 0,
even though $x \neq y$
- How important to preserve

$$x = y \iff x - y = 0$$

- if $(x \neq y)$ \{ $z = 1/(x-y)$; \}
The statement is true, but $z$ becomes $\infty$
Tracking such bugs is frustrating
Denormalized number II

- IEEE uses denormalized numbers

  Guarantee $x = y \iff x - y = 0$

  Details of how this is done are not discussed here

- Most controversial part in IEEE standard

  It caused long delay of the standard

- If denormalized number is used, $0.6 \times 10^{-98}$ is also a floating-point number

- Remember we do not store 1 of $1.d \cdots d$

- How to represent denormalized numbers?

  Recall for valid value, $e \geq e_{\text{min}}$ and we have $1.d \cdots d \times 2^e$
For denormalized numbers, we let $e = e_{\text{min}} - 1$ and the corresponding value be

$$0.d \cdots d \times 2^{e+1} = 0.d \cdots d \times 2^{e_{\text{min}}}$$

Why not

$$1.d \cdots d \times 2^{e_{\text{min}}-1}$$

can’t represent

$$0.0x \cdots x \times 2^{e_{\text{min}}}$$

$$6.87 \times 10^{-97} - 6.81 \times 10^{-97} \Rightarrow \text{underflow due to cancellation}$$
Denormalized number IV

Underflow: smaller than the smallest floating-point number

- An example of using denormalized numbers

\[
\frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i
\]

If \( c \) or \( d > \sqrt{\beta} \beta^{e_{\text{max}}/2} \) \( \Rightarrow \) overflow
Denormalized number V

overflow: larger than the maximal floating-point number

Smith’s formula

\[ a + bi = \begin{cases} \frac{a+b(d/c)}{c+d(d/c)} + \frac{b-a(d/c)}{c+d(d/c)}i & \text{if } \left| d \right| < \left| c \right| \\ \frac{b+a(c/d)}{d+c(c/d)} + \frac{-a+b(c/d)}{d+c(c/d)}i & \text{if } \left| d \right| \geq \left| c \right| \end{cases} \]

avoid overflow

However, using Smith’s formula, without denormalized numbers
Denormalized number VI

If

\[ a = 2 \times 10^{-98}, \quad b = 1 \times 10^{-98}, \quad c = 4 \times 10^{-98}, \quad d = 2 \times 10^{-98} \]

then

\[
\frac{d}{c} = 0.5, \quad c + d(\frac{d}{c}) = 5 \times 10^{-98}, \\
b(\frac{d}{c}) = 1 \times 10^{-98} \times 0.5 = 0 \\
a + b(\frac{d}{c}) = 2 \times 10^{-98}
\]

Solution = 0.4, wrong
If denormalized numbers are used, $0.5 \times 10^{-98}$ can be stored,

$$a + b(d/c) = 2.5 \times 10^{-98} \Rightarrow 0.5$$

the correct answer

- Usually hardware does not support denormalized numbers directly
- Using software to simulate
- Programs may be slow if a lot of underflow
We have mentioned things like overflow, underflow. What are other exceptional situations?

Motivation: usually when exceptional condition like 1/0 happens, you may want to know

IEEE requires vendors to provide a way to get status flags

IEEE defines five exceptions: overflow, underflow, division by zero, invalid operation, inexact

Overflow: larger than the maximal floating-point number
Exception, Flags, Trap handlers II

Underflow: smaller than the smallest floating-point number

- **Invalid:**
  \[ \infty + (-\infty), 0 \times \infty, 0/0, \infty/\infty, \]
  \[ x \text{ REM } 0, \infty \text{ REM } y, \sqrt{x}, x < 0, \text{ any comparison involves a NaN} \]

- **Invalid returns NaN; NaN may not be from invalid operations**

- **Inexact:** the result is not exact
  \[ \beta = 10, p = 3, 3.5 \times 4.2 = 14.7 \text{ exact}, \]
  \[ 3.5 \times 4.3 = 15.05 \Rightarrow 15.0 \text{ not exact} \]
inexact exception is raised so often, usually we ignore it

**Exception** | when trap disabled | argument to handler
--- | --- | ---
overflow | $\pm \infty$ or $\pm 1.1 \cdots 1 \times 2^{e_{\text{max}}}$ | $\text{round}(x 2^{-\alpha})$
underflow | $0, \pm 2^{e_{\text{min}}}$, or denormal | $\text{round}(x 2^{\alpha})$
division by zero | $\infty$ | operands
invalid | NaN | operands
inexact | $\text{round}(x)$ | $\text{round}(x)$

- **Trap handler**: special subroutines to handle exceptions
You can design your own trap handlers

- In the above table, “when trap disabled” means results of operations if trap handlers not used
- \( \alpha = 192 \) for single, \( \alpha = 1536 \) for double
- reason: you cannot really store \( x \)

Examples of using trap handlers described later
Compiler Options I

- Compiler may provide a way so the program stops if an exception occurs
- Easy for debugging
- Example: SUN’s C compiler (I learned this on an old machine)
- Reason: gcc doesn’t have this to explicitly detect exceptions
- \texttt{-ftrap}=t
Compiler Options II

- t: %all, %none, common, [no%]invalid, [no%]overflow, [no%]underflow, [no%]division, [no%]inexact.

- common: invalid, division by zero, and overflow.

- The default is -ftrap=%none.

- Example: -ftrap=%all,no%inexact means set all traps, except inexact.

- If you compile one routine with -ftrap=t, compile all routines of the program with the same -ftrap=t option

otherwise, you can get unexpected results.
Example: on the screen you will see

Note: IEEE floating-point exception flags raised:
Inexact; Underflow;
See the Numerical Computation Guide, ieee_flags

gcc:
- fno-trapping-math: default -ftrapping-math
  Setting this option may allow faster code if one relies on “non-stop” IEEE arithmetic
- ftrapv
Generates traps for signed overflow on addition, subtraction, multiplication
Trap Handler I

- Example:

\[
\text{do } \{ \\
\quad \ldots \\
\} \text{ while } \{\text{not } x \geq 100;\}
\]

If \( x = \text{NaN} \), an infinite loop
Any comparison involving NaN is wrong
- A trap handler can be installed to abort it
- Example:
Calculate $x_1 \times \cdots \times x_n$ may overflow in the middle (the total may be ok!):

```c
for (i = 1; i <= n; i++)
    p = p * x[i];
```

- $x_1 \times \cdots \times x_r, r \leq n$ overflow but $x_1 \times \cdots \times x_n$ may be in the range
- $e^{\sum \log(x_i)} \Rightarrow$ a solution but less accurate and costs more
- A possible solution
for (i = 1; i <= n; i++) {
    if (p * x[i] overflow) {
        p = p * pow(10,-a);
        count = count + 1;
    }
    p = p * x[i];
}

p = p * pow(10, a*count);
An Example of Handlers I

- Example using SUN’s numerical computation guide
  Again, old. Reason of not using existing systems such a glibc: so you can have HW

- standard math library libm.a
  exp, pow, log, ...

- On SUN machines, there are additional math library: libsunmath.a
  exp2, exp10, ..., ieee_flags, ieee_handler, ieee_retrospective

- A program:
An Example of Handlers II

```c
#include <stdio.h>
#include <sys/ieeefp.h>
#include <sunmath.h>
#include <siginfo.h>
#include <ucontext.h>

void handler(int sig, siginfo_t *sip, ucontext_t *uap)
{
    unsigned code, addr;

    code = sip->si_code;

    code = sip->si_code;
}
```
An Example of Handlers III

```c
addr = (unsigned) sip->si_addr;
fprintf(stderr, "fp exception %x at address %x \n", code, addr);
}
int main()
{
    double x;

    /* trap on common floating point exceptions */
    if (ieee_handler("set", "common", handler) != 0)
```
printf("Did not set exception handler \n");

/* cause an underflow exception (not reported) */
x = min_normal();
printf("min_normal = %g \n", x);

x = x / 13.0;
printf("min_normal / 13.0 = %g \n", x);

/* cause an overflow exception (reported) */
x = max_normal();
printf("max_normal = \%g \n", x);
x = x * x;
printf("max_normal * max_normal = \%g \n", x);

ieee_retrospective(stderr);
return 0;
}

• Result:
An Example of Handlers VI

\[
\begin{align*}
\text{min\_normal} & = 2.22507 \times 10^{-308} \\
\text{min\_normal} / 13.0 & = 1.7116 \times 10^{-309} \\
\text{max\_normal} & = 1.79769 \times 10^{308} \\
\text{fp exception} & = \text{4 at address 10d0c} \\
\text{max\_normal} \times \text{max\_normal} & = 1.79769 \times 10^{308} \\
\end{align*}
\]

Note: IEEE floating-point exception flags raised:
- Inexact;
- Underflow;

IEEE floating-point exception traps enabled:
- overflow;
- division by zero;
- invalid operation;

See the Numerical Computation Guide, ieee\_flags(3M),
ieee\_handler(3M)
An Example of Handlers VII

- invalid, division, and overflow sometimes called common exceptions here
  `ieee_handler("set", "common", handler)` means handlers used for common exceptions
- `min_normal / 13.0`: using denormalized numbers
  handler: subroutines to handle exceptions
- HW 3-1: regenerate this example using GNU C library
How to find GNU C library information: on linux, type
% info libc
check the category of “Arithmetics” and “Signal Handling”
The Use of Flags: An Example I

- Calculate $x^n$, $n$ : integer

```c
double pow(double x, int n)
{
    double tmp = x, ret = 1.0;

    for(int t=n; t>0; t/=2)
    {
        if(t%2==1) ret*=tmp;
        tmp = tmp * tmp;
    }

    return ret;
}
```
The Use of Flags: An Example II

\[
\begin{align*}
\times^{16} &= (\times^2)^8 = \cdots, \\
\times^{15} &= \times(\times^2)^7, \text{ treat } \times^2 \text{ as the new } \times \\
\times^{15} &= \times(\times^2)^7 = \times(\times^2)(\times^4)^3 = \times(\times^2)(\times^4)(\times^8)^1
\end{align*}
\]

- If \( n < 0 \), we need to use

\[
x^n = (1/x)^{-n} = 1/(x)^{-n}
\]

\( \text{pow}(1/x, -n) \) less accurate, \( 1/\text{pow}(x, -n) \) is better

There is already error on \( 1/x \)
The Use of Flags: An Example III

Example: $2^{-5} = (1/2)^5$ and $1/(2^5)$

- A small problem on using $1/pow(x, -n)$:
  - If $pow(x, -n)$ underflow (i.e. when $x < 1$, $n < 0$),
    - either underflow trap handler or underflow status flag set $\Rightarrow$ incorrect
  - $x^{-n}$ underflow, $x^n$ overflow or be in range
    - $(e_{\text{min}} = -126, 2^{-e_{\text{min}}} = 2^{126} < 2^{127} = 2^{e_{\text{max}}})$

- Turn off overflow & underflow trap enable bits, save overflow & underflow status bits
  - Compute $1/pow(x, -n)$
The Use of Flags: An Example IV

If neither overflow nor underflow status is set ⇒ restore them
If one is set, restore & calculate $\text{pow}(1/x, -n)$, which causes correct exception to occur

- Practically the calculation of $\text{pow}()$ is more complicated
  - e.g. google $\text{e_pow.c}$ and $\text{e_log.c}$
  - In glibc-2.17/sysdeps/ieee754/dbl-64, $\text{e_pow.c}$ has 420 lines
Another example: calculate arccos using arctan

\[
\arccos x = 2 \arctan \sqrt{\frac{1 - x}{1 + x}}
\]

\[
\cos \theta = x = 2 \cos^2 \frac{\theta}{2} - 1 = 1 - 2 \sin^2 \frac{\theta}{2}
\]

\[
\cos \frac{\theta}{2} = \sqrt{\frac{x + 1}{2}}, \quad \sin \frac{\theta}{2} = \sqrt{\frac{1 - x}{2}}, \quad \tan \frac{\theta}{2} = \sqrt{\frac{1 - x}{1 + x}}
\]

Hence

\[
\arccos x = 2 \arctan \sqrt{\frac{1 - x}{1 + x}}
\]
The Use of Flags: An Example VI

- Consider \( x = -1 \)

\[
\arctan(\infty) = \frac{\pi}{2} \Rightarrow \arccos(-1) = \pi
\]

- A small problem:

\[
\frac{1-x}{1+x}
\]

causes the divide-by-zero flag set though \( \arccos(-1) \) not exceptional

- Solution: save divide-by-zero flag, restore after \( \arccos \) computation
Let’s start with a simple example

```c
#include <stdio.h>

int main()
{
    float a = 123.123;
    printf("%.10f\n", a);
    printf("%.10f\n", a*a);

    a = 123.125;
    printf("%.10f\n", a);
```
printf("%.10f\n", a*a);

}  

- Results are
  $gcc test.c;./a.out
  123.1230010986
  15159.2734375000
  123.1250000000
  15159.7656250000
  $gcc -m32 test.c;./a.out
  123.1230010986
15159.2733995339
123.1250000000
15159.7656250000

- m 32 generates code for a 32-bit environment (because we don’t have a 32-bit machine)

That is, same code gives different results under 32 and 64-bit environments

Why?
On 32 bit, 387 floating-point coprocessor is used. From gcc manual, “The temporary results are computed in 80-bit precision instead of the precision specified by the type, resulting in slightly different results compared to most of other chips.”

In other words, they somehow violate IEEE standard

But 123.123 has infinite digits after transformed to binary
Compiler options can help to make things more consistent. For example, "-ffloat-store": “Do not store floating-point variables in registers, and inhibit other options that might change whether a floating-point value is taken from a register or memory.”

$gcc -ffloat-store test.c; ./a.out
123.1230010986
15159.2734375000
123.1250000000
15159.7656250000
$gcc -ffloat-store -m32 test.c; ./a.out
123.1230010986
A Real Study VI

15159.2734375000
123.1250000000
15159.7656250000

- Note that other issues such as order of operations can also affect results.
- Consider running a real example using a machine learning software LIBSVM
- 64 bit:
A Real Study VII

$ .:/svm-train -c 100 -e 0.00001 heart_scale
        *...*
optimization finished, #iter = 2872
nu = 0.148045
obj = -2526.925470, rho = 1.145512
nSV = 107, nBSV = 9
Total nSV = 107

- 32bit:
A Real Study VIII

$ ./svm-train -c 100 -e 0.00001 heart_scale

.........*...*
optimization finished, #iter = 2819
nu = 0.148045
obj = -2526.925470, rho = 1.145515
nSV = 107, nBSV = 9
Total nSV = 107

- They are different
- Adding -ffloat-store -mfpmath=387 is not enough
- 64 bit:
A Real Study IX

$ make clean; make
$ ./svm-train -c 100 -e 0.00001 heart_scale
rm -f *~ svm.o svm-train svm-predict svm-scale

g++ -Wall -Wconversion -O3 -fPIC -ffloat-store -mfpmath=387 -c svm.cpp
g++ -Wall -Wconversion -O3 -fPIC -ffloat-store -mfpmath=387 svm-train.c svm.o -o svm-train -lm
g++ -Wall -Wconversion -O3 -fPIC -ffloat-store -mfpmath=387 svm-predict.c svm.o -o svm-predict -lm
g++ -Wall -Wconversion -O3 -fPIC -ffloat-store -mfpmath=387 svm-scale.c -o svm-scale

........*..*
optimization finished, #iter = 2863
nu = 0.148045
obj = -2526.925470, rho = 1.145512
nSV = 107, nBSV = 9
Total nSV = 107

- We also need to disable all optimization

- 64bit:

  
  $ \text{make clean; make}$

  $ \text{./svm-train \ -c 100 \ -e 0.00001 \ heart\_scale}$

  $ \text{rm -f *~ svm.o svm-train svm-predict svm-scale}$

  $ \text{g++ \ -ffloat\_store \ -mfpmath=387 \ -c \ svm.cpp}$

  $ \text{g++ \ -ffloat\_store \ -mfpmath=387 \ svm\_train.c \ svm\_scale.c}$

  $ \text{g++ \ -ffloat\_store \ -mfpmath=387 \ svm\_predict.c}$

  $ \text{g++ \ -ffloat\_store \ -mfpmath=387 \ svm\_scale.c}$

  ..........*....*
optimization finished, #iter = 3051
nu = 0.148045
obj = -2526.925470, rho = 1.145515
nSV = 107, nBSV = 9
Total nSV = 107

32 bit:

$ make clean; make
$ ./svm-train -c 100 -e 0.00001 heart_scale
rm -f *~ svm.o svm-train svm-predict svm-scale

g++ -m32 -ffloat-store -mfpmath=387 -c svm.cpp
g++ -m32 -ffloat-store -mfpmath=387 svm-train
g++ -m32 -ffloat-store -mfpmath=387 svm-predict.c svm.o -o svm-predict -lm

g++ -m32 -ffloat-store -mfpmath=387 svm-scale.c -o svm-scale

..........*....*

optimization finished, #iter = 3051
nu = 0.148045
obj = -2526.925470, rho = 1.145515
nSV = 107, nBSV = 9
Total nSV = 107