Floating Point Arithmetic is not Real

Tim Mattson

git clone https://github.com/tgmattso/Princeton2025.git

Acknowledgements: I borrowed some content from lectures on floating point arithmetic by Ianna Osborne and Wahid Redjeb.

Should we trust computer arithmetic?

Sleipner Oil Rig Collapse (8/23/91). Loss: **$700 million.**

$1.6 Billion in 2024 dollars

See http://www.ima.umn.edu/~arnold/disasters/sleipner.html

Linear elastic model using NASTRAN underestimated shear stresses by 47% resulted in concrete walls that were too thin.

NASTRAN is the world's most widely used finite element code … in heavy use since 1968

Outline

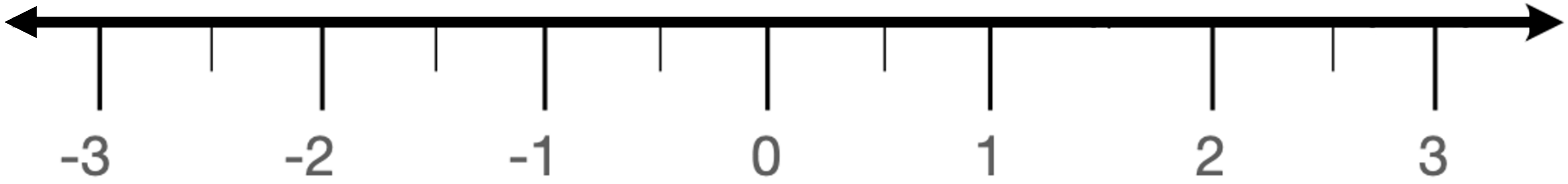
- Numbers for humans.   Numbers for computers

- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard

- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
  - Algebraic Properties of Floating Point Arithmetic

- Wrap-up/Conclusion

Numbers for Humans

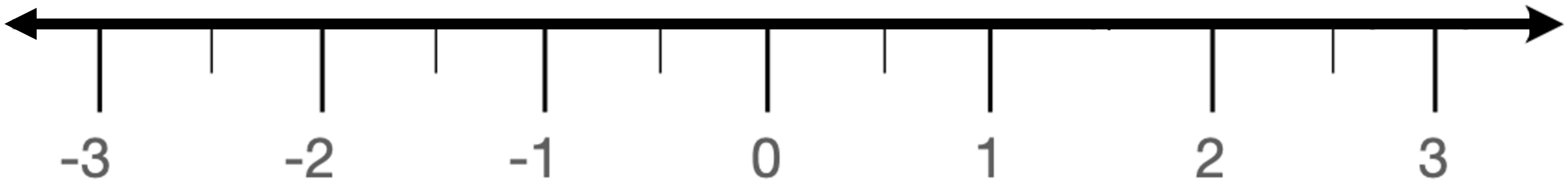


$\mathbb{R}$

Real Numbers: viewed as points on a line … pairs of real numbers can be arbitrarily close

Numbers for Humans

Arithmetic over Real Numbers



$\mathbb{R}$

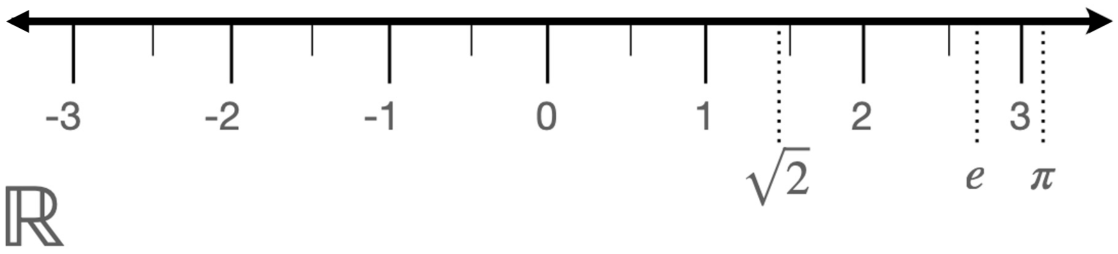
For the arithmetic operators, real numbers define a **closed set** … for well defined operations and any input real numbers, the arithmetic operation returns a real number.

A few key properties of Real Arithmetic:

- **Commutative over addition and multiplication**: (a+b) = (b+a) a*b = b*a
- **Associative**: (a+b)+c = a+(b+c) (a*b)*c = a*(b*c)
- **Multiplication distributes over addition**: c * (a+b) = c*a + c*b

Numbers for Humans

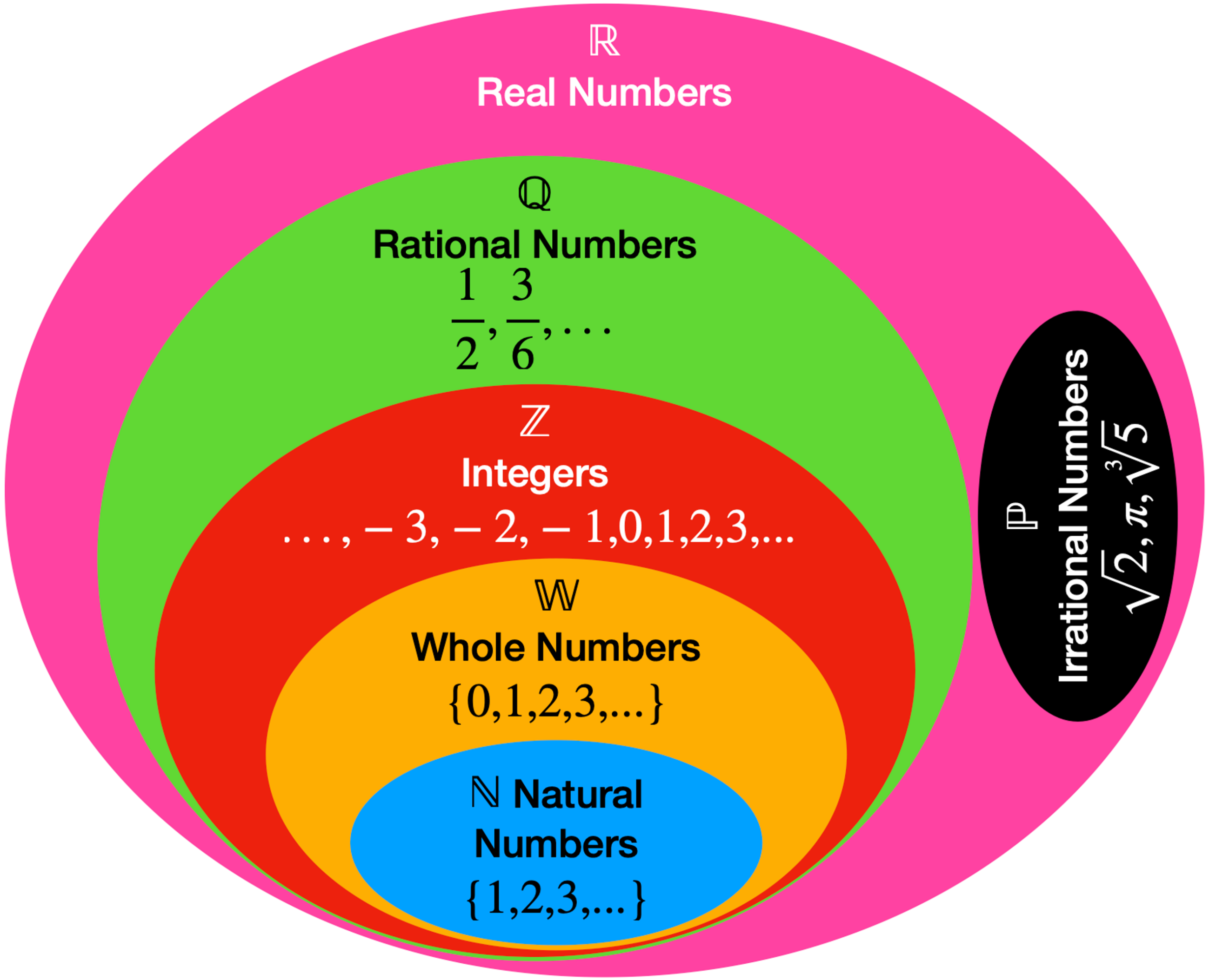
People use many different kinds of numbers



We are used to thinking of different kinds of numbers contained within the set of real numbers

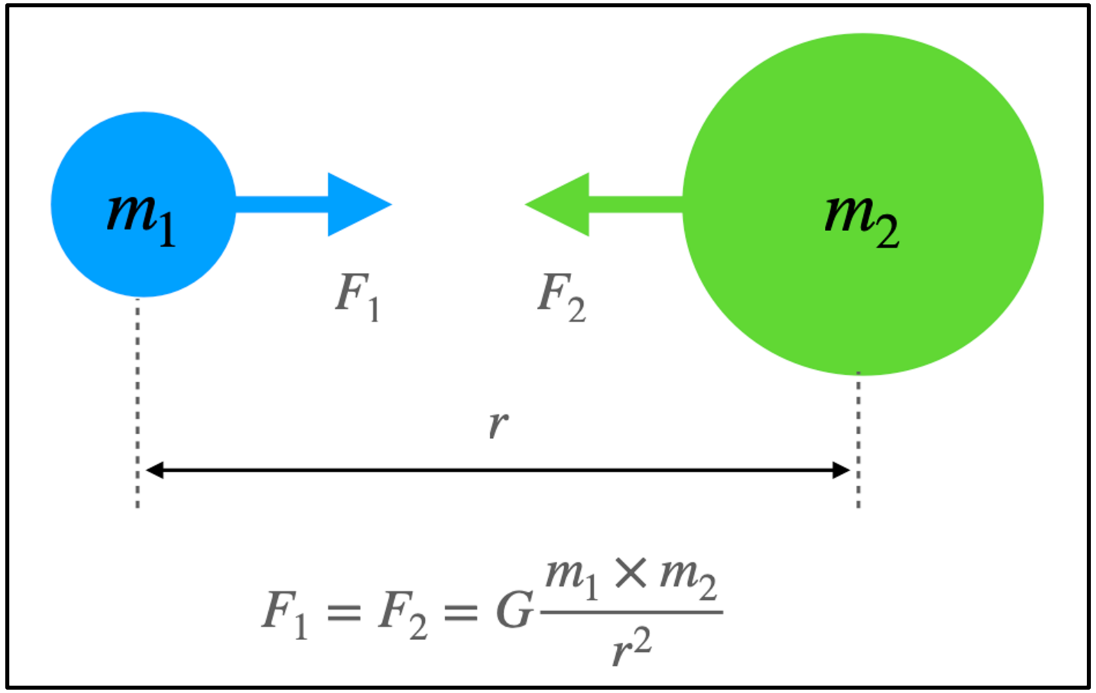
- **Integers**: equally spaced numbers without a fractional part
- **Rational numbers**: Ratios of integers
- **Whole numbers**: Positive integers and zero
- **Natural numbers**: Positive integers and not zero
- **Irrational numbers**: numbers that cannot be represented as a ratio of integers

The numbers on a computer are just another set of numbers embedded in the set of real numbers



Based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

Numbers for Humans: Example



$$G \approx 0.00000000006674 \ \frac{m^3}{kg * s^2}$$

Scientific Notation

$$0.00000000006674 \longrightarrow 6.674 \times 10^{-11}$$

The exponent tells us how far to "float" the decimal point.

significand *exponent*

$$G \approx 6.674 \times 10^{-11} \ m^3 \cdot kg^{-1} \cdot s^{-2}$$

radix

Based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

Numbers for Humans

How do we represent Real Numbers?

significand *exponent*

$G \approx 6.674 \times 10^{-11}$ m^3·kg^{-1}·s^{-2}

radix

$$G \approx (6 \cdot 10^0 + 6 \cdot 10^{-1} + 7 \cdot 10^{-2} + 4 \cdot 10^{-3}) \cdot 10^{-11}$$

We can generalize the above to any real number as …

$$x = (-1)^{sign} \sum_{i=0}^{\infty} d_i \mathbf{b}^{-i} \mathbf{b}^{exp}$$

… where ***b*** is the **radix**

$sign \in \{0,1\}$, $b \geq 2$, $d_i \in \{0, \ldots, (b-1)\}$, $d_0 > 0 \; when \; x \neq 0$, $\mathbf{b}, i, \exp \in [integer]$

Based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

Numbers for Humans

How do we represent Real Numbers?

significand *exponent*

$G \approx 6.674 \times 10^{-11}$ $m^3 \cdot kg^{-1} \cdot s^{-2}$

radix

$$G \approx (6 \cdot 10^0 + 6 \cdot 10^{-1} + 7 \cdot 10^{-2} + 4 \cdot 10^{-3}) \cdot 10^{-11}$$

What about numbers for computers?

We can generalize the above to any real number as …

$$x = (-1)^{sign} \sum_{i=0}^{\infty} d_i b^{-i} b^{exp}$$

Human's deal nicely with ∞.
Computers do not.

$sign \in \{0,1\}$, $b \geq 2$, $d_i \in \{0, \ldots, (b-1)\}$, $d_0 > 0$ $when$ $x \neq 0$, $b, i, \exp \in [integer]$

Humans like a radix = 10.
Which radix is best for a computer?

Based on slides from Ianna Osborne, CoDaS-HEP, July 19, 2023

Numbers for Computers

Computers work with a restricted subset of real numbers…

Finite precision … restricted to N digits.

N is tied to the length of a "word" in a computer's architecture. This is typically the width of the registers in a microprocessor's register file.

$$x = (-1)^{sign} \sum_{i=0}^{N} d_i b^{-i} b^{exp}$$

$$sign \in \{0,1\}, \quad b \geq 2, \quad d_i \in \{0, \ldots, (b-1)\}, \quad d_0 > 0 \; when \; x \neq 0, \quad \mathrm{b}, i, \exp \in [integer]$$

Which radix (b) is best for a computer?

Binary has $d_i \in \{0,1\}$. Naturally maps onto representation as transistors used to implement computer logic.

Decimal has $d_i \in \{0, \ldots, 9\}$. Requires four bits per digit … which wastes space (since four bits can encode {0,…,15}).

Exercise: Playing with "numbers for computers"

- You are a software engineer working on a device that tracks objects in time and space.

- The device increments time in "clock ticks" of 0.01 seconds.

- Write a program that tracks time by **accumulating N clock-ticks**. N is typically large … around 100 thousand. Output from the function is elapsed seconds expressed as a float.
  - Assume you are working with an embedded processor that does not support the type double.
  - This is part of an interrupt driven, real time system, hence track "time" not "number of ticks" since this time may be needed at any moment.

- What does your program generate for large N?

Accumulating clock ticks (0.01): Solution

```
#include <stdio.h>
#define time_step  0.01f

float CountTime(int Count)
{
    float sum     = 0.0f;

    for (int i=0; i<Count;i++)
        sum += time_step;

    return sum;
}
int main()
{
   int Count = 500000;
   float time_val;

   time_val = CountTime(Count);
   printf(" sum = %f or %f\n",time_val,time_step*Count);
}
```

```
% gcc -O0 hundreth.c
% ./a.out
 sum = 4982.411132. or 5000.000000
%
```

Why did summing 0.01 fail?

I saw this slogan on a T-shirt years ago

Converting a decimal number (0.01) to fixed point binary

0.01 is equal to $\frac{1}{100}$.

The fraction $\frac{1}{2^N}$ nearest but less than or equal to $\frac{1}{100}$ is $\frac{1}{128}$ (N=7)	$0.01_{10} \approx 0.0000001_2$
The remainder $\frac{1}{100} - \frac{1}{128} = \frac{7}{3200} \approx \frac{1}{457}$. The fraction $\frac{1}{2^N}$ nearest but less than or equal to this remainder is $\frac{1}{512}$ (N=9)	$0.01_{10} \approx 0.000000101_2$
The remainder $\frac{7}{3200} - \frac{1}{512} = \frac{3}{12800} \approx \frac{1}{4266}$. The fraction $\frac{1}{2^N}$ nearest but less than or equal to this remainder is $\frac{1}{8196}$ (N=13)	$0.01_{10} \approx 0.0000001010001_2$

- Continuing to 32 bits we get 0.00000010100011110101110000101000… but it's still not done.
- The denominator of the number 1/100 includes a relative prime (5) to the radix of binary numbers (2). Hence, there is no way to exactly represent 1/100 in binary!

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\frac{1}{2^N}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$	$\frac{1}{1024}$	$\frac{1}{2048}$	$\frac{1}{4096}$	$\frac{1}{8196}$	$\frac{1}{16384}$	$\frac{1}{32768}$	$\frac{1}{65536}$	$\frac{1}{131072}$

Real numbers on a computer are represented as finite precision numbers

- **Conclusion**: Not all decimal numbers have an exact representation as binary numbers.
  - You can have computations where the answer does NOT have an exact binary representation … in other words, **fixed precision arithmetic is NOT a closed set**.

```
float c, b = 1000.2f;
c = b - 1000.0;
printf (" %f", c);
```

Output: 0.200012

- The best we can hope for is that the computer does the computation “exactly” then rounds to the nearest binary number.

Real numbers on a computer are represented as finite precision numbers

- **Conclusion**: Many decimal numbers do not have an exact representation as binary numbers.
  - You can have computations where the answer does NOT have an exact binary representation … in other words, **fixed precision arithmetic is NOT a closed set**.

```
float c, b = 1000.2f;
c = b - 1000.0;
printf (" %f”, c);
```

Output: 0.200012

Who cares?
Does this really matter?

- The best we can hope for is that the computer does the computation “exactly” then rounds to the nearest binary number.

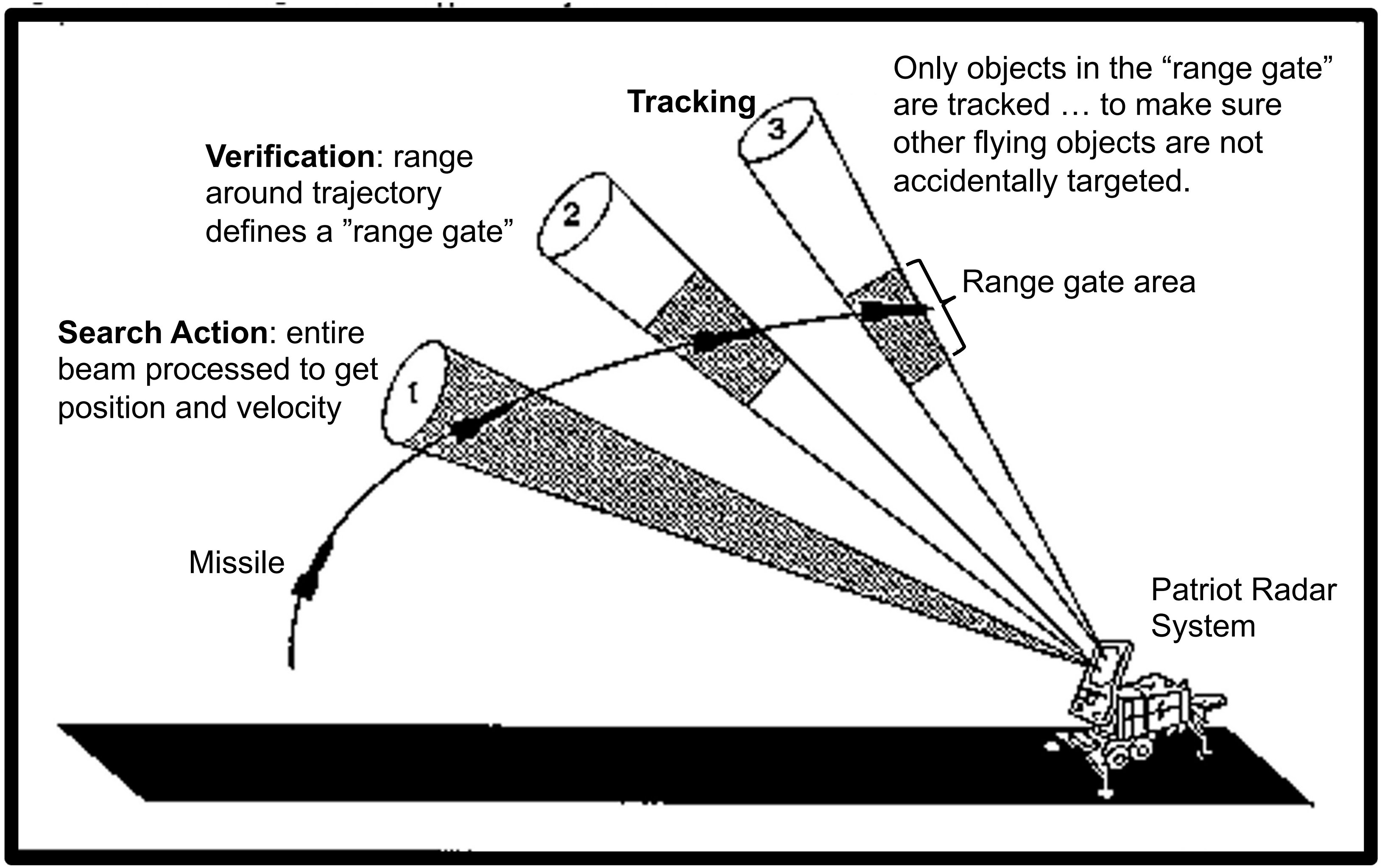
Patriot Missile system

Patriot missile incident (2/25/91) . Failed to stop a scud missile from hitting a barracks,

See http://www.fas.org/spp/starwars/gao/im92026.htm

Patriot missile system: how it works

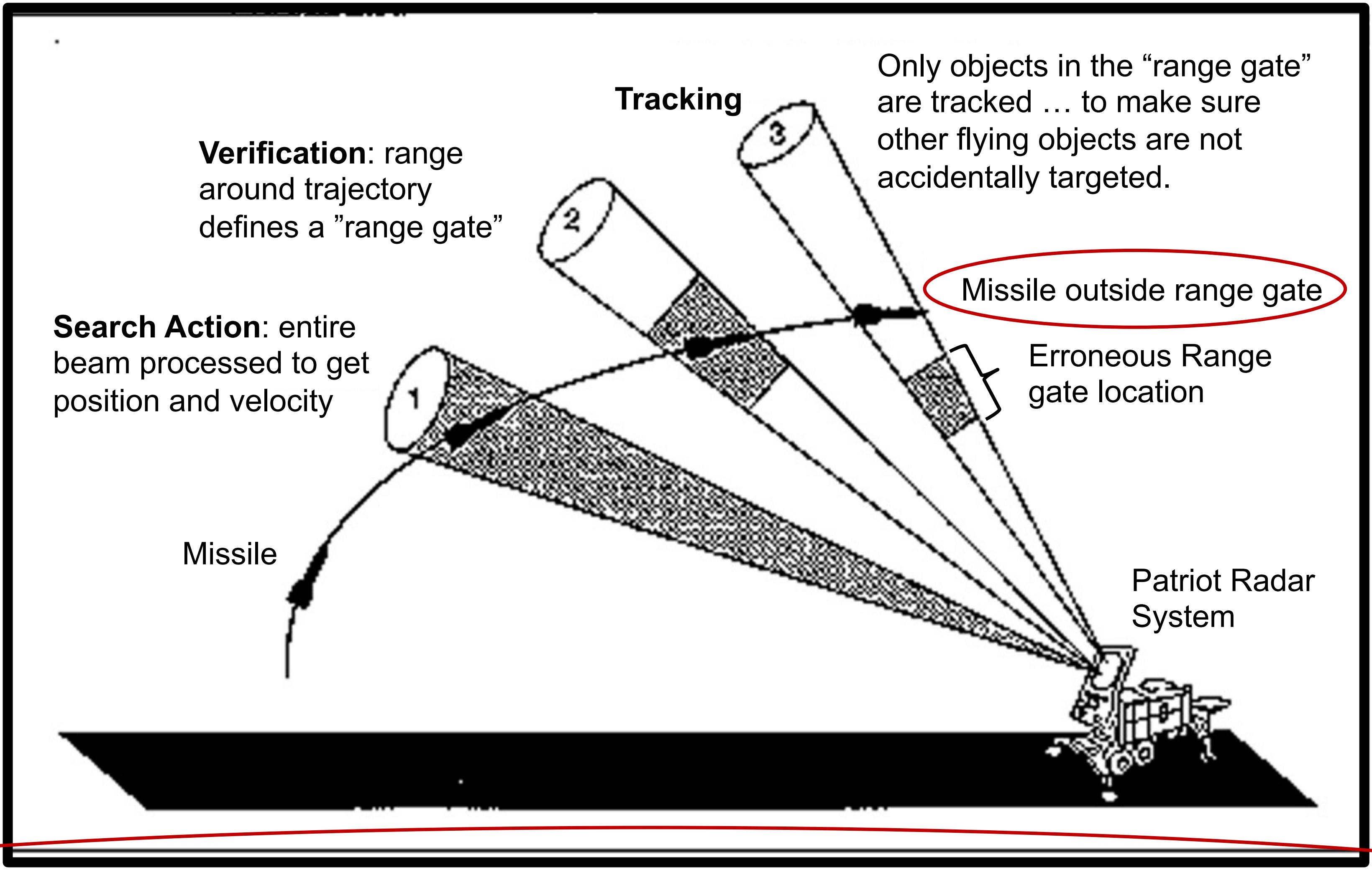
Incoming object detected as an enemy missile due to properties of the trajectory. Velocity and position from Radar fixes trajectory



24 bit clock counter defines time. Range calculations defined by real arithmetic, so convert to floating point numbers.

Patriot missile system: Disaster Strikes

Incoming object detected as an enemy missile due to properties of the trajectory. Velocity and position from Radar fixes trajectory



28 soldiers killed

Accumulating clock-ticks (int) by the float representation of 0.01 led to an error of 0.3433 seconds after 100 hours of operation which, when you are trying to hit a missile moving at Mach 5, corresponds to an error of 687 meters

Exercise: Playing with "numbers for computers"

- You are a software engineer working on a device that tracks objects in time and space.

- The device increments time in "clock ticks" of 0.01 seconds. **Propose (and test) a value for the clock tick that makes the program work.**

- Write a program that tracks time by **accumulating N clock-ticks**. N is typically large … around 100 thousand. Output from the function is elapsed seconds expressed as a float.
  - Assume you are working with an embedded processor that does not support the type double.
  - This is part of an interrupt driven, real time system, hence track "time" not "number of ticks" since this time may be needed at any moment.

- What does your program generate for large N?

Exercise: Playing with "numbers for computers"

- You are a software engineer working on a device that tracks objects in time and space.

- The device increments time in "clock ticks" of 0.01 seconds. **Propose (and test) a value for the clock tick that makes the program work.**

- Wr ... und 10

```
> ./a.out
 dt = 0.00781250000000000.  // dt=1.0/128.0 … one over a power of two
 sum = 39062.50000000000000000, dt*Count=39062.50000000000000000
```

  - Assume you are working with an embedded processor that does not support the type double.
  - This is part of an interrupt driven, real time system, hence track "time" not "number of ticks" since this time may be needed at any moment.

- What does your program generate for large N?

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set

Outline

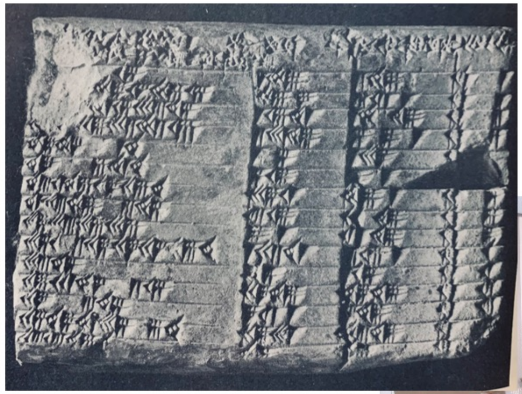
- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
  - Algebraic Properties of Floating Point Arithmetic
- Wrap-up/Conclusion

PRINCETON UNIVERSITY

iris hep

Floating-point Arithmetic Timeline

"…the next generation of application programmers and error analysts will face new challenges and have new requirements for standardization. Good luck to them!"
https://grouper.ieee.org/groups/msc/ANSI_IEEE-Std-754-2019/background/ieee-computer.pdf



Babylonians worked with floating-point sexagesimal numbers

1750 B.C.

- Average calculation speed: addition – 0.8 seconds, multiplication – 3 seconds[1]
- Arithmetic unit: Binary floating-point, 22-bit, add, subtract, multiply, divide, square root[1]

By Venusianer, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3632073

1941 Zeus Z3

1960's a mainframe era

1964 **IBM 360**
CDC 6600
DEC 10

DEC PDP-11

1970's boom of minicomputers

1976

1977 **IEEE p745**

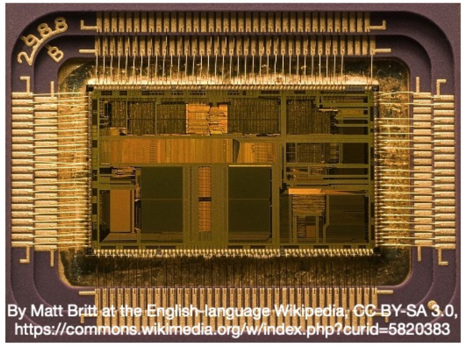
Intel began to design a floating-point co-processor for its i8086/8 and i432 microprocessors

1985 IEEE 754-1985

IEEE 754-1987

Intel 80486

the first tightly-pipelined[c] x86 design as well as the first x86 chip to include more than one million transistors. It offered a large on-chip cache and an integrated floating-point unit.



By Matt Britt at the English-language Wikipedia, CC-BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5820383

1989

The floating point unit fully conforms to the ANSI/IEEE standard 754-1985 for floating point arithmetic.

IEEE 754-2008

IEEE 754-2019

you are here

IEEE 754-2029

binary and decimal floating-point arithmetic

subjected to review at least every 10 years

"new kinds of computational demands might eventually encompass new kinds of standards, particularly for fields like artificial intelligence, machine vision and speech recognition, and machine learning. Some of these fields obtain greater accuracy by processing more data faster rather than by computing with more precision – rather different constraints from those for traditional scientific computing."

- each hardware manufacturer had its own type of floating point
- different machines from the same manufacturer might have different types of floating point
- when floating point was not supported in the hardware, the different compilers emulated different floating point types

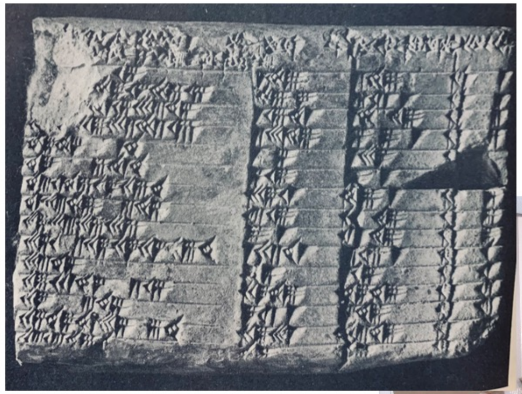
Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

PRINCETON UNIVERSITY

Summarizing the key points …

Floating-point Arithmetic Timeline

“…the next generation of application programmers and error analysts will face new challenges and have new requirements for standardization. Good luck to them!”
https://grouper.ieee.org/groups/msc/ANSI_IEEE-Std-754-2019/background/ieee-computer.pdf



Babylonians worked with floating-point sexagesimal numbers

The concept of floating point numbers is very old (1750 BC)

- Average calculation speed: addition – 0.8 seconds, multiplication – 3 seconds[1]
- Arithmetic unit: Binary floating-point, 22-bit, add, subtract, multiply, divide, square root[1]

By Venusianer, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3632073

1941 Zeus Z3

1960's mainframe era

1964 IBM 360

CDC 6600

The era of floating point chaos

...manufacturer had its own typ...
...es from the same manufactu...
...point
...int was not supported in the hardware, the different compilers emulated different floating point types

DEC

1970's boom of minicomputers

IEEE-754 floating point is born … thanks to a team led by William Kahan

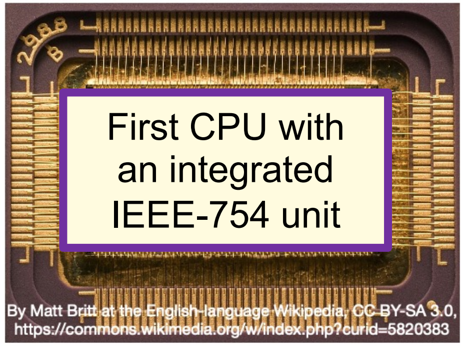
1976 1977

Intel produces the first chip to support IEEE-754 in 1980 (the 8087 coprocessor)

1985

Intel 80486

the first tightly-pipelined[c] x86 design as well as the first x86 chip to include more than one million transistors. It offered a large on-chip cache and an integrated floating-point unit.



First CPU with an integrated IEEE-754 unit

By Matt Britt at the English-language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=5820383

The floating point unit fully conforms to the ANSI/ IEEE standard 754-1985 for floating point arithmetic.

1989

754-1987

...sign a floating-point ... its i8086/8 and i432

“new kinds of ... might eventua... kinds of stand... fields like arti... machine visio... recognition, a... Some of these... accuracy by p... faster rather t... more precisio... constraints fro... scientific computing.”

IEEE-754 floating point continues to evolve … next version expected in 2029

IEEE 754-2008

IEEE 754-2019

binary and decimal floating-point arithmetic

subjected to review at least every 10 years

you are here

IEEE 754-2029

Floating Point Number systems

Computers work with finite precision, floating point numbers …

$$x = (-1)^{sign} \sum_{i=0}^{p} d_i b^{-i} b^e = \pm d_0.d_1 \ldots d_{p-1} \times b^e$$

$sign \in \{0,1\}$

$d_i \in \{0, \ldots, (b-1)\}$

$e_{min} \leq e \leq e_{max}$

$b \geq 2$	The radix … usually 2 or 10 (but occasionally 8 or 16)
$p \geq 1$	The precision … the number of digits in the significand
e_{max}	The largest exponent
e_{min}	The smallest exponent (generally $1 - e_{max}$)

These four numbers define a unique set of floating point numbers … written as $\boxed{F(b, p, e_{min}, e_{max})}$

Floating Point Number systems: Normalized numbers

Consider representations of the decimal number 0.1

$$1.0\times10^{-1}, \qquad 0.1\times10^{0}, \qquad 0.01\times10^{1}$$

- These are all the same number, just represented differently depending on the choice of exponent.
- That ambiguity is confusing, so we require that $d_0 \neq 0$ so numbers between b^{min} and b^{max} have a single unique representation.
- We call these normalized floating point numbers $\boxed{F^*(b, p, e_{min}, e_{max})}$

$$x = (-1)^{sign}\sum_{i=0}^{p} d_i b^{-i} b^e = \pm d_0 1.d_1 \ldots d_{p-1} \times b^e$$

$$sign \in \{0,1\}$$

$$d_i \in \{0, \ldots, (b-1)\}$$

$$d_0 \neq 0$$

$$e_{min} \leq e \leq e_{max}$$

- $x = 0 \ \ and \ x < b^{e_{min}}$ do not have normalized representations.

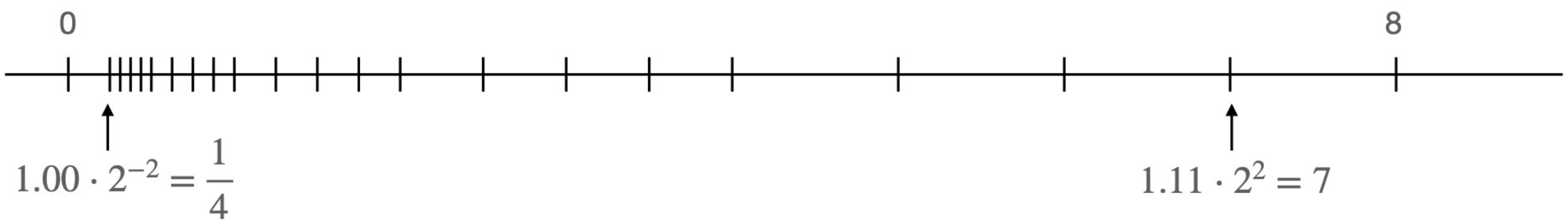
$b \geq 2$	The radix … usually 2 or 10 (but occasionally 8 or 16)
$p \geq 1$	The precision … the number of digits in the significand
e_{max}	The largest exponent
e_{min}	The smallest exponent (generally $1 - e_{max}$)

Normalized Representation

$F^*(2,3,-2,2)$

$d_0 . d_1 d_2$	$e=-2$	$e=-1$	$e=0$	$e=1$	$e=2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7

Equivalent decimal values for all patterns of normaliezed binary digits and exponents



F*(Radix, Precision, emin, emax)

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

Outline

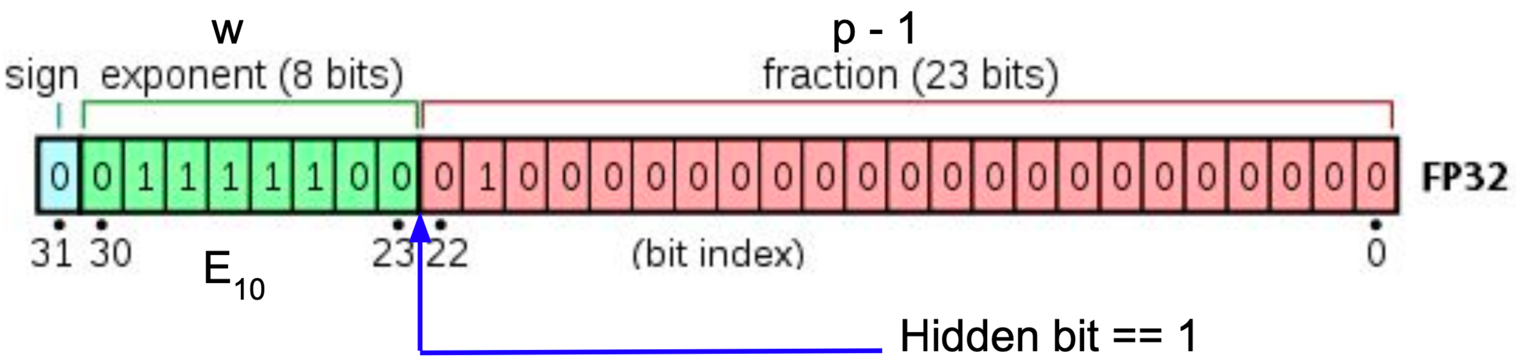
- Numbers for humans.   Numbers for computers

- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard

- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
  - Algebraic Properties of Floating Point Arithmetic

- Wrap-up/Conclusion

IEEE 754 Floating Point Numbers

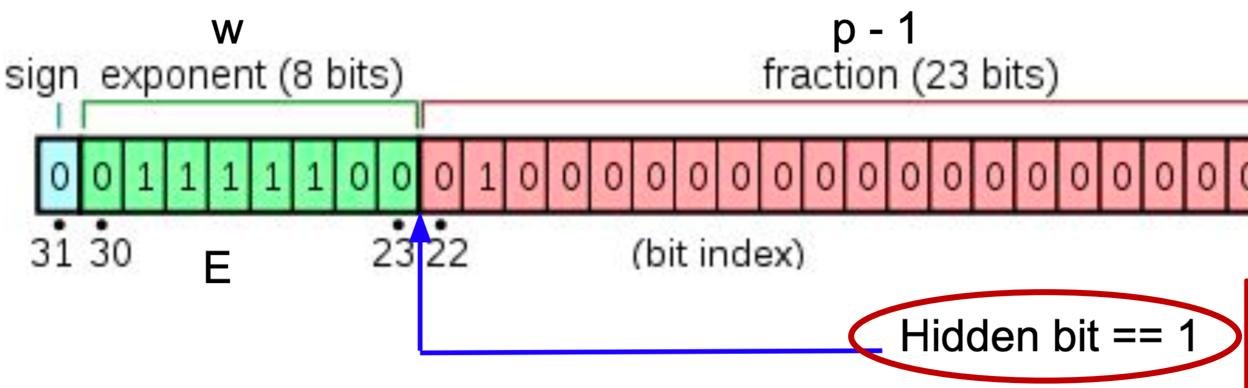


IEEE Name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

- Exponent: $E = e - e_{min} + 1$, *w* bits
- $e_{max} = -e_{min} + 1$

Source: Wahid Redjeb, ESC'22

IEEE 754 Floating Point Numbers



For normalized, binary floating point numbers, the lead bit is "always" one so there is no reason to store it. The bit is "hidden"

IEEE Name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

- Exponent: $E = e - e_{min} + 1$, *w* bits
- $e_{max} = -e_{min} + 1$

Source: Wahid Redjeb, ESC'22

Exceptions

- Certain situations outside “normal” behavior are defined as **Exceptions**.   Two cases:
  1. **Silent**: An exception occurs, a status flag is set, a result is returned and the computation proceeds. This is the typical case.
  2. **Signaled**: The exception occurs, a signal is raised, and an optional trap function is invoked. Trapping can be set through compiler switches but can seriously slow down code. This is very rarely done … except by professionals writing low-level math libraries.

- The Exceptions defined by IEEE 754 include the following
  - **Underflow**: The result is too small to be represented as a normalized float. Produces a signed zero or a denormalized float.
  - **Overflow**: The result is too large to be represented by a normalized float. Produces a signed infinity.
  - **Divide-by-zero**: A float is divided by zero. The appropriate infinity is returned.
  - **Invalid**: The operation or its result is ill-defined (such as 0.0/0.0). A NaN is returned.
  - **Inexact**: The result of the floating point operation is not exact and must be rounded. The rounded result is returned

Special values

- The IEEE 754 standard defines a number of special values

	The special value	exponent	fraction
Regular normalized floating point numbers.	$1.f \times 2^{e}$	$e_{min} \leq e \leq e_{max}$	Any pattern of 1's and 0's
Denormalized Numbers … too small to represent as a normalized number.	$0.f \times 2^{e_{min}}$	All 0's $(e_{min} - 1)$	$f \neq 0$
	± 0	All 0's $(e_{min} - 1)$	$f = 0$
0 and ∞ have signs to work with limits	$\pm \infty$	All 1's $(e_{max} + 1)$	$f = 0$
Not a Number (undefined math such as 0/0).	NaN	All 1's $(e_{max} + 1)$	$f \neq 0$

- Typically, we do not test for these cases in code, but they do show up from time to time (especially NaN) so its good to be aware of them.

More about NaNs (Not a Number)

- Here are the cases where a NaN can be produced.

Operation	NaN produced by …
$+$	$\infty + (-\infty)$
$\times$	$0 \times \infty$
$/$	$0/0, \qquad \infty/\infty$
REM	$x\ REM\ 0, \qquad \infty REM\ y$
$\sqrt{}$	$\sqrt{x} \quad when \quad x < 0$

- There are two kinds of NaNs:
  - A quiet NaN … A NaN condition is identified but no further information is provided. The fraction bits are all zero other than the first one.
  - A signaling NaN … Additional implementation dependent information is encoded into the fraction bits.

Source: What every computer computer scientist should know about floating point arithmetic, David Goldberg, Computing Surveys, 1991 https://dl.acm.org/doi/pdf/10.1145/103162.103163

Writing IEEE 754 numbers in binary

- The number 42.0 written in binary

sign	exponent, w, 11 bits	fraction, p-1, 52 bits
0	1 0 0 0 0 0 0 0 1 0 0	0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Keeping track of all 64 locations and writing all those zeros is painful

IEEE name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary 64	double	64	11	53	-1022	1023

Writing IEEE 754 numbers in binary/hexadecimal

- The number 42.0 written in binary with the equivalent hexadecimal (base 16) form beneath.

sign exponent, w, 11 bits fraction, p-1, 52 bits

	0 1 0 0	0 0 0 0	0 1 0 0	0 1 0 1	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
0x	4	0	4	5	0	0	0	0	0	0	0	0	0	0	0	0

- It is dramatically easier to write things down in hexadecimal than binary.

- The following are notable examples of key “numbers” in hexadecimal.

-42	0xC045000000000000
Largest normal	0x7FEFFFFFFFFFFFFF
Smallest normal	0x0010000000000000
Largest subnormal	0x000FFFFFFFFFFFFF
Smallest subnormal	0X0000000000000001

+ zero	0x0000000000000000
-zero	0x8000000000000000
+infinity	0x7FF0000000000000
-infinity	0x8FF0000000000000
NaN	0X7FF-anything but all-zero

IEEE name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary 64	double	64	11	53	-1022	1023

Decimal, hexadecimal, and binary

0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

NaN: not a number

A normal is a number that can be written in a normalized floating point format

A subnormal is too small to be written as a normalized number … the exponent would need to be less than e_{min}.

Outline

- Numbers for humans.   Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
  - Algebraic Properties of Floating Point Arithmetic
- Wrap-up/Conclusion

Addition with floating point numbers

- Lets keep things simple … we will use F*(10, 3, -2, 2)

- Find the sum … $1.23 \times 10^1 + 3.11 \times 10^{-1}$
  - Align smaller number to the exponent of the larger number

    0.0311×10^1
  - Add the two aligned numbers …..

$$
\begin{array}{rl}
 & 1.23 \times 10^1 \\
+ & 0.0311 \times 10^1 \\
\hline
 & 1.2611 \times 10^1
\end{array}
$$

  - Round to nearest (the default rounding in IEEE 754).

$$1.26 \times 10^1$$

Adding numbers with greatly different magnitudes causes loss of precision
(you lose the low order bits from the exact result).

F*(Radix, Precision, emin, emax)

Exercise: summing numbers

- Compute the finite sum: $$sum = \sum_{i=1}^{N} \frac{1.0}{i}$$

- This is a simple loop. Run it forward (i=1,N) and backwards (i=N,1) for large N (`10000000`). Try both double and float

- Are the results different? Why?

Exercise: summing numbers

- Compute the finite sum: $$sum = \sum_{i=1}^{N} \frac{1.0}{i}$$
- This is a simple loop. Run it forward (i=1,N) and backwards (i=N,1) for large N (10000000). Try both double and float
- Are the results different? Why?

float or double

```
#include<stdio.h>
int main(){


    float sum=0.0;
    long N = 10000000;


    for(int i= 1;i<N;i++){
       sum += 1.0/(float)i;
    }
    printf(" sum forward  = %14.8f\n",sum);


    sum = 0.0;
    for(int i= N-1;i>=1;i--){
       sum += 1.0/(float)i;
    }
    printf(" sum backward = %14.8f\n",sum);
}
```

Exercise: summing numbers

- Compute the finite sum: $$sum = \sum_{i=1}^{N} \frac{1.0}{i}$$

- This is a simple loop. Run it forward (i=1,N) and backwards (i=N,1) for large N (`10000000`). Try both double and float

- Are the results different? Why?
  - In the forward direction, the terms in the sum get smaller as you progress. This leads to loss of precision as the smaller terms lates in the summation are added to the much larger accumulated partials sum.
  - In the backwards direction, the terms in the sum start small and grow … so reduced loss of precision adding small numbers to much larger numbers.
  - Using double precision eliminated this problem.

```c
#include<stdio.h>
int main(){


    float sum=0.0;
    long N = 10000000;


    for(int i= 1;i<N;i++){
       sum += 1.0/(float)i;
    }
    printf(" sum forward  = %14.8f\n",sum);


    sum = 0.0;
    for(int i= N-1;i>=1;i--){
       sum += 1.0/(float)i;
    }
    printf(" sum backward = %14.8f\n",sum);
}
```

float or double

	double	float
forward	16.695311265857270655	15.40368271
backward	16.695311265859963612	16.68603134

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set
With arbitrary precision, there is no loss of accuracy when adding real numbers	Adding numbers of different sizes can cause loss of low order bits.

Outline

- Numbers for humans.   Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
  - Algebraic Properties of Floating Point Arithmetic
- Wrap-up/Conclusion

What can go wrong with subtraction? Cancelation

- Consider two numbers …

3.141592653589793	16 digits of pi
3.141592653585682	12 digits of pi

Their difference (in real arithmetic) → 0.000000000004111 = 4.111×10^{-12}

Storage of numbers and difference with float → 0.000000e+00 **Complete loss of accuracy**

Storage of numbers and difference with double → 4.110933815582030e-12 **Partial loss of accuracy**

- The machine epsilon for a double is `2.22045e−16.` The above error is large compared to epsilon.

Subtracting two number of similar magnitude cancels high order bits.

Source: Ianna Osborne, CoDaS-HEP, July 19, 2023

Exercise: Implement a Series summation to find e^x

- A Taylor/Maclaurin series expansion for e^x

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

1. Compare to the exp(x) function in math.h for a range of x values **greater than zero**.
   - How do your results compare to the exp(x) library function?

2. Compute e^x for $x<0$. Consider small negative to large negative values.
   - Do you continue to match the exp(x) library function?

Exercise: Implement a Series summation to find e^x

Hints

Compare computation of e^x directly and as $e^{-x} = 1/ e^x$.

- A Taylor/Maclaurin series expansion for e^x

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

- The computation of x^n and n! are expensive but worse … they lead to large numbers that could overflow the storage format.
- A better approach is to use the relation:

$$\frac{x^n}{n!} = \frac{x}{n} \bullet \frac{x^{n-1}}{(n-1)!}$$

- Terminating the sum … obviously you don't want to go to infinity. How do you terminate the sum? A good approach is to end the sum when new terms do not significantly change the sum. Or think about what you did when computing the machine epsilon.

Solution: Implement a Series summation to find e^x

- A Taylor/Maclaurin series expansion for e^x

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

- The computation of x^n and n! are expensive but worse … they lead to large numbers that could overflow the storage format.
- A better approach is to use the relation:

$$\frac{x^n}{n!} = \frac{x}{n} \bullet \frac{x^{n-1}}{(n-1)!}$$

- Terminating the sum … obviously you don't want to go to infinity. How do you terminate the sum? A good approach is to end the sum when new terms do not significantly change the sum.

- For $x< 0$, compare computation of e^x directly and as $e^x = 1/ e^x$.

```
#define TYPE float
TYPE MyExp (TYPE x) {
    long counter = 0;
    TYPE delta = (TYPE)1.0;
    TYPE e_tothe_x = (TYPE)1.0;
    while((1.0 + delta) != 1.0) {
        counter++;
        delta *= x/counter;
        e_tothe_x +=delta;
    }
    return e_tothe_x;
}
```

When x>0 in series, no cancelation and MyExp matches exp from the standard math library (math.h)

x	exp(x) math.h	MyExp(x)
5	148.413	148.413
10	22026.5	22026.5
15	3.26902e+06	3.26902e+06
20	4.85165e+08	4.85165e+08

x	exp(x) math.h	MyExp(x)	1/MyExp(|x|)
-5	6.73795e-03	6.73714e-03	6.73795e-03
-10	4.53999e-05	-5.23423e-05	4.53999e-05
-15	3.05902e-07	-2.23869e-02	3.05902e-07
-20	2.06115e-09	-1.79703	2.06115e-09

When x<0 in series, MyExp does not match exp from math.h due to cancelation. Results become nonsensical for x= -10 and beyond.

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set
With arbitrary precision, there is no loss of accuracy when adding real numbers	Adding numbers of different sizes can cause loss of low order bits.
With arbitrary precision, there is no loss of accuracy when subtracting real numbers	Subtracting two numbers of similar size cancels higher order bits

Outline

- Numbers for humans.   Numbers for computers

- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard

- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
  - Algebraic Properties of Floating Point Arithmetic

- Wrap-up/Conclusion

IEEE 754 arithmetic and rounding

- The IEEE 754 standard requires that the result of basic arithmetic ops (+, -, *, /, FMA) be equal to the result from “infinitely precise arithmetic” rounded to the storage format (e.g., float or double).

- Consider the following problem … subtract two IEEE 754 32 bit numbers (F*(2,24,-126,127)):

$$
\begin{array}{rl}
 & (1.00000000000000000000000)_2 \cdot 2^{0} \\
- & (1.00000000000000000000001)_2 \cdot 2^{-25}
\end{array}
$$

- We normalize them to the same exponent and carry out the operation exactly

$$
\begin{array}{rl}
 & (1.00000000000000000000000 \qquad\qquad\qquad\qquad\qquad\qquad)_2 \cdot 2^{0} \\
- & (0.000000000000000000000000100000000000000000000001)_2 \cdot 2^{0} \\
\hline
= & (0.111111111111111111111111101111111111111111111110)_2 \cdot 2^{0}
\end{array}
$$

- Then normalize the result

$$(1.11111111111111111111111011111111111111111111111)_2 \cdot 2^{-1}$$

- Then round to nearest to fit into the destination format

$$(1.11111111111111111111111)_2 \cdot 2^{-1}$$

IEEE 754 arithmetic and rounding

- The IEEE 754 standard requires that the result of basic arithmetic ops (+, -, *, /, FMA) be equal to the result from “infinitely precise arithmetic” rounded to the storage format (e.g., float or double).

- Consider the following problem … subtract two IEEE 754 32 bit numbers (F*(2,24,-126,127)):

$$\begin{array}{rl} & (1.00000000000000000000000)_2 \cdot 2^{0} \\ - & (1.00000000000000000000001)_2 \cdot 2^{-25} \end{array}$$

- We normalize them to the same exponent and carry out the operation exactly

$$\begin{array}{rl} & (1.00000000000000000000000 \qquad\qquad\qquad\qquad\qquad\qquad\quad)_2 \cdot 2^{0} \\ - & (0.00000000000000000000000\,010000000000000000000001)_2 \cdot 2^{0} \\ \hline = & (0.11111111111111111111111\,101111111111111111111110)_2 \cdot 2^{0} \end{array}$$

The exact result doubled the number of bits in the fraction. Do we really need all those bits?

- Then normalize the result

$$(1.11111111111111111111111\,01111111111111111111111)_2 \cdot 2^{-1}$$

- Then round to nearest to fit into the destination format

$$(1.11111111111111111111111)_2 \cdot 2^{-1}$$

IEEE 754 arithmetic and rounding

Next 2 bits in the answer:
the guard and rounding bits

- Turns out you only need three extra bits … the Guard bit, the Rounding bit, and the Sticky Bit (GRS)

$$
\begin{array}{rll}
 & (1.00000000000000000000000\boxed{0}\boxed{0}\boxed{0})_2 & \cdot 2^0 \\
- & (0.00000000000000000000000\boxed{0}\boxed{1}\boxed{1})_2 & \cdot 2^0 \\
\hline
 & (0.11111111111111111111111\boxed{1}\boxed{0}\boxed{1})_2 & \cdot 2^0 \\
\text{Normalize} & (1.11111111111111111111111\boxed{0}\boxed{1})_2 & \cdot 2^{-1} \\
\text{Round to nearest} & (1.11111111111111111111111)_2 & \cdot 2^{-1}
\end{array}
$$

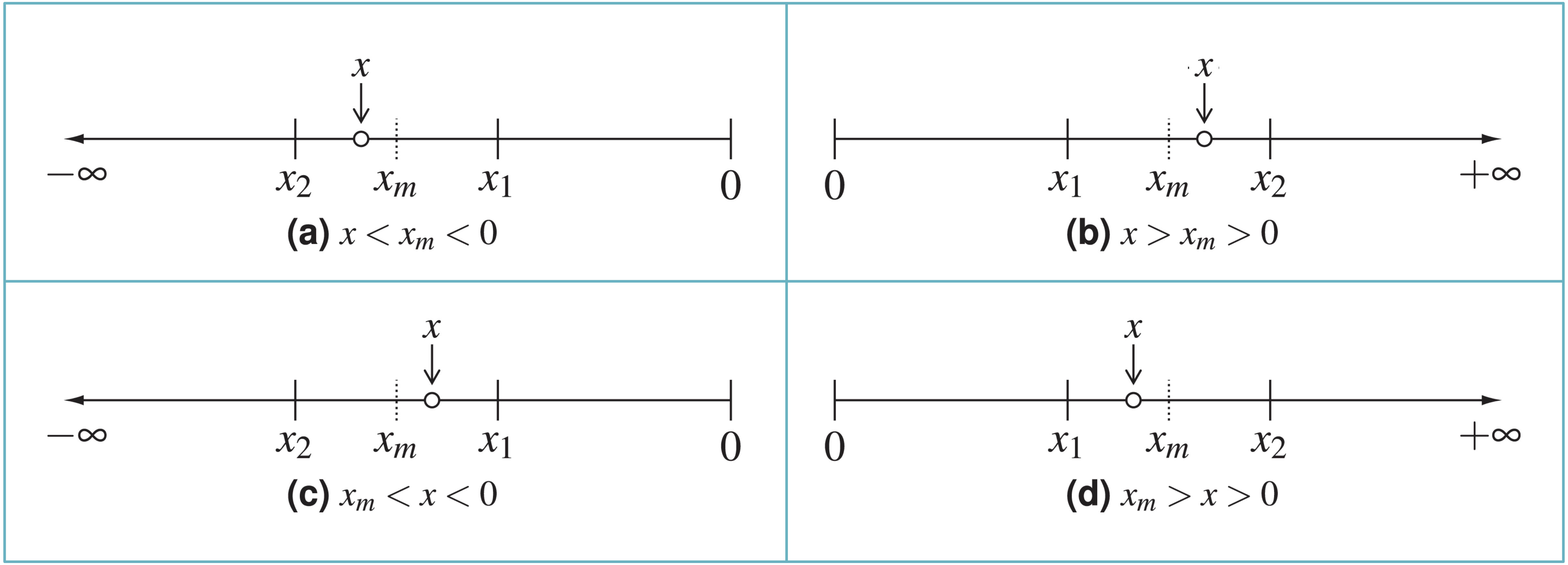
The Sticky bit: the logical or of all the other bits
(i.e., if any bit is “1” then the sticky bit is “1”)

- The Guard, Rounding, and Sticky bits are sufficient to support all the IEEE 754 rounding modes to yield the same result you’d get from an exact computation followed by rounding into the target format.

- Exactly rounded results are required for the basic arithmetic operations (including FMA) but also **square root**, **remainder**, and **conversion between Integer and Floating point numbers** … but not for conversion between decimal and binary floating point.

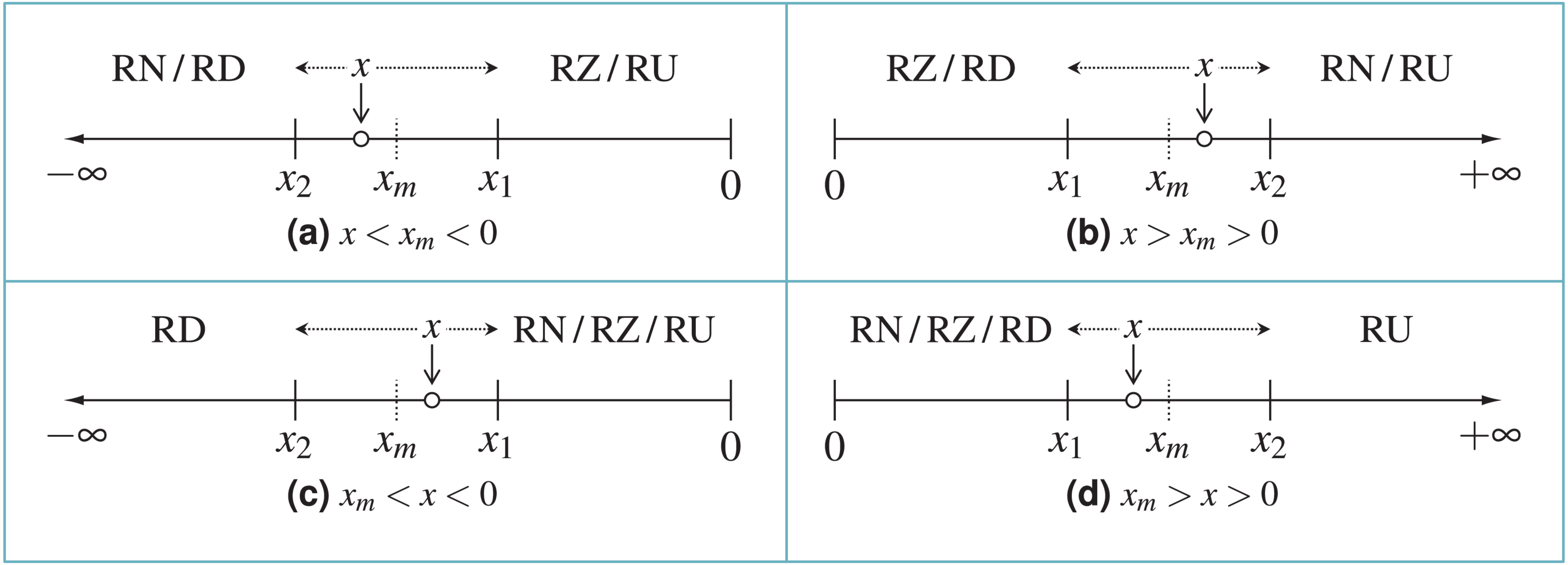
IEEE 754 Rounding Modes

Consider a real number x that falls between its two nearest floating point numbers (x_1 and x_2). At the midpoint between x_1 and x_2 is the real number x_m. We have four cases to consider when thinking about rounding.



IEEE 754 Rounding Modes

Consider a real number x that falls between its two nearest floating point numbers (x_1 and x_2). At the midpoint between x_1 and x_2 is the real number x_m. The horizontal dotted line shows the floating point numbers selected for the different rounding modes (RN, RD, RZ, RU) for position of x vs x_m and which side of zero x is on.



RN: Round to Nearest. RD: Round Downward RZ: Round towards zero RU: Round upward

Source: Numerical behavior of NVIDIA tensor cores, M. Fasi, N. J. Higham, M. Mikaitis, and S. Pransesh, *PeerJ Comp. Sci.* 7:e330, Feb 10, 2021, DOI: 10.7717/peerj-cs.330/fig-1

You must be careful how you manage rounding…

Vancouver stock exchange index undervalued by 50%

(Nov. 25, 1983)

See http://ta.twi.tudelft.nl/usersvuik/wi211/disasters.html

Index managed on an IBM/370. 3000 trades a day and for each trade, the index was truncated to the machine's REAL*4 format, loosing 0.5 ULP per transaction. After 22 months, the index had lost half its value.

ULP: Unit in the last place

Working with IEEE 754 rounding modes

Two versions of round to nearest…
- Nearest, on a tie, round to even
- Nearest, on a tie, away from zero

Default rounding mode

The 4 rounding modes in IEEE 754

```
rounding mode     C name
-------------------------------
to nearest        FE_TONEAREST
toward zero       FE_TOWARDZERO
to +infinity      FE_UPWARD
to -infinity      FE_DOWNWARD
```

Three directed roundings

C

```
#include <fenv.h>
//#pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = fegetround( );

// establish the desired rounding mode
fesetround(FE_TOWARDZERO);

// do whatever you need to do ...
// ... and restore the original mode afterwards
fesetround(originalRounding);
```

Clang and GCC compilers do not recognize the STDC pragma (even though they are technically required to).

Fortunately, rounding mode control seems to work without it.

If not, try the compiler flag –frounding-math

C++

```
#include <cfenv>
// #pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = std::fegetround( );

// establish the desired rounding mode
std::fesetround(FE_TOWARDZERO);

// do whatever you need to do ...
// ... and restore the original mode afterwards
std::fesetround(originalRounding);
```

Exercise: IEEE 754 rounding modes

- Explore how different rounding modes change the answers of programs you have on your system.
- What does it tell you if answers change as rounding modes change?

The 4 rounding modes in IEEE 754

rounding mode	C name
to nearest	FE_TONEAREST
toward zero	FE_TOWARDZERO
to +infinity	FE_UPWARD
to -infinity	FE_DOWNWARD

C

```
#include <fenv.h>
//#pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = fegetround( );

// establish the desired rounding mode
fesetround(FE_TOWARDZERO);

// do whatever you need to do ...
// ... and restore the original mode afterwards
fesetround(originalRounding);
```

Clang and GCC compilers do not recognize the STDC pragma (even though they are technically required to).

Fortunately, rounding mode control seems to work without it.

If not, try the compiler flag –frounding-math

C++

```
#include <cfenv>
// #pragma STDC FENV_ACCESS ON

// store the original rounding mode
const int originalRounding = std::fegetround( );

// establish the desired rounding mode
std::fesetround(FE_TOWARDZERO);

// do whatever you need to do ...
// ... and restore the original mode afterwards
std::fesetround(originalRounding);
```

Outline

- Numbers for humans.   Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
  - Algebraic Properties of Floating Point Arithmetic
- Wrap-up/Conclusion

Properties of Floating point arithmetic

- IEEE 754 defines the concept of an ***exactly rounded*** results of operation.

  - An exactly rounded result is equivalent to a computation carried out to infinite precision rounded to fit in the designated storage format (e.g., float or double).

- The standard requires a set of operations that must be exactly rounded
  - Add, subtract, multiply, divide, remainder
  - Square root, fused multiply-add, minimum, maximum
  - Comparisons and total ordering
  - Conversions between formats

- Exact rounding is recommended (but not required) for:
  - Exponentials
  - Logarithms
  - Reciprocal square root
  - Trigonometric functions

Floating point arithmetic is not associative or distributive

- **IEEE 754 guarantees that a single arithmetic operation produces a correctly rounded result … but that guarantee does not apply to multiple operations in sequence.**

- Floating point numbers are:
  - Commutative: A * B = B * A
  - NOT Associative: A * (C * B) ≠ (A * C) * B
  - NOT Distributive: A*(B+C) ≠ A*B + A*C

- All these computations where done in a C program using type float

```
a = 11111113;  b= -11111113;   c = 7.51111111f;
     (a + b) + c = 7.511111
    a + (b + c) = 8.000000

  Correct answer:  7.511111
```

```
a = 20000;  b= -6;   c = 6.0000003;
     (a*b + a*c) = 0.007812
      a*(b + c) = 0.009537

   Correct answer: 0.006000
```

- Python promotes floating point number to double, but even with the extra precision, you can run into trouble

Python

```
a = 1.e20;   b= -1.e20;   c=1.0
(a+b) + c = 1.0
a+(b+c)  = 0.0

Correct answer: 1.0
```

... But C using float gets this case right →

```
a = 1.e20f;   b= -1.e20f;   c=1.0f
(a + b) + c = 1.000000
a + (b + c) = 1.000000
```

Python 13.11.7   and Gnu gcc version 14.2.0

Exercise: Summation with floating point arithmetic

- We have provided a C program called summation.c
- In the program, we generate a sequence of floating point numbers (all greater than zero).
  - **Don't look at how we create that sequence** … treat the sequence generator as a black box (in other words, just work on the sequence, don't use knowledge of how it was generated).
- Write code to sum the sequence of numbers. You can compare your result to the estimate of the correct result provided by the sequence generator.
  - Only use float types (it's cheating to use double … at least to start with).
- Using what you know about floating point arithmetic, is there anything you can think of doing to improve the quality of your sum?

Summation program

```
#include "UtilityFunctions.h"  // FillSequence() comes from this module

#define N 100000               //length of sequence of numbers to work with
int main ()
{
   float seq[N];    //Sequence to sum
   float True_sum;  //The best estimate of the actual sum
   float sum = 0.0f;

   FillSequence(N, seq, &True_sum); // Fill seq with N values > 0


   for(int i=0; i<N; i++)sum += seq[i];

   printf(" Sum = %f, Estimated sum = %f\n",sum,True_sum);

}
```

```
> gcc summation.c UtilityFunctions.c
> ./a.out
> Sum = 2502476.500000, Estimated sum = 2502458.750000
```

This result is kind of awful

Summation program

Let's do the sum in parallel and see how the answer varies with the number of threads

```
#include <omp.h>
#include "UtilityFunctions.h"  // FillSequence() comes from this module

#define N 100000                //length of sequence of numbers to work with
int main ()
{
   float seq[N];     //Sequence to sum
   float True_sum;   //The best estimate of the actual sum
   float sum = 0.0f;

   FillSequence(N, seq, &True_sum); // Fill seq with N values > 0

   #pragma omp parallel for reduction(+:sum)
   for(int i=0; i<N; i++)sum += seq[i];

   printf(" Sum = %f, Estimated sum = %f\n",sum,True_sum);

}
```

Values with 1 to 8 threads

1	2502476.5
2	2502457.0
4	2502459.25
8	2502459.0

True value = 2502458.75

Sequence Generation

- I created a particularly awful sequence to sum

```
void FillSequence(int N, float *seq, float *True_sum)
{
   float shift_up   = 100.0f;
   float shift_down =     0.000001f;
   double up_sum   = 0.0d,  down_sum = 0.0d;

   for(int i=0;i<N; i++){
     if(i%2==0){
         seq[i]   = (float) frandom() * shift_up;
         up_sum   += (double) seq[i];
     }
     else {
         seq[i]   = (float) frandom() * shift_down;
         down_sum += (double) seq[i];
     }
   }
   *True_sum = (float)(up_sum + down_sum);
}
```

Alternating big and small numbers to maximize opportunities for loss of precision when summing the numbers.

Notice how I estimate the “true” value by summing big numbers and little numbers separately before combining them.

Summation program

Sort from small to large before summing the sequence

```
#include "UtilityFunctions.h"  // FillSequence() comes from this module

#define N 100000              //length of sequence of numbers to work with
int main ()
{
   float seq[N];    //Sequence to sum
   float True_sum;  //The best estimate of the actual sum
   float sum = 0.0f;

   FillSequence(N, seq, &True_sum); // Fill seq with N values > 0

   qsort(seq, N, sizeof(int), compare); // Sort from smallest to largest
   for(int i=0; i<N; i++)sum += seq[i];

   printf(" Sum = %f, Estimated sum = %f\n",sum,True_sum);

}
```

The sorted sequence decreases loss of precision since magnitudes of numbers are closer together in a sorted sequence

```
> gcc summation.c UtilityFunctions.c
> ./a.out
> Sum = 2502455.500000, Estimated sum = 2502458.750000
```

From 2502476.5 to 2502455.5 That's a big improvement

Floating Point Numbers are not Real: Lessons Learned

Real Numbers	Floating Point numbers
Any number can be represented … real numbers are a closed set	Not all numbers can be represented … operations can produce numbers that cannot be represented … that is, floating point numbers are NOT a closed set
With arbitrary precision, there is no loss of accuracy when adding real numbers	Adding numbers of different sizes can cause loss of low order bits.
With arbitrary precision, there is no loss of accuracy when subtracting real numbers	Subtracting two numbers of similar size cancels higher order bits
Basic arithmetic operations over Real numbers are commutative, distributive and associative.	Basic operations over floating point numbers are commutative, but NOT associative or distributive.

Outline

- Numbers for humans.   Numbers for computers
- Finite precision, floating point numbers
  - General case
  - IEEE 754 floating point standard
- Working with IEEE 754 floating point arithmetic
  - Addition
  - Subtraction
  - Rounding
  - Algebraic Properties of Floating Point Arithmetic
- Wrap-up/Conclusion

The Problem

- How often do we have "working" software that is "silently" producing inaccurate results?
  - We don't know … nobody is keeping count.

- But we do know this is an issue for 2 reasons:
  (see Kahan's desperately needed Remedies…)
  - Numerically Naïve (and unchallenged) formulas in text books (e.g. solving quadratic equations).
  - Errors found after years of use (Rank estimate in use since 1965 and in LINPACK, LAPACK, and MATLAB (Zlatko Drmac and Zvonimir Bujanovic 2008, 2010). Errors in LAPACK's _LARFP found in 2010.)

… and then every now and then, a disaster reminds us that floating point arithmetic is not Real

Here is a famous example …

Sleipner Oil Rig Collapse (8/23/91). Loss: **$700 million.**

See http://www.ima.umn.edu/~arnold/disasters/sleipner.html

Inaccurate linear elastic model used with NASTRAN underestimated shear stresses by 47% resulted in concrete walls that were too thin.

We can’t trust FLOPS … let’s give up and return to slide rules

Public Domain, https://commons.wikimedia.org/w/index.php?curid=17480483

(an elegant weapon for a more civilized age)

Image source: Presbrey advertising agency for International Business Machines, 1951

Sleipner Oil Rig Collapse: The slide-rule wins!!!

It was recognized that finding and correcting the flaws in the computer analysis and design routines was going to be a major task. Further, with the income from the lost production of the gas field being valued at perhaps $1 million a day, it was evident that a replacement structure needed to be designed and built in the shortest possible time.

A decision was made to proceed with the design using the pre-computer, slide-rule era techniques that had been used for the first Condeep platforms designed 20 years previously. By the time the new computer results were available, all of the structure had been designed by hand and most of the structure had been built. On April 29, 1993 the new concrete gravity base structure was successfully mated with the deck and Sleipner was ready to be towed to sea (See photo on title page).

The failure of the Sleipner base structure, which involved a total economic loss of about $700 million, was probably the most expensive shear failure ever. The accident, the subsequent investigations, and the successful redesign offer several lessons for structural engineers. No matter how complex the structure or how sophisticated the computer software it is always possible to obtain most of the important design parameters by relatively simple hand calculations. Such calculations should always be done, both to check the computer results and to improve the engineers' understanding of the critical design issues. In this respect it is important to note that the design errors in Sleipner were not detected by the extensive and very formal quality assurance procedures that were employed.

How should we respond?

- Programmers should conduct mathematically rigorous analysis of their floating point intensive applications to validate their correctness.
- But this won't happen … training of modern programmers all but ignores numerical analysis. The following tricks* help and are better than nothing …

1. Repeat the computation with arithmetic of increasing precision, increasing it until a desired number of digits in the results agree.
2. Repeat the computation in arithmetic of the same precision but rounded differently, say *Down* then *Up* and perhaps *Towards Zero,* then compare results (this wont work with libraries that require a particular rounding mode).
3. Repeat computation a few times in arithmetic of the same precision but with slightly different input data, and see how widely results vary.

These are useful techniques, but they don't go far enough. How can the discerning skeptic confidently use FLOPs?

*Source: W. Kahan: How futile are mindless Assessments of Roundoff in floating-point computation?

Conclusion

- Floating point arithmetic usually works and you can "almost always" be comfortable using it.
- We covered the most famous issues with floating point arithmetic, but we largely skipped numerical analysis. Floating point arithmetic is mathematically rigorous. You can prove theorems and develop formal error bounds.
- Unfortunately, almost nobody learns numerical analysis these days … so be careful.
  - Modify rounding modes as an easy way to see if round-off errors are a problem.
  - Recognize that unless you impose an order of association, every order is equally valid. If your answers change as the number of threads changes, that is valuable information suggesting in ill-conditioned problem.
  - Anyone who suggests the need for bitwise identical results from a parallel code should be harshly criticized/punished.

My favorite picture of my wife

Kayaker: Pat Welle at Cascade head. Photo by T. Mattson.

References

- What every computer computer scientist should know about floating point arithmetic, David Goldberg, Computing Surveys, 1991.
  - https://dl.acm.org/doi/pdf/10.1145/103162.103163
- W. Kahan: How futile are mindless Assessments of Roundoff in floating-point computation?
  - https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf
- History of IEEE-754: an interview with William Kahan
  - https://people.eecs.berkeley.edu/~wkahan/ieee754status/754story.html

git clone https://github.com/tgmattso/CompSciForPhys.git