Floating Point Numbers Aren't Real.
And your random numbers aren't random.

What EVERY computational scientist needs to know about numbers on computers

Tim Mattson, University of Bristol

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

We work with lots of different numbers on a computer.

But do you ever wonder ….
Should we trust computer arithmetic?

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

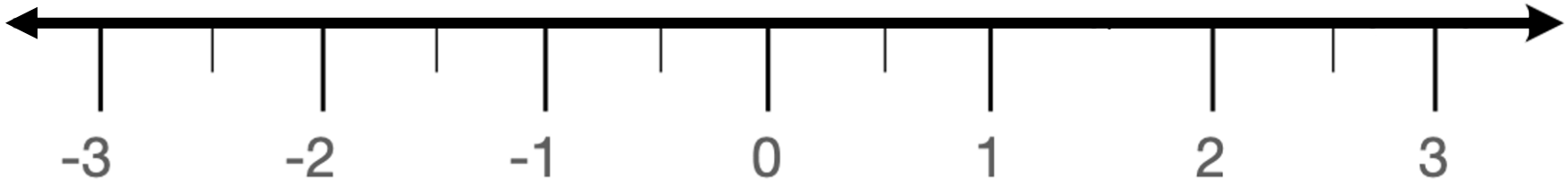
Linear elastic model using NASTRAN underestimated shear stresses by 47% resulted in concrete walls that were too thin.

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 and the IEEE 754 Standard
- The use and abuse of Random Numbers
- Wrap-up/Conclusion

Numbers for Humans

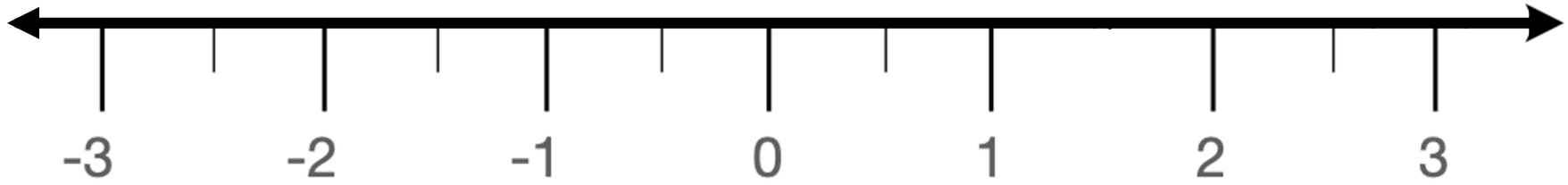


$\mathbb{R}$

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

Numbers for Humans

Arithmetic over Real Numbers



$\mathbb{R}$

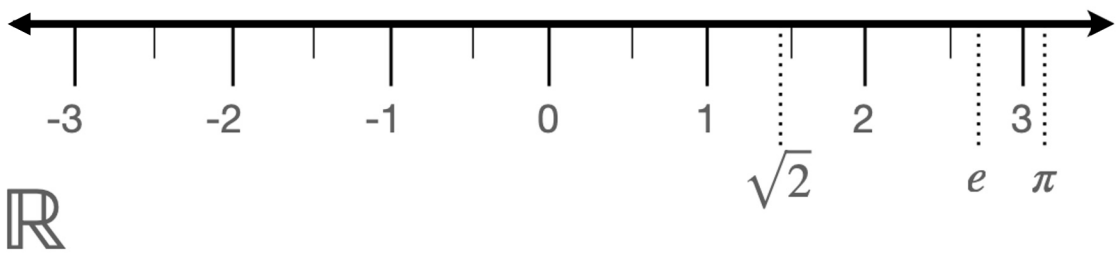
For the arithmetic operators, real numbers define a **closed set** … for well defined operations and any input real numbers, the arithmetic operation returns a real number.

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

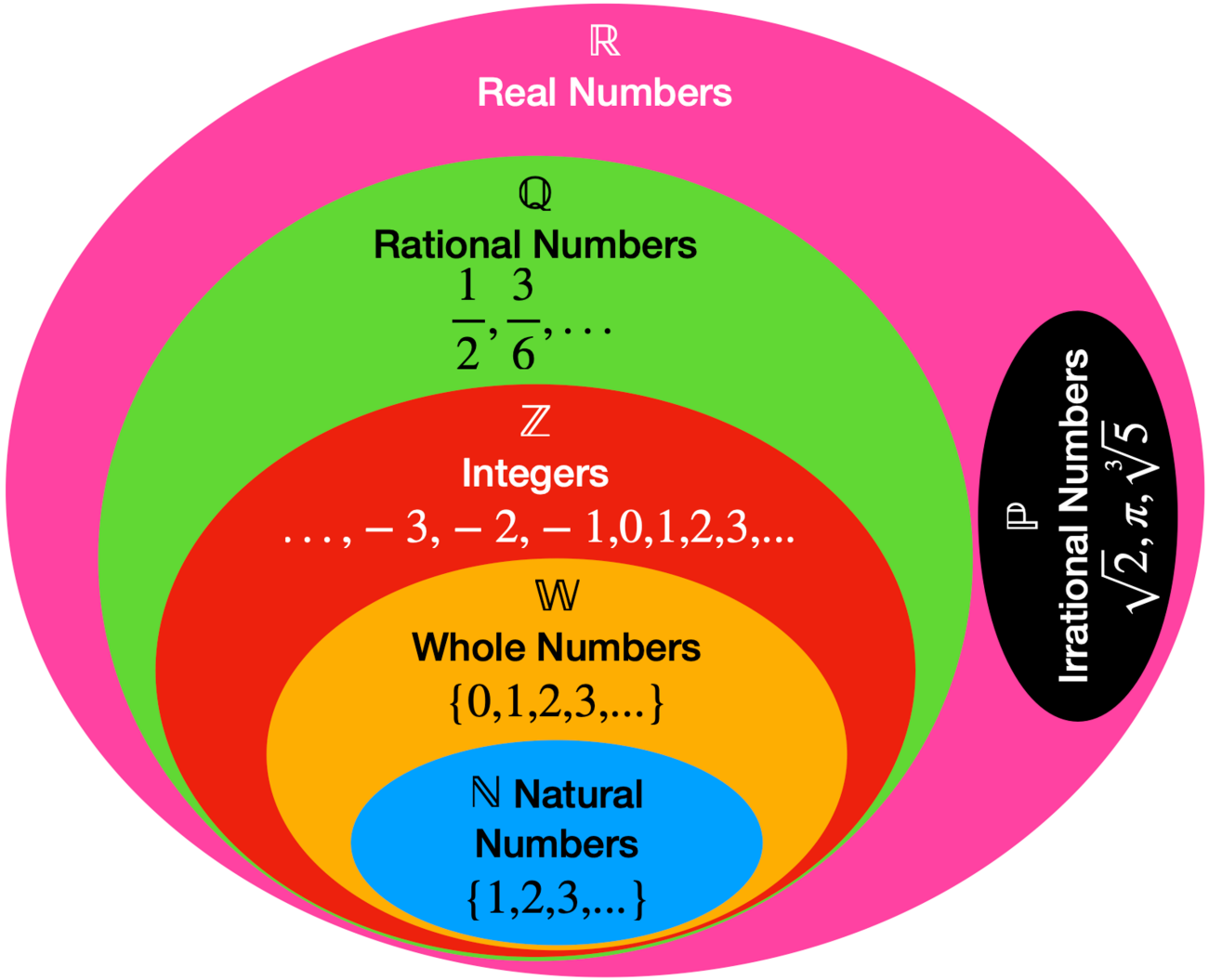
People use many different kinds of numbers



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

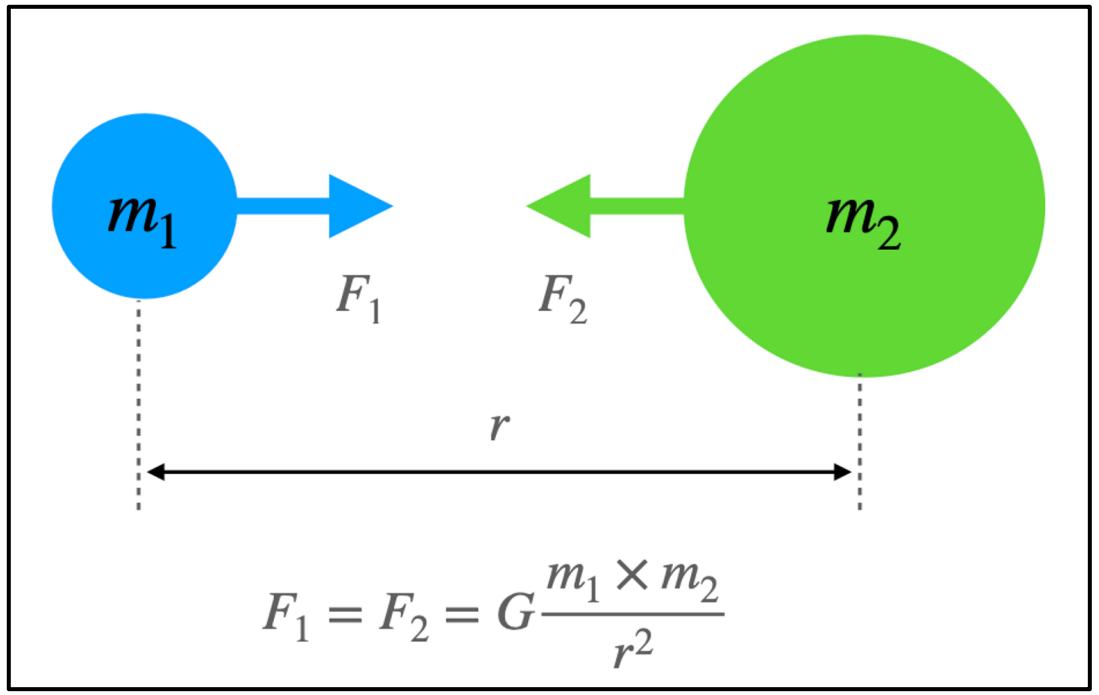
- **Integers**: equally spaced numbers without a fractional part
- **Rational numbers**: Ratios of integers
- **Whole numbers**: Positive integers and zero
- **Natural numbers**: Positive integers and not zero
- **Irrational numbers**: numbers that cannot be represented as a ratio of integers

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*

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

radix

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

Numbers for Humans

How do we represent Real Numbers?

significand *exponent*

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

radix

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

We can generalize the above to any real number as …

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

… where $\boldsymbol{b}$ is the **radix**

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

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

Numbers for Humans

How do we represent Real Numbers?

significand *exponent*

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

radix

What about numbers for computers?

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

We can generalize the above to any real number as …

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

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

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

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.

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

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

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
$\frac{1}{2^N}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{254}$	$\frac{1}{512}$	$\frac{1}{1024}$	$\frac{1}{2048}$	$\frac{1}{4096}$	$\frac{1}{8196}$	$\frac{1}{16384}$	$\frac{1}{32768}$	$\frac{1}{65536}$	$\frac{1}{131072}$

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)	$0.01_{10} \approx 0.0$ 00001010001_2

Who cares?
Does this really matter?

- Continuing to 32 bits we get 0.00000010100011110101 ... 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
$\frac{1}{2^N}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{254}$	$\frac{1}{512}$	$\frac{1}{1024}$	$\frac{1}{2048}$	$\frac{1}{4096}$	$\frac{1}{8196}$	$\frac{1}{16384}$	$\frac{1}{32768}$	$\frac{1}{65536}$	$\frac{1}{131072}$

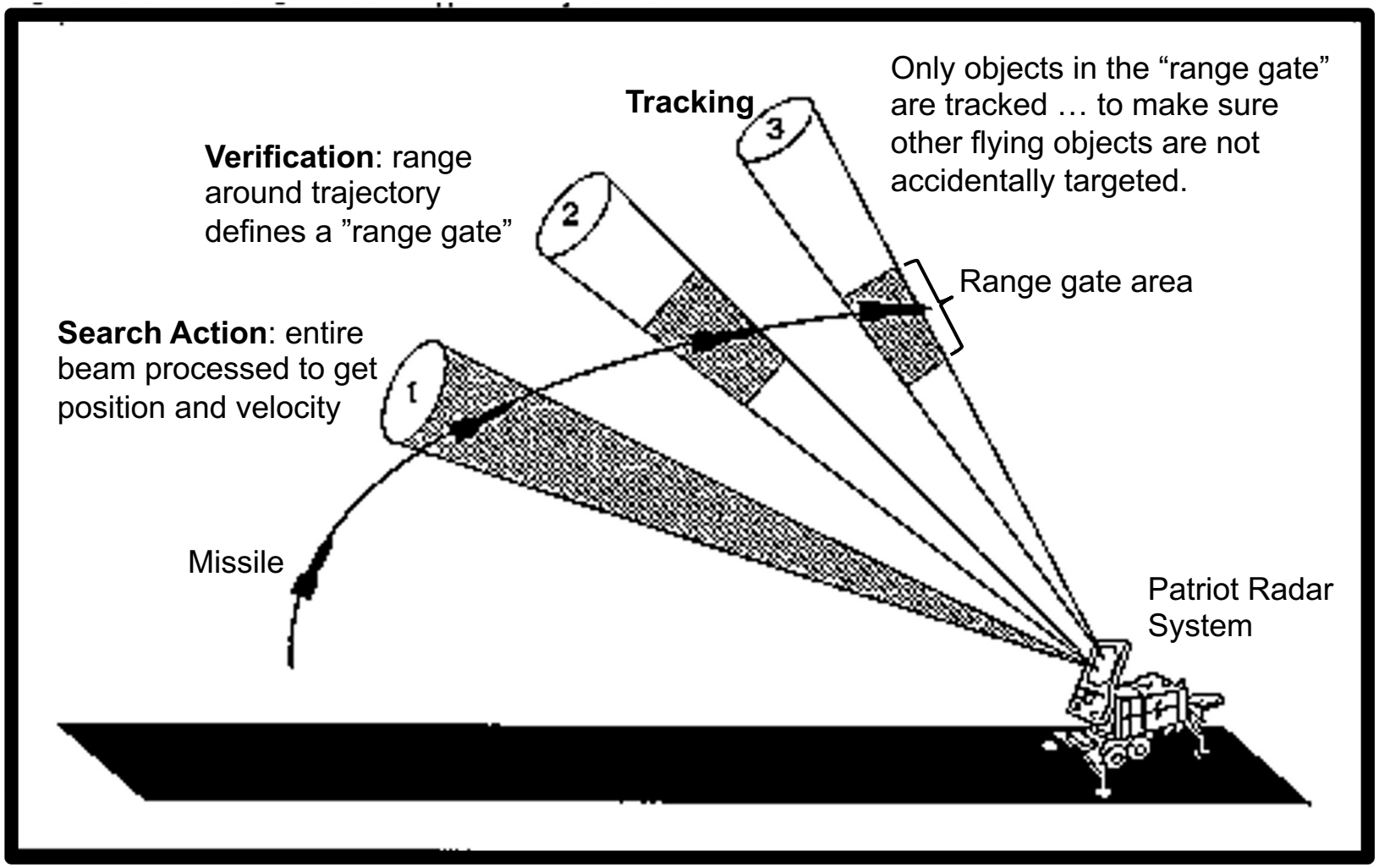
Patriot Missile system

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

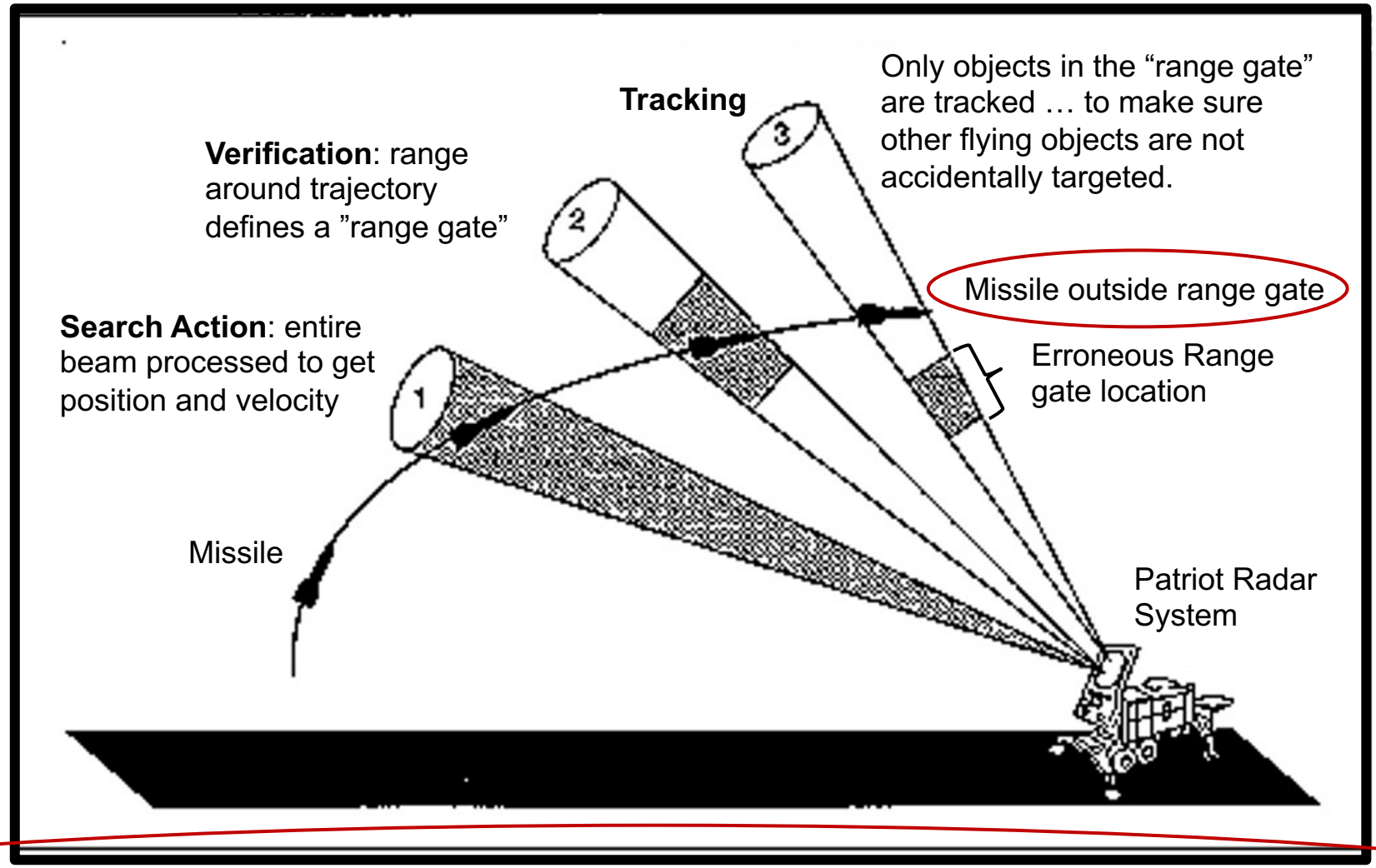
Incoming object detected as an enemy missile due to properties of the trajectory. Velocity and position from Radar fixes trajectory



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



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
  - ~~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.

```
> ./a.out
 dt = 0.007812500000000000.  // dt=1.0/128.0 … one over a power of two
 sum = 39062.500000000000000000, dt*Count=39062.500000000000000000
```

- 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

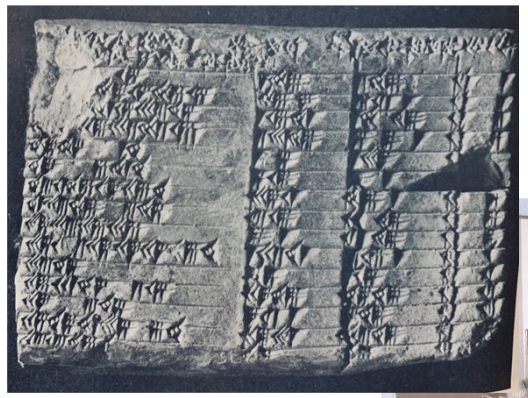
- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers and the IEEE 754 Standard
- The use and abuse of Random Numbers
- Wrap-up/Conclusion

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

- 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]

Zeus Z3

DEC PDP-11

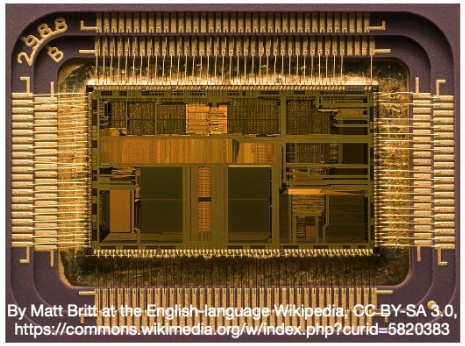
1970's boom of minicomputers

IEEE p745

IEEE 754-1985

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.



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

"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."

1750 B.C. | 1941 | 1960's a mainframe era | 1964 IBM 360, CDC 6600, DEC 10 | 1976 1977 | 1985 | 1989 | IEEE 754-1987 | IEEE 754-2008 | IEEE 754-2019 | you are here | IEEE 754-2029

- 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

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

binary and decimal floating-point arithmetic

subjected to review at least every 10 years

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

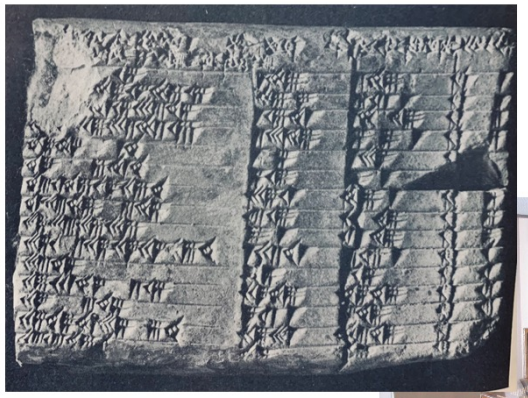
PRINCETON UNIVERSITY

Summarizing the key points …

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

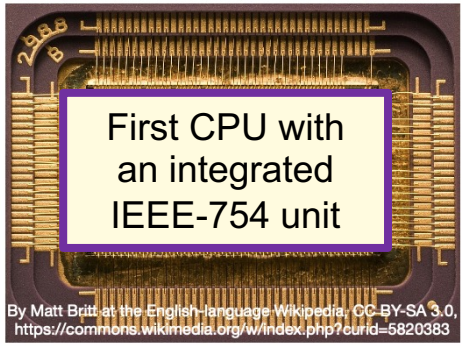


Babylonians worked with floating-point sexagesimal numbers

- 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]

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.



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

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

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

First CPU with an integrated IEEE-754 unit

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

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

1941 Zeus Z3

1960's inframe era

DEC

1970's boom of minico…

1976 1977 CDC 6600

1985

1989

you are here

IEEE 754-1987

IEEE 754-2008 IEEE 754-2019 IEEE 754-2029

The era of floating point chaos

…anufacturer 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

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

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

binary and decimal floating-point arithmetic

subjected to review at least every 10 years

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 $F(b, p, e_{min}, e_{max})$

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 $F^*(b, p, e_{min}, e_{max})$

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

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

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

$$d_0 \neq 0$$

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

- $x = 0 \ \ and \ x < b^{e_{min}}$ 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 $1 - e_{max}$)

PRINCETON UNIVERSITY

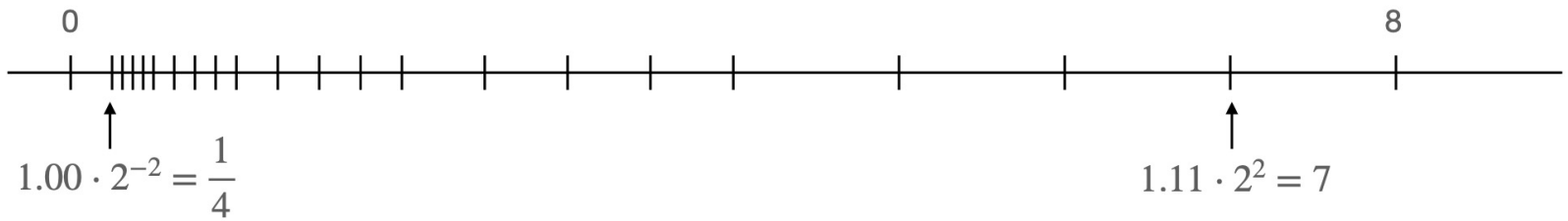
iris hep

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

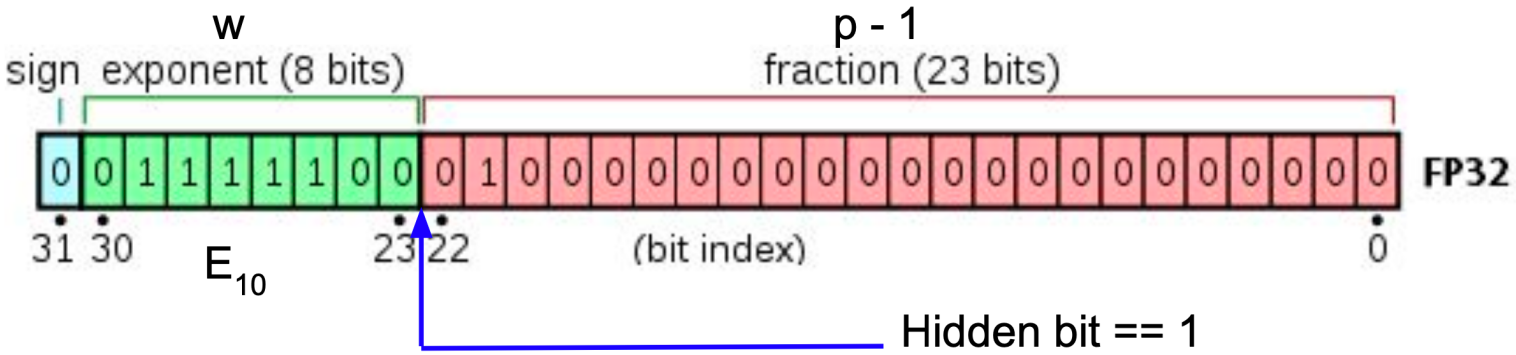


F*(Radix, Precision, emin, emax)

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

… More than you ever wanted to know about the IEEE 754 standard for floating point arithmetic

IEEE 754 Floating Point Numbers

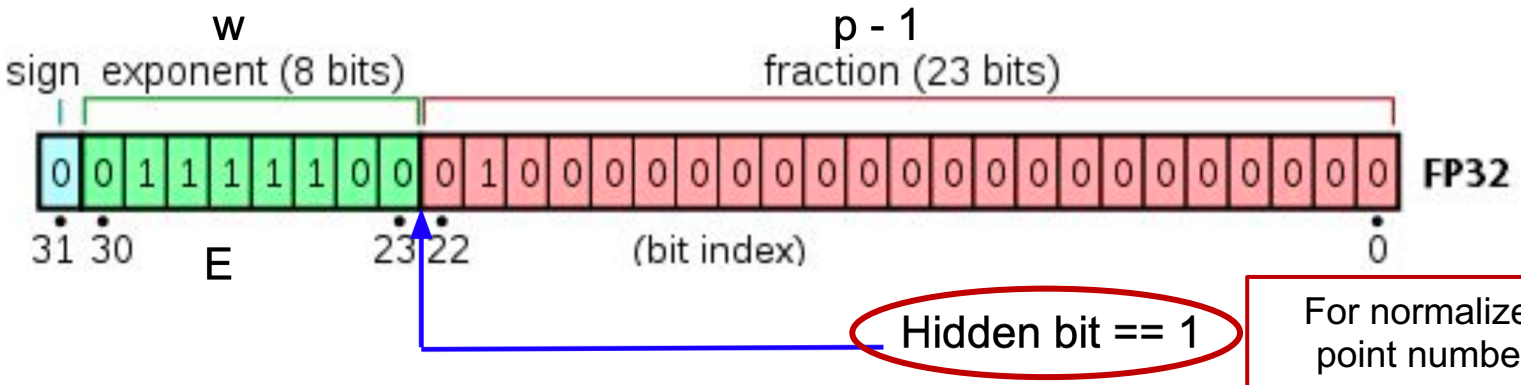


IEEE Name	Precision	N bits	Exponent w	Fraction p	e_{min}	e_{max}
Binary32	Single	32	8	24	-126	+127
Binary64	Double	64	11	53	-1022	+1023
Binary128	Quad	128	15	113	-16382	+16383

- 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

Special values

- The IEEE 754 standard defines a number of special values

The special value	exponent	fraction
$1.f \times 2^e$	$e_{min} \leq e \leq e_{max}$	Any pattern of 1's and 0's
$0.f \times 2^{e_{min}}$	All 0's $(e_{min} - 1)$	$f \neq 0$
± 0	All 0's $(e_{min} - 1)$	$f = 0$
$\pm \infty$	All 1's $(e_{max} + 1)$	$f = 0$
NaN	All 1's $(e_{max} + 1)$	$f \neq 0$

Regular normalized floating point numbers.

Denormalized Numbers … too small to represent as a normalized number.

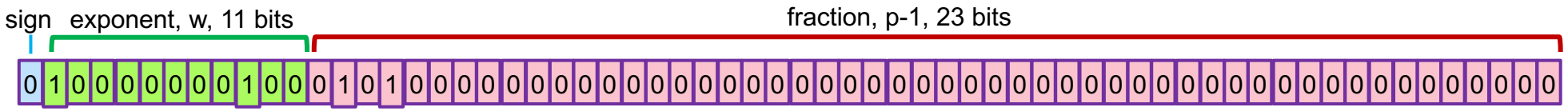
0 and ∞ have signs to work with limits

Not a Number (undefined math such as 0/0).

- 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.

Writing IEEE 754 numbers in binary

- The number 42.0 written in binary

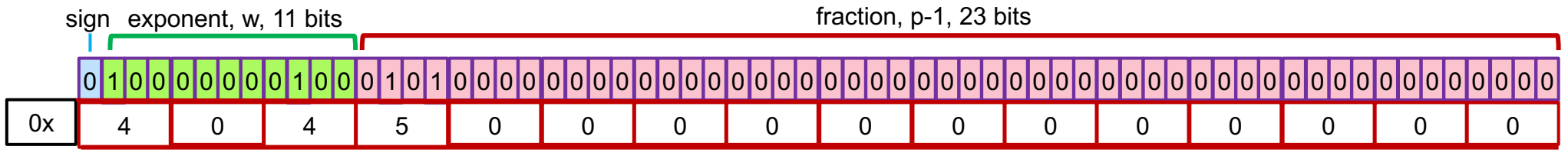


Keeping track of all 64 locations and
writing all those zeros is painful

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.



- 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	0x7FEFFFFFFFFFFF
Smallest normal	0x0010000000000000
Largest subnormal	0x000FFFFFFFFFFF
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}.

What could possibly go wrong with addition?

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 …..

$$\begin{array}{rrrrrrr} & 1 & . & 2 & 3 & & \times 10^{1} \\ + & 0 & . & 0 & 3 & 1 & 1 \times 10^{1} \\ \hline & 1 & . & 2 & 6 & 1 & 1 \times 10^{1} \end{array}$$

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

$$1 \; . \; 2 \; 6 \times 10^{1}$$

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?

```
#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

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.

```
#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.

Subtraction is basically addition where the signs on the operands are different.

Is there anything special with subtraction that we need to worry about?

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 x 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

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

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

- A Taylor/Maclaurin series expansion for e^x

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

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

- 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

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

- 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.

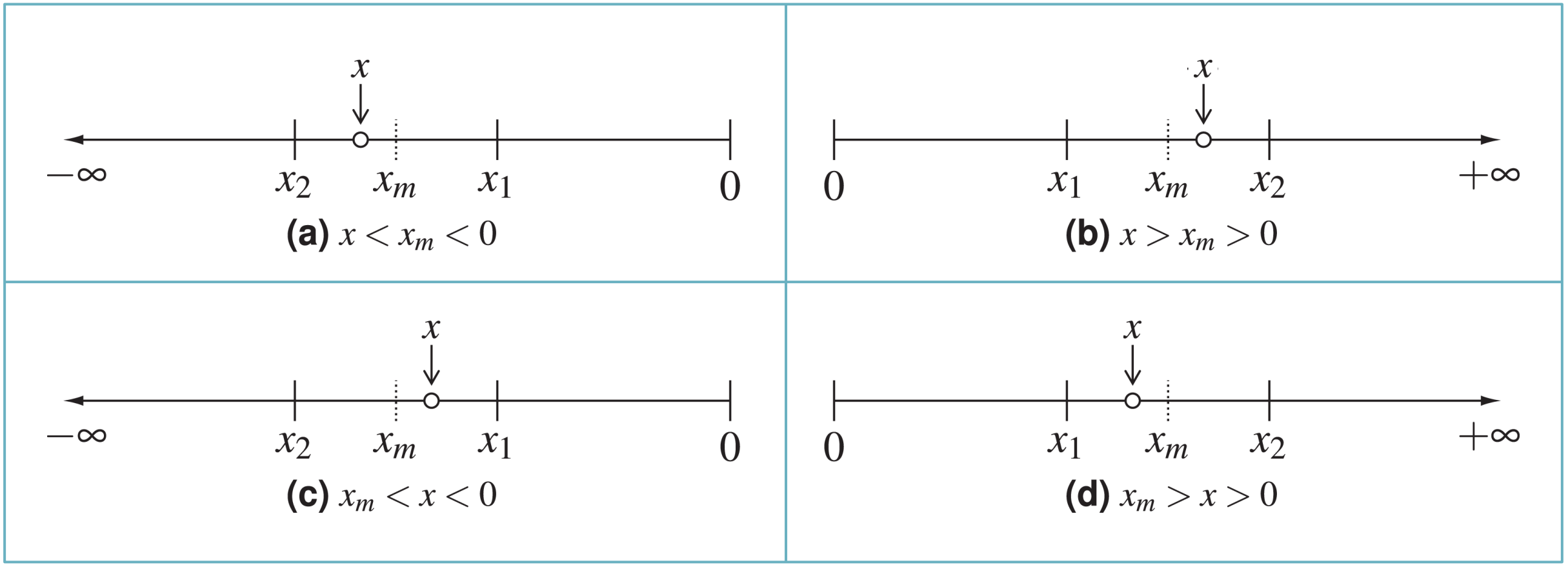
Floating Point Numbers are not Real: Lessons Learned

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

If you have a finite number of bits, then you need to round results to fit the storage format.

IEEE 754 Rounding Modes

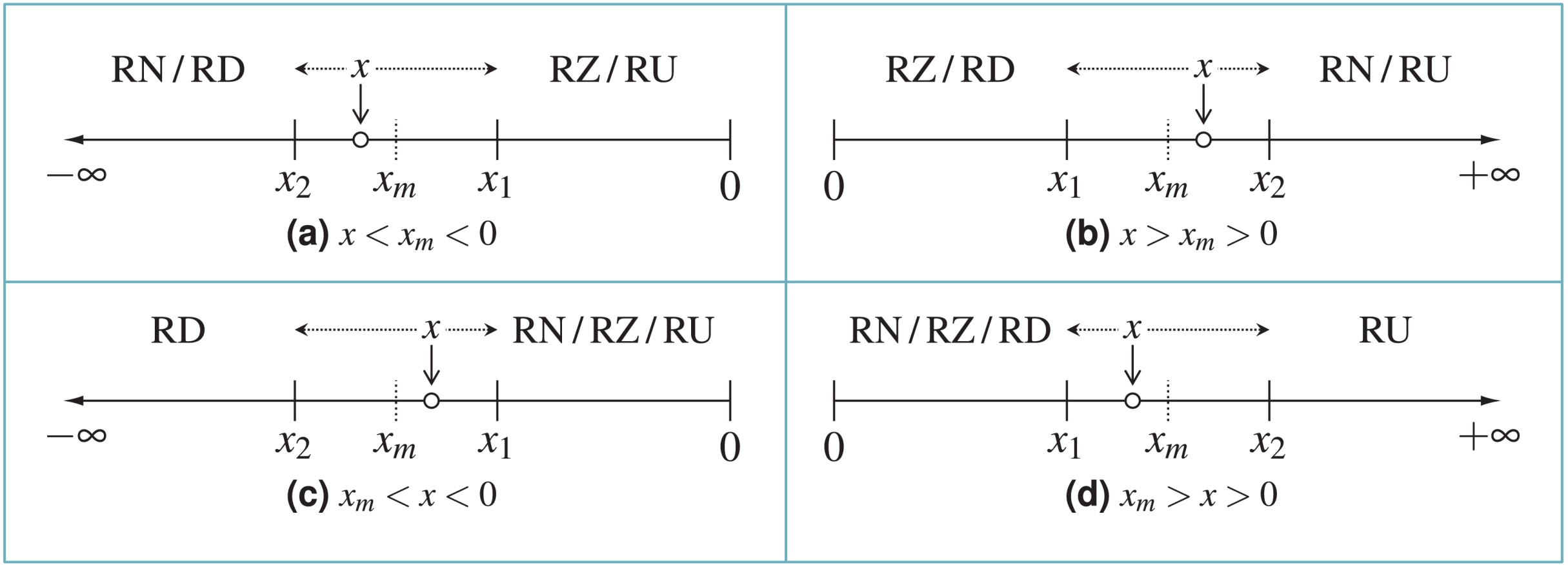
Consider a real number x that falls between its two nearest floating point numbers (x_1 and x_2). At the midpoint between x_1 and x_2 is the real number x_m. We have four cases to consider when thinking about rounding.



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

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

Algebra … what happens when we put sequences of arithmetic operations together?

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.

This is more detail about computer arithmetic than most people ever want to know … lets wrap-up and move on

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

(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.

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.

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?

Outline

- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers and the IEEE 754 Standard
- Responding to “issues” in floating point arithmetic
- The use and abuse of Random Numbers
- Wrap-up/Conclusion

Floating point numbers are very important … but there is another class of important numbers we work with.

Consider a sequence of numbers that fairly* sample a range of values and are unpredictable*

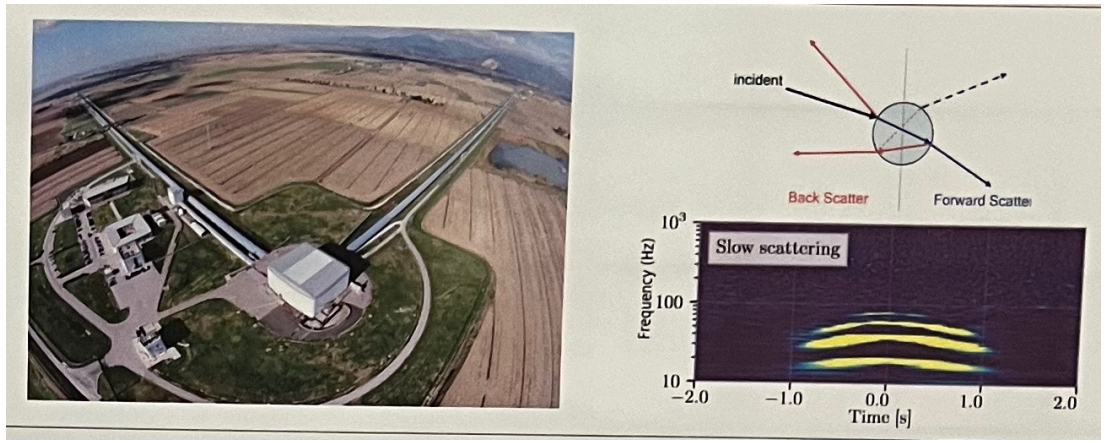
We call these random numbers

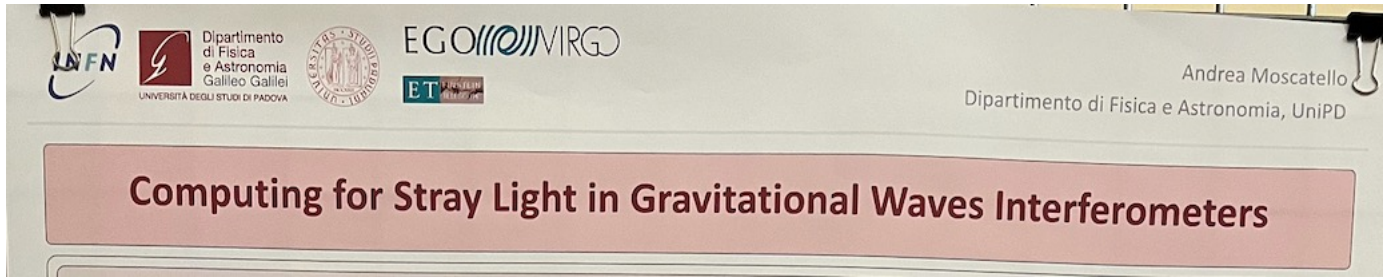
*fairness: all values in the range are equally likely to be selected. There are no favorites.
*unpredictability: For any number in the sequence, knowing all previous numbers does not help you predict the next number.

Solving problems with random samples

- For a large and important class of algorithms, we sample a problem domain and then use a statistical analysis over those samples to generate an answer. These are often called Monte Carlo algorithms.

- Algorithms based on random sampling are very common … for example, in high energy physics …
  - Monte Carlo physics generators (e.g. Pythia)
  - Monte Carlo detector simulation (e.g. Geant4)
  - Monte Carlo digitization (e.g. electronics simulation)
  - Statistical analysis (e.g. significance tests, importance sampling)





Example: Monte Carlo Integration

- A simple and direct method to approximate definite integrals
- The definite integral for the integrand $f(\vec{x})$ over a d-dimensional domain $\vec{x}$ is:

$$I[f] = \int_0^1 f(\vec{x}) d\vec{x}$$

- The limits of the integral are normalized over a d-dimensional cube $[0,1]^d$
- Randomly sample points within $[0,1]^d$ over a uniform distribution to create a sequence $\{\vec{x}_i\}$.
- The empirical approximation to the definite integral is $I_N[f]$.

$$I_N[f] = \frac{1}{N}\sum_{i=1}^{N} f(\vec{x}_i) \qquad \lim_{N\to\infty} I_N[f] \to I[f]$$

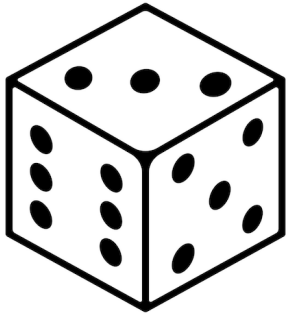
- The rate of convergence … independent of the dimension, d … is $O(N^{-1/2})$
- This is a slow rate of convergence, but it is independent of the dimension of the integral. For integrals solved over grids over $[0,1]^d$ the rate of convergence is $O(N^{-k/d})$ where k is the order of the numerical quadrature method. Also, defining a grid over $[0,1]^d$ for large d results in a prohibitively large number of points to sample.

Monte Carlo integration is robust, easy to implement, and for higher dimension problems (any time k/d < ½) the rate of convergence (while still slow) beats traditional numerical quadrature methods.

Choosing the random samples

- For Monte Carlo algorithms to work, the random samples must be:
  - Distributed according to the statistics required of the problem … that is, uniformly distributed or as samples of a predefined distribution (e.g., Gaussian, Poison, etc.).
  - Each Sample must be unpredictable given knowledge of other samples.
- A sequence of such numbers are called *Random Numbers.*
- We can generate a sequence of Random numbers from natural processes (e.g. white noise from a thermocouple), but not by any algorithm running on a deterministic machine (i.e., a computer).

```
// function returns a
// random number

int random()
{
   return 3;
}
```



The best we can do on a computer is produce numbers that appear to be random … that lack correlations between numbers or other features in the sequence that make the numbers predictable. We call these **Pseudo-Random numbers**.

Monte Carlo Methods require high quality pseudo-random numbers

Pseudo-Random Numbers

- High Quality Pseudo-Random numbers are indistinguishable from true Random numbers.

- They are generated by deterministic algorithms which means they can generate the same sequence between runs of a program (critical for validation purposes) … Reproducibility is your friend!!!!

- Pseudo-Random numbers, however, present their own challenges.
  - They aren't truly Random … the key is to use formal testing to show they are random enough.

  - It is depressingly easy to generate bad sequences of pseudo-random numbers and never know that your scientific results are garbage.

  - Its easy to write Pseudo-Random number generators but extremely difficult to write ones that are dependably random enough in all situations. Leave creating such generators to the pros … use libraries.

How to create sequences of Pseudo-Random Numbers

- We call the software that generates our pseudo-random numbers a random number generator or RNG

- There are at least two parts to an RNG … the algorithm and the parameters.

- Some common algorithms (we'll talk about parameters later)
  - Linear Congruential Generator (LCG)
  - Lagged Fibonacci Generator
  - Mersenne Twister
  - XORshift generator
  - Wichmann-Hill generator

- There is no single "best" generator … the key is to pick the generator best suited to your needs.
  - LCG is easy to implement and has decent quality if you get the parameters right.
  - Wichmann-Hill is a family of independent generators … quite handy for parallel applications
  - XORshift is very efficient (3 shift and 3 XOR operations)

Random Numbers: Linear Congruential Generator (LCG)

- LCG: Easy to write, cheap to compute, portable, OK quality

```
random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
random_last = random_next;
```

- If you pick the multiplier and addend correctly, LCG has a period of PMOD.
- Picking good LCG parameters is complicated, so look it up (Numerical Recipes is a good source). I often use the following:
  - MULTIPLIER = 1366
  - ADDEND = 150889
  - PMOD = 714025

If the ADDEND is zero, then we have a Multiplicative Linear Congruential Generator. (MLCG)

If you are careful in selecting the MULTIPLIER and PMOD, MLCG can be quite good.

LCG code

```
static long MULTIPLIER  = 1366;
static long ADDEND      = 150889;
static long PMOD        = 714025;
long random_last = 1597;
double drandom ()
{
   long random_next;

   random_next = (MULTIPLIER  * random_last + ADDEND)% PMOD;
   random_last = random_next;

   return  ((double)random_next/(double)PMOD);
}
```

Seed the pseudo random sequence by setting random_last

I often just pick a prime number that is less than PMOD.

Be careful with your random number generators

Famous PseudoRandom Generators: RANDU

- RANDU was a standard RNG from IBM. It was used heavily on their systems in the 1960s and 1970s.

- RANDU is a **Multiplicative Linear congruential** generator with multiplier, M, equal to 65539 and the modulus, mod, equal to 2^{31}. The seed (X_0) must be odd.

$$X_{n+1} = (M \cdot X_n)\%mod$$

- The following is Python code for RANDU

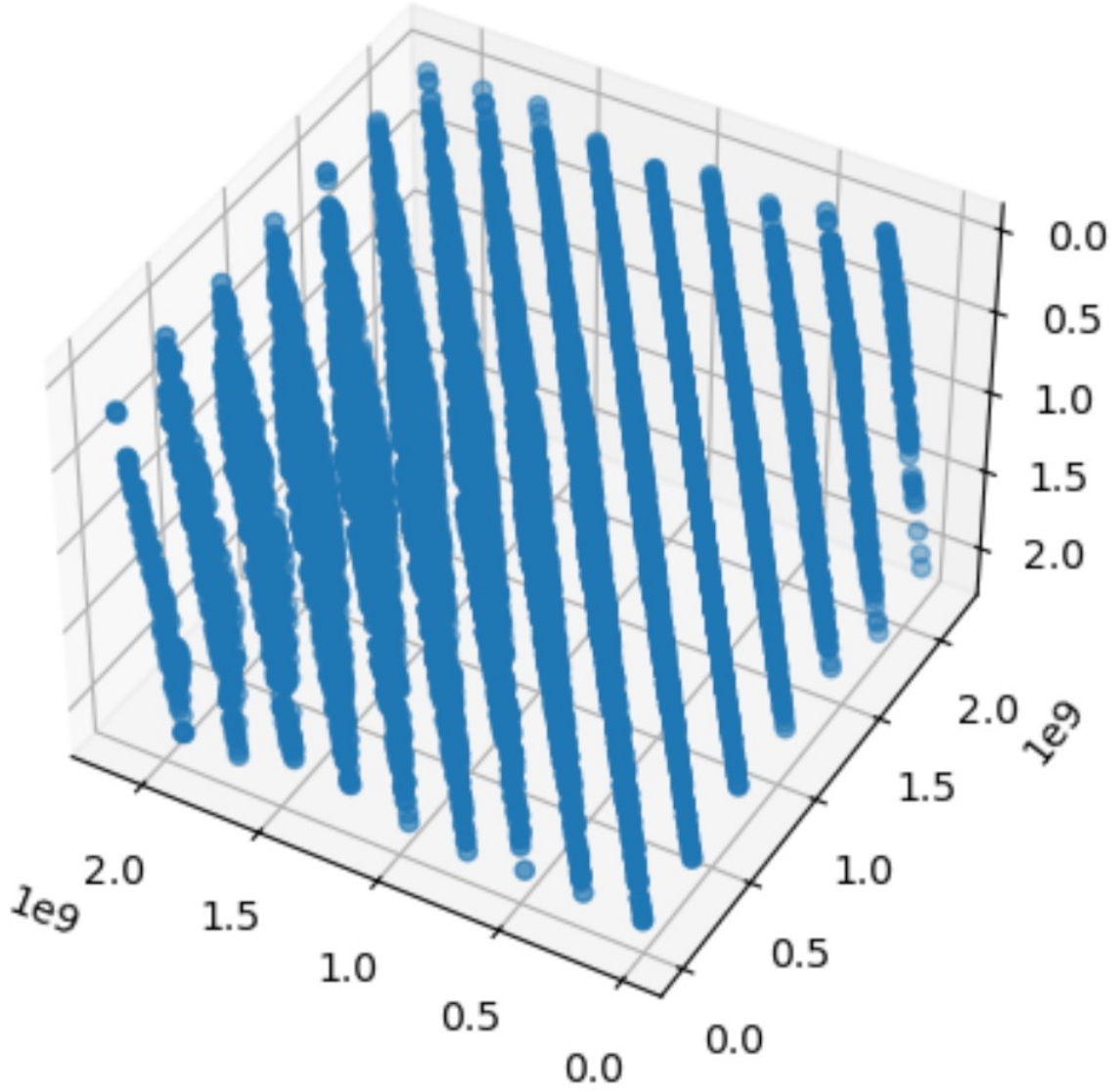
```
class RANDU:
  def __init__(self, seed=483647):
      self.seed = xval
      self.Mod  = 2_147_483_648
      self.Mult = 65539
      self.last = seed

   def random(self):
      self.last = (self.Mult*self.last)%self.Mod
      return self.last
```

RANDU generates integers ranging from 1 to (2^{31}-1)

- RANDU passes basic frequency tests (build a histogram of a sequence. Use a chi-squared test to verify numbers per bin are appropriately equal)

… but RANDU has problems

- It passes frequency tests, but those test overall statistics. They can't find local correlations.
- To look for such correlations, we can take consecutive blocks of three values and plot them as x,y,z coordinates in a 3D scatter plot.
- We see that the values fall along 15 hyperplanes. The generator exhibits local correlations between values.
- This means Monte Carlo results with this generator are suspect since such methods assume uniform random sampling.
- Problems with this generator were known as early as 1963.
- It wasn't until the 1990s that it was widely eliminated, though some Fortran compilers were found using it as late as 1999.



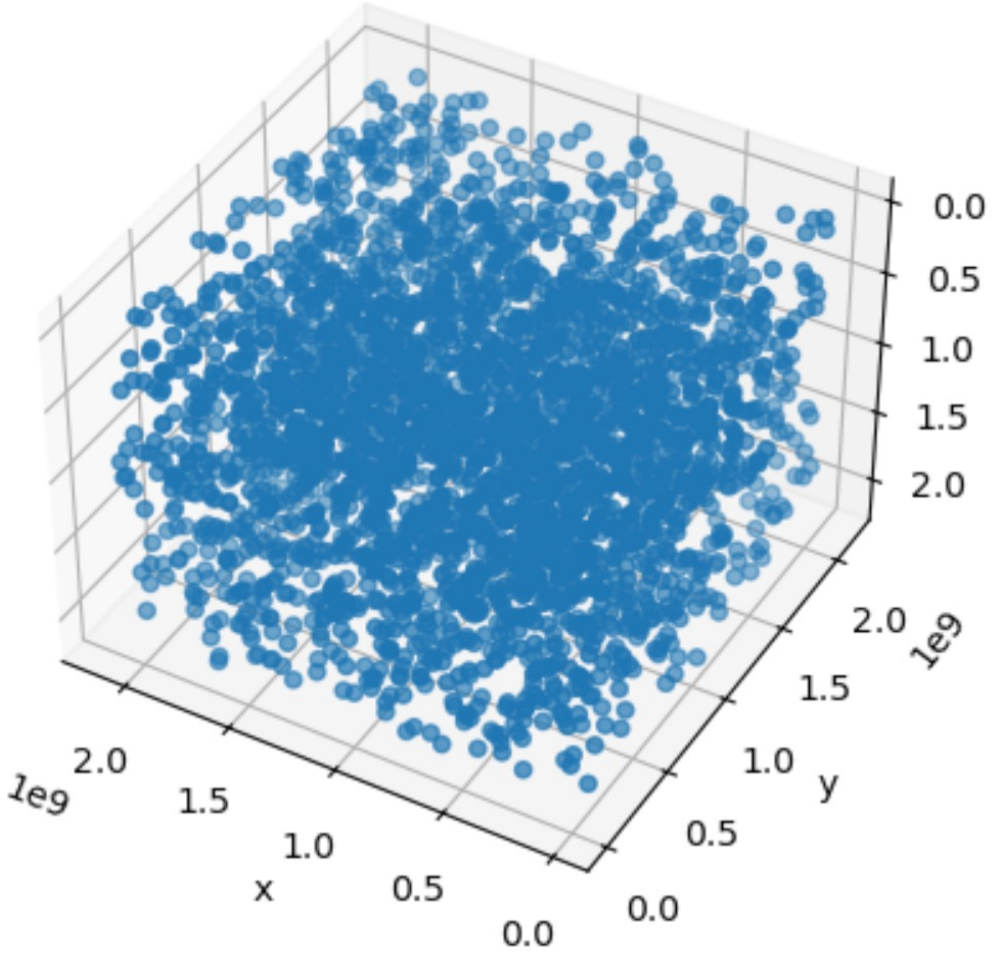
Each point in the plot (x, y, z)) is three consecutive values from RANDU. The points all fall into 15 planes.

Fixing RANDU

- We can fix this generator by more carefully selecting the Multiplier and the modulus value.

- Looking up values from a rigorous (peer reviewed) mathematical work* I updated the values in RANDU:

```
class MultLCG:
  def __init__(self, seed=483647):
      self.seed = xval
      self.Mod  = 2_147_483_647
      self.Mult = 1_583_458_089
      self.last = seed

   def random(self):
      self.last = (self.Mult*self.last)%self.Mod
      return self.last
```



- This new generator passed my frequency tests and removed the local correlations

*Pierre L'Ecuyer, "Tables of Linear Congruential Generators of different sizes and good lattice structure", Mathematics of Computation, Vol 68, Numb 225, jan 1999, pp 249-260

Key Lesson from the RANDU mess

- Maintain a healthy level of skepticism for any default, built-in Random number generator.

- Run your own tests to make sure the numbers are random enough.

- Insist on knowing:
  - Which method the generator is using (e.g. LCG, lagged Fibonacci, Mersenne Twister, etc.)
  - That the period of the generator is sufficient for your problem.
  - That the parameters used in the generator are good and from a reputable source

- I often write my own generator if I can't verify the details of built-in generators, but that is dangerous. It is best to find (and use) a reputable library.
  - Scalable Parallel Random Number Generators (SPRNG) (sprng.org) from Michael Mascagni (University of Florida and NIST)

Lets explore Monte Carlo methods and pseudo random number generators with a classic problem

Monte Carlo methods in Popular Culture

SMBC by Zack Weinersmith

… famous for the best high level explanation of Quantum Computing EVER published:

https://www.smbc-comics.com/comic/the-talk-3

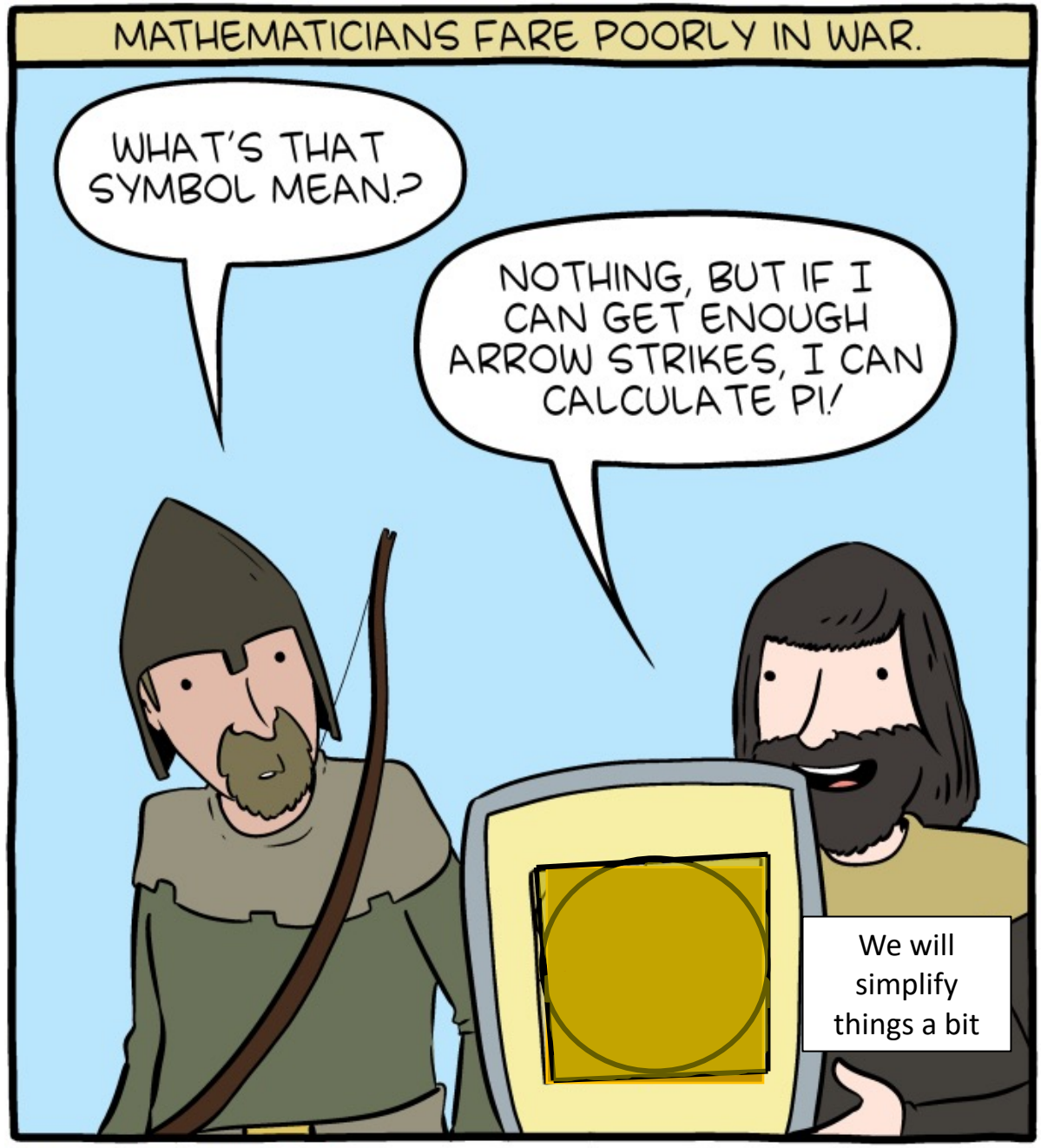
https://www.smbc-comics.com/comic/math-and-war

Monte Carlo methods in Popular Culture

SMBC by Zack Weinersmith

… famous for the best high level explanation of Quantum Computing EVER published:

https://www.smbc-comics.com/comic/the-talk-3

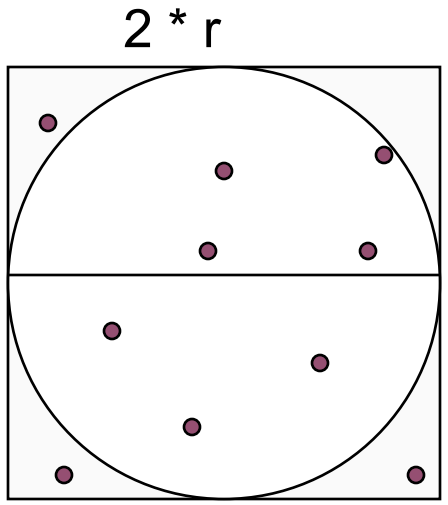


https://www.smbc-comics.com/comic/math-and-war

Monte Carlo Calculations

Using random numbers to solve problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$
- Compute π by randomly choosing points; π is four times the fraction that falls in the circle

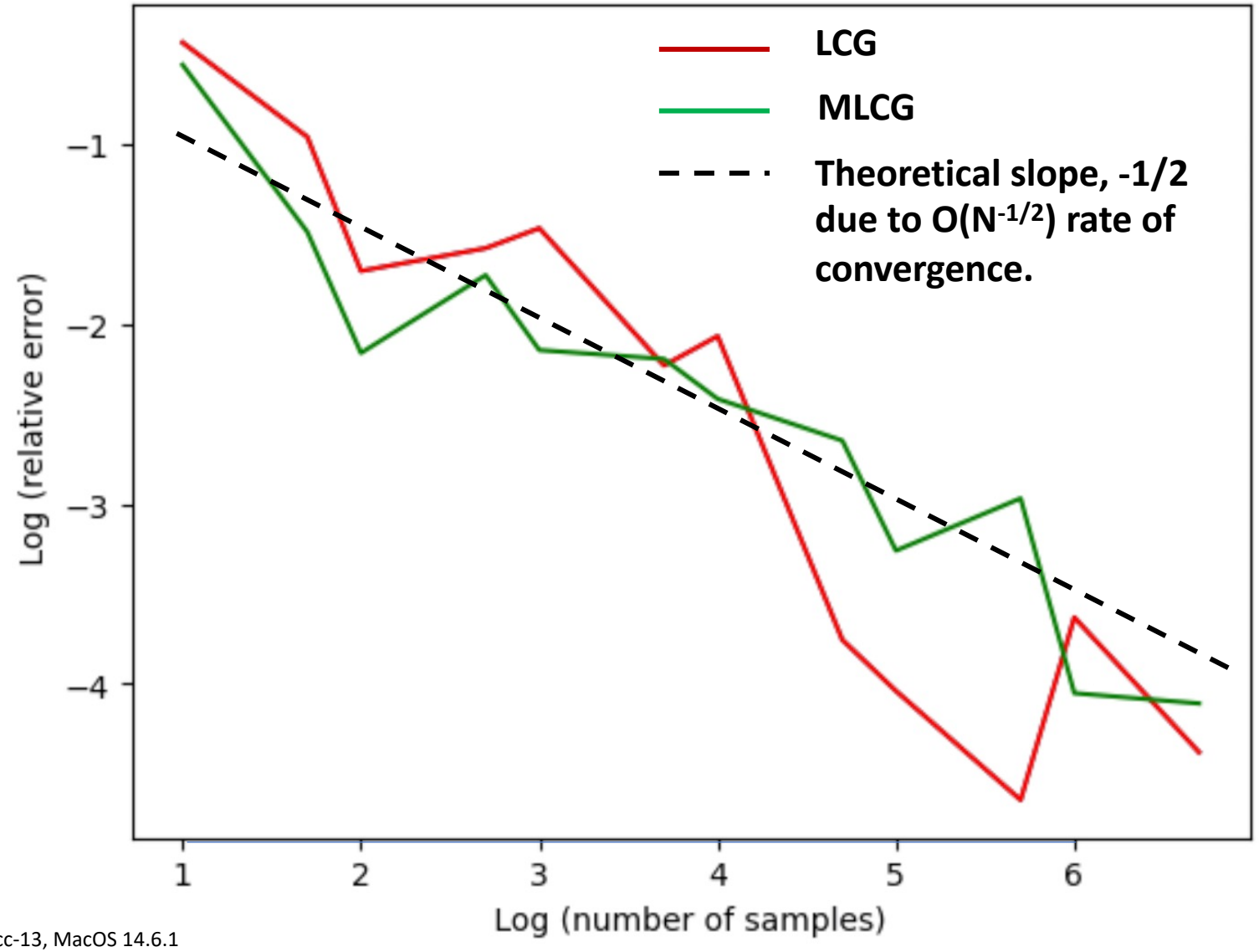
Monte Carlo algorithms: estimating π

```
#include random.h
static long num_trials = 10000;
int main ()
{
  long i;     long Ncirc = 0;       double pi, x, y;
  double r = 1.0;   // radius of circle. Side of squrare is 2*r
  seed(-r, r);          // The circle and square are centered at the origin
  for(i=0;i<num_trials; i++)
  {
    x = drandom();        y = drandom();
    if ( x*x + y*y) <= r*r)   Ncirc++;
   }

   pi = 4.0 * ((double)Ncirc/(double)num_trials);
   printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Single thread results π Monte Carlo: LCG and MLCG

LCG: Linear Congruential Generator $\quad ranNext = (M_{LCG} * ranLast + A)\%mod_{LCG}$

MLCG: Multiplicative Linear Congruential Generator. $\quad ranNext = (M_{MLCG} * ranLast)\%mod_{MLCG}$



// LCG parameters

```
static long MULTIPLIER  = 2416;
static long ADDEND      = 37441;
static long PMOD        = 1771875;
static long SEED        = 7919;
```

// MLCG parameters

```
static long MULTIPLIER  = 1583458089;
static long PMOD        = 2147483647;
static long SEED        = 7325973;
```

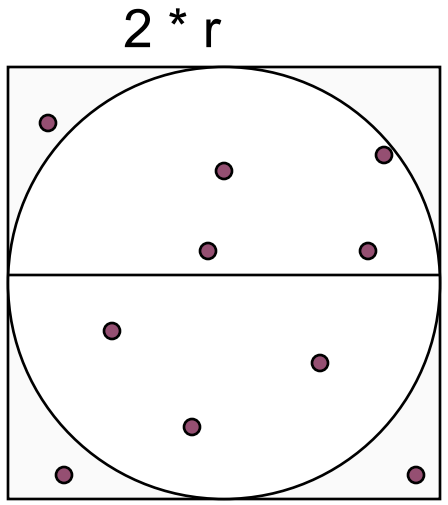
Apple M2 chip, gcc-13, MacOS 14.6.1

… let’s go parallel

Monte Carlo Calculations

Using random numbers to solve problems

- Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc.
- Example: Computing π with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- Throw darts at the circle/square.
- Chance of falling in circle is proportional to ratio of areas:

  $A_c = r^2 * \pi$

  $A_s = (2*r) * (2*r) = 4 * r^2$

  $P = A_c/A_s = \pi /4$
- Compute π by randomly choosing points; π is four times the fraction that falls in the circle

Parallel Programmers love Monte Carlo algorithms

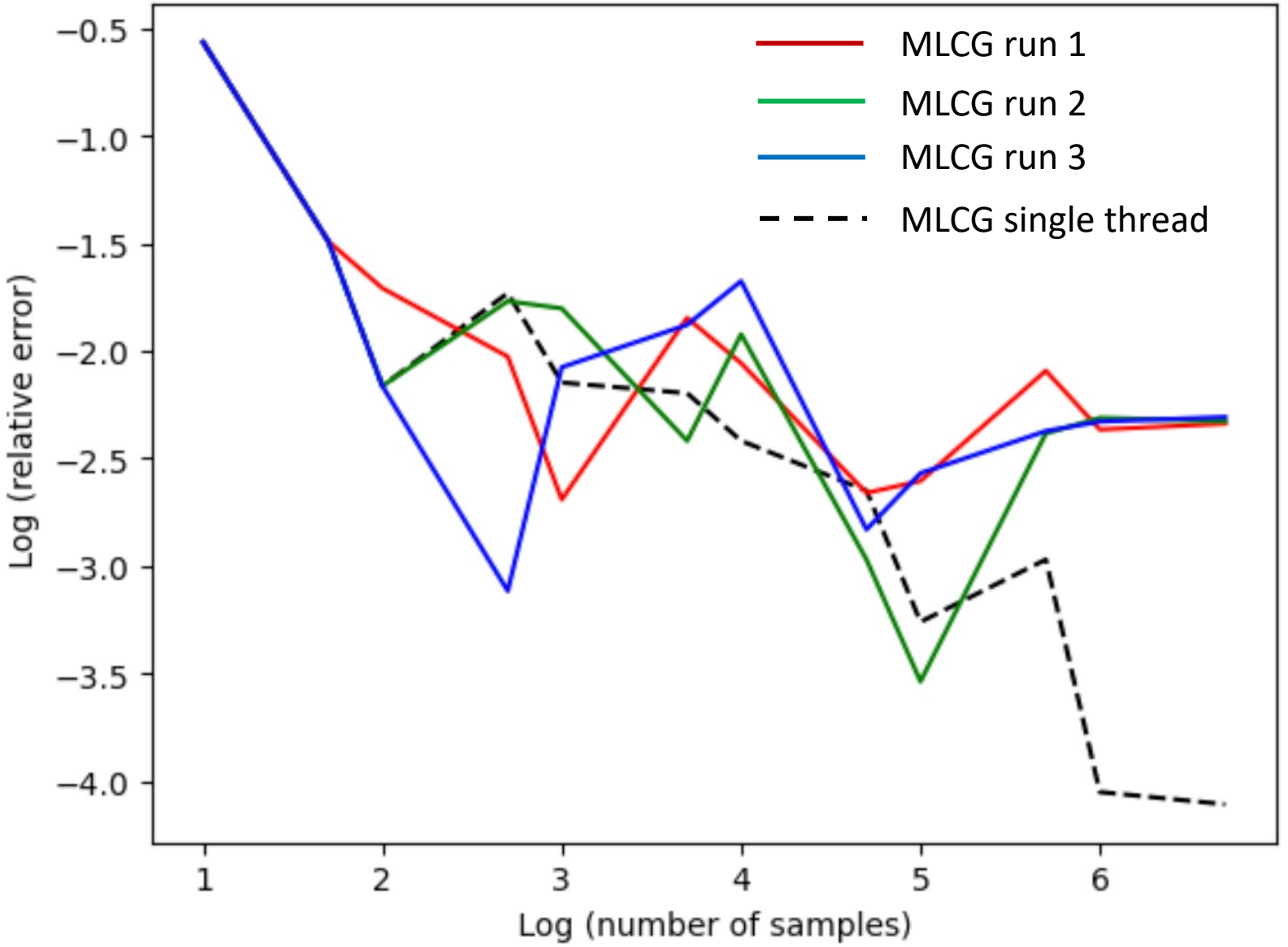
```
#include “omp.h”
static long num_trials = 10000;
int main ()
{
  long i;     long Ncirc = 0;      double pi, x, y;
  double r = 1.0;   // radius of circle. Side of squrare is 2*r
  seed(0,-r, r);  // The circle and square are centered at the origin
  #pragma omp parallel for private (x, y) reduction (+:Ncirc)
  for(i=0;i<num_trials; i++)
  {
    x = random();        y = random();
    if ( x*x + y*y) <= r*r)   Ncirc++;
   }

   pi = 4.0 * ((double)Ncirc/(double)num_trials);
   printf("\n %d trials, pi is %f \n",num_trials, pi);
}
```

Embarrassingly parallel: the parallelism is so easy its embarrassing.

Add two lines and you have a parallel program.

π Monte Carlo with 8 threads: LCG and MLCG

MLCG: Multiplicative Linear Congruential Generator. ranNext = $(M_{MLCG} * ranLast) \% mod_{MLCG}$



Run the same program the same way and get different answers!

That is not acceptable!

Issue: The MLCG generator is not threadsafe

```
// MLCG parameters
static long MULTIPLIER  = 1583458089;
static long PMOD        = 2147483647;
static long SEED        = 7325973;
```

Apple M2 chip, gcc-13, MacOS 14.6.1

MLCG code: threadsafe version

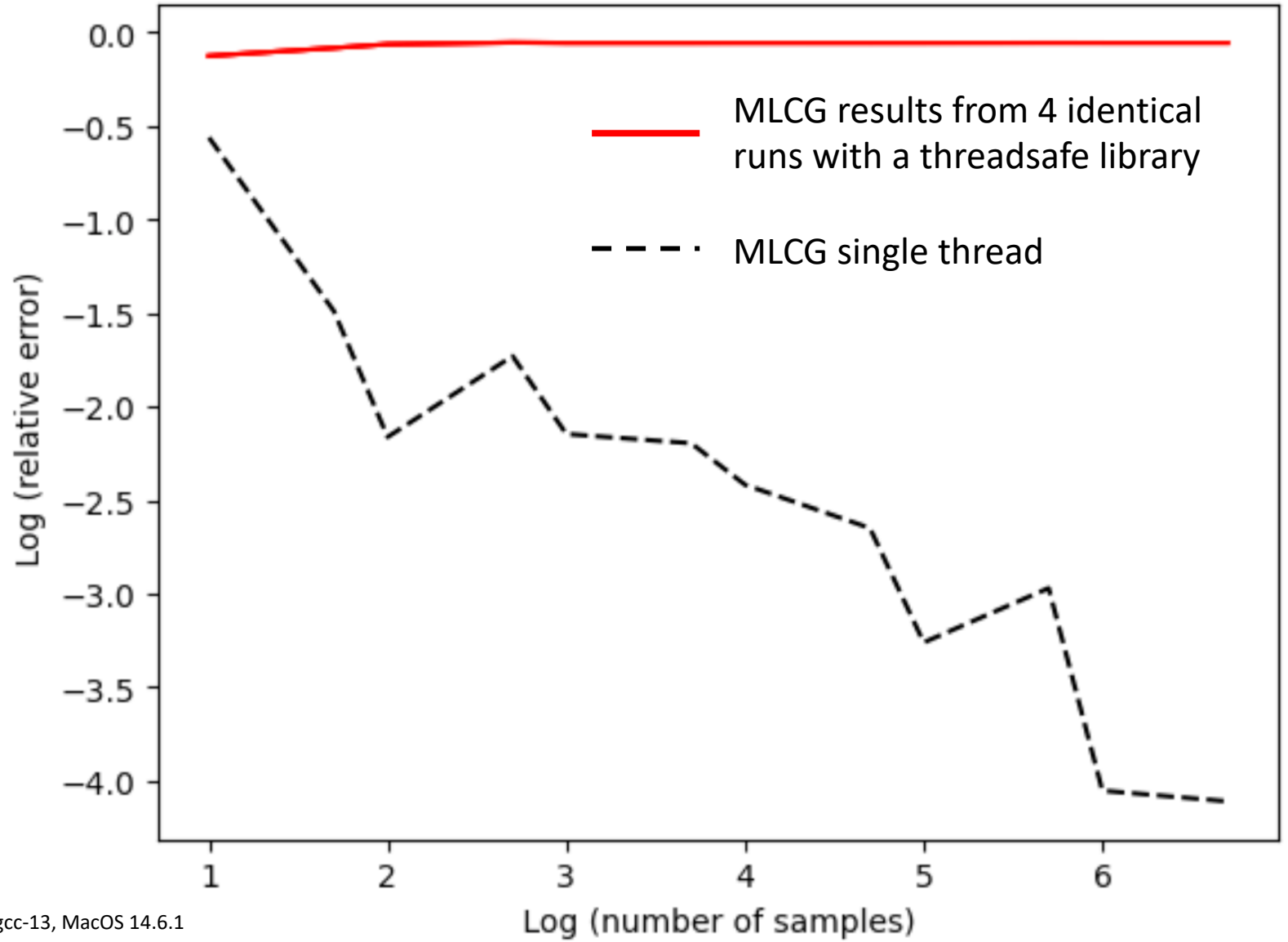
```
static long MULTIPLIER  = 1366;
static long PMOD        = 714025;
long random_last = 0;
#pragma omp threadprivate(random_last)
double random ()
{
   long random_next;

   random_next = (MULTIPLIER  * random_last)% PMOD;
   random_last = random_next;

  return  ((double)random_next/(double)PMOD);
}
```

random_last carries state between random number computations,

To make the generator threadsafe, make random_last threadprivate so each thread has its own copy.

π Monte Carlo with 8 threads: Threadsafe RNG library

MLCG: Multiplicative Linear Congruential Generator. ranNext = $(M_{MLCG} * ranLast)\%mod_{MLCG}$



The library is threadsafe … we get the same results from one run to the next, but the results are awful.

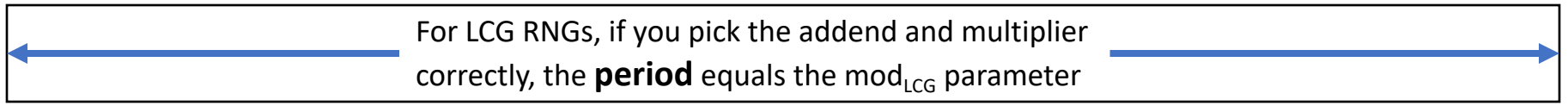
Why?

```
// MLCG parameters
static long MULTIPLIER  = 1583458089;
static long PMOD        = 2147483647;
static long SEED        = 7325973;
```

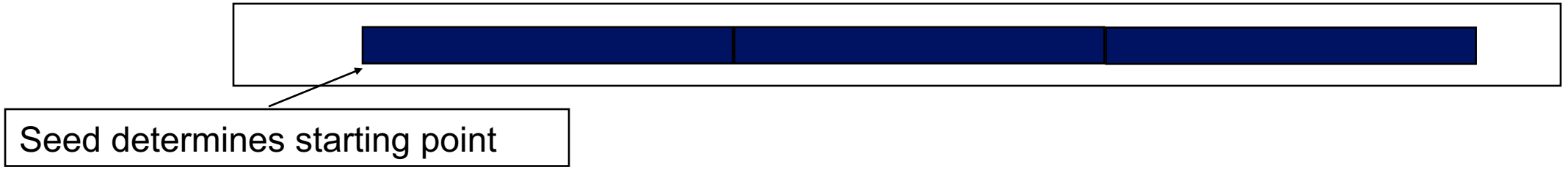
Apple M2 chip, gcc-13, MacOS 14.6.1

Pseudo Random Sequences

- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG



- In a typical problem, you grab a subsequence of the RNG period



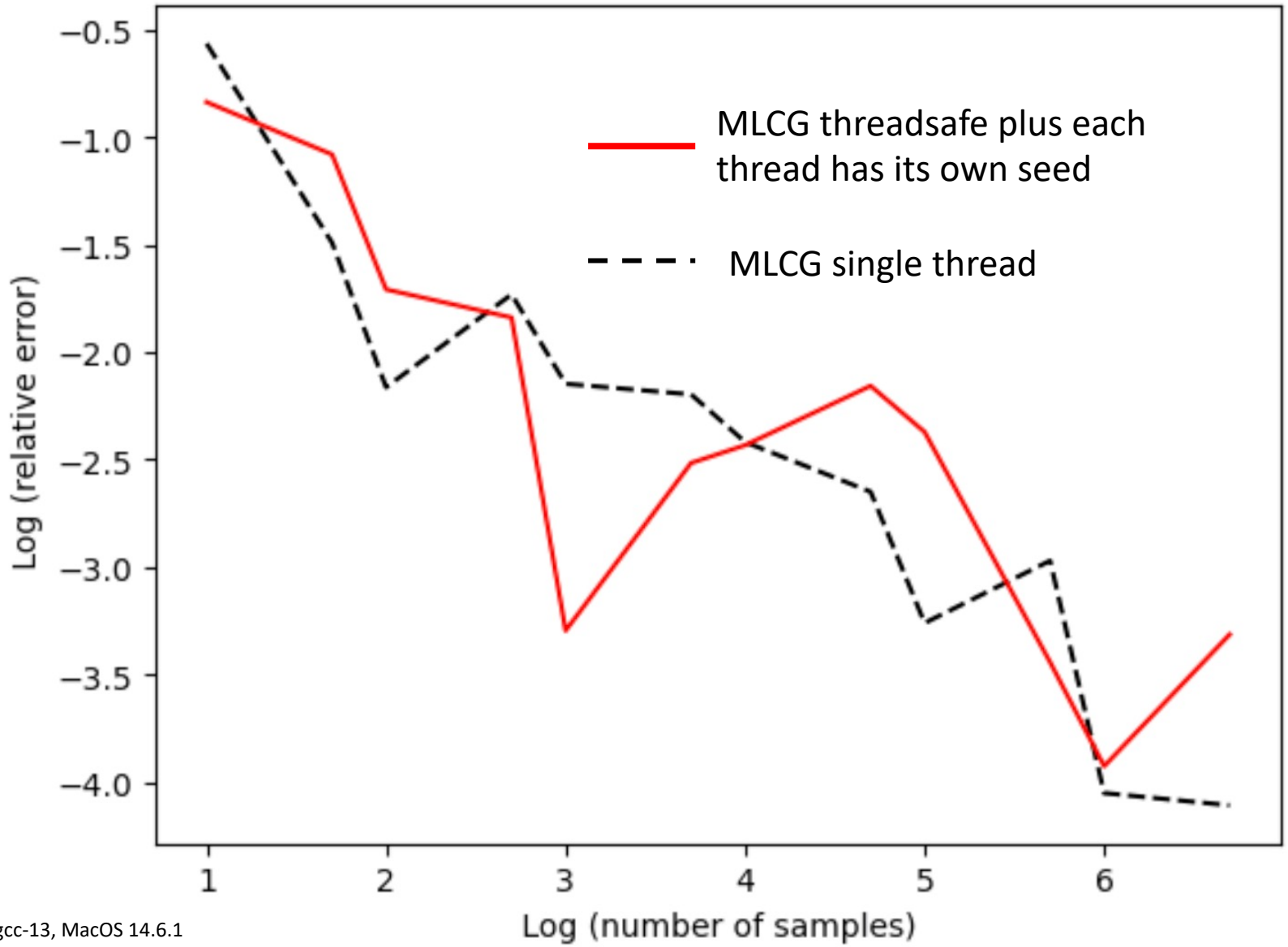
- IF each thread has the same seed, you just sample the same points over and over from each thread.

LCG: Linear Congruential Generator ranNext = $(M_{LCG}*ranLast + A)\%mod_{LCG}$

π Monte Carlo with 8 threads: Different seed per thread

MLCG: Multiplicative Linear Congruential Generator. ranNext = $(M_{MLCG} * ranLast) \% mod_{MLCG}$

Seed(): Called inside parallel region by each thread to set ranLast = SEED * (thread_ID + 1)



Results are much better, but are erratic and degrade for larger cases

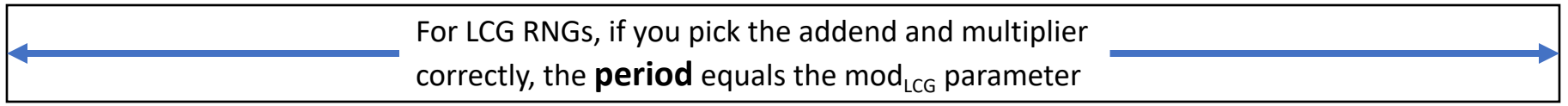
Why?

```
// MLCG parameters
static long MULTIPLIER  = 1583458089;
static long PMOD        = 2147483647;
static long SEED        = 7325973;
```

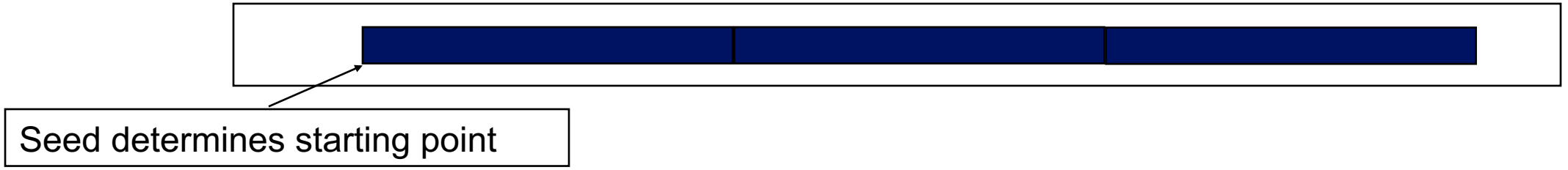
Apple M2 chip, gcc-13, MacOS 14.6.1

Pseudo Random Sequences

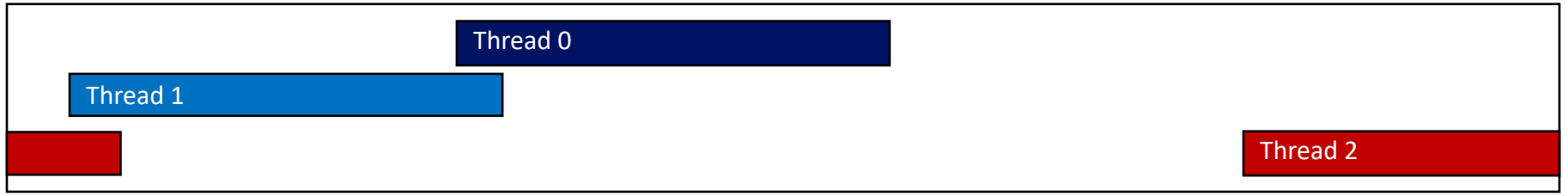
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG



- In a typical problem, you grab a subsequence of the RNG period



- Grab arbitrary seeds and you may generate overlapping sequences



- Overlapping sequences = over-sampling some points and bad statistics … lower quality or even wrong answers!

LCG: Linear Congruential Generator ranNext = $(M_{LCG}*ranLast + A)\%mod_{LCG}$

Parallel random number generators

- Multiple threads cooperate to generate and use random numbers.
- Solutions:
  - Pick a seed and hope for the best (a common approach)
  - Give each thread a separate, independent generator
  - Have one thread generate all the pseudo-random numbers.
  - Leapfrog … deal out sequence values "round robin" as if dealing a deck of cards.
  - Block method … pick your seed so each threads gets a distinct contiguous block.

If done right, can generate the same sequence regardless of the number of threads …

Important for validation and debugging, but not needed for high quality results.

- Other than "replicate and hope", these are difficult to implement. Be smart … get a math library that does it right.

The state of the art is the Scalable Parallel Random Number Generators Library (SPRNG): http://www.sprng.org/ from Michael Mascagni's group at Florida State University.

Leap Frog (skipping) Method (for MLCG)

- Interleave samples in the sequence of pseudo random numbers:
  - Thread i starts at the i^{th} number in the sequence
  - Stride through sequence, stride length = number of threads.
- Result … the same sequence of values regardless of the number of threads.

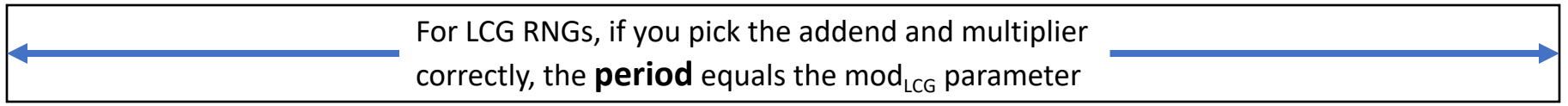
```
#pragma omp single
{   nthreads = omp_get_num_threads();
    int id = omp_get_thread_num();
    iseed = PMOD/MULTIPLIER;     // just pick a seed
    pseed[0] = iseed;
    mult_n = MULTIPLIER;
    for (i = 1; i < nthreads; ++i)
   {
       iseed = (unsigned long long)((MULTIPLIER * iseed) % PMOD);
       pseed[i] = iseed;
       mult_n = (mult_n * MULTIPLIER) % PMOD;
   }

 }
 random_last = (unsigned long long) pseed[id];
```

One thread computes offsets and a strided multiplier (mult_n)

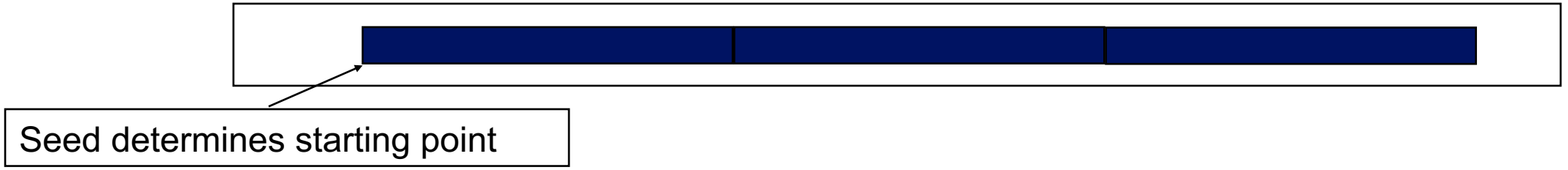
Each thread stores offset starting point into its threadprivate “random_last” value

Pseudo Random Sequences

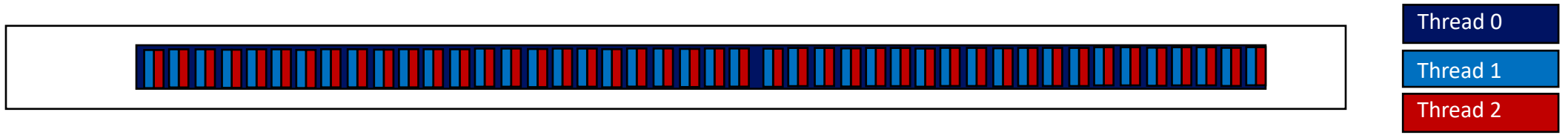
- Random number Generators (RNGs) define a sequence of pseudo-random numbers of length equal to the period of the RNG



- In a typical problem, you grab a subsequence of the RNG period



- Skip by the number of threads and start at seeds offset by a number of positions = to thread ID



- Parallel threads sample the same sub-sequence … you get the same answer regardless of the number of threads.

LCG: Linear Congruential Generator ranNext = $(M_{LCG}*ranLast + A)\%mod_{LCG}$

LeapFrog: Same sequence with many threads.

- We can use the leapfrog method to generate the same answer for any number of threads.
- These results are for Leapfrog with the MLCG generator (r_{new} = (Mult*r_{last})%Mod)

Samples	One thread	2 threads	4 threads
1000000	3.139852	3.139852	3.139852
5000000	3.140930	3.140930	3.140930
10000000	3.140884	3.140884	3.140884
50000000	3.141199	3.141199	3.141199
100000000	3.141348	3.141348	3.141348

- Used two streams of pseudo-random numbers … one for x and one for y. This was needed to make (x,y) pairs consistent as number of threads changed.
- Stream1 MCLG:
  - `Mult: 1583458089`
  - `Mod: 2147483647`
  - `Seed: 2147483647`
- Stream 2 MCLG:
  - `Mult: 295397169;`
  - `Mod: 1073741789`
  - `Seed: 7727`

Outline

- Numbers for humans. Numbers for computers
- Finite precision, floating point numbers and the IEEE 754 Standard
- Responding to “issues” in floating point arithmetic
- The use and abuse of Random Numbers
- Wrap-up/Conclusion

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.
- You now know how to use (and abuse) pseudo-random numbers.
- It is shockingly easy to use them incorrectly … I lack detailed survey-data but based on anecdotal evidence, I suspect a large number of published papers using Monte Carlo methods are broken due to abuse of pseudo-random numbers.
- Important rules to follow:
  - Be careful with default, built-in random number generators.
  - Know the method your generator is using and confirm that the parameters give you the period you need.
  - Use a quality (tested/validated) generator. They are fun to write, but it’s a job best left to professionals.
  - Don’t be stupid about using generators in parallel. There are parallel generators ”out there” (such as SPRNG). Use them. http://www.sprng.org/

Be careful. There is some extremely bad advice “out there”. For example, from https://luscher.web.cern.ch/luscher/ranlux/ …
“The ranlux generator is widely used in Monte Carlo simulation programs. Such simulations are often performed on parallel computers, where each MPI process runs a private copy of the generator (with different seeds).”

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

- The Scalable Parallel Random Number Generator (SPRNG).
  - http://www.sprng.org/

- Slides and exercises for my lecture series, Computer Science for Physics-people
  - git clone https://github.com/tgmattso/CompSciForPhys.git