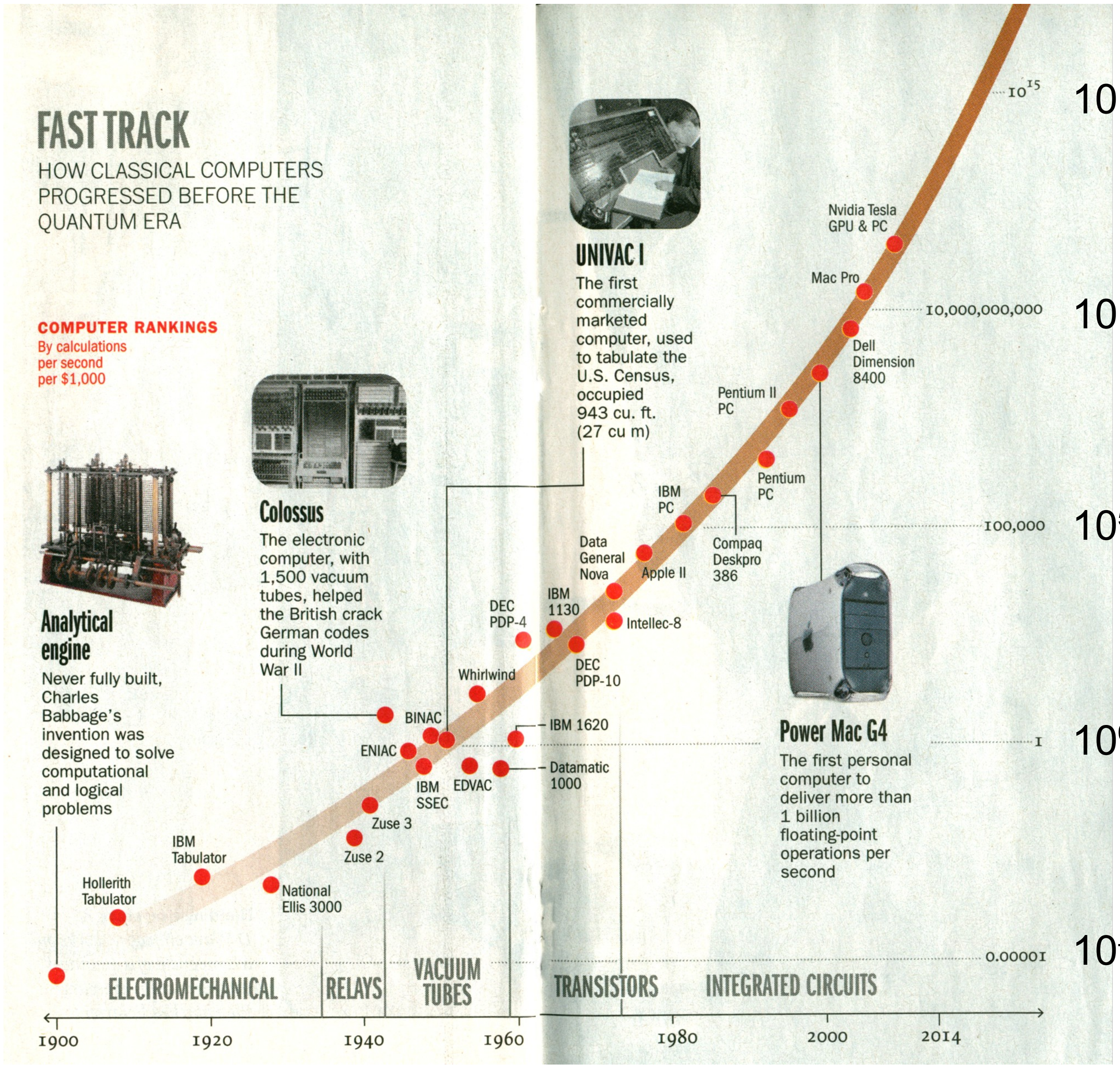
Dr. John Fattaruso
Background

- Ph.D. Electrical Engineering, U. C. Berkeley; minors Electromagnetic theory, Statistics
- ~22 years at Texas Instruments; Analog circuit and solid state device design; Distinguished Member of the Technical Staff
- ~40 years of numerical programming in machine languages, Fortran, C, C++, Java
- Fall Semesters 2011-2020 taught Physics 3340 at SMU

History of Computational Economy



10^{15}

10^{10}

10^{5}

10^{0}

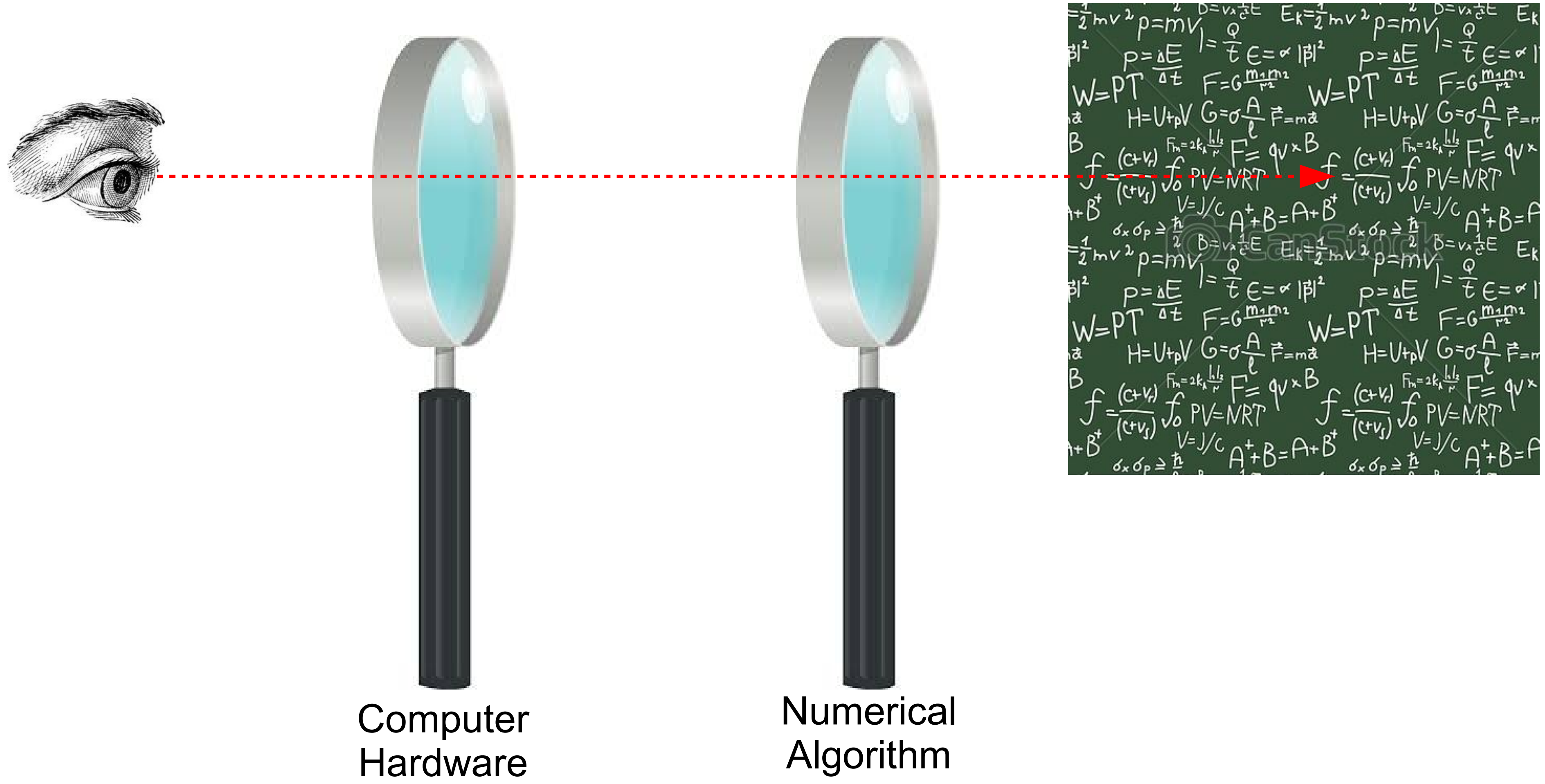
10^{-5}

Calculations per second per $1,000

(Time magazine, 17 Feb 2014)

1900 1920 1940 1960 1980 2000 2014

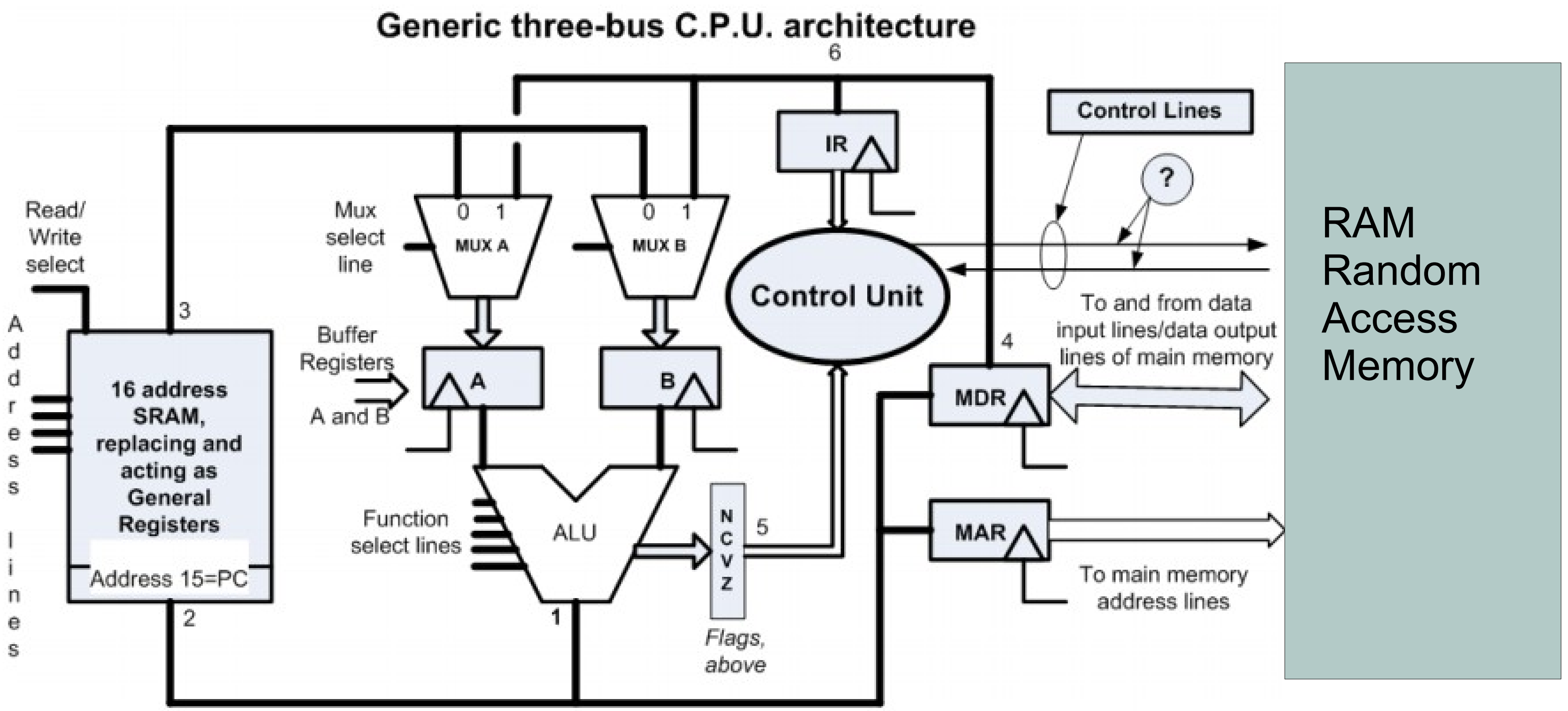
Solving Physics Problems Numerically



Despite the exponential growth in computing power, each of these tools has its own properties and limitations that must be understood

“Never in the history of mankind has it been possible to produce so many wrong answers so quickly” - Carl Erik Fröberg

A Glance at Typical Computer Architecture



The computer memory, the registers and the arithmetic/logic unit (ALU) have fixed width of a number of parallel binary bits, so each of the binary arithmetic steps needed for a calculation are rounded to a limited, though large, precision.

The Control Data Corporation 6600

At the beginnings of widespread scientific computation, the CDC6600 was designed with a 60-bit word size. Nowadays we have very inexpensive 64-bit machines available. Why are computation widths of this order?

Preserving Numerical Precision

Look at the rules of basic addition of numbers in base 10, for example, 3210 + 4.56:

```
3210.00
+  4.56
-------
3214.56
```

We shift numbers horizontally so that the decimal points are in vertical alignment

Now how about using "scientific notation", for example, $3.21\times10^{6} + 4.56\times10^{-4}$:

```
3.210000000000×10⁶
0.000000000456×10⁶
-----------------
3.210000000456×10⁶
```

Now the shifting also includes exponent addition or subtraction so that the decimal points are in vertical alignment **and** the exponents agree

What about an example of $3.21\times10^{18} + 4.56\times10^{-17}$? Do we have enough width on our computation space for the horizontal shifting?

Complete “Loss of Precision” Error

Look at example $3.21{\times}10^{6} + 2.46{\times}10^{-6}$:

```
3.2100000000000×10^6
0.0000000000246×10^6
----------------
3.2100000000246×10^6
```

Correct sum

Width limit of arithmetic storage and processor

```
3.2100000000000×10^6
0.0000000000246×10^6
----------------
3.21000000000×10^6
```

Rounded sum has completely lost any contribution from one operand

Width limit of arithmetic storage and processor

```
3.2100000000000×10^6
0.0000000000246×10^6
----------------
3.210000000000×10^6
```

Rounded sum has completely lost any contribution from one operand

Loss of Precision and Roundoff Error

Look at example $3.21\times10^{6} + 2.46\times10^{-4}$:

```
3.210000000000×10⁶
0.000000000246×10⁶
--------------
3.210000000246×10⁶
```

Correct sum

Width limit of arithmetic storage and processor

```
3.210000000000×10⁶
0.000000000246×10⁶
--------------
3.2100000002×10⁶
```

Rounded sum = correct sum – 4.6×10^{-5} roundoff error

Width limit of arithmetic storage and processor

```
3.210000000000×10⁶
0.000000000246×10⁶
--------------
3.21000000025×10⁶
```

Rounded sum = correct sum + 4.0×10^{-6} roundoff error

Roundoff Error

Look at example 1.0 / 3.0:

Width limit of arithmetic storage and processor

```
3.33333333333333333...×10⁻¹

3.33333333333×10⁻¹
```

Rounded result = correct result – 3.33...$\times 10^{-13}$ roundoff error

Look at example 2.0 / 3.0:

Width limit of arithmetic storage and processor

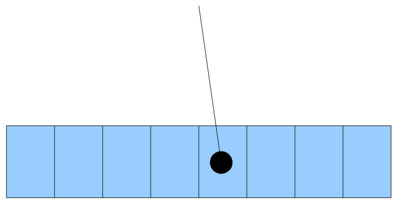
```
6.66666666666666666...×10⁻¹

6.66666666667×10⁻¹
```

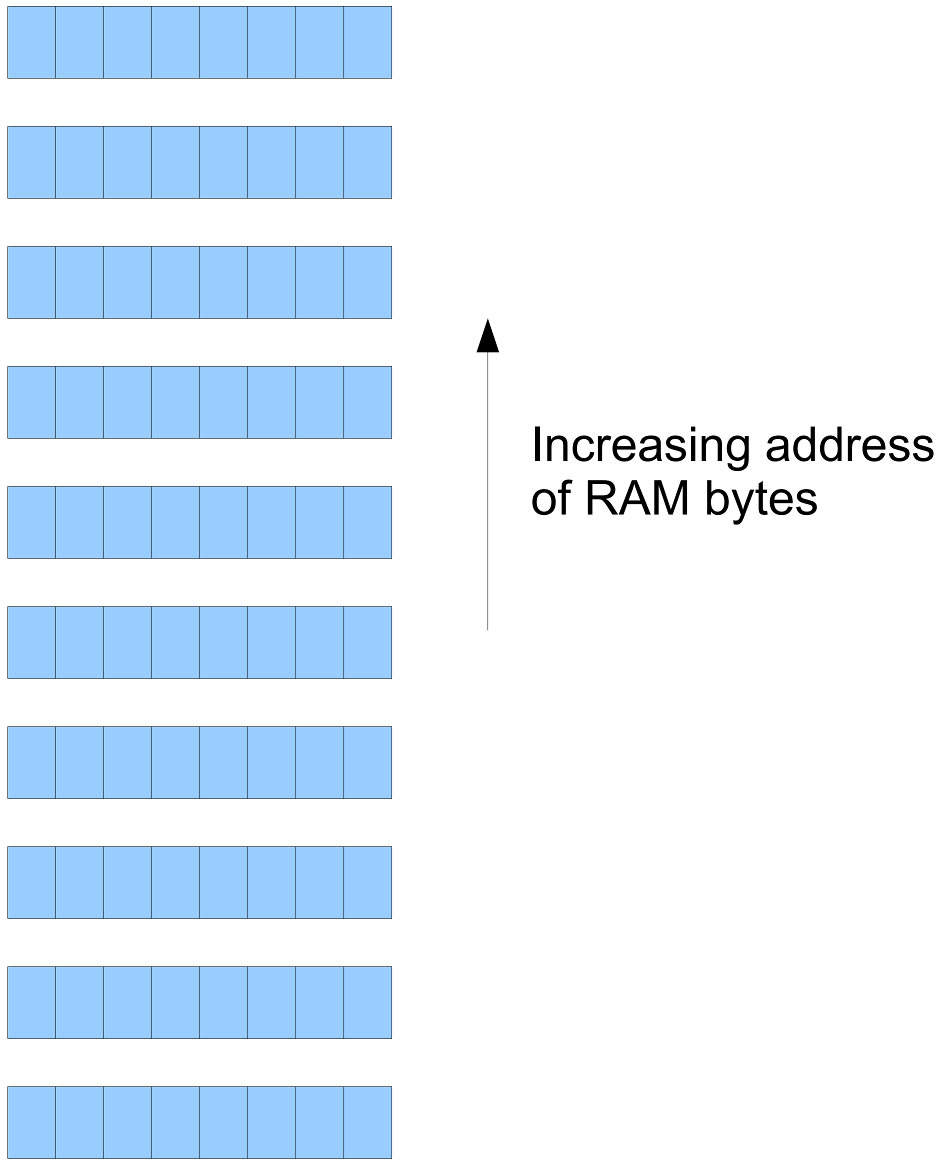
Rounded result = correct result + 3.33...$\times 10^{-13}$ roundoff error

Understanding numerical limits and roundoff error is essential in scientific computing. Of course, computer hardware uses binary arithmetic, but the same principles apply.

Apparent Map of Bytes in Modern RAM

One byte = 8 bits:
The minimum
addressable chunk
of data



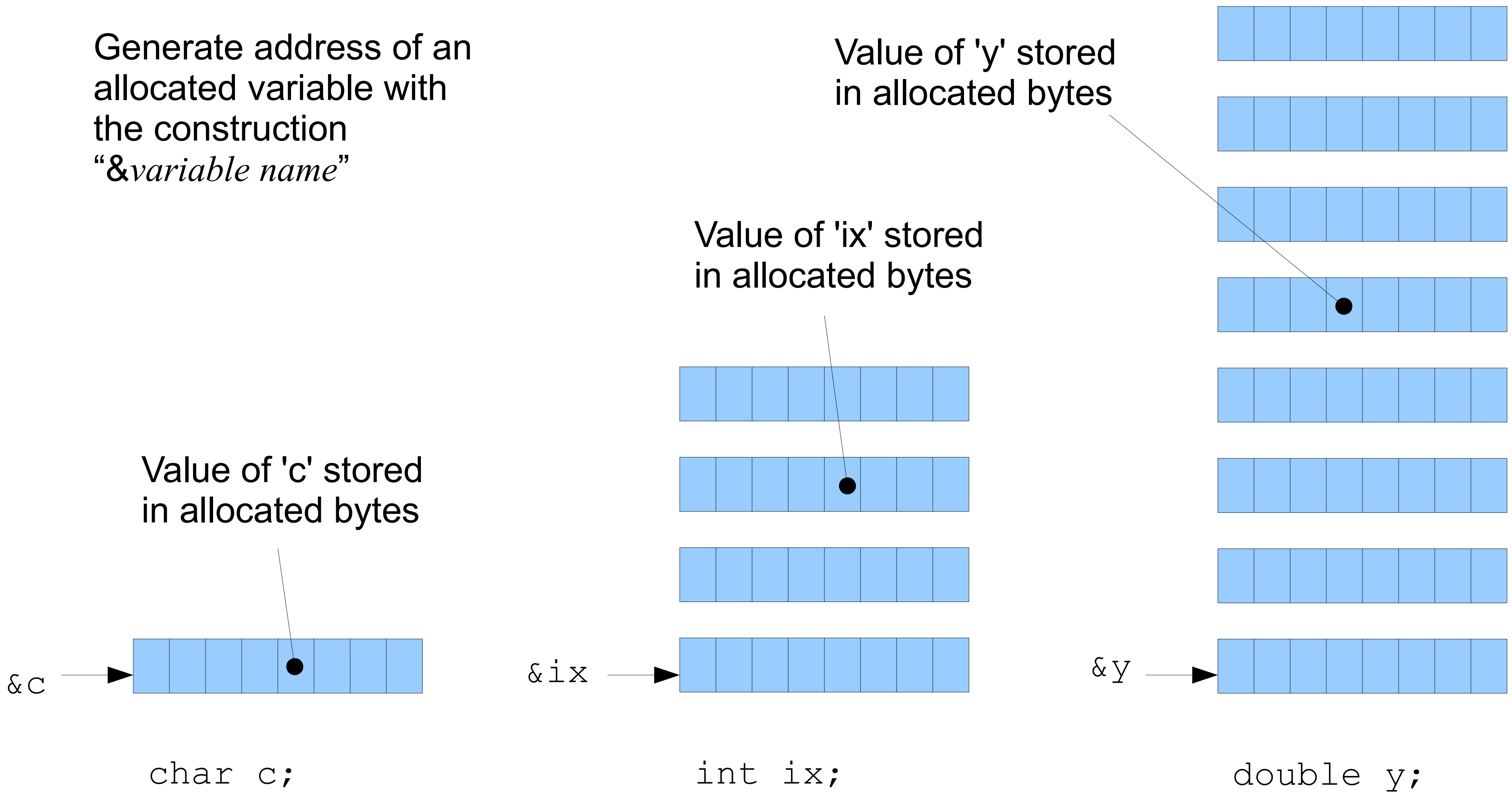


Larger binary structures, such as 32-bit or 64-bit numbers, are spread over multiple bytes starting with a first addressable byte. Memory is actually read and written in these larger aggregates of bytes in hardware.

C Data Types for Numerical Programming

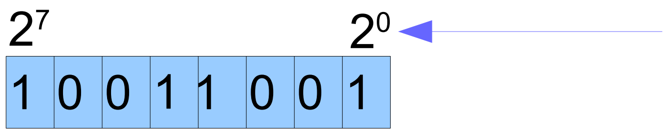
- Integer
  - Named “int”
  - Usually 32 bits, 4 bytes, or 64 bits, 8 bytes
  - Least significant byte at lowest memory address on Intel based processors
- Single precision floating point
  - Named “float”
  - Usually 32 bits, 4 bytes
- Double precision floating point
  - Named “double”
  - Usually 64 bits, 8 bytes
- Character
  - Named “char”
  - Usually 8 bits, 1 byte

Allocation of C Data Types in RAM


Generate address of an allocated variable with the construction "&*variable name*"
Value of 'y' stored in allocated bytes
Value of 'ix' stored in allocated bytes
Value of 'c' stored in allocated bytes
&c
&ix
&y
char c;
int ix;
double y;

Total Possible Bit Arrangements

For any collection of *n* items that can *independently* assume one of two states...



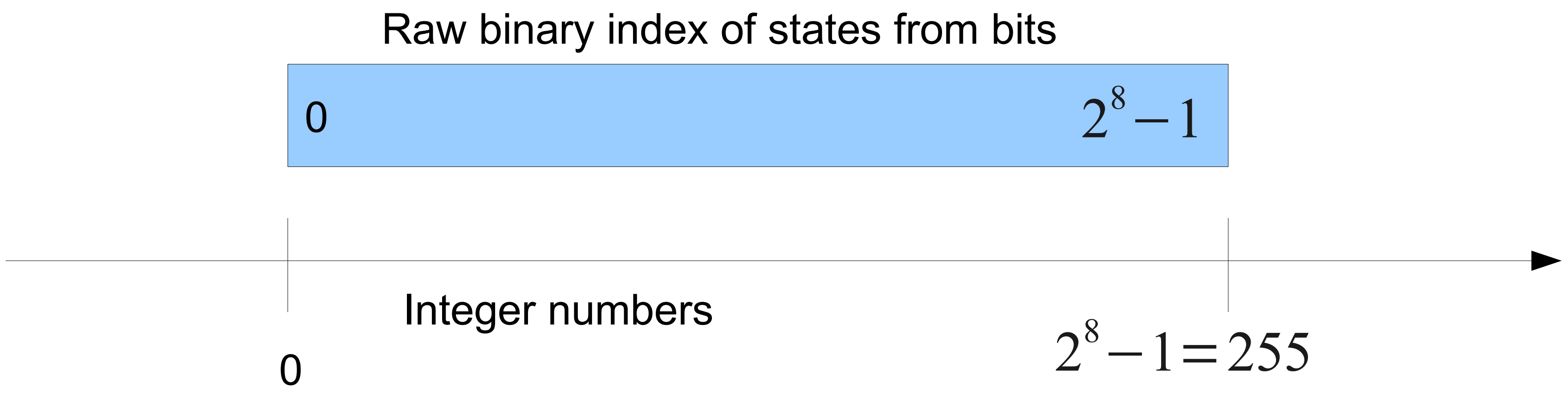
Binary weights to form an intrinsic index of all states

The total number of possible unique arrangements is ...

Unsigned 8-bit Character Values

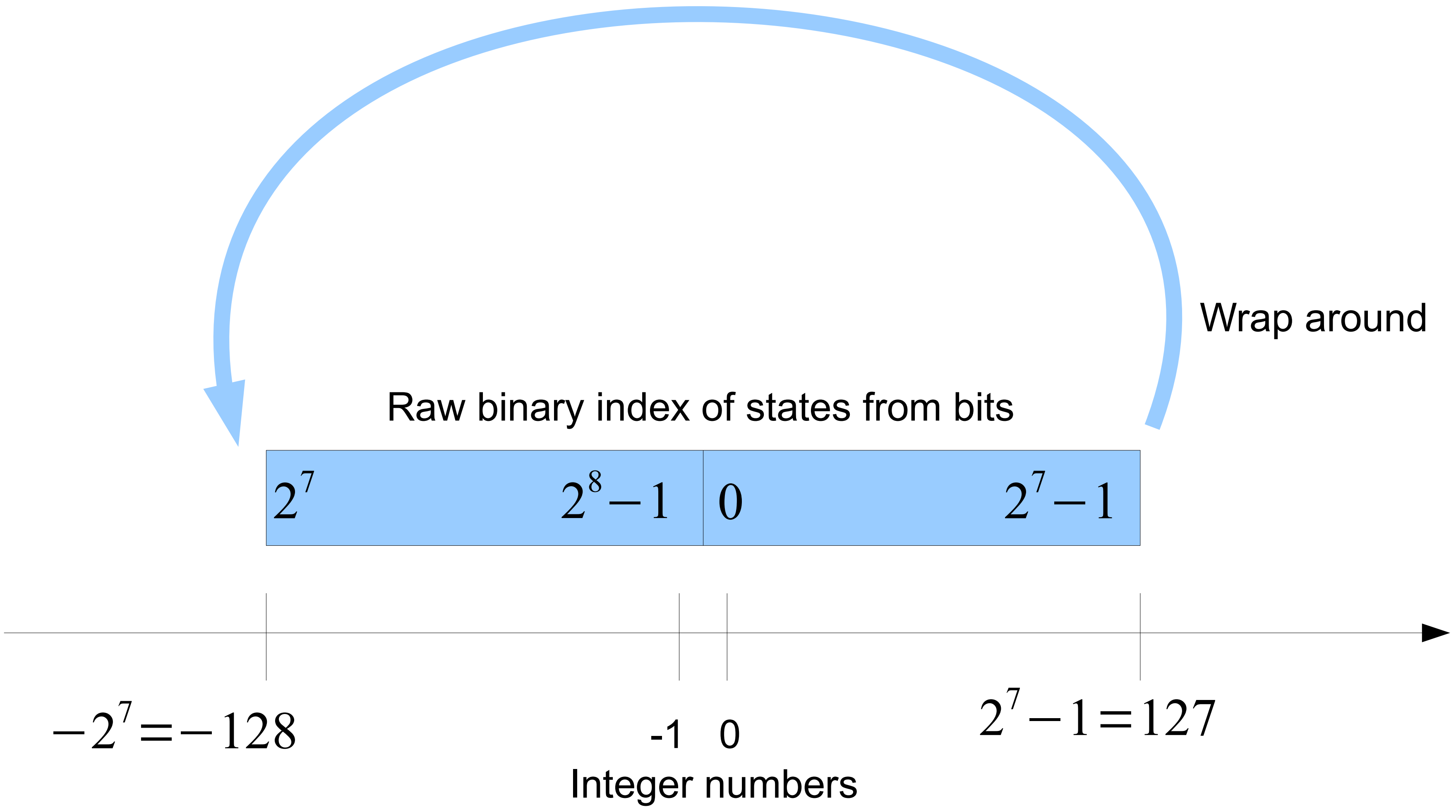
Total number of possible bit arrangements = $2^n = 2^8 = 256$

How do we assign numerical values to each of these arrangements?

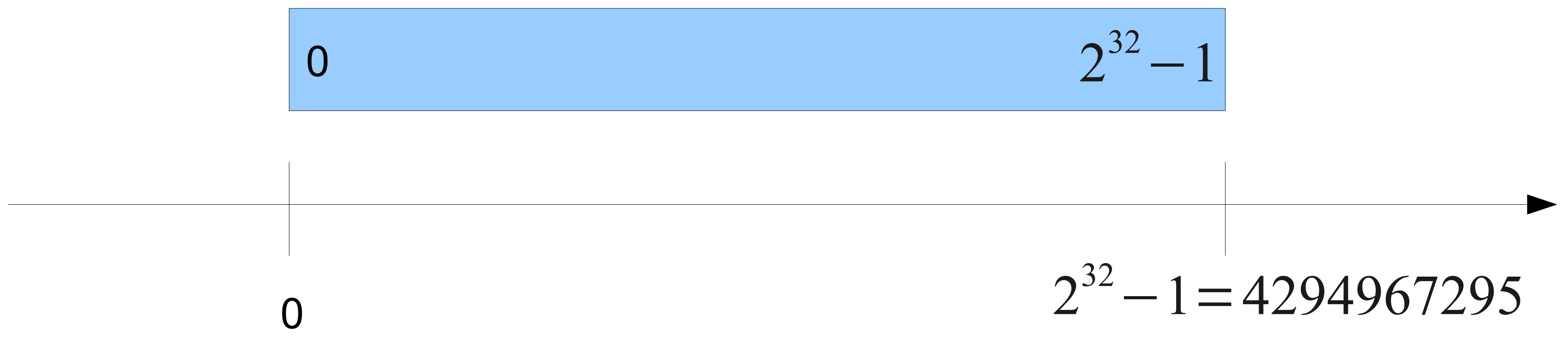


Signed 8-bit Character Values

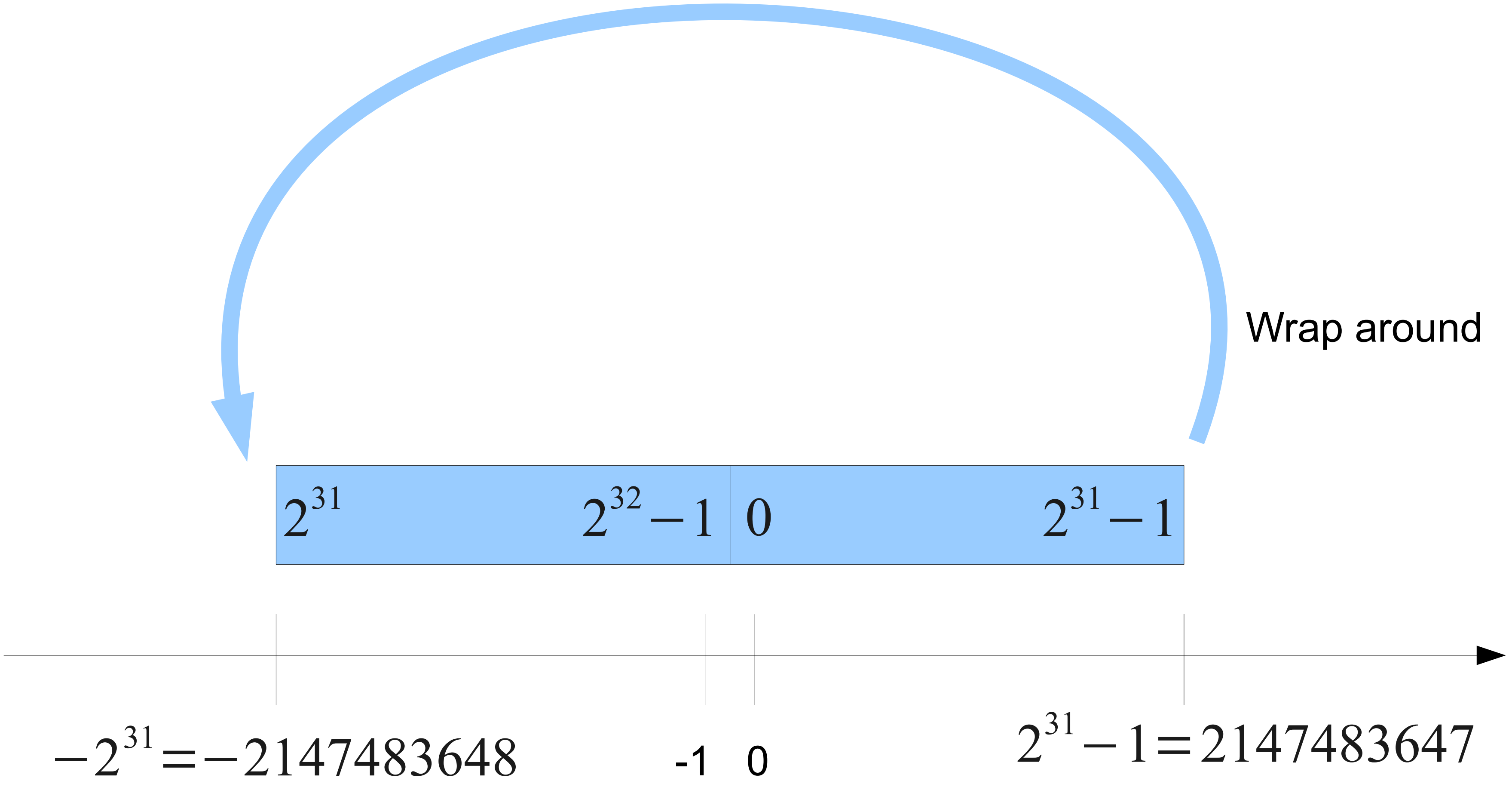
Same possible bit arrangements, but a different numerical mapping...



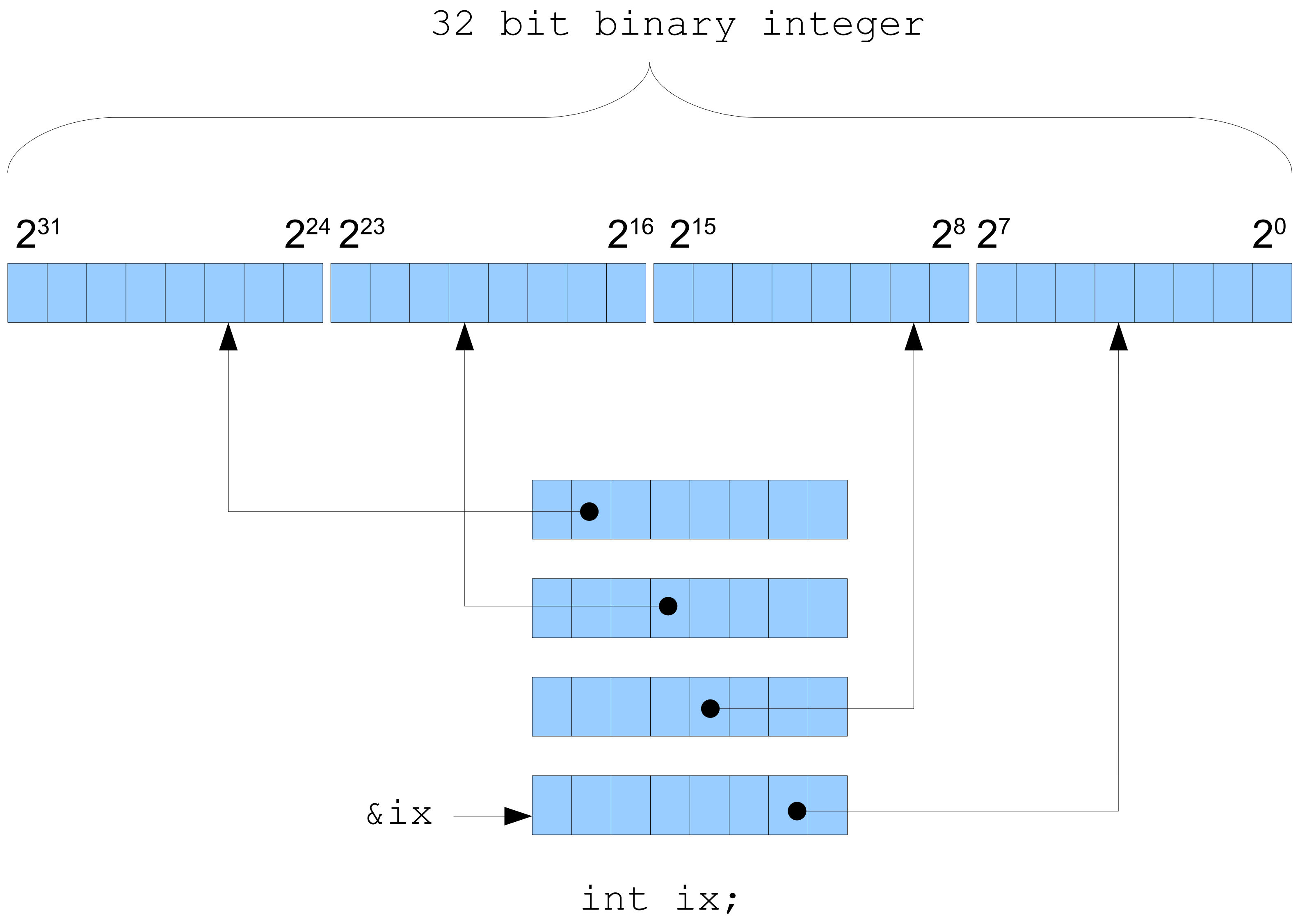
Unsigned 32-bit Integer Values



Signed 32-bit Integer Values



Integer Bit Arrangement for “Little Endian” Processors



Pseudocode for Seeing Integer Wraparound

```
**integer** $i, s$;
$s \leftarrow 0$
**for** $i = 1$ **to** $50$ **do**
    $s \leftarrow s + 268435456$
    **output** $s$
**end for**
```

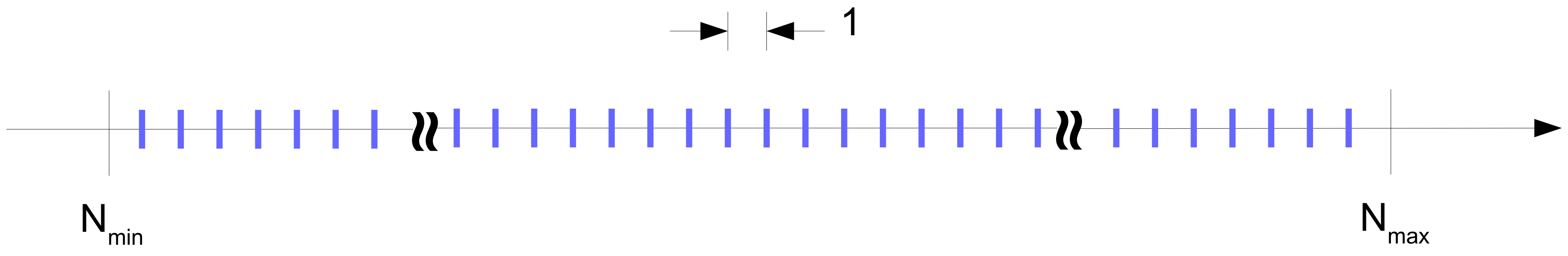
Note: 268435456 = 2^{28}

Wraparound C Code Output

```
s=268435456
s=536870912
s=805306368
s=1073741824
s=1342177280
s=1610612736
s=1879048192
s=-2147483648
s=-1879048192
s=-1610612736
s=-1342177280
s=-1073741824
s=-805306368
s=-536870912
s=-268435456
s=0
s=268435456
s=536870912
s=805306368
s=1073741824
s=1342177280
s=1610612736
s=1879048192
s=-2147483648
```

Density on Number Line

For all forms of integer variables, the density of numbers represented is uniform between the minimum and maximum numbers, with constant interval of 1



Integer numbers

Estimating Large Powers of 2

$$2^{10} = 1024 \approx 10^{3}$$

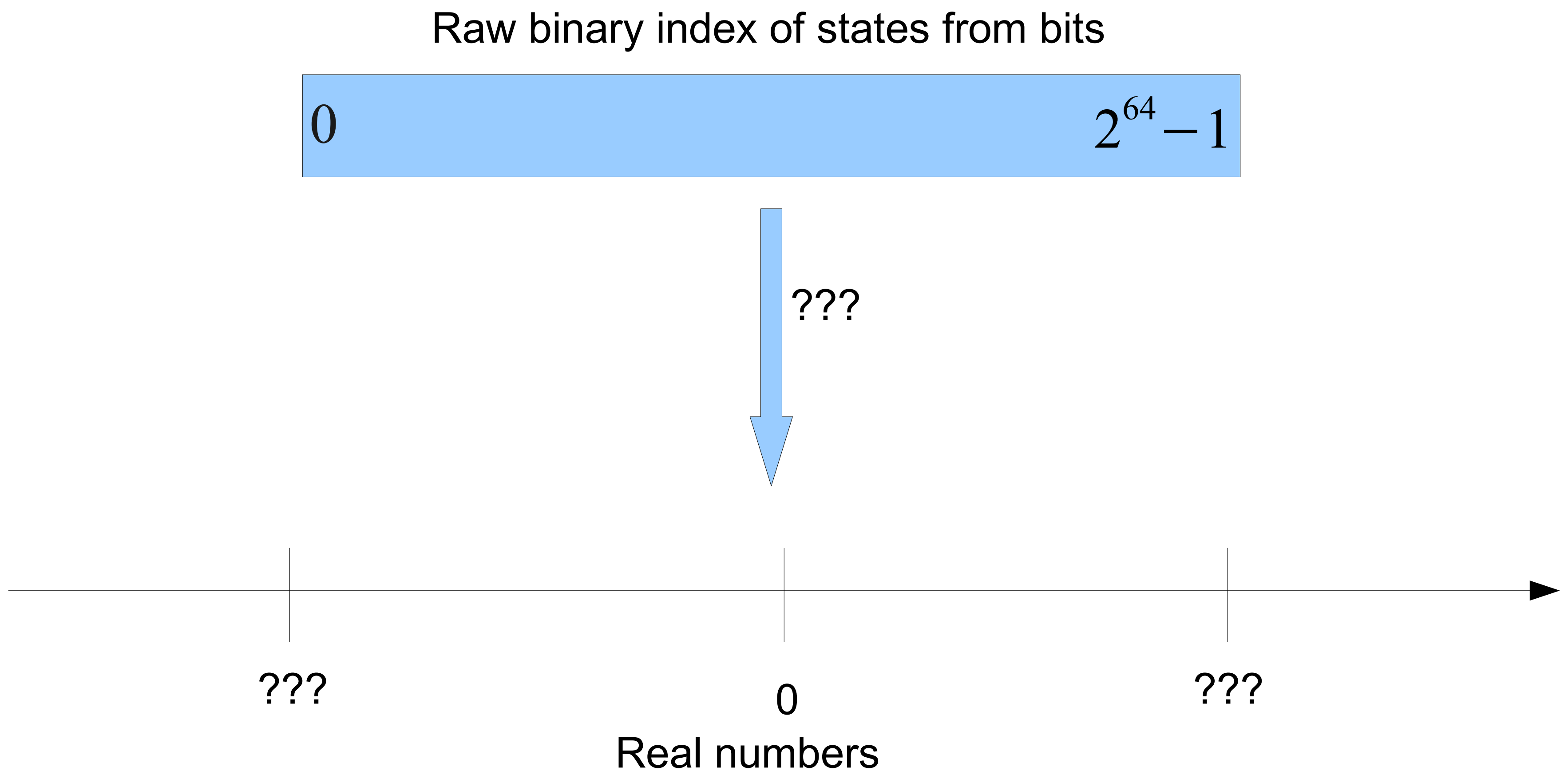
So for every 10 binary bits, count 3 powers of 10

Examples:

$$2^{32} = 2^{2} \cdot 2^{30} \approx 2^{2} \cdot 10^{3 \cdot 3} \approx 4 \cdot 10^{9}$$
$$2^{64} = 2^{4} \cdot 2^{60} \approx 2^{4} \cdot 10^{6 \cdot 3} \approx 16 \cdot 10^{18} = 1.6 \cdot 10^{19}$$

Representing “Real” Values

There are a finite number of bits available in RAM. We usually allot either 32 or 64 bits for each “real” variable in computer code. There are still either 2^{32} or 2^{64} possible bit combinations. How do we map values on an infinitely divisible line of real numbers?



Floating Point Normalization

When we use “Scientific Notation”, we naturally keep numbers in a normalized form. That is, we typically would write the number 123 as

1.23×10^{2}

Or sometimes

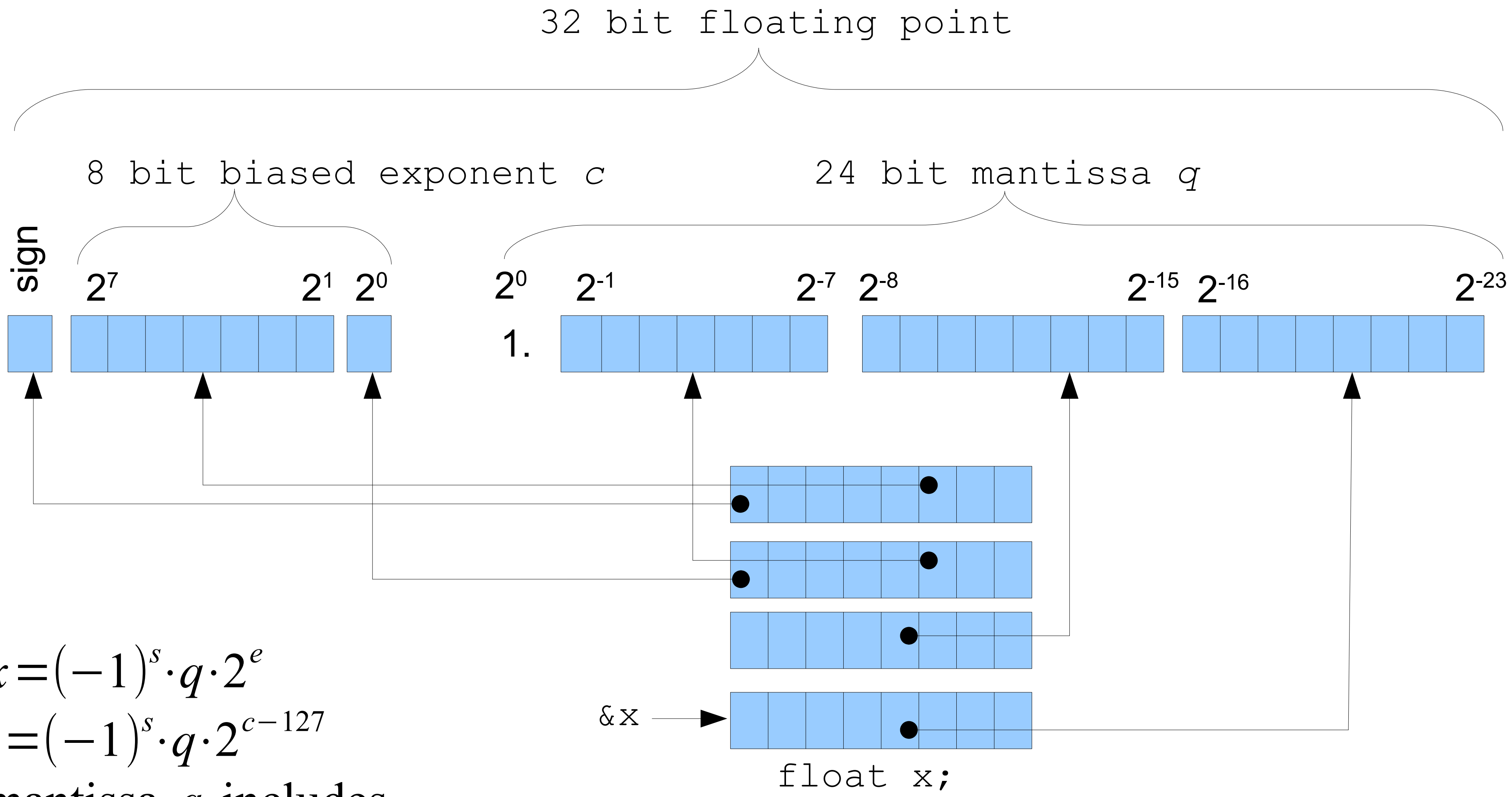
0.123×10^{3}

But not any of

12.3×10^{1}
0.0123×10^{4}
123.0×10^{0}
0.00123×10^{5}
etc...

Similarly, machine hardware will keep floating point numbers normalized, so that the most significant '1' bit is always the hidden '1' bit.

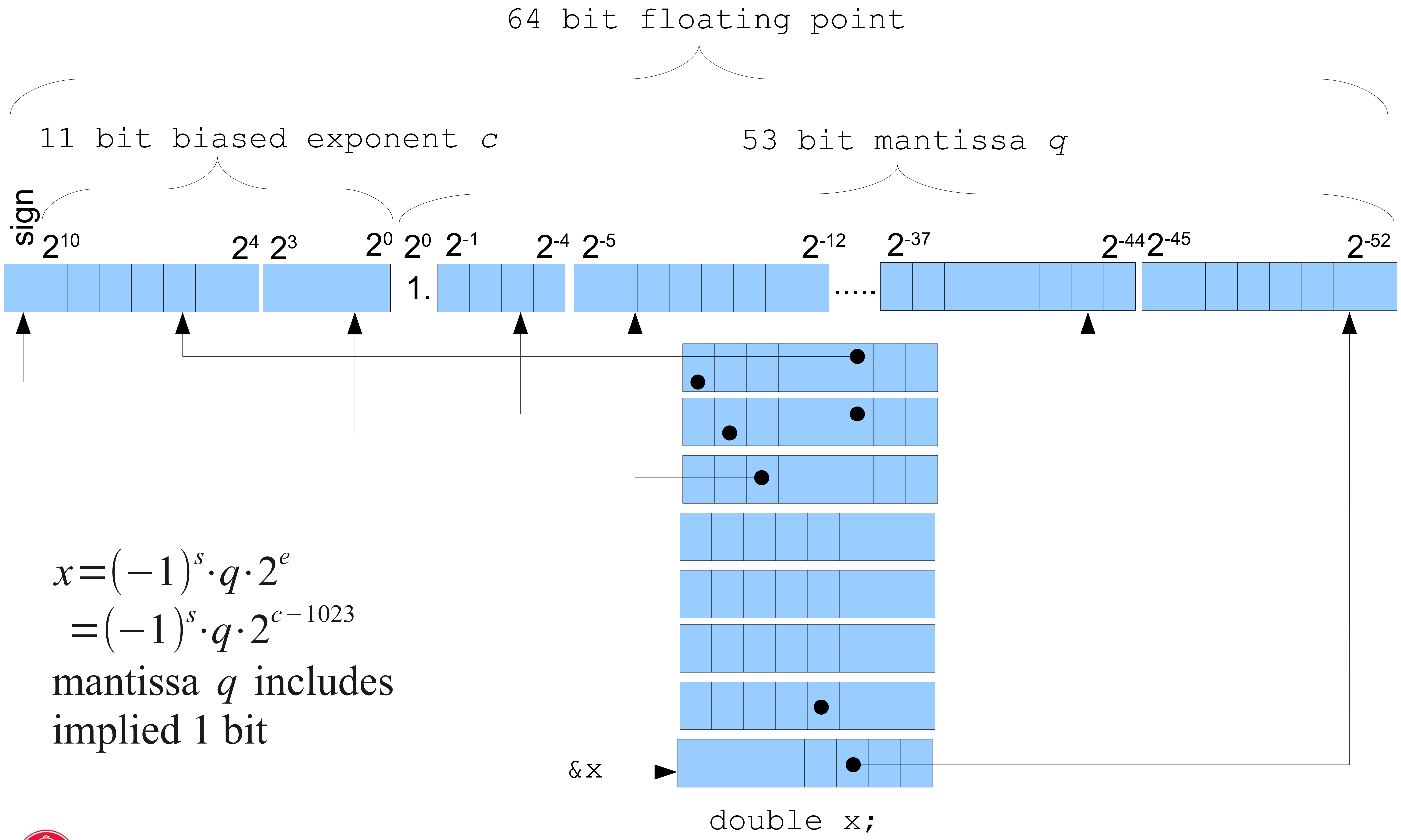
Single Precision Floating Point Bit Arrangement



$$x = (-1)^s \cdot q \cdot 2^e$$
$$= (-1)^s \cdot q \cdot 2^{c-127}$$

mantissa q includes implied 1 bit

Double Precision Floating Point Bit Arrangement



$$x = (-1)^s \cdot q \cdot 2^e$$
$$= (-1)^s \cdot q \cdot 2^{c-1023}$$

mantissa q includes implied 1 bit

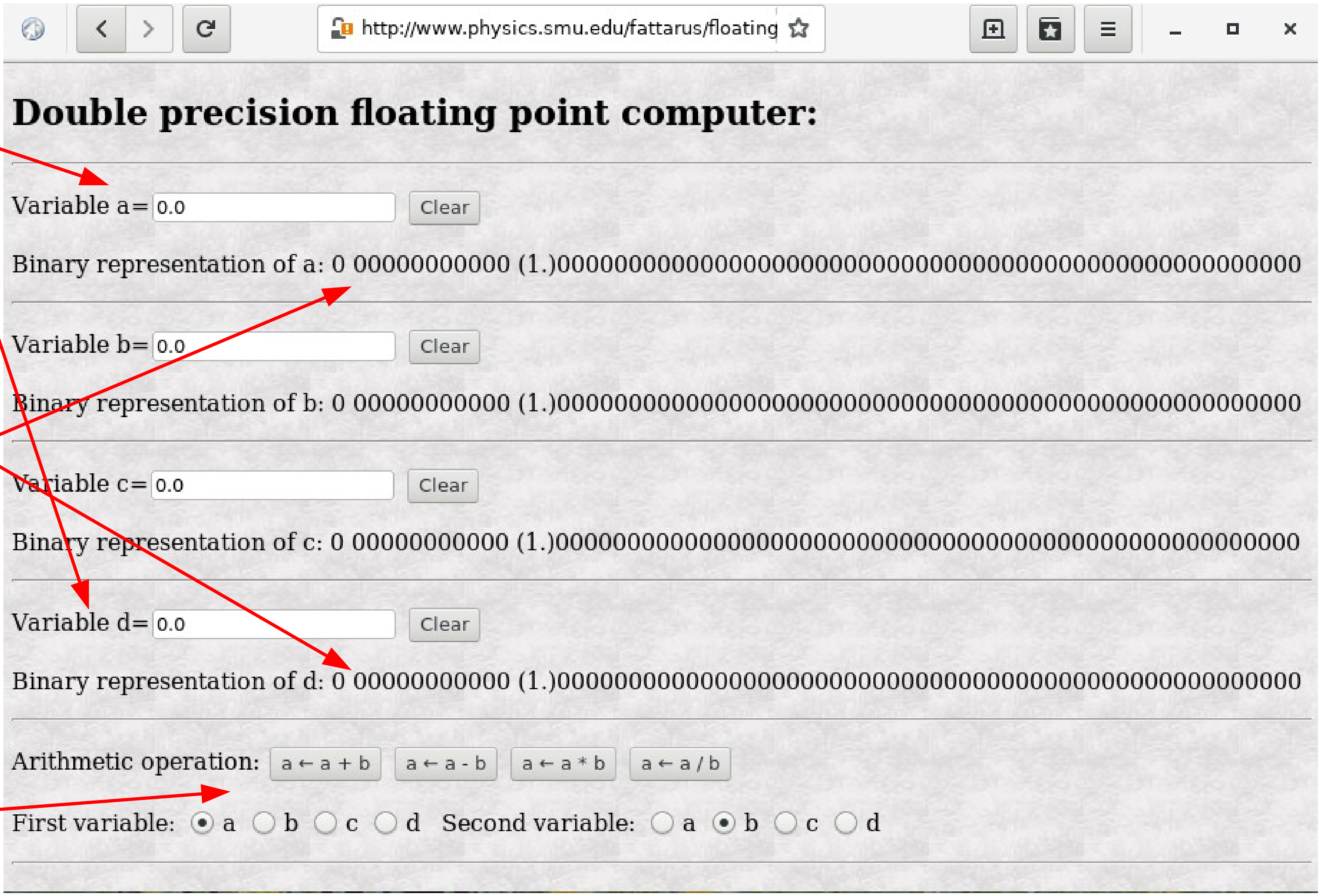
Floating Point Web Browser Calculator

http://www.physics.smu.edu/fattarus/floating_point.html

Four double precision variables: a,b,c,d

Binary representation of each double precision variable: sign bit, biased exponent, mantissa with implied 1.

Double precision arithmetic operations between any two variables



Double precision floating point computer:

Variable a= 0.0

Binary representation of a: 0 00000000000 (1.)0000000000000000000000000000000000000000000000000000

Variable b= 0.0

Binary representation of b: 0 00000000000 (1.)0000000000000000000000000000000000000000000000000000

Variable c= 0.0

Binary representation of c: 0 00000000000 (1.)0000000000000000000000000000000000000000000000000000

Variable d= 0.0

Binary representation of d: 0 00000000000 (1.)0000000000000000000000000000000000000000000000000000

Arithmetic operation:

First variable: [✓ a] b c d Second variable: a [✓ b] c d

Double Precision Examples

Actual bits shown in black, implied '1' bit shown in red

$$1.0 = \underset{s\,=\,0}{\uparrow 0}\ \ \underbrace{01111111111}_{e\,=\,1023\,-\,1023\,=\,0}\ \ \underbrace{1.\ 0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000}_{q\,=\,1.0}\ \Rightarrow\ 1.0=(-1)^0\cdot 1.0\cdot 2^0$$

$$-1.0 = \underset{s\,=\,1}{\uparrow 1}\ \ \underbrace{01111111111}_{e\,=\,1023\,-\,1023\,=\,0}\ \ \underbrace{1.\ 0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000}_{q\,=\,1.0}\ \Rightarrow\ -1.0=(-1)^1\cdot 1.0\cdot 2^0$$

$$2.0 = \underset{s\,=\,0}{\uparrow 0}\ \ \underbrace{10000000000}_{e\,=\,1024\,-\,1023\,=\,1}\ \ \underbrace{1.\ 0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000}_{q\,=\,1.0}\ \Rightarrow\ 2.0=(-1)^0\cdot 1.0\cdot 2^1$$

Double Precision Examples

Actual bits shown in black, implied '1' bit shown in red

$$0.0 = \underbrace{0}_{s\,=\,0}\ \ \underbrace{00000000000}_{}\ \ \underbrace{1.\,0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000}_{}$$

s = 0 all actual bits 0 $\Rightarrow$ 0.0 by convention

$$10.0 = \underbrace{0}_{s\,=\,0}\ \ \underbrace{10000000010}_{e\,=\,1026\,-\,1023\,=\,3}\ \ \underbrace{1.\,0100\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000}_{q\,=\,1.25}$$

$$\Rightarrow\quad 10.0=(-1)^0\cdot 1.25\cdot 2^3$$

$$0.1 = \underbrace{0}_{s\,=\,0}\ \ \underbrace{01111111011}_{e\,=\,1019\,-\,1023\,=\,-4}\ \ \underbrace{1.\,1001\ 10011001\ 10011001\ 10011001\ 10011001\ 10011001\ 10011010}_{q\approx 1.6}$$

$$\Rightarrow\quad 0.1\approx(-1)^0\cdot 1.6\cdot 2^{-4}$$

Double Precision Examples

Actual bits shown in black, implied '1' bit shown in red

$$0.5 = \underbrace{0}_{s\,=\,0}\quad \underbrace{01111111110}_{e\,=\,1022\,-\,1023\,=\,-1}\quad \underbrace{1.\ 0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000}_{c\,=\,1.0} \Rightarrow 0.5=(-1)^{0}\cdot 1.0\cdot 2^{-1}$$

$$0.25 = \underbrace{0}_{s\,=\,0}\quad \underbrace{01111111101}_{e\,=\,1021\,-\,1023\,=\,-2}\quad \underbrace{1.\ 0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000}_{c\,=\,1.0} \Rightarrow 0.25=(-1)^{0}\cdot 1.0\cdot 2^{-2}$$

1.0000000000000002 =

$$\underbrace{0}_{s\,=\,0}\quad \underbrace{01111111111}_{e\,=\,1023\,-\,1023\,=\,0}\quad \underbrace{1.\ 0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001}_{c\,=\,1.0000000000000002}$$

$$\Rightarrow 1.0000000000000002=(-1)^{0}\cdot 1.0000000000000002\cdot 2^{0}$$

Double Precision Examples

Actual bits shown in black, implied '1' bit shown in red

Largest possible number

$$= 0\;\; \underbrace{11111111110}_{e\,=\,2046\,-\,1023\,=\,1023}\;\; \underbrace{1.\ 1111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111}_{c\,\approx\,2.0}$$

$s = 0$ $\qquad e = 2046 - 1023 = 1023 \qquad c \approx 2.0$

$$\Rightarrow \quad (-1)^0 \cdot 2.0 \cdot 2^{1023} \approx 1.8 \cdot 10^{308}$$

Example: The known size of the universe is $9.1{\times}10^{10}$ light years = $8.6{\times}10^{26}$ m = $8.6{\times}10^{36}$ Å

Smallest possible number, excluding subnormal numbers

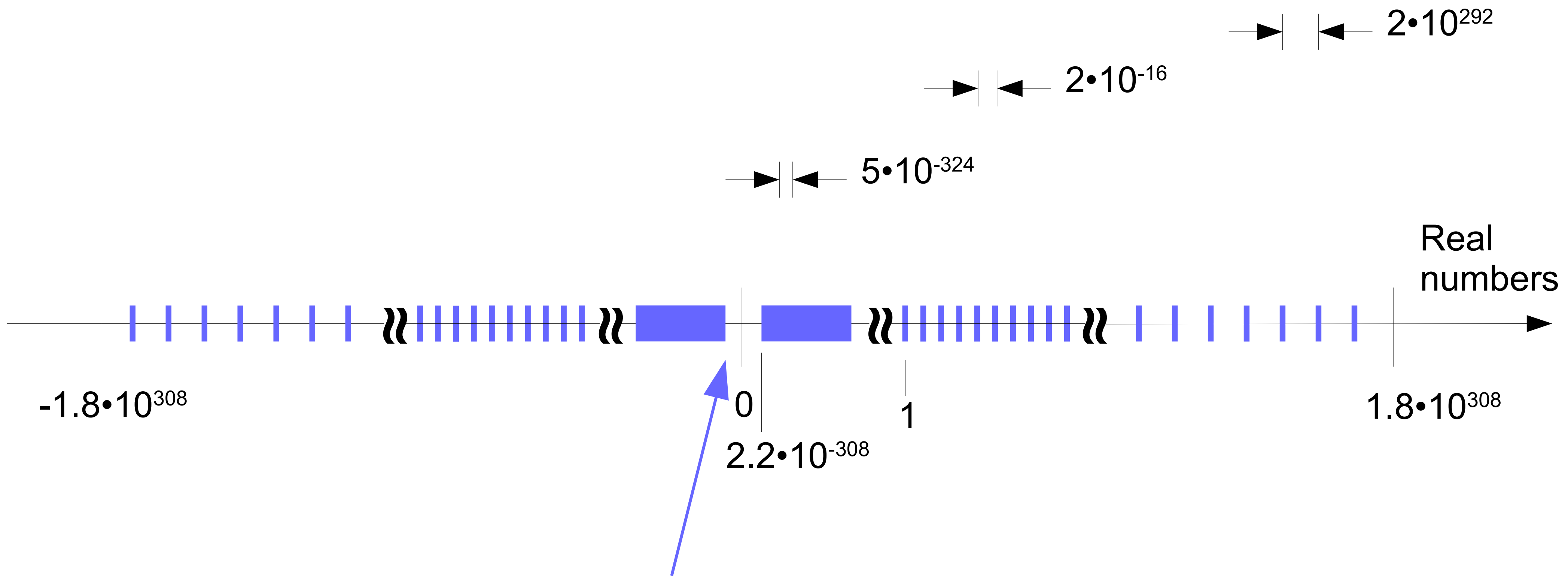
$$= 0\;\; \underbrace{00000000001}_{e\,=\,1\,-\,1023\,=\,-1022}\;\; \underbrace{1.\ 0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000001}_{c\,\approx\,1.0}$$

$s = 0$ $\qquad e = 1 - 1023 = -1022 \qquad c \approx 1.0$

$$\Rightarrow \quad (-1)^0 \cdot 1.0 \cdot 2^{-1022} \approx 2.2 \cdot 10^{-308}$$

Density on Number Line

For double precision floating point variables, the density of numbers represented is nonuniform between the minimum and maximum numbers



"Gap" around zero usually filled in somewhat with "subnormal" numbers

(Not to scale)

Pseudocode for Exploring Precision Limit

Mapping a finite number of bit states to an infinitely divisible real number line will clearly leave “gaps” in the line. This is an intrinsic limit of machine precision.

```
integer $i$; real $x, y, z$;
$x \leftarrow 1.0$
$y \leftarrow 0.125$
for $i = 1$ to $20$ do
    $z \leftarrow x + y$
    output $x, y, z$
    $y \leftarrow y \cdot 0.125$
end for
```

C Code Output Using Double Precision

```
x=1.000000000000000 y=1.250000000000000e-01 z=1.125000000000000
x=1.000000000000000 y=1.562500000000000e-02 z=1.015625000000000
x=1.000000000000000 y=1.953125000000000e-03 z=1.001953125000000
x=1.000000000000000 y=2.441406250000000e-04 z=1.000244140625000
x=1.000000000000000 y=3.051757812500000e-05 z=1.000030517578125
x=1.000000000000000 y=3.814697265625000e-06 z=1.000003814697266
x=1.000000000000000 y=4.768371582031250e-07 z=1.000000476837158
x=1.000000000000000 y=5.960464477539062e-08 z=1.000000059604645
x=1.000000000000000 y=7.450580596923828e-09 z=1.000000007450581
x=1.000000000000000 y=9.313225746154785e-10 z=1.000000000931323
x=1.000000000000000 y=1.164153218269348e-10 z=1.000000000116415
x=1.000000000000000 y=1.455191522836685e-11 z=1.000000000014552
x=1.000000000000000 y=1.818989403545856e-12 z=1.000000000001819
x=1.000000000000000 y=2.273736754432321e-13 z=1.000000000000227
x=1.000000000000000 y=2.842170943040401e-14 z=1.000000000000028
x=1.000000000000000 y=3.552713678800501e-15 z=1.000000000000004
x=1.000000000000000 y=4.440892098500626e-16 z=1.000000000000000
x=1.000000000000000 y=5.551115123125783e-17 z=1.000000000000000
x=1.000000000000000 y=6.938893903907228e-18 z=1.000000000000000
x=1.000000000000000 y=8.673617379884035e-19 z=1.000000000000000
```

C Code Output Using Single Precision

```
x=1.00000000 y=1.25000000e-01 z=1.12500000
x=1.00000000 y=1.56250000e-02 z=1.01562500
x=1.00000000 y=1.95312500e-03 z=1.00195312
x=1.00000000 y=2.44140625e-04 z=1.00024414
x=1.00000000 y=3.05175781e-05 z=1.00003052
x=1.00000000 y=3.81469727e-06 z=1.00000381
x=1.00000000 y=4.76837158e-07 z=1.00000048
x=1.00000000 y=5.96046448e-08 z=1.00000000
x=1.00000000 y=7.45058060e-09 z=1.00000000
x=1.00000000 y=9.31322575e-10 z=1.00000000
x=1.00000000 y=1.16415322e-10 z=1.00000000
x=1.00000000 y=1.45519152e-11 z=1.00000000
x=1.00000000 y=1.81898940e-12 z=1.00000000
x=1.00000000 y=2.27373675e-13 z=1.00000000
x=1.00000000 y=2.84217094e-14 z=1.00000000
x=1.00000000 y=3.55271368e-15 z=1.00000000
x=1.00000000 y=4.44089210e-16 z=1.00000000
x=1.00000000 y=5.55111512e-17 z=1.00000000
x=1.00000000 y=6.93889390e-18 z=1.00000000
x=1.00000000 y=8.67361738e-19 z=1.00000000
```

Machine ε

The "Machine ε" is the smallest number ε such that $1 + \varepsilon \neq 1$

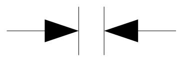
For 32-bit float data type, $\varepsilon = 2^{-24} \approx 5.96\times10^{-8}$
A number slightly larger than this ε will be rounded up to 2^{-23} when added to 1.0 and turn the least significant bit to '1'

For 64-bit double data type, $\varepsilon = 2^{-53} \approx 1.11\times10^{-16}$
A number slightly larger than this ε will be rounded up to 2^{-52} when added to 1.0 and turn the least significant bit to '1'

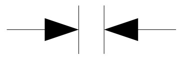
Note: ε is defined relative to 1. If the scale of our numbers for a particular computation range up or down, the minimum number we can add to change a value will also scale up or down. For example, for double precision, the minimum value we can add to $2^{-300} \approx 4.91\times10^{-91}$ is $2^{-353} \approx 5.45\times10^{-107}$, or the minimum value we can add to $2^{+300} \approx 2.04\times10^{90}$ is $2^{247} \approx 2.26\times10^{74}$

Machine ε on Number Line

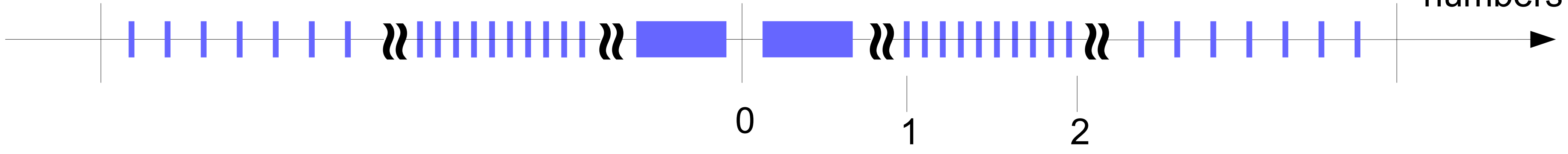
The Machine ε is half the distance between allowed values between 1 and 2



1.192•10^{-7} for single precision, so
ε = 5.96×10^{-8}

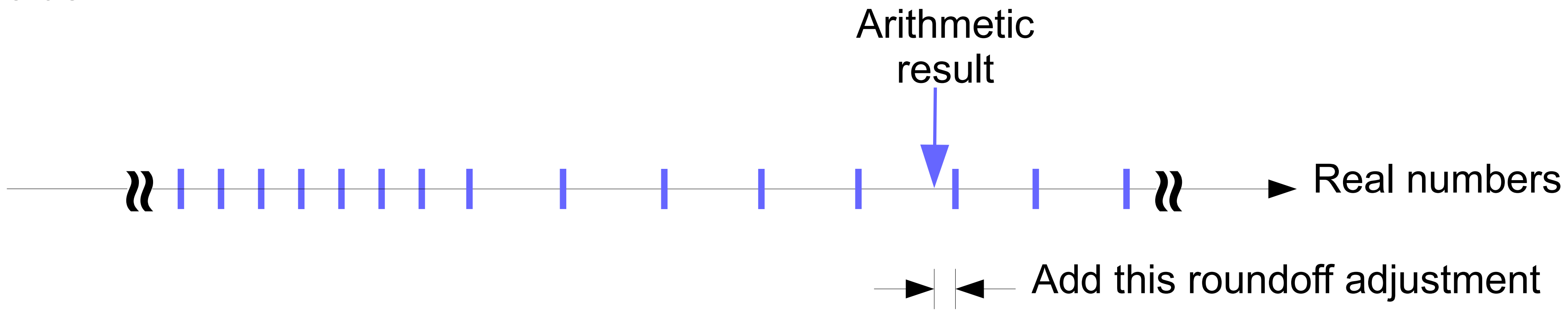


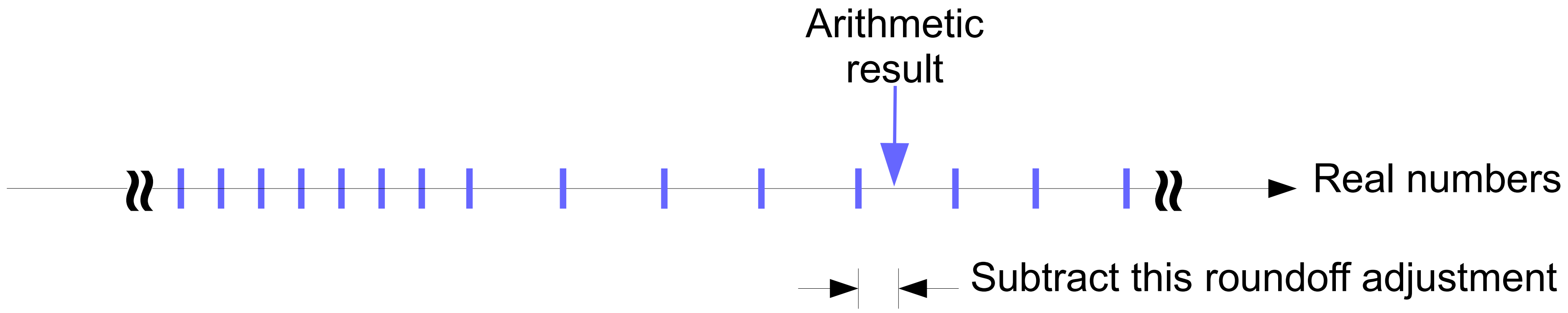
2.22•10^{-16} for double precision, so
ε = 1.11•10^{-16}



Roundoff Error

When the result of an arithmetic operation lies between two allowed values on the real number line, "rounding off" to some allowed value before the value is stored in memory is equivalent to adding or subtracting some roundoff adjustment to end up at an allowed value





The accumulated roundoff error resulting at the end of a large number of arithmetic operations can be treated statistically

Pseudocode for Accumulating Roundoff Error

Use 80-bit long doubles for these reals

Use 64-bit doubles for these reals

Use 32-bit floats for this real

```
integer $i$; real $r, xld, yld$; real $xd, yd$; real $xf, yf$;
$yld \leftarrow 0.0$
$yd \leftarrow 0.0$
$yf \leftarrow 0.0$
$r \leftarrow 1.0$
for $i=1$ to $10000$ do
    $r \leftarrow (16807.0 \cdot r) \% 2147483647.0$
    $xld \leftarrow \frac{r}{2147483647.0} + 1.0$
    $xd \leftarrow xld$
    $xf \leftarrow xld$
    $yld \leftarrow yld + xld$
    $yd \leftarrow yd + xd$
    $yf \leftarrow yf + xf$
    $errorxd \leftarrow xd - xld$
    $errorxf \leftarrow xf - xld$
    $erroryd \leftarrow yd - yld$
    $erroryf \leftarrow yf - yld$
    output $i$, $error$
end for
```

An example computation for some solution algorithm (chosen here for a uniform distribution of *xld* values between 1.0 and 2.0)

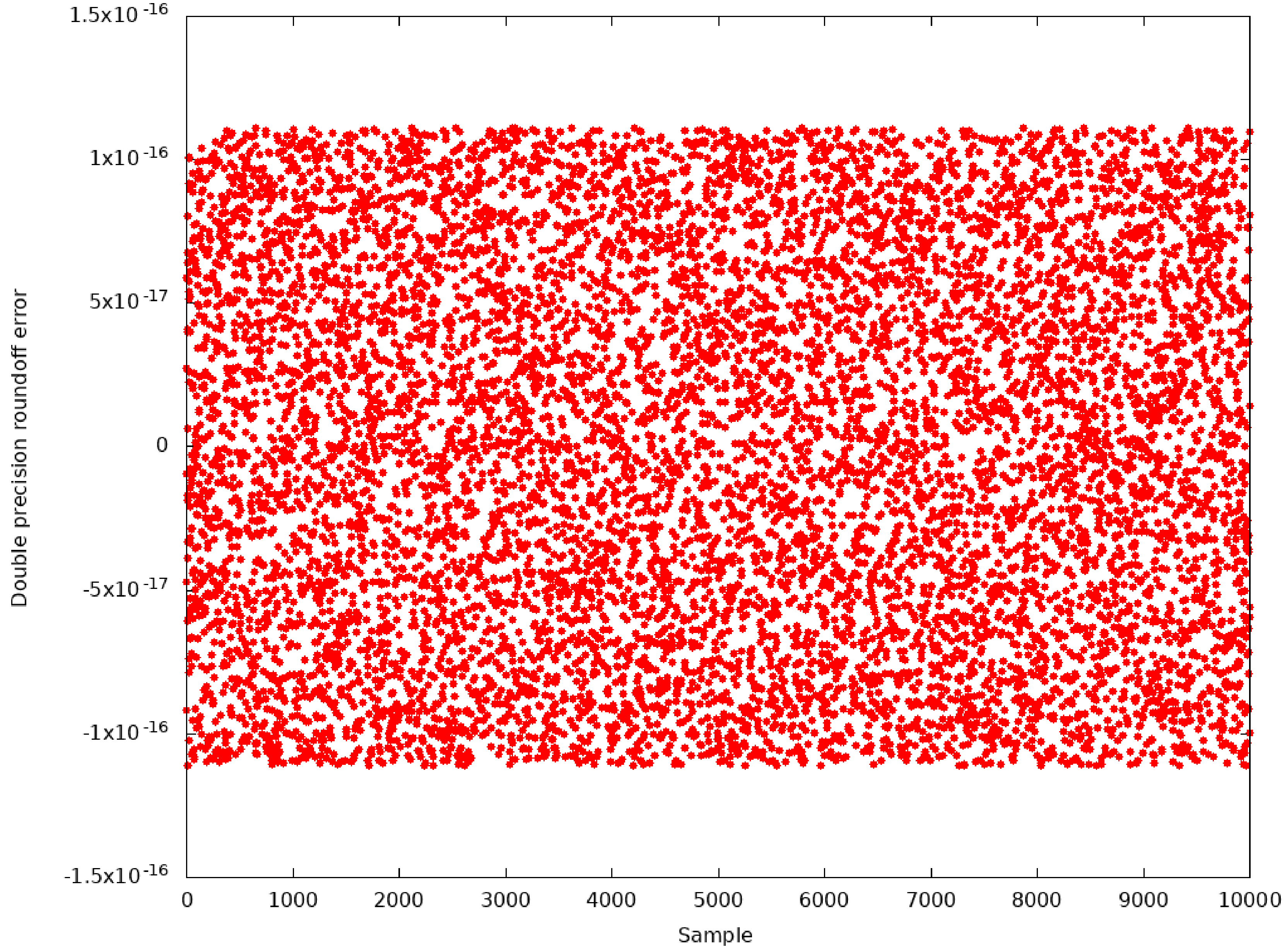
Assign long double to both float and double

Accumulate sum of pseudorandom numbers in both float and double

Interpret differences as round off error for doubles and floats

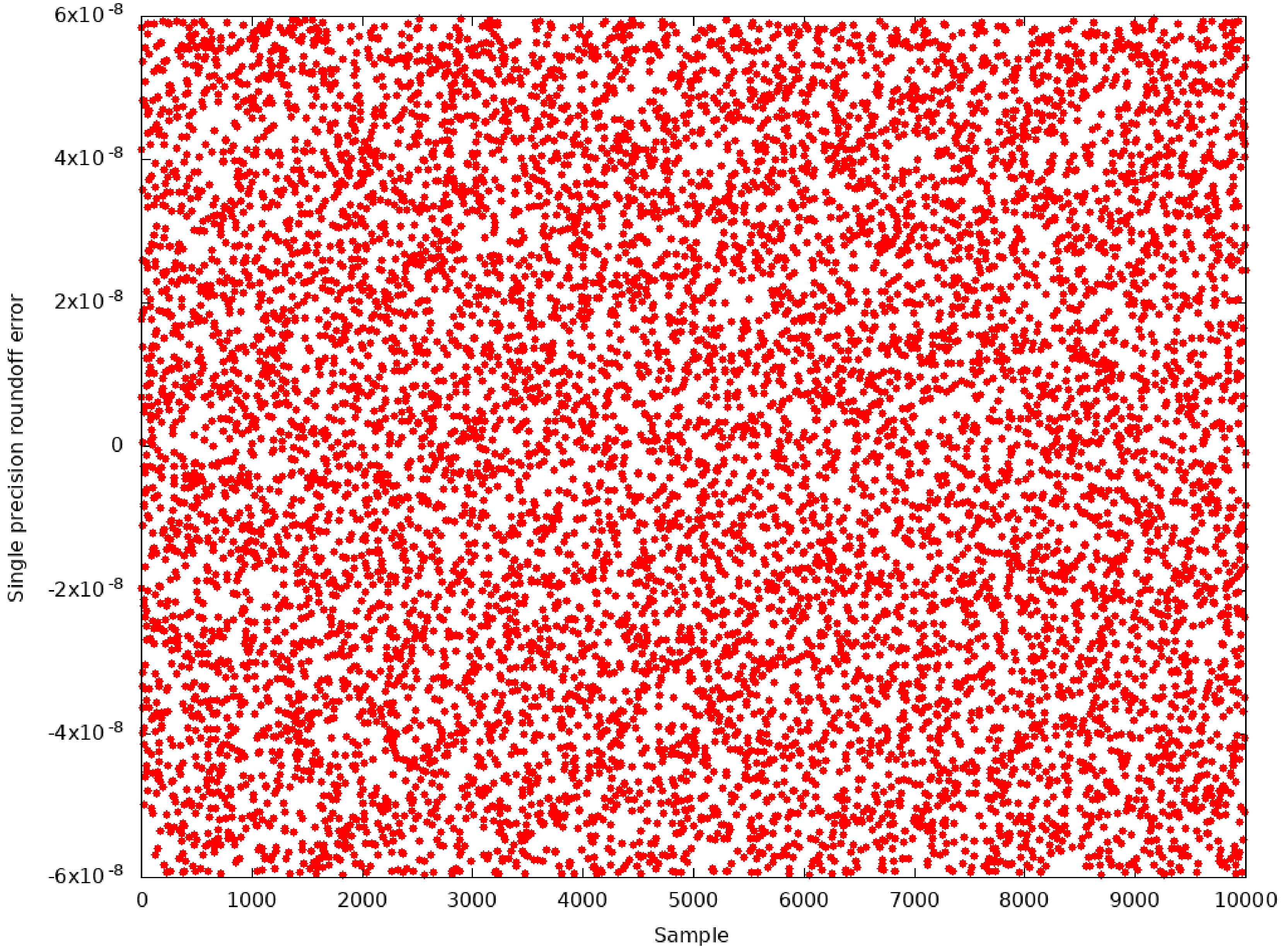
Round off error from each assignment to double

Appears like uniform random variable between -ε and +ε



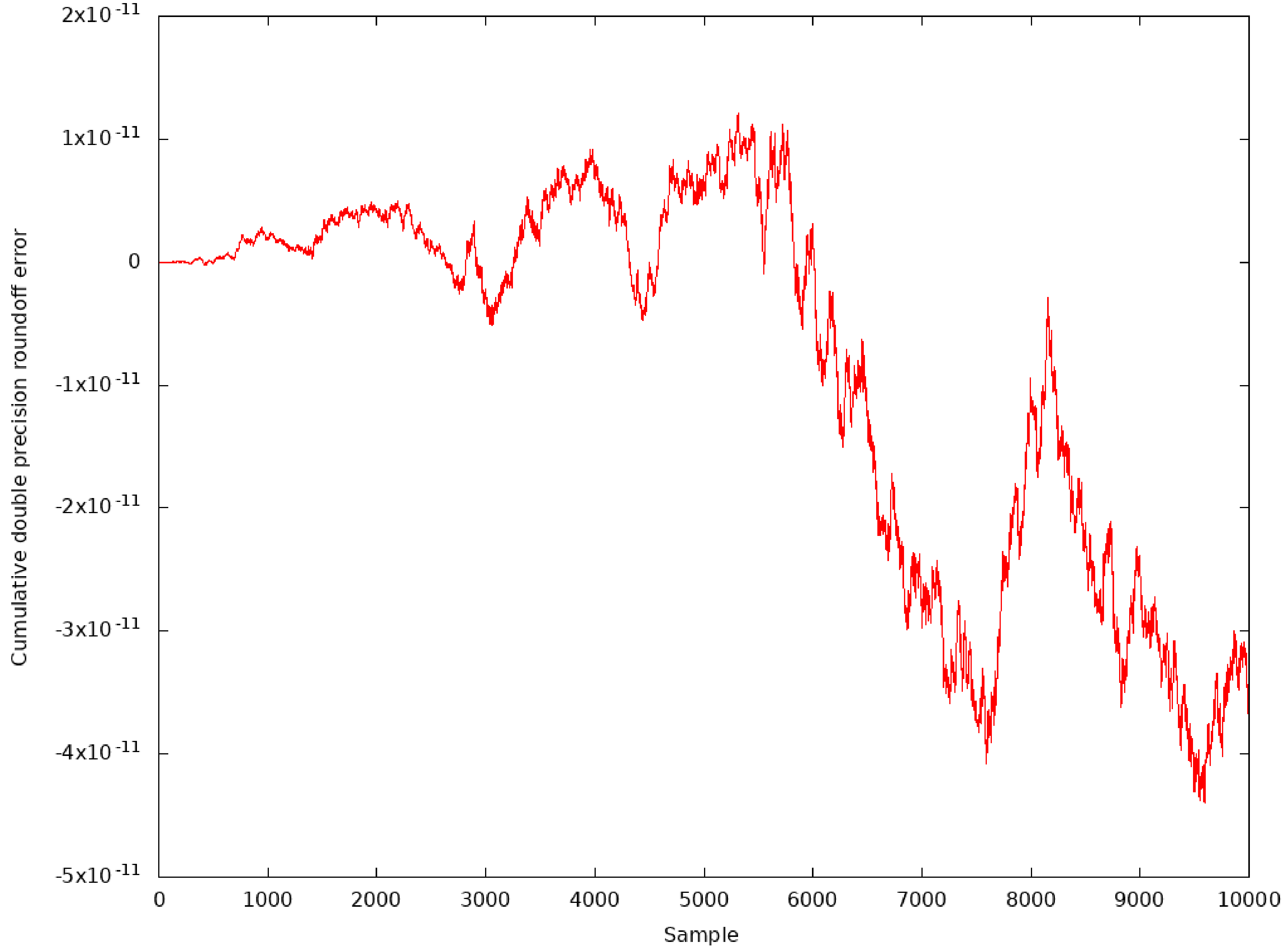
Round off error from each assignment to float

Appears like uniform random variable between -ε and +ε



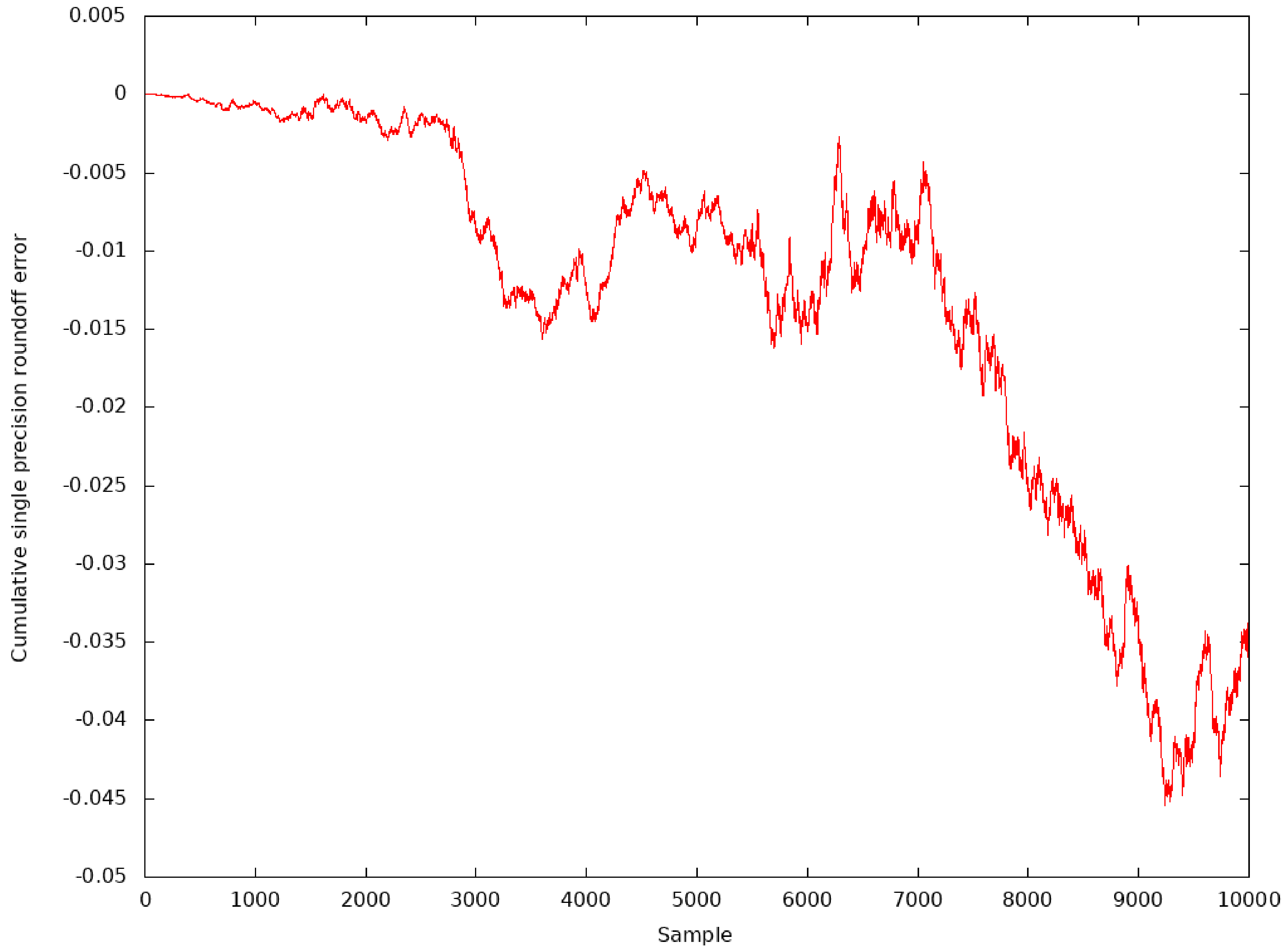
Accumulated round off error for doubles

Accumulated roundoff can get to large numbers relative to ε as the scale of the sum gets large



Accumulated round off error for floats

Accumulated roundoff can get to large numbers relative to ε as the scale of the sum gets large



Evaluation of Polynomial with Multiple Roots

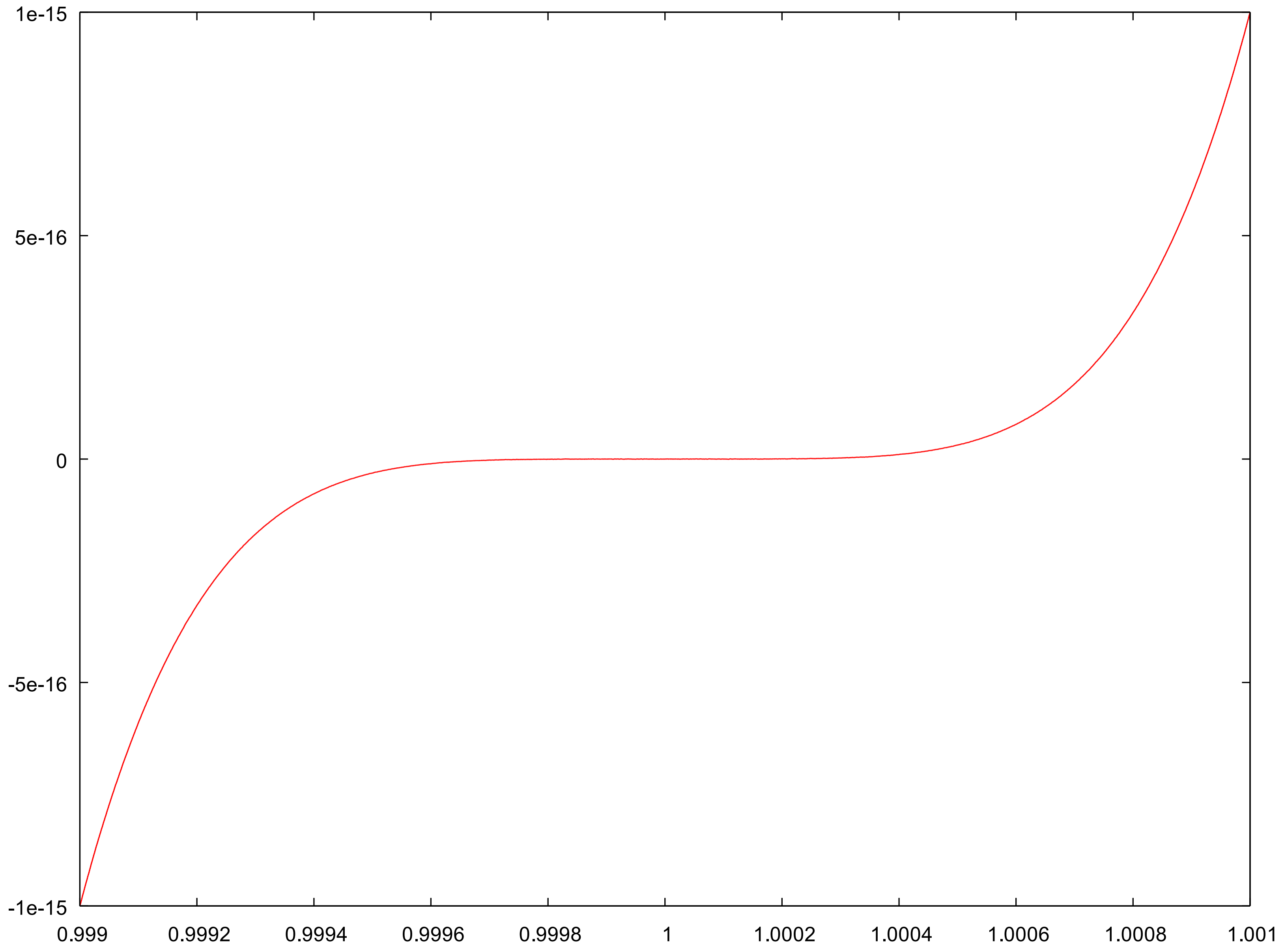
$$y = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$$

Has root of multiplicity 5 at $x = 1$

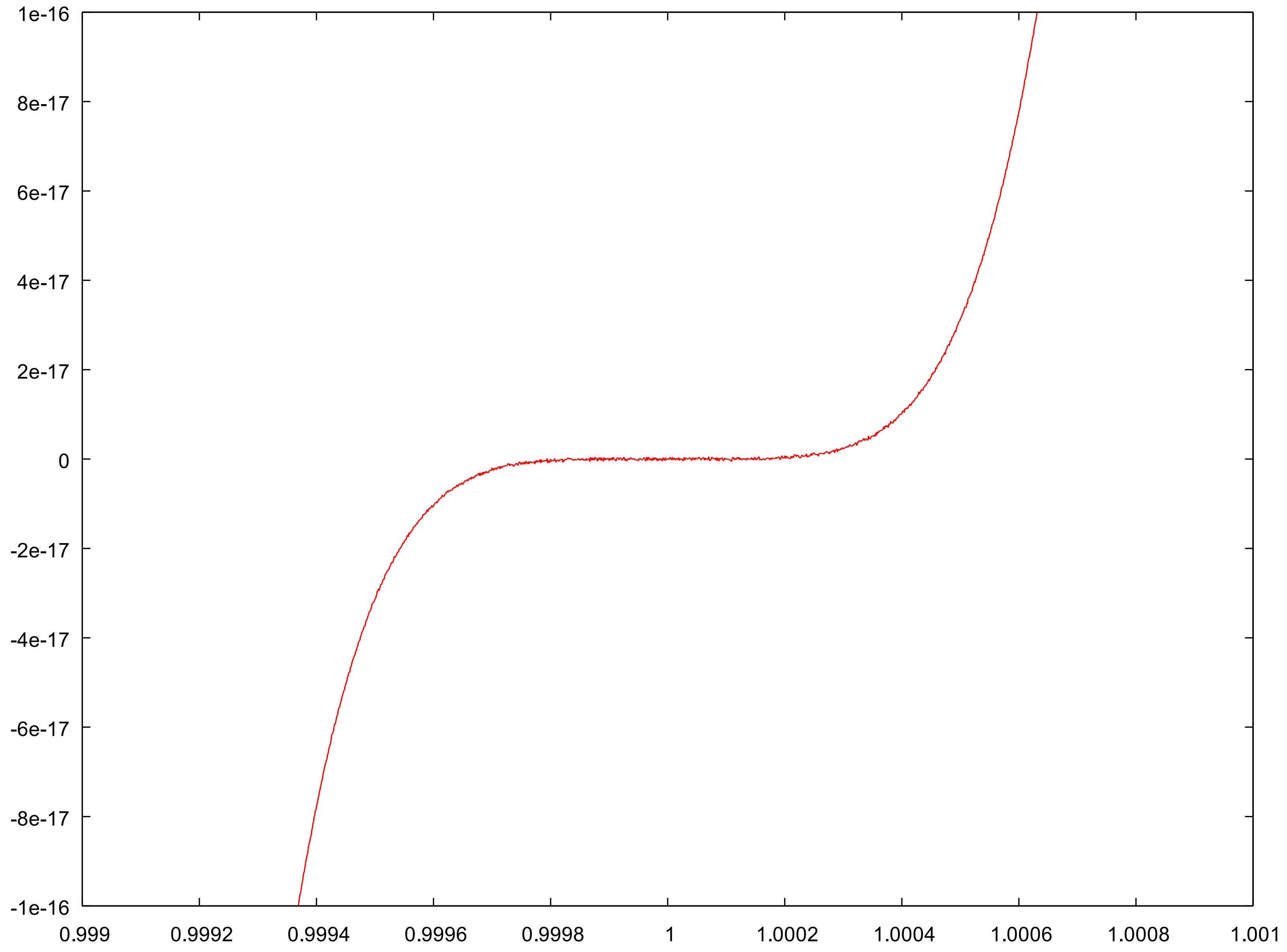
Of course, we actually evaluate this as

$$y = -1 + x \cdot (5 + x \cdot (-10 + x \cdot (10 + x \cdot (-5 + x))))$$

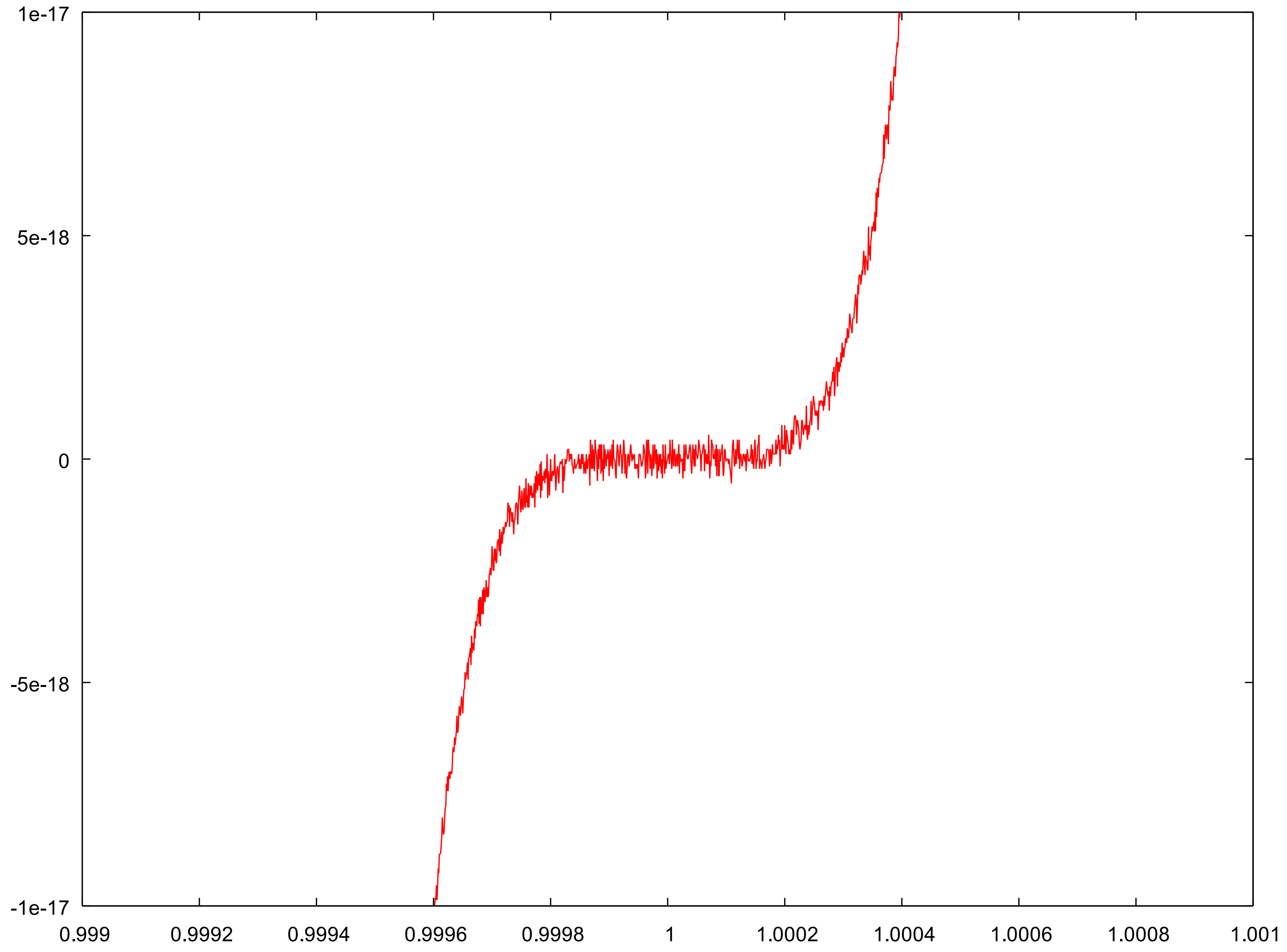
Region between x=0.999 and x=1.001



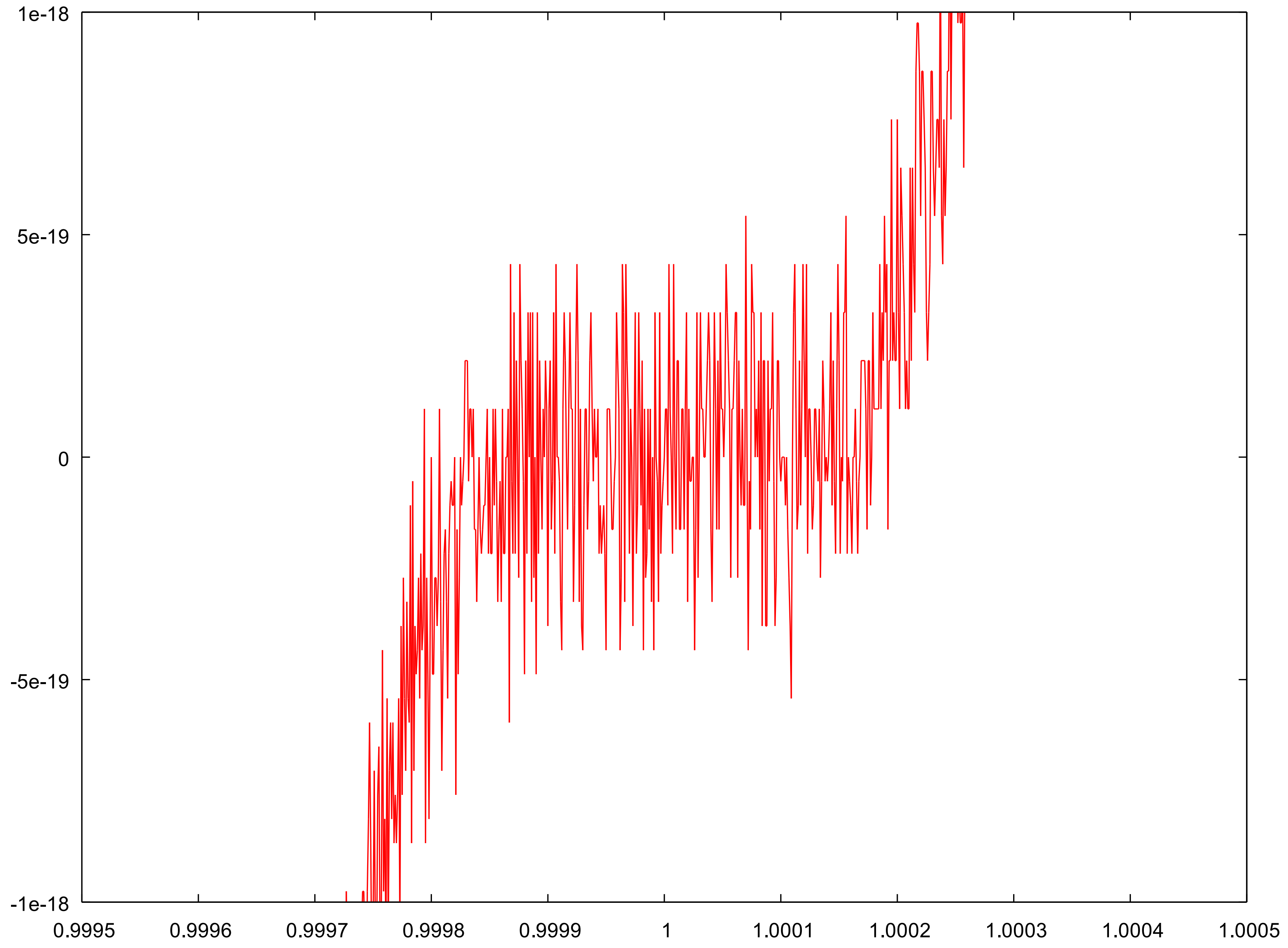
Zooming in on y-axis



Zooming in on y-axis



Zooming in on x-axis and y-axis



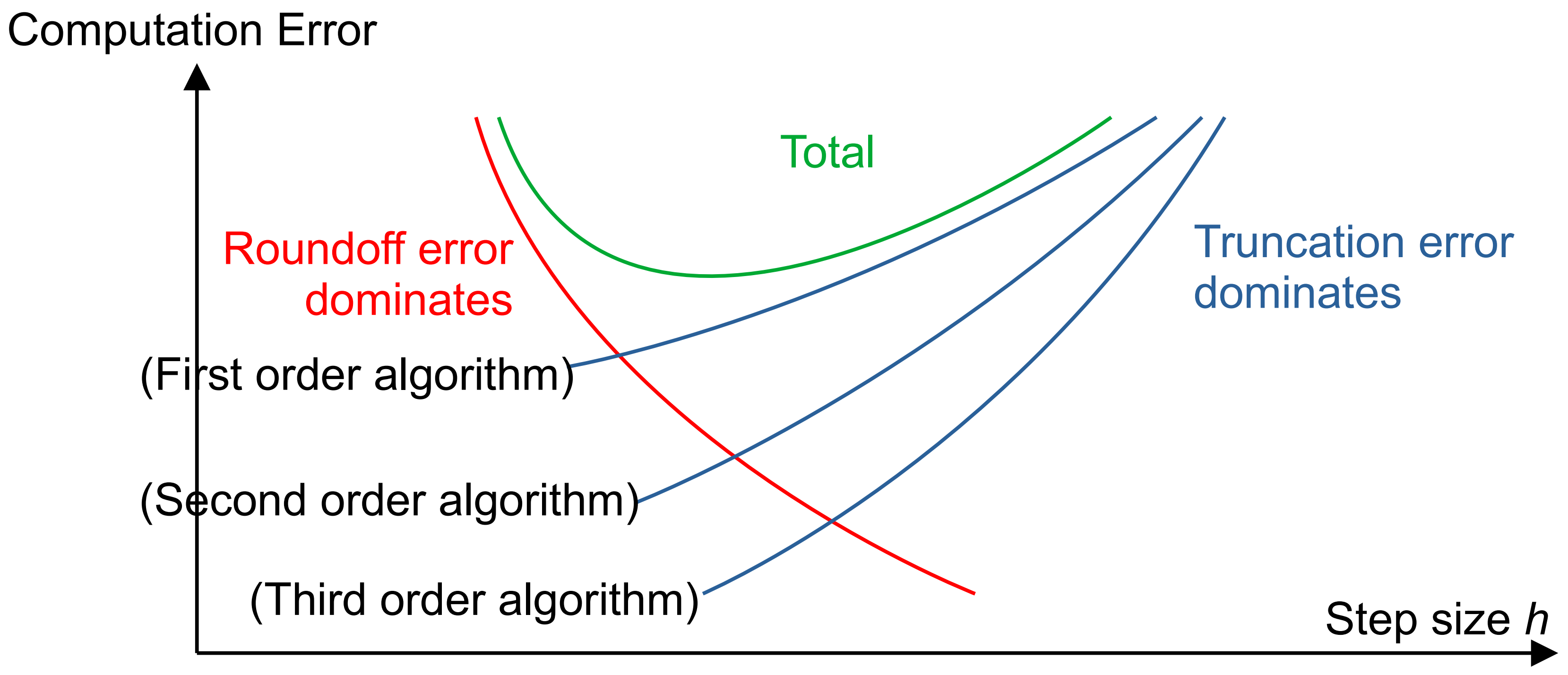
Raw Data from C Code

```
...
0.9999 -3.7947076e-19
0.999901 1.0842022e-19
0.999902 2.1684043e-19
0.999903 -1.6263033e-19
0.999904 -5.4210109e-20
0.999905 3.2526065e-19
0.999906 -2.1684043e-19
0.999907 4.3368087e-19
0.999908 0
0.999909 0
0.99991 -5.4210109e-20
0.999911 -3.2526065e-19
0.999912 -4.3368087e-19
0.999913 1.0842022e-19
0.999914 3.2526065e-19
0.999915 2.1684043e-19
0.999916 0
0.999917 -1.6263033e-19
0.999918 1.0842022e-19
0.999919 3.2526065e-19
0.99992 1.0842022e-19
...
```

Numerical “noise” from accumulated roundoff errors

Tradeoff Between Roundoff and Truncation Errors

For numerical methods involving a truncated Taylor series approximations to a function: Root finding, integration, differentiation, interpolation, differential equations, etc.



Example: Approximate first derivative with single sided first order ratio:

$$\frac{df}{dx}(x) \approx \frac{f(x+h)-f(x)}{h}$$

Truncation error introduced by the curvature of the function *f*(*x*) when the step size *h* is large.
Roundoff error introduced by subtracting two very similar numbers in numerator when *h* is small.

Programming has Sand Traps

Programming language manuals and textbooks describe the smooth fairways and greens, but what are the sand traps?

Beware of Mixing Data Types in Expressions!

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int i,j,k,m;
  double x,y,z;

  i = 7;
  j = 4;
  k = i / j;
  m = i % j;
  printf("k=%d m=%d\n",k,m);
  i = 1;
  j = 10;
  k = i / j;
  m = i % j;
  printf("k=%d m=%d\n",k,m);
  x = 1.0;
  y = x + 1.0 / 10.0;
  z = x + 1 / 10;
  printf("y=%.2f z=%.2f\n",y,z);
  x = 1.0;
  y = x + 0.1;
  z = x + i / j;
  printf("y=%.2f z=%.2f\n",y,z);
  exit(0);
}
```

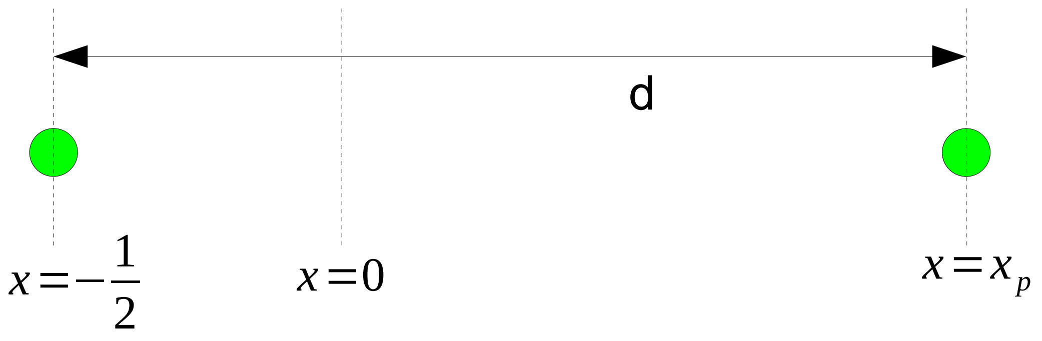
Integer division results in whole number quotients and remainders

Mixing constants or expressions of integer type into floating point expressions can lead to numerical errors!

Output:

```
k=1 m=3
k=0 m=1
y=1.10 z=1.00
y=1.10 z=1.00
```

From System to Pseudocode to C Code



Pseudocode:

```
real $x, x_p, d$
$x_p \leftarrow \cdots$
$\vdots$
$d \leftarrow x_p + \frac{1}{2}$
```

C code:

```
double x,xp,d;

d = xp + (1/2);         ←   Bug!
d = xp + (1.0/2.0);     ←   Fixed!
d = xp + 0.5;           ←   Compiled as
```

Integer Constants in Floating Point Expressions

```
#include <stdio.h>
#include <stdlib.h>

static double x,y;

int main(int argc,char *argv[]) {
  x = 5.0;
  y = x + 1/2;
  printf("x=%.2f y=%.2f\n",x,y);
  y = x + 0.0;
  printf("x=%.2f y=%.2f\n",x,y);
  y = x + 1.0/2.0;
  printf("x=%.2f y=%.2f\n",x,y);
  y = x + 0.5;
  printf("x=%.2f y=%.2f\n",x,y);
  exit(0);
}
```

```
Output:

x=5.00 y=5.00
x=5.00 y=5.00
x=5.00 y=5.50
x=5.00 y=5.50
```

Integer Constants in Floating Point Expressions

Look at compiled binary program with objdump tool:

```
...
  x = 5.0;
  11:   dd 05 10 00 00 00        fld    QWORD PTR ds:0x10
  17:   dd 1d 00 00 00 00        fstp   QWORD PTR ds:0x0
  y = x + 1/2;
  1d:   dd 05 00 00 00 00        fld    QWORD PTR ds:0x0
  23:   d9 ee                    fldz
  25:   de c1                    faddp  st(1),st
  27:   dd 1d 08 00 00 00        fstp   QWORD PTR ds:0x8
...
  y = x + 0.0;
  4f:   dd 05 00 00 00 00        fld    QWORD PTR ds:0x0
  55:   d9 ee                    fldz
  57:   de c1                    faddp  st(1),st
  59:   dd 1d 08 00 00 00        fstp   QWORD PTR ds:0x8
...
  y = x + 1.0/2.0;
  81:   dd 05 00 00 00 00        fld    QWORD PTR ds:0x0
  87:   dd 05 18 00 00 00        fld    QWORD PTR ds:0x18
  8d:   de c1                    faddp  st(1),st
  8f:   dd 1d 08 00 00 00        fstp   QWORD PTR ds:0x8
...
  y = x + 0.5;
  b7:   dd 05 00 00 00 00        fld    QWORD PTR ds:0x0
  bd:   dd 05 18 00 00 00        fld    QWORD PTR ds:0x18
  c3:   de c1                    faddp  st(1),st
  c5:   dd 1d 08 00 00 00        fstp   QWORD PTR ds:0x8
```

5.0 stored at hex 10 = 16

x stored at hex 00 = 0

Floating point load 0! Bug!

y stored at hex 08 = 8

Load floating point 0

0.5 stored at hex 18 = 24

fld=floating point load stack
fldz=load stack with zero
faddp=add and pop stack
fstp=store and pop stack

Actual Lines of Code from Homework

Probably will get away with it, but poor form!

```
double a,b,x;
x = x*b*(1+a)/(1+b);
```

The slippery slope downhill!! This will work, but only through a compiler's "promotion" rules.

```
double a,n = 1;
e_approx = pow((1 + 1/n),n);
a = 0.5*(t + 1/t);
```

```
 x = 2 + pow(2,1/2);
```

Potential disaster!!! What gets assigned to `x`? Not what the programmer intended!!!

```
 y = 1/3*cos(1.0*x-1.0*sin(x));
```

Totally trashes the computation!!!

My Coding Practice for Arithmetic Expressions

- Decide if an arithmetic assignment statement is to produce an integer or floating point result
- If floating point, code all constants in the expressions as floating point, that is, including a decimal point or exponent

Intentionally Mixing Data Types

Example: Compute mean of floating point values:

$$\mu=\frac{1}{n}\sum_{i=0}^{n-1} x_i$$

```
int i,n;
double sum,mean;

n = ...;

i = 0;
sum = 0.0;
while (i < n) {
   sum = sum + ...;
   i++;
}
mean = sum / ((double) n);
```

Integer variable holds count of values for loop control

Option 1: Use “cast” operator to cast argument as a different type in a documented way
integer n is cast as a double to match type of division expression

Intentionally Mixing Data Types

Example: Compute mean of floating point values:

```
int i,n;
double sum,mean,n_double;

n = ...;
n_double = n;

i = 0;
sum = 0.0;
while (i < n) {
   sum = sum + ...;
   i++;
}
mean = sum / n_double;
```

Integer variable holds count of values for loop control

Option 2: Use assignment statement to convert type across assignment operator
n_double is already of type double to match type of division expression

Be Careful Converting double to int!

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  double x,y,z;
  int i,j;

  x = 10.0;
  y = 1.0e-18;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  y = 1.0e-17;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  y = 1.0e-16;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  y = 1.0e-15;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  y = 1.0e-14;
  z = x - y;  i = z;  j = z + 0.5;
  printf("y=%.3g z=%20.16f i=%2d j=%2d\n",y,z,i,j);
  exit(0);
}
```

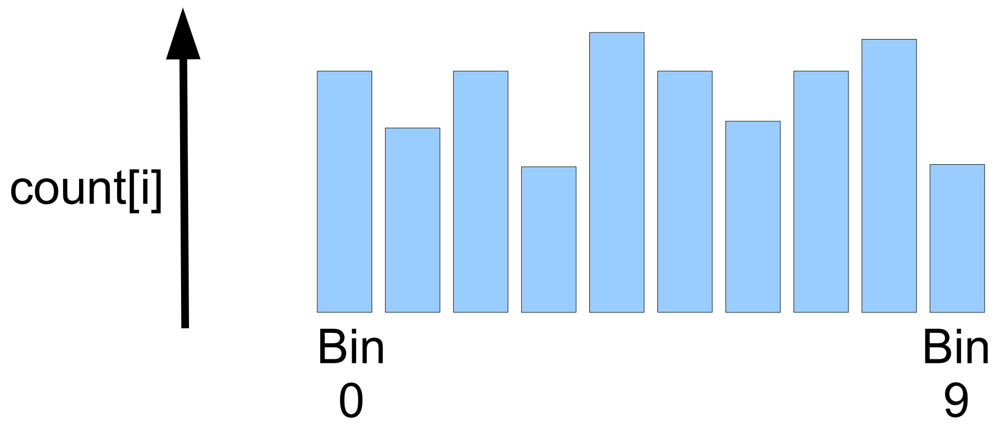
Conversion of double to int across assignment operator gives a chop operation, not a round! Explicitly add 0.5 before conversion to get a round operation!

Output:

```
y=1e-18 z= 10.0000000000000000 i=10 j=10
y=1e-17 z= 10.0000000000000000 i=10 j=10
y=1e-16 z= 10.0000000000000000 i=10 j=10
y=1e-15 z=  9.9999999999999982 i= 9 j=10
y=1e-14 z=  9.9999999999999893 i= 9 j=10
```

Intentionally Mixing Data Types

Example: For a histogram, increment bin count corresponding to a floating point value:



```
int count[10];
double bin,value;

bin = value / 10.0;

count[(int) (bin + 0.5)]++;
```

Floating point variable holds a bin index corresponding to a floating point value

Option 1: Use “cast” operator to cast argument as a different type in a documented way
double bin is cast as an int to use as an array index
note adding 0.5 to turn conversion to int, by default a 'floor' operation, into a round operation

Intentionally Mixing Data Types

Example: For a histogram, increment bin count corresponding to a floating point value:

```
int i_bin,count[10];
double bin,value;

bin = value / 10.0;

i_bin = bin + 0.5;
count[i_bin]++;
```

Floating point variable holds a bin index corresponding to a floating point value

Option 2: Use assignment statement to convert type across assignment operator
double bin is cast as an int to use as an array index
note adding 0.5 to turn conversion to int, by default a 'floor' operation, into a round operation

Be Careful What You Cast To!

```
double w,x,y,z;

w = 1 / 3;
x = 1 / (float) 3;
y = 1 / (double) 3;
z = 1.0 / 3.0;
printf("w=%.17f x=%.17f y=%.17f z=%.17f\n",w,x,y,z);
```

An integer constant be cast to either a 32-bit single precision “float” or a 64-bit double precision “double”. Does it make a difference?

Output:

```
w=0.00000000000000000 x=0.33333334326744080 y=0.33333333333333331 z=0.33333333333333331
```

Why are we losing all these digits of precision?

Be Careful With Comparison Operators

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,char *argv[]) {
  double x;
  x = 0.0;
  while (x < 1.001) {
    printf("%.2f %d %d %d %d\n",x,0.45 < x,x < 0.65,0.45 < x < 0.65,0.45 < x && x < 0.65);
    x += 0.1;
  }
  exit(0);
}
```

How do we code the algebraic test
$0.45 < x < 0.65$?

What does this compiler warning mean?

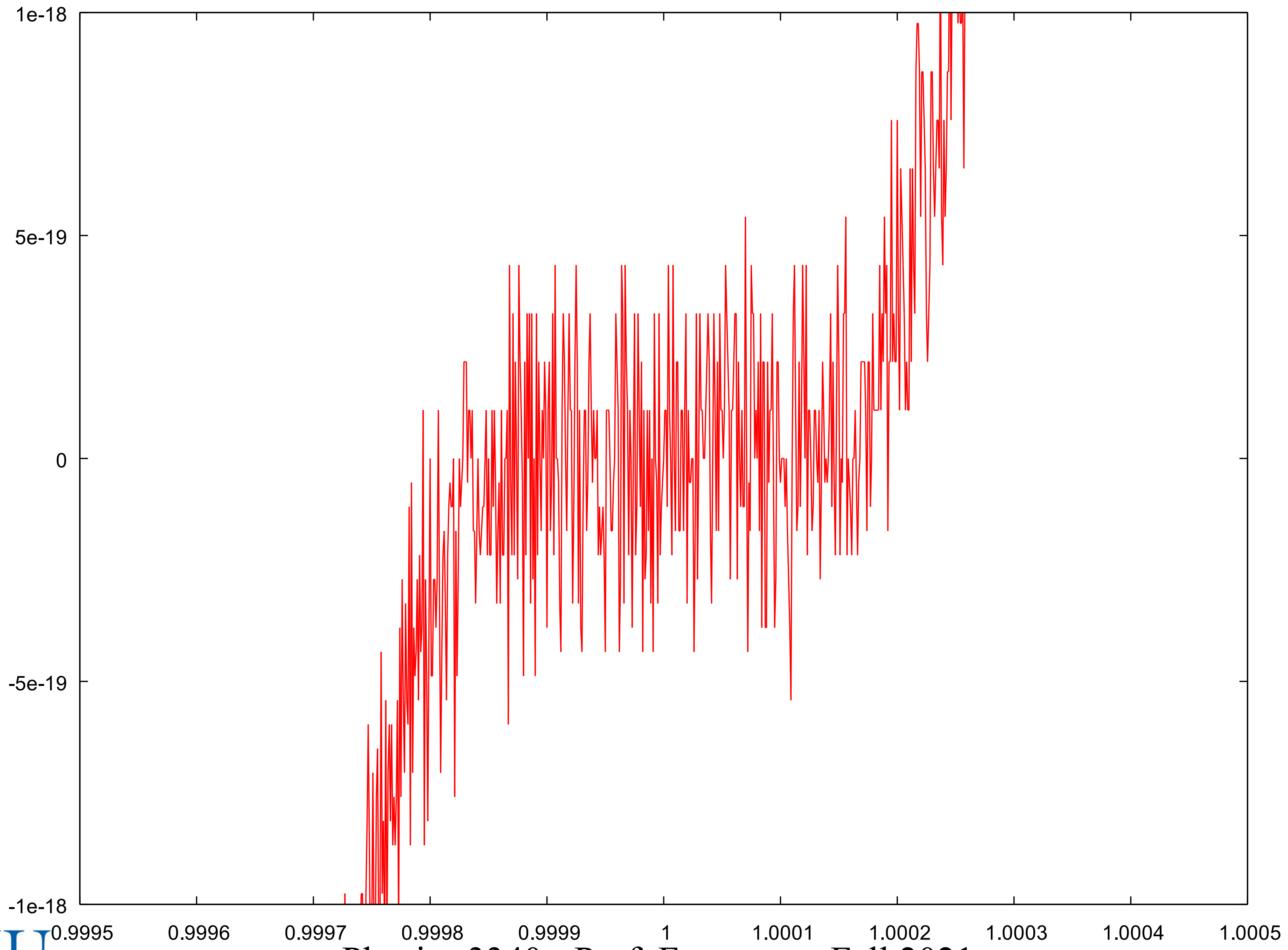
```
gcc -Wall -Wshadow -o test test.c -lm
test.c: In function ‘main’:
test.c:10:58: warning: comparisons like ‘X<=Y<=Z’ do not have their mathematical meaning [-Wparentheses]
     printf("%.2f %d %d %d %d\n",x,0.45 < x,x < 0.65,0.45 < x < 0.65,0.45 < x && x < 0.65);
```

Output:

```
0.00 0 1 1 0
0.10 0 1 1 0
0.20 0 1 1 0
0.30 0 1 1 0
0.40 0 1 1 0
0.50 1 1 0 1
0.60 1 1 0 1
0.70 1 0 0 0
0.80 1 0 0 0
0.90 1 0 0 0
1.00 1 0 0 0
```

Using Floating Point Variables with Comparison Operators

Remember this plot from the lecture on numerical roundoff errors?



Using Floating Point Variables with Comparison Operators

The C '>' greater than test will involve **all 64 bits** of a double precision variable with the exactness of digital logic:

1.0000000000000002 =

0 01111111111 1. 0000 00000000 00000000 00000000 00000000 00000000 00000001

Is greater than

1.0000000000000000 =

0 01111111111 1. 0000 00000000 00000000 00000000 00000000 00000000 00000000

Is greater than

0.99999999999999989=

0 01111111110 1. 1111 11111111 11111111 11111111 11111111 11111111 11111111

The C '==' equality test will not report equality unless **all 64 bits** of double precision variables exactly match! A test with a bit of an uncertainty window to allow for a little roundoff error must be explicitly coded!

Using Floating Point Variables with Comparison Operators

Output:

```
x=0.7000000000000000
x=1.3999999999999999
x=2.0999999999999996
x=2.7999999999999998
x=3.5000000000000000
x=4.2000000000000002
x=4.9000000000000004
x=5.6000000000000005
x=6.3000000000000007
x=7.0000000000000009
x is greater than 7.0
```

The expected value is 7.0

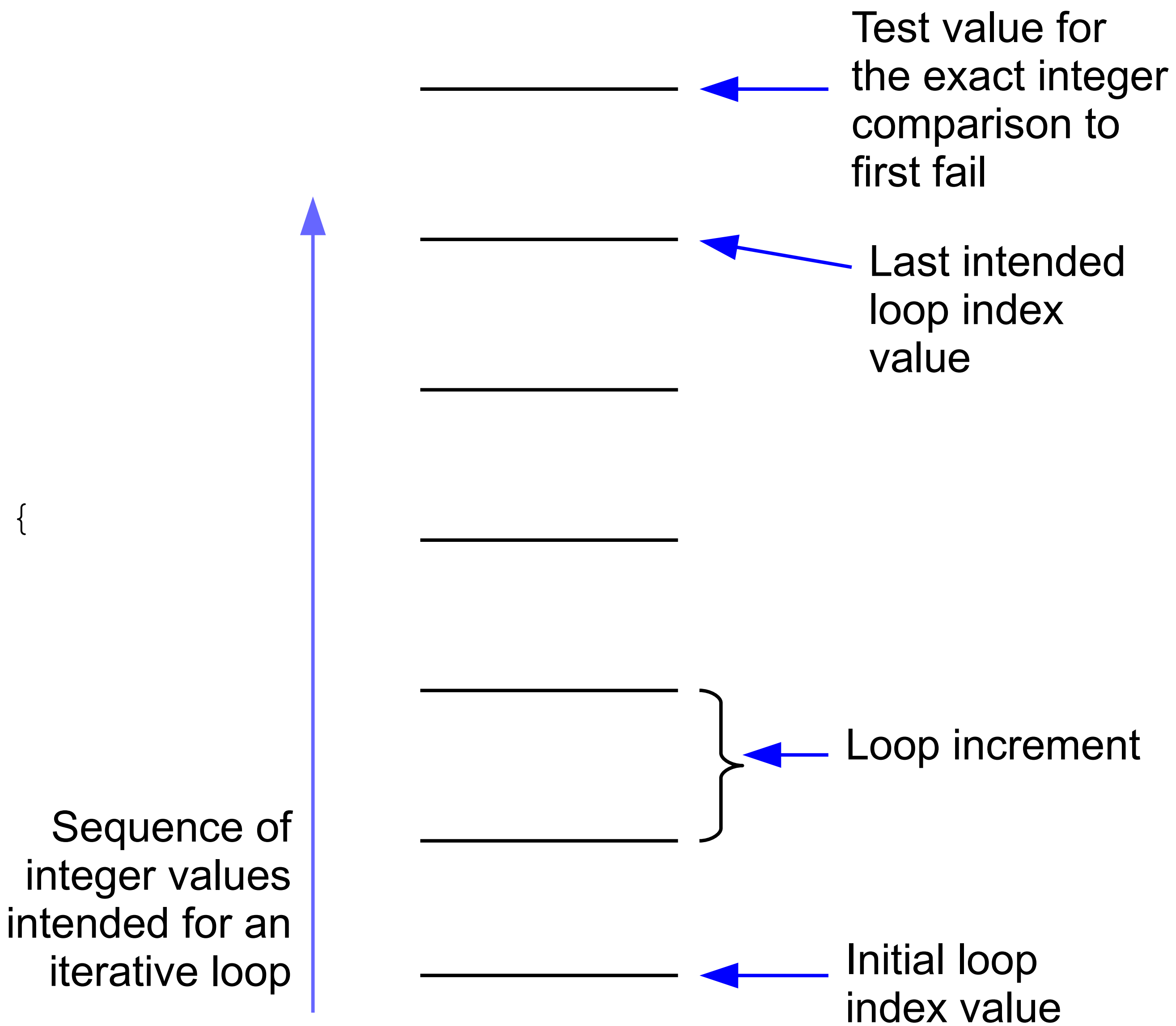
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int i;
  double x;

  i = 0;
  x = 0.0;
  while (i < 10) {
    x = x + 0.7;
    printf("x=%.16f\n",x);
    i++;
  }
  if (x < 7.0) printf("x is less than 7.0\n");
  else if (x == 7.0) printf("x is equal to 7.0\n");
  else printf("x is greater than 7.0\n");
  exit(0);
}
```

Loop adds 0.7 ten times

Indexing Loops with Integer Variables

```
int i;

i = 0;
while (i < 10) {
  velocity[i] = 0.0;
  i++;
}

for (i = 0; i < 10; i++) {
  velocity[i] = 0.0;
}
```

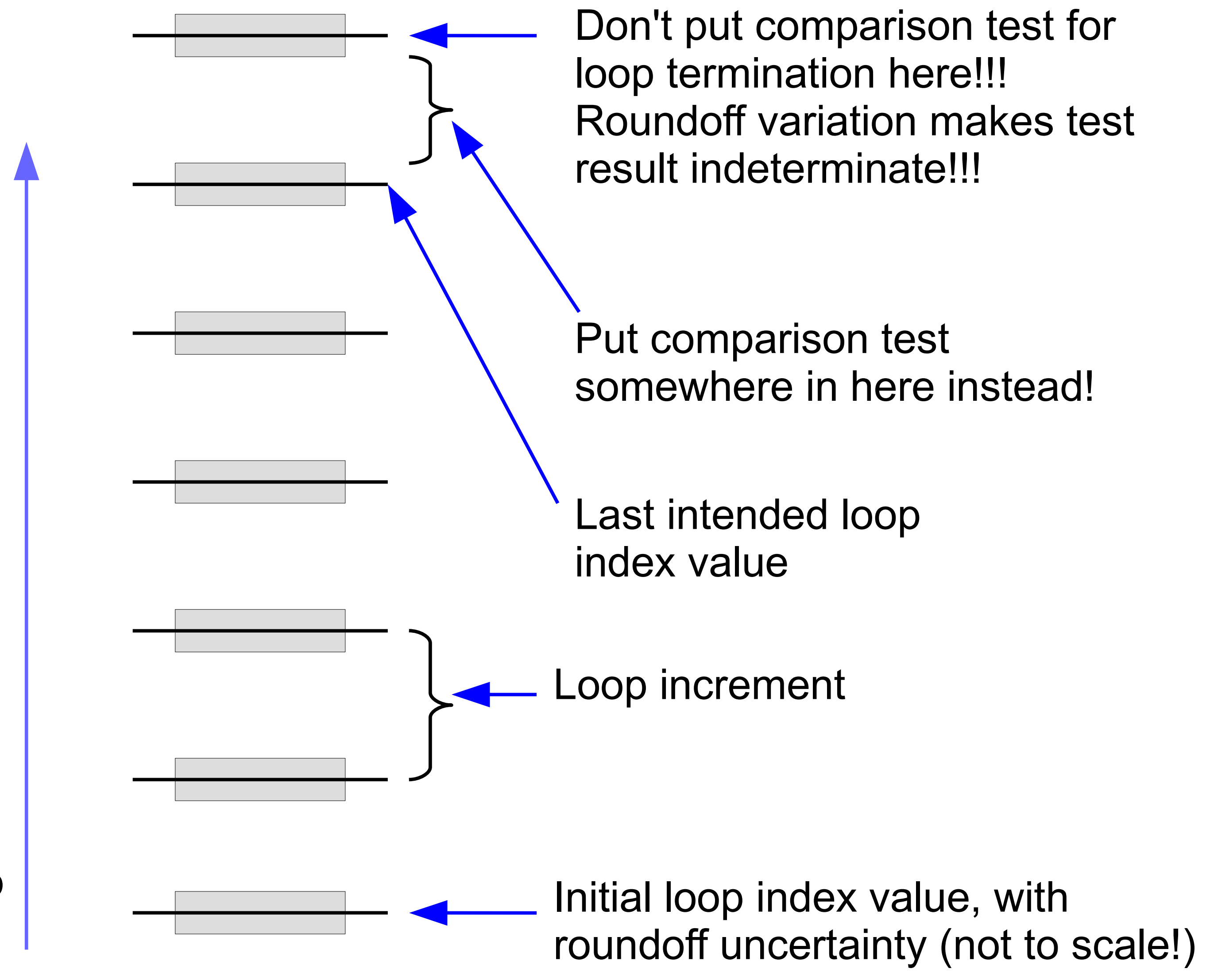
How many times is the comparison expression `i < 10` evaluated?



Test value for the exact integer comparison to first fail
Last intended loop index value
Loop increment
Initial loop index value
Sequence of integer values intended for an iterative loop

Using Floating Point Variables with Comparison Operators

```
double t,x,w;

t = 0;
while (t < 1.0) {
  x = cos(w * t);
  printf(%f %f\n”,t,x);
  t = t + 0.1;
}
```


Don't put comparison test for loop termination here!!! Roundoff variation makes test result indeterminate!!!
Put comparison test somewhere in here instead!
Last intended loop index value
Loop increment
Initial loop index value, with roundoff uncertainty (not to scale!)

Sequence of double values intended for an iterative loop

Example Program to Plot Damped Oscillator Response

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc,char *argv[]) {
  double t,disp,w0,wd,zeta;

  w0 = 6.28318530718;
  zeta = 0.05;
  wd = sqrt(w0*w0 * (1.0 - (zeta*zeta)));
  t = 0.0;
  while (t < 10.005) {
    disp = exp(-w0 * zeta * t) * cos(wd * t);
    printf("%.8g %.8g\n",t,disp);
    t = t + 0.01;
  }
  exit(0);
}
```

Standard header files in /usr/include
Declare function and data types for using functions in C standard library

Start of main program block

Declare variables

Initialize some variables for natural frequency ω_0, damping factor ζ, damped frequency ω_d

Iterative loop using '`while`' steps over time

Compute amplitude as a function of time

Send time and amplitude data points to standard output

Exit main program

Using Floating Point Variables with Comparison Operators

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc,char *argv[]) {
  int i;
  double x;

  i = 0;
  x = 0.0;
  while (x < 1.0) {
    printf("i = %d x=%.16f\n",i,x);
    i++;
    x = x + 0.1;
  }
  exit(0);
}
```

Output:

```
i = 0 x=0.0000000000000000
i = 1 x=0.1000000000000000
i = 2 x=0.2000000000000000
i = 3 x=0.3000000000000000
i = 4 x=0.4000000000000000
i = 5 x=0.5000000000000000
i = 6 x=0.6000000000000000
i = 7 x=0.7000000000000000
i = 8 x=0.7999999999999999
i = 9 x=0.8999999999999999
i = 10 x=0.9999999999999999
```

The loop is executed 11 times instead of the expected 10 times

Changing the expression to test for
`(x < 0.95)`
would be an easy fix

Using Floating Point Variables with Comparison Operators

```
double x,xstart,xstop,xinc;

  ...
xstart = ...;
xstop = ...;
xinc = ...;
  ...
  ...
xstop = xstop + 0.5 * xinc;
x = xstart;
while (x < xstop) {
  ...
  x = x + xinc;
}
```

Example of changing the value to test for loop termination, in this case to allow one final iteration at the stop value

Using Floating Point Variables with Comparison Operators

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "constants.h"

int main(int argc,char *argv[]) {
  double theta,sin_theta;
  int on_axis;

  theta = 0.0;
  while (theta < 20.0 * PI + 0.001) {
    sin_theta = sin(theta);
/* Some calculation here on sin(theta) */
    if (sin_theta == 0.0) on_axis = 1;
    else on_axis = 0;
/* Some calculation here if theta is on x-axis */
    printf("%f %.16f%d\n",theta,sin_theta,on_axis);
    theta = theta + 0.1 * PI;
  }
  exit(0);
}
```

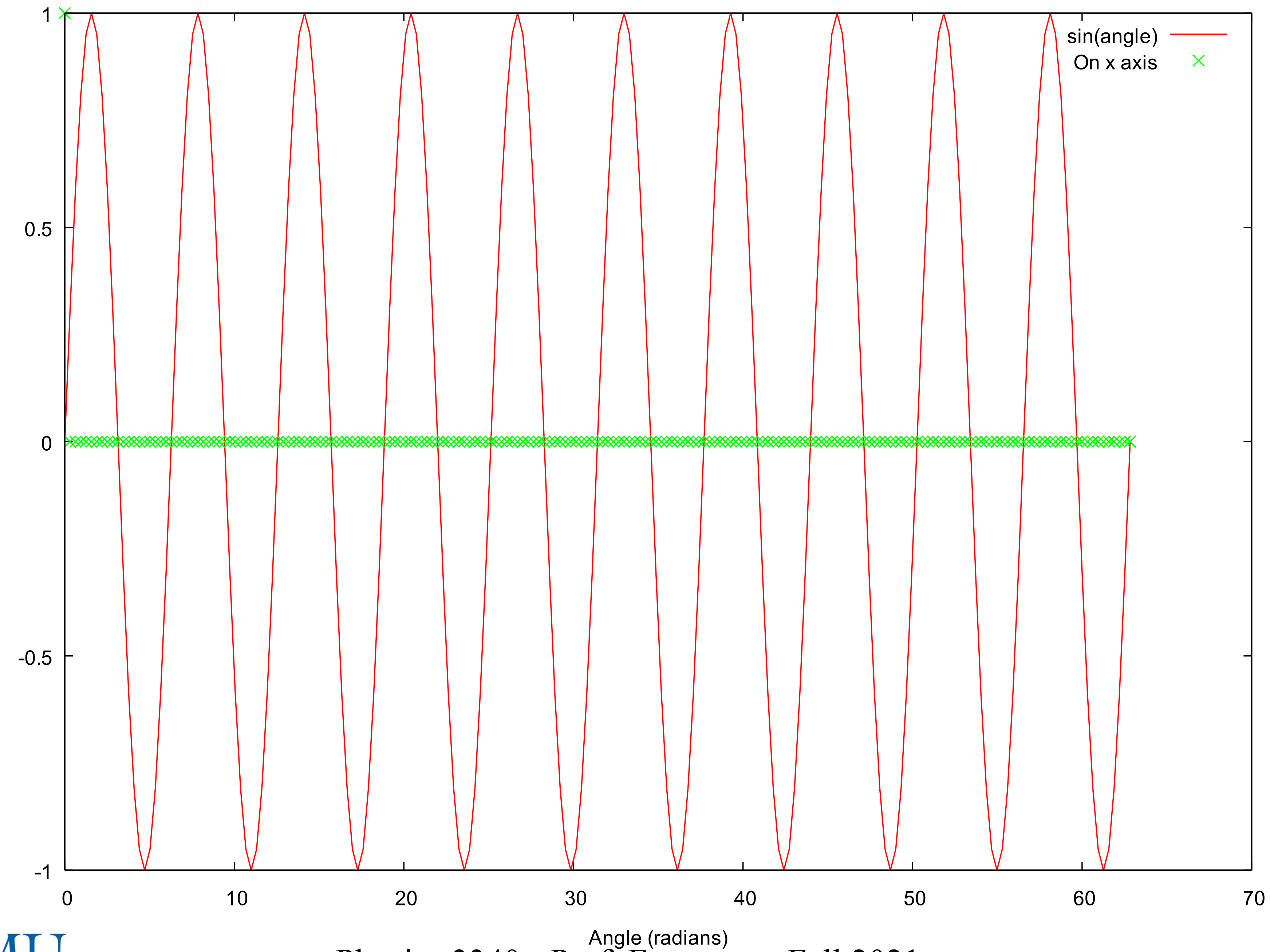
This test demands equality in every one of 64 bits

Output:

```
0.000000 0.0000000000000000 1
0.314159 0.3090169943749474 0
0.628319 0.5877852522924731 0
0.942478 0.8090169943749475 0
1.256637 0.9510565162951535 0
1.570796 1.0000000000000000 0
1.884956 0.9510565162951536 0
2.199115 0.8090169943749475 0
2.513274 0.5877852522924732 0
2.827433 0.3090169943749475 0
3.141593 0.0000000000000001 0
3.455752 -0.3090169943749473 0
3.769911 -0.5877852522924730 0
4.084070 -0.8090169943749473 0
4.398230 -0.9510565162951535 0
4.712389 -1.0000000000000000 0
5.026548 -0.9510565162951536 0
5.340708 -0.8090169943749476 0
5.654867 -0.5877852522924734 0
5.969026 -0.3090169943749476 0
6.283185 -0.0000000000000002 0
6.597345 0.3090169943749472 0
. . . .
```

A bit of roundoff error in theta causes these ‘on-axis’ tests to be missed

Using Floating Point Variables with Comparison Operators



Using Floating Point Variables with Comparison Operators

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "constants.h"

int main(int argc,char *argv[]) {
  double theta,sin_theta;
  int on_axis;

  theta = 0.0;
  while (theta < 20.0 * PI + 0.001) {
    sin_theta = sin(theta);
/* Some calculation here on sin(theta) */
    if (fabs(sin_theta) < 1.0e-12) on_axis = 1;
    else on_axis = 0;
/* Some calculation here if theta is on x-axis */
    printf("%f %.16f%d\n",theta,sin_theta,on_axis);
    theta = theta + 0.1 * PI;
  }
  exit(0);
}
```

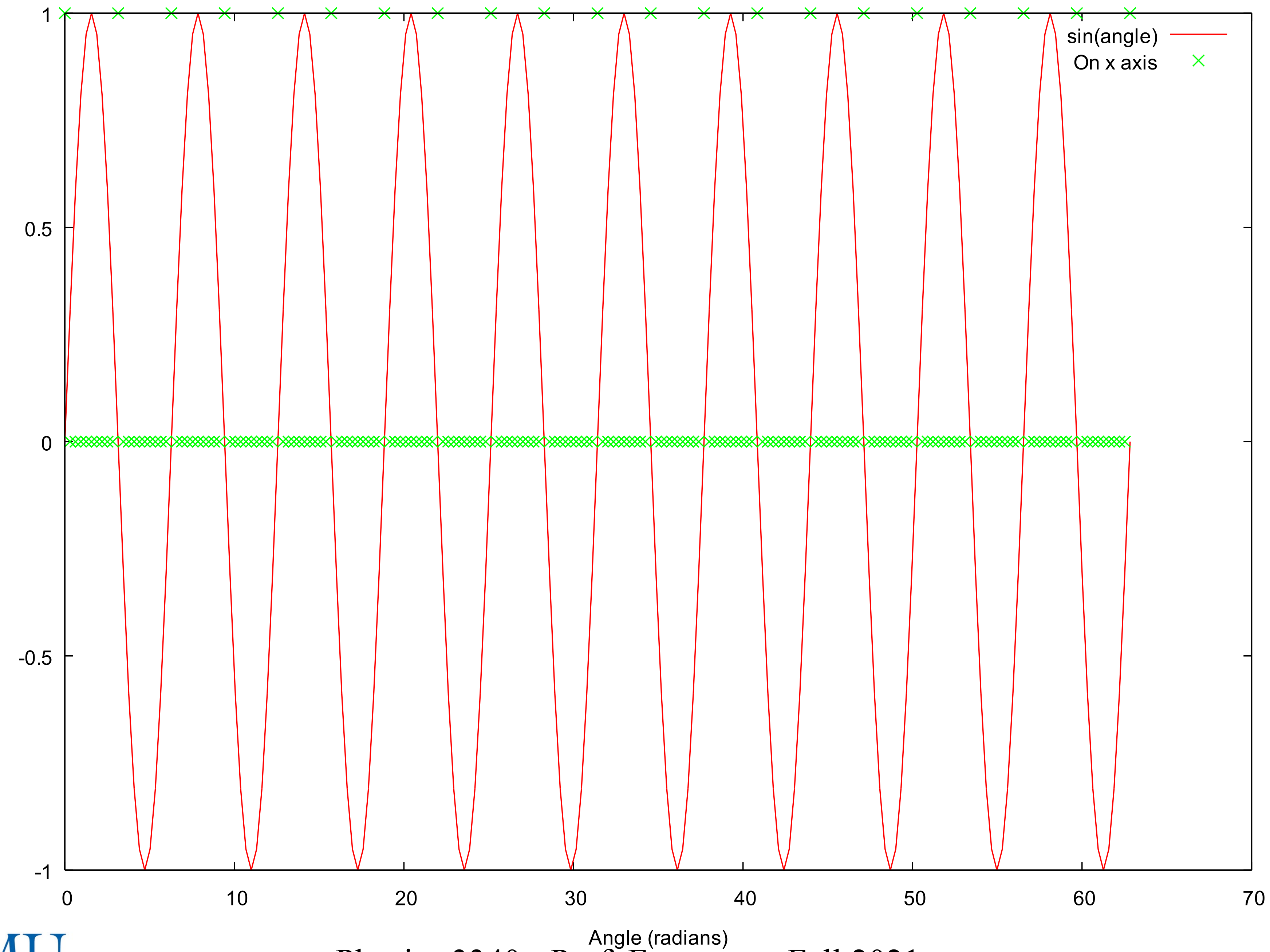
This test includes a small uncertainty window for roundoff error

Output:

```
0.000000 0.0000000000000000 1
0.314159 0.3090169943749474 0
0.628319 0.5877852522924731 0
0.942478 0.8090169943749475 0
1.256637 0.9510565162951535 0
1.570796 1.0000000000000000 0
1.884956 0.9510565162951536 0
2.199115 0.8090169943749475 0
2.513274 0.5877852522924732 0
2.827433 0.3090169943749475 0
3.141593 0.0000000000000001 1
3.455752 -0.3090169943749473 0
3.769911 -0.5877852522924730 0
4.084070 -0.8090169943749473 0
4.398230 -0.9510565162951535 0
4.712389 -1.0000000000000000 0
5.026548 -0.9510565162951536 0
5.340708 -0.8090169943749476 0
5.654867 -0.5877852522924734 0
5.969026 -0.3090169943749476 0
6.283185 -0.0000000000000002 1
6.597345 0.3090169943749472 0
. . . .
```

All 'on-axis' tests now correct

Using Floating Point Variables with Comparison Operators



Beware of Random Walks of Roundoff Error!

Example: edit integration lab program to try to compute pi:

$$\pi = \int_0^1 4\sqrt{1-x^2}\,dx$$

Beware of Random Walks of Roundoff Error!

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "integrate.h"

double func(double x) {
  return(4.0 * sqrt(1.0 - x*x));
}

int main(int argc,char *argv[]) {
  double xinc;

  if (argc != 2) {
    fprintf(stderr,"%s <xinc>\n",argv[0]);
    exit(1);
  }
  xinc = atof(argv[1]);
  printf("Trapezoidal rule:\n");
  trapezoidal(func,0.0,1.0,xinc);
  printf("Simpson's rule:\n");
  simpson(func,0.0,1.0,xinc);
  exit(0);}
```

Beware of Random Walks of Roundoff Error!

```
$ ./pi_integrate 0.01
Trapezoidal rule:
-nan
Simpson's rule:
-nan
$ ./pi_integrate 0.0025
Trapezoidal rule:
3.14144566781
Simpson's rule:
3.14153524166
```

Try two different step sizes through the integration routines

So what is this nan business???

NaN Binary Storage Convention

$$1.0 = \underset{s\,=\,0}{0}\ \ \underbrace{01111111111}_{e\,=\,1023\,-\,1023\,=\,0}\ \ \underbrace{1.\ 0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000}_{q\,=\,1.0} \Rightarrow 1.0=(-1)^0\cdot 1.0\cdot 2^0$$

$$0.0 = \underset{s\,=\,0}{0}\ \ \underbrace{00000000000}\ \ \underbrace{1.\ 0000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000}_{\text{all actual bits 0}} \Rightarrow \text{0.0 by convention}$$

$$\text{NaN} = \underset{s\,=\,1}{1}\ \ \underbrace{11111111111}_{\text{all exponent bits =1}}\ \ \underbrace{1.\ \underset{\text{Most significant mantissa bit=1}}{1}000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000} \Rightarrow \text{NaN by convention}$$

The man Page for sqrt()

```
SQRT(3)                    Linux Programmer's Manual                    SQRT(3)

NAME
       sqrt, sqrtf, sqrtl - square root function

SYNOPSIS
       #include <math.h>

       double sqrt(double x);
       float sqrtf(float x);
       long double sqrtl(long double x);

       Link with -lm.

DESCRIPTION
       The sqrt() function returns the nonnegative square root of x.

RETURN VALUE
       On success, these functions return the square root of x.
       If x is a NaN, a NaN is returned.
       If x is +0 (-0), +0 (-0) is returned.
       If x is positive infinity, positive infinity is returned.
       If x is less than -0, a domain error occurs, and a NaN is returned.
```

!!!