Class Progress

Basics of Linux, gnuplot, C
Visualization of numerical data
Roots of nonlinear equations
(Midterm 1)
Solutions of systems of linear equations
Solutions of systems of nonlinear equations
Monte Carlo simulation
Interpolation of sparse data points
Numerical integration
(Midterm 2)
Solutions of ordinary differential equations

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, $3.21\times10^{6} + 4.56\times10^{-4}$:

$$3.210000000000\times10^{6}$$
$$0.000000000456\times10^{6}$$
$$---------------$$
$$3.210000000456\times10^{6}$$

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?

Image for RAM Storage

Each box has a numerical address or pointer

What is the numerical capacity of each box?

Roundoff Error

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

```
3.210000000000×10^6
0.000000000246×10^6
---------------
3.210000000246×10^6
```

Correct sum

Width limit of arithmetic storage and processor

```
3.210000000000×10^6
0.000000000246×10^6
---------------
3.210000000×10^6
```

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

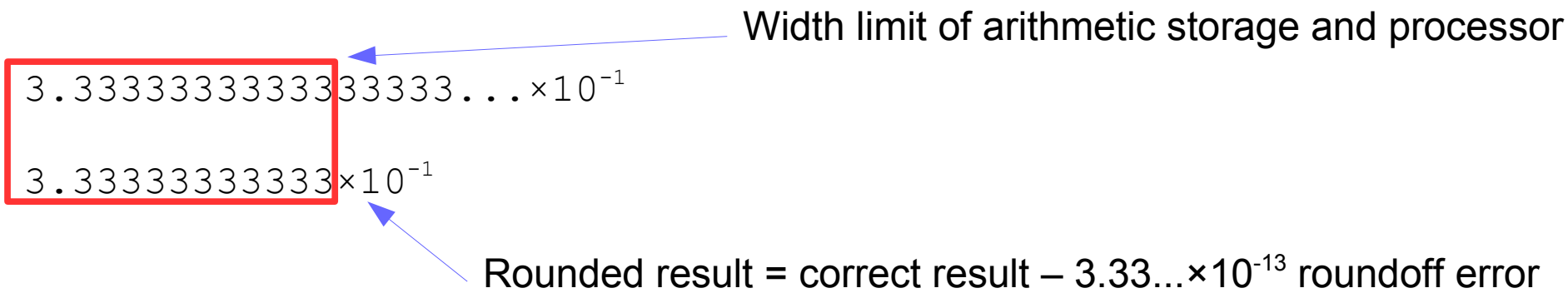
Width limit of arithmetic storage and processor

```
3.210000000000×10^6
0.000000000246×10^6
---------------
3.210000000×10^6
```

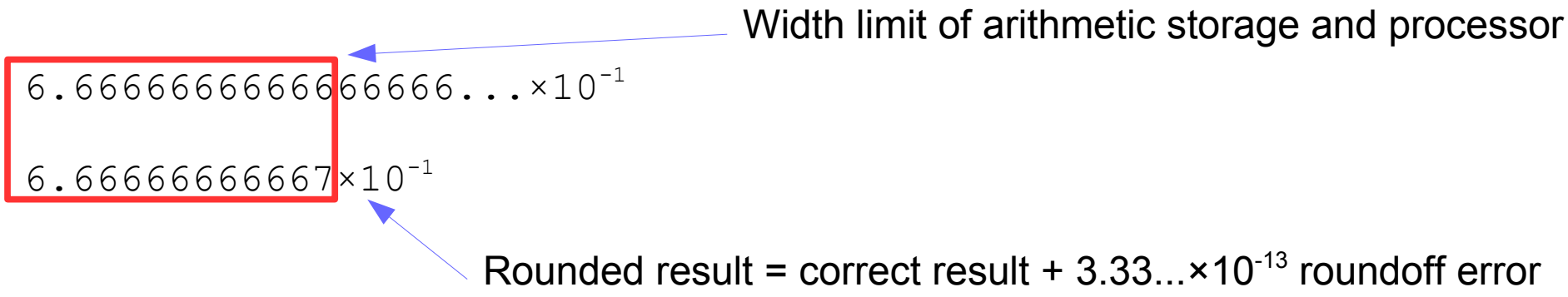
Rounded sum = correct sum + 4.0×10^{-6} roundoff error

Roundoff Error

Look at example 1.0 / 3.0:



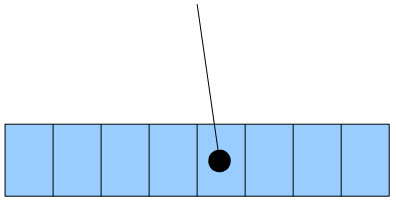
Look at example 2.0 / 3.0:

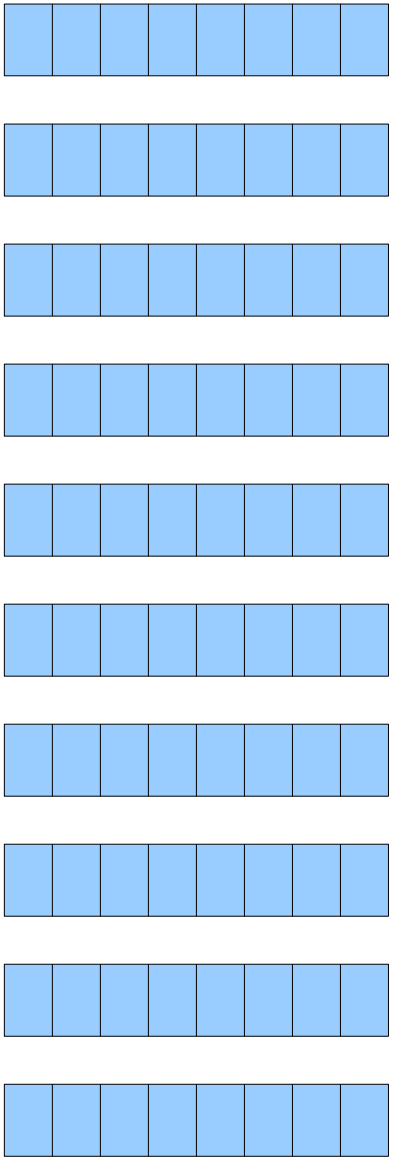


Understanding numerical limits and roundoff error is essential in scientific computing. Of course, computer hardware uses binary arithmetic, but the same principles apply.

Map of Bytes in RAM

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





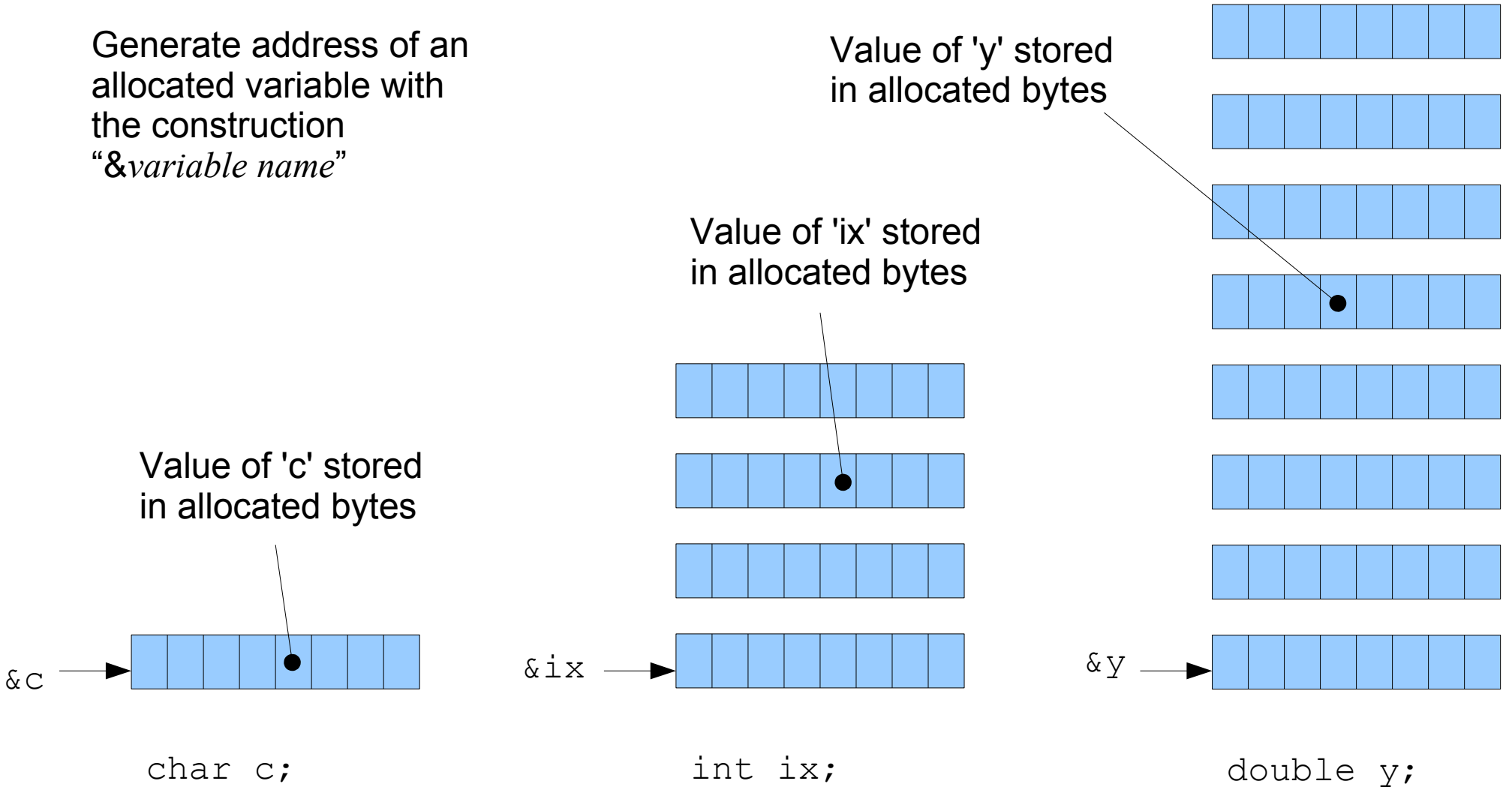
Increasing address of RAM bytes

C Data Types for Numerical Programming

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

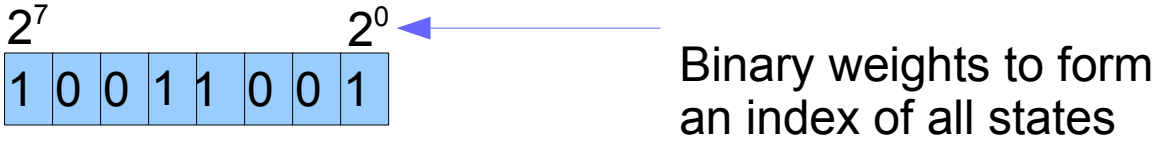
Allocation of C Data Types in RAM

Generate address of an allocated variable with the construction "&*variable name*"



Total Possible Bit Arrangements

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



The total number of possible unique arrangements is ...

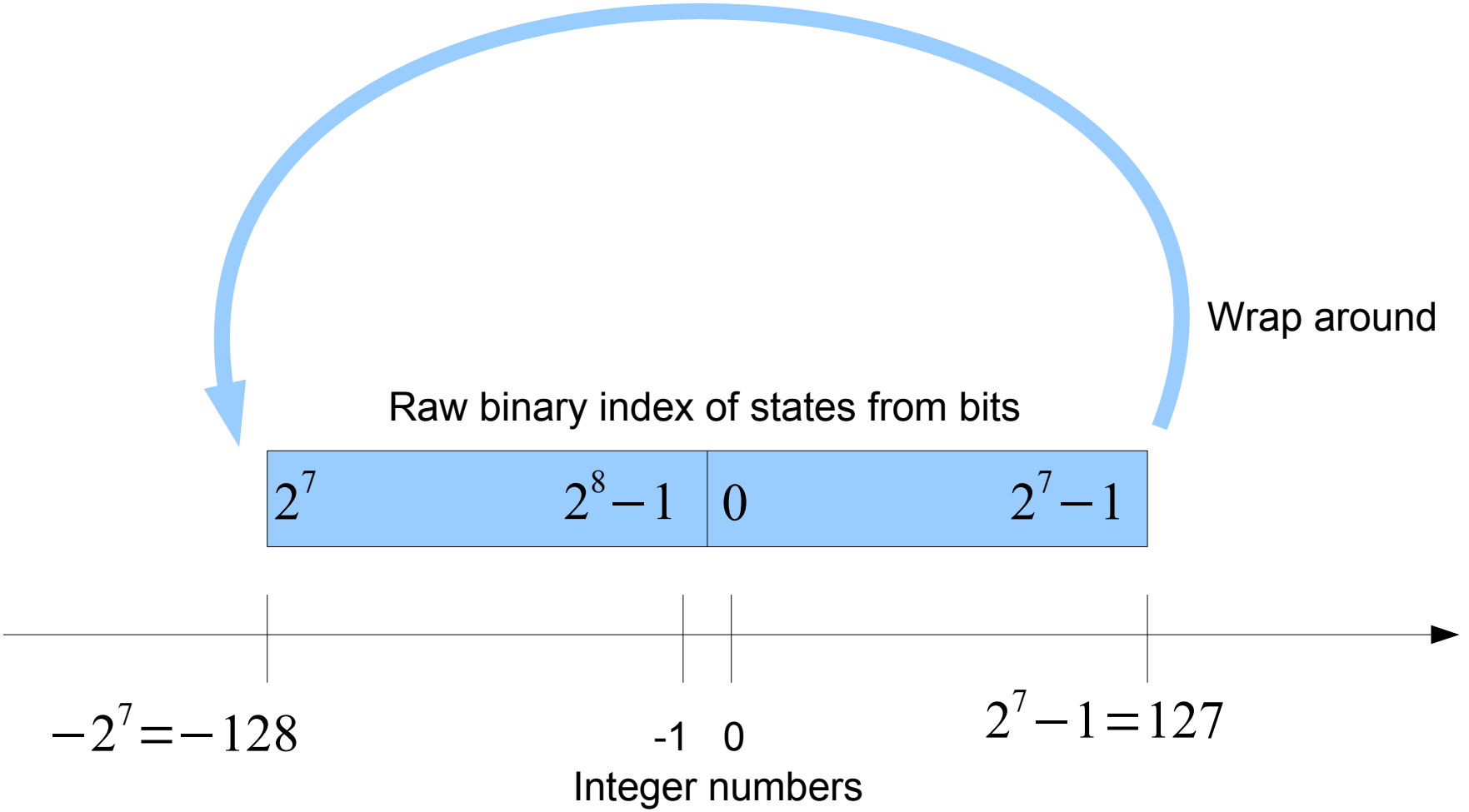
Unsigned 8-bit Character Values

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

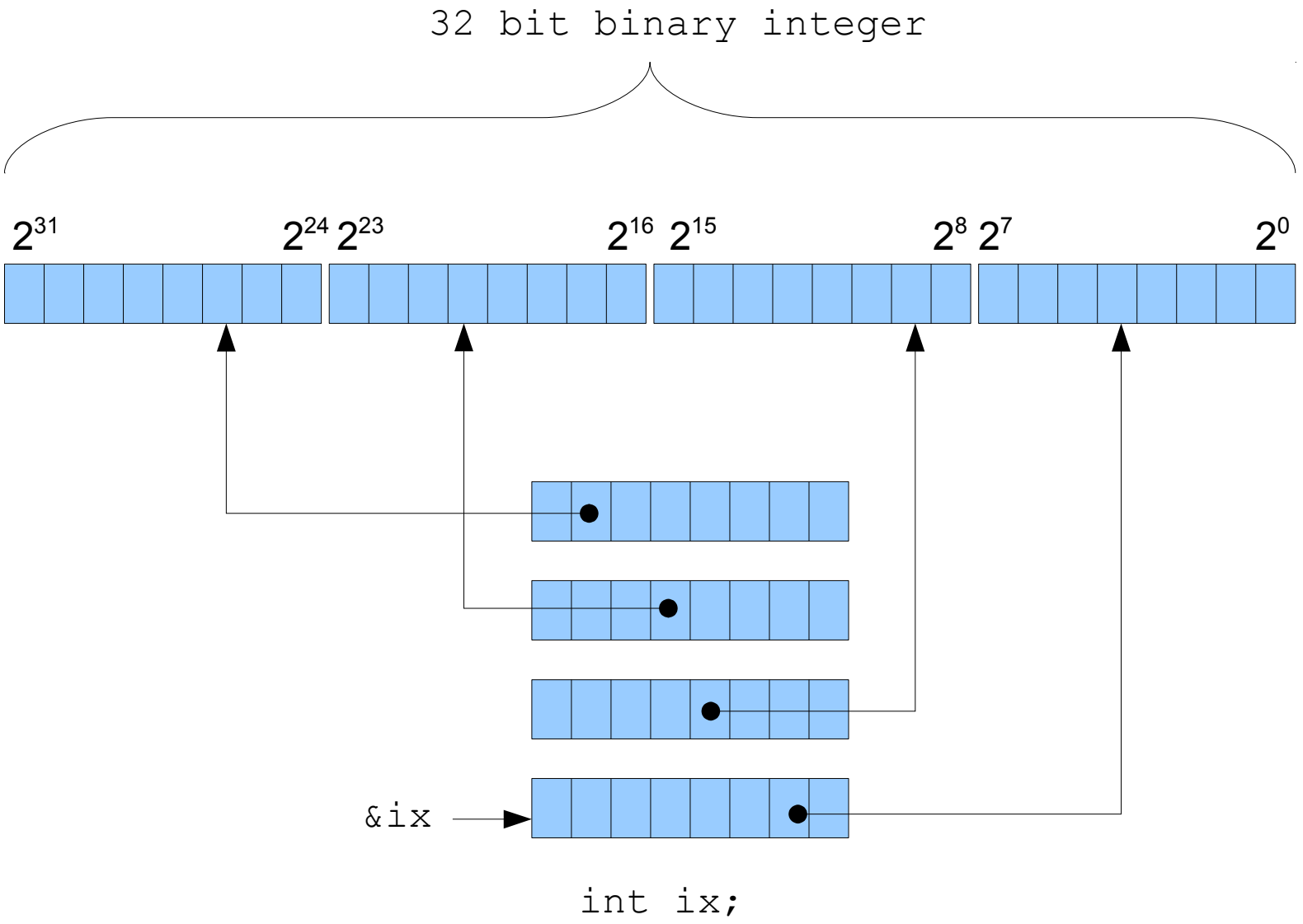
How do we assign numerical values to each of these arrangements?

Signed 8-bit Character Values

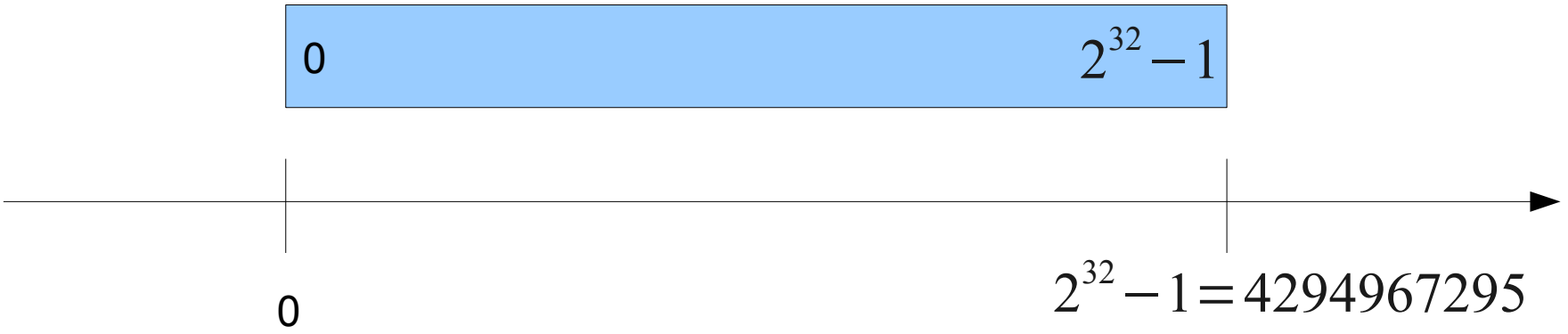
Same possible bit arrangements, but a different numerical mapping...



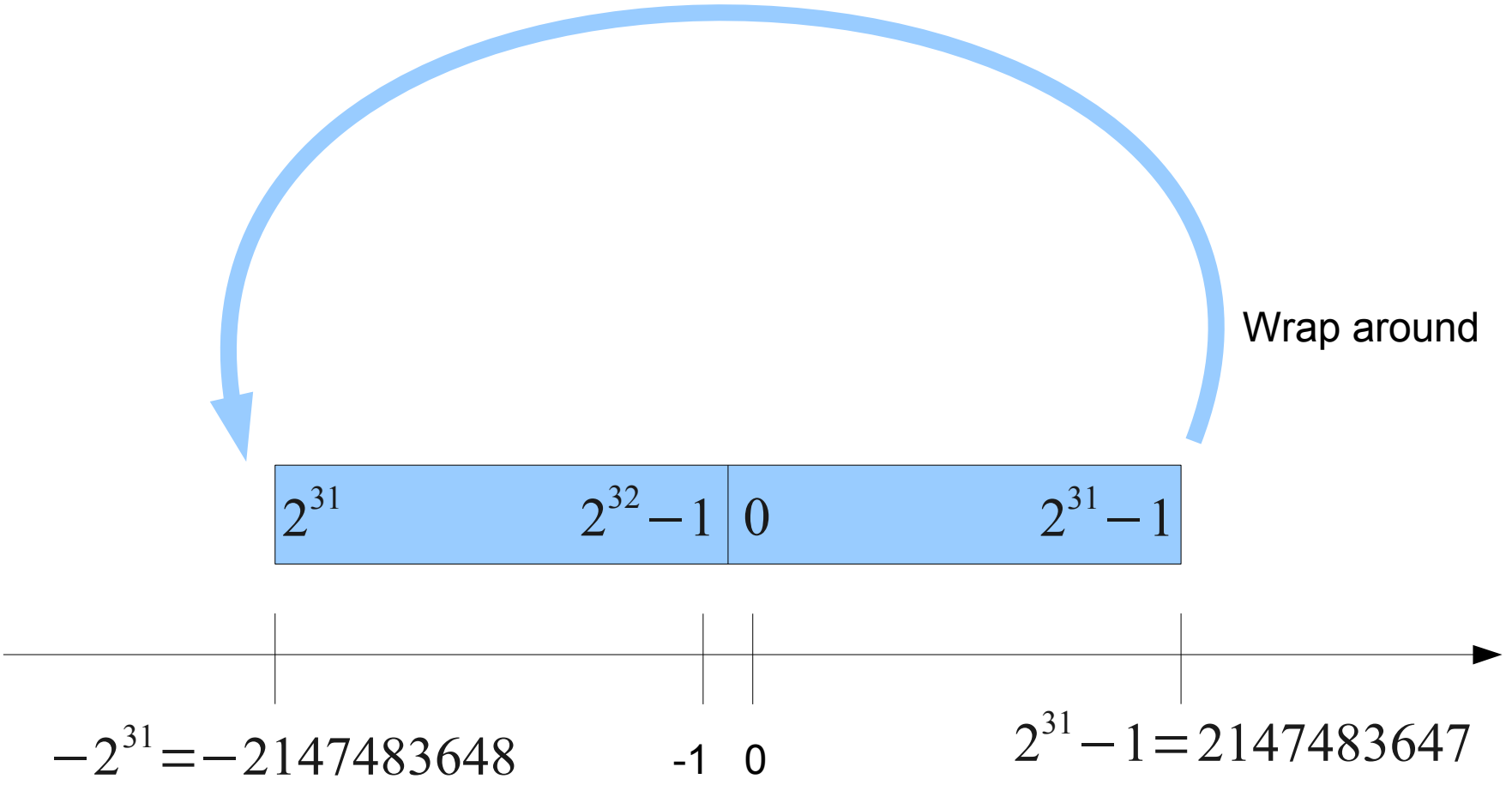
Integer Bit Arrangement for “Little Endian” Processors



Unsigned 32-bit Integer Values



Signed 32-bit Integer Values



Pseudocode for Seeing Integer Wraparound

```
integer i , s ;
s ← 0
for i = 1 to 50 do
      s ← s + 268435456
      output s
end for
```

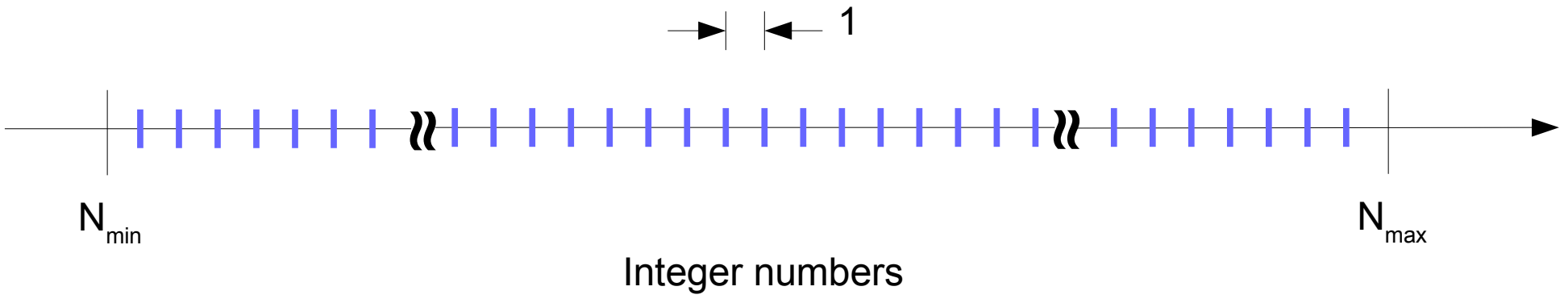
Note: $268435456 = 2^{28}$

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
```

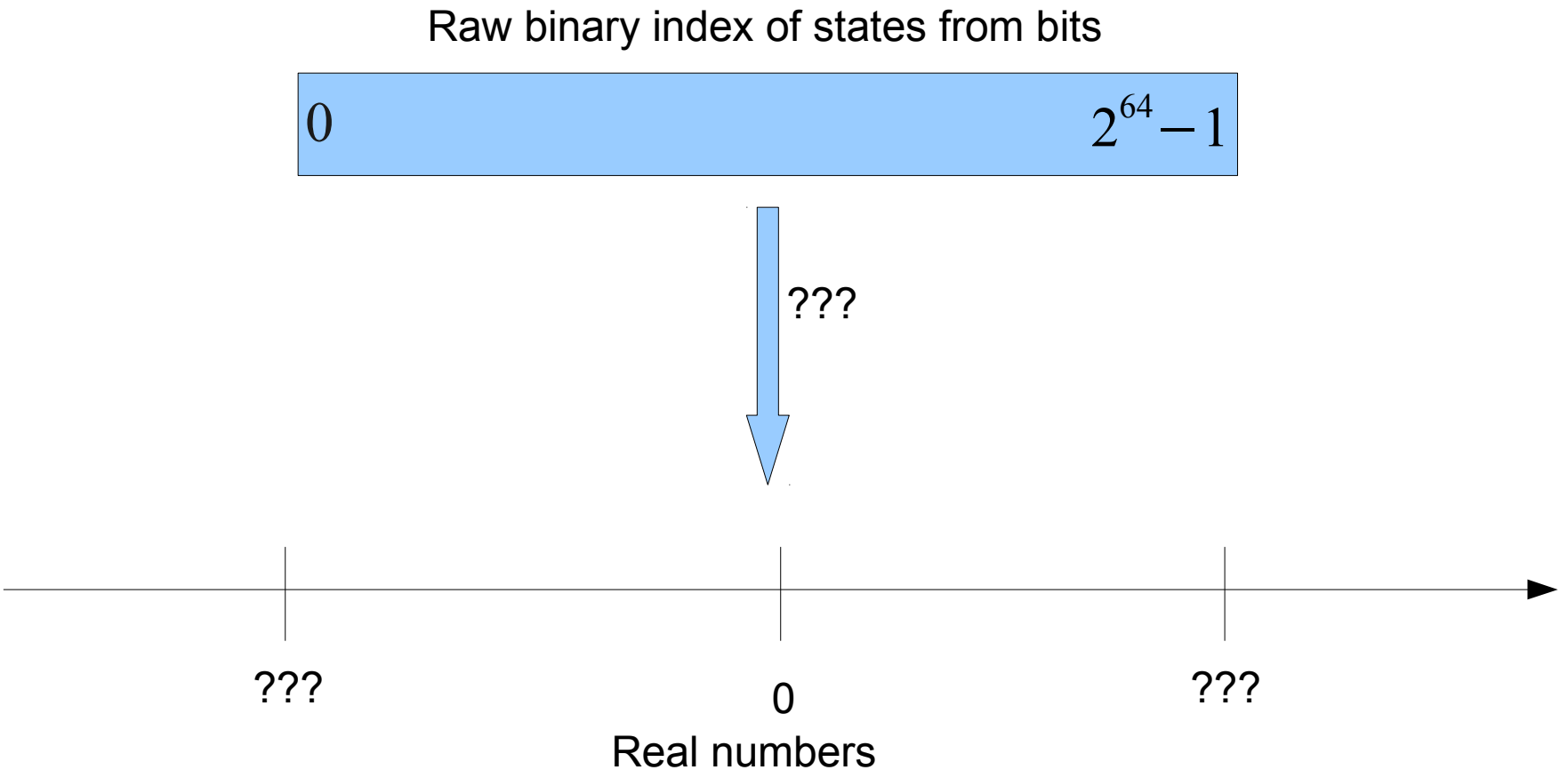
Density on Number Line

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



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

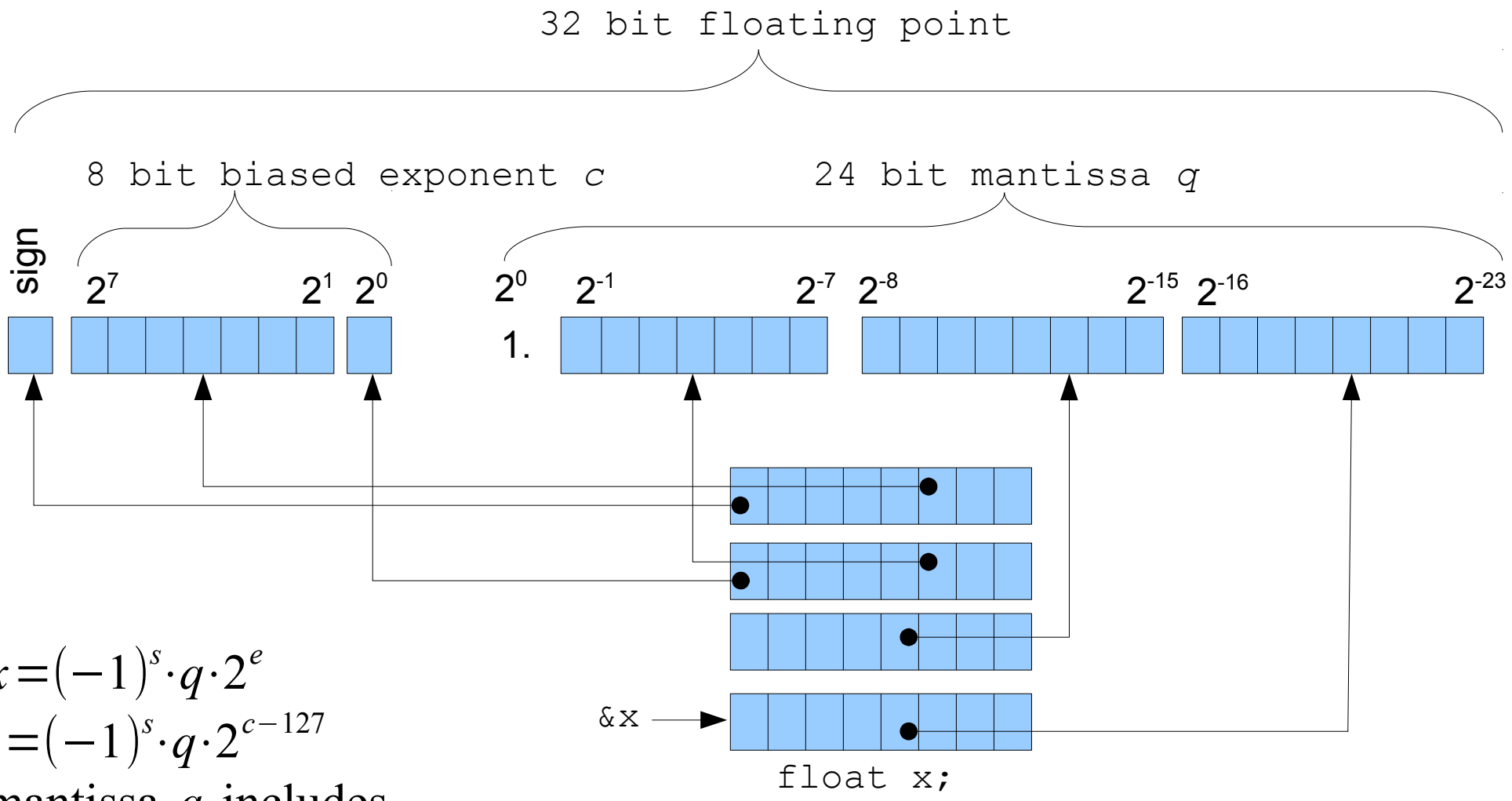
0.123×10^{3}

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.

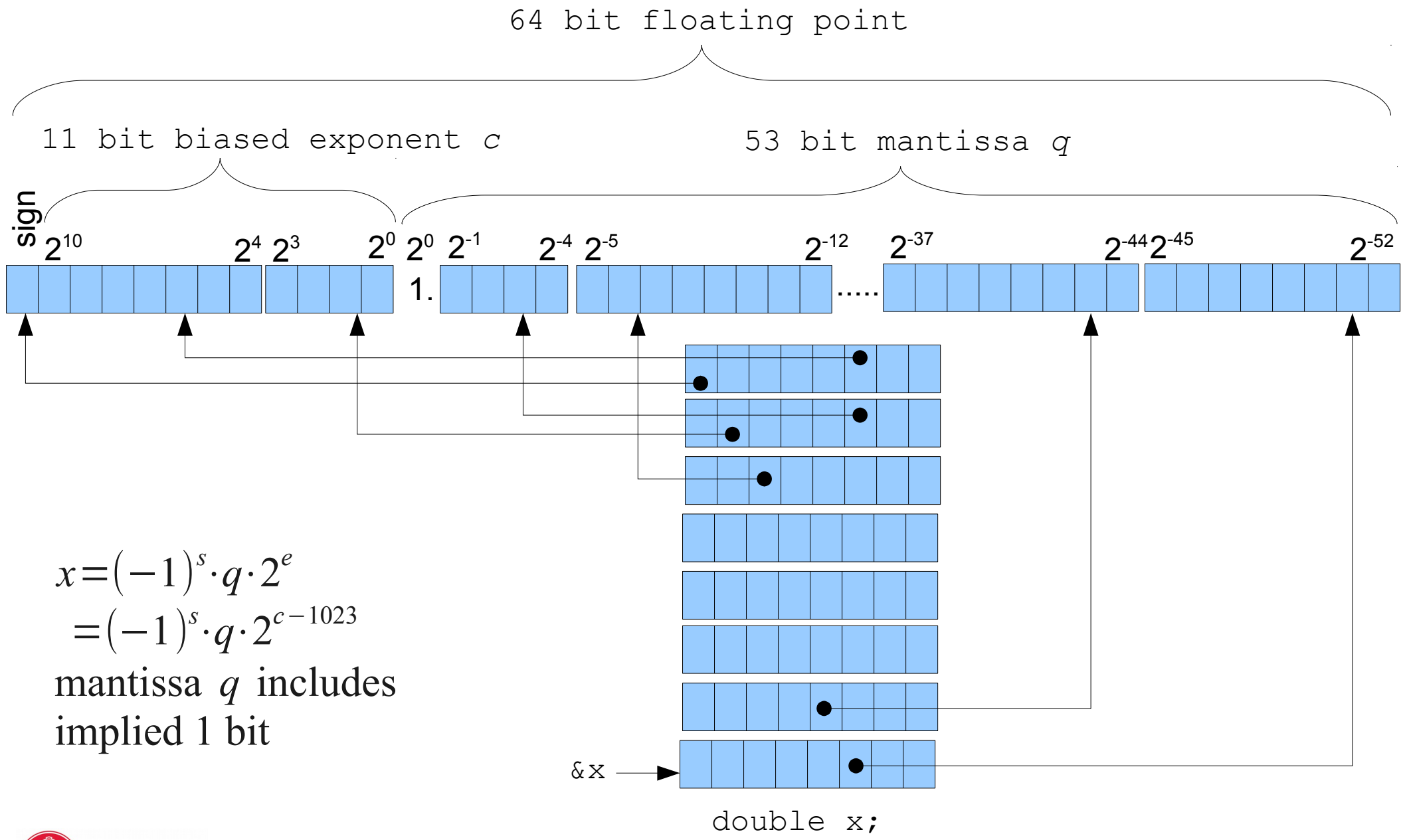
Single Precision Floating Point Bit Arrangement



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

mantissa q includes implied 1 bit

Double Precision Floating Point Bit Arrangement



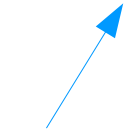
$$x = (-1)^s \cdot q \cdot 2^e$$
$$= (-1)^s \cdot q \cdot 2^{c-1023}$$

mantissa q includes implied 1 bit

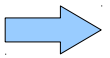
Double Precision Examples

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

1.0 = 0 01111111111 1. 0000 00000000 00000000 00000000 00000000 00000000



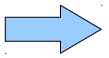
s = 0 *e* = 1023 - 1023 = 0 *q* = 1.0



$$1.0 = (-1)^0 \cdot 1.0 \cdot 2^0$$

-1.0 = 1 01111111111 1. 0000 00000000 00000000 00000000 00000000 00000000

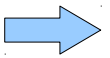
s = 1 *e* = 1023 - 1023 = 0 *q* = 1.0



$$-1.0 = (-1)^1 \cdot 1.0 \cdot 2^0$$

2.0 = 0 10000000000 1. 0000 00000000 00000000 00000000 00000000 00000000

s = 0 *e* = 1024 - 1023 = 1 *q* = 1.0



$$2.0 = (-1)^0 \cdot 1.0 \cdot 2^1$$

Double Precision Examples

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

0.0 = 0 00000000000 1. 0000 00000000 00000000 00000000 00000000 00000000 00000000

s = 0 all actual bits 0 → 0.0 by convention

10.0 = 0 10000000010 1. 0100 00000000 00000000 00000000 00000000 00000000 00000000

s = 0 *e* = 1026 - 1023 = 3 *q* = 1.25 → $10.0 = (-1)^0 \cdot 1.25 \cdot 2^3$

0.1 = 0 01111111011 1. 1001 10011001 10011001 10011001 10011001 10011001 10011010

s = 0 *e* = 1019 - 1023 = -4 $q \approx 1.6$ → $0.1 \approx (-1)^0 \cdot 1.6 \cdot 2^{-4}$

Double Precision Examples

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

0.5 = 0 01111111110 1. 0000 00000000 00000000 00000000 00000000 00000000

s = 0 *e* = 1022 - 1023 = -1 *c* = 1.0

$$0.5 = (-1)^0 \cdot 1.0 \cdot 2^{-1}$$

0.25 = 0 01111111101 1. 0000 00000000 00000000 00000000 00000000 00000000

s = 0 *e* = 1021 - 1023 = -2 *c* = 1.0

$$0.25 = (-1)^0 \cdot 1.0 \cdot 2^{-2}$$

1.0000000000000002 =

0 01111111111 1. 0000 00000000 00000000 00000000 00000000 00000001

s = 0 *e* = 1023 - 1023 = 0 *c* = 1.0000000000000002

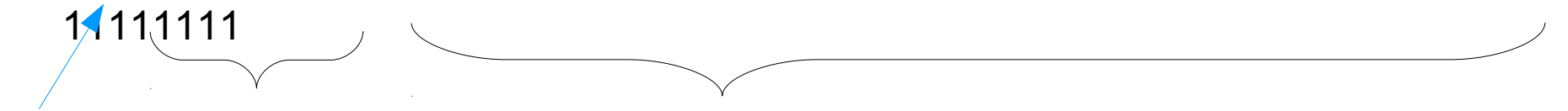
$$1.0000000000000002 = (-1)^0 \cdot 1.0000000000000002 \cdot 2^0$$

Double Precision Examples

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

Largest possible number

= 0 11111111110 1. 1111 11111111 11111111 11111111 11111111 11111111 11111111

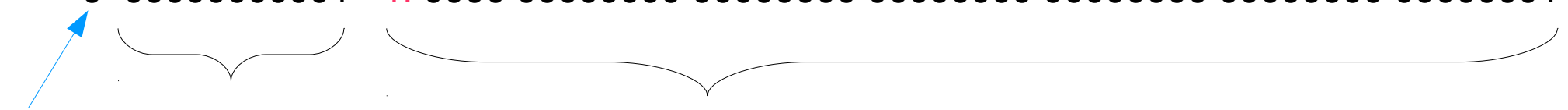


s = 0 *e* = 2046 - 1023 = 1023 *c* ≈ 2.0 $(-1)^0 \cdot 2.0 \cdot 2^{1023} \approx 1.8 \cdot 10^{308}$

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

Smallest possible number, excluding subnormal numbers

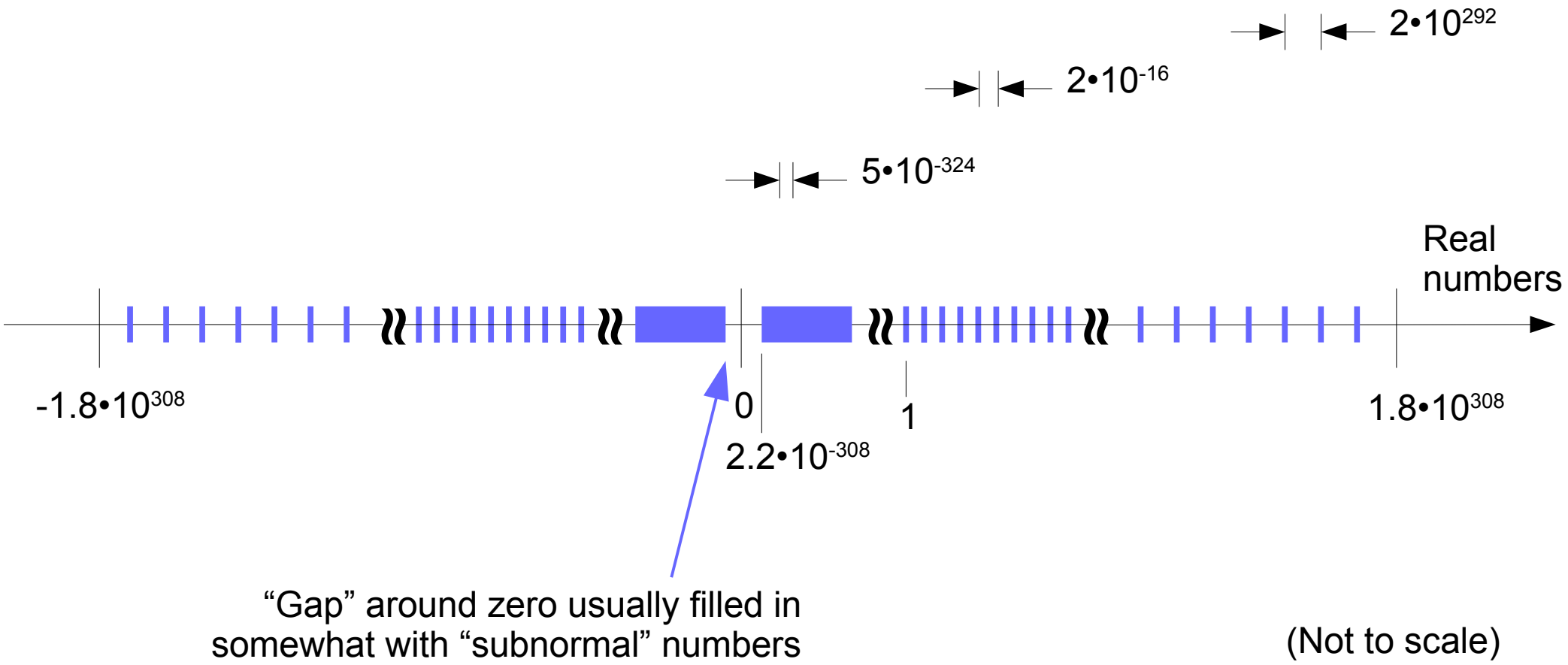
= 0 00000000001 1. 0000 00000000 00000000 00000000 00000000 00000000 00000001



s = 0 *e* = 1 - 1023 = -1022 *c* ≈ 1.0 $(-1)^0 \cdot 1.0 \cdot 2^{-1022} \approx 2.2 \cdot 10^{-308}$

Density on Number Line

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



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 ← 1.0
y ← 0.125
for i = 1 to 20 do
    z ← x + y
    output x, y, z
    y ← y · 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=1.000003814697265
x=1.000000000000000 y=4.768371582031250e-07 z=1.000000476837158
x=1.000000000000000 y=5.960464477539062e-08 z=1.000000059604644
x=1.000000000000000 y=7.450580596923828e-09 z=1.000000007450580
x=1.000000000000000 y=9.313225746154785e-10 z=1.000000000931322
x=1.000000000000000 y=1.164153218269348e-10 z=1.000000000116415
x=1.000000000000000 y=1.455191522836685e-11 z=1.000000000014551
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=1.000000000000003
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
```

Pseudocode for Exploring Precision Limit

```
integer i ; real t , x , y , z ;
x ← 1.0
t ← 0.1
for i = 1 to 20 do
      y ← x + t
      z ← y − x
      output x , y , z
      x ← x · 8.0
end for
```

C Code Output Using Double Precision

```
x=1.000000000000000e+00 y=1.100000000000001e+00 z=0.100000000000001
x=8.000000000000000e+00 y=8.099999999999996e+00 z=0.099999999999996
x=6.400000000000000e+01 y=6.409999999999994e+01 z=0.099999999999943
x=5.120000000000000e+02 y=5.121000000000002e+02 z=0.100000000000227
x=4.096000000000000e+03 y=4.096100000000004e+03 z=0.100000000003638
x=3.276800000000000e+04 y=3.276809999999999e+04 z=0.099999999985448
x=2.621440000000000e+05 y=2.621440999999998e+05 z=0.099999999767169
x=2.097152000000000e+06 y=2.097152100000001e+06 z=0.100000000931323
x=1.677721600000000e+07 y=1.677721610000001e+07 z=0.100000014901161
x=1.342177280000000e+08 y=1.342177280999999e+08 z=0.099999940395355
x=1.073741824000000e+09 y=1.073741824099999e+09 z=0.099999046325684
x=8.589934592000000e+09 y=8.589934592100004e+09 z=0.100003814697266
x=6.871947673600000e+10 y=6.871947673610006e+10 z=0.100061035156250
x=5.497558138880000e+11 y=5.497558138880998e+11 z=0.099975585937500
x=4.398046511104000e+12 y=4.398046511104096e+12 z=0.099609375000000
x=3.518437208883200e+13 y=3.518437208883210e+13 z=0.101562500000000
x=2.814749767106560e+14 y=2.814749767106561e+14 z=0.125000000000000
x=2.251799813685248e+15 y=2.251799813685248e+15 z=0.000000000000000
x=1.801439850948198e+16 y=1.801439850948198e+16 z=0.000000000000000
x=1.441151880758558e+17 y=1.441151880758558e+17 z=0.000000000000000
```

Machine ε

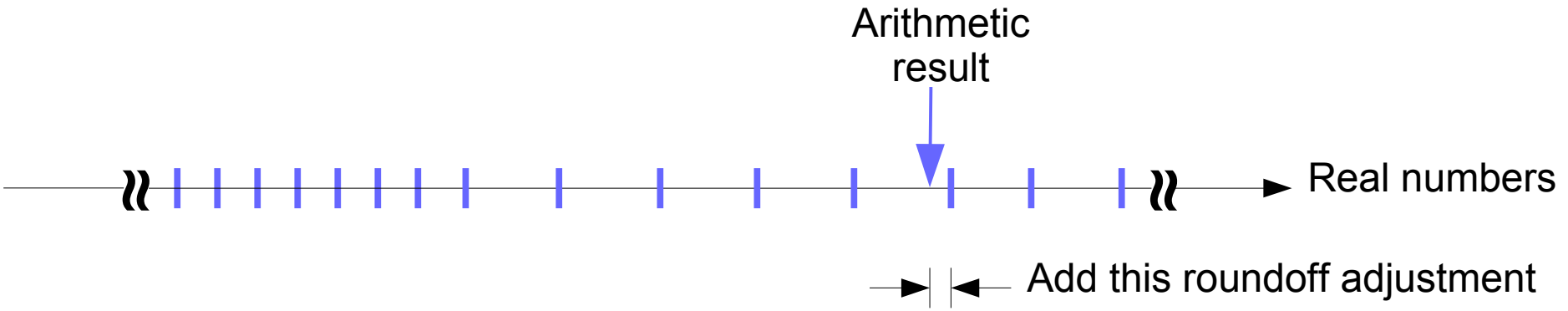
The “Machine ε” is the smallest number ε such that $1 + \varepsilon \neq 1$

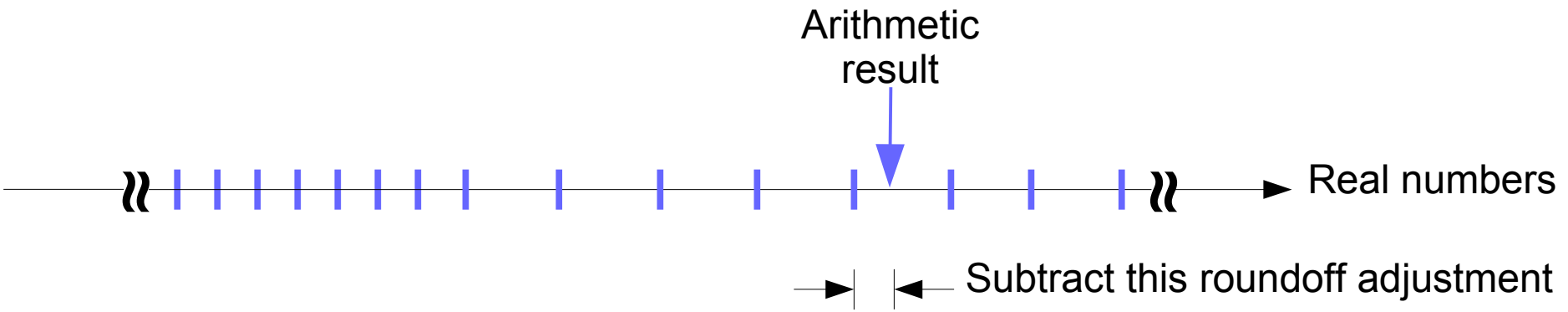
For 32-bit float data type, $\varepsilon = 2^{-24} \approx 5.96 \times 10^{-8}$
A number slightly larger than this ε will be rounded up to 2^{-23} when added to 1.0 and turn the least significant bit to '1'

For 64-bit double data type, $\varepsilon = 2^{-53} \approx 1.11 \times 10^{-16}$
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'

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 64-bit doubles for these reals

Use 32-bit float for this real

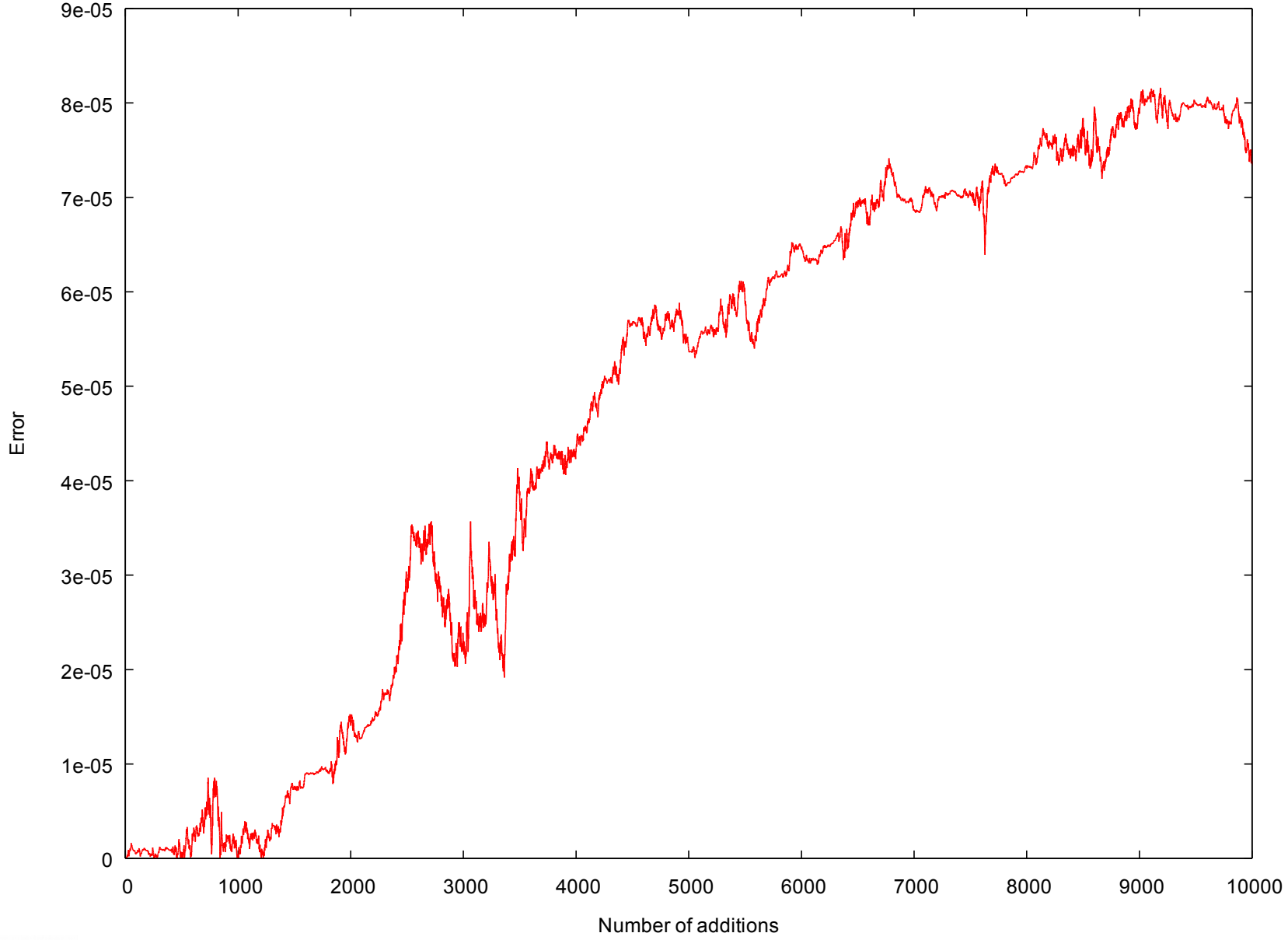
```
integer i ; real r , xd ; real xf ;
xd ← 0.0
xf ← 0.0
for i = 1 to 10000 do
    r ← pseudorandom number between -1 and 1
    xf ← xf + r
    xd ← xd + r
    error ← xf − xd
    output i , error
end for
```

Accumulate sum of pseudorandom numbers in both float and double

Interpret difference as accumulated roundoff error for float

Accumulated roundoff error

Each roundoff event can be seen as an addition of a random number in the range -ε to +ε



Evaluation of Polynomial with Multiple Roots

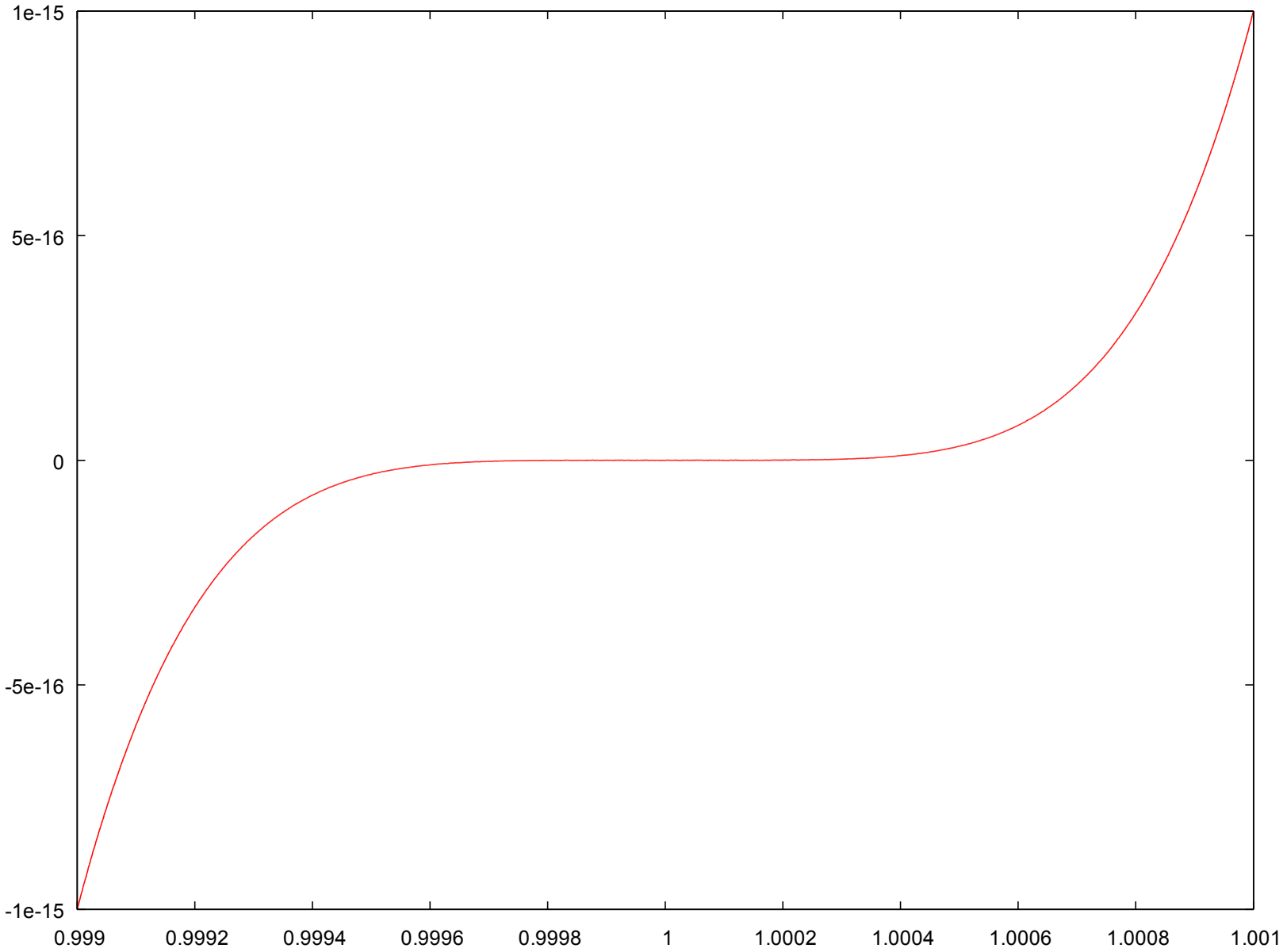
$$y = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1$$

Has root of multiplicity 5 at $x = 1$

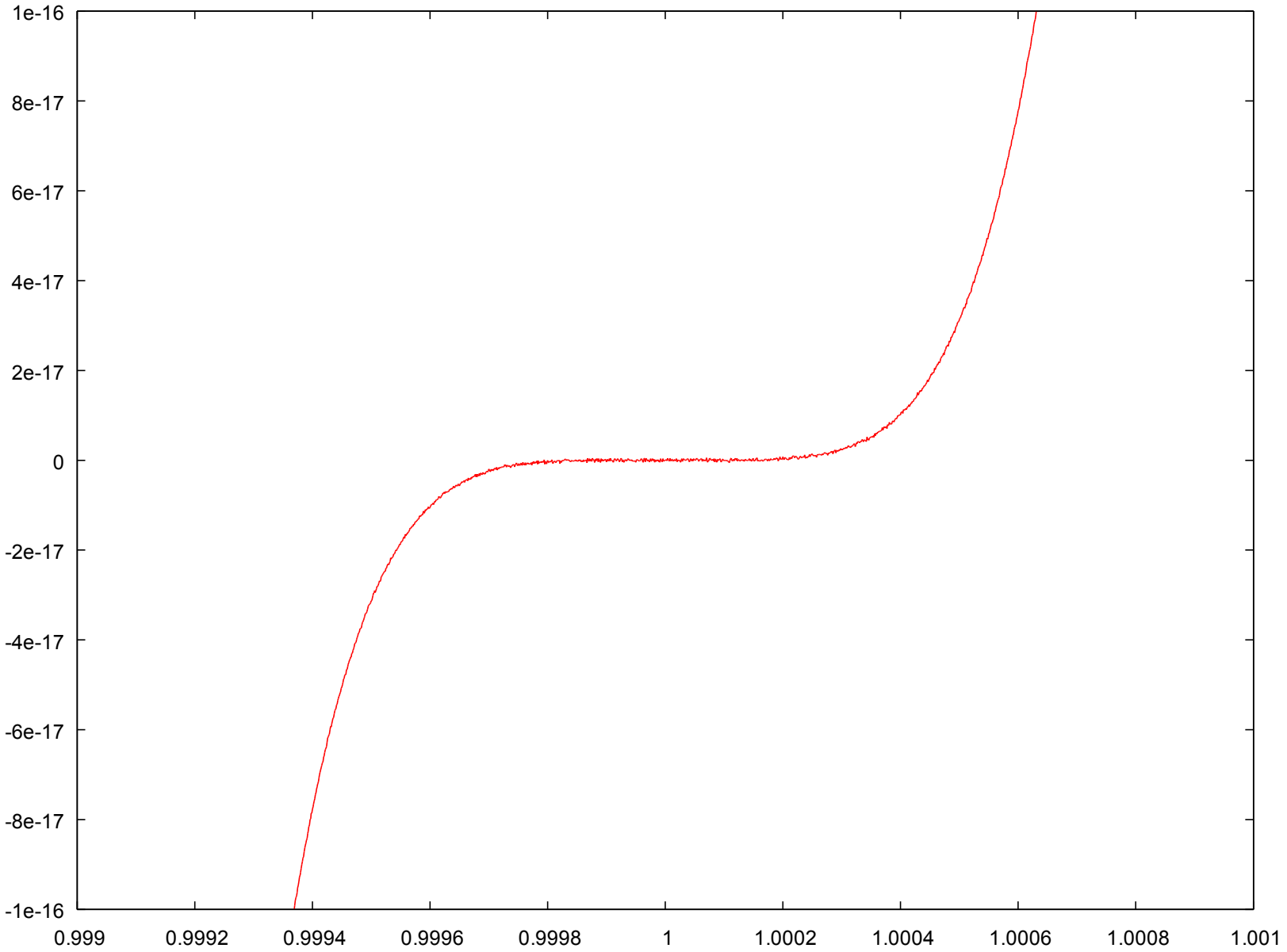
Of course, we actually evaluate this as

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

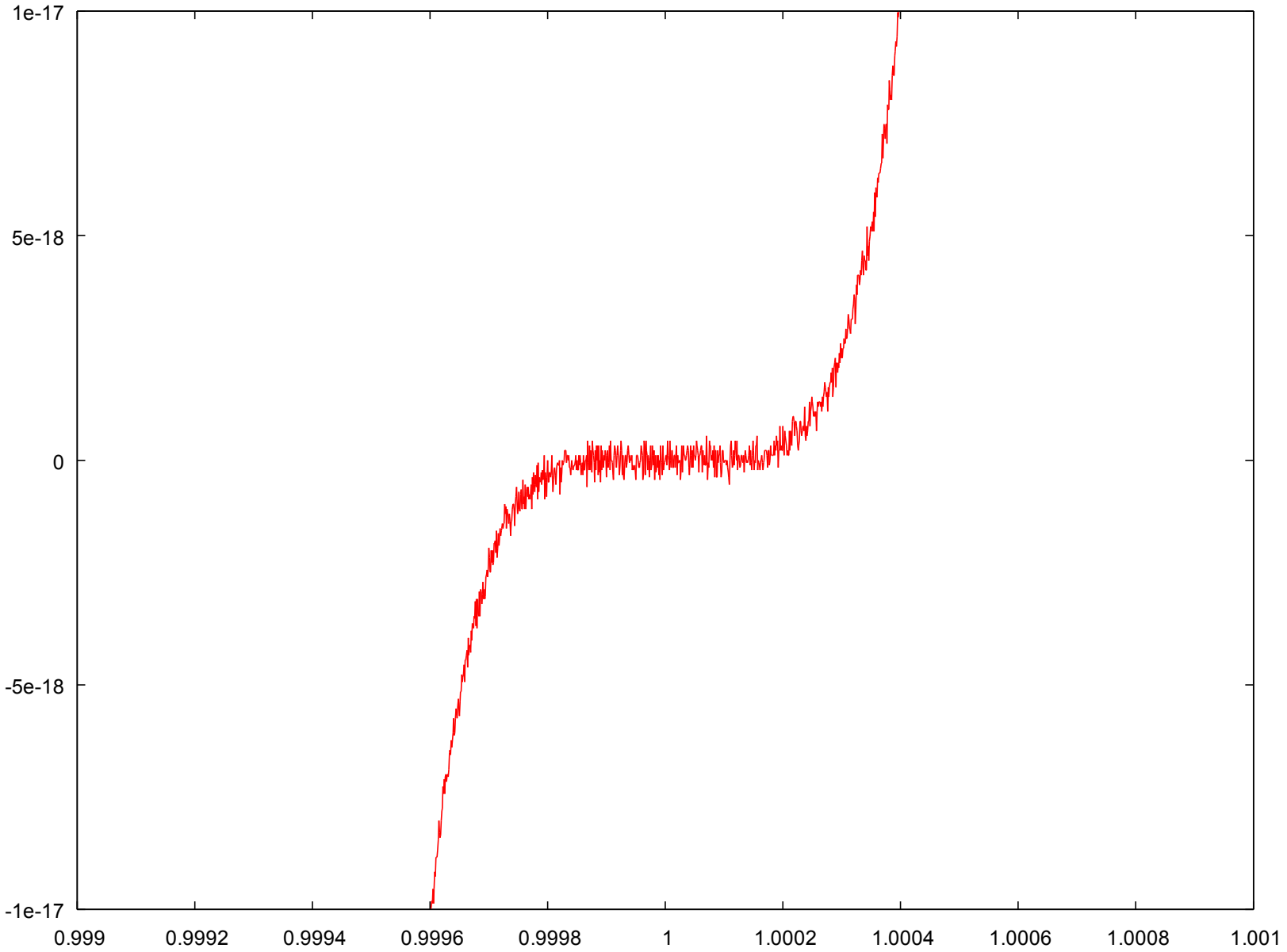
Region between x=0.999 and x=1.001



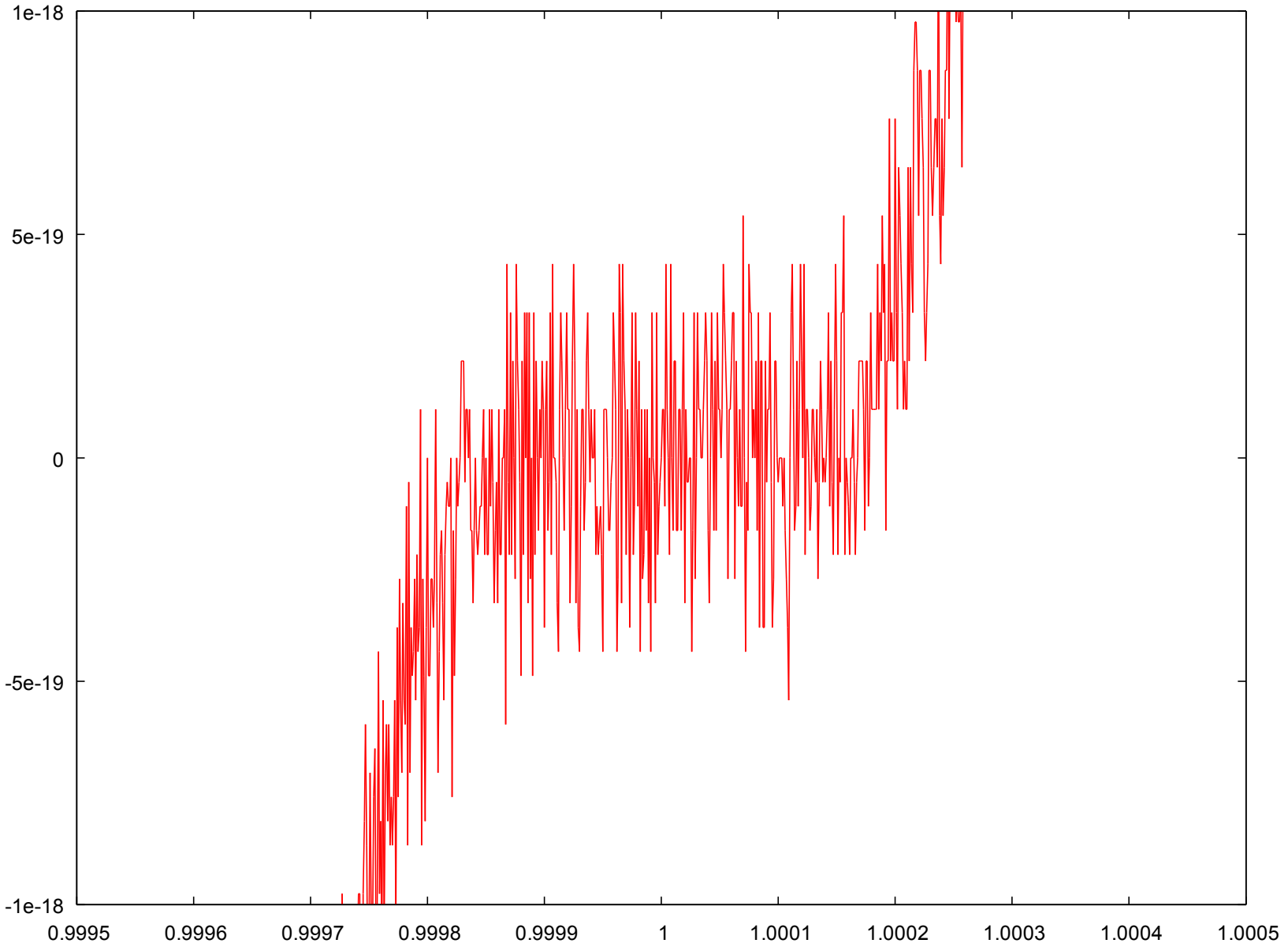
Zooming in on y-axis



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