# Computer Representation of Numbers

Deepayan Sarkar

## Computer representation of numbers

- Statistical computations mostly deal with numerical data
- The numbers we work with are usually integers ($\mathbb{N}, \mathbb{Z}$) or real numbers ($\mathbb{R}$)
- These are infinite sets, but computers have "finite" storage!
- Natural questions:
  - Which numbers can computers store?
  - How are they stored?
  - What happens when a calculation results in a number that cannot be stored?

## What could be possible models?

- Design constraints
  - Finite storage
  - Physical representation needs to be encoded using 0/1
- Some terminology:
  - Bit : "binary digit" — basic unit of representation; can be either 0 or 1
  - Byte : 8 bits; by convention, this is the smallest unit that can be manipulated
- Motivation: How does the decimal system work?
- Non-negative Integers:

$$\sum_{i=0}^{n-1} d_i \times 10^i = d_{n-1} d_{n-2} \dots d_1 d_0$$

- Signed integers: Same as above, along with a sign
- Fractions (between 0 and 1):

$$\sum_{i=1}^{n} d_i \times 10^{-i} = 0.d_1 d_2 \dots d_{n-1} d_n$$

- General real numbers:

$$\sum_{i=-m}^{n} d_i \times 10^i$$

## Integers

- Can we adapt this to use binary digits?
- Not very difficult for integers
- Unsigned integers (n bits):

$$\sum_{i=0}^{n-1} d_i \times 2^i = d_{n-1} d_{n-2} \ldots d_1 d_0$$

- For example, the number `20` (decimal) in binary is `10100`
- This is in fact how positive integers are usually stored
- How can we verify this?
- No built-in tool in R
- Easy with some semi-advanced C, using pointers and bitwise operators
- Instead, we will use a convenient built-in function in Julia

```
bitstring(20)
```

```
"0000000000000000000000000000000000000000000000000000000000010100"
```

## Integer types

- If we count, we will see that the representation has 64 bits
- Why 64? Summary:
  - By convention, the basic memory storage unit is 8 bits (a byte)
  - This has a long history, but basically became standard in the 1970s
  - Subsequently, integer representations of 8, 16, 32, and 64 bits have become standard
  - These are also commonly referred to as "types"
- Unfortunately, these types traditionally have somewhat confusing names; e.g., in C,
  - `char` : 8-bit integer; named `char` because of correspondence with ASCII
  - `short int` : 16-bit integer
  - `int` : At least 16-bit, but usually 32-bit integer
  - `long int` : At least 32-bit integer
  - `long long int` : At least 64-bit integer
- A more modern terminology, used by Julia (and in C / C++ using the stdint.h header):
  - `Int8`, `Int16`, `Int32`, `Int64` : `IntN` represents `N`-bit integer
  - `UInt8`, `UInt16`, `UInt32`, `UInt64`: Corresponding *unsigned* types (non-negative integers only)

```
bitstring(convert(UInt8, 20))
```

```
"00010100"
```

```
[bitstring(convert(UInt8, x)) for x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
```

```
10-element Array{String,1}:
 "00000001"
 "00000010"
 "00000011"
 "00000100"
 "00000101"
 "00000110"
 "00000111"
 "00001000"
 "00001001"
 "00001010"
```

### Representation of unsigned integers

- Clearly, we can only store a finite number of integers with $n$ bits (specifically, $2^n$)

```
typemin(UInt8)
```

```
0x00
```

```
typemax(UInt8)
```

```
0xff
```

```
bitstring(typemax(UInt8))
```

```
"11111111"
```

- Numbers prefixed with `0x` means hexadecimal coding, with 4-bit digits `0123456789abcdef` meaning 0–16
- How can we add two numbers?
- Simply add bit by bit, with $1 + 1 = 0$ carrying 1, and $1 + 1 + 1 = 1$ carrying 1.

```
 111

001110 +   (14)
000111 =   (07)

010101     (21)
```

- In Julia,

```
function add8(x, y)
    convert(UInt8, x) + convert(UInt8, y)
end
```

```
add8 (generic function with 1 method)
```

```
[bitstring(convert(UInt8, x)) for x = [14, 7, add8(14, 7)]]
```

```
3-element Array{String,1}:
 "00001110"
 "00000111"
 "00010101"
```

- What happens if we add 1 to the maximum possible value?

```
bitstring(typemax(UInt8))
```

```
"11111111"
```

```
bitstring(add8(typemax(UInt8), convert(UInt8, 1)))
```

```
"00000000"
```

- This is known as "overflow"
- For efficiency (to minimize time needed to do checking), many systems will silently overflow without informing the user that something might have gone wrong

## Representation of signed integers

- How do we store signed integers?
  - By convention, the first (leftmost) bit is used for the sign
  - In addition, negative numbers are stored using a convention known as "two's complement"
  - This stores an $(N-1)$-bit negative number as the result of subtracting the number from $2^N$
  - Advantages: addition, multiplication, etc., are performed using the same algorithm as unsigned integers
  - Also, zero has a unique representation, so $2^N$ distinct numbers can be stored

```julia
[bitstring(convert(Int8, x)) for x = -8:8]
```

```
17-element Array{String,1}:
 "11111000"
 "11111001"
 "11111010"
 "11111011"
 "11111100"
 "11111101"
 "11111110"
 "11111111"
 "00000000"
 "00000001"
 "00000010"
 "00000011"
 "00000100"
 "00000101"
 "00000110"
 "00000111"
 "00001000"
```

## Real numbers - floating point representation

- Numerical computations usually require working with "real numbers"
- Analogous to decimal representation, we can think of them as binary numbers of the form

$$b_1 b_2 \dots b_k \; [.] \; b_{k+1} \dots b_n$$

- We could perhaps store this as the pair $(b_1 \dots b_n, k)$
- This is actually fairly close to what is done in practice
- The numbers that can be represented exactly have the form

$$\text{significand} \times \text{base}^{\text{exponent}}$$

- For example,

$$1.2345 = 12345 \times 10^{-4}$$

- Or, with base=2 and binary digits,

$$110.1001 = 1.101001 \times 2^{10}$$

- This is known as the *floating point representation*
- Note that in binary, the first non-zero digit in the significand is redundant, as it must be 1 (except for 0)
- We still need to decide how to store the significand and the exponent
- Modern computers have two standard storage conventions for floating point representations:
  - 32-bit : known as single precision / `float` / `Float32`
  - 64-bit : known as double precision / `double` / `Float64`
- The conventions are detailed in the IEEE 754 standard
- Summarized in the following table

| | Float32 | Float64 |
|---|---|---|
| sign | 1 | 1 |
| exponent | 8 | 11 |
| fraction | 23 (+1) | 52 (+1) |
| total | 32 | 64 |
| exponent offset | -127 | -1023 |

- For example, bits in a `Float64` is laid out in the following way:

$$b_{63}\ b_{62} \dots b_{52}\ b_{51} \dots b_0$$

- where,
  - $s = b_{63}$ is the sign bit,
  - $e = b_{62} \dots b_{52}$ is the exponent interpreted as an 11-bit unsigned integer,
- The number represented is calculated as

$$(-1)^s (1.b_{51} \dots b_0) \times 2^{e-1024}$$

- Note that the fraction has an implicit 1 before the binary point that is not explicitly stored
- This is a form of "normalization" that
  - Ensures uniqueness of the representation, and
  - implicitly allows an extra bit of precision.
- The minimum (0) and maximum (2047) possible value of $e$ are reserved for special use
- They are used as representations for
  - Special numbers $\pm\infty$, `NaN`, $\pm 0$, and
  - "Subnormal" numbers between 0 and $1.0 \times 2^{-1023}$
  - With usual interpretation of $e = 0$, the smallest representable positive number would be $2^{-1023}$

- Non-terminating representations

```
bitstring(0.1)
```

"0011111110111001100110011001100110011001100110011001100110011010"

```
bitstring(0.1 + 0.1 + 0.1)
```

"0011111111010011001100110011001100110011001100110011001100110100"

```
bitstring(0.6 / 2)
```

"0011111111010011001100110011001100110011001100110011001100110011"

```
bitstring(0.3)
```

"0011111111010011001100110011001100110011001100110011001100110011"

- 0 and subnormal numbers ($e = 0$)

```
bitstring(0.0)
```

"0000000000000000000000000000000000000000000000000000000000000000"

```
bitstring(-0.0)
```

"1000000000000000000000000000000000000000000000000000000000000000"

```
bitstring(0.125) # exact (terminating) binary expansion
```

"0011111111000000000000000000000000000000000000000000000000000000"

```
bitstring(0.125 * 1.0 * 2.0^(-1023))
```

"0000000000000001000000000000000000000000000000000000000000000000"

- `Inf` and `NaN` ($e = 2047$)

```
bitstring(Inf)
```

"0111111111110000000000000000000000000000000000000000000000000000"

```
bitstring(-Inf)
```

"1111111111110000000000000000000000000000000000000000000000000000"

```
bitstring(NaN)
```

"0111111111111000000000000000000000000000000000000000000000000000"

- Note in the above example that `0.1 + 0.1 + 0.1` has a different representation than `0.3`
- Binary representation of `0.1`, `0.2`, `0.3`, etc., are recurring, and cannot be represented exactly
- When represented as floating point numbers, they need to be approximated
- Ideally, approximation should be the nearest representable number
- This does not always happen in practice
- Depending on intermediate calculations, results that are supposed to be the same may not be

```
0.2 + 0.1 == 0.4 - 0.1
```

true

```
0.2 + 0.1 == 0.6 / 2
```

false

```
0.2 + 0.1
```

```
0.30000000000000004
```

- Note that in the representation (of what is supposed to be 0.3) is an approximation in all cases
- The equality tests above only check whether two approximations derived differently agree or not
- The same behaviour is seen in python

```
print(0.2 + 0.1 == 0.4 - 0.1)
```

```
True
```

```
print(0.2 + 0.1 == 0.6 / 2)
```

```
False
```

```
print(0.2 + 0.1)
```

```
0.30000000000000004
```

- As well as R

```
0.2 + 0.1 == 0.4 - 0.1
```

```
[1] TRUE
```

```
0.2 + 0.1 == 0.6 / 2
```

```
[1] FALSE
```

```
0.2 + 0.1
```

```
[1] 0.3
```

- Except that R tries to be "user-friendly" and rounds the result before printing (to 7 digits by default)

```
print(0.2 + 0.1, digits = 22)
```

```
[1] 0.3000000000000000444089
```

## Consequences

- In the case of integer calculations, unreported overflow is the most common problem
- For example, in Julia, defining:

```
function Factorial(x)
    if (x == 0) tmp = 1
    else tmp = x * Factorial(x-1)
    end
    println(tmp)
    tmp
end
```

```
Factorial (generic function with 1 method)
```

```
Factorial(25)
```

```
1
1
2
6
24
120
720
5040
```

```
40320
362880
3628800
39916800
479001600
6227020800
87178291200
1307674368000
20922789888000
355687428096000
6402373705728000
121645100408832000
2432902008176640000
-4249290049419214848
-1250660718674968576
8128291617894825984
-7835185981329244160
7034535277573963776
```

```
7034535277573963776
```

## Consequences (floating point version)

```
Factorial(25.0)
```

```
1
1.0
2.0
6.0
24.0
120.0
720.0
5040.0
40320.0
362880.0
3.6288e6
3.99168e7
4.790016e8
6.2270208e9
8.71782912e10
1.307674368e12
2.0922789888e13
3.55687428096e14
6.402373705728e15
1.21645100408832e17
2.43290200817664e18
5.109094217170944e19
1.1240007277776077e21
2.585201673888498e22
6.204484017332394e23
1.5511210043330986e25
```

```
1.5511210043330986e25
```

## Integer overflow in R

- R behaves similarly, except that it detects integer overflow
- The `1L` in the code is to force integer calculations when `x` is integer
- In R, the literal `1` is interpreted as a floating point value and `1L` as integer

```
Factorial <- function(x) {
    if (x == 0) {
        tmp <- 1L
    }
    else {
        tmp <- x * Factorial(x-1L)
    }
    print(tmp)
    tmp
}
```

```
Factorial(15L)
```

```
[1] 1
[1] 1
[1] 2
[1] 6
[1] 24
[1] 120
[1] 720
[1] 5040
[1] 40320
[1] 362880
[1] 3628800
[1] 39916800
[1] 479001600
```

```
Warning in x * Factorial(x - 1L): NAs produced by integer overflow
```

```
[1] NA
[1] NA
[1] NA
```

```
[1] NA
```

## Floating point version

```
Factorial(25.0)
```

```
[1] 1
[1] 1
[1] 2
[1] 6
[1] 24
[1] 120
[1] 720
[1] 5040
[1] 40320
[1] 362880
[1] 3628800
[1] 39916800
```

```
[1] 479001600
[1] 6227020800
[1] 87178291200
[1] 1.307674e+12
[1] 2.092279e+13
[1] 3.556874e+14
[1] 6.402374e+15
[1] 1.216451e+17
[1] 2.432902e+18
[1] 5.109094e+19
[1] 1.124001e+21
[1] 2.585202e+22
[1] 6.204484e+23
[1] 1.551121e+25
```

```
[1] 1.551121e+25
```

## Integer (non)-overflow in Python

- Python has more interesting behaviour
- It detects integer overflow and automatically
- Shifts to using a less efficient but higher precision representation

```python
def Factorial(x):
    if x == 0:
        tmp = 1
    else:
        tmp = x * Factorial(x-1)
    print(tmp)
    return tmp
```

```python
Factorial(25)
```

```
1
1
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
6227020800
87178291200
1307674368000
20922789888000
355687428096000
6402373705728000
121645100408832000
2432902008176640000
51090942171709440000
```

```
1124000727777607680000
25852016738884976640000
620448401733239439360000
15511210043330985984000000
15511210043330985984000000
```

### Floating point version

```
Factorial(25.0)
```

```
1
1.0
2.0
6.0
24.0
120.0
720.0
5040.0
40320.0
362880.0
3628800.0
39916800.0
479001600.0
6227020800.0
87178291200.0
1307674368000.0
20922789888000.0
355687428096000.0
6402373705728000.0
1.21645100408832e+17
2.43290200817664e+18
5.109094217170944e+19
1.1240007277776077e+21
2.585201673888498e+22
6.204484017332394e+23
1.5511210043330986e+25
1.5511210043330986e+25
```

### Floating point arithmetic

- In practice, numerical calculations are done using floating point arithmetic
- Possible problems here are more subtle
- One obvious limitation:
  - numbers of much larger magnitude can be represented, but
  - only the first few significant digits are actually stored
- So, in all the examples above, we get something like

```
x <- factorial(25.0) # built-in factorial function in R
x
```

```
[1] 1.551121e+25
```

```
x == x + 1
```

```
[1] TRUE
```

- Given a value, how far away the closest representable value is depends on the value
- In Julia, `eps(x)` is such that `x + eps(x)` is the next representable value larger than `x`

```
eps(1.0e-27)
```

```
1.793662034335766e-43
```

```
eps(1.55e+25)
```

```
2.147483648e9
```

```
eps(0.0)
```

```
5.0e-324
```

```
eps(1.0)
```

```
2.220446049250313e-16
```

```
eps(1000.0)
```

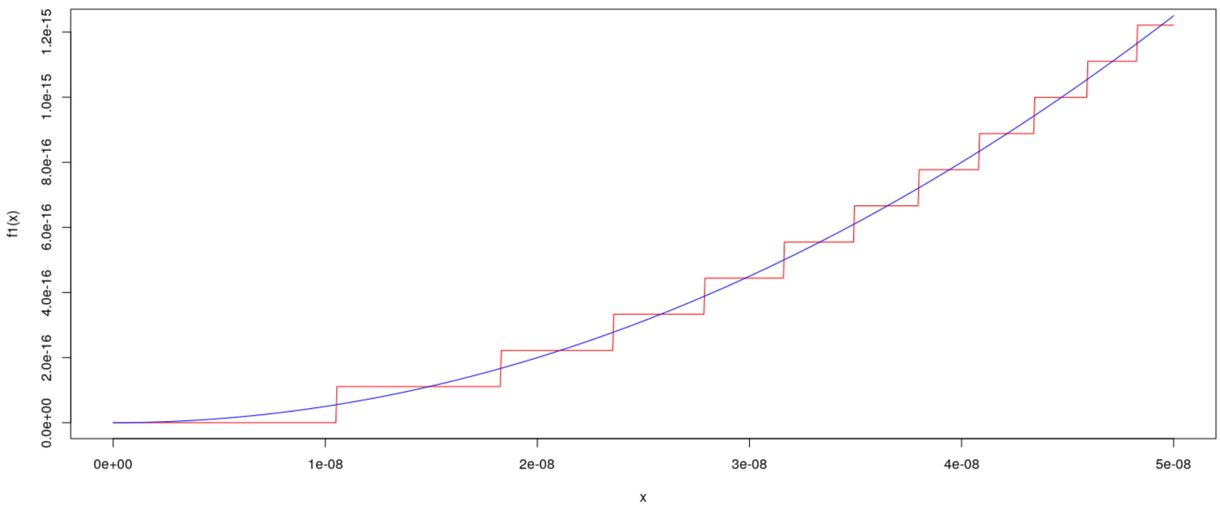
```
1.1368683772161603e-13
```

- Another extreme example of this behaviour is the following
- Consider the mathematical identity

$$f(x) = 1 - \cos(x) = \sin^2(x)/(1 + cos(x))$$

- Suppose that we are interesting in evaluating $f(x)$ for a given value of $x$
- In theory, we can use either formula
- In practice, near $x = 0$, $\cos(x)$ is very close to 1, so $1 - \cos(x)$ loses precision

```
f1 <- function(x) { 1 - cos(x) }
f2 <- function(x) {
    u <- sin(x)
    (u * u) / (1 + cos(x))
}
curve(f1, from = 0.0, to = 5e-8, n = 1001, col = "red")
curve(f2, from = 0.0, to = 5e-8, add = TRUE, n = 1001, col = "blue")
```




- Why does this happen?
- This is partly due to intrinsic limitations, but also partly due to choice of formula
- It is actually not very difficult to model this kind of behaviour formally
- We will not go into detail, but only cover the basic concepts