Numerics

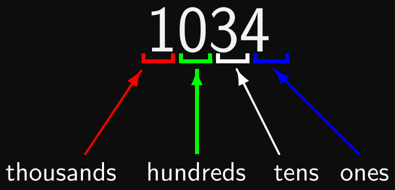
Ramses van Zon

PHY1610, Winter 2023

Numbers

How do we represent quantities?

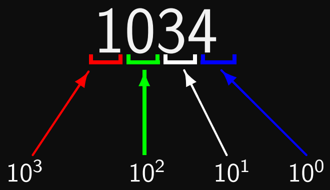
- We use numbers, of course.
- In grade school we are taught that numbers are organized in columns of digits. We learn the names of these columns.
- The numbers are understood as multiplying the digit in the column by the number that names the column.



$$1034 = (1 \times 1000) + (0 \times 100) + (3 \times 10) + (4 \times 1)$$

Other ways to represent a quantity

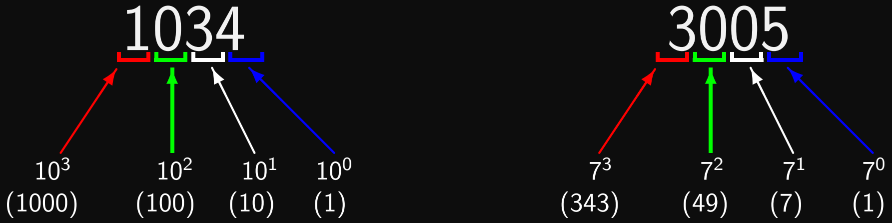
- Instead of using 'tens' and 'hundreds', let's represent the columns by powers of what we will call the 'base'.
- Our normal way of representing numbers is 'base 10', also called decimal.
- Each column represents a power of ten, and the coefficient can be one of 10 numerals (0-9).



$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

You can choose any base you want

How do we represent the quantity 1034 if we change bases? What about base 7 (septenary)?



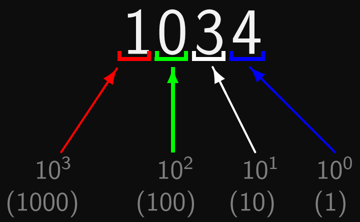
$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

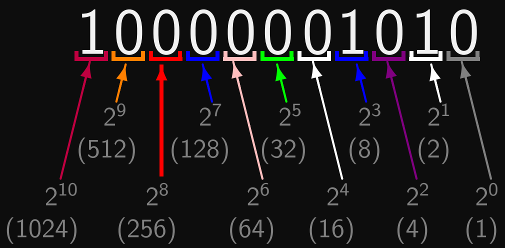
$$1034 = (3 \times 7^3) + (0 \times 7^2) + (0 \times 7^1) + (5 \times 7^0)$$

In base 7 the numerals have the range 0-6.

Who cares?

The reason we care is because computers do not use base 10 to store their data. Computers use base 2 (binary). The numerals have the range 0-1.





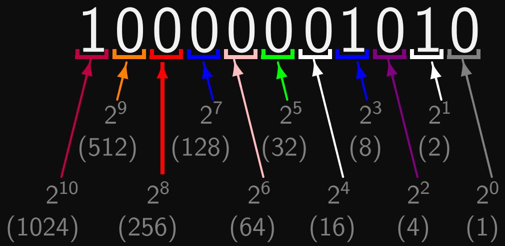
$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

$$\begin{aligned} 1034 \quad = \quad & (1 \times 2^{10}) + (0 \times 2^9) + (0 \times 2^8) + (0 \times 2^7) \\ & +(0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) \\ & +(0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \end{aligned}$$

Why do computers use binary numbers?

Why use binary?

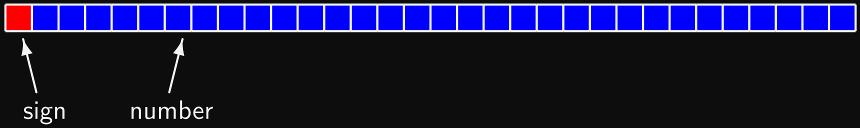
- Modern computers operate using circuits that have one of two states: ‘on’ or ‘off’.
- This choice is related to the complexity and cost of building binary versus ternary circuitry.
- Binary numbers are like series of ‘switches’: each digit is either ‘on’ or ‘off’.
- Each ‘switch’ in the number is called a ‘bit’.



Finite Binary Representations of Numeric Types

Integers

- All integers are exactly representable.
- Different sizes of integer variables are available, depending on your hardware, OS, and programming language.
- For most languages, a typical integer is 32 bits,
- 1 bit for the sign, which, when set, subtracts 2^32 of the number.
- Finite range: can go from -2^{31} to $2^{31} - 1$ (-2,147,483,648 to 2,147,483,647).
- Unsigned integers: $0 \ldots 2^{32} - 1$.
- All operations ($+$, -, *) between representable integers are represented unless there is overflow.

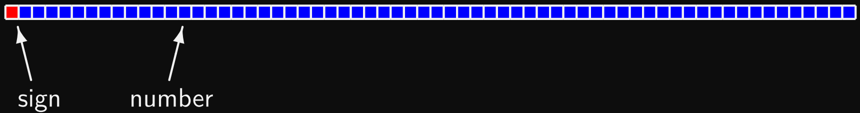


A typical int $=$ 32 bits $=$ 4 bytes.

Long integers

- Long integers are like regular integers, just with a bigger memory size, usually 64 bits.
- And consequently a bigger range of numbers.

- One bit for sign.
(when set, subtracts $2^6$4 of the number)
- Can go from -2^{63} to $2^{63} - 1$
(-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
- Unsigned long integers: $0 ... 2^{64} - 1$.



A typical long int = 64 bits = 8 bytes.

Integers in C++

Type	Typical size
char	1 byte
short	2 bytes
int	4 (2) bytes
long	8 bytes
long long	8 bytes (C++11)

```
char c;
short int si;         // valid
short s;              // preferred
int i;
long int li;          // valid
long l;               // preferred
long long int lli;    // valid
long long ll;         // preferred
signed char c;
signed short s;       // unnecessary
signed int i;         // unnecessary
signed long l;        // unnecessary
signed long long ll;// unnecessary
unsigned char c;
unsigned short s;
unsigned int i;
unsigned long l;
unsigned long long ll;
```

Size/Type	Range
1 byte signed	-128 .. 127
1 byte unsigned	0 .. 255
2 byte signed	-32,768 .. 32,767
2 byte unsigned	0 .. 65,535
4 byte signed	-2,147,483,648...2,147,483,647
4 byte unsigned	0 .. 4,294,967,295
8 byte signed	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807
8 byte unsigned	0..18,446,744,073,709,551,615

Integer OverFlow

```cpp
#include <iostream>

int main()
{
    using namespace std;
    unsigned short x = 65535; // largest 16-bit unsigned value possible
    cout << "x was: " << x << endl;
    x = x + 1; // 65536 is out of our range -- we get overflow because x can't hold 17 bits
    cout << "x is now: " << x << endl;
    return 0;
}
```

```
$ g++ -std=c++17 int_exampleOF1.cpp
$ ./a.out
x was: 65535
x is now: 0
```

```cpp
#include <iostream>

int main()
{
    using namespace std;
    unsigned short x = 0; // smallest 2-byte unsigned value possible
    cout << "x was: " << x << endl;
    x = x - 1; // overflow!
    cout << "x is now: " << x << endl;
    return 0;
}
```

```
$ g++ -std=c++17 int_exampleOF2.cpp
$ ./a.out
x was: 0
x is now: 65535
```

Fixed point numbers

How do we deal with decimal places?

- We could treat real numbers like integers: 0 . . . INT_MAX, and only keep, say, the last two digits behind the decimal point.
- This is known as 'fixed point' numbers, since the decimal place is always in the same spot.
- This is often used for financial timeseries data, since they only use a finite number of decimal places.
- But this is terrible for scientific computing. Relative precision varies with magnitude; we need to be able to represent small and large numbers at the same time.

Floating point numbers

Floating point numbers

- Analog of numbers in scientific notation.
- Inclusion of an exponent means the decimal point is 'floating'.
- Again, one bit is dedicated to sign.

A single precision real = 32 bits = 4 bytes.
A double precision real = 64 bits = 8 bytes.

Floats in C++

Type	Size
float	4 bytes
double	8 bytes
long double	8/12/16 bytes

```
float fValue;
double dValue;
long double dValue2;

int n(5); // 5 means integer
double d(5.0); // 5.0 means fp (double by default)
float f(5.0f); // 5.0 means fp, f suffix means float

double d1(5000.0);
double d2(5e3); // another way to assign 5000
double d3(0.05);
double d4(5e-2); // another way to assign 0.05
```

Size/Type	Range	Precision
4 bytes	$\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$	6-9 sign.digits, typically 7
8 bytes	$\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$	15-18 sign.digits, typ. 16
12 bytes	$\pm 3.65 \times 10^{-4951}$ to $\pm 1.18 \times 10^{4932}$	18-21 significant digits
16 bytes	$\pm 3.36 \times 10^{-4932}$ to $\pm 1.18 \times 10^{4932}$	33-36 significant digits

Special ‘numbers’ & Numeric Limits Interface

Special numbers

This format for storing floating point numbers comes from the IEEE 754 standard.

There's room in the format for the storing of a few special numbers.

- Signed infinities **(+Inf, -Inf)**: result of overflow, or divide by zero.
- Signed zeros: signed underflow, or divide by +/-**Inf**.
- Not a Number (**NaN**): square root of a negative number, 0/0, Inf/Inf, *etc.*
- The events which lead to these are usually errors, and can be made to cause exceptions.

#include <limits>

```
template<> class numeric_limits<bool>;
template<> class numeric_limits<char>;
template<> class numeric_limits<signed char>;
template<> class numeric_limits<unsigned char>;
template<> class numeric_limits<wchar_t>;
template<> class numeric_limits<char16_t>; // C++11
template<> class numeric_limits<char32_t>; // C++11
template<> class numeric_limits<short>;
template<> class numeric_limits<unsigned short>;
```

```
template<> class numeric_limits<int>;
template<> class numeric_limits<unsigned int>;
template<> class numeric_limits<long>;
template<> class numeric_limits<unsigned long>;
template<> class numeric_limits<long long>;
template<> class numeric_limits<unsigned long long>;
template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
template<> class numeric_limits<long double>;
```

Member Functions

`min`	returns the smallest finite value of the given type
`lowest`	(C++11) returns the lowest finite value of the given type
`max`	returns the largest finite value of the given type
`epsilon`	returns the difference between 1.0 and the next representable value of the given floati
`round_error`	returns the maximum rounding error of the given floating-point type
`infinity`	returns the positive infinity value of the given floating-point type
`quiet_NaN`	returns a quiet NaN value of the given floating-point type
`signaling_NaN`	returns a signaling NaN value of the given floating-point type
`denorm_min`	returns the smallest positive subnormal value of the given floating-point type

C++ Numeric Limits – example

```cpp
#include <limits>
#include <iostream>

int main()
{
    std::cout << "type\tlowest\thighest\n";
    std::cout << "int\t"
              << std::numeric_limits<int>::lowest() << '\t'
              << std::numeric_limits<int>::max() << '\n';
    std::cout << "float\t"
              << std::numeric_limits<float>::lowest() << '\t'
              << std::numeric_limits<float>::max() << '\n';
    std::cout << "double\t"
              << std::numeric_limits<double>::lowest() << '\t'
              << std::numeric_limits<double>::max() << '\n';
}
```

```
$ g++ -std=c++17 limits.cpp
$ ./a.out
type    lowest          highest
int     -2147483648     2147483647
float   -3.40282e+38    3.40282e+38
double  -1.79769e+308   1.79769e+308 }
```

IEEE-754/854 Standards

`float` (single precision real) = 32 bits = 4 bytes $\rightsquigarrow$ IEEE-754

`double` (double precision real) = 64 bits = 8 bytes $\rightsquigarrow$ IEEE-854

Property	Value for float	Value for double
Largest rep.nbr.	3.402823466e+38	1.7976931348623157e+308
Smallest nbr. without losing precision	1.175494351e-38	2.2250738585072014e-308
Smallest rep.nbr.	1.401298464e-45	5e-324
Mantissa bits	23	52
Exponent bits	8	11
Epsilon	1.1929093e-7	2.220446049250313e-16

ieee754.h

Example of a routine to tell if two floats are equal to a certain number of significant decimal digits:

```
#include <ieee754.h>
#include <cmath>
bool flt_equals(float a, float b, int sigfigs)
{
    if (a == b)
        return true;
    union ieee754_float
        *pa = reinterpret_cast<union ieee754_float*>(&a),
        *pb = reinterpret_cast<union ieee754_float*>(&b);
    unsigned int
        aexp = pa->ieee.exponent,
        bexp = pb->ieee.exponent;
    if (aexp != bexp or pa->ieee.negative != pb->ieee.negative)
        return false;
    pa->ieee.exponent = pb->ieee.exponent = IEEE754_FLOAT_BIAS;
    float sig_mag = pow(10, -(float)sigfigs);
    if (fabs(a-b) < sig_mag/2)
        return true;
    return false;
}
```

Note: The header file `ieee754.h` might not exist in your system.

Limitations in floating point mathematics

There are limitiations inherent in using finite-length floating point variables.

- Except for numbers that fit exactly into a base two representation, assigning a real number to a floating point variable involves truncation.
- Think about how you represent 1/3. Is it 0.3? 0.33? 0.333?
- You end up with an error of 1/2 ULP
  (Unit in Last Place)

Unrepresentable numbers

In base two, 0.1 is an infinitely repeating fraction: 0.000110011001100110011...

So this cannot be represented exactly in finite binary!

Limited accuracy

Single precision: 1 part in $2^{-24} \sim$ 6e-8.
Double precision: 1 part in $2^{-53} \sim$ 1e-16.

Aside: want to see more digits in output?

```
#include <iostream>
#include <iomanip>

void output(float f, double d) {
  std::cout << "f = " << f << '\n';
  std::cout << "d = " << d << '\n';
  std::cout << "f+d = " << f+d << '\n';
  std::cout << "f-d = " << f-d << '\n';
  std::cout << "f*d = " << f*d << '\n';
  std::cout << "f/d = " << f/d << '\n';
  std::cout << "d/f = " << d/f << '\n';
}

int main()
{
  output(0.01f, 1.0e-17);
  // set fixed floating format
  std::cout.setf(std::ios::fixed);
  // change fixed format precision
  std::cout.precision(5);
  //
  output(0.01f, 1.0e-17);
}
```

```
$ g++ -std=c++17 fp_ariths.cpp
$ ./a.out
f = 0.01
d = 1e-17
f+d = 0.01
f-d = 0.01
f*d = 1e-19
f/d = 1e+15
d/f = 1e-15
f = 0.01000
d = 0.00000
f+d = 0.01000
f-d = 0.01000
f*d = 0.00000
f/d = 999999977648258.12500
d/f = 0.00000
```

Equality testing

Testing for equality

Never ever ever ever test for equality with floating point numbers!

- Because of rounding errors in floating point numbers, you don't know exactly what you're going to get.
- Instead, test to see if the *absolute difference* is below some *tolerance* that is near epsilon.
- Testing for equality with integers is ok, however, because integers are exact.

```
$ g++ -std=c++17 fp_tol.cpp
$ ./a.out
f*f = 0.01
g = 0.01
False
True
```

```cpp
#include <iostream>
#include <cmath>

int main()
{
    float f = 0.1;
    float g = 0.01;

    std::cout << "f*f = " << f*f << '\n'
              << "g = " << g << '\n';

    if (f*f == g)
        std::cout << "True" << '\n';
    else
        std::cout << "False" << '\n';

    float TOL=1e-7;
    if (fabs(f*f - g) < TOL)
        std::cout << "True" << '\n';
    else
        std::cout << "False" << '\n';
}
```

Roundoff errors

Roundoff error occurs when you're not being careful with which combinations of types of numbers you are operating on:

$$(a+b)+c \neq a+(b+c)$$

```
// oundOff.cpp
#include <iostream>
int main()
{
    double a = 1.0, b = 1.0, c = 1e-16;

    std::cout << (a - b) + c << std::endl;
    std::cout << a + (-b + c) << std::endl;
    return 0;
}
```

```
$ g++ -std=c++17 RoundOff.cpp
$ ./a.out
1e-16
1.11022e-16
```

Roundoff errors, continued

Roundoff errors can occur anytime you start operating near machine precision.

- *Machine precision* (or *machine epsilon*) is the upper bound on the relative error due to rounding. This is generally $\approx$ 1e-8 for single precision (float) and 1e-16 for double precision.
- Roundoff errors are most common when subtracting or dividing two non-integer numbers that are about the same size, thus forcing the computer to do arithmetic near machine epsilon.
- Do your best to modify your algorithms to avoid such calculations.

Machine epsilon

Machine epsilon

Let's do some addition, to demonstrate what could go wrong.

- Problem: $1.0 + 0.001$
- Let's work in base 10.
- Let's assume that we only have a mantissa precision of 3, and exponent precision of 2.

$$\begin{array}{r} 1.00 \times 10^{0} \\ +\,1.00 \times 10^{-3} \\ \hline 1.00 \times 10^{0} \\ +\,0.001 \times 10^{0} \\ \hline 1.00 \times 10^{0} \end{array}$$

- So what happened?
- Mantissa only has a precision of 3! The final answer is beyond the range of the mantissa!

Machine epsilon

- Machine epsilon* gives you the limits of the *precision* of the machine.
- It is defined to be the smallest x such that $1 + x \neq 1$.
(or sometimes, the largest x such that $1 + x = 1$.)
- *Machine epsilon* is named after the mathematical term for a small positive infinitesimal.

```
#include <iostream>
#include <cmath>

int main()
{
  float f = 1.0;
  float g = 1.e-18;

  std::cout << "f =" << f << '\n';
  std::cout << "g =" << g << '\n';

  std::cout << " (1. - 1.)+ 1.e-18 = " << (f-f)+g << '\n';
  std::cout << " (1. + 1.e-18) - 1.0 = " << (f+g)-f << '\n';
  std::cout << " (1. + 1.e-18)  = " << (f+g) << '\n';
}
```

```
$ g++ -std=c++17 fp_machEpsilon.cpp
$ ./a.out
f =10
g =1e-18
(1. - 1.)+ 1.e-18 = 1e-18
(1. + 1.e-18) - 1.0 = 0
(1. + 1.e-18)  = 1
```

Beware: subtraction

Be very wary of subtracting very similar numbers.

- Problem: subtract 1.22 from 1.23.
- Assume that we only have a mantissa precision of 3, and exponent precision of 2.
- By performing this subtraction, we eliminate most of the information, and end up with '*catastrophic cancellation*'.
- We go from 3 significant digits to 1.
- Dangerous in intermediate results.

The same problem can occur when dividing large numbers.

3 sig. digits

$$\begin{array}{r} 1.23 \times 10^0 \\ -\,1.22 \times 10^0 \\ \hline 1.00 \times 10^{-2} \end{array}$$

1 sig. digit

Overflow

Overflow occurs when the result of a calculation exceeds the representable range of the variable type.

- It can happen with different types of numerical types: real (FP), integers, ...
- E.g. 8-bit integers have a range of -128 to 127.
- E.g. 4-bytes floats have a range of $\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$.
- Always be sure to use variables that are big enough for what you're doing.

```cpp
#include <iostream>

int main()
{
  float f = 1.0e15;

  std::cout << "f =" << f << '\n';
  std::cout << "f*f =" << f*f << '\n';
  std::cout << "f*f*f =" << f*f*f << '\n';

}
```

```
$ g++ -std=c++17 fp_machEpsilon.cpp
$ ./a.out
f =1e+15
f*f =1e+30
f*f*f =inf
```

Underflow

An *underflow* error is the opposite of an overflow error: you are attempting to make a number which is smaller than the variable can hold.

- 32-bit floats integers have a range of -3.4e38 to +3.4e38: $(\pm 1.18 \times 10^{-38}, \pm 3.4 \times 10^{+38}]$
- An overflow error will result if you attempt to go beyond this range.
- An underflow error results if you try to go too small.

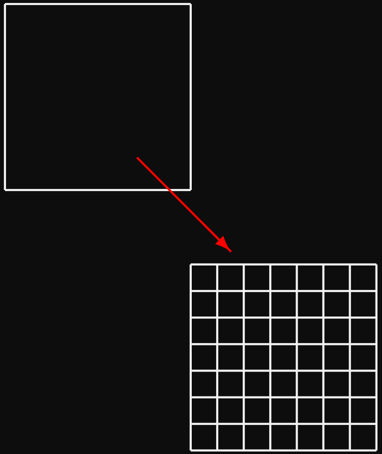
```cpp
#include <iostream>

int main()
{
  float f = -1.0e35;
  float g = -1.0e44;
  float h = 1.0e40;
  float k = 1.0e-44;
  float l = 1.0e-46;

  std::cout << "f =" << f << '\n';
  std::cout << "g =" << g << '\n';
  std::cout << "h =" << h << '\n';
  std::cout << "k =" << k << '\n';
  std::cout << "l =" << l << '\n';
}
```

```
$ g++ -std=c++17  fp_undFlow.cpp
$ ./a.out
f =-1e+35
g =-inf
h =inf
k =9.80909e-45
l =0
```

Discretizations error

What is discretization error? Where does it come from?

- In the real world space and time are continuous. But simulations and calculations are not.
- Variables must be converted from continuous to discrete.
- Space is sliced up into grids. Time is changed to steps.
- The density of the grids and steps goes up with increasing resolution.



Discretization errors, continued

Discretization error is the error introduced to a calculation by the act of discretizing the variables. What's the problem?

- As a source of error, you want to make sure that these errors are kept small; they cannot be avoided.
- One must be sure the grid density (resolution) is high enough that discretization errors are at an acceptable level.
- What resolution is high enough? This depends on what is being discretized (time versus space), the type of calculation, and other factors.
- There are relationships between the discretization of the various variables that need to be respected, to keep discretization errors under control (and to prevent numerical instabilities).

Summary: things to remember

- Integers are stored exactly.
- Floating point numbers are, in general, NOT stored exactly. Rounding error will cause the number to be slightly off.
- DO NOT test floating point numbers for equality. Instead test `(fabs(a - b) < cutoff)`
- Know the approximate value of epsilon for the machine that you are using.
- Know the limits of your precision: if your calculations span as many orders of magnitude as the inverse of epsilon you're going to lose precision.
- Try not to subtract floating point numbers that are very close to one another. '*Catastrophic cancellation*' leads to loss of precision.
- Be aware of overflow and underflow: use variable sizes that are appropriate for your problem.

r