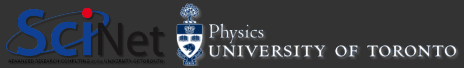
PHY1610H - Scientific Computing: Numerics & Numerical Errors

Ramses van Zon

SciNet HPC Consortium/Physics Department
University of Toronto

February 17, 2022



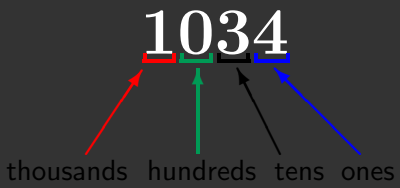
Today's class

Today we will discuss the following topics:

- Number Representations.
- How computers store different types of numbers.
- Classes of Numerical Errors.

How do we represent quantities?

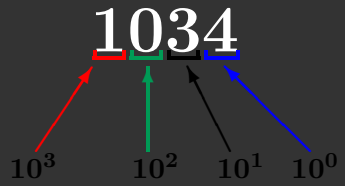
- We use numbers, of course.
- In grade school we are taught that numbers are organized in columns of digits. We learn the names of these columns.
- The numbers are understood as multiplying the digit in the column by the number that names the column.



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

Other ways to represent a quantity

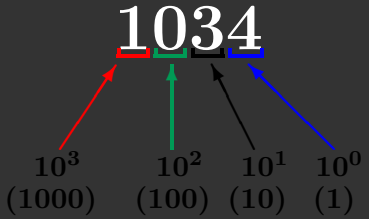
- Instead of using 'tens' and 'hundreds', let's represent the columns by powers of what we will call the 'base'.
- Our normal way of representing numbers is 'base 10', also called decimal.
- Each column represents a power of ten, and the coefficient can be one of 10 numerals (0-9).



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

You can choose any base you want

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

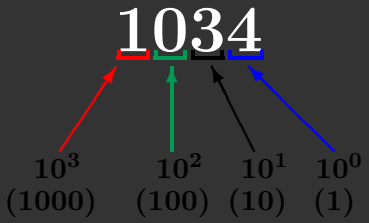


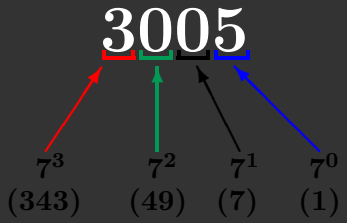
$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

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

You can choose any base you want

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





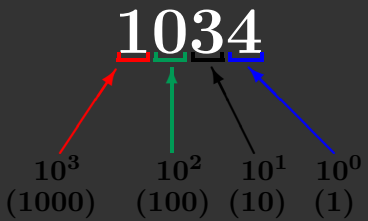
$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

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

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

Who cares?

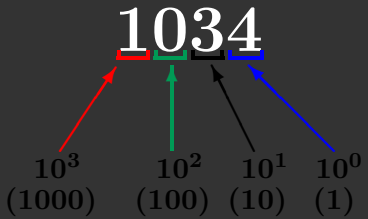
The reason we care is because computers do not use base 10 to store their data. Computers use base 2 (binary). The numerals have the range 0-1.

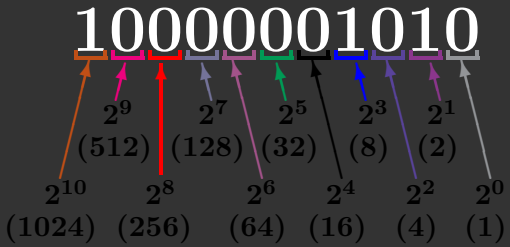


$$1034 = (1 \times 10^3) + (0 \times 10^2) + (3 \times 10^1) + (4 \times 10^0)$$

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.





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

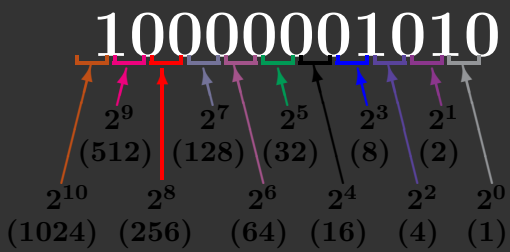
$$\begin{aligned} 1034 \;=\; & (1 \times 2^{10}) + (0 \times 2^9) + (0 \times 2^8) + (0 \times 2^7) \\ & + (0 \times 2^6) + (0 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) \\ & + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \end{aligned}$$

SciNet Physics UNIVERSITY OF TORONTO

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'.

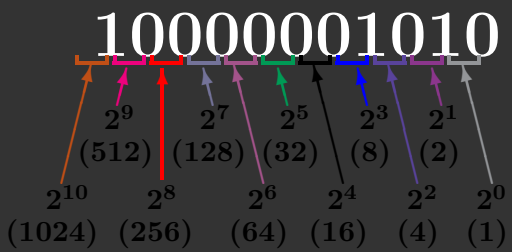


SciNet Physics UNIVERSITY OF TORONTO

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'.



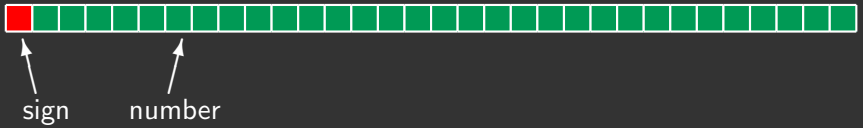
Pretend that each finger on one of your hands represents one bit. Count to 31 (2^5 - 1) on one hand in binary!

SciNet Physics UNIVERSITY OF TORONTO

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.

- 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.
- can go from -$\mathbf{2^{63}}$ to $\mathbf{2^{63} - 1}$
- -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- Unsigned long integers: $\mathbf{0 \ldots 2^{64} - 1}$.

A typical long int = 64 bits = 8 bytes.

Integers in C++

Type	Minimum size
char	1 byte
short	2 bytes
int	2 (4) bytes
long	4 bytes
long long	8 bytes (C99/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;
signed int i;
signed long l;
signed long long ll;

unsigned char c;
unsigned short s;
unsigned int i;
unsigned long l;
unsigned long long ll;
```

Integers in C++

Type	Minimum size
char	1 byte
short	2 bytes
int	2 (4) bytes
long	4 bytes
long long	8 bytes (C99/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;
signed int i;
signed long l;
signed long long ll;

unsigned char c;
unsigned short s;
unsigned int i;
unsigned long l;
unsigned long long ll;
```

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

Integer OverFlow

```
#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++ int_exampleOF1.cpp
```

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++ int_exampleOF1.cpp
$ ./a.out
```

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++ int_exampleOF1.cpp
$ ./a.out
x was: 65535
x is now: 0
```

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++ 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++ int_exampleOF2.cpp
```

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++ 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++ int_exampleOF2.cpp
$  ./a.out
```

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++ 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++ int_exampleOF2.cpp
$  ./a.out
x was:  0
x is now:  65535
```

Fixed point numbers

How do we deal with decimal places?

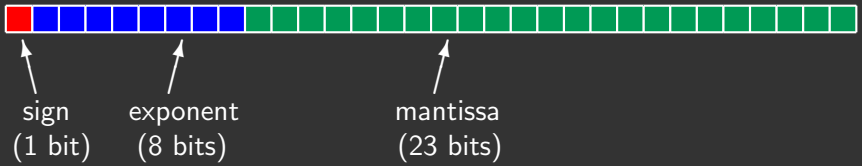
- We could treat real numbers like integers: 0 ... INT_MAX, and only keep, say, the last two digits behind the decimal point.
- This is known as 'fixed point' numbers, since the decimal place is always in the same spot.
- It is often used for financial timeseries data, since they only use a finite number 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

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

$$-1.34 \times 10^{-7}$$

sign mantissa base exponent



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

SciNet Physics UNIVERSITY OF TORONTO

Floats in C++

Type	Minimum size
float	4 bytes
double	8 bytes
long double	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

IEEE-754

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.

C Numeric Limits Interface

```
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 feature
template<> class numeric_limits<char32_t>; // C++11 feature
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 floating-point type
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 Interface – 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';
}
```

C Numeric Limits Interface – 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++14 limits.cpp
```

C Numeric Limits Interface – 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++14 limits.cpp
$  ./a.out
```

C Numeric Limits Interface – 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++14 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 ⇝ IEEE-754
`double` (double precision real) = 64 bits = 8 bytes ⇝ IEEE-854

```
 s|eeeeeeee|mmmmmmmmmmmmmmmmmmmmmmm     meaning
31                                0     bit #
```

`s` = sign bit, `e` = exponent, `m` = mantissa

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 that can tell you if two floats are equal to a certain number of significant decimal digits:

```c
#include <ieee754.h>

int flt_equals(float a, float b, int sigfigs)
{
    union ieee754_float *pa, *pb;
    unsigned int aexp, bexp;
    float sig_mag;

    if (a == b)
        return 1;
    pa = (union ieee754_float*)&a;
    pb = (union ieee754_float*)&b;
    aexp = pa->ieee.exponent;
    bexp = pb->ieee.exponent;
    if (aexp != bexp || pa->ieee.negative != pb->ieee.negative)
        return 0;
    pa->ieee.exponent = pb->ieee.exponent = IEEE754_FLOAT_BIAS;
    sig_mag = pow(10, -(float)sigfigs);
    if (fabs(a-b) < sig_mag/2)
        return 1;
    return 0;
}
```

DISCLAIMER:
Check if the header file ⟨ieee754.h⟩ exist in your system, e.g. use

```
locate ieee754.h
```

Errors in floating point mathematics

There are errors 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*).

In base two, 0.1 is an infinitely repeating fraction:
0.0001100110011001100110011...

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

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

void output(float f, double d);

int main()
{
  float f = 0.01;
  double d = 1.e-17;

  output(f,d);

  std::cout.setf(std::ios::fixed,std::ios::
       floatfield); // set fixed floating format
  std::cout.precision(5); // for fixed format, two
       decimal places

  output(f,d);
}

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

SciNet Physics UNIVERSITY OF TORONTO

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 **difference** is below some *tolerance* that is near epsilon.
- Testing for equality with integers is ok, however, because integers are exact.

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

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

  g = f*f;

  std::cout << "g = f*f" << g << std::endl;

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

  float TOL=1e-15;
    if (cmath::abs(f*f - g) < TOL)
     std::cout << "True" << std::endl;
  else
      std::cout << "False" << std::endl;
}
```

```
$ g++ -std=c++14 fp_tol.cpp
```

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 **difference** is below some *tolerance* that is near epsilon.
- Testing for equality with integers is ok, however, because integers are exact.

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

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

  g = f*f;

  std::cout << "g = f*f" << g << std::endl;

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

  float TOL=1e-15;
    if (cmath::abs(f*f - g) < TOL)
     std::cout << "True" << std::endl;
  else
      std::cout << "False" << std::endl;
}
```

```
$ g++ -std=c++14 fp_tol.cpp
$ ./a.out
```

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 **difference** is below some *tolerance* that is near epsilon.
- Testing for equality with integers is ok, however, because integers are exact.

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

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

  g = f*f;

  std::cout << "g = f*f" << g << std::endl;

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

  float TOL=1e-15;
    if (cmath::abs(f*f - g) < TOL)
     std::cout << "True" << std::endl;
  else
      std::cout << "False" << std::endl;
}
```

```
$ g++ -std=c++14 fp_tol.cpp
$ ./a.out
g = f*f =0.01
False
True
```

Roundoff errors

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

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

```cpp
#include <iostream>          // RoundOff.cpp
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;
}
```

Roundoff errors

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

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

```cpp
#include <iostream>          // RoundOff.cpp
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++ RoundOff.cpp -o RoundOff
$./RoundOff
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

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.

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

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

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

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

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

Machine epsilon

Machine epsilon gives you the limits of the *precision* of the machine.

- *Machine epsilon* 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 << std::endl;
  std::cout << "g␣=" << g << std::endl;

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

```
$ g++ -std=c++14 fp_machEpsilon.cpp
```

Machine epsilon

Machine epsilon gives you the limits of the *precision* of the machine.

- *Machine epsilon* 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 << std::endl;
  std::cout << "g␣=" << g << std::endl;

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

```
$ g++ -std=c++14 fp_machEpsilon.cpp
$ ./a.out
```

Machine epsilon

Machine epsilon gives you the limits of the *precision* of the machine.

- *Machine epsilon* 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 << std::endl;
  std::cout << "g = " << g << std::endl;

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

```
$ g++ -std=c++14 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
```

Determinig the Machine Epsilon

One way to approximate machine epsilon is by repeatedly halving a number and testing it.

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

double halve(double f)
{
   if ((1.0+(f/2.)) > 1.0)
      halve(f/2);
   else
       return(f/2.);
}

int main()
{
  double eps = 1.0;
  double halveEps;

  halveEps = halve(eps);
  std::cout << "halving...␣" << halveEps << std::endl;

  std::cout << "double␣eps:␣"
            << std::numeric_limits<double>::epsilon() << '\n';
}
```

```
$ g++ -std=c++14 halvingEps.cpp
```

Determinig the Machine Epsilon

One way to approximate machine epsilon is by repeatedly halving a number and testing it.

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

double halve(double f)
{
   if ((1.0+(f/2.)) > 1.0)
      halve(f/2);
   else
       return(f/2.);
}

int main()
{
  double eps = 1.0;
  double halveEps;

  halveEps = halve(eps);
  std::cout << "halving...␣" << halveEps << std::endl;

  std::cout << "double␣eps:␣"
            << std::numeric_limits<double>::epsilon() << '\n';
}
```

```
$ g++ -std=c++14 halvingEps.cpp
$ ./a.out
```

Determinig the Machine Epsilon

One way to approximate machine epsilon is by repeatedly halving a number and testing it.

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

double halve(double f)
{
   if ((1.0+(f/2.)) > 1.0)
      halve(f/2);
   else
      return(f/2.);
}

int main()
{
  double eps = 1.0;
  double halveEps;

  halveEps = halve(eps);
  std::cout << "halving...␣" << halveEps << std::endl;

  std::cout << "double␣eps:␣"
            << std::numeric_limits<double>::epsilon() << '\n';
}
```

```
$ g++ -std=c++14 halvingEps.cpp
$ ./a.out
halving...  1.11022e-16
double eps:  2.22045e-16
```

BTW the technique used in the `halve()` function, is called *recursivity*.

Or we can use the *Numeric Limits Interface:*

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

int main()
{
    std::cout << "int␣eps:␣"
              << std::numeric_limits<int
                 >::epsilon() << '\n';
    std::cout << "float␣eps:␣"
              << std::numeric_limits<
                   float>::epsilon() <<
                   '\n';
    std::cout << "double␣eps:␣"
              << std::numeric_limits<
                   double>::epsilon() <<
                    '\n';
}
```

```
$ g++ -std=c++14 detEpsilon.cpp
```

Determinig the Machine Epsilon

One way to approximate machine epsilon is by repeatedly halving a number and testing it.

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

double halve(double f)
{
   if ((1.0+(f/2.)) > 1.0)
      halve(f/2);
   else
      return(f/2.);
}

int main()
{
  double eps = 1.0;
  double halveEps;

  halveEps = halve(eps);
  std::cout << "halving...␣" << halveEps << std::endl;

  std::cout << "double␣eps:␣"
            << std::numeric_limits<double>::epsilon() << '\n';
}
```

```
$ g++ -std=c++14 halvingEps.cpp
$ ./a.out
halving...  1.11022e-16
double eps:  2.22045e-16
```

BTW the technique used in the `halve()` function, is called *recursivity*.

Or we can use the *Numeric Limits Interface:*

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

int main()
{
    std::cout << "int␣eps:␣"
              << std::numeric_limits<int
                 >::epsilon() << '\n';
    std::cout << "float␣eps:␣"
              << std::numeric_limits<
                   float>::epsilon() <<
                   '\n';
    std::cout << "double␣eps:␣"
              << std::numeric_limits<
                   double>::epsilon() <<
                    '\n';
}
```

```
$ g++ -std=c++14 detEpsilon.cpp
$ ./a.out
```

SciNet Physics UNIVERSITY OF TORONTO

Determinig the Machine Epsilon

One way to approximate machine epsilon is by repeatedly halving a number and testing it.

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

double halve(double f)
{
   if ((1.0+(f/2.)) > 1.0)
      halve(f/2);
   else
      return(f/2.);
}

int main()
{
  double eps = 1.0;
  double halveEps;

  halveEps = halve(eps);
  std::cout << "halving...␣" << halveEps << std::endl;

  std::cout << "double␣eps:␣"
            << std::numeric_limits<double>::epsilon() << '\n';
}
```

```
$ g++ -std=c++14 halvingEps.cpp
$ ./a.out
halving...  1.11022e-16
double eps:  2.22045e-16
```

BTW the technique used in the `halve()` function, is called *recursivity*.

Or we can use the *Numeric Limits Interface:*

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

int main()
{
    std::cout << "int␣eps:␣"
              << std::numeric_limits<int
                 >::epsilon() << '\n';
    std::cout << "float␣eps:␣"
              << std::numeric_limits<
                   float>::epsilon() <<
                   '\n';
    std::cout << "double␣eps:␣"
              << std::numeric_limits<
                   double>::epsilon() <<
                    '\n';
}
```

```
$ g++ -std=c++14 detEpsilon.cpp
$ ./a.out
int eps:  0
float eps:  1.19209e-07
double eps:  2.22045e-16
```

SciNet Physics UNIVERSITY OF TORONTO

The limits of precision: look out below!

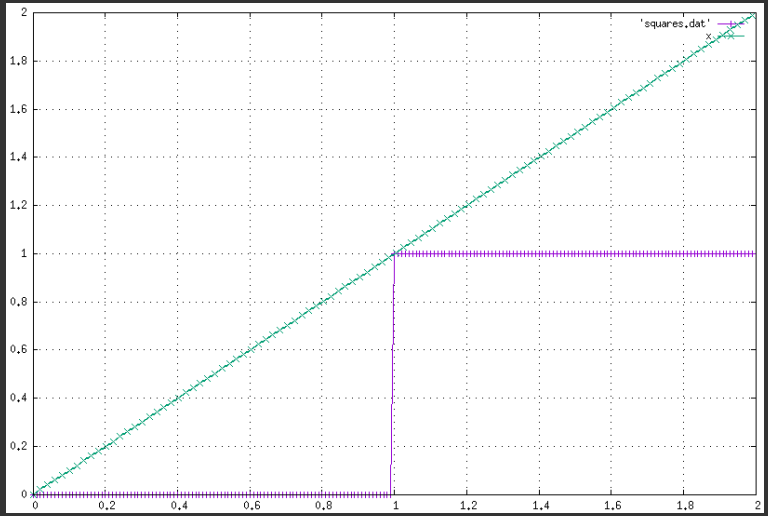
Problems will occur when the result of a calculation spans more orders of magnitude than the inverse of machine epsilon.

Try the following:

- For the range of numbers between 0 and 2, repeatedly take square roots of the numbers, then repeatedly square the numbers.
- Plot the result, from 0..2.
- What should you get? What do you get?
- Loss of precision in early stages of a calculation causes problems.

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

int main()
{
  int M = 200;
  int K = 100;
  int rep = 200;
  double *x = new double[M];

  std::ofstream dataFile("squares.dat");

  for (int j=0; j<M; j++)
      x[j] = j/K;

  for (int j=0; j<M; j++)
      for (int i=0; i<rep; i++)
          x[j] = sqrt(x[j]);

  for (int j=0; j<M; j++)
      {
       for (int i=0; i<rep; i++)
           x[j] = x[j]*x[j];
       dataFile << j/K << "\t" << x[j] << "\n";
      }

  dataFile.close();

  delete [] x;
  return 0;
}
```

Precision issues...



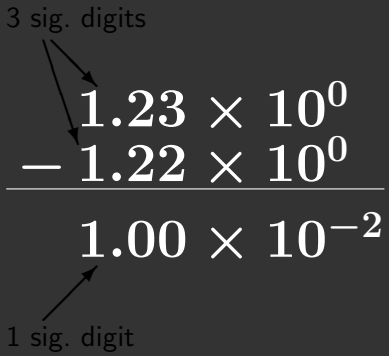
If the argument is below 1.0, `sqrt` pushes it up to epsilon below 1.0.
If the argument is above 1.0, `sqrt` pulls it down to exactly 1.0.

SciNet
Physics
UNIVERSITY OF TORONTO

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.

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, ...
- eg. 8-bit integers have a range of -128 to 127.
- eg. 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.

```
#include <iostream>

int main()
{
  float f = 1.0e15;

  std::cout << "f␣=" << f << std::endl;
  std::cout << "f*f␣=" << f*f << std::endl;
  std::cout << "f*f*f␣=" << f*f*f << std::endl;

}
```

```
$ g++ -std=c++14 fp_machEpsilon.cpp
```

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, ...
- eg. 8-bit integers have a range of -128 to 127.
- eg. 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 << std::endl;
  std::cout << "f*f␣=" << f*f << std::endl;
  std::cout << "f*f*f␣=" << f*f*f << std::endl;

}
```

```
$ g++ -std=c++14 fp_machEpsilon.cpp
$ ./a.out
```

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, ...
- eg. 8-bit integers have a range of -128 to 127.
- eg. 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 << std::endl;
  std::cout << "f*f␣=" << f*f << std::endl;
  std::cout << "f*f*f␣=" << f*f*f << std::endl;

}
```

```
$ g++ -std=c++14 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.

```
#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 << std::endl;
  std::cout << "g␣=" << g << std::endl;
  std::cout << "h␣=" << h << std::endl;
  std::cout << "k␣=" << k << std::endl;
  std::cout << "l␣=" << l << std::endl;
}
```

```
$ g++ -std=c++14 fp_undFlow.cpp
```

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.

```
#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 << std::endl;
  std::cout << "g␣=" << g << std::endl;
  std::cout << "h␣=" << h << std::endl;
  std::cout << "k␣=" << k << std::endl;
  std::cout << "l␣=" << l << std::endl;
}
```

```
$ g++ -std=c++14 fp_undFlow.cpp
$ ./a.out
```

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.

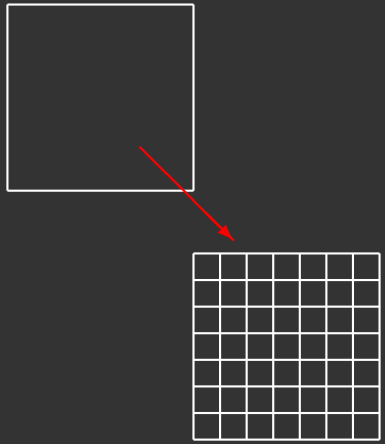
```
#include <iostream>

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 << std::endl;
  std::cout << "g␣=" << g << std::endl;
  std::cout << "h␣=" << h << std::endl;
  std::cout << "k␣=" << k << std::endl;
  std::cout << "l␣=" << l << std::endl;
}
```

```
$ g++ -std=c++14 fp_undFlow.cpp
$ ./a.out

f =-1e+35
g =-inf
h =inf
k =9.80909e-45
l =0
```

Discretization 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.



SciNet Physics UNIVERSITY OF TORONTO

Discretization error, 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). Eg. $CFL \equiv \frac{u\Delta t}{\Delta x} < C_{max}$

Know your accuracy!

All algorithms have an accuracy associated with the discretization error. Be sure that you know the accuracy of your algorithm!

$$f'(x_j) = \frac{-f(x_{j+2}) + 8f(x_{j+1}) - 8f(x_{j-1}) + f(x_{j-2})}{12\Delta x} + \mathcal{O}(\Delta x^4)$$

$$f''(x_j) = \frac{\partial^2 f(x_j)}{\partial x^2} = \frac{f(x_{j+1}) - 2f(x_j) + f(x_{j-1})}{\Delta x^2} + \mathcal{O}(\Delta x^2)$$

Truncation error

Truncation error occurs when an expansion in your calculation is truncated. Meaning, instead of using this:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \ldots$$

we use this:

$$e^x \simeq 1 + x + \frac{x^2}{2}.$$

Obviously truncation is necessary. How do we determine where to truncate? How many terms should we keep?

Where to truncate?

Choosing where to truncate is sometimes more art than science. The question you need to answer: is what I am throwing away important to the calculation?

Sometimes the answer is obvious. In the case of e^x, we can sensibly truncate when we reach machine precision, meaning choose n such that

$$\left|\frac{x^n}{n!}\right| < \epsilon$$

where ϵ is machine precision.

Here's how I approach the problem: determine some metric for what you are expanding which captures its importance (size, magnitude, energy, ...) and then compare the largest term you are throwing away to the largest non-trivial term. I like to have at least one order of magnitude size difference between them, preferably two orders.

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 `(abs(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.