Introduction to High Performance Computing

Alejandro Cárdenas-Avendaño

Numerics and 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.

1037

$$1037 = (1\times1000)+(0\times100)+(3\times10)+(7\times1)$$

$$1037 = (1\times10^3)+(0\times10^2)+(3\times10^1)+(7\times10^0)$$

- Or in base 2

$$1037 = 10000001101$$

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.

1037 = 10000001101

- 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’.

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: $0...2^{32} - 1$.
- All operations (+, -, *) between representable integers are represented unless there is overflow.

Sign Number

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

Sign Number

A typical 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;
```

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++ intexampleOF1.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++ intexampleOF2.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.

$$-1.37 \times 10^{-12}$$

A typical single precision real = 32 bits = 4 bytes.

Sign **Exponent** **Mantissa**

1 (bit) **8 (bits)** **23 (bits)**

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
```

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 – example

```
#include <climits>
#include <iostream>

int main () {

  std::cout << "type\tlowest\thighest\n";
  std::cout << "int\t"
        << std::numeric_limits <float >::lowest() << "\t"
        << std::numeric_limits<int>::max()<< "\n";
  std::cout <<"float\t"
        << std::numeric_limits<int>::lowest()<< "\t"
            << std::numeric_limits <float >::max() << "\n";
  std::cout << "double\t"
        << std::numeric_limits <double >::lowest()<< "\t"
            << std::numeric_limits <double >::max() << "\n";
  std::cout << "long double\t"
        << std::numeric_limits <long double >::lowest()<< "\t"
            << std::numeric_limits <long double >::max() << "\n";
}
```

Testing for equality or inequalities

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

int main () {

  double f = 0.1;
  double g;

  g = f*f;

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

  if (f*f == 0.01)
    std::cout << "True" << std::endl;
  else
    std::cout << "False" << std::endl;

  double TOL=1e-15;

  if (abs(f*f - g) < TOL)
    std::cout << "True" << std::endl;
  else
    std::cout << "False" << std::endl;
}
```

Testing for equality or inequalities

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.

```
$ g++ equality.cpp
$ ./a.out
g = f*f 0.01
False
True
```

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

int main () {

  double f = 0.1;
  double g;

  g = f*f;

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

  if (f*f == 0.01)
    std::cout << "True" << std::endl;
  else
    std::cout << "False" << std::endl;

  double TOL=1e-15;

  if (abs(f*f - g) < TOL)
    std::cout << "True" << std::endl;
  else
    std::cout << "False" << std::endl;
}
```

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 != a+(b+c)

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

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

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 != 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;

}
```

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 != 1.
- (or sometimes, the largest x such that 1+x = 1.)
- Machine epsilon is named after the mathematical term for a small positive infinitesimal.

```cpp
#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++ machEpsilon.cpp
$ ./a.out
f =1
g =1e-18
(1. - 1.)+ 1.e-18 = 1e-18
(1. + 1.e-18) - 1.0 = 0
(1. + 1.e-18) = 1
```

Determining the Machine Epsilon

```
#include <climits>
#include <iostream>

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

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

  halveEps = halve(eps);
  std::cout << "halving..." << halveEps << std::endl;
  std::cout << "double eps: "
            << std::numeric_limits <long double>::epsilon() << '\n';

}
```

```
$ g++ compmachineeps.cpp
$ ./a.out
halving...2.26975e-314
double eps: 1.0842e-19
```

Know your accuracy!

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

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

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

Libraries

Libraries

Libraries

- In some case, several object files for different modules that need to be linked together.
- In the example below, **thisapp.c** contains the main function and **helper.c/helper.h** are a module.

```
# makefile for 'thisapp'
CXX=g++
CXXFLAGS=-g -O2 -std=c++11

all: thisapp

thisapp.o: thisapp.cc helper.h
   ${CXX} ${CXXFLAGS} -c -o thisapp.o
thisapp.cc

helper.o: helper.cc helper.h
   ${CXX} ${CXXFLAGS} -c -o helper.o
helper.cc

thisapp: thisapp.o helper.o
${CXX} -g -o thisapp thisapp.o helper.o
```

- What if we could use **helper** in another project called **newapp**, without recompiling helper.c?
- Copy **.o** and **.h** to separate directories:

  helper.h -> /base/include/helper**.h**

  helper.o -> /base/lib/helper**.o**
- Must let compiler know where they are:

  Add **-I** flag for include directories.

  Absolute path for object file (only for now!).

```
# makefile for 'newapp'
CXX=g++
CXXFLAGS=-I/base/include -g -O2 -std=c++11

all: newapp

newapp.o: newapp.cc /base/include/helper.h
  ${CXX} ${CXXFLAGS} -c -o newapp.o newapp.cc

newapp: newapp.o /base/lib/helper.o
  ${CXX} -g -o newapp newapp.o /base/lib/
helper.o
```

Libraries

What we just did is a poor man's library building.

Real libraries are similar; they have

- to be installed (and perhaps built first)
- header files (.h or .hpp) in some folder
- library files (object code) in a related folder.

Linux: library filenames start with lib and end in **.a** or **.so.**

Instead of giving the explicit path in makefile rule, we specify:

- the path to the **library's object** using the **-L** option, stored in variable **LDFLAGS**;
- the object code using **-lNAME** (with a lower case letter l) stored in variable **LDLIBS**.
- libraries should come after the object files that use them.

```
# makefile for 'newapp'
CXX=g++
CXXFLAGS=-I/base/include -g -O2 -std=c++11

all: newapp

newapp.o: newapp.cc
  ${CXX} ${CXXFLAGS} -c -o newapp.o newapp.cc

newapp: newapp.o
  ${CXX} -g -o newapp newapp.o /base/lib/
libhelper.a
```

```
# makefile for 'newapp'
CXX=g++
CXXFLAGS=-I/base/include -g -O2 -std=c++11
LDFLAGS=-g -L/base/lib
LDLIBS=-lhelper

all: newapp

newapp.o: newapp.cc
  ${CXX} ${CXXFLAGS} -c -o newapp.o newapp.cc

newapp: newapp.o
  ${CXX} ${LDFLAGS}-o newapp newapp.o ${LDLIBS}
```

Libraries: with a clean rule

Adding a clean rule and extracting the common path, the Makefile for newapp will look like this:

```
# makefile for 'newapp'
CXX=g++
HELPERBASE?=/base/
HELPERINC=${HELPERBASE}include
HELPERLIB=${HELPERBASE}lib
CXXFLAGS=-I${HELPERINC} -g -O2 -std=c++11
LDFLAGS=-g -L${HELPERLIB}
LDLIBS=-lhelper

all: newapp

newapp.o: newapp.cc
  ${CXX} ${CXXFLAGS} -c -o newapp.o newapp.cc

newapp: newapp.o
  ${CXX} ${LDFLAGS}-o newapp newapp.o ${LDLIBS}

clean:
   \rm -f newapp.o
```

Note:

- C++ standard libaries (*vector,cmath,...*) do not need any **-l...**'s.
- There are standard directories for libraries that needn't be specified in **-I** or **-L** options (/usr/include,...)
- Libraries installed through a package manager end up in standard paths; they just need **-l...** options in *LDLIBS*.
- If you compile your own libraries in non-standard locations, you do need **-I** and **-L** options (as well as the **-INAME** clause)

Installing libraries from source

What to do when your package manager does not have that library, or you do not have permission to install packages in the standard paths?

Or, what if you are on a Cluster/WorkStation (where you do not have permissions to install using the package manager) and there isn't a module for that library already?

Compile from source code with a "base" or "prefix" directory.

Common installation procedure (but **read** documentation!):

```
$ ./configure --prefix=<BASE>
$ make
$ make install
```

You choose the <BASE>, but it should be a directory that you have write permission to, e.g., a subdirectory of your $HOME.

If the documentation says to do **sudo make install**, but you do not have administrative right to the machine (like on a cluster), it's **wrong**; dig deeper into the documentation to see where you can provide a prefix or base directory.

Using libraries that are not in standard directories

For libraries that are not installed in standard directories, you need **-I<BASE>/include** and **-L<BASE>/lib** options in your Makefile.

Alternatively, you can omit these for g++ under linux by setting some **linux environment variables**:

```
export CPATH="$CPATH:<BASE>/include"                  # compiler looks here for include files
export LIBRARY_PATH="$LIBRARY_PATH:<BASE>/lib"        # and here for library files
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:<BASE>/lib" # runtime linker looks here
```

You either enter these commands on the linux prompt before calling make, or, to set these automatically when you log in, you can add these lines to the **.bashrc** file in your home directory. (*Note: putting these in Makefiles does not work!*)

- The last one (LD_LIBRARY_PATH) may be necessary to run the application, even when it was successfully built and linked already.
- If the library installs binary applications (i.e. commands) as well, you'll also need to set

```
export PATH="$PATH:<BASE>/bin" # linux shell looks for executables here
```

GNU Scientific Library

GNU Scientific Library (GSL)

Is a C library containing many useful scientific routines, such as:

- Root finding
- Minimization
- Sorting
- Integration, differentiation, interpolation, approximation
- Statistics, histograms, fitting
- Monte Carlo integration, simulated annealing
- ODEs
- Polynomials, permutations
- Special functions
- Vectors, matrices

Note: C library means we'll likely need to deal with some pointers and casts

GSL root finding example

Suppose we want to find where $f(x) = a \sin(\cos(v - wx)) + bx - cx^2$ is zero.

```
// gslroot.cpp
#include <iostream>
#include <gsl/gsl_roots.h>

struct Params {
 double a,v,w,b,c;
};

double examplefunction(double x, void* param){
 Params* p = (Params*)param;
 return p->a*sin(cos(p->v-p->w*x))+p->b*x-p->c*x*x;
}

int main() {

 double x_lo = -4.0;
 double x_hi = 5.0;
 Params args = {0.3, 2/3.0, 2.0, 1/1.3, 1/30.0};
 gsl_root_fsolver* solver;
 gsl_function      fwrapper;
 solver = gsl_root_fsolver_alloc(
             gsl_root_fsolver_brent);
 fwrapper.function = examplefunction;
 fwrapper.params = &args;
```

```
gsl_root_fsolver_set(solver,&fwrapper,x_lo,x_hi);
 std::cout << "Iter lower upper root Error\n";

 int status = 1;

 for (int iter=0; status and iter < 100; ++iter)
   {

   gsl_root_fsolver_iterate(solver);

   double x_rt = gsl_root_fsolver_root(solver);
   double x_lo = gsl_root_fsolver_x_lower(solver);
   double x_hi = gsl_root_fsolver_x_upper(solver);
   std::cout << iter <<" "<< x_lo <<" "<< x_hi
             <<" "<< x_rt <<" "<<x_hi-x_lo<<"\n";
   status=gsl_root_test_interval(x_lo,x_hi,0,1e-6);
   }

 gsl_root_fsolver_free(solver);

 return status;

}
```

Compilation and linkage

- Lots of gsl... stuff.

  All of the algorithms come from the GSL.
- The rest is just wrappers, setting up parameters and calling the appropriate functions.
- There are pointers and typecasts, because we're dealing with a C library.
- How to compile on the command line?

MAC:

```
$ brew install gcc
$ brew install gsl
```

```
$ g++ gslroot.cpp -lgsl -o root
$ ./root
```

Unix:

```
$ sudo apt-get update
$ sudo apt-get install g++
$ sudo apt-get make
$ sudo apt-get install libgsl-dev
$ g++ gslroot.cpp -lgsl -o root
```

```
$ wget ftp://ftp.gnu.org/gnu/gsl/gsl-2.5.tar.gz
$ tar -zxvf gsl-2.5.tar.gz
$ cd gsl-2.5
$ mkdir /home/yourname/gsl
$ ./configure --prefix=/home/yourname/gsl
$ make
$ make check
$ make install

$ gcc -Wall -I/home/yourname/gsl/include -c
gslroot.cpp
$ gcc -L/home/yourname/gsl/lib gslroot.o -lgsl -
lgslcblas -lm
```

Result

```
Iter lower upper root Error
0 -4 1.21802 1.21802 5.21802
1 -4 -0.114081 -0.114081 3.88592
2 -0.258832 -0.114081 -0.258832 0.144751
3 -0.258832 -0.18507 -0.18507 0.0737623
4 -0.187249 -0.18507 -0.187249 0.00217951
5 -0.187249 -0.187243 -0.187243 6.44945e-06
6 -0.187249 -0.187243 -0.187243 6.4436e-06
7 -0.187249 -0.187243 -0.187243 6.4436e-06
8 -0.187243 -0.187243 -0.187243 2.77556e-17
```

Convergence and Stability

	method convergence	stability
Bisection	$\epsilon_{n+1} = \frac{1}{2}\epsilon_n$	Stable
Secant	$\epsilon_{n+1} = c\epsilon_n^{1.6}$	No bracket guarrantee
False position	$\epsilon_{n+1} = \frac{1}{2}\epsilon_n - c\epsilon_n^{1.6}$	Stable
Ridders'	$\epsilon_{n+2} = c\epsilon_n^2$	Stable
Brent	$\epsilon_{n+1} = \frac{1}{2}\epsilon_n - c\epsilon_n^2$	Stable
Newton-Raphson	$\epsilon_{n+1} = c\epsilon_n^2$	Can be unstable

Don't reinvent the wheel

- There are many possible algorithms to implement for root finding.
- But they are all pretty standard.
- Surely, someone must have done this already? **Correct!**
- The GNU Scientific Library is one such library.
- Don't implement this yourself if there is a library that does it for you.

Summary

- 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.

References

- SciNet Education (https://support.scinet.utoronto.ca/education/help/help_about.php)
- International HPC Summer School 2018 Lectures
- ARCHER UK National Supercomputing Service Training Courses (http://www.archer.ac.uk/training/)
- Muna, D & Price-Whelan, A. SciCoder Workshop. scicoder.org
- Peter Norvig, “What to demand from a Scientific Computing Language”, Mathematical Sciences Research Institute, 2010.