Computational Physics

Part II Physics

David Buscher <db106@cam.ac.uk>

January 2017

Introduction

Goals of this course

- Introduce you to scientific programming in `Python`
- Develop your (mathematical) programming skills in `Python`/`C++`
- Develop a deeper understanding of some specific physics examples (diffraction, dynamics, magnetism, ...) along the way
- Prepare you for the optional Part II Computing Project
- Prepare you for taking on a Part III Physics Project

Computing is a key research skill.

Goals of this course

Computing is also a key transferable skill. Lots of people can program, but physicists' analytical and mathematical skills makes them uniquely well placed to do sophisticated software development — mathematical or otherwise.

1. Engineering, CAD, ...
2. Graphics: rendering etc
3. Gaming: 'physics engines'
4. Networks: data compression etc
5. Financial modelling: noisy data plus dodgy models (?)
6. etc etc

Course Structure

1. Lectures, weeks 1–4
   - Mixture of background to some numerical techniques and practical examples of programming in `C++` and `Python`.
   - This is not a formal course in numerical methods.
2. Compulsory Practical Exercises
   - Practical classes in the PWF/MCS room
   - Counts for 0.2 units of further work in total
   - Runs in Weeks 4–7
3. Optional Computing Project
   - Counts for 1.0 units of further work

Handouts/Materials

This is a practical course: learn by doing.

Not assessed by written examination.

Key Written Material

- Slides used in lectures (available on TiS)
- Lab manual for the exercises, including a guide to some useful `C++` and `Python` topics
- Project descriptions

In addition, the web now contains many high-quality tutorials, FAQs, etc.

The course web site (`http://www.mrao.cam.ac.uk/~dfb/teaching/computationalphysics`) will provide links to some of these.

Compulsory Practical Work

- Runs in Weeks 4–7: Starting Friday 10th Feb
- MCS sessions run 1400-1730 on **Fridays, Mondays, Wedesdays**
- Please *let me know if it is not possible to attend one of these*
- You can work from elsewhere if you wish
- Part II students have priority access at these times
- Demonstrators available during these times. Ask!
- You will solve three problems
- Hand in your solutions (source code, plots, text file) via your course pigeonhole (explained in the class manual).
- Deadline for all exercises: end of Lent Term (Friday 17 March) at 16:00.
- Manual available in week 2/3

Assessment

The spirit of self-assessment of the IB lab continues. Advice on working together and plagiarism from the IB manual also applies (working in pairs is OK subject to the usual rules).

You will need to submit:

- some working `C++` or `Python` code
- a `readme.txt` or `readme.pdf` file a few lines long which says what you did (or mentions any problems) and may contain an answer to a question in the manual.
- Relevant plots as `PDF` files — these are easy to generate from `gnuplot` or `pyplot`. Alternatively, plots can be included in the `readme.pdf` file.

Note: Any files larger than 1 MB will be assumed to be data files and will be ignored/discarded.

A mark out of 6 will be awarded for the submitted work to exercises 1 and 2, and out of 8 for exercise 3. Please, Read The Friendly Manual.

Optional Project

Details

Do in your own time — start this term, continue through vacation.

Deadline: Monday 1st May at 16:00

Physics Linux MCS facilities available (remote access via `ssh`: see course website for more details)

Your project must compile and run on the Physics Linux MCS, and core numerics must be in the `C++` language or in `Python`.

We cannot support your own computer system (but your College may help?)

Choice of projects. Further advice will be given in a later lecture.

Your own independent work

Manual available in week 4

Topics

Both Computational Physics and `C++`/`Python` would by themselves be large topics for a lecture and practical course.

We must be very selective!

The idea is not a comprehensive course, but exposure to a sufficient variety of algorithms and computing techniques that you become confident enough to work on your own.

Topics

Numerical Concepts

Ordinary Differential Equations (ODEs)

Monte-Carlo Techniques

Discrete Fourier transforms in 1 and 2 dimensions

Dealing with data: fitting models

Linear Algebra (e.g. eigenmodes)

Understanding the accuracy of our solutions is a key concept. How accurate is a solution? What limits this? Trade off with compute time?

Topics

Computing Concepts and Skills

Revising your `C++` knowledge and introducing `Python`

Arrays in one or more dimensions

Interfacing to external code libraries

Writing good quality code

Approaches to Scientific Computing

Computers have revolutionised physics (e.g. formation of single star 1969; crystallography; Schrödinger equation; fluid flow...)

In general we care a lot about performance and accuracy, and not very much about user interfaces (GUIs etc).

In the beginning there was `FORTRAN`...Now we can talk about `C FORTRAN C++ C# Haskell Python Java Scala IDL Matlab yourFavouriteLanguageHere` and many more

Simplifying grossly, we can choose between

1. "high-level" interpreted languages with extensive toolboxes built-in, often with no strong typing: e.g. `MATLAB,IDL,Python`
2. "low level" compiled and strongly **typed** languages: `C, C++, FORTRAN`

In practice we need to know about both.

Why learn C++?

This is not the easiest language to learn for beginners to computing, but it is worth the effort:

- Good mix of high-level and low-level features:
  - Provides powerful high-level abstractions e.g. objects
  - Allows low-level control of machine resources for memory- and CPU-intensive applications.
- An Industry Standard
- Well-developed standards and an extensive set of standard libraries
- Lots of free and commercial libraries available that extend its use
- Not designed *per se* for numerical work, but highly capable and fast

Why learn Python?

An up-and-coming language that is fast becoming a standard for scientific computing:

- Rich, high-level language designed to be **easy to learn**
- An Industry Standard - powers much of Google, Facebook, etc.
- Very extensive set of standard libraries
- Lots of free and commercial libraries available that extend its use
- The `numpy` and `scipy` libraries in particular provide a standard framework for scientific computing

A knowledge of *either* `C++` *or* `Python` programming is required for this and later courses. A knowledge of *both* will put you in an enviable position in the job/PhD markets post-graduation.

Course Pre-requisites

Knowledge of IB physics — dynamics, diffraction, magnetism, statistical physics...

Basic familiarity with `unix`: you should be able to login to the Linux MCS and edit a text file.

Basic programming skills — you should be able to:

- write a simple program to solve a simple physical problem
- compile/link/run a `C++` program or run a `Python` program on the MCS.
- debug your program when errors occur (which they will), using simple (`cout<<`) or `print()` statements, using a debugger e.g. `gdb` or `pdb`, and/or by asking a demonstrator or a colleague

If you can’t do these things, you need to refresh the IB material. The first exercise is intended to be straightforward and remind you of the basics of programming.

Books

There are lots of good programming courses on the web, but fewer comprehensive numerical methods websites. Useful books on numerical methods include:

1. "Numerical Recipes", 3rd edition. (also in C, C++, FORTRAN, Pascal, etc), by Press, Teukolsky, Vetterling & Flannery (CUP). Excellent encyclopedic summary of theory of many numerical methods and techniques. Almost a bible of methods for researchers – first place to look. But accompanying source code (which is not free) of patchy quality
2. 'Computational Physics', Giordano & Nakanishi. Nice introduction at the right level to several commonly-used techniques

The best texts separate implementation details (ie how you use the language) from the theoretical details of the methods.

A taste of Python3

Python looks like C++ without the curly braces

```
x=5
print("The square of",x,"is",x**2)
```

```
The square of 5 is 25
```

Blocks of code are demarcated by their **indentation**.

```
from numbers import Number
for x in [5, 0.5, "A circle"]:
    if isinstance(x,Number):
        print("The square of",x,"is",x**2)
    else:
        print(x,"cannot be squared")
```

```
The square of 5 is 25
The square of 0.5 is 0.25
A circle cannot be squared
```

Numpy adds numerical capabilities

```
import numpy
print(numpy.sqrt(2))
```

```
0.707106781187
```

```
x=numpy.array([1,2,3])
print("The cube of",x,"is",x**3)
```

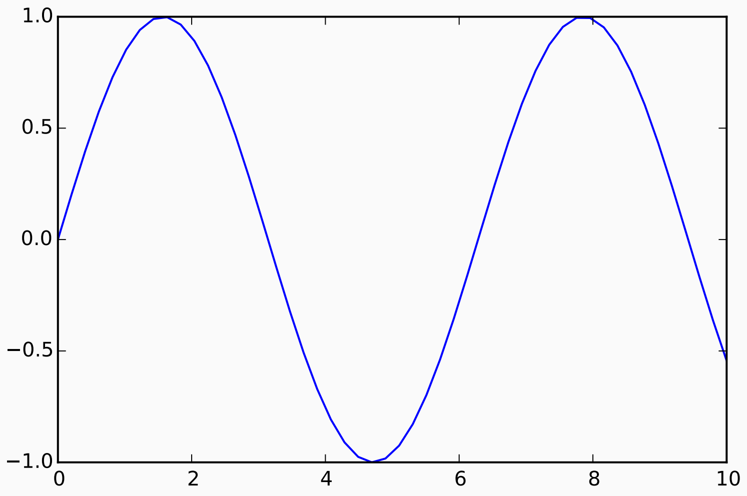
```
The cube of [1 2 3] is [ 1 8 27]
```

```
print("The sum of 1..100 is:",
      numpy.sum(numpy.arange(1,101)))
```

```
The sum of 1..100 is: 5050
```

Pyplot allows the result of computations to be visualised

```python
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10)
plt.plot(x, np.sin(x))
```



Computer Representation of Numbers

Computer Numbers

Everything stored as groups of binary **bits**, with (almost always) 8 contiguous bits making up a **byte**. Collections of bits can be used to represent **signed** and **unsigned** integers *exactly* — as long as we have enough bits. With 32 bits per integer we have $2^{32} \approx 4.3 \times 10^9$ possibilities.

Floating point numbers cannot be represented exactly in this way. The **precision** of the representation is limited.

Precision

Number of significant digits in the number's representation

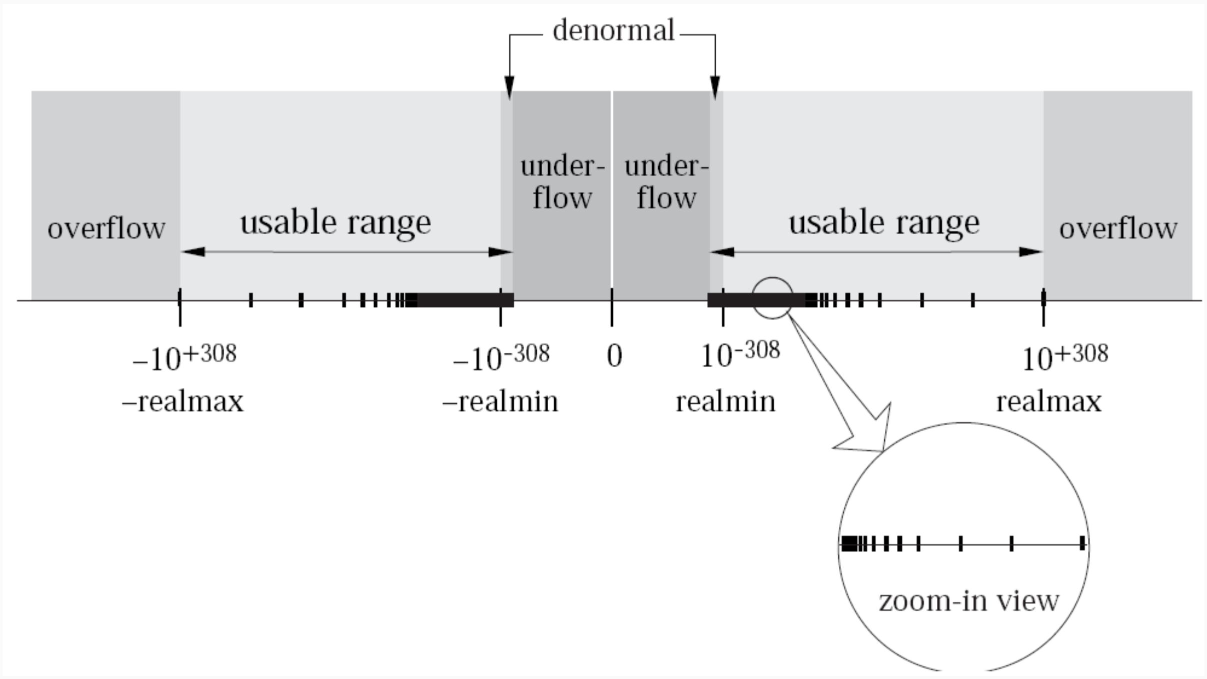
Accuracy

How close to the true value is the number?

A number may be precise but inaccurate.

The Floating Point Number Line

Floating point numbers lie at *discrete points* on the real number line:



Roundoff error

We choose the nearest floating point number to represent a given real number. The spacing between floating-point numbers increases as we move to higher values, although the fractional change between adjacent numbers does not.

Roundoff Error

The difference between the true value and the nearest representable value

It's just luck whether a number has an exact binary representation. For example, 0.15625 does, but 0.01 does not.

Example C++ code

```cpp
#include <cmath>
#include <iostream>
using namespace std;
int main()
{
  cout.precision(18);
  cout << (float) 0.15625 << " " << (float) 0.01 << endl;
}
```

```
0.15625 0.00999999977648258209
```

Accumulating Roundoff Errors

```
float average(float x, int count){
  float sum=0.0;
  for (int i=0; i<count; i++){ sum += x; }
  return(sum/count);
}
int main(){
  cout.precision(18);
  cout << (float) 0.15625 << " " << average(0.15625, 100000)
       << endl;
  cout << (float) 0.15624 << " " << average(0.15624, 100000)
       << endl;
}
```

```
0.15625 0.15625
0.156240001320838928 0.156249791383743286
```

Floating point numbers are not real

Floating point numbers do not obey the usual rules of arithmetic: for example there is no guarantee of associativity or commutativity. We may find that

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

$$(ab)c \neq a(bc)$$

$$a \times (1/a) \neq 1$$

Although we can usually rely on:

$$ab = ba$$

$$a+b = b+a$$

Comparing Floating Point Numbers

You must **never** compare two floating point numbers for equality.

```
for i in range(1,60):
    y = (1.0 / i) * i
    if y != 1.0:
        print("Surprise:","( 1.0 /",i,") *",i,
              "- 1.0 =", y-1.0)
```

```
Surprise: ( 1.0 / 49 ) * 49 - 1.0 = -1.1102230246251565e-16
```

The best you can do is compare 2 numbers to within some absolute or relative amount. It may help to write a small function that looks at the *difference* between the two numbers and decides whether they are “the same” in the context of your problem.

Representing Numbers

Before 1985 (yes, we had computers then...), there were lots of different standards for floating point representations. Thankfully, IEEE 754-1985 defined a specific binary representation of floating point values which is now almost universally used.

It also defines $\pm\infty$, not a number (NaN), what should happen when things go wrong — e.g. divide by zero — and how to convert between types.

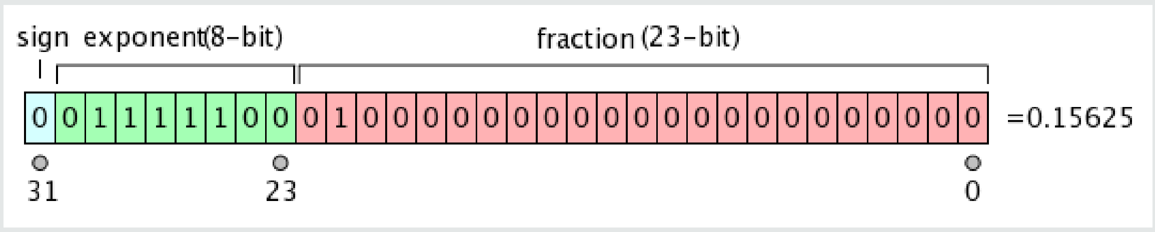
Compatibility of binary data files between machines is now straightforward. (In the dark ages, we used to have to scale the floats to integers, transfer integers, then convert back).

Use scientific notation in binary form: a number is represented as s, e and f where

$$(-1)^s \times 2^e \times 1.f$$

IEEE 754-2008 floating point

binary32 (“single precision”, `float` in C++, `float32` in Python)



Sign bit, 8 exponent bits, 23 fraction (or significand) bits

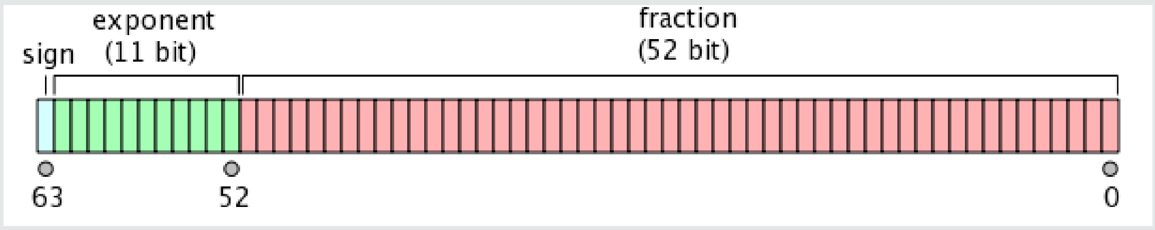
Here: $s = 0$, $e = (1111100)_2 = 124$ and $f = (1.01)_2 = 1.25$, so the number is

$$(-1)^0 \times 2^{124-127} \times 1.25 = 0.15625$$

(Note the use of a biased exponent, biased by $+127$.)

IEEE 754-2008 floating point

binary64 (“double precision”, `double` in C++, `float64` in Python)



Sign bit, 11 exponent bits, 52 fraction bits

128-bit long doubles, or quads, exist for super-high-precision work, but are not generally widely supported by libraries or compilers. If you are interested, you can find download a `C++` package that does arbitrary precision arithmetic at `http://crd.lbl.gov/~dhbailey/mpdist/`

Machine Precision

For every computer there exists a number ϵ such that

$$1 + \delta = 1 \quad \text{if} \quad \delta < \epsilon$$

when the values on the left and right hand side of the equation are rounded to machine-representable numbers. The value of ϵ is known as the **machine precision** and is dependent on the numeric representation and the rounding employed.

An alternative definition for ϵ is *The difference between 1 and the least value greater than 1 that is exactly representable in the given floating point type*. Note that this definition differs yields a value of ϵ which is **twice** that given in the equation above.

Determining the machine precision

In **C++** the value of ϵ can be found in a header file `<cfloat>`.

In **Python** the machine precision can be retrieved using the `numpy.finfo()` function:

```
import numpy
for dtype in [numpy.float32, numpy.float64]:
    epsilon=numpy.finfo(dtype).eps
    print("{:.12g} {:.18f} {:.18f}".format(epsilon,
                                           dtype(1+epsilon*0.51),
                                           dtype(1+epsilon*0.50)))
```

```
1.192092896e-07 1.000000119209289551 1.000000000000000000
2.220446049e-16 1.000000000000000222 1.000000000000000000
```

Note that $1.19 \times 10^{-7} = 2^{-23}$ and $2.22 \times 10^{-16} = 2^{-52}$.

Integers

Less problematic than floats but can still overflow...

- a `char` is usually one byte ($2^8 = 256$ values)
- a `short int` is usually 2 bytes ($2^{16} = 65536$ values)
- an `int` is usually 4 bytes ($2^{32} \sim 4 \times 10^9$ values)
- a `long int` is usually 8 bytes ($2^{64} \sim 2 \times 10^{19}$ values)

Unsigned integers

Integers are available in C++ in signed and unsigned forms. Unsigned types are suitable for things that cannot be negative e.g. allocated memory, or the number of items in an array.

You may come across the type `size_t`: it is commonly used to store unsigned quantities — it is just an alias (strictly a `typedef`) to a large unsigned integer type.

You might see compilers complaining about comparing signed and unsigned integers (normally benign but worth understanding)

```
int main()
{
  std::vector<int> a(10);
  // a.size() returns type size_t
  // Compiler warning likely.
  for ( int i = 0; i < a.size(); i++ )
    a[i] = i*i;
```

2 divided by 3 equals...

```cpp
#include <iostream>
int main()
{
  double x = 2/3;
  double y = 2./3;
  double z = 2%3;
  std::cout << x << " " << y << " " << z << "\n";
}
```

```
0 0.666667 2
```

Because the compiler treats the numbers as integers, and 2 divided by 3 is zero, with 2 left over. You probably wanted 2.0/3.0 (promotes to floating point before dividing).

The remainder ($a\%b$) is sometime useful too...

C++ pointers

A pointer is used to address a location in memory. It is an unsigned integer of some size.

Because $2^{32} \approx 4 \times 10^9$, we can only address 4 GByte of memory using 32-bit pointers. Most new machines are '64 bit' i.e. they use 64-bit integers for pointers: can address $2^{64} \sim 10^{19}$ locations.

Remember the basic `C++` idea: send pointers to big objects, or objects you want other code to change, don't send the object itself.

C++ pointers

It is really important you completely understand this snippet:

```
#include <iostream>
int main()
{
  int i      = 0;
  int* i_ptr = &i;
  *i_ptr     = 1;
  std::cout << "i = " << i << " " << i_ptr << "\n";
  i = 2;
  std::cout << "i = " << i << " " << i_ptr << "\n";
}
```

```
i = 1 0x7fff5c9d5124
i = 2 0x7fff5c9d5124
```

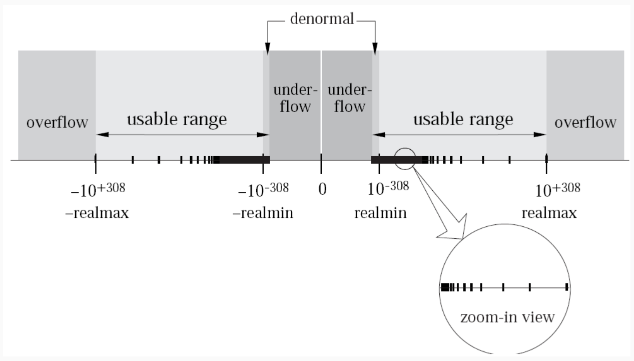
Pass by value and reference

It is really important you completely understand this too:

```
#include <iostream>
void f( int* a, int b, int& c )
{
  a  += 1;
  b  += 1;
  c  += 1;
}
int main()
{
  int a = 0, b = 0, c = 0;
  f( &a, b, c );
  std::cout << a << " " << b << " " << c << "\n";
}
```

```
0 0 1
```

Underflow and Overflow



When numbers get too small or too large, underflow and overflow can occur (less often a problem with 64-bit floats).

Avoid this by keeping your numbers within sensible ranges. Often this will involve re-scaling the physics to sensible units e.g. use eV rather than Joules for quantum calculations.

Evaluate in the right order: $(\hbar/k_B)^5$ is better than $\hbar^5/k_B^5$

Numerical Limits

```
#include <climits>
#include <cfloat>
int main(){
  std::cout << "Max/Min values of various types on this computer\n";
  std::cout << "int                 = " << INT_MAX << "\t"
            << INT_MIN << "\n";
  std::cout << "unsigned int        = " << UINT_MAX << "\t"
            << 0L << "\n";
  std::cout << "long int            = " << LONG_MAX << "\t"
            << LONG_MIN << "\n";
  std::cout << "unsigned long int = " << ULONG_MAX << "\t"
            << 0UL  << "\n";
  std::cout << "float               = " << FLT_MAX << "\t\t"
            << FLT_MIN << "\n";
  std::cout << "double              = " << DBL_MAX << "\t"
            << DBL_MIN << "\n";
  std::cout << "long double         = " << LDBL_MAX << "\t"
            << LDBL_MIN << "\n";
  std::cout << "epsilon (float)     = " << FLT_EPSILON << "\n";
  std::cout << "epsilon (double)    = " << DBL_EPSILON << "\n";
}
```

Values

```
Max/Min values of various types on this computer
int                = 2147483647        -2147483648
unsigned int       = 4294967295        0
long int           = 9223372036854775807        -9223372036854775808
unsigned long int  = 18446744073709551615        0
float              = 3.40282e+38                 1.17549e-38
double             = 1.79769e+308        2.22507e-308
long double        = 1.18973e+4932        3.3621e-4932
epsilon (float)    = 1.19209e-07
epsilon (double)   = 2.22045e-16
```

In `Python`, floating-point numbers are almost always 64-bit and integers are almost always 64-bit signed, which is a **good thing**.

NaN/Infinity

```cpp
int main()
{
  std::cout << "-1.0/0.0    = " << ( -1.0 / 0.0) << "\n";
  std::cout << " 1.0/0.0    = " << ( 1.0 / 0.0) << "\n";
  std::cout << " 0.0/0.0    = " << ( 0.0 / 0.0) << "\n";
  std::cout << " exp(1000)  = " << exp(1000) << "\n";
  std::cout << " exp(-1000) = " << exp(-1000) << "\n";
  double xnan = 0.0 / 0.0;
  double xinf = 1.0 / 0.0;
  std::cout << "2 * inf     = " << 2 * xinf << "\n";
  std::cout << "2 * nan     = " << 2 * xnan << "\n";
  if ( xinf > 1 ) std::cout << "xinf is greater than 1\n";
  if ( xnan < 1 ) std::cout << "xnan is greater than 1\n";
  std::cout << "xnan is " << ( std::isnan( xnan  ) ?
                               "nan\n" : "OK\n" );
  std::cout << "xinf is " << ( finite( xinf ) ?
                               "finite\n" : "infinite\n" );
}
```

NaN/Infinity

```
-1.0/0.0    = -inf
 1.0/0.0    = inf
 0.0/0.0    = nan
 exp(1000)  = inf
 exp(-1000) = 0
2 * inf     = inf
2 * nan     = nan
xinf is greater than 1
xnan is nan
xinf is infinite
```

Summary

1. Floating point numbers are not real
2. Single precision (32-bit) `float` arithmetic is precise at a level of about 1 in 10^7 ($\sim 2^{23}$)
3. Double precision (64-bit) `double` arithmetic is precise at a level of about 1 in 10^{15} ($\sim 2^{52}$)
4. Most serious numerical work uses 64-bit precision
    - But 8, 16, or 32 bits often sufficient for storage of data or doing graphics e.g. audio CD standard has 16-bit sampling (at 44.1 kHz). Data from a radio telescope with low signal-to-noise ratio can be stored at 1 or 2 bits per sample.
5. Integers have exact representation of course, and come in various sizes. Overflow can still be a problem.