Computational Physics

Part II Physics

David Buscher <db106@cam.ac.uk>

January 2018

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 9th 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 16 March**) at 16:00.
- Manual available next week

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 30th April** 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

Deep Learning

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?

A language that is 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

Grounding in IB-level 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

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)
```

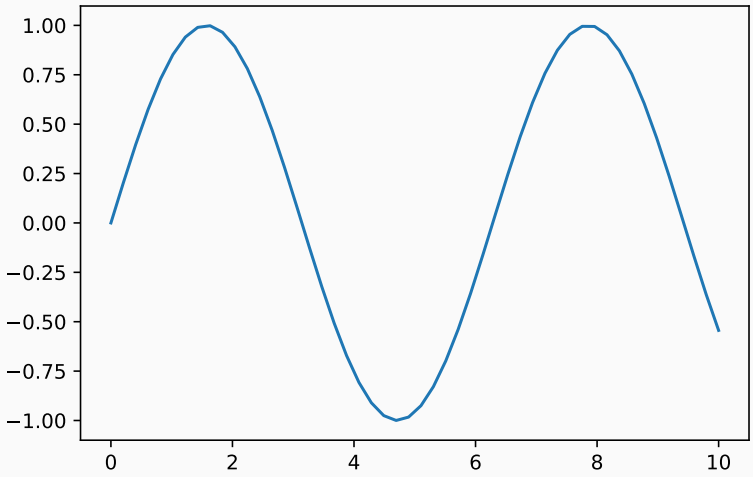
```
The cube of [1 2 3] is [ 1 8 27]
```

```
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

```
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

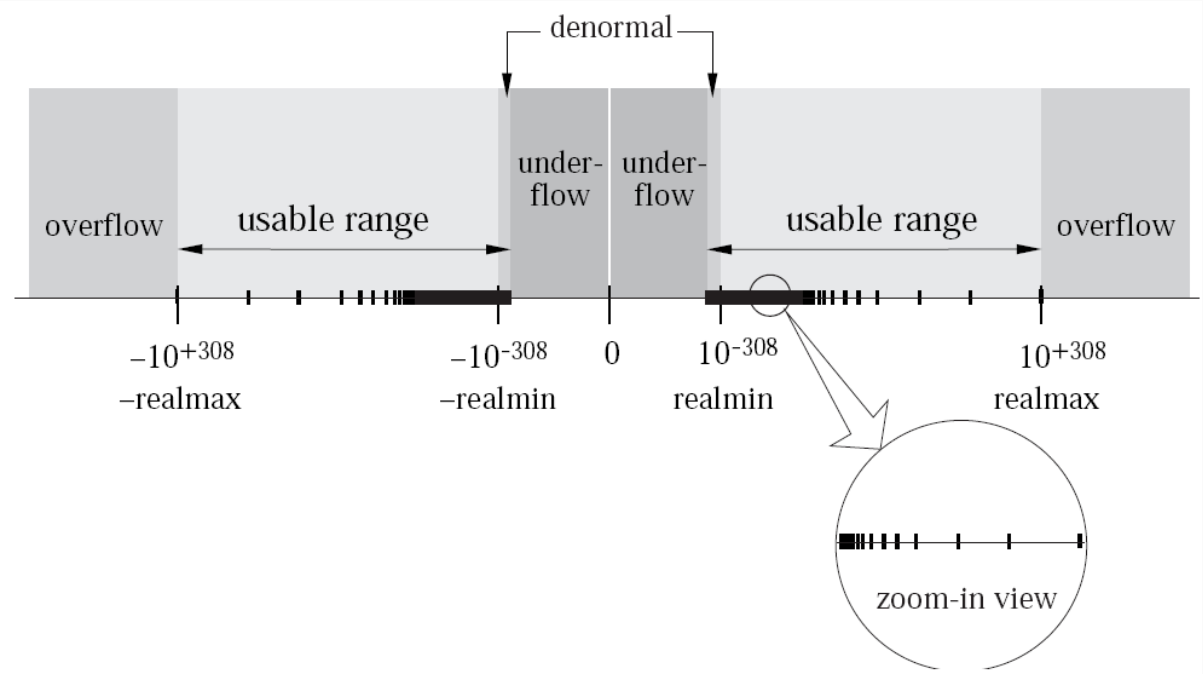
Accuracy

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

The industry standard IEEE 754 defines 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.

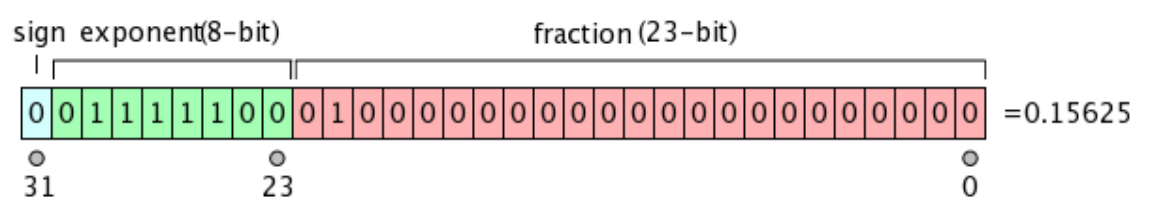
Compatibility of binary data files between machines is now straightforward. (In the dark ages, we used to have to scale the floats to integers, transfer integers, then convert back).

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

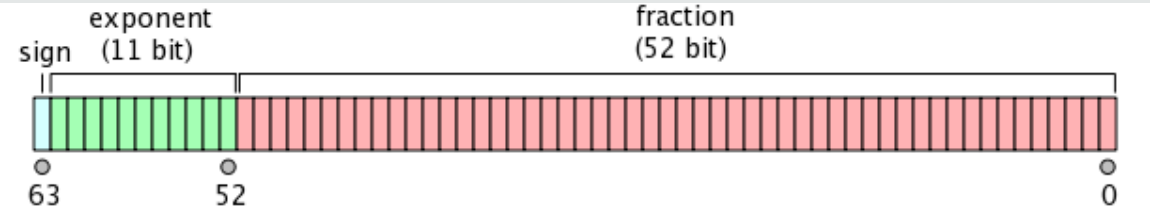
Here: $s = 0$, $e = (1111100)_2 = 124$ and $f = (0.01)_2 = 0.25$, so the number is

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

(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.

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>`.

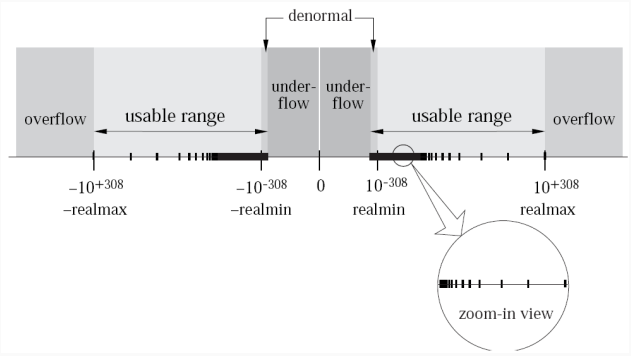
In `Python` the machine precision can be retrieved using the `numpy.finfo()` function:

```
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 $2.22 \times 10^{-16} = 2^{-52}$.

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 $\hbar^5/k_B^5$

Infinity, NaN, overflow and underflow

```
import numpy as np
#np.seterr(divide='ignore', invalid='ignore')
print(np.array([1.0, -1.0, 0.0])/0.0)
```

```
[ inf -inf  nan]
```

```
/usr/local/lib/python3.6/site-packages/ipykernel/__main__.py:3: RuntimeWa
rning: divide by zero encountered in true_divide
  app.launch_new_instance()
/usr/local/lib/python3.6/site-packages/ipykernel/__main__.py:3: RuntimeWa
rning: invalid value encountered in true_divide
  app.launch_new_instance()
```

```
print(" exp(1000)  = ", np.exp(1000) )
print(" exp(-1000) = ", np.exp(-1000) )
```

```
 exp(1000)  =  inf
 exp(-1000) =  0.0
```

```
/usr/local/lib/python3.6/site-packages/ipykernel/__main__.py:1: RuntimeWa
rning: overflow encountered in exp
  if __name__ == '__main__':
```

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)

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

Integer division in C++

```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 0x7ffee7537944
i = 2 0x7ffee7537944
```

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
```


Side effects in Python

It is not *essential* that you understand this, but it helps...

```
def test(a,b,c):
    a += 1
    b += (1,)
    c += [1]
    return a,b,c

a, b, c = [1, (1,), [1]]
print("a =",a,"b =",b,"c =",c)
d, e, f = test(a, b, c)
print("a =",a,"b =",b,"c =",c)
print("d =",d,"e =",e,"f =",f)
```

```
a = 1 b = (1,) c = [1]
a = 1 b = (1,) c = [1, 1]
d = 2 e = (1, 1) f = [1, 1]
```

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 is the main problem to watch out for.

Ordinary Differential Equations

Ordinary Differential Equations

An nth order ODE is one where the dependent variables are functions of a single variable, and the highest order derivative is n. A second-order ODE might look like

$$m \frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = f(x, \dot{x}, t)$$

We can reduce to a system of 2 first-order ODEs by a change of variables: define

$$Y_0 = x, Y_1 = \frac{\mathrm{d}x}{\mathrm{d}t}$$

And we now have:

$$\frac{\mathrm{d}Y_0}{\mathrm{d}t} = Y_1, \frac{\mathrm{d}Y_1}{\mathrm{d}t} = f(Y_0, Y_1, t)/m$$

So we need to focus on solving coupled first-order ODEs. We can reduce many interesting problems to this form.

Simple ODE example: Spinning Ring in Magnetic Field

Old IB problem:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{2}{\tau} \sin^2\theta \frac{\mathrm{d}\theta}{\mathrm{d}t}$$

with approximate solution (for light damping)

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} \approx \omega_0 e^{-t/\tau}$$

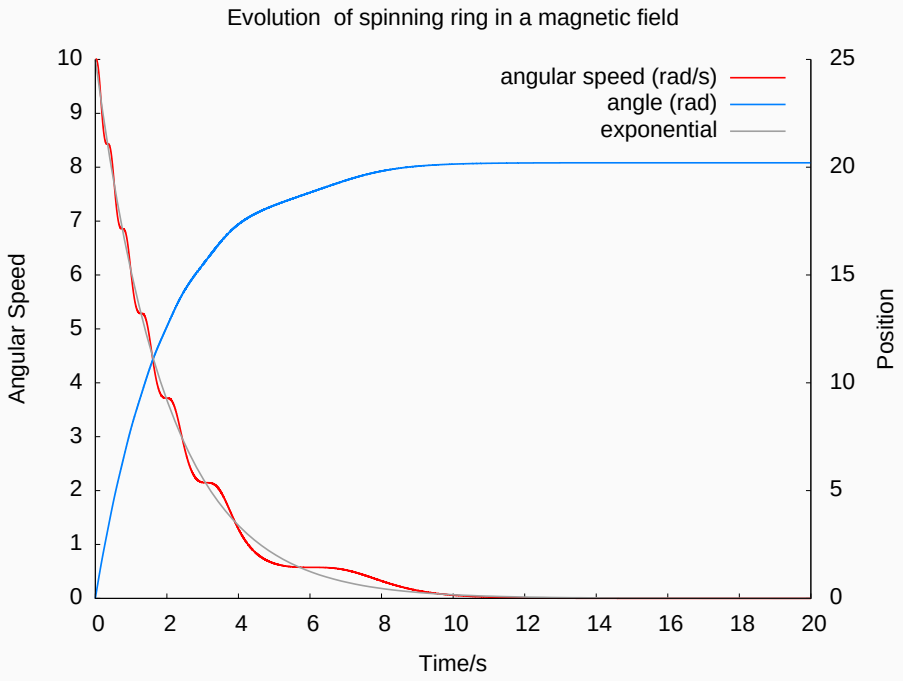
We set

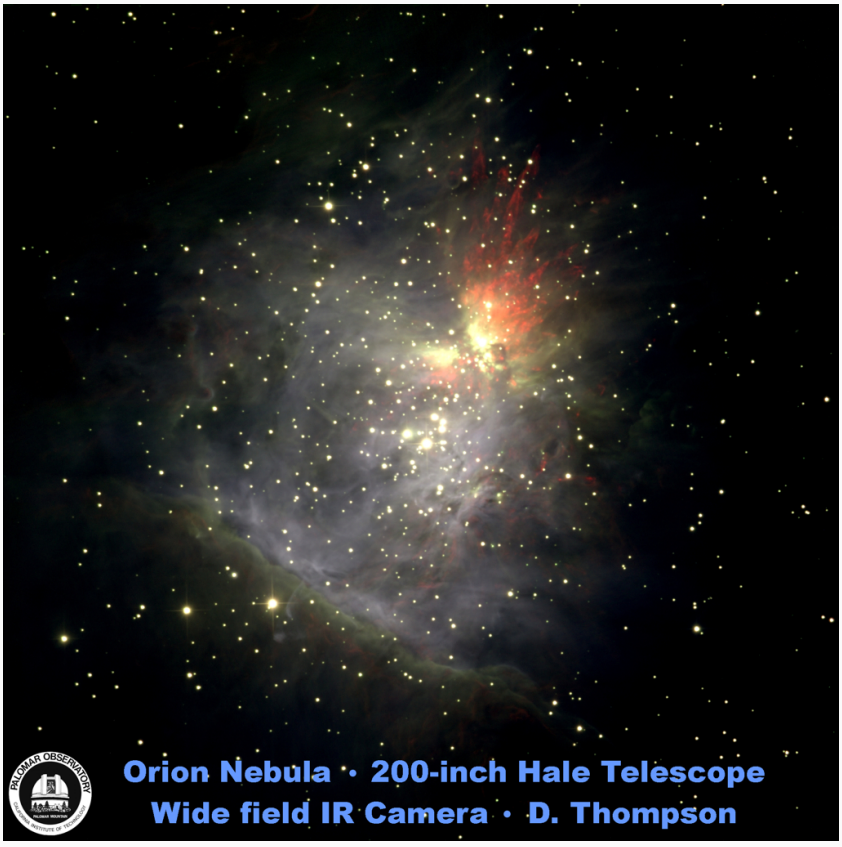
$$Y_0 \equiv \theta, Y_1 \equiv \dot{\theta}$$

to obtain

$$\begin{aligned} \dot{Y_0} &= Y_1 \\ \dot{Y_1} &= -\frac{2}{\tau} \sin^2(Y_0)\, Y_1 \end{aligned}$$

Simple ODE example: Spinning Ring in Magnetic Field





N-body problem example

For N particles moving in 3 dimension, we will need $6N$ coupled first-order ODEs to describe the system.

- $3N$ positions
- $3N$ velocities

For the case of gravity for example:

$$m_i \frac{\mathrm{d}^2 \mathbf{r_i}}{\mathrm{d}t^2} = -\sum_{i \neq j} \frac{G m_i m_j}{r_{ij}^3} \mathbf{r_{ij}}$$

with the particle index running i from 0 to $(N-1)$.

Learn to love 0-based arrays

We transform to a new set of variables, $Y_i, i = 0...(6N-1)$. Define

$$Y_0 = x_0, Y_1 = y_0, Y_2 = z_0$$

$$Y_3 = x_1, Y_4 = y_1, Y_5 = z_1$$

etc for the positions, the final position being stored in Y_{3N-1}. The speeds are then

$$Y_{3N} = \dot{x}_0, Y_{3N+1} = \dot{y}_0, Y_{3N+2} = \dot{z}_0$$

$$Y_{3N+3} = \dot{x}_1, Y_{3N+4} = \dot{y}_1, Y_{3N+5} = \dot{z}_1$$

etc. with the final element being $Y_{6N-1} = \dot{z}_{N-1}$.

We now have $3N$ trivial first order ODEs relating speeds:

$$\begin{aligned}
\dot{Y}_0 &= Y_{3N} \\
\dot{Y}_1 &= Y_{3N+1} \\
\dot{Y}_2 &= Y_{3N+2} \dots \\
\dot{Y}_{3N-1} &= Y_{6N-1}
\end{aligned}$$

and $3N$ equations specifying the accelerations:

$$\begin{aligned}
\dot{Y}_{3N} &= F_{x,0}/m_0 \\
\dot{Y}_{3N+1} &= F_{y,0}/m_0 \\
\dot{Y}_{3N+2} &= F_{z,0}/m_0
\end{aligned}$$

etc

Integrating ODEs: Euler's method

Consider one 1st order ODE

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y)$$

with a boundary condition $y(x_0) = y_0$. This is an initial value problem. (More complex problems involve specifying the solution at more than one location and other methods are needed.)

We want to find an approximate solution to this continuous function y, at a set of discrete points x_i which we assume are equally spaced (for now).

Euler's 18th century algorithm

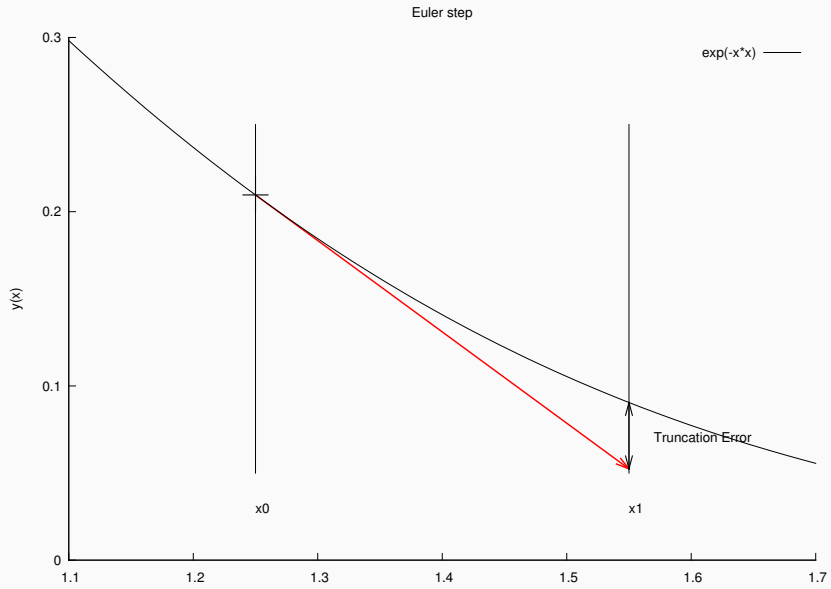
Choose a suitable step size h

- Find the gradient y'_i at the current location x_i
- Set $y_{i+1} = y_i + h\,y'_i$
- Set $x_{i+1} = x_i + h$

Repeat as necessary.

Euler's Algorithm

Illustrated for the ODE: $y' = -2xy$ (which has a solution $A\exp(-x^2)$)



Truncation Error in Euler's method

We approximate the function as a straight line over the distance h. There is an associated truncation error.

Truncation Error

Error in the estimated value due to finite step size

A Taylor expansion yields

$$y(x_i + h) = y(x_i) + h\,y'(x_i) + \frac{h^2}{2}y''(x_i) + \mathcal{O}(h^3)$$

In Euler's method we truncate the series after the linear term. The truncation error is $\mathcal{O}(h^2)$ in each step.

Truncation Error in Euler's method

- Now if we want to integrate over a range of order unity ($x = 0$ to 1 for example), then we need $\mathcal{O}(h^{-1})$ steps, so the total truncation error is $\mathcal{O}(h)$ if we assume (pessimistically) that the errors accumulate.
- So an accuracy of 1 part in 1 million needs of order a million steps. (Of course this is not true if the function is actually a straight line...)
- Euler's Method is called first order since its error over a finite scale goes as h^1. An n-th order method has a truncation error per step $\mathcal{O}(h^{n+1})$.

- So can we take an infinite number of small steps and get a perfect answer?
- No, because round-off error due to finite precision arithmetic also occurs. At each step we get a round-off error of some value ϵ, which depends on the computer's binary representation of numbers.
- Integrating over a finite range we accumulate a total round-off error $\sim \epsilon/h$, for a total error of

$$E \sim \frac{\epsilon}{h} + h$$

Moral

If we use small steps: round-off error dominates.

If we use large steps: truncation error dominates

- The minimum error occurs for $h \approx \epsilon^{1/2}$ and has a magnitude $E_{\min} \sim \epsilon^{1/2}$.
- The value of ϵ is machine-dependent:

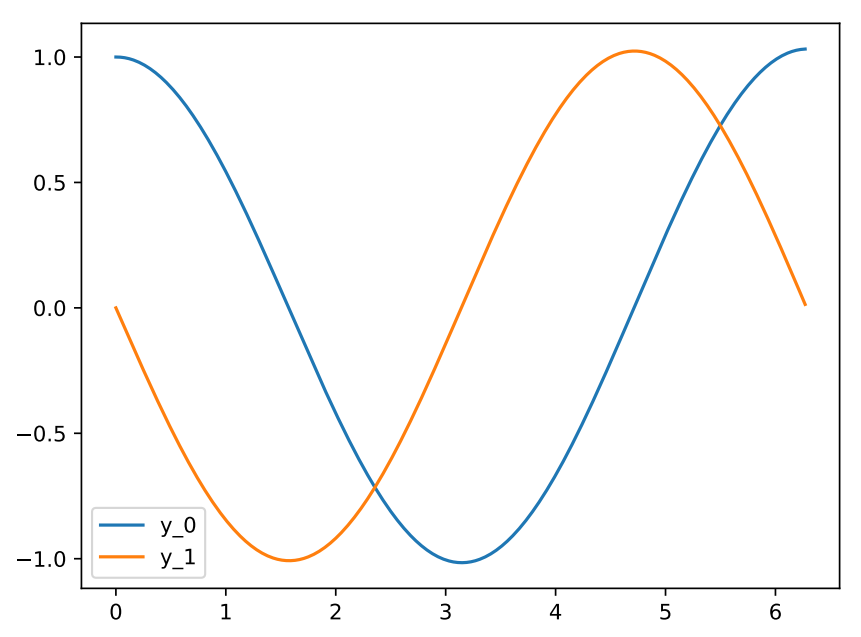
$$\texttt{float} : \epsilon = 1.19 \times 10^{-7}$$

- So in single precision arithmetic, the minimum practicable step size is $h \sim 3 \times 10^{-4}$ yielding an accuracy of about 3×10^{-4}. This is generally not good enough.
- But in double precision, the minimum practicable step is $h \sim 1 \times 10^{-8}$ yielding an accuracy of about 1×10^{-8}. This is much better.

Euler implementation example

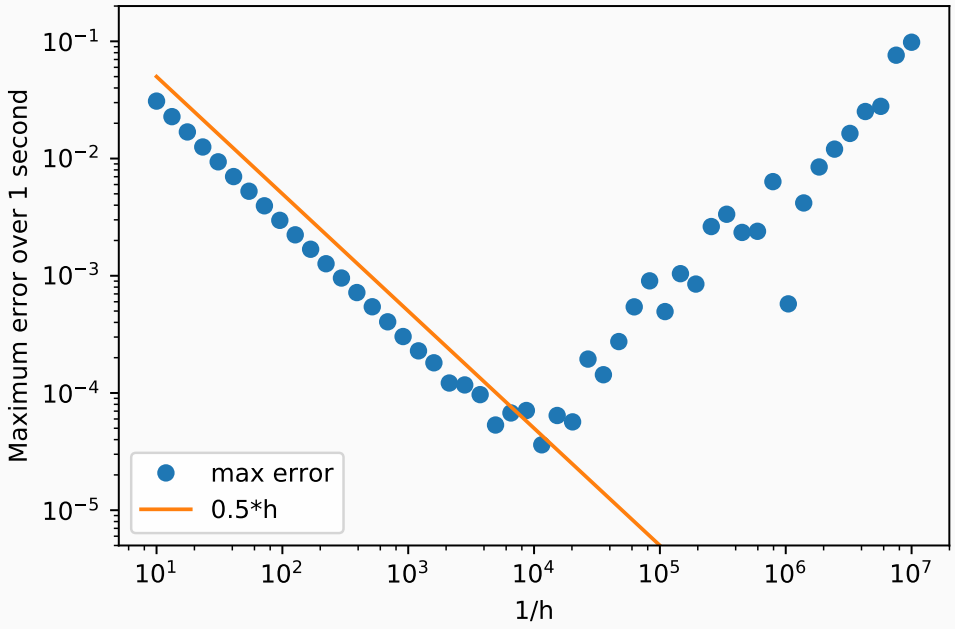
```
# Euler solution to the SHM equation x'' = -x, with x(0)=1, x'(0)=0
import numpy as np
import matplotlib.pyplot as plt

# Integrate the SHM ODE from t=0 to t=t_max using stepsize h.
# Returns three arrays: t, the time coordinate, y_0, the position
# and y_1, the velocity, for each step of the integration.
def integrate_shm(h,t_max):
    num_step=int(t_max/h)
    # Convention is y_0=x; y_1=x'
    y_0=np.zeros(num_step,dtype=np.float32)
    y_1=np.zeros(num_step,dtype=np.float32)
    t=np.zeros(num_step,dtype=np.float32)
    y_0[0]=1.0
    for i in range(num_step-1):
        dydx_0= y_1[i] # y_1= y_0'
        dydx_1=-y_0[i] # y_0=-y_1'
        y_0[i+1]=y_0[i]+h*dydx_0
        y_1[i+1]=y_1[i]+h*dydx_1
        t[i+1]=t[i]+h
    return t,y_0,y_1
```

```
t,y_0,y_1=integrate_shm(0.01,2*np.pi)
plt.plot(t,y_0,label="y_0")
plt.plot(t,y_1,label="y_1")
plt.legend()
plt.savefig("euler1.pdf",bbox_inches="tight")
```



Error in Euler's method

```
# Compare to analytic solution over 1 second
for h in np.logspace(-7,-1):
    t,y_0,y_1=integrate_shm(h,1.0)
    err=np.amax(np.abs(y_0-np.cos(t)))
    print(h,err)
```

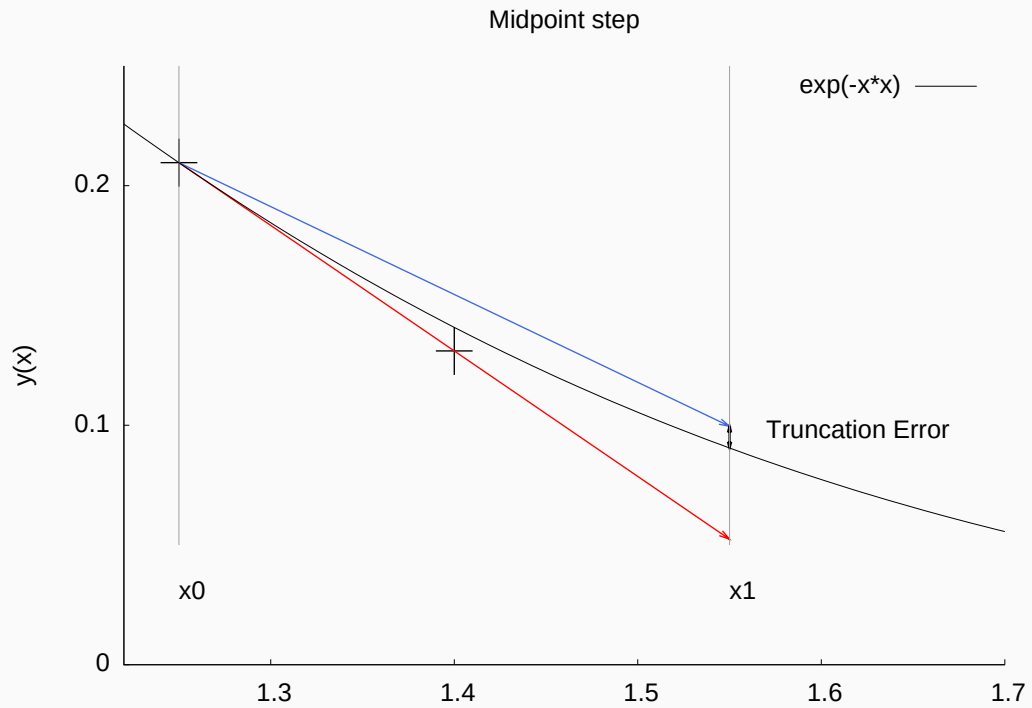


Higher Order Methods

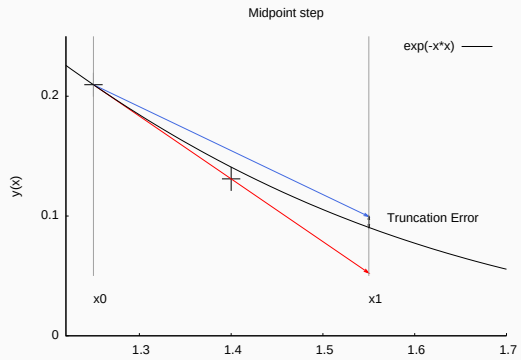
- Euler's method not generally good enough for serious computation, although in some problems it suffices, and with simple tweaks can be a useful workhorse.
- But there exist more accurate and efficient methods. The problem with Euler's method is that it is very asymmetric: it only evaluates the gradient at the beginning of the interval x_i.
- Higher order methods evaluate derivatives at sub-interval points and achieve higher accuracy. For example, the second-order mid-point method would take half a step and evaluate the gradient there.

Midpoint method: $\mathcal{O}(h^2)$

Illustrated for $x'' = -2xy$.



Midpoint method: $\mathcal{O}(h^2)$

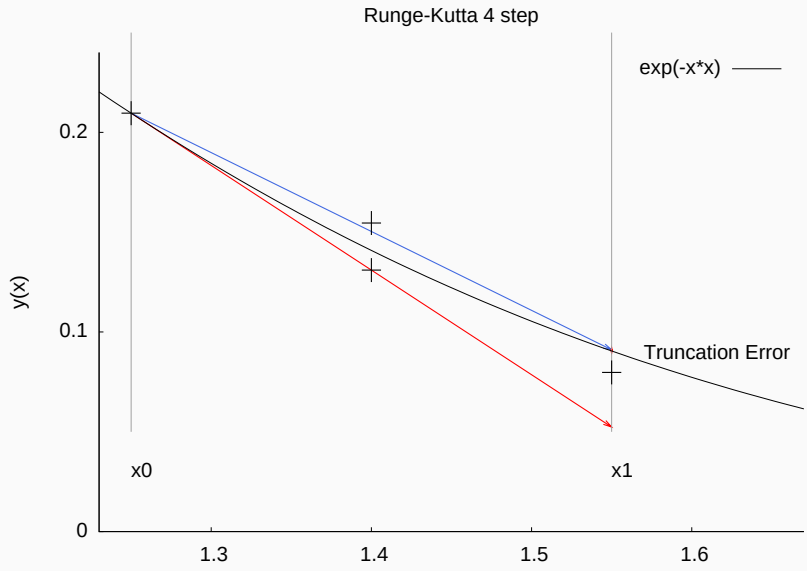


Mathematically, we have:

$$\begin{aligned} k_1 &= h\,y'(x_i, y_i) \\ k_2 &= h\,y'(x_i + h/2, y_i + k_1/2) \\ y_{i+1} &= y_i + k_2 + \mathcal{O}(h^3) \end{aligned}$$

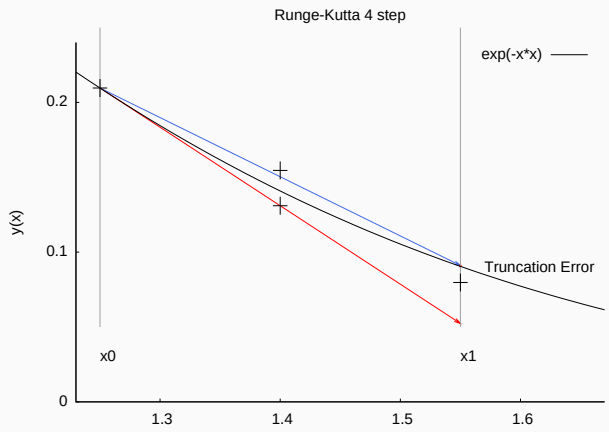
The first order errors cancel. This is called a “2nd Order Runge-Kutta method”. The price to be paid for this higher accuracy is more function evaluations (2) per step. But this is normally worthwhile.

4th order Runge-Kutta



4-th order Runge-Kutta schemes are popular, with evaluations at 4 trial points per step: the start, two midpoints, and one end point.

4th order Runge-Kutta



Mathematically:

$$\begin{aligned} k_1 &= h\,y'(x_i, y_i) \\ k_2 &= h\,y'(x_i + h/2, y_i + k_1/2) \\ k_3 &= h\,y'(x_i + h/2, y_i + k_2/2) \\ k_4 &= h\,y'(x_i + h, y_i + k_3) \end{aligned}$$

which yields a best step of

$$y_{i+1} = y_i + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + \mathcal{O}(h^5)$$

4th order Runge-Kutta

The total error E in an n'th order method is

$$E = \frac{\epsilon}{h} + h^n$$

for which h is minimised for

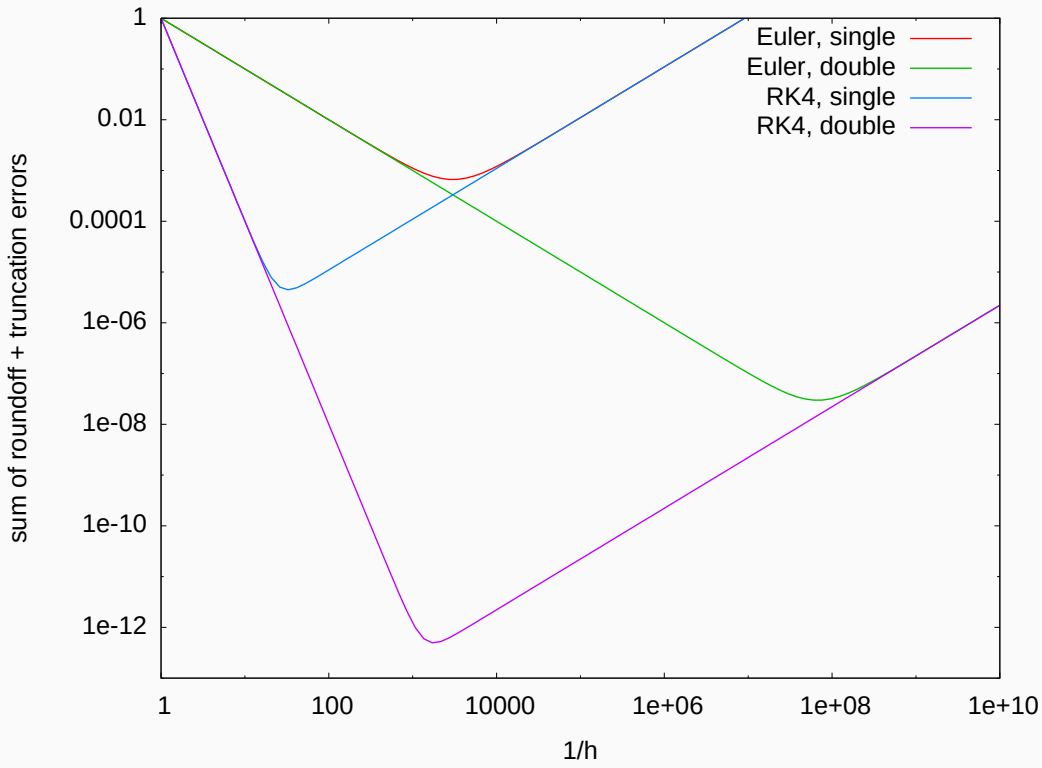
$$h_{\min} = \left(\frac{\epsilon}{n}\right)^{1/(n+1)}$$

The minimum error is then

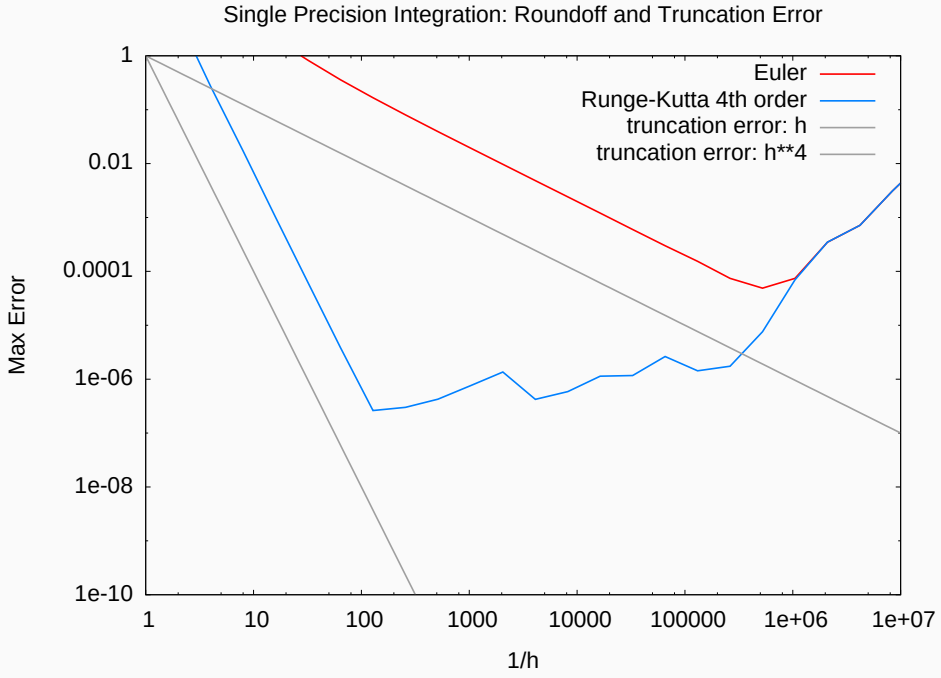
$$E_{\min} = \left(\frac{\epsilon}{n}\right)^{n/(n+1)}$$

For double precision arithmetic we get a sensible step size of $h_{\min} \sim 6 \times 10^{-4}$ and a corresponding error of $E_{\min} \approx 6 \times 10^{-13}$. (Compare this with the best values of $h \sim 10^{-8}$ and $E \sim 10^{-8}$ we obtain for Euler's method.)

Theoretical Errors in R-K integrations



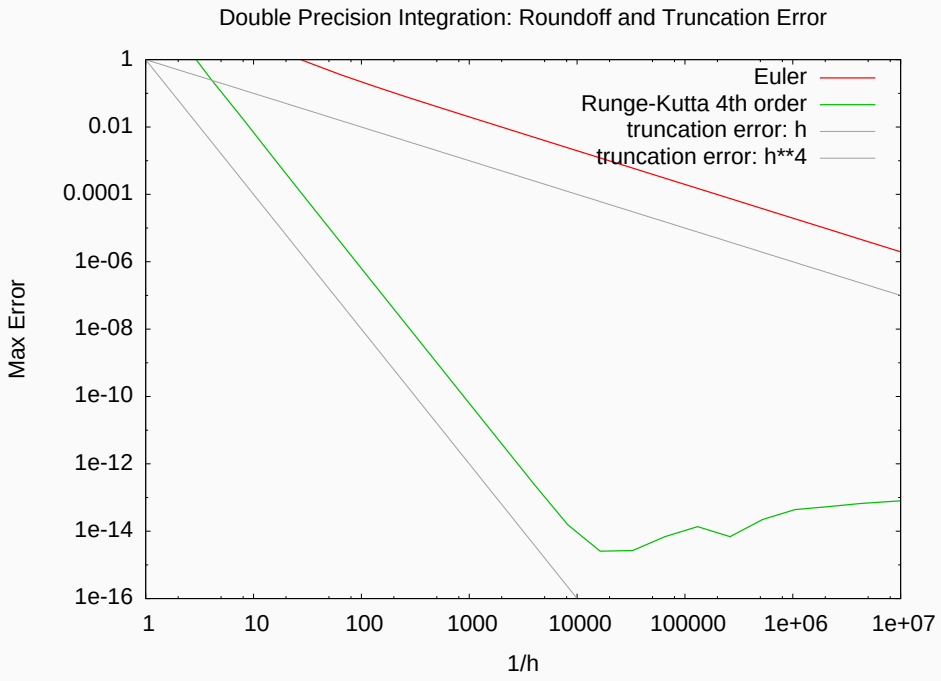
Real Errors in Runge-Kutta schemes: single-precision



Superiority of Runge-Kutta 4th order scheme is obvious.

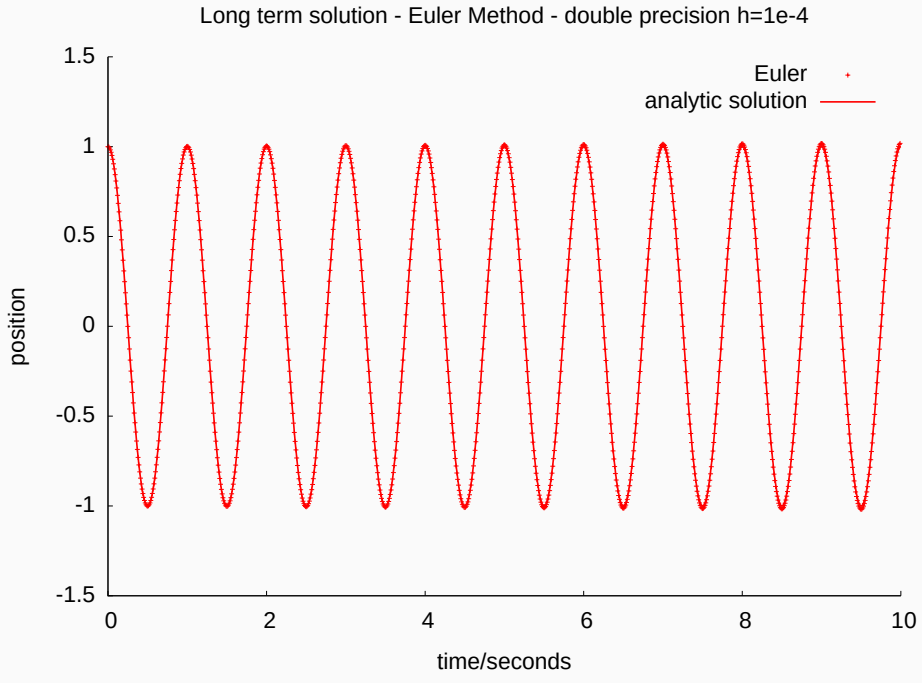
(Complete code to generate these data and the plot is in `ode1.cc` and `ode1.gp`)

Real Errors in Runge-Kutta schemes: double precision



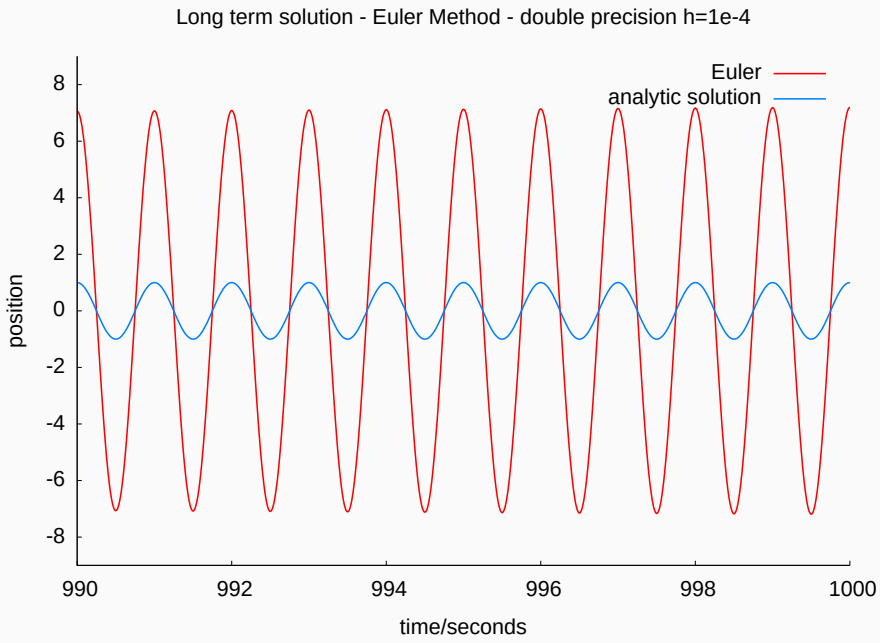
Superiority of double precision is clear. For this problem $h < 10^{-4}$ is reasonable

Long term solution using Euler. $\ddot{x} = -4\pi^2 x$



But Euler's method appears to be doing a good job for the first 10 periods (seconds)...maybe it's crude but effective. Let's keep integrating...

Long term solution using Euler



Amplitude $\sim$ 8 times too large after 1000 oscillations.

Origin of the Numerical Instability in Euler's Method

The scheme we are using updates position and velocity according to:

$$\begin{aligned} x_{i+1} &= x_i + \dot{x}_i \, \Delta t \\ \dot{x}_{i+1} &= \dot{x}_i + \ddot{x}_i \, \Delta t \end{aligned}$$

The total energy is just

$$E_i = \frac{1}{2} m \dot{x}_i^2 + \frac{1}{2} k x_i^2$$

So that after a little work we find

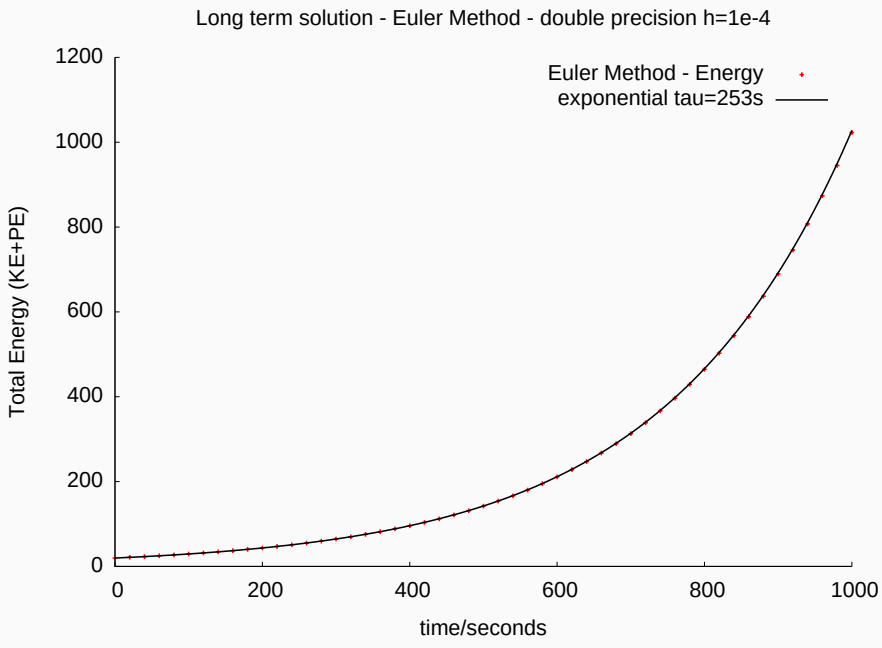
$$E_{i+1} = E_i + \left(\frac{1}{2} m \ddot{x}_i^2 + \frac{1}{2} k \dot{x}_i^2 \right) (\Delta t)^2$$

We can write this as

$$\frac{\Delta E}{\Delta t} = E\frac{k\Delta t}{m}$$

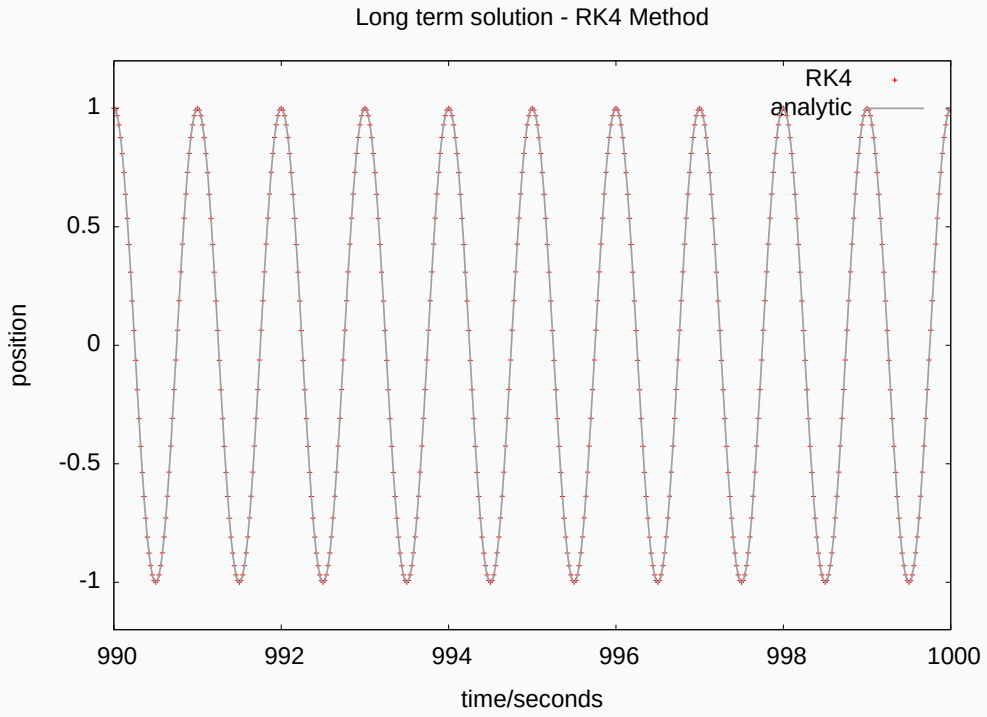
- We see that E will increase on each step by an amount proportional to E: we get exponential growth with a timescale $\tau = m/(k\Delta t)$. Even if we make the step smaller, the energy will still grow without limit.
- The method is intrinsically unstable for this oscillatory equation.
- But the same does not necessarily apply to other equations.

Long term solution using Euler



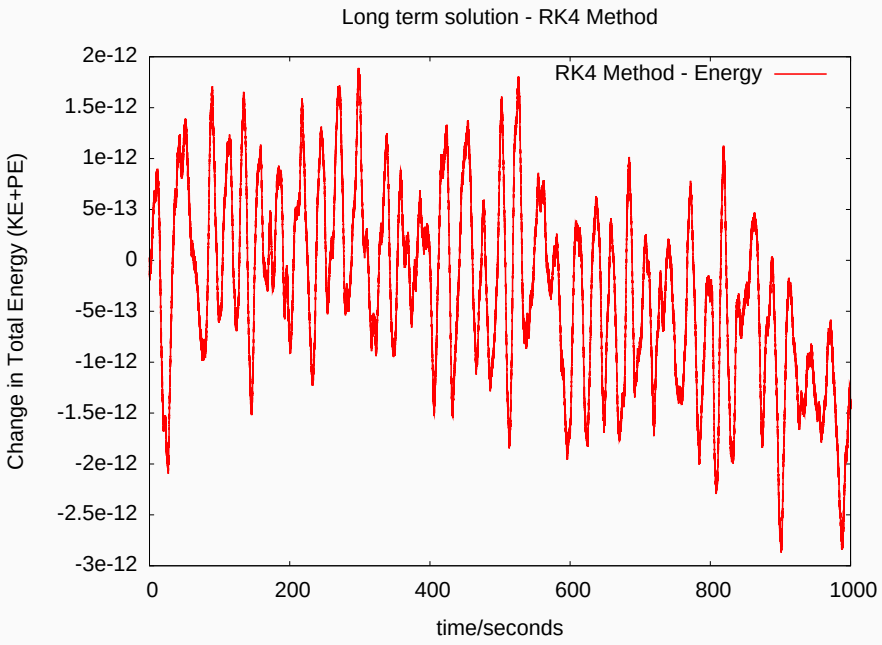
Initial energy = $2\pi^2$.
Exponential growth timescale $\tau = m/(k\Delta t) = 1/(4\pi^2\Delta t) = 253$s.

Long term solution using Runge-Kutta 4th order



This looks a lot better...

Long term solution using RK4



The energy stays constant to 1 part in 10^{12} again using $h = 10^{-4}$. True even after 10^6 oscillations.

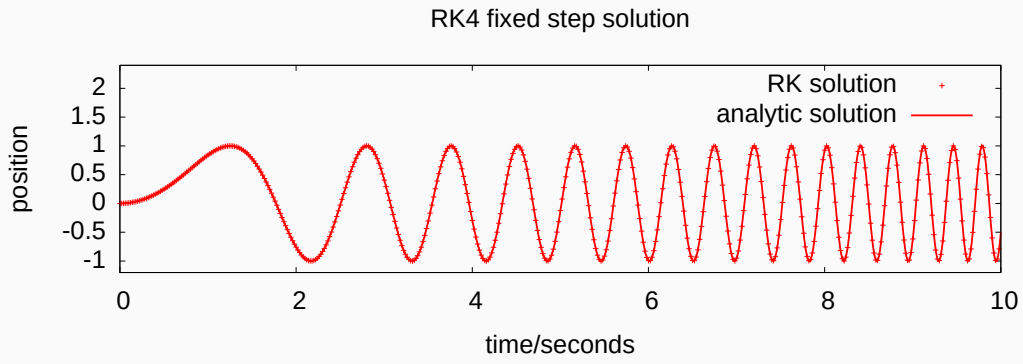
Adaptive Stepping

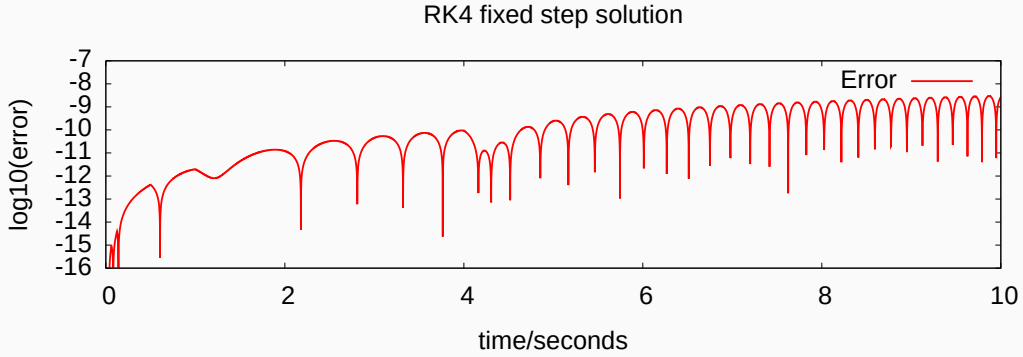
- In the simple SHM problem studied earlier, the period is constant, and using a constant step size h to advance the integration made sense.
- But for many ODEs, the scale over which the solution changes appreciably is not constant. For example

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -4\,t^2\,\sin(t^2) + 2\,\cos(t^2)$$

  has a solution $x = \sin(t^2)$ which changes progressively more quickly as t increases.
- A physical example might be the close approach of two bodies interacting via an inverse square law potential — stars scattering off each other in a cluster, or the case of Rutherford scattering.
- In such cases, a constant step size is not sensible. To get an accurate answer we need to run with a very small step size which is very inefficient.

Look how the error evolves if we use a 4th order Runge-Kutta method with fixed step size of 10^{-5}:





Adaptive Step Methods try to find optimal step lengths for a given problem. One possible way of doing this is to

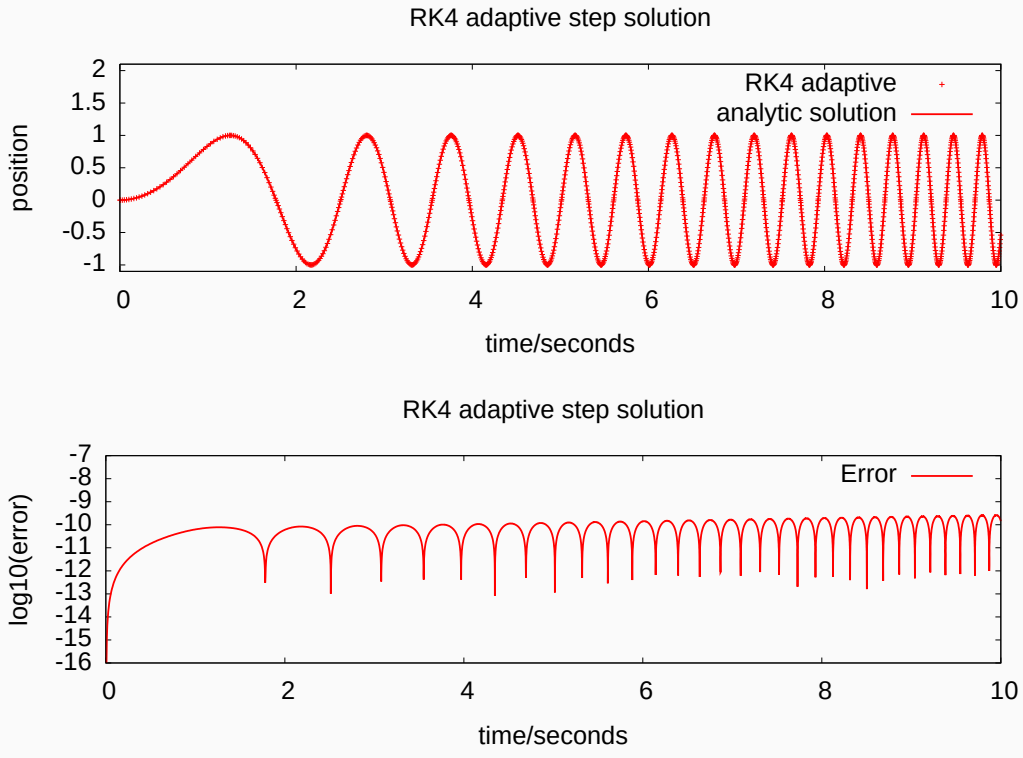
- Take a trial step of size h
- Take two trial steps of length $h/2$

The difference between these two solutions is an estimate of the truncation error in one step E.

For a 4th order Runge-Kutta scheme, the truncation error goes as h^5, so we can estimate the desired step length using

$$h_{\text{pred}} = h \left| \frac{E_0}{E} \right|^{1/5} \tag{1}$$

where E_0 is the desired error.



Adaptive step solution is more accurate and took only 6000 steps; the fixed-step solution took a million. So even though there were more function evaluations per step, it was more efficient.

Other Topics

There is a lot we have not had time to cover:

1. More sophisticated adaptive stepping methods: you will see the method of "Burlischer-Stoer" recommended
2. Shooting methods: where we have boundary conditions at different points.
3. Further theory on stability of various schemes
4. Partial Differential Equations

Beond the level of this course, and not needed for the problems we cover. Look in "Numerical Recipes" and references therein if you want to become an expert.

ODEs: Lessons Learnt

1. Single precision (32-bit) arithmetic is not good enough for serious numerical work.
2. But double precision (64-bit) arithmetic is very good. We got better than 1 part in 10^{14} accuracy with our RK4 integration for our simple test problem.
3. Test your code against simple cases where analytic solutions exist. Run codes over long periods to test for stability. Have you discovered new physics or a numerical issue?
4. If you have no analytic solution for comparison, measure conserved quantities (energy, angular momentum, momentum) to see how well they are conserved in your simulation.
5. Writing your own integrators is instructive, but many years of experience have gone into library routines, so once you understand the basics, use them.
6. There are few hard and fast rules in solving ODEs. Every problem is different, but Runge-Kutta 4th order with adaptive steps is often a reliable algorithm.

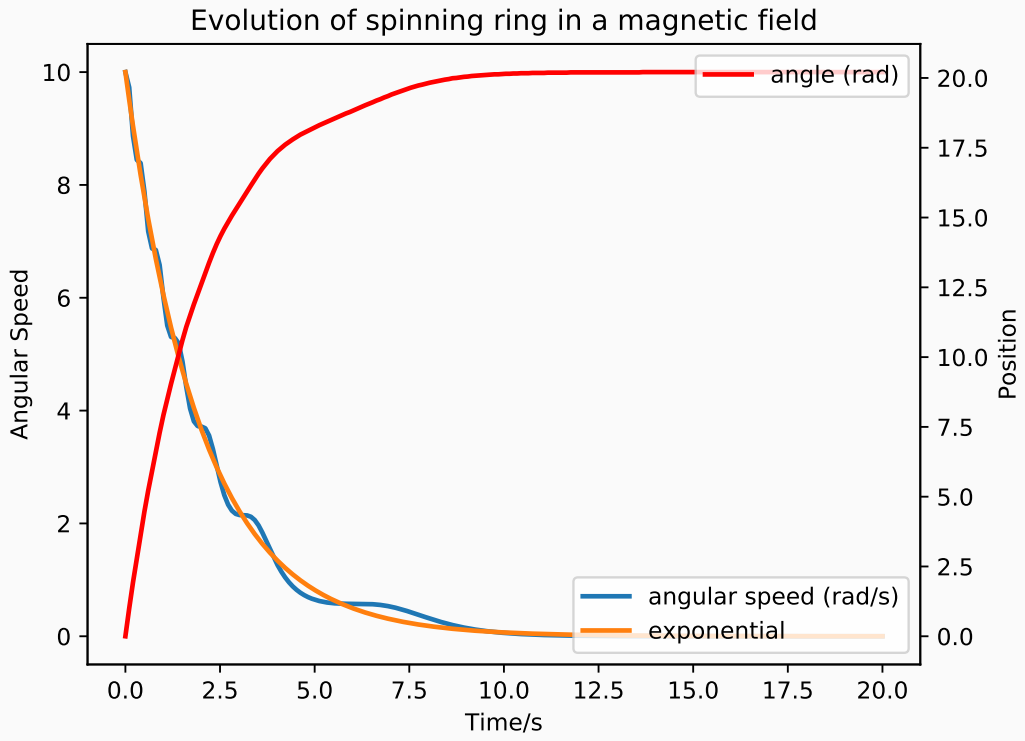
Simple ODE example: ode_ring1_scipy.py

```
# Solve the ODE for a conducting ring spinning in a magnetic field.
import numpy as np
import scipy.integrate

# This function evaluates the derivatives for the equation
# d^2 theta/dt^2 = - (2/tau) * sin^2(theta) * d theta/dt
# We work in the transformed variables y[0] = theta, y[1] = d(theta)/dt
def derivatives(y,t,tau):
    return [y[1], -(2.0/tau)*np.sin(y[0])**2*y[1]]

# Main code starts here
t=np.linspace(0.0, 20.0, 200)
y0=[0.0, 10.0]
tau=2.0
y=scipy.integrate.odeint(derivatives,y0,t,args=(tau,))
for i in range(len(y)):
    print("{:8.4g} {:8.4g} {:8.4g}".format(t[i],y[i,0],y[i,1]))
```


Simple ODE example: ode_ring1_pyplot.py

```
import matplotlib.pyplot as plt
import numpy as np
# Load the data
x,y,dydx=np.loadtxt("ode_ring1_scipy.out",unpack=True)
# Plot angular speed and analytical approximation
fig, ax1 = plt.subplots()
ax1.plot(x,dydx,lw=2,label="angular speed (rad/s)")
ax1.plot(x,10*np.exp(-x/2.0),lw=2,label="exponential")
ax1.set_xlabel("Time/s")
ax1.set_ylabel("Angular Speed")
ax1.set_title("Evolution of spinning ring in a magnetic field")
ax1.legend(loc="lower right")
# Angular position plotted on same plot with second set of axes
ax2=ax1.twinx()
ax2.plot(x,y,lw=2,label="angle (rad)",color="red")
ax2.set_ylabel("Position")
ax2.legend(loc="upper right")
plt.savefig("ode_ring1_scipy.pdf",bbox_inches="tight",
            transparent=True)
```

Simple ODE example: result



ODEs with the Gnu Science Library: ode_ring1.cc

```
// GSL solution to the ODE for a spinning ring in a magnetic field
// d^2 theta/dt^2 = - (2/tau) * sin^2(theta) * d theta/dt
#include <iostream>
#include <cmath>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_odeiv.h>
// Evaluate the derivatives: we work in the transformed variables
// y[0] = theta, y[1] = d(theta)/dt
int calc_derivs(double t, const double y[], double dydx[],
                void *params) {
  // Extract the parameters from the pointer param*.  In this case
  // there is only one - the value tau in the differential equation.
  double tau = *(double *)(params);
  dydx[0] = y[1];
  dydx[1] = -(2.0 / tau) * pow(sin(y[0]), 2) * y[1];
  return GSL_SUCCESS;
}
```

ODEs with the Gnu Science Library: ode_ring1.cc (cont'd)

```
int main() {
  // Initial conditions:
  double tau = 2.0;
  const int n_equations = 2;
  double y[n_equations] = {0, 10};
  double t = 0.0;
  // Create a stepping function:
  gsl_odeiv_step *gsl_step =
      gsl_odeiv_step_alloc(gsl_odeiv_step_rk4, n_equations);
  // Adaptive step control: let's use fixed steps here:
  gsl_odeiv_control *gsl_control = NULL;
  // Create an evolution function:
  gsl_odeiv_evolve *gsl_evolve = gsl_odeiv_evolve_alloc(n_equations);
  // Set up the system needed by GSL: The 4th arg is a pointer
  // to any parameters needed by the evaluator. The 2nd arg
  // points to the jacobian function if needed (it's not needed here).
  gsl_odeiv_system gsl_sys = {calc_derivs, NULL, n_equations, &tau};
  double t_max = 20.0;
  double h = 1e-3;
```

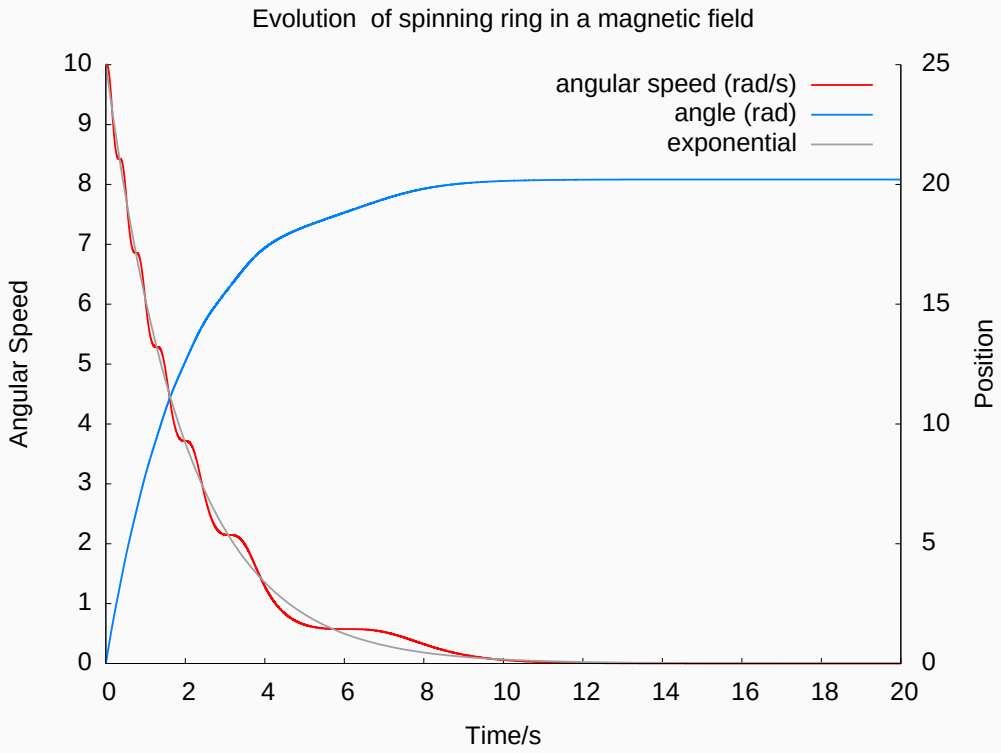

ODEs with the Gnu Science Library: ode_ring1.cc (cont'd)

```
  // Main loop: advance solution until t_max reached.
  while (t < t_max) {
    std::cout << t << " " << y[0] << " " << y[1] << "\n";
    int status = gsl_odeiv_evolve_apply(gsl_evolve, gsl_control,
                                        gsl_step, &gsl_sys, &t,
                                        t_max, &h, y);
    if (status != GSL_SUCCESS) break;
  }
  // Tidy up the GSL objects for neatness:
  gsl_odeiv_evolve_free(gsl_evolve);
  gsl_odeiv_step_free(gsl_step);
  return 0;
}
```

Simple ODE example: gnuplot code

```
reset
set term png size 1200,900 font "LiberationSans-Regular" 22
set output 'ode_ring1.png'
set xlabel 'Time/s'
set ylabel 'Angular Speed'
set y2label 'Position'
set y2tics
set title 'Evolution  of spinning ring in a magnetic field'
f='ode_ring1.out'
set border 11
set tics nomirror
tau=2.0
theory(x)=10*exp(-x/tau)
plot f using 1:3 w l lc 8 lw 3 title 'angular speed (rad/s)',\
     f u 1:2 axes x1y2 w l lc -1 lw 3 title 'angle (rad)',\
     theory(x) w l lc 1 lw 2 title 'exponential'
```


Simple ODE example: GSL result



Partial Differential Equations (PDEs)

We spent a lot of time looking at ODEs. We note that the GNU Science Library has extensive support for ODE solution. But there is nothing for PDEs. Why is this?

PDEs come in many different types and there are many methods of solution. Specialised techniques are created for the different types.

Physical applications include

- Wave equation: $\frac{\partial^2 \psi}{\partial t^2} = c^2 \frac{\partial^2 \psi}{\partial x^2}$
- Navier-Stokes equation for fluids
- Laplace's equation
- Diffusion equation

Solution methods include

- Finite Differences
- Fourier methods

Often developed specifically for a particular type of problem.

One-dimensional and higher-dimensional arrays

Sampled data

Theoretical models of physical phenomena and experimental data from sensors are often represented in terms of **continuous functions** $y(x)$ of a coordinate x. To represent these functions in a computer we often use **sampled** values: samples of the function at a discrete set of coordinates.

In other cases, our models are naturally **discrete**, for example the spin at each lattice location in the Ising model.

This kind of data can be represented as a set of points (x_i, y_i) consisting of N values y_i at locations x_i. Of course, the positions and the values can have more than one dimension:

- A picture $I(x_i, y_i)$
- Position of a particle at different times $\mathbf{r}(t_i)$
- Magnetic field measured in 3-D space $\mathbf{B}(\mathbf{r}_i)$ — need 3 scalars for each of $\mathbf{B}$ and $\mathbf{r}_i$

Arrays

These data are naturally represented in computers as arrays: collections of objects (usually numbers) which can be accessed using an index.

We will restrict our attention to the case where the objects in a given array are of the **same type** (for example 64-bit floating-point numbers), and the index is a set of one or more **integers**.

One-dimensional arrays in Python

A single standard array type forms the basis of the **numpy** package

```
import numpy as np
a=np.array([0,1,2,3])
b=np.arange(4)
print(a,b)
print(a[1],a[:2],np.sqrt(a))
```

```
[0 1 2 3] [0 1 2 3]
1 [0 1] [ 0.          1.          1.41421356  1.73205081]
```


One-dimensional Arrays in C and C++

Statically defined C-style arrays are great for simple applications:

```
int main() {
  const int n = 10;
  double h[n];
  for (int i = 0; i < n; i++)
    h[i] = sqrt(i);
  std::cout << h << " " << &h[0] << "\n";
}
```

```
0x7ffeea735910 0x7ffeea735910
```

Note **h** is a pointer to the first element of the array; it is exactly the same as `&h[0]`

The C++ vector class

You are strongly recommended to use the `std::vector` container class of the C++ Standard Template Library (STL) for storing your arrays.

Using the C++ `vector` container we write:

```
#include <vector>
#include <cmath>
int main() {
  const int n = 10;
  std::vector<double> h(n);
  for (int i = 0; i < n; i++)
    h[i] = sqrt(i);
}
```

So far, this container object `h` looks rather like our normal C array. But we can do much more with it...


```
int main() {
  vector<int> a(3, 1);           // a = 1,1,1
  a[2] = 9;                      // a = 1,1,9
  a.push_back(8);                // a = 1,1,9,8
  a.insert(a.begin(), 5);        // a = 5,1,1,9,8
  a.clear();                     // a is empty
  a.resize(6);                   // a= 0 0 0 0 0 0
  // a[99]=0;                    // Bounds error, unhelpful crash!
  cout << a.at(99);              // Bounds error, throws useful exception
}
```

```
libc++abi.dylib: terminating with uncaught exception
of type std::out_of_range: vector
```

The `vector` container is very powerful:

- dynamic, and looks after memory management for us;
- we can pass data to a function expecting a “normal” C or C++ array by taking the address of the first element, `&a[0]` ;
- If you define a new class, you can easily make a vector of that type which will behave in the ways you would expect.

Computing with Multi-dimensional Arrays

Arrays in more than one dimension are common in physics: as well as matrices, we have for example images $I(x, y)$, or a 2-D array of spins in the Ising model.

In matrix notation we denote position in the array with a 2-dimensional index

$$M = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \end{bmatrix}$$

This is straightforward to replicate in python

```
import numpy as np
M=np.array([[0,1,2,3],
            [4,5,6,7],
            [8,9,10,11]])
print(M[2,2])
```

```
10
```


Memory organisation of multi-dimensional arrays

Computers have a one-dimensional address space: every byte of memory has a unique single integer address. So we have to decide how to arrange our multi-dimensional data in memory.

We can find out how Python does this using the `numpy.reshape()` function:

```
M1=np.reshape(M, [12])
print(M1)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Python and C++ use the same storage format for multi-dimensional arrays, so-called "row-major" storage. FORTRAN and MATLAB use "column-major storage", so the 2-dimensional array M above would appear in memory as `0, 4, 8, 1, 5, 9, 2, 6`....

The difference is not material except when transferring data between programs written in the different languages.

Implementing multi-dimensional arrays in C++

Static C-style arrays are simple and occasionally the right thing to use, but their sizes are fixed. For example `double cube[8][16][32];` creates $8 \times 16 \times 32$ contiguous numbers in memory.

In general, however, it is best to create a one-dimensional array big enough to hold your multi-dimensional array and do your own indexing.

```
#include <vector>
int main() {
  int n_rows = 3;
  int n_cols = 2;
  std::vector<double> a(n_rows * n_cols, 0.0);
  for (int row = 0; row < n_rows; row++)
    for (int col = 0; col < n_cols; col++)
      a[row * n_cols + col] = row + col; // note indexing
}
```


Multi-dimensional arrays using C++ classes

An alternative is to use a class which we can use to represent two-dimensional arrays.

```
int main() {
  int n_row = 3;
  int n_col = 2;
  Matrix M(n_row, n_col);
  for (int row = 0; row < n_row; row++)
    for (int col = 0; col < n_col; col++)
      M(row, col) = col + row; // or something more useful...
}
```

Note that we use round brackets for access not square ones (because access is actually a function call). Look in `cavlib/arrays.hh` for how this `Matrix` class is implemented.

Other implementations exist, e.g. the BOOST libraries — recommended, but not for this course.

1 and 2-D data structures in GSL

Note that GSL defines its own structures to represent vectors and matrices.

To use the routines, we either

1. use the GSL data structures for our arrays, or
2. use our own C++ arrays (`vector<double>` for example) and allow GSL to view and manipulate these arrays.

The second approach is encouraged because C++ vectors are good things.

The Fast Fourier Transform

Fourier transform of a continuous function

$$H(f) = \int_{-\infty}^{\infty} h(t) \exp(-2\pi i f t) \mathrm{d}t$$
$$h(t) = \int_{-\infty}^{\infty} H(f) \exp(2\pi i f t) \mathrm{d}f$$

with h and H being continuous functions of t and f.

Recall that...

- the FT of a real signal has $H(f) = H^*(-f)$
- The power spectral density (PSD) of a signal is the power between f and $f + \mathrm{d}f$:

$$P(f) = |H(f)|^2 + |H(-f)|^2$$

For a real signal $h(t)$, the two terms are equal.

Fourier Series for a periodic signal

If our signal is measured over a period $0 \leq t \leq T$, and if we assume $h(t)$ is periodic, the Fourier transform now has components at discrete frequencies $f_n = n/T$. Fourier Series:

$$H_n = \int_0^T h(t) \exp(-2\pi i n t/T) \mathrm{d}t$$
$$h(t) = \frac{1}{T} \sum_{n=-\infty}^{\infty} H_n \exp(2\pi i n t/T)$$

i.e. an infinite number of discrete frequencies can represent a continuous periodic signal.

Sampling and the Nyquist frequency

- Consider sampling the signal $h(t)$ at N uniformly-spaced points:

$$t_k = k\Delta, k = 0, 1, \ldots (N-1)$$

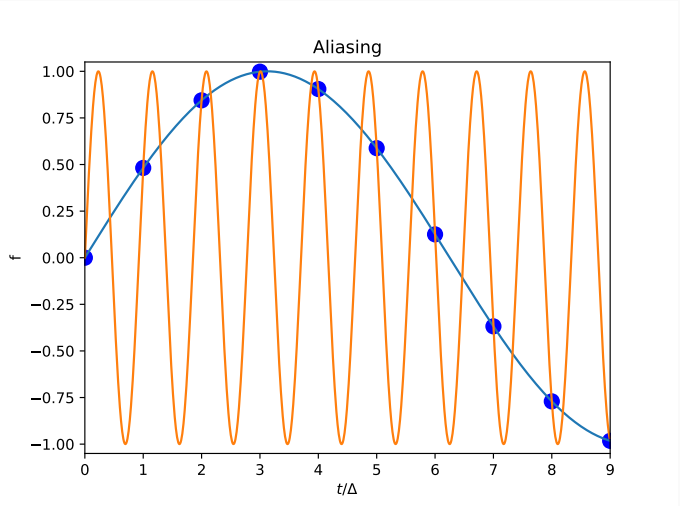
  with $T = N\,\Delta$
- The sampling rate is $1/\Delta$.
- There is a maximum representable frequency in such a signal, called the Nyquist critical frequency f_c, where we have two samples per cycle:

$$f_c = \frac{1}{2\Delta}$$

- Sampling Theorem: If $h(t)$ is band limited to frequencies $< f_c$ (i.e. if $H(f) = 0$ for $f > f_c$) then $h(t)$ is completely determined by h_n.

Aliasing

Higher frequencies than the Nyquist frequency are aliased into the range $-f_c < f < f_c$ because the frequency $f + 2f_c$ [i.e. $f + (1/\Delta)$] produces exactly the same samples as f:



Ideally we bandpass filter the signal before sampling to ensure it is bandwidth-limited and then no aliasing can occur.

Discrete Fourier Transform

Given our N samples h_k, we can construct N frequencies which approximate the continuous Fourier transform, with the highest frequency being the critical frequency f_c. It's simplest for now to assume that N is even. We define the discrete Fourier transform as

$$H_n = \sum_{k=0}^{N-1} h_k \, e^{-2\pi \, \mathrm{i} \, k \, n/N}$$

which maps N time-domain samples into N frequencies, which are

$$f_n = \frac{n}{N\Delta} = \frac{2n}{N} f_c$$

We now have a discrete signal in the time and frequency domains, with the functions **periodic in both domains**: $h_{k+N} = h_k$ and $H_{n+N} = H_n$.

The discrete frequencies are $f_n = n/(N\Delta) = 2nf_c/N$, with n running from $n = 0$ to $(N-1)$:

- $n = 0$ is zero frequency (sum of input values)
- $1 \leq n \leq (N/2)$ are positive frequencies, with $(N/2)$ being the highest (critical frequency f_c)
- $(N/2)+1 \leq n \leq (N-1)$ can be thought of as negative frequencies: we can subtract $2f_c = 1/\Delta$ from them and they are the same because H_n is periodic

There is an exact inverse:

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n \, e^{2\pi \, \mathrm{i} \, k \, n/N}$$

The inverse is the same as forward transform except change the sign in the exponential and divide by N. Note it does not contain Δ in the definition. If we want the true FT values, we have $H(f_n) \approx \Delta H_n$.

Fast Fourier Transform (FFT)

- The FFT is an efficient method for calculating the Discrete Fourier Transform (DFT). It is important as it has revolutionised signal processing in many fields.
- How much computation is involved in a DFT? We can write

$$H_n = \sum_{k=0}^{N-1} W^{nk} h_k \tag{2}$$

where

$$W \equiv \exp(-2\pi \mathrm{i}/N) \tag{3}$$

- This looks like a matrix multiplication with a square matrix W whose $N \times N$ elements W_{nk} multiply the vector h_k of length N. This is an $\mathcal{O}(N^2)$ process i.e. its compute time is dominated by a number of complex multiplications proportional to N^2

FFT

- In fact, the FFT algorithm can do the same job in $\mathcal{O}(N \log_2 N)$ operations.
- Time saving is huge: for $N = 10^6$, $N^2/(N \log_2 N) \approx 50,000$. (50,000s is 14 hours)
- "Discovered" by Danielson & Lanczos (1942), computer discovery by Cooley and Tukey (1965), but original ideas goes back at least as far as Gauss (1802).

A DFT of length N can be written as the sum of two DFTs of length $N/2$. So if N is a power of 2, we can apply this theorem over and over again, $\log_2(N)$ times in fact, until we end up with $N = 1$. Split the DFT into odd and even terms: consider the expression for H_n which we can write

$$\sum_k^{\text{even}} h_k \exp(-2\pi \mathrm{i} kn/N) + \sum_k^{\text{odd}} h_k \exp(-2\pi \mathrm{i} kn/N) =$$

$$\sum_{m=0}^{\mathrm{N}/2-1} h_k \exp(-2\pi \mathrm{i}(2m)n/N) + \sum_{m=0}^{\mathrm{N}/2-1} h_k \exp(-2\pi \mathrm{i}(2m+1)n/N) =$$

$$\sum_{m=0}^{\mathrm{N}/2-1} h_k \exp(-2\pi \mathrm{i} mn/(N/2)) + \exp(-2\pi \mathrm{i} n/N) \sum_{m=0}^{\mathrm{N}/2-1} h_k \exp(-2\pi \mathrm{i} mn/(N/2))$$

Because $2(N/2)^2 < N^2$ this is $\sim 2\times$ quicker to calculate. And further more we can apply repeatedly until N is small.

- Best case: If N is a power of two, get very efficient FFTs. Highly recommended to use 2^N if efficiency important. Either design your experiment correctly, or pad with zeros until your data blocks have length 2^N!
- But fast techniques also exist when N can be factorised.
- Worst case: N is prime.

But computer are so fast these days you may never need to worry about this for simple applications...

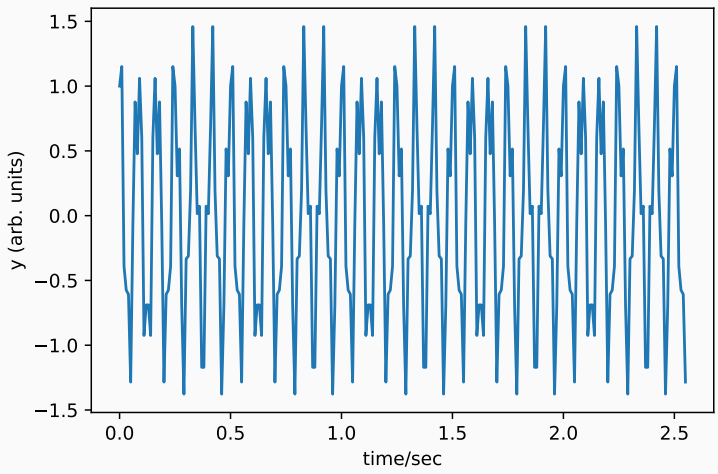
Ideas extend to more than 1-dimension, so that e.g. fast 2-D transforms are possible. Most common example is perhaps image processing by Fourier filtering.

FFT Applications

1. Convolution of two signals: FFT each, multiply the FFTs, then inverse FFT back. For example, smooth an image using a Gaussian kernel.
2. Filtering a signal – closely related to convolution. We take a signal, FFT it, multiply the FFT by a function, then FFT back, e.g. low- or high-pass filtering.
3. Crystallography
4. Find the power spectrum (PSD) $|H_n|^2$
5. Optics: Fraunhofer (and Fresnel) diffraction
6. Optics: the spatial/temporal coherence function is equal to the Fourier transform of the brightness distribution/power spectrum.
7. Signal processing. e.g. Freeview signals are transmitted using FFT: the data are cut into 8192-piece chunks, ($8192 = 2^{13}$), FFT'ed, transmitted, inverse FFT'ed on reception. In fact, FFTs are central algorithm in Digital Signal Processing (DSP)

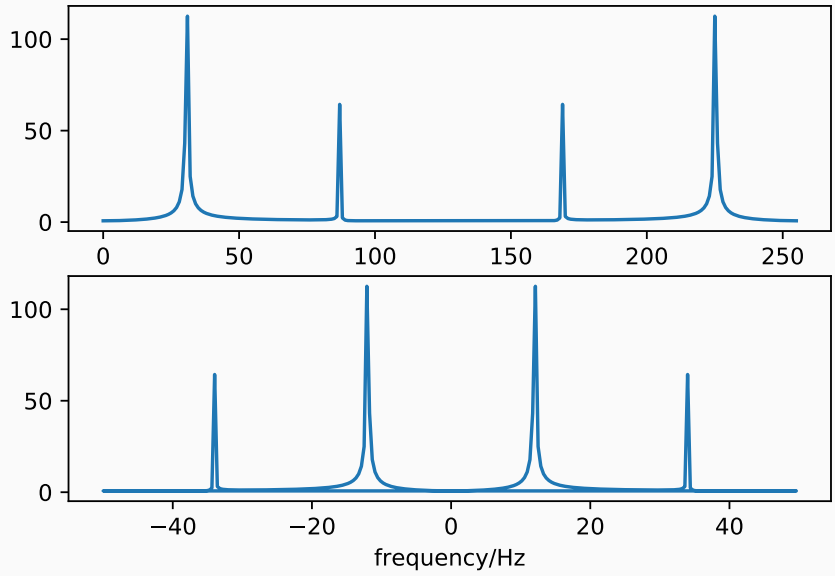
FFTs in Python

```
import numpy as np
import matplotlib.pyplot as plt
dt=0.01
fftsize=256
t=np.arange(fftsize)*dt
#Generate some fake data at 12 Hz and 34 Hz
y=np.cos(2*np.pi*12*t)+0.5*np.sin(2*np.pi*34*t)
plt.plot(t,y)
```



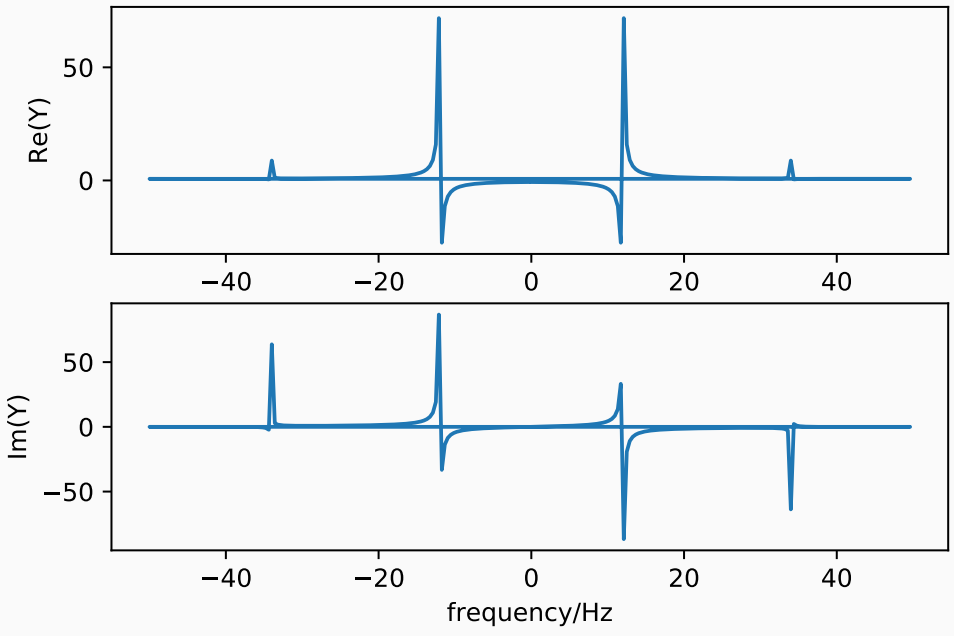
Negative frequencies come *after* positive frequencies

```
Y=np.fft.fft(y)
# Plot FFT modulus versus array index
plt.subplot(2,1,1); plt.plot(abs(Y))
# Now use the correct frequency coordinates
f=np.fft.fftfreq(fftsize,dt)
plt.subplot(2,1,2); plt.plot(f,abs(Y))
```



Negative frequencies are not always needed

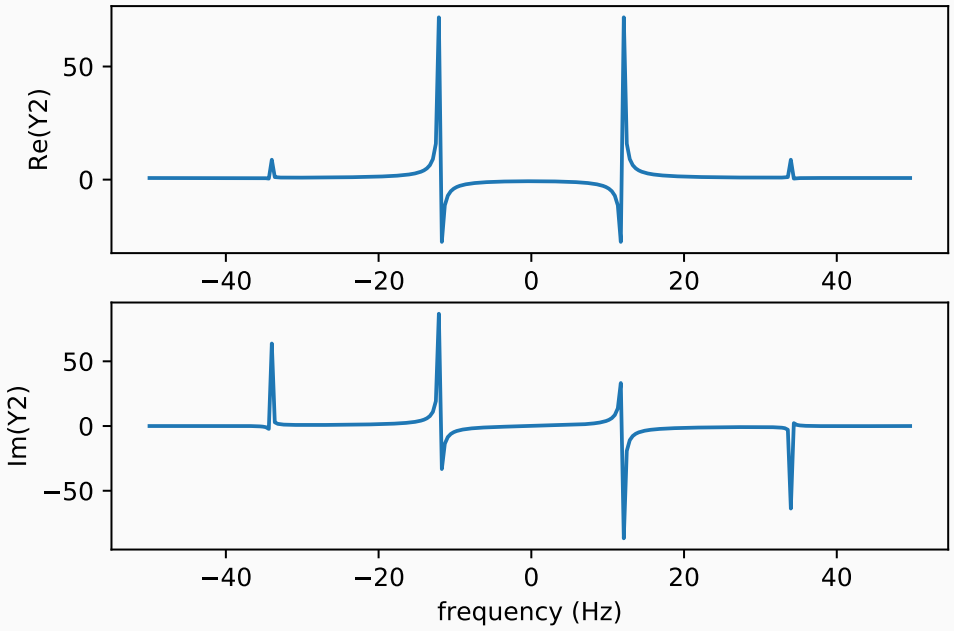
```
plt.subplot(2,1,1); plt.plot(f,Y.real)
plt.subplot(2,1,2); plt.plot(f,Y.imag)
```



Recall that the FT of a real signal has $H(f) = H^*(-f)$

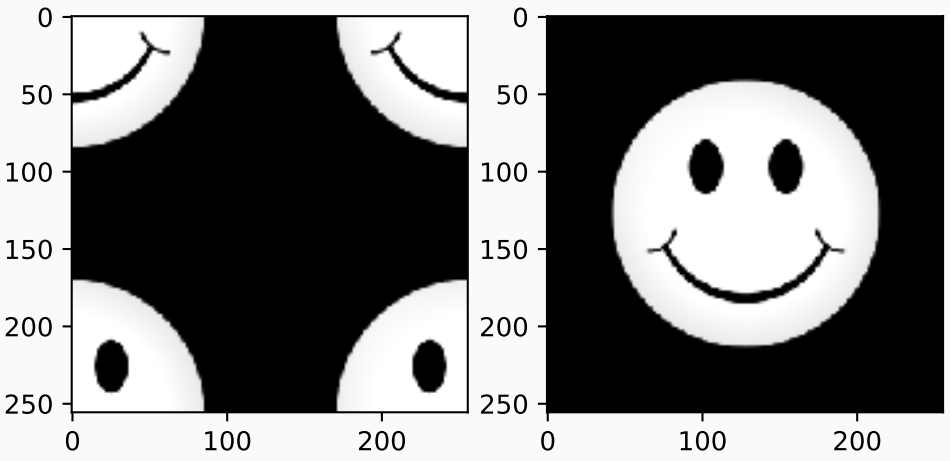
Re-ordering the array makes plots tidier

```
Y2=np.fft.fftshift(Y)
f2=np.fft.fftshift(f)
plt.subplot(2,1,1); plt.plot(f2,Y2.real)
plt.subplot(2,1,2); plt.plot(f2,Y2.imag)
```



In 2-d this helps to visualise Fourier results

```
plt.subplot(1,2,1)
plt.imshow(smiley,cmap="gray")
plt.subplot(1,2,2)
plt.imshow(np.fft.fftshift(smiley),cmap="gray")
```



Fast Fourier Transforms in C++

Key points

- Use a library. `FFTW` is good.
- Understand how the data are stored in memory.

Usually a complex number is just represented by two consecutive foating-point numbers, being the real and imaginary parts. Most libraries expect this! So if you want to FFT N numbers, just create an array of $2N$ values, and store the numbers sequentially as

$$[R_0, I_0, R_1, I_1, \ldots R_{N-1}, I_{N-1}]$$

where (R_i, I_i) are the real and imaginary parts of the $i'th$ number in the series.

Typical FFT Storage Order for $N = 8$

memory offset	type	time	freq
0	real	$t = 0$	$f = 0$
1	imag	$t = 0$	$f = 0$
2	real	$t = \Delta$	$f = 1/(8\Delta)$
3	imag	$t = \Delta$	$f = 1/(8\Delta)$
4	real	$t = 2\Delta$	$f = 2/(8\Delta)$
5	imag	$t = 2\Delta$	$f = 2/(8\Delta)$
6	real	$t = 3\Delta$	$f = 3/(8\Delta)$
7	imag	$t = 3\Delta$	$f = 3/(8\Delta)$
8	real	$t = 4\Delta$	$f = 4/(8\Delta) = 1/(2\Delta) = f_c$
9	imag	$t = 4\Delta$	$f = 4/(8\Delta) = 1/(2\Delta) = f_c$
10	real	$t = 5\Delta$	$f = -3/(8\Delta)$
11	imag	$t = 5\Delta$	$f = -3/(8\Delta)$
12	real	$t = 6\Delta$	$f = -2/(8\Delta)$
13	imag	$t = 6\Delta$	$f = -2/(8\Delta)$
14	real	$t = 7\Delta$	$f = -1/(8\Delta)$
15	imag	$t = 7\Delta$	$f = -1/(8\Delta)$

Simple FFT example

```
#include <vector>
#include <iostream>
#include <cmath>
#include <fftw3.h>
int main() {
  const int n = 8;
  std::vector<double> inp(2 * n, 0);
  std::vector<double> out(2 * n, 0);
  // Set the real parts to something quasi-random:
  inp[0] = -4; inp[2] = 0; inp[4] = -3; inp[6] = 6;
  inp[8] = -2; inp[10] = 9; inp[12] = -6; inp[14] = 5;
  // FFTW wants the addresses of the input and output arrays, but has
  // its own own complex type (actually a typedef). Use a cast:
  fftw_complex *finp = (fftw_complex *)&inp[0];
  fftw_complex *fout = (fftw_complex *)&out[0];

  fftw_plan plan_forward =
      fftw_plan_dft_1d(n, finp, fout, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute(plan_forward);
```

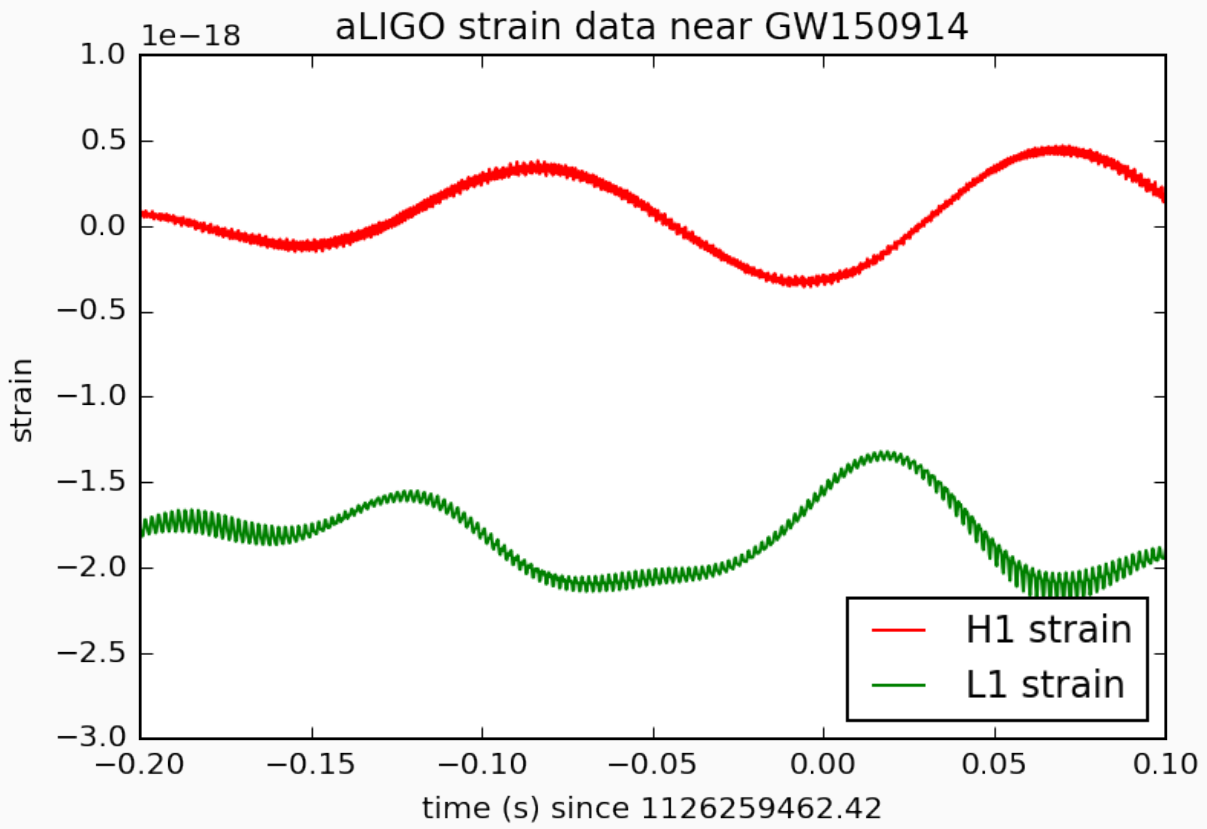

Simple FFT example (cont'd)

```
  std::cout.precision(3);
  for (int i = 0; i < n; i++)
    std::cout << "(" << inp[2 * i] << "," << inp[2 * i + 1] << ")  ("
              << out[2 * i] << ", " << out[2 * i + 1] << ")\n";
  return 0;
}
```

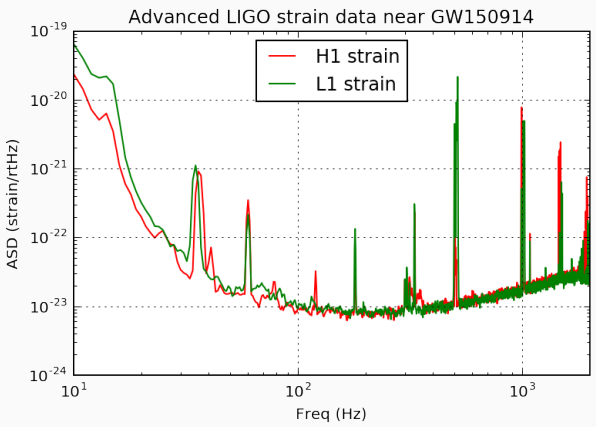
```
(-4,0)  (5, 0)
(0,0)  (-9.07, 2.66)
(-3,0)  (3, 2)
(6,0)  (5.07, 8.66)
(-2,0)  (-35, 0)
(9,0)  (5.07, -8.66)
(-6,0)  (3, -2)
(5,0)  (-9.07, -2.66)
```

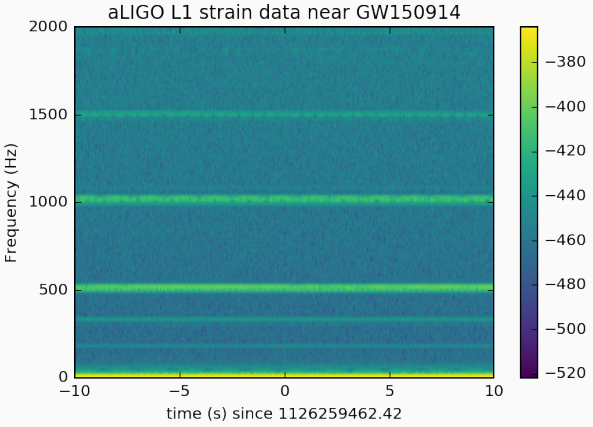
See `fft2.cc` and `fft2f.gp` in the examples on the MCS systems

Signal Processing Example - LIGO

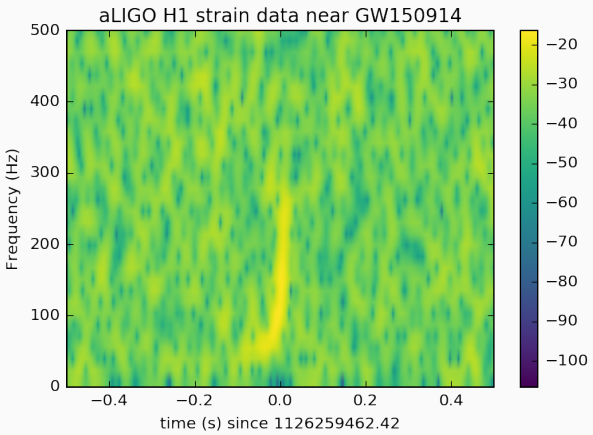


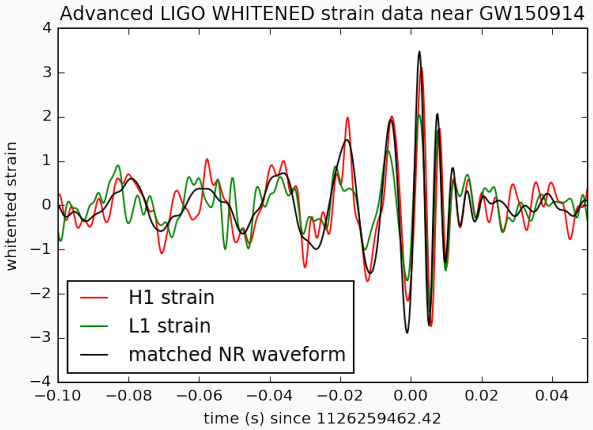
LIGO data in the Fourier domain





Fourier filtered data





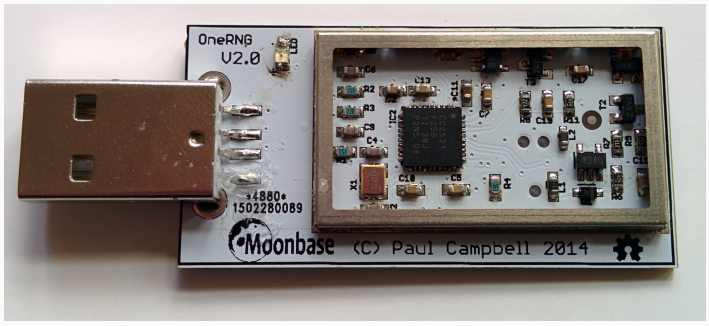
Random Numbers

The need for random numbers

We often need to generate random number sequences to simulate physical phenomena, e.g.

- make a decision based on a probability of an event e.g. simulate a random walk, pick random points/directions in space;
- generate simulated noise according to a known distribution.

True random number generators (TRNG) use analogue processes e.g. the noise of the microwave background radiation or a "noise diode" to generate sequences which are non-deterministic (hopefully).



Pseudo-Random Numbers

It is much faster and easier to use Pseudo Random Number Generators (PRNG) which create deterministic sequences of numbers. We need ideally a long sequence with no clustering of values and no short-range or long-range correlations from number to number.

A simple example is the linear congruential method, which generates a sequence of integers according to

$$I_{n+1} = (I_n * A + C) \mod M \tag{4}$$

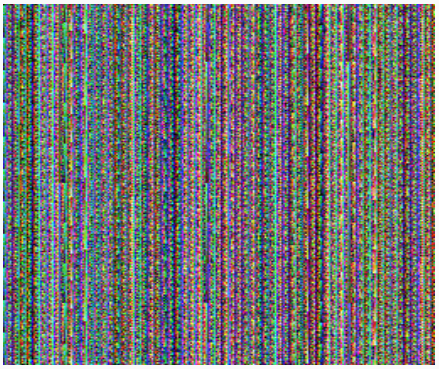
starting from a "seed" value I_0.

Good choices of C, A and M lead to sequences which have length up to M. Poor choices can lead to cycles which are shorter than M and/or correlations betweens numbers in a cycle.

Remember that the sequence is deterministic. Call a generator with the same seed and you get the same sequence each time.

Choosing a PRNG

There is a long and interesting history of PRNG algorithms, and a large technical iterature on the subject. It is easy to make a bad PRNG (and potentially dangerous to do so).

Bottom line: Make sure you use a well tested generator for any serious work.

Pseudo Random Numbers using GSL

The GSL algorithm has a period of $2^{19937} - 1$ (about 10^{6000}) and is equi-distributed in 623 dimensions.

```
#include <iostream>
#include "gsl/gsl_rng.h"
int main() {
  gsl_rng *rng = gsl_rng_alloc(gsl_rng_default);
  gsl_rng_set(rng, time(NULL)); // Different seed every time
  for (int i = 0; i < 5; i++)
    std::cout << gsl_rng_uniform(rng) << ", ";
  std::cout << "\n";
  gsl_rng_free(rng);
}
```

```
0.747422, 0.822479, 0.905515, 0.820809, 0.913495,
```

Setting the seed to a fixed value at the start will still give the same "random" sequence every time the program is run. However the repeat period is much longer than for the linear congruential method and long-range correlations between numbers have been stringently tested.

Random Numbers from Given Distributions

Often we want a random deviate y drawn from non-uniform distributions — Poisson, Gaussian, Gamma, ...— described by a probability density $p(y)$, normalised such that $\int p\,\mathrm{d}y = 1$.

Consider $y = f(x)$. Now, y will have a probability distribution such that $|p_y(y)\,\mathrm{d}y| = |p_x(x)\,\mathrm{d}x|$, i.e.

$$p_y(y) = p_x(x)\left|\frac{\mathrm{d}x}{\mathrm{d}y}\right|$$

For example, if $y = -\log(x)$ and the distribution of x is uniform, we have

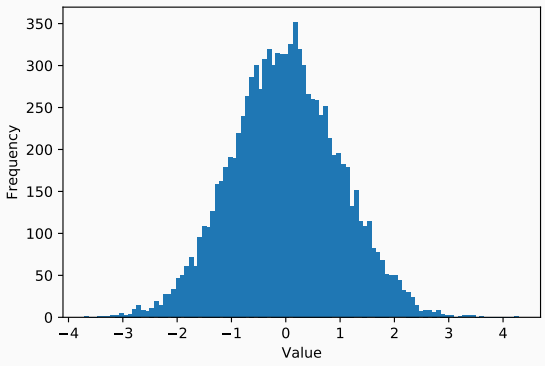
$$p_y(y) = \exp(-y)$$

i.e. y has an exponential distribution.

Other methods to generate arbitrary distributions are available, e.g. the "rejection method".

Example Python code

```
import numpy.random as random
import matplotlib.pyplot as plt
a=random.normal(size=10000)
plt.hist(a,bins=100)
```



Note that `numpy.random` initialises the seed to a non-repeatable state by default.

Routines exist also for Poisson, Binomial, etc distributions, and also in GSL. See example `prng3.cc` for an example.

Random Numbers: summary

- “Random number” generators make pseudo-random number sequences
- Same sequence for each seed — beware!
- Use a high-quality library for serious computation — don’t write your own
- GSL or `numpy.random` will probably provide all the generators you will ever need.

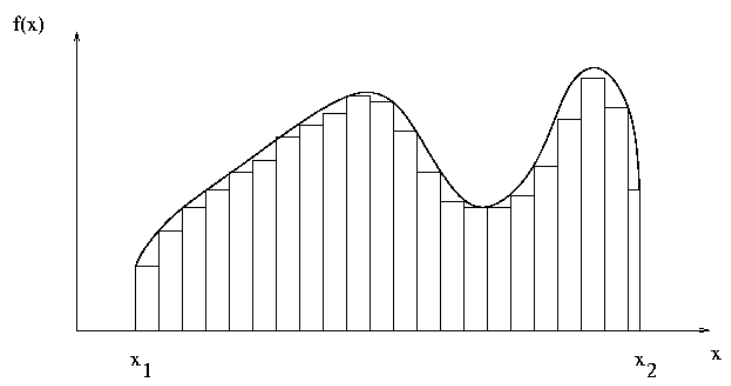
Numerical Integration

Numerical Integration

Many integrals are not analytic. A pendulum swinging with amplitude θ_m has $\ddot{\theta} = -(g/l)\sin(\theta)$. The period of the swing can be expressed as an elliptical integral

$$T = \sqrt{\frac{8l}{g}} \int_0^{\theta_m} \frac{d\theta}{\sqrt{\cos(\theta) - \cos(\theta_m)}}, \tag{5}$$

which must be evaluated numerically. The techniques required have much in common with those developed for ODEs.



- We sample systematically the function at N equally-spaced abcissae, each of which has width $h = (b - a)/N$. The techniques are intrinsically nested in that we can further subdivide each interval to get higher accuracy, and reuse the function evaluations already made.
- Often uses "Newton-Cotes" methods, evaluating the function at regular intervals. For example, sampling at the start of N panels:

$$\int_a^b f(x)\,dx \approx h \sum_{i=0}^{N-1} f(x_i) \tag{6}$$

- The error per panel is $\mathcal{O}(h^2)$, with $N \propto (1/h)$ panels, yielding a total error $\mathcal{O}(h)$ which is $\mathcal{O}(N^{-1})$. (c.f. Euler method for ODEs).

- As with ODEs, the best routines use higher-order methods. For an n'th order method (Trapezium Rule/Mid Point: $n = 2$, Simpson's Rule $n = 4$) the error is $\mathcal{O}(h^{n+1})$ per panel, yielding a total error is $\mathcal{O}(h^n)$ i.e. $\mathcal{O}(N^{-n})$.
  - So Simpson's Rule error scales like that in the Runge-Kutta 4th order method for ODEs
- There are very clever ways of choosing non-uniform abcissa spacings, called Gaussian quadrature, which can yield very high accuracy for many types of functions.
- GSL or `scipy.integrate.quad()` provides enough for your needs. Simply write an evaluator function and pass this to the integration function.
- Examples `quad1.cc` and `quad.py` are available in examples directory.
- Example class interface in `cavlib` header file `gslint_integ.hh`

Monte-Carlo Techniques

Very-high-dimensional integrals

- Conventional quadrature methods are most efficient in small numbers of dimensions. Higher-dimensional integrals are harder to evaluate accurately by these methods.
- In d dimensions, if we do a total of N function evaluations, we have $N^{1/d}$ panels in each dimension, yielding a total error of $\mathcal{O}(N^{-n/d})$ for an n'th order method. Using Simpson's rule, $n = 4$, and the error is $\mathcal{O}(N^{-4/d})$. As d increases, the number of evaluations N required for a given error increases rapidly.
- We often meet multi-dimensional integrals in physics, e.g. partition function of m particles with potential $\phi(r_{ij})$,

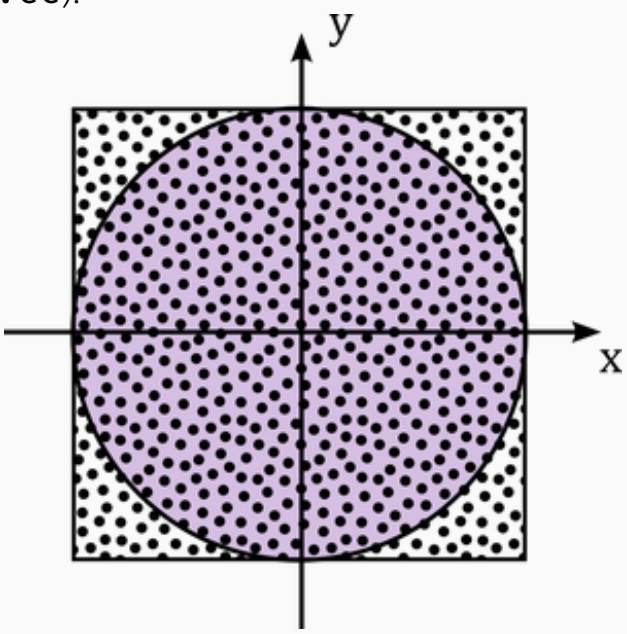
$$Z = \int \cdots \int \exp\left[-\sum_{\text{pairs}} \frac{\phi(r_{ij})}{k_B T}\right] d^3 r_0 \cdots d^3 r_{m-1},$$

  is a $3m$-dimensional integral. If we do 10 evaluations in each dimension we need to make 10^{3m} function evaluations. And the error is $\mathcal{O}(N^{-n/(3m)})$ which falls rather slowly with N.

- Such integrals also arise when doing parameter estimation where we want to integrate over ("marginalise") a likelihood or probability function of many model parameters.
- We might not need a very accurate result.
- Monte-Carlo integration is a simple to code, robust technique which generalises easily to many dimensions.

Integration using random numbers

Recall the estimation of π by choosing N random points in a unit square. We expect $\pi/4 = M/N$ where M points lie inside the unit circle. (Coded up in the example `pi.cc`).



Monte Carlo Integration

In Monte-Carlo integration we use the fact that the integral is equal to the mean function value times the (hyper)volume of the integral.

Rather than systematically evaluating the integrand throughout the volume as in classical quadrature techniques, we can make an estimate of the average by choosing random points throughout the hypervolume:

$$Z = \left\langle \exp\left(-\sum_{\text{pairs}} \frac{\phi(r_{ij})}{k_B T} \right) \right\rangle V^n$$

where V is the three-dimensional volume in this example.

Note that in the estimation of π, we implicitly used an integrand which is unity inside the circle and zero outside.

Simple Monte-Carlo Integration Formula

To integrate a function f of n parameters (i.e. an n-dimensional vector $\mathbf{r}$) over a multi-dimensional hypervolume V, we take N samples distributed at random points $\mathbf{r}_i$ throughout the hypervolume V, and use:

$$\int f \mathrm{d}V \approx V \langle f \rangle \pm V \left(\frac{\langle f^2 \rangle - \langle f \rangle^2}{N} \right)^{1/2}$$

with

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=0}^{N-1} f(\mathbf{r}_i)$$

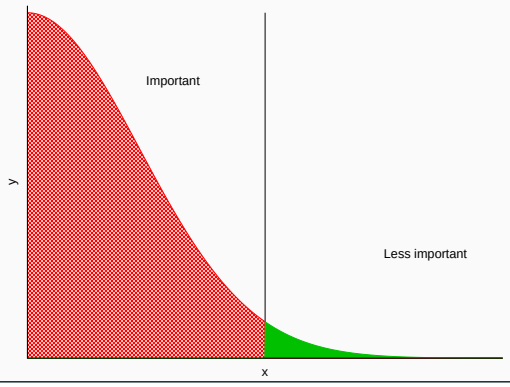
$$\langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=0}^{N-1} f^2(\mathbf{r}_i)$$

Note that this error estimate is not guaranteed to be very good, and is not Gaussian-distributed: treat it as indicative only of the error.

- This is an example where we need extremely good random numbers. We may create many million sample points in the hypervolume V: if there is any clustering along hyperplanes, we will get a biased sampling of the integral and the wrong result.
- Note that the error goes as $N^{-1/2}$ independent of the number of dimensions, so that this method can beat Newton-Cotes quadrature methods in higher dimensions e.g. $d = 8$ and $n = 4$ yields the same $N^{-1/2}$ dependence. For higher d, Monte-Carlo is more efficient.

Reducing Errors with Importance Sampling

- As long as our PRNG is good, our result should converge to the right answer with the error falling off as $N^{-0.5}$
- But convergence can be improved by taking more samples where the integrand's value is larger. Sampling lots of points where the integrand is small is wasteful. (Think about estimating π by placing balls randomly onto a football pitch and counting the ones in the centre circle: it works but it is slow as most balls miss the circle).
- This is generally known as importance sampling, or sometimes variance reduction in the case of MC integration.



Importance Sampling in Monte-Carlo Integration

We don't generally know *a priori* where the integral has a large value, so the algorithm needs to learn about the function as it progresses.

Example Algorithm

1. Divide integration region into 2 disjoint subregions.
2. Use simple MC technique with a small number of samples to estimate value of the integral in each subregion and the estimated error.
3. Use these estimates to decide how to divide optimally the future samples between the two volumes — we will concentrate on the one where the integral has a higher value.
4. Break the volumes up once more, now into 4 subvolumes, and use recursion to break the problem into ever smaller regions, doing most evaluations where the integrand is large.

Monte-Carlo Integration in Practice

- It is easy and instructive to code your own simple Monte-Carlo method (Exercise 1). You can also investigate the error on your estimate by repeating the evaluation n_t (~ 20?) times using different random points, and estimating the sample variance of these n_t estimates.
- The GSL provides functions for both simple MC and two importance sampling methods, called `gsl_monte_plain_integrate`, `gsl_monte_miser_integrate`, and `gsl_monte_vegas_integrate`.
- `scipy` does not provide an importance sampling algorithm, but there are open-source Python packages to do so.

The Ising Model of Ferromagnetism

An example of real physics insight gained using Monte-Carlo methods.

The Problem

- Ferromagnetism is an intrinsically many-body quantum phenomenon: electron spins align creating a macroscopic dipole moment.
- The magnetisation disappears above some critical temperature.
- How do the spin-spin interactions cause this effect?
- Can we predict the magnetisation **M**, energy E, heat capacity C, and and magnetic susceptibility χ as a function of temperature?

In the Ising model, it is assumed that only adjacent spins interact, and we write the energy of the system as

$$E = -J \sum_{<ij>} s_i s_j - \mu H \sum_{i=1}^{M} s_i \tag{7}$$

where the sum $< ij >$ is over nearest neighbour pairs of atoms only.

We are assuming a two-dimensional array of M spins ($M = N \times N$), so that each spin has 4 nearest neighbours.

J is the exchange energy, μ the magnetic moment, and H the applied magnetic field. We use spin values $s_i = \pm 1$ (could use 1/2 by rescaling J).

More elaborate models treat interactions in more detail (next-nearest neightbours etc) but simple model captures the essence of the behaviour.

Ising Model

- The energy is negative for aligned spins: tendency for alignment to either $s_i = +1$ or $s_i = -1$ for all spins
- But when in contact with a heat bath at temperature T, individual spins will flip back and forth, exchanging energy with the bath.
- With M spins in total there are 2^M microstates which the system can explore. For $M = 32 \times 32 = 2^{10}$ say, we have $2^{2^{10}} \approx 10^{308}$ microstates to explore.
- They are all equally likely but there are too many to compute.
- Onsager developed an analytic solution to the Ising model in 2D by evaluating the partition function directly, but generally analytic techniques are intractable. Need to simulate, but how?

The Metropolis Algorithm

We can't compute all the microstates — there are too many! But the probability P of finding a system in a state with energy E should be proportional to the Boltzmann factor

$$P \propto \exp\left(-\frac{E}{k_B T}\right) \tag{8}$$

We want our simulation to generate a sample of the microstates with this probability distribution. We can then derive thermodynamic quantities by a simple average over the microstates visited by the simulation. Metropolis et al (1953) invented just such an algorithm (N.C. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller, J. Chem. Phys. 21 (1953) 1087-1092)

"Instead of choosing configurations randomly, then weighting them with exp(-E/kT), we choose configurations with a probability exp(-E/kT) and weight them evenly."

Metropolis Algorithm (aka Metropolis-Hastings algorithm)

- Given a system in a known microstate A with energy E_A, consider a nearby microstate B of energy E_B.
  - if $E_B < E_A$, allow the system to make a transition to the new microstate (because it has lower energy);
  - else if $E_B > E_A$, either make the jump with a probability $p = \exp(-(E_A - E_B)/(k_B T))$

So as the temperature increases, the system can explore more distant (higher energy) states.

The algorithm works (that is, is thermodynamically sensible) because it encodes the principle of detailed balance between states when thermal equilibrium applies: $p_A T_{AB} = p_B T_{BA}$ where T_{AB} is the transition rate $A \to B$.

$$\frac{p_A}{p_B} = \frac{T_{BA}}{T_{AB}} = \frac{\exp(-E_A/(k_B T))}{\exp(-E_B/(k_B T))} \tag{9}$$

This is another example of importance sampling.

Ising Model using Metropolis Algorithm

1. Create a system of spins in a particular microstate.
2. Repeat a large number of sweeps through the lattice. For each sweep (a single Monte-Carlo "step"):
   - Select a lattice site, either randomly or in a sequence
   - Find the energy ΔE required to flip the spin at the given lattice point using Equation 7
     - if $\Delta E < 0$, flip the spin;
     - else if $\exp(-\Delta E/(k_B T)) > p$, where p is a uniform random number in the range $[0, 1]$, then flip the spin;
     - else do nothing.
   - Repeat for another lattice site, until you have visited all the sites in turn, or visited a large fraction of the total number of sites.
   - Record relevant variables (energy, magnetisation,...), plot, save state.

Some technicalities

1. Spins on the lattice edges don't have 4 neighbours, and this will introduce artefacts. It is easiest to use periodic boundary conditions, so that every spin has 4 neighbours: imagine the lattice on a torus. (This is easy to implement by carefully looking after the indexing of the array used to store the spins.)
2. You need to let the simulation run for a 'long time' to ensure equilibrium is approached. How long will depend on the initial set up.
3. Visiting lattice sites randomly or in a regular sequence will both work, but the time to approach equilibrium will vary again depending on inital conditions and temperature.
4. The compute time varies steeply with N, but desktop PCs should be able to handle $N \sim 10 \times 10$ and somewhat larger in reasonable time. Memory requirements are small as long as the data are not stored.
5. Numerical results will depend on N; should investigate how...
6. Easy to generalise to higher dimensions, or add more interactions.

Other Applications using Monte-Carlo Methods

1. Estimating the error on parameters determined by experiment.
2. Solving complex systems of differential equations. For example, radiative transfer codes in 3-dimensions.
3. Random walks e.g. polymer folding
4. “Travelling Salesman”/simulated annealing problems

Many others...