

# Lec1 - intro, basic tools

- Mechanics, syllabus
- General comments
- What is computational science ?
- Derivatives, integrals and root finding
- Intro to Python

# General comments

- Not programming course. Not traditional physics course.
- Topics drawn from many different areas of physics including recent developments eg. self-organized critical phenomena, black-hole orbits, fractals
- Brief discussion of programming issues - *only as much as needed to do the science*
- Python/VPython makes life easy. Don't need to know much to do some quite complicated stuff (graphics built-in)

# Computational Science

2 main facets:

- Provide set of tools (algorithms, efficient code) to produce (numerical) *quantitative* solutions to known scientific laws (expressed in math form). Typically small number of degrees of freedom Essentially zero error.
- Simulate a complex model of the system. Extract *qualitative* as well as quantitative information. Often statistical errors - computer experiment. Sometimes coming up with the model is part of the game. Use to explore different possible theories. Again good tools, algorithms needed.

# Examples

Type 1: throwing a ball into the air - time to reach ground ?

a) Newton's laws, analytic solution (neglect air resistance, model as point particle)

b) Real world: put in air resistance, spin of ball, can *integrate the equations numerically* with high accuracy.

Type 2: Show that water freezes when cooled.

Features:

- Many d.o.f, complex interactions
- Qualitative change of state of system.

Errors: finite simulation time. Finite number of particles.

# Derivatives

Want a numerical approx to  $\frac{df}{dx}$

Why ? Perhaps derivative too hard to compute analytically. Or perhaps  $f(x)$  produced by some other procedure at only a finite set of points.

How ? Taylor expand:

$$f(x + h) = f(x) + f^1(x)h + f^2(x)h^2/2 + \dots$$

Many possibilities:

$$f^1 \sim \frac{f(x + h) - f(x)}{h} + O(h)$$

or

$$f^1 \sim \frac{f(x + h) - f(x - h)}{2h} + O(h^2)$$

Improve ?

$$f^1 \sim \frac{f_{-2} - f_2 - 8f_{-1} + 8f_1}{12h} + O(h^4)$$

## Pros/Cons

Why not use high order approx ?

Formulae assume  $f(x)$  approx well by polynomial at that point.

Polynomial approx is notoriously unreliable at high orders.

Can produce examples where answer *gets worse* with high order.

Also more calculation required, more points needed.

## Python code

eg.  $df/dx$  for  $f = x^2$  at  $x = 1$

```
from math import *  
x=1.0  
h=0.1  
print (exp(x+h)-exp(x-h))/(2*h)
```

## Improvements

Rather than type in the code from scratch each time I want to change  $h, x$  can create function

```
from math import *  
def deriv(x,h):  
    print (exp(x+h)-exp(x-h))/(2*h)
```

Execute: eg `deriv(1,0.1)`, `deriv(1,0.01)` etc

# Integration

Eg.  $\int_a^b f(x)dx$ :

Split up integral into number of small slices width  $2h$

$$\int_a^b = \int_a^{a+2h} + \int_{a+2h}^{a+4h} + \dots + \int_{b-2h}^b$$

Approx  $f$  by some polynomial in the interval  $2h$

Eg. Linear approxs from  $c \rightarrow c + h$

$$\int_c^{c+h} (f_c + x/h(f_{c+h} - f_c)) = \frac{h}{2}(f_c + f_{c+h})$$

Thus over  $2h$  interval:

$$\frac{h}{2}(f_c + 2f_{c+h} + f_{c+2h})$$

Trapezoidal rule:

$$\int_a^b f(x)dx \sim \frac{h}{2}(f(a) + 2f(a+h) + \dots + f(b))$$

## Better

Use quadratic approx to  $f$  in small interval

$$f(x) = f_0 + \frac{(f_1 - f_{-1})}{2h}x + \frac{f_1 - 2f_0 + f_{-1}}{2h^2}x^2 + \dots$$

$$\int_{-h}^h f(x)dx \sim \frac{h}{3}(f_1 + 4f_0 + f_{-1})$$

Leading to Simpson's rule:

$$\begin{aligned} \int_a^b f dx &= \frac{h}{3} (f(a) + 4f(a+h) + 2f(a+2h) + \dots \\ &+ \dots 4f(b-h) + f(b)) \end{aligned}$$

# Python code

Eg. integrate  $e^x$  over  $0 \rightarrow 1$

```
def int(a,b,h):  
    x=a  
    integral=exp(a)*h/2.0  
    while x<(b-h):  
        x=x+h  
        integral=integral+h*exp(x)  
  
    integral=integral+exp(b)*h/2.0  
    print integral
```

- Note indentation - very important in Python.
- Note while loop.
- call: `int(0,1,0.1)`
- Place in separate file - called say `int.py`

## Root finding - bisection

Solve  $f(x) = 0$  for  $a < x < b$

- Suppose root in interval. Means  $f(a) * f(b) < 0$
- Compute midpoint  $m = (a + b)/2$ .
- If  $f(a) * f(m) > 0$  root in  $m < a < b$ .
- Else  $f(b) * f(m) > 0$  and root in  $a < x < m$ .
- Root now located in interval of  $1/2$  size
- Keep going until root located in small region

Robust but slow.

## Python code

Eg. solve  $e^x - 0.5 = 0$  in range  $-1 \rightarrow 0$

```
def root(a,b,tol):
    l=a
    u=b
    fl=exp(a)-0.5
    fu=exp(b)-0.5
    while (u-l) > tol:
        m=(u+l)/2.0
        fm=exp(m)-0.5
        if(fm*fu>0.0):
            u=m
            fu=fm
        else:
            l=m
            fl=fm

    print m
```

# Comments

- Note `if/else` structure
- `if` statement nested within `while` loop. Indentation tells you which bits of code are associated with which control structures
- call with: `root(0,1,0.01)`

# Newton-Raphson

Faster convergence if can evaluate derivative:

$$f(x + h) = f(x) + hf^1(x) \sim 0$$

$$x_{i+1} = x_i - f(x_i)/f^1(x_i)$$

- Bisect then Newton-Raphson good compromise
- Replace  $f^1$  by discrete approx yields **secant alg**

# Summary

- General comments.
- Workhorse alg. for derivatives, integrals and roots
- Simple Python codes for doing this
- Simple assignment, arithmetic, I/O, `while` loops and `if` statements. Indentation
- Function definitions