COMPUTATIONAL PHYSICS

Numerical methods

Numerical derivatives

Numerical integration

Roots of equations

COMPUTATIONAL PHYSICS

Numerical methods

Numerical derivatives

- ✓ *First order derivatives*
- ✓ *Second order derivatives*
- ✓ *Spline interpolation derivatives*

Numerical derivatives - Introduction

- In the beginning there was the *Taylor series*…
- *Calculus* gives us the Taylor expansion of a differentiable function:

$$f(x) \cong f(x_i) + f'(x_i)(x - x_i) + \frac{f''(x_i)}{2!}(x - x_i)^2 + \cdots + \frac{f^{(n)}(x_i)}{n!}(x - x_i)^n$$

or

$$\boxed{f(x_i + h) \cong f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \cdots + \frac{f^{(n)}(x_i)}{n!}h^n}$$

But also

$$\boxed{f(x_i - h) \cong f(x_i) - f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \cdots + (-1)^n \frac{f^{(n)}(x_i)}{n!}h^n}$$

And more…later !

Numerical derivatives – 1st derivative

- Different finite differences can be derived, of different order of accuracy !

forward differences

$$f(x_i+h)-f(x_i) \cong f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \cdots \quad \longrightarrow \quad \boxed{f'(x_i) = \frac{f(x_i+h)-f(x_i)}{h} + O(h)}$$

backward differences

$$f(x_i-h)-f(x_i) \cong -f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \cdots \quad \longrightarrow \quad \boxed{f'(x_i) = \frac{f(x_i)-f(x_i-h)}{h} + O(h)}$$

central differences (subtract the 2 above !)

$$f(x_i+h)-f(x_i-h) \cong 2f'(x_i)h + \frac{2f'''(x_i)}{3!}h^3 + \cdots$$

$$\boxed{f'(x_i) = \frac{f(x_i+h)-f(x_i-h)}{2h} + O(h^2)}$$

…now 2nd order accurate !

- ❑ How small ***h*** is too small ?
- ❑ Recall subtractive cancellation ?

$$f(x_i+h)-f(x_i-h) \cong f(x_i)\varepsilon_M \sim \frac{2f'''(x_i)}{3!}h^3$$

➔ $\boxed{h \sim (\varepsilon_M)^{1/3}}$ $\approx 10^{-5}$ for double precision

Numerical derivatives – 1st derivative (cont.)

- To get higher order accuracy, "*stencils*" need to stretch away from x_0

$$f(x_i \pm h) \cong f(x_i) \pm f'(x_i)h + \frac{f''(x_i)}{2!}h^2 \pm \frac{f'''(x_i)}{3!}h^3 + \cdots + \frac{f^{(n)}(x_i)}{n!}h^n$$

$$f(x_i \pm 2h) \cong f(x_i) \pm f'(x_i)2h + 4\frac{f''(x_i)}{2!}h^2 \pm 8\frac{f'''(x_i)}{3!}h^3 + \cdots + 2^n\frac{f^{(n)}(x_i)}{n!}h^n$$

A suitable combination yields $f'(x_0)$ with error order $O(h^3)$.
Tip: eliminate $f''(x_0)$ using $f(x_0 \pm h)$ and then also $f(x_0 \pm 2h)$

$$\boxed{f'(x_i) = \frac{1}{12h}\left[\left(f(x_i - 2h) + 8f(x_i + h)\right) - \left(8f(x_i - h) + f(x_i + 2h)\right)\right] + O(h^4)}$$

- ***Caveat***: this formula (*and the central differences*) can't be used at the extrema of the dataset → *forward* and *backward* schemes needed !

Higher order forward/backward schemes

- As before, larger stencils and suitable combinations to the rescue…

$$\boxed{f'(x_i) = \frac{1}{2h}\left[4f(x_i+h) - \left(f(x_i+2h) + 3f(x_i)\right)\right] + O(h^2)}$$

Forward differences

$$\boxed{f'(x_i) = \frac{1}{2h}\left[f(x_i-2h) - 4f(x_i-h) + 3f(x_i)\right] + O(h^2)}$$

Backward differences

Example: a projectile is launched vertically from $h(t=0)=h_0$ with velocity $v(t)$. Obtain $h(t)$.

$$v(t_i) = \frac{x(t_i+h) - x(t_i-h)}{2h} + O(h^2) \quad \text{so} \quad x(t_i+h) - x(t_i-h) = 2hv(t_i) \quad \text{with} \quad h = \Delta t$$

→ At i=0, $x(t_i\text{-}h)$ is meaningless since i know at most $x(t_0)$! So, use it as boundary condition !
→ At i=N, $x(t_i\text{+}h)$ remains undetermined since i have less equations than variables ! Use backward differences instead.

$$x_2 - h_0 = 2hv_1$$

$$x_{N-2} - 4x_{N-1} + 3x_N = 2hv_N$$

$$x_{i+1} - x_{i-1} = 2hv_i \quad i = 2,..,N-1$$

✓ *N equations*
✓ *N unknowns*

Non-uniform dataset finite differences

- If the dataset $x_0,x_1,\ldots,x_N$ is unevenly spaced, ***h*** is multivalued !
- Consider $h_i=x_{i+1}-x_i$

$$f(x_i+h_i)=f(x_i)+f'(x_i)h_i+\frac{f''(x_i)}{2!}h_i^2+\ldots \qquad f(x_i-h_{i-1})\cong f(x_i)-f'(x_i)h_{i-1}+\frac{f''(x_i)}{2!}h_{i-1}^2+\ldots$$

As before, eliminate the f''(x_i) term by suitable combination of $f(x_i+h_i)$ and $f(x_i-h_{i-1})$

$$\boxed{f'(x_i)=\frac{h_{i-1}^2 f_{i+1}+\left(h_{i-1}^2-h_i^2\right)f_i-h_i^2 f_{i-1}}{h_{i-1}h_i\left(h_{i-1}+h_i\right)}+O(h^2)}$$

- ***Check***: you can easily verify that the central difference expression is recovered for constant h_i…
- *Question*: can we do any better ? After all, *stencil* above doesn't care about whether the derivative is "regular" *(continuous....) or not...*

Numerical derivative by interpolation

- Simple:
  1. Pick your dataset (x_i,y_i) and cubic spline it (*obtain the curvature coefficients*)
  2. Use curvature coefficients to obtain the “analytical” expression for the derivative !
  3. Done.

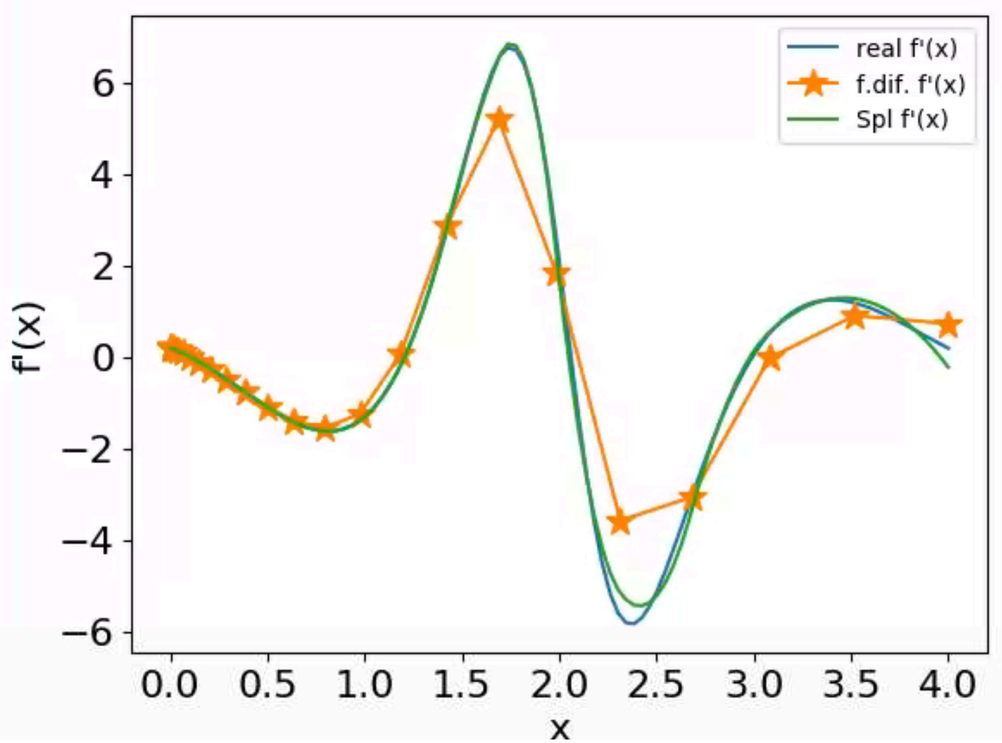
$$C_{i,i+1}(x) = \frac{K_i}{6}\left[\frac{(x-x_{i+1})^3}{x_i - x_{i+1}} - (x-x_{i+1})(x_i - x_{i+1})\right] - \frac{K_{i+1}}{6}\left[\frac{(x-x_i)^3}{x_i - x_{i+1}} - (x-x_i)(x_i - x_{i+1})\right] + \frac{y_1(x-x_{i+1}) - y_{i+1}(x-x_i)}{x_i - x_{i+1}}$$

$$C'_{i,i+1}(x) = \frac{K_i}{6}\left[\frac{3(x-x_{i+1})^2}{x_i - x_{i+1}} - (x_i - x_{i+1})\right] - \frac{K_{i+1}}{6}\left[\frac{3(x-x_i)^2}{x_i - x_{i+1}} - (x_i - x_{i+1})\right] + \frac{y_1 - y_{i+1}}{x_i - x_{i+1}}$$

and as a bonus …

$$C''_{i,i+1}(x) = K_i \frac{x - x_{i+1}}{x_i - x_{i+1}} - K_{i+1}\frac{x - x_i}{x_i - x_{i+1}}$$

Finite differences *vs* spline derivative



Test case

$$f(x) = \frac{\cos(3x)}{0.4 + (x-2)^2}$$

- Irregular x-grid proves challenging for FD.
- Cubic splines ***clearly better*** but unsurprising → spline is $O(h^4)$ for the function and $O(h^3)$ for the derivative

Side note:

- Cubic splines yield coefficients (*curvatures*) for a functional expansion of f(x) over data "*segments*" → problems involving determining f(x) e.g. differential equations, translates into solving for the curvature coefficients rather than f(x) itself on a grid !
- Fundamental concept for **F**inite **E**lement **M**ethods

Finite differences *: 2nd order derivative*

- Same principle as for 1st order derivative i.e. use $f(x_i+h_i)$ and $f(x_i-h_{i-1})$

$$f(x_i \pm h) \cong f(x_i) \pm f'(x_i)h + \frac{f''(x_i)}{2!}h^2 \pm \frac{f'''(x_i)}{3!}h^3 + \cdots + \frac{f^{(n)}(x_i)}{n!}h^n$$

  - Eliminate odd-order derivatives adding $f(x_i+h_i)$ and $f(x_i-h_{i-1})$!

$$\boxed{f''(x_i) = \frac{f(x_i+h) - 2f(x_i) + f(x_i-h)}{h^2} + O(h^2)}$$

  - Higher accuracy using larger stencil e.g. 5 points also exist

$$\boxed{f''(x_i) = \frac{-f(x_i+2h) + 16f(x_i+h) - 30f(x_i) + 16f(x_i-h) - f(x_i-2h)}{12h^2} + O(h^4)}$$

--❖--

Applications: so many second order derivative equations e.g. **Poisson equation** !

$$\boxed{-\phi''(x_i) = \rho_{charge}(x)/\varepsilon}$$

COMPUTATIONAL PHYSICS

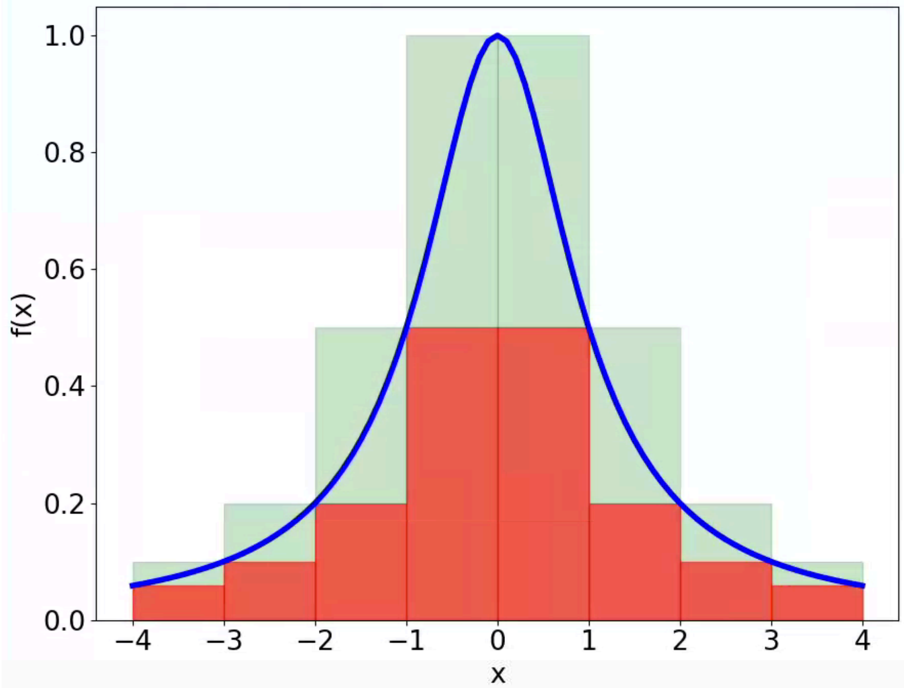
Numerical methods

Numerical integration

- ✓ *Trapezoidal*
- ✓ *Simpson*
- ✓ *Gaussian quadrature*

Numerical integrations - Introduction

- In the beginning there were the *Darboux sums*…



$$F = \int_a^b f(x)dx$$

Upper Darboux sum

$$S_\tau = \sum_i \sup_{x\in[x_i,x_{i+1}]} [f(x)](x_{i+1} - x_i)$$

Lower Darboux sum

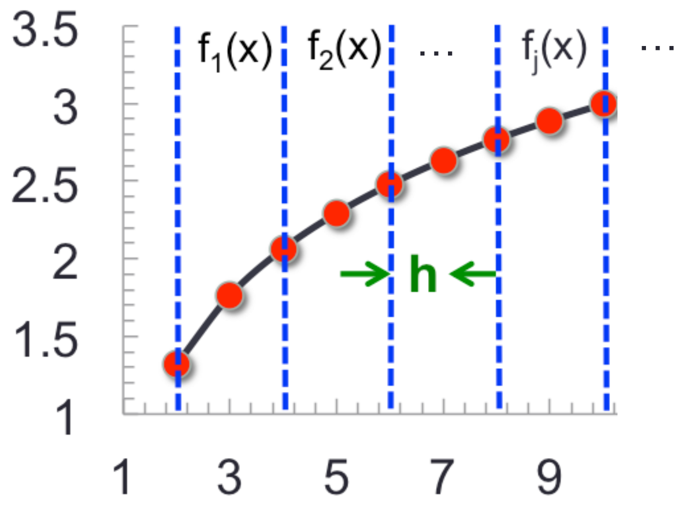
$$s_\tau = \sum_i \inf_{x\in[x_i,x_{i+1}]} [f(x)](x_{i+1} - x_i)$$

- We can undoubtledly be smarter:
  - ✓ We can locally ***interpolate*** the sampled f(x) and integrate those analytical expressions !
  - ✓ This corresponds to the **Trapezoidal** (*linear*) and **Simpson** (*quadratic*) schemes.
  - ✓ This Newton-Cotes formulation is particularly useful whenever interpolation behaves. 

Numerical integration: *Newton-Cotes basics*

- Divide the integration range *[a,b]* into equally spaced **n**-segments with nodes x_i (i=0,...,n) and spacing *h=(b-a)/n*.
- Approximate the tabulated function by polynomials (*interpolate*) using consecutive **k**-points clusters ($N_{clusters}$) and integrate:

$$\boxed{f_j(x) = \sum_{i=0}^{k} f(x_{j*k+i})\mathcal{L}_i^{(j)}(x)}$$



- The integral over [a,b] is expressed as:

$$\boxed{F = \int_a^b f(x)dx = \sum_{j=0}^{N_{clusters}} \int_{Cluster(j)} f_j(x)dx = \sum_{i=0}^{n} f(x_i)w_i}$$

- If we take clusters of 2 consecutive points → Trapezoidal rule
- If we take cluster of 3 consecutive points → Simpson rule

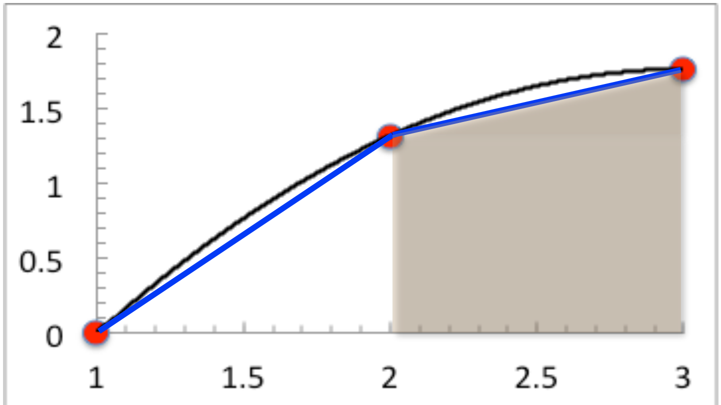
Numerical integration : *Trapezoidal rule*

- Linear interpolation of the sampled function → degree-1 polynomial
- On each integration element one has:

$$dF = \int_{x_i}^{x_{i+1}} f(x)\,dx = \int_{x_i}^{x_{i+1}} f(x_i)\frac{x - x_{i+1}}{x_i - x_{i+1}} + f(x_{i+1})\frac{x - x_i}{x_{i+1} - x_i}\,dx = \sum_{j=0}^{1} f(x_{i+j})w_j$$

- Integrating the polynomial basis is immediate and one obtains:

$$dF = \frac{h}{2}\left[f(x_i) + f(x_{i+1})\right]$$



- Adding up all elements one gets:

$$F = \sum_{i=0}^{n-1}\frac{h}{2}\left[f(x_i) + f(x_{i+1})\right] = \frac{h}{2}\left[f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-11}) + f(x_n)\right]$$

Trapezoidal rule error

- Upper estimate on Lagrange interpolation error $\varepsilon_{\mathcal{L}_i}(\mathrm{x}) \leq \frac{\left|\mathrm{f}^{(n+1)}(\xi(x))\right|}{(n+1)!}\left(x_n - x_0\right)^{n+1}$

  but we will use the actual formula

$$\varepsilon_{\mathcal{L}_i}(\mathrm{x}) = \frac{\left|\mathrm{f}^{(n+1)}(\xi(x))\right|}{(n+1)!}\prod_{i=1}^{n+1}\left(x - x_i\right)$$

- Integration over an interval $[x_i, x_{i+1}]$:

$$\boxed{\delta F_i = \frac{1}{(n+1)!}\int_{x_i}^{x_{i+1}} f''(\xi(x))(x-x_i)(x-x_{i+1})\,dx \cong \frac{f''(\xi)}{2!}\frac{(x-x_{i+1})^3}{6} = -\frac{h^3}{12}f''(\xi)}$$

- Over the whole interval [a,b] one has a n-sum to do and ***n=(b-a)/h***

$$\boxed{\delta F = -\frac{h^2}{12}(b-a)\left\langle f''(\xi)\right\rangle_{[a,b]}}$$

=> Higher accuracy ? *Increase n* !

Calculus time again: can you estimate the *average* of f''(x) over [a,b] ?!

Trapezoidal rule: variants

- Just calculated $F = \int_a^b f(x)dx$ and not satisfied by the large error ?...
  - *Simple answer :* double the number of *"slices"* of our partion i.e. 2^{k-1} with k=1,2,3,…

Recursive Trapezoidal formula

$$F_k = \frac{1}{2}F_{k-1} + \frac{b-a}{2^{k-1}}\sum_{i=1}^{2^{k-2}} f\left(a + (2i-1)\frac{b-a}{2^{k-1}}\right)$$

where $h = \dfrac{b-a}{2^{k-1}}$

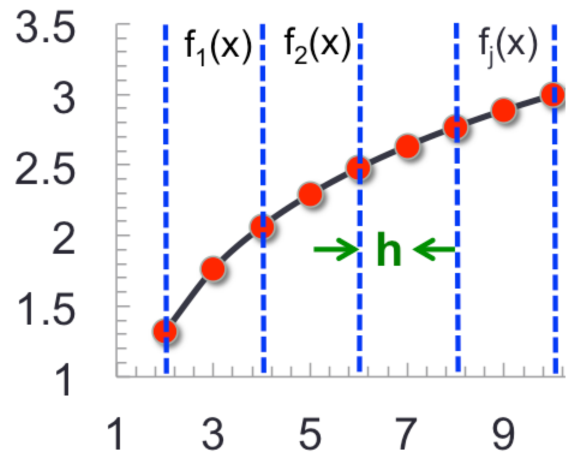
- Still unsatisfied since the function has large derivative at some interval…
  - Refine the "grid" at the slices where the errors exceeds your tolerance…but mind greed...
  - Use midpoint rule (rectangles instead of trapeze with *midpoint function evaluation*)
- Still unsatisfied ? Move one order up in the interpolation !

→ ***Simpson's rule***

Simpson rule: algorithm

- Use *quadratic* interpolator over *3-point* sub-partition of the interval [a,b].
- Ideally the number of points of [a,b] partition is *odd* since we need 3-point per "*slice*" (*more later*).

$$dF = \int_{x_i}^{x_{i+2}} f(x)\,dx = \sum_{j=0}^{2} f(x_{i+j})w_j dx$$



- The interpolator shall give exact value to the integral if the integrand is a quadratic function i.e. using f(x)=1, x or x^2 must give exact results.
- Three basis condition for 3 coefficients (*w*) results in

$$dF = \int_{x_i}^{x_{i+2}} f(x)\,dx = \frac{h}{3}\left[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right]$$

$$d\delta F = -\frac{h^5}{90} f^{(4)}(\xi)$$

Simpson rule: algorithm *(cont.)*

- As before assume h=(b-a)/n. Sum over all sub-intervals $[x_i, x_{i+2}]$

$$F = \int_a^b f(x)dx = \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right]$$

and for the error one has to note that there are n/2 sub-intervals (3 nodes each)…

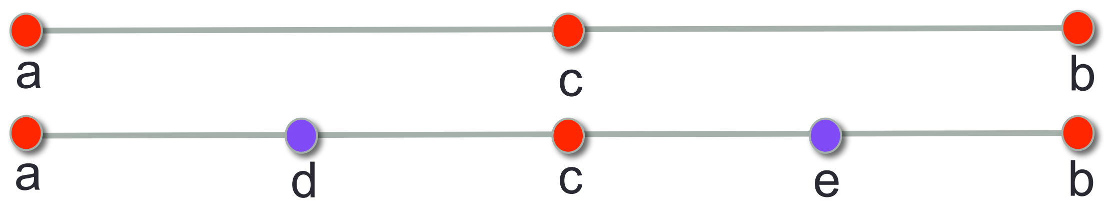
$$\delta F = -\frac{h^4}{180}(b-a)\left\langle f^{(4)}(\xi)\right\rangle$$

- Finally, if the number of points (n+1) is not odd i.e. (b-a)/h not even, there is still hope:
    - use Simpson for all sub-intervals and $$\int_{x_{n-1}}^{x_n} f(x)dx = \frac{h}{12}\left[-f(x_{n-2}) + 8f(x_{n-1}) + 5f(x_n)\right]$$
    - Use Simpson up to $[x_{n-6}, x_{n-4}]$ and then Simpson-3/8 for last 4 nodes (*cubic interpolation*) !

$$\int_{x_{n-3}}^{x_n} f(x)dx = \frac{3h}{8}\left[f(x_{n-3}) + 3f(x_{n-2}) + 3f(x_{n-1}) + f(x_n)\right]$$

Simpson rule: recursive algorithm

- Similarly to the Trapezoidal algorithm, there is a recursive variant for the Simpson rule.
- Suppose we divide the interval [a,b] into 2 and 4 segments



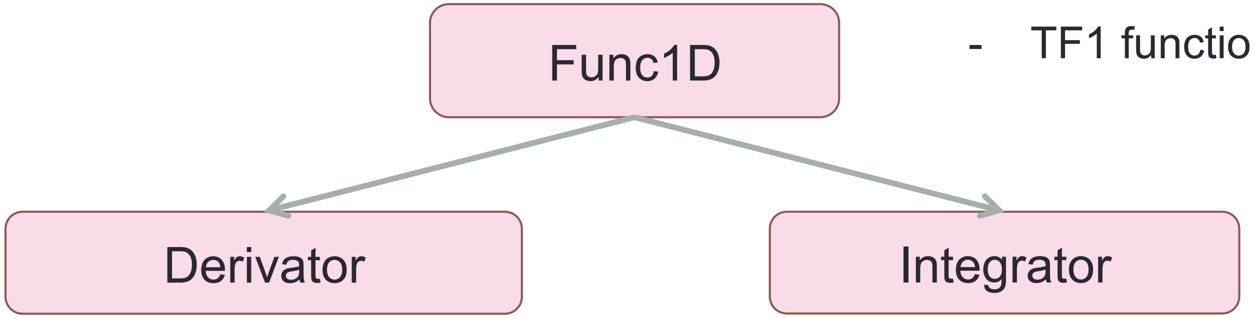
✓ Evaluate first for 2 segments $S_1 = \int_a^b f(x)\,dx$ and then for 4 segments $S_2 = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx$ and compare:

$$\boxed{\frac{|S_2 - S_1|}{15} < \varepsilon}$$

ε - tolerance required.

✓ If it fails the criteria then subdivide each of the smaller S_2 segments in two (to get S_3) and then compare S_2 with S_3....repeat until tolerance met.

Class scheme suggested



- TF1 function holder and methods

```
class Func1D{
public:
  Func1D(TF1* ff=NULL);
  ~Func1D();
  void SetFunc(TF1*);
  TF1* GetFunc()const;
  void Draw();
  double Evaluate(double x);
protected:
  TF1* F;
  static int Nplots;
};
```

```
class Integrator: public Func1D {
 public:
  Integrator(double xbeg=0, double xend=0, TF1* func=NULL) :
     x0(xbeg), x1(xend), Func1D(func) {;}
  ~Integrator() {;}
  void SetInterval(double a,double b);
  void TrapezoidalRule(int n, double& result, double& error);
  void SimpsonRule(int n, double& result, double& error);
 protected:
  double x0;
  double x1;
};
```

```
class Derivator: public Func1D {
  public:
    Derivator(TF1 *f=NULL);
    ~Derivator();
    double Deriv_1(double x, double h, int type=0);
    double Deriv_2(double x, double h, int type=0);
    …};
```

Gaussian quadrature methods

- When deriving Trapezoidal and Simpson rules, it was established that the rules yield ***exact*** values to the integral if the integrand is a linear or quadratic function respectively.

- Perhaps unnoticed, the results above have a *strong limitation* → evenly spaced nodes !

- Let's recap the goal: $$\boxed{F = \int_{-1}^{1} f(x)\,dx = \sum_{i=0}^{n} f(x_i) w_i}$$

  but now making it more interesting, no node position is assumed.

- **Case n=0** f(x)=1 → $1 \times w_0 = 2$ f(x)=x → $w_0 x_0 = 0$ so $x_0 = 0$

$$\boxed{F = \int_{-1}^{1} f(x)\,dx \cong 2 f(0)}$$

➢ Not very impressive….but let's see with n=1 i.e 2 unknown nodes and weights. Meaning we can target polynomials up to 3rd degree

Gaussian quadrature methods

$$F = \int_{-1}^{1} f(x)\,dx = \sum_{i=0}^{n} f(x_i) w_i$$

- **Case n=1**

f(x)=1 → $1\times w_0 + 1\times w_1 = 2$ f(x)=x → $w_0 x_0 + w_1 x_1 = 0$

f(x)=x^2 → $w_0 x_0^2 + w_1 x_1^2 = \frac{2}{3}$ f(x)=x^3 → $w_0 x_0^3 + w_1 x_1^3 = 0$

- The solution turns out to be $w_0 = w_1 = 1$ and $x_0 = -\frac{1}{\sqrt{3}}$, $x_1 = \frac{1}{\sqrt{3}}$

$$\boxed{F = \int_{-1}^{1} f(x)\,dx \cong f\left(\frac{-1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right)}$$

- Surprisingly good, can be continuously improved to higher degrees !
- Not limited to [-1,1] since one can easily transform to [a,b]

$$t = \frac{b + a + x(b-a)}{2}$$

COMPUTATIONAL PHYSICS

Numerical methods

Roots of equations

Bisection method

- Objective: find a solution (*root*) of the equation ***f(x)=0***
- Possibly the most simple algorithm stems from *Bolzano theorem* i.e.

  "If a continuous function has values of opposite sign inside an interval [a,b], it has a root in that interval"

- In practice:
  - Start with $x_L=a$ and $x_R=b$ such that $f(a)f(b)<0$. Take $x_0=(a+b)/2$ as solution guess and
  - Divide into

$$[x_L, x_R] = \left[a, \frac{a+b}{2}\right] \longrightarrow f(x_L)f(x_R) < 0\,?$$

…. New x_0, divide again...

$$[x_L, x_R] = \left[\frac{a+b}{2}, b\right] \longrightarrow f(x_L)f(x_R) < 0\,?$$

…. New x_0, divide again...

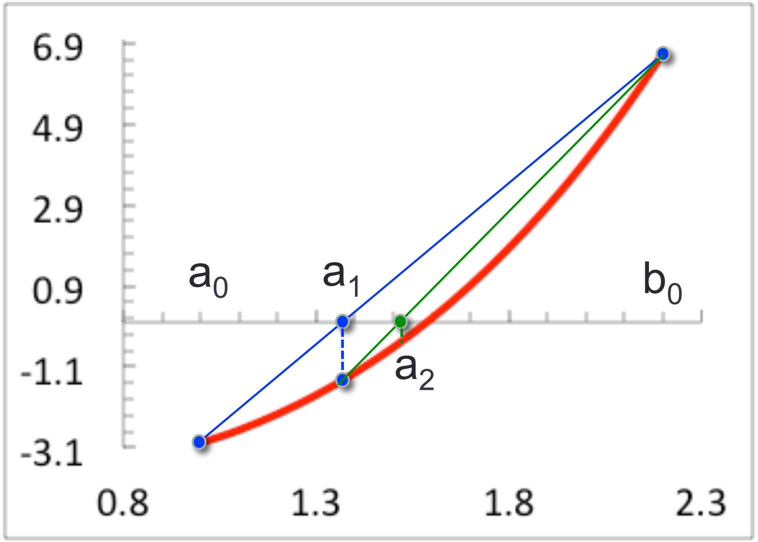
  - Iterate until x_L and x_R differ less than a given tolerance (ε) : solution = $(x_L + x_R)/2$
  - Although one eventually converges, the error bound is only halved at each iteration → *very slow method*…

Regula falsi

- Innovation: replace midpoint by root of linear polynomial passing through x_L and x_R.

- Calculate root from

$$f(x_0) = f(a_j) + \frac{f(b_i) - f(a_j)}{b_i - a_j}(x_0 - a_j) = 0$$



Then check again for which one is true:

- $f(a_j)f(x_0)<0 \rightarrow b_{i+1}=x_0$
- $f(x_0)\, f(b_i)<0 \rightarrow a_{j+1}=x_0$

- Repeat until converging in x_0

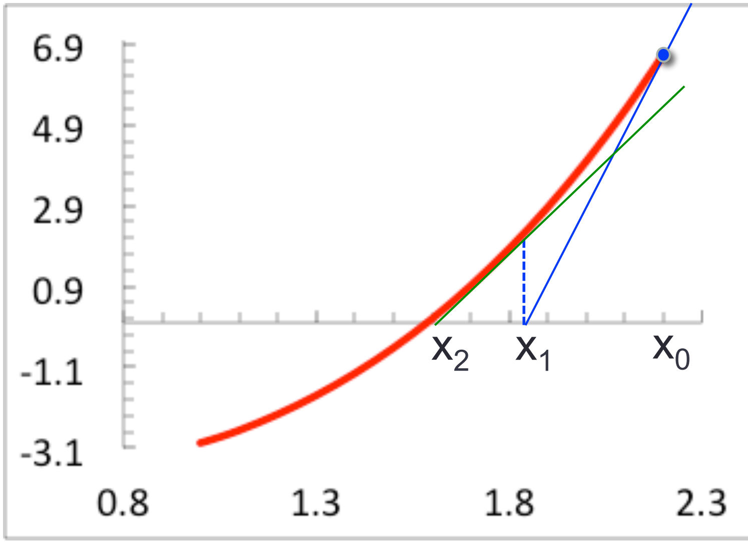
Newton-Raphson

- Innovation: rather than 2 "*moving boundary*" points, follow the function's slope !

Taylor expansion to 1st order

$$f(x_{i+1}) = f(x_i) + (x_{i+1} - x_i) f'(x_i) \approx 0$$

$$\boxed{x_{i+1} = x_i - f(x_i) / f'(x_i)}$$



- Extremely fast converge….but not guaranteed at all !

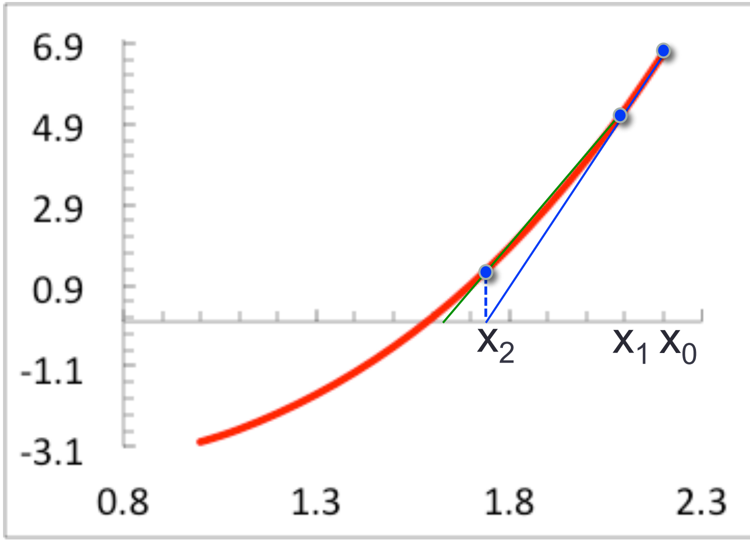
- It is key to start close to the actual root ***AND*** the function should not have derivative nulls !

- *Good practice*: start with some few "bisections" and then proceed to Newton-Raphson.

Secant method

- Innovation: replace the analytical derivative of Newton-Raphson by the numerical equivalent using latest 2 estimates !

$$x_{i+1} = x_i - f(x_i)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$



- Moderately fast convergence….but not guaranteed at all !
- Does not require explicit calculation of the derivative f'(x) !