2020

# INTRODUCTION TO NUMERICAL ANALYSIS

**Lecture 2-2:**
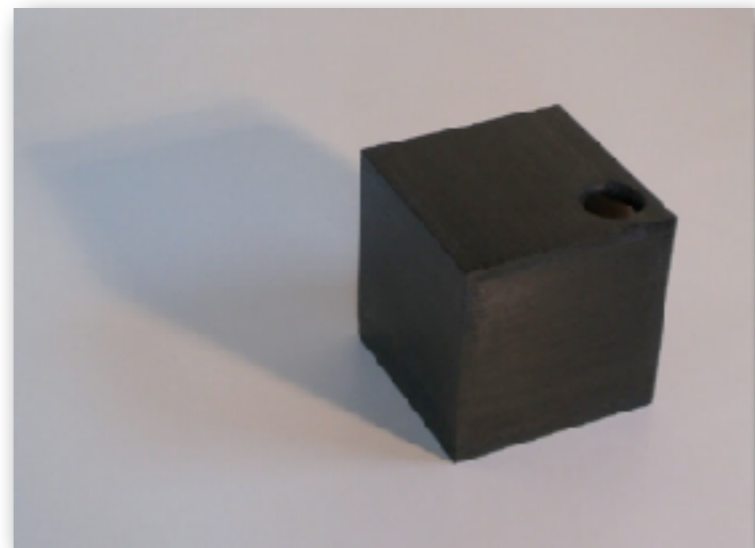**Numerical Differential & Integration**

Kai-Feng Chen
National Taiwan University

# ANALYTICAL VERSUS NUMERICAL

**A GENERAL RULE:**

- If you know the exact form, it's always better to do the calculus analytically unless it's not really doable.

- Although we could do the calculation numerically without a problem, but the precision is always a big issue.

- In this lecture, we will discuss the derivatives & integration for a black box function f(x).

$$f(x) =$$

# ANALYTICAL VERSUS NUMERICAL

## IF THE EXACT FORM IS KNOWN…

- Mathematica could be you good friend…
  https://www.wolframalpha.com/calculators/derivative-calculator/

WolframAlpha computational intelligence™

derivative x*sin(x)*cos(x)

∫π∑∂ Extended Keyboard  ⬆ Upload  ⦂⦂⦂ Examples  ⤬ Random

Derivative:  ☑ Step-by-step solution

$$\frac{d}{dx}(x \sin(x) \cos(x)) = -x \sin^2(x) + x \cos^2(x) + \sin(x) \cos(x)$$

Plots:

y

# ANALYTICAL VERSUS NUMERICAL

**ON THE OTHER HAND:**

- Even if you can do your derivatives or integrations analytically, it is still very useful to do the same thing in a numerical way as a very good cross check (ie. debug).

- Suppose, you have >50 different functions to be implement in your code, and you are calculating their derivatives analytically, even you have already calculated everything by yourself, but it does not guarantee you have no typo in your code!

Numerical calculus will give you a quick and easy check first!

# NUMERICAL DERIVATIVES

- Suppose, you have a function f(x), and now you want to compute f'(x), it's pretty easy, right?

$$\text{By definition, for } h \to 0 \quad f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- In principle we could insert a small **h**, maybe as small as possible under the conversion of the numerical calculations. But *THIS IS NOT TRUE* for numerical derivatives.

- So, let's try such a simple function that we could actually do the exact calculations easily:

$$f(x) = x^2 + \exp(x) + \log(x) + \sin(x)$$

$$f'(x) = 2x + \exp(x) + \frac{1}{x} + \cos(x)$$

# LET'S GIVE IT A QUICK TRY!

```python
import math

def f(x):
    return x**2+math.exp(x)+math.log(x)+math.sin(x)
def fp(x):
    return 2.*x+math.exp(x)+1./x+math.cos(x)

x, h = 0.5, 1E-2   ⇐ Starting from h = 1E-2
fp_exact = fp(x)

while h>1E-15:
    fp_numeric = (f(x+h) - f(x))/h
    print('h = %e' % h)
    print('Exact = %.16f,' % fp_exact, end=' ')
    print('Numeric = %.16f,' % fp_numeric, end=' ')
    print('diff = %.16f' % abs(fp_numeric-fp_exact))
    h /= 10.   ⇐ retry with smaller h!
```

l202-example-01.py

# A QUICK TRY...?

- Output:

```
         Exact = 5.5263038325905010

h = 1e-02, Numeric = 5.5224259820642496, diff = 0.0038778505262513
h = 1e-03, Numeric = 5.5258912717413011, diff = 0.0004125608491998
h = 1e-04, Numeric = 5.5262623253238274, diff = 0.0000415072666735
h = 1e-05, Numeric = 5.5262996793148380, diff = 0.0000041532756629
h = 1e-06, Numeric = 5.5263034173247396, diff = 0.0000004152657613
h = 1e-07, Numeric = 5.5263037901376313, diff = 0.0000000424528697
h = 1e-08, Numeric = 5.5263038811759193, diff = 0.0000000485854184
h = 1e-09, Numeric = 5.5263038589714579, diff = 0.0000000263809570
h = 1e-10, Numeric = 5.5263038589714579, diff = 0.0000000263809570
h = 1e-11, Numeric = 5.5263127407556549, diff = 0.0000089081651540
h = 1e-12, Numeric = 5.5262461273741783, diff = 0.0000577052163226
h = 1e-13, Numeric = 5.5311311086825290, diff = 0.0048272760920280
h = 1e-14, Numeric = 5.5511151231257818, diff = 0.0248112905352809
```

# OK, WHAT'S THE PROBLEM?

- For a small **h**, let's perform the Taylor expansions:

$$f(x + h) \approx f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \dots$$

*This is what we are calculating:* →

$$\frac{f(x + h) - f(x)}{h} \approx f'(x) + \boxed{\frac{h}{2}f''(x)} + \frac{h^2}{6}f'''(x) + \dots$$

In principle, we have an approximation error of **O(h)**, for such calculations. But there is another round-off error, close related to the machine precisions:

$$f(x + h) \approx f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \dots + \boxed{\epsilon_m}$$

# THE PROBLEM?

■ So, if we account for the numerical derivatives:

$$f'_{\text{numerical}}(x) = \frac{f(x+h) - f(x)}{h} \approx f'(x) + \left[\frac{h}{2}f''(x) + \frac{h^2}{6}f'''(x) + ...\right] + O\left(\frac{\epsilon_m}{h}\right)$$

The total error ~ $O(h) + O\left(\frac{\epsilon_m}{h}\right)$

For a double precision number: $\epsilon_m \approx O(10^{-15}) - O(10^{-16})$

The total error will saturation at: $h \approx O(\sqrt{\epsilon_m}) \approx O(10^{-8})$

This simply limit the precision of numerical derivatives,
and it cannot be better then 10⁻⁸, unless…

# THE TRICK IS ACTUALLY VERY SIMPLE...

$$f(x + \frac{h}{2}) \approx f(x) + \frac{h}{2} f'(x) + \frac{h^2}{8} f''(x) + \frac{h^3}{48} f'''(x) + ...$$

$$f(x - \frac{h}{2}) \approx f(x) - \frac{h}{2} f'(x) + \frac{h^2}{8} f''(x) - \frac{h^3}{48} f'''(x) + ...$$

$$f'_{\text{numerical}}(x) \approx \boxed{\frac{f(x + \frac{h}{2}) - f(x - \frac{h}{2})}{h}} \approx f'(x) + \left[ \frac{h^2}{24} f'''(x) + O(h^4)... \right] + O\left( \frac{\epsilon_m}{h} \right)$$

The total error ~ $O(h^2) + O\left( \frac{\epsilon_m}{h} \right) \approx O(h^2) + \left( \frac{10^{-16}}{h} \right)$

The total error will saturation at **O(10⁻¹⁰)** if $h \approx O(\epsilon_m^{1/3}) \approx O(10^{-5})$

This is the **"central difference"** method.

# A QUICK TRY AGAIN!

```python
import math

def f(x):
    return x**2+math.exp(x)+math.log(x)+math.sin(x)
def fp(x):
    return 2.*x+math.exp(x)+1./x+math.cos(x)

x, h = 0.5, 1E-2
fp_exact = fp(x)

while h>1E-15:
    fp_numeric = (f(x+h/2.) - f(x-h/2.))/h    ⇐ Update here
    print('h = %e' % h)
    print('Exact = %.16f,' % fp_exact, end=' ')
    print('Numeric = %.16f,' % fp_numeric, end=' ')
    print('diff = %.16f' % abs(fp_numeric-fp_exact))
    h /= 10.
```

l202-example-01a.py

# A QUICK TRY AGAIN! (II)

■ Output:

```
      Exact = 5.5263038325905010

h = 1e-02, Numeric = 5.5263737163485871, diff = 0.0000698837580861
h = 1e-03, Numeric = 5.5263045313882486, diff = 0.0000006987977477
h = 1e-04, Numeric = 5.5263038395758635, diff = 0.0000000069853625
h = 1e-05, Numeric = 5.5263038326591731, diff = 0.0000000000686722
h = 1e-06, Numeric = 5.5263038325481508, diff = 0.0000000000423501
h = 1e-07, Numeric = 5.5263038323261062, diff = 0.0000000002643947
h = 1e-08, Numeric = 5.5263038367669983, diff = 0.0000000041764974
h = 1e-09, Numeric = 5.5263036369268530, diff = 0.0000001956636480
h = 1e-10, Numeric = 5.5263038589714579, diff = 0.0000000263809570
h = 1e-11, Numeric = 5.5263349452161474, diff = 0.0000311126256465
h = 1e-12, Numeric = 5.5266902165840284, diff = 0.0003863839935274
h = 1e-13, Numeric = 5.5266902165840284, diff = 0.0003863839935274
h = 1e-14, Numeric = 5.5511151231257818, diff = 0.0248112905352809
```

# PRECISION VERSUS FINITE DIFFERENCE

- Naturally the full precision does depend on both the **step size** as well as the actual formulation:
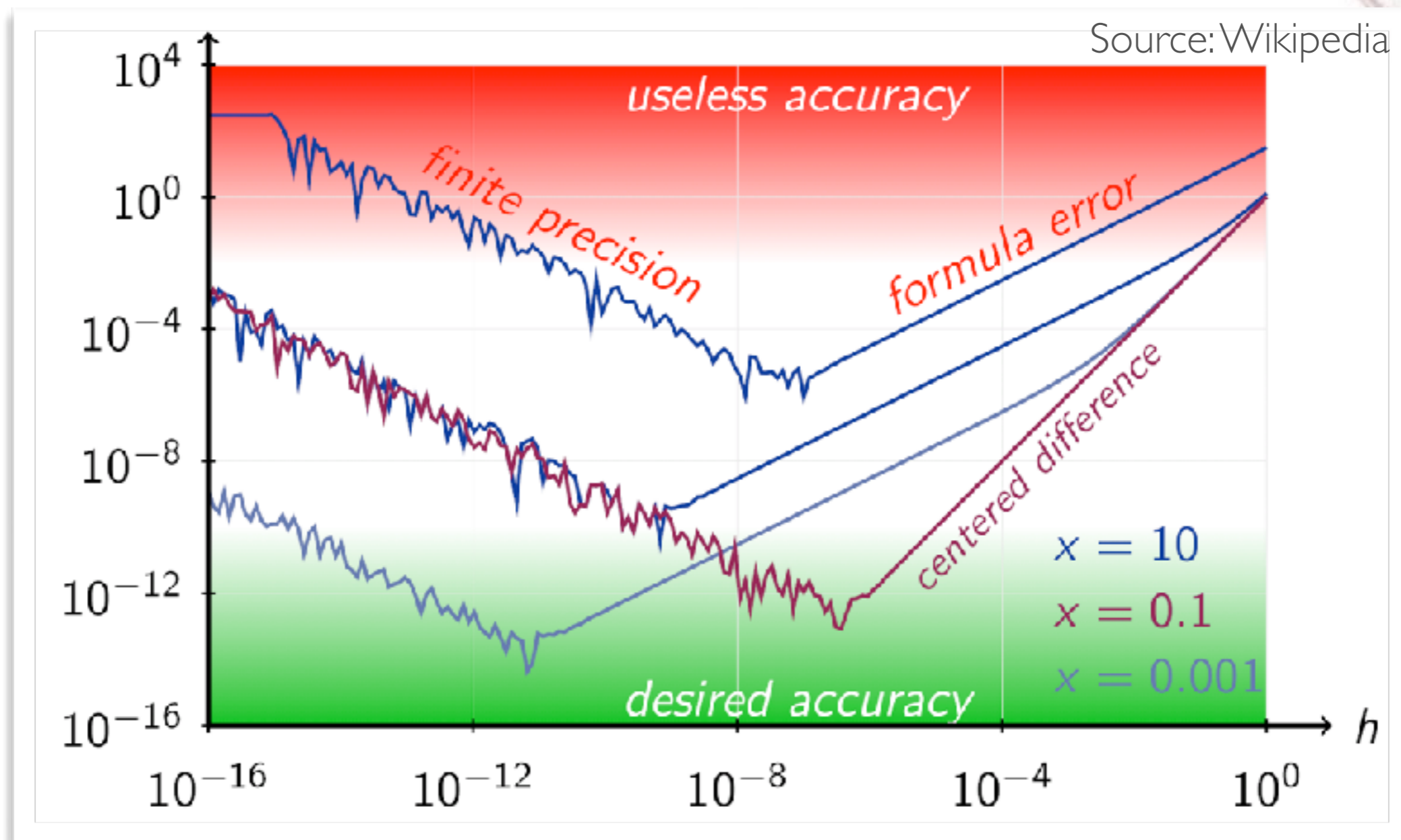
# A FURTHER IMPROVEMENT

- Let's repeat the trick of **"cancellation"**:

$$f(x + \frac{h}{4}) \approx f(x) + \frac{h}{4}f'(x) + \frac{h^2}{32}f''(x) + \frac{h^3}{384}f'''(x) + \dots$$

$$f(x - \frac{h}{4}) \approx f(x) - \frac{h}{4}f'(x) + \frac{h^2}{32}f''(x) - \frac{h^3}{384}f'''(x) + \dots$$

$$\frac{f(x + \frac{h}{4}) - f(x - \frac{h}{4})}{h} \approx \frac{1}{2}f'(x) + \boxed{\frac{h^2}{192}f'''(x)} + O(h^4)\dots$$

$$\frac{f(x + \frac{h}{2}) - f(x - \frac{h}{2})}{h} \approx f'(x) + \boxed{\frac{h^2}{24}f'''(x)} + O(h^4)\dots$$

*Simply repeat the same trick to remove the h² term.*

# A FURTHER IMPROVEMENT (II)

- Then

$$8\left[\frac{f(x+\frac{h}{4})-f(x-\frac{h}{4})}{h}\right]-\left[\frac{f(x+\frac{h}{2})-f(x-\frac{h}{2})}{h}\right]\approx 3f'(x)+[O(h^4)...]+O\left(\frac{\epsilon_m}{h}\right)$$

$$f'_{\text{numerical}}(x)\approx$$

$$\boxed{\frac{8f(x+\frac{h}{4})-8f(x-\frac{h}{4})-f(x+\frac{h}{2})+f(x-\frac{h}{2})}{3h}}+[O(h^4)...]+O\left(\frac{\epsilon_m}{h}\right)$$

*Take this term and neglect the rest*

The total error ~ $O(h^4)+O\left(\frac{\epsilon_m}{h}\right)\approx O(h^4)+\left(\frac{10^{-16}}{h}\right)$

The total error will saturation at **O(10⁻¹³)** if $h\approx O(\epsilon_m^{1/5})\approx O(10^{-3})$

15

# JUST CHANGE A LINE...

```python
import math

def f(x):
    return x**2+math.exp(x)+math.log(x)+math.sin(x)
def fp(x):
    return 2.*x+math.exp(x)+1./x+math.cos(x)

x, h = 0.5, 1E-2
fp_exact = fp(x)

while h>1E-15:
    fp_numeric = \                    ✎ Update here (note: a backslash "\" can wrap a python line)
    (8.*f(x+h/4.)+f(x-h/2.)-8.*f(x-h/4.)-f(x+h/2.))/(h*3.)
    print('h = %e' % h)
    print('Exact = %.16f,' % fp_exact, end=' ')
    print('Numeric = %.16f,' % fp_numeric, end=' ')
    print('diff = %.16f' % abs(fp_numeric-fp_exact))
    h /= 10.
```

l202-example-01b.py

16

# JUST CHANGE A LINE...(II)

■ Output results:

```
Exact = 5.5263038325905010

h = 1e-02, Numeric = 5.5263038315869801, diff = 0.0000000010035208
h = 1e-03, Numeric = 5.5263038325903402, diff = 0.0000000000001608
h = 1e-04, Numeric = 5.5263038325925598, diff = 0.0000000000020588
h = 1e-05, Numeric = 5.5263038327701954, diff = 0.0000000001796945
h = 1e-06, Numeric = 5.5263038328442100, diff = 0.0000000002537091
h = 1e-07, Numeric = 5.5263038249246188, diff = 0.0000000076658822
h = 1e-08, Numeric = 5.5263037257446959, diff = 0.0000001068458051
h = 1e-09, Numeric = 5.5263040070011948, diff = 0.0000001744106939
h = 1e-10, Numeric = 5.5263127407556549, diff = 0.0000089081651540
h = 1e-11, Numeric = 5.5263497481898094, diff = 0.0000459155993084
h = 1e-12, Numeric = 5.5258020381643282, diff = 0.0005017944261727
h = 1e-13, Numeric = 5.5215091758024446, diff = 0.0047946567880564
h = 1e-14, Numeric = 5.5807210704491190, diff = 0.0544172378586181
```

# HOW ABOUT THE SECOND DERIVATIVES?

■ In fact it is nothing special, just **CALCULATE IT TWICE**:

$$f'(x) \approx \frac{f(x + \frac{h}{2}) - f(x - \frac{h}{2})}{h}$$

$$f''(x) \approx \frac{f'(x + \frac{h}{2}) - f'(x - \frac{h}{2})}{h} \approx \frac{\frac{f(x+h)-f(x)}{h} - \frac{f(x)-f(x-h)}{h}}{h}$$

$$\boxed{f''(x) \approx \frac{f(x + h) + f(x - h) - 2f(x)}{h^2}}$$

The total error ~ $O(h^2) + O\left(\frac{\epsilon_m}{h^2}\right) \approx O(h^2) + \left(\frac{10^{-16}}{h^2}\right)$

The total error will saturation at **O(10⁻⁸)** if $h \approx O(\epsilon_m^{1/4}) \approx O(10^{-4})$

# HOW ABOUT THE SECOND DERIVATIVES? (II)

```python
import math

def f(x):
    return x**2+math.exp(x)+math.log(x)+math.sin(x)
def fp(x):
    return 2.*x+math.exp(x)+1./x+math.cos(x)
def fpp(x):
    return 2.+math.exp(x)-1./(x*x)-math.sin(x)

x, h = 0.5, 1E-2
fpp_exact = fpp(x)

while h>1E-15:
    fpp_numeric = \
    (f(x+h)+f(x-h)-2.*f(x))/(h*h)
    print('h = %e' % h)
    print('Exact = %.16f,' % fpp_exact, end=' ')
    print('Numeric = %.16f,' % fpp_numeric, end=' ')
    print('diff = %.16f' % abs(fpp_numeric-fpp_exact))
    h /= 10.
```

Nothing really different comparing to the previous code…

l202-example-01c.py

# HOW ABOUT THE SECOND DERIVATIVES? (III)

$$f(x) = x^2 + \exp(x) + \log(x) + \sin(x)$$

The analytical solution.

$$f''(x) = 2 + \exp(x) - \frac{1}{x^2} - \sin(x)$$

■ Output results:

```
Exact = -0.8307042679040748

h = 1e-02, Numeric = -0.8314867467085207, diff = 0.000782478804459
h = 1e-03, Numeric = -0.8307120906714260, diff = 0.0000078227673512
h = 1e-04, Numeric = -0.8307043497524091, diff = 0.0000000818483343
h = 1e-05, Numeric = -0.8307043941613300, diff = 0.000000126257552
h = 1e-06, Numeric = -0.8304468224196168, diff = 0.000257445844581
h = 1e-07, Numeric = -0.8437694987151185, diff = 0.0130652308110437
h = 1e-08, Numeric = +4.4408920985006244, diff = 5.2715963664046992
h = 1e-09, Numeric = +0.0000000000000000, diff = 0.8307042679040748
```

You can see the precision for 2nd order derivative is (much) worse if we only take the leading term.

# HOMEMADE CODE VS PUBLIC CODE

■ Although we have practiced some of the classical algorithms, you may use them in your own daily work. But sometimes is still recommended to use the well-tested professional code if they are available.



Homemade(?) Bat Car



A Porsche

# HOMEMADE CODE VS PUBLIC CODE (II)

## Homemade Code

- **Pro**
  - [ ] As the author you know the code to details. Not a black box.
  - [ ] Can be optimized for special cases (*may be faster for your own application*).
- **Con**
  - [ ] Less tested (may break at some special condition)
  - [ ] Less optimal (may be slower in general cases)

## Public Code

- **Pro**
  - [ ] Well tested, good protections (*less chance to break down at some extreme case*).
  - [ ] More optimized, can be faster in most of the cases.
- **Con**
  - [ ] A black box unless you really go through the codes.
  - [ ] May not fully fit your needs.

The actual choice: depends on your problem!

# GETTING START WITH NUMPY & SCIPY

**FROM THE OFFICIAL WEBSITE:**

- **NumPy**'s array type augments the Python language with an efficient data structure useful for numerical work, e.g., manipulating matrices. NumPy also provides basic numerical routines.

- **SciPy** contains additional routines needed in scientific work: for example, routines for computing integrals numerically, solving differential equations, optimization, etc.

> In short:
> **NumPy** = extended array + some routines
> **SciPy** = scientific tools based on NumPy

# TYPICAL WORK FLOW

Working on your own research topic (TH/EXP)

*You can think **NumPy/SciPy** are nothing more than a bigger math module. Don't think they are something very fancy!*

Need numerical analysis for resolving some numerical problems

still not enough...

Write your code with standard math module

if not enough...

Adding NumPy/SciPy/etc.

Other solutions: Google other package / write your own algorithm / Use a different language / etc...

**Problem solved!**

# NUMERICAL DERIVATIVES IN SCIPY

■ Just google — and you'll find it's just a simple function:

Google    scipy numerical derivative    🔍

全部    圖片    影片    新聞    地圖    更多         設定    工具

約有 68,500 項結果 (搜尋時間：0.39 秒)

scipy.misc.derivative — SciPy v0.18.1 Reference Guide
https://docs.scipy.org/doc/scipy-0.18.1/.../scipy.misc.derivative.html ▼ 翻譯這個網頁
Parameters: func : function. Input function. x0 : float. The point at which n-th derivative is found. dx :
float, optional. Spacing. n : int, optional. Order of the derivative. Default is 1. args : tuple, optional.
Arguments. order : int, optional. Number of points to use. must be odd.

Scipy.org    Docs    SciPy v1.0.0 Reference Guide    Miscellaneous routines (`scipy.misc`)

## scipy.misc.derivative

scipy.misc.**derivative**(*func, x0, dx=1.0, n=1, args=(), order=3*)

Find the n-th derivative of a function at a point.

Given a function, use a central difference formula with spacing $dx$ to compute the $n$-th derivative at $x0$.

http://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.derivative.html

# LET'S GIVE IT A TRY

```python
import math
import scipy.misc as misc    ⇐ import scipy.misc module

def f(x):
    return x**2+math.exp(x)+math.log(x)+math.sin(x)
def fp(x):
    return 2.*x+math.exp(x)+1./x+math.cos(x)

x, h = 0.5, 1E-2
fp_exact = fp(x)

while h>1E-15:
    fp_numeric = misc.derivative(f, x, h)    ⇐ just call it
    print('h = %e' % h)
    print('Exact = %.16f,' % fp_exact, end=' ')
    print('Numeric = %.16f,' % fp_numeric, end=' ')
    print('diff = %.16f' % abs(fp_numeric-fp_exact))
    h /= 10.
```

l202-example-02.py

# LET'S GIVE IT A TRY (II)

■ This gives us the best precision of **O(10⁻¹⁰)** when **h~10⁻⁶**.

```
            Exact = 5.5263038325905010

h = 1e-02, Numeric = 5.5265834157978029, diff = 0.0002795832073019
h = 1e-03, Numeric = 5.5263066277866368, diff = 0.0000027951961359
h = 1e-04, Numeric = 5.5263038605413151, diff = 0.0000000279508141
h = 1e-05, Numeric = 5.5263038328479110, diff = 0.0000000002574101
h = 1e-06, Numeric = 5.5263038326591731, diff = 0.0000000000686722
h = 1e-07, Numeric = 5.5263038323261062, diff = 0.0000000002643947
h = 1e-08, Numeric = 5.5263038589714588, diff = 0.0000000263809579
h = 1e-09, Numeric = 5.5263038589714579, diff = 0.0000000263809570
h = 1e-10, Numeric = 5.5263038589714579, diff = 0.0000000263809570
h = 1e-11, Numeric = 5.5263127407556549, diff = 0.0000089081651540
h = 1e-12, Numeric = 5.5260240827692533, diff = 0.0002797498212477
h = 1e-13, Numeric = 5.5278004396086535, diff = 0.0014966070181526
h = 1e-14, Numeric = 5.5289106626332787, diff = 0.0026068300427777
```

Very similar situation found!

# GO TO HIGHER ORDER

■ This gives us the best precision of **O(10⁻¹¹~10⁻¹²)** when **h~10⁻⁴**. Not a dramatically improvement…

```python
x, h = 0.5, 1E-2
fp_exact = fp(x)

while h>1E-15:                                    ⇓ update here
    fp_numeric = misc.derivative(f, x, h, order=5)
    print('h = %e' % h)
```

l202-example-02a.py (partial)

```
h = 1e-02, Numeric = 5.5263035753822134, diff = 0.00000002572082876
h = 1e-03, Numeric = 5.5263038325648601, diff = 0.0000000000256408
h = 1e-04, Numeric = 5.5263038325881197, diff = 0.00000000000023812
h = 1e-05, Numeric = 5.5263038325537019, diff = 0.0000000000367990
h = 1e-06, Numeric = 5.5263038325481508, diff = 0.0000000000423501
h = 1e-07, Numeric = 5.5263038328812177, diff = 0.0000000002907168
```

# COMMENTS

■ You may already observed during our tests above, in the numeral derivatives, it is important to minimize the total error rather than the approximation error only:

□ Reducing the spacing **h** to a very small number is not a good idea in principle; cancellation of higher order terms are more effective.

□ **In any case the numeral derivative cannot be very precise.**

□ Some algorithms can reduce the spacing according to the estimated approximation error. This is called **"Adaptive Stepping"**, e.g.

$$h' = h \cdot \left( \frac{\epsilon_R}{2\epsilon_T} \right)^{\frac{1}{3}}$$

$\epsilon_R$ : rounding error

$\epsilon_T$ : approximation error

Updated stepping    Initial stepping

➡ *for your own further study.*

# INTERMISSION

- You have learned that the **central difference method** cancels the term up to $f''$, and the improved higher order method cancels the term up to $f''''$. You may try the code (`1202-xample-01a.py` and `1202-example-01b.py`) and calculate the numerical derivative for a polynomial up to **x²** and **x³**. Can the calculation be 100% precise or not?

- For example you may try such a simple function:

$$f(x) = 5x^3 + 4x^2 + 3x + 2$$

$$\rightarrow f'(x) = 15x^2 + 8x + 3$$

30

# NUMERICAL INTEGRATION

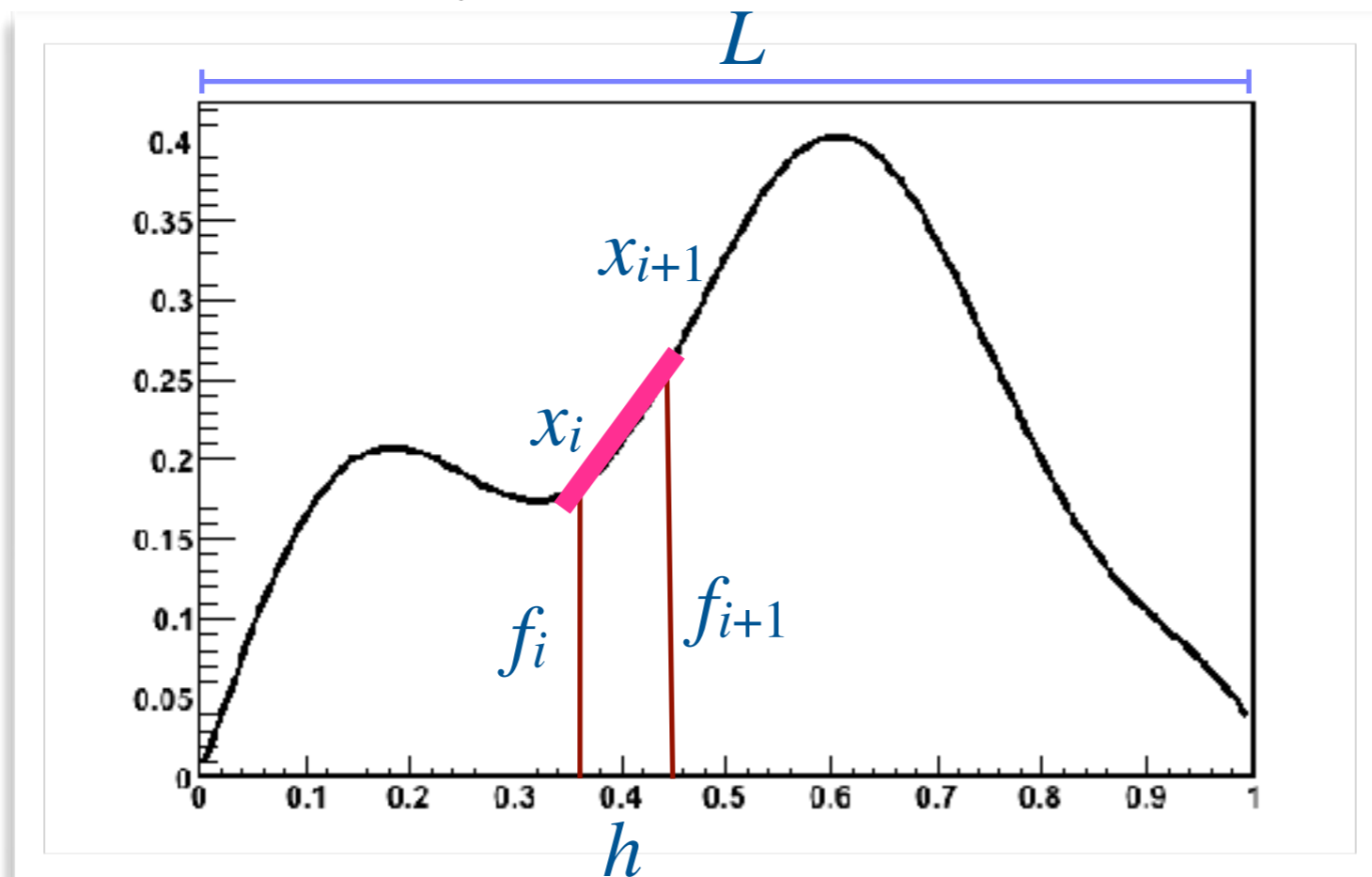- Starting from some **super basic** integration rules:
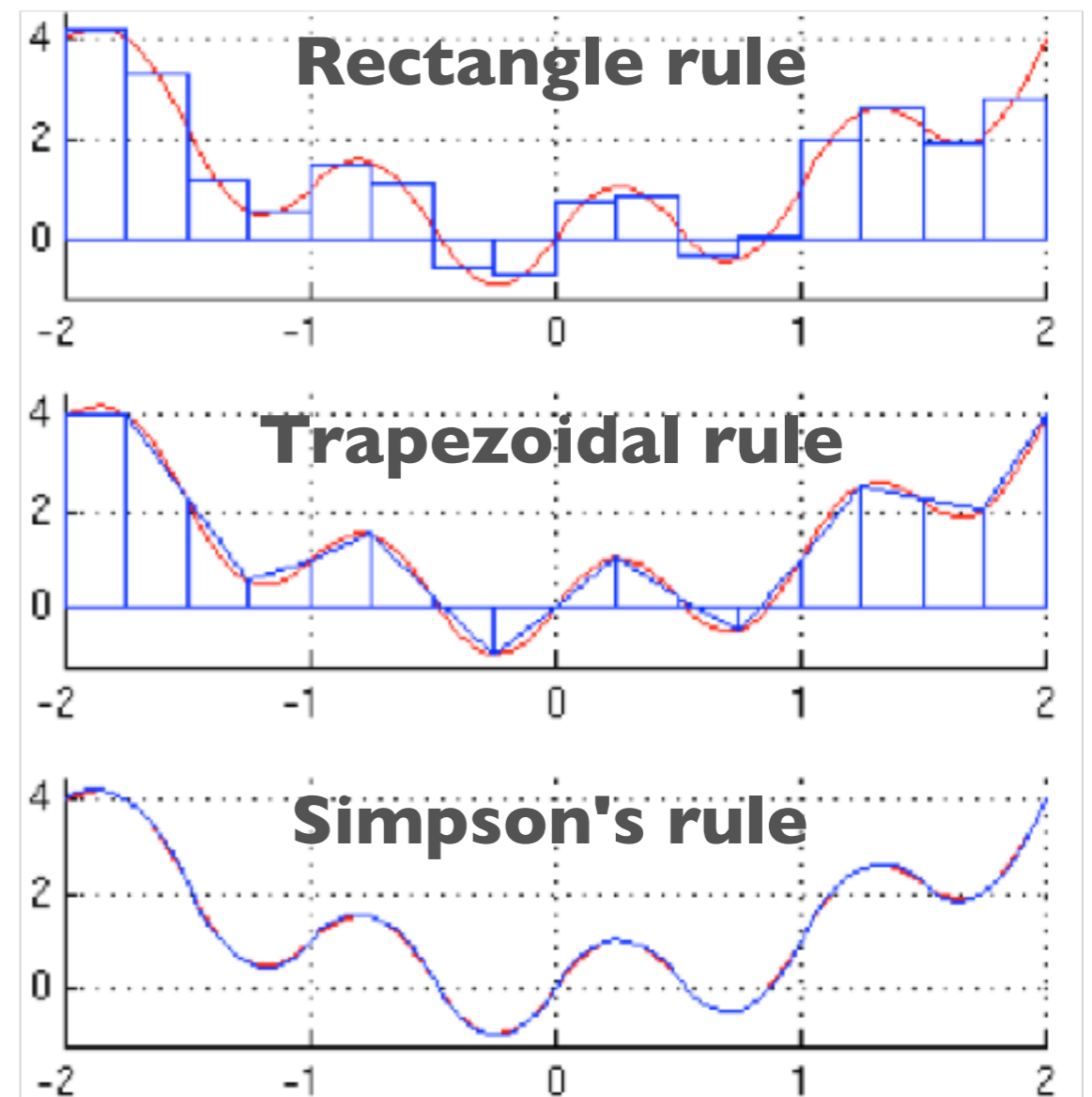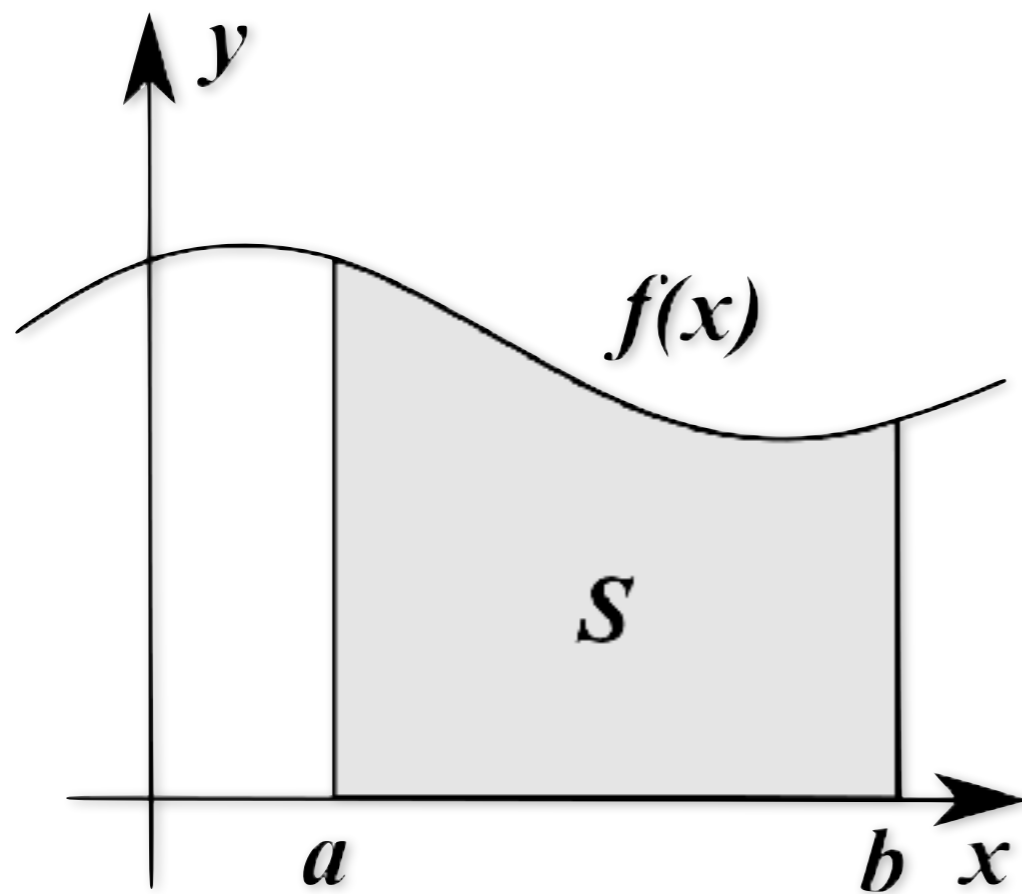
# NUMERICAL INTEGRATION (II)

■ Let's practice a classical integration: the **trapezoidal rule**, e.g.

$$f(x) = x - x^2 + x^3 - x^4 + \frac{\sin(13x)}{13}$$

$$\int f(x)dx = \frac{x^2}{2} - \frac{x^3}{3} + \frac{x^4}{4} - \frac{x^5}{5} - \frac{\cos(13x)}{169}$$

# TRAPEZOIDAL RULE: IMPLEMENTATION

```python
import math

def f(x):
    return x - x**2 + x**3 - x**4 + math.sin(x*13.)/13.
def fint(x):
    return x**2/2. - x**3/3. + x**4/4. - x**5/5. - math.cos(x*13.)/169.

fint_exact = fint(1.2)-fint(0.)
area, x, h = 0., 0., 1E-3   ⇐ start with h = 10⁻³
f0 = f1 = f(x)
while x<1.2-h*0.5:
    f0, f1 = f1, f(x+h)
    x += h
    area += f0+f1
area *= h/2.

print('Exact: %.16f, Numerical: %.16f, diff: %.16f' \
% (fint_exact,area,abs(fint_exact-area)))
```

```
Exact:     0.1765358676046381,
Numerical: 0.1765352854227494,
diff:      0.0000005821818886
```

l202-example-03.py

33

# HOW ABOUT A SMALLER STEP SIZE?

- As expected, the precision cannot be improved by simply using a smaller **h**.

- It's very time consuming: smaller h, more operations, more computing time needed.

```
        Exact = 0.1765358676046381

h = 1e-02, Numeric = 0.1764776451750985, diff = 0.0000582224295395
h = 1e-03, Numeric = 0.1765352854227494, diff = 0.0000005821818886
h = 1e-04, Numeric = 0.1765358617829089, diff = 0.0000000058217292
h = 1e-05, Numeric = 0.1765358675475263, diff = 0.0000000000571118
h = 1e-06, Numeric = 0.1765358676034689, diff = 0.0000000000011692
h = 1e-07, Numeric = 0.1765358677680409, diff = 0.0000000001634028
h = 1e-08, Numeric = 0.1765358661586719, diff = 0.0000000014459662
```

# ERROR ANALYSIS: APPROXIMATION ERROR

- Consider Taylor expansions for $f(x)$:

$$f(x + h) \approx f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \dots$$

Exact integration:

$$\int_0^h f(x + \eta)d\eta \approx hf(x) + \frac{h^2}{2}f'(x) + \frac{h^3}{6}f''(x) + \frac{h^4}{24}f'''(x) + \dots$$

Trapezoidal rule:

$$\frac{h}{2}[f(x) + f(x + h)] \approx hf(x) + \frac{h^2}{2}f'(x) + \frac{h^3}{4}f''(x) + \frac{h^4}{12}f'''(x) + \dots$$

Error per interval: $\delta \approx \dfrac{h^3}{12}f''(x) + \dots$

Approximation error: $\epsilon_{\text{approx}} \approx O(h^3) \times \dfrac{L}{h} \approx O(h^2)$

35

# ERROR ANALYSIS: TOTAL ERROR

- If we believe the theory:

$$\epsilon_{\text{roundoff}} \approx O(\sqrt{N}\epsilon_m) \qquad N \propto \frac{L}{h} = \text{total no. of operation steps.}$$

- The total error:

$$\epsilon_{\text{total}} \approx O(\sqrt{N}\epsilon_m) + O(h^2) \approx O\left(\frac{\epsilon_m}{\sqrt{h}}\right) + O(h^2)$$

For a double precision float point number, $\epsilon_m \approx O(10^{-15}) - O(10^{-16})$

The best precision will be of **O(10⁻¹²)** when $h \approx O(\epsilon_m^{1/2.5}) \approx O(10^{-6})$

Well, this is just an order of magnitude guess,
usually it's highly dependent on the algorithm and your exact coding.
*(also, smaller h means much more computing time!)*

# AN EASY IMPROVEMENT

- Another classical method: **Simpson's Rule**.
- Instead of liner interpolation, we could use a 2nd-order (parabola) interpolation along 3 points:
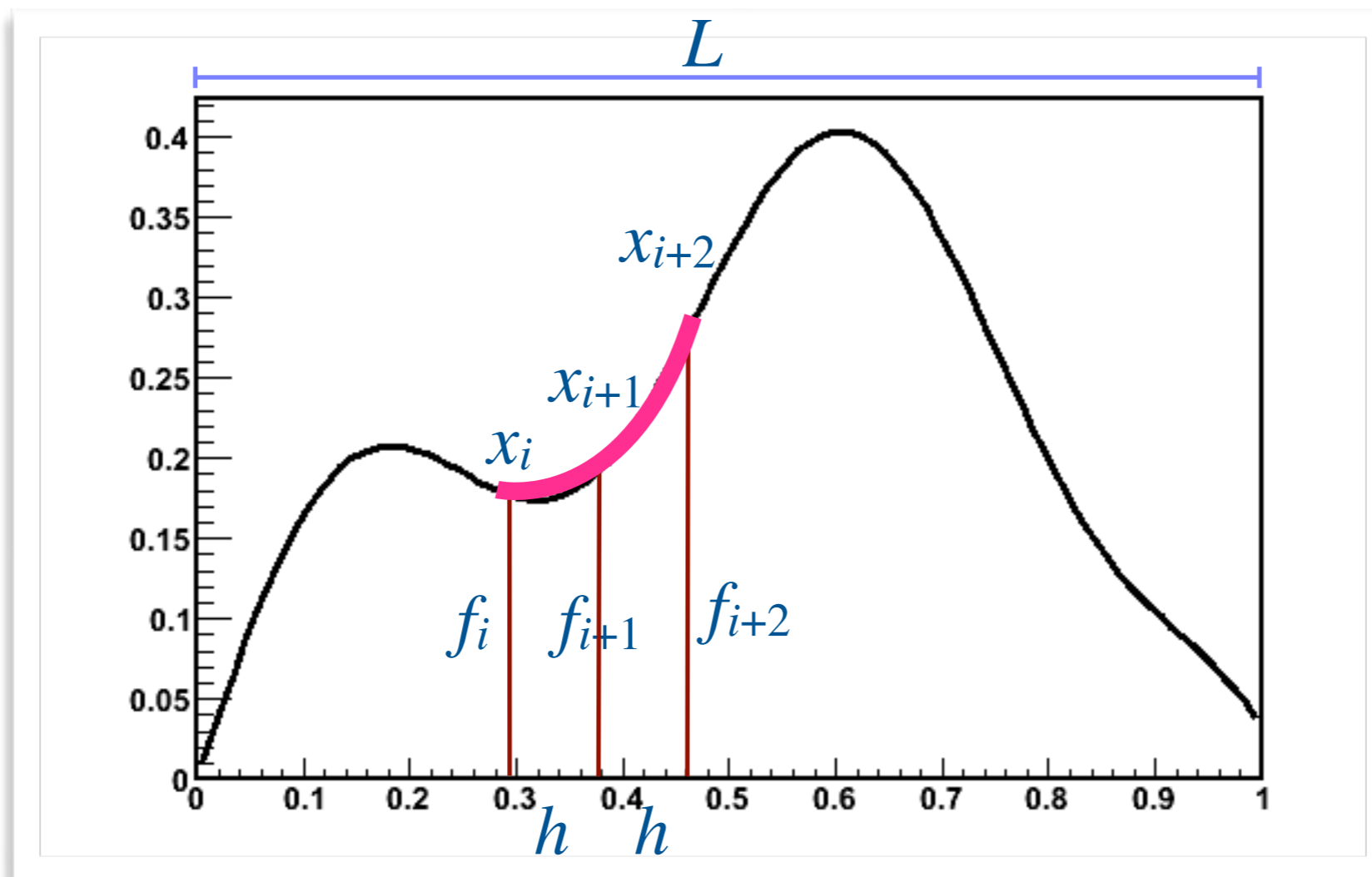
# THE FORMULAE

■ Treat the function as a parabola between the interval **[–1,+1]**:

$$f(x) \approx ax^2 + bx + c \quad \Longrightarrow \quad \int_{-1}^{+1} f(x)dx = \left[\frac{a}{3}x^3 + \frac{b}{2}x^2 + cx\right]_{-1}^{+1} = \frac{2a}{3} + 2c$$

$$\begin{cases} f(+1) \approx a + b + c \\ f(0) \approx c \\ f(-1) \approx a - b + c \end{cases} \quad \textit{Solve a,b,c}: \int_{-1}^{+1} f(x)dx = \frac{f(-1)}{3} + \frac{4f(0)}{3} + \frac{f(+1)}{3}$$

Simpson's rule: $\int_{0}^{2h} f(x+\eta)d\eta \approx \frac{h}{3}f(x) + \frac{4h}{3}f(x+h) + \frac{h}{3}f(x+2h)$

Total integration:

$$\int f(x)dx \approx \frac{h}{3}f_1 + \frac{4h}{3}f_2 + \frac{2h}{3}f_3 + \frac{4h}{3}f_4 + \frac{2h}{3}f_5 + \ldots + \frac{4h}{3}f_{N-1} + \frac{h}{3}f_N$$

# SIMPSON'S RULE: IMPLEMENTATION

```python
import math

def f(x):
    return x - x**2 + x**3 - x**4 + math.sin(x*13.)/13.
def fint(x):
    return x**2/2. - x**3/3. + x**4/4. - x**5/5. -
math.cos(x*13.)/169.

fint_exact = fint(1.2)-fint(0.)
area, x, h = 0., 0., 1E-3
f0 = f1 = f2 = f(x)
while x<1.2-h*0.5:
    f0, f1, f2 = f2, f(x+h), f(x+h*2.)
    x += h*2.
    area += f0+f1*4.+f2
area *= h/3.

print('Exact: %.16f, Numerical: %.16f, diff: %.16f' \
% (fint_exact,area,abs(fint_exact-area)))
```

```
Exact:     0.1765358676046381,
Numerical: 0.1765358676063498,
diff:      0.00000000000017117
```

l202-example-04.py

39

# SIMPSON'S RULE: ERROR ANALYSIS

■ Could we cancel the **O(h³)** and **O(h⁴)** term?

$$f(x + h) \approx f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + \ldots$$

$$f(x + 2h) \approx f(x) + 2hf'(x) + 2h^2 f''(x) + \frac{4h^3}{3}f'''(x) + \frac{2h^4}{3}f^{(4)}(x) + \ldots$$

$$\frac{h}{3}f(x) + \frac{4h}{3}f(x + h) + \frac{h}{3}f(x + 2h)$$

$$\approx 2hf(x) + 2h^2 f'(x) + \frac{4h^3}{3}f''(x) + \frac{2h^4}{3}f'''(x) + \frac{5h^5}{18}f^{(4)}(x) + \ldots$$

$$\int_0^{2h} f(x + \eta)d\eta \approx 2hf(x) + 2h^2 f'(x) + \frac{4h^3}{3}f''(x) + \frac{2h^4}{3}f'''(x) + \frac{4h^5}{15}f^{(4)}(x) + \ldots$$

$$\delta \approx \frac{h^5}{90}f^{(4)}(x) + \ldots \quad \Longrightarrow \quad \epsilon_{\text{approx}} \approx O(h^5) \times \frac{L}{h} \approx O(h^4)$$

# SIMPSON'S RULE: ERROR ANALYSIS (II)

- The total error is given by:

$$\epsilon_{\text{total}} \approx O(\sqrt{N}\epsilon_m) + O(h^4) \approx O\left(\frac{\epsilon_m}{\sqrt{h}}\right) + O(h^4)$$

The best precision could be of **O(10⁻¹⁴)** when $h \approx O(\epsilon_m^{1/4.5}) \approx O(10^{-4})$

*Is it true? Not too bad in principle...*

```
           Exact = 0.1765358676046381

h = 1e-02, Numeric = 0.1765358847654857, diff = 0.0000000171608476
h = 1e-03, Numeric = 0.1765358676063498, diff = 0.0000000000017117
h = 1e-04, Numeric = 0.1765358676047102, diff = 0.0000000000000721
h = 1e-05, Numeric = 0.1765358676043926, diff = 0.0000000000002455
h = 1e-06, Numeric = 0.1765358676131805, diff = 0.0000000000085424
h = 1e-07, Numeric = 0.1765358676224454, diff = 0.0000000000178073
h = 1e-08, Numeric = 0.1765358675909871, diff = 0.0000000000136510
```

# COMMENTS

- Maybe you already realized the general rule:
  - The approximate error of numerical integration heavily depends on the algorithm (*cancellation of higher order error*).
  - The round-off error and speed of calculation depend on the number of steps.
  - The best algorithm: **as less steps/points as possible, with as higher order as possible**.
  - **Adaptive stepping can be a solution.**
  - Many integration rules can be generalized as **sum of the weights times the function f(x) values**, ie.

$$\int f(x)dx \approx \sum_{i=1}^{N} w_i \cdot f(x_i)$$

*The art is to find the best approximation of $\mathbf{w_i}$ with smallest $\mathbf{N}$!*

# INTERMISSION

- Those "fixed points" integration rules have several limitations — such as you cannot integrate over singularities. Try to integrate over some functions with singularities and see what will you get?

- Consider a function of polynomials up to **$x^3$** but without knowing its exact form. **How many points of $f(x_i)$ are required to calculate its exact integration at least?**

# THE TRICK?

- Consider a function of polynomials up to $x^3$ but without knowing its exact form. **How many points of $f(x_i)$ are required to calculate its exact integration at least?**

- Maybe you are thinking of **4 times** since one needs already 3 points to describe a full parabola (up to $x^2$). But in fact we only need to calculate **TWICE**.

Consider a function like: $f(x) = c_3 x^3 + c_2 x^2 + c_1 x + c_0$

In fact you only need to calculate f(x) twice
to get an exact integration in [-1,+1]

$$I = \int_{-1}^{+1} f(x)dx = \sum w_i f(x_i) = f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right)$$

# HOW IT COMES?

- Assuming we can do it with two points, ie. **4 unknowns**:

$$I = \int_{-1}^{+1} f(x)dx = \sum w_i f(x_i) = w_1 f(x_1) + w_2 f(x_2)$$

$$f(x) = 1 \quad \Rightarrow \quad I = \int_{-1}^{+1} 1dx = 2 = w_1 + w_2$$

And this integration should valid for any f(x) up to O(x³):

$$f(x) = x \quad \Rightarrow \quad I = \int_{-1}^{+1} xdx = 0 = w_1 x_1 + w_2 x_2$$

$$f(x) = x^2 \quad \Rightarrow \quad I = \int_{-1}^{+1} x^2 dx = \frac{2}{3} = w_1 x_1^2 + w_2 x_2^2$$
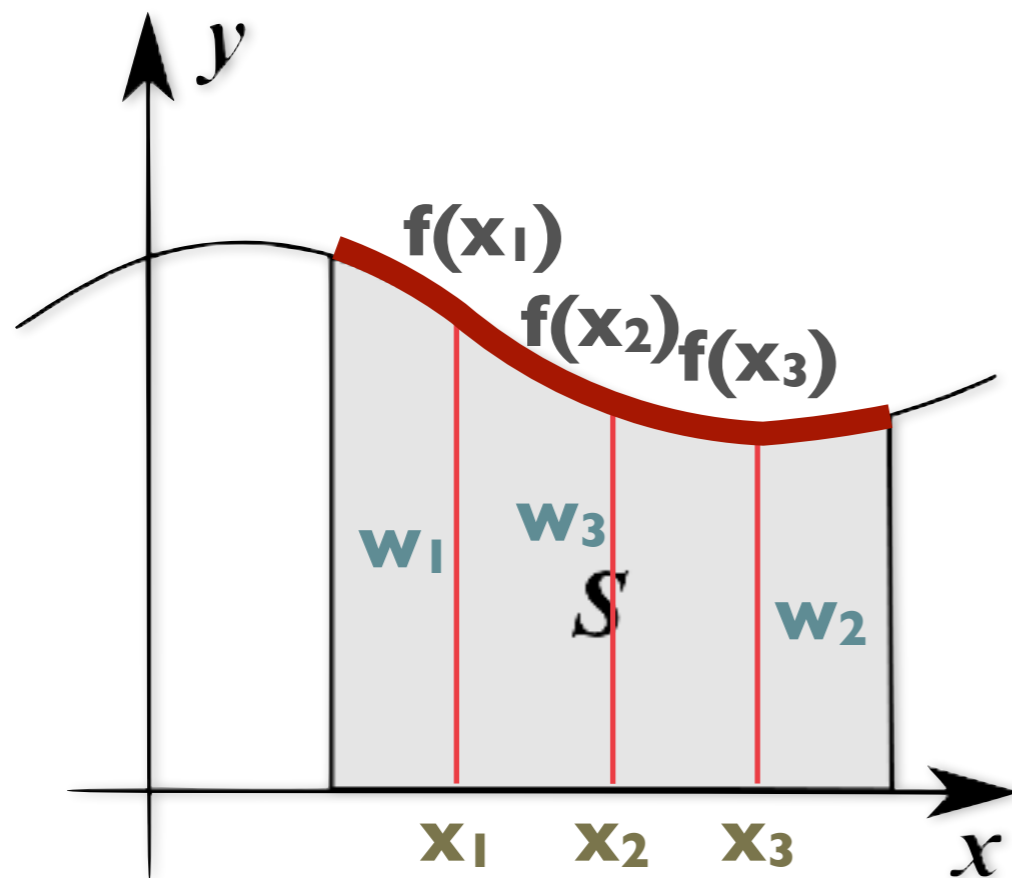
$$f(x) = x^3 \quad \Rightarrow \quad I = \int_{-1}^{+1} x^3 dx = 0 = w_1 x_1^3 + w_2 x_2^3$$

Solve 4 questions for 4 unknowns:

$$w_1, w_2 = 1, \quad x_1, x_2 = \pm\frac{1}{\sqrt{3}}$$

45

# HOW ABOUT HIGHER ORDER SOLUTIONS?

- In a similar way one consider the case of 3 points, which should be able to solve the exact integration up to $x^5$.



**2 points**
- ➡ 4 unknowns $(x_i, w_i)$
- ➡ solve up to $x^0, x^1, x^2, x^3$

**3 points**
- ➡ 6 unknowns $(x_i, w_i)$
- ➡ solve up to $x^0, x^1, x^2, x^3, x^4, x^5$

⋮

# HOW ABOUT HIGHER ORDER SOLUTIONS?

**6 unknowns**

$$I = \int_{-1}^{+1} f(x)dx = w_1 f(x_1) + w_2 f(x_2) + w_3 f(x_3)$$

$$f(x) = 1 \quad \Rightarrow \quad I = \int_{-1}^{+1} 1\,dx = 2 = w_1 + w_2 + w_3$$

$$f(x) = x \quad \Rightarrow \quad I = \int_{-1}^{+1} x\,dx = 0 = w_1 x_1 + w_2 x_2 + w_3 x_3$$

$$f(x) = x^2 \quad \Rightarrow \quad I = \int_{-1}^{+1} x^2\,dx = \frac{2}{3} = w_1 x_1^2 + w_2 x_2^2 + w_3 x_3^2$$

$$f(x) = x^3 \quad \Rightarrow \quad I = \int_{-1}^{+1} x^3\,dx = 0 = w_1 x_1^3 + w_2 x_2^3 + w_3 x_3^3$$

$$f(x) = x^4 \quad \Rightarrow \quad I = \int_{-1}^{+1} x^4\,dx = \frac{2}{5} = w_1 x_1^4 + w_2 x_2^4 + w_3 x_3^4$$

$$f(x) = x^5 \quad \Rightarrow \quad I = \int_{-1}^{+1} x^5\,dx = 0 = w_1 x_1^5 + w_2 x_2^5 + w_3 x_3^5$$

$$\begin{cases} w_1 = \dfrac{5}{9} \\[2mm] w_2 = \dfrac{8}{9} \\[2mm] w_3 = \dfrac{5}{9} \\[2mm] x_1 = -\sqrt{\dfrac{3}{5}} \\[2mm] x_2 = 0 \\[2mm] x_3 = +\sqrt{\dfrac{3}{5}} \end{cases}$$

Solve 6 questions for 6 unknowns.

# GAUSSIAN QUADRATURE

■ In fact this is called **Gaussian quadrature** of 2 and 3 points. By choosing proper locations of $x$ and the associated weights on $f(x)$, one can minimize the needs of calculation and get the best estimation of a fixed 1D integration.
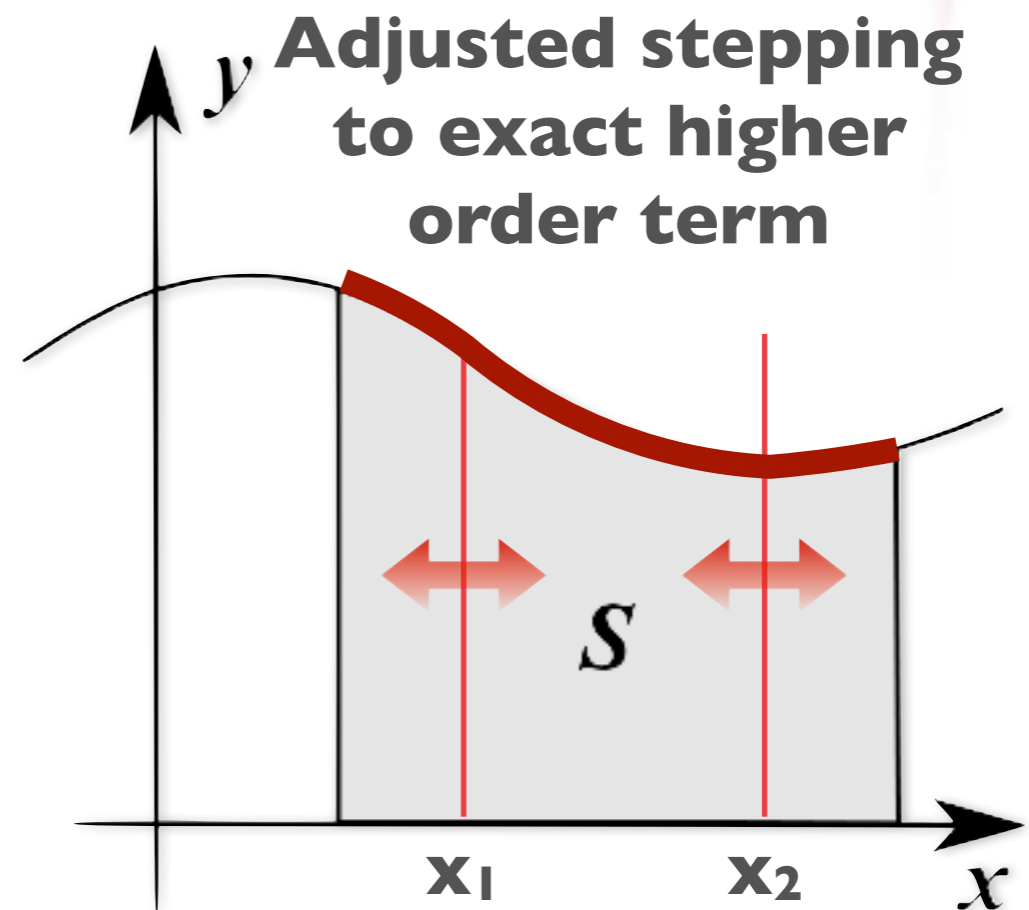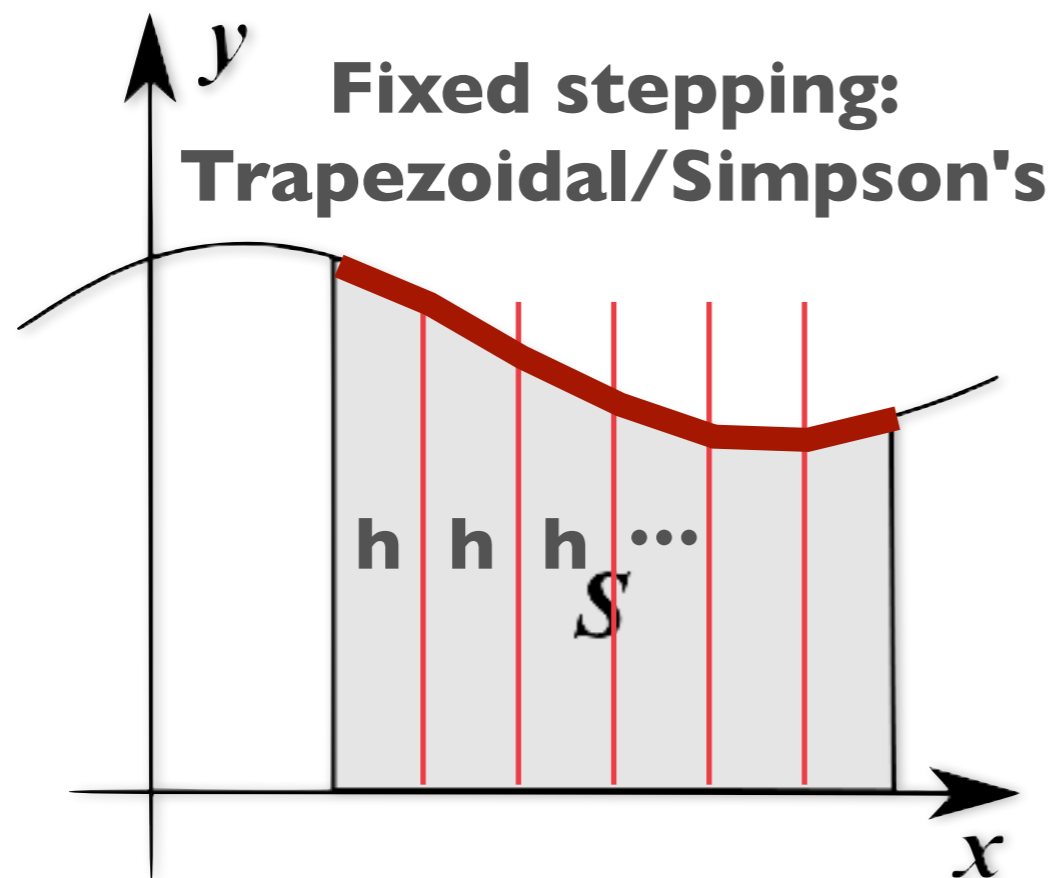
# GAUSSIAN QUADRATURE (CONT.)

- For a simple integration problem we discussed above, the associated polynomials are **Legendre polynomials $P_n(x)$**, and the method is usually known as **Gauss-Legendre quadrature**.
- Given with n-points we get the exact integration up to $x^{2n-1}$ power, the next term $x^{2n}$ is the approximation error.
- Several lower order points ($x_i$ are the roots of $P_n(x) = 0$) ➡

| Number of points, $n$ | Points, $x_i$ | Weights, $w_i$ |
|---|---|---|
| 1 | 0 | 2 |
| 2 | $\pm\sqrt{\frac{1}{3}}$ | 1 |
| 3 | 0 | $\frac{8}{9}$ |
|   | $\pm\sqrt{\frac{3}{5}}$ | $\frac{5}{9}$ |
| 4 | $\pm\sqrt{\frac{3}{7} - \frac{2}{7}\sqrt{\frac{6}{5}}}$ | $\frac{18+\sqrt{30}}{36}$ |
|   | $\pm\sqrt{\frac{3}{7} + \frac{2}{7}\sqrt{\frac{6}{5}}}$ | $\frac{18-\sqrt{30}}{36}$ |
| 5 | 0 | $\frac{128}{225}$ |
|   | $\pm\frac{1}{3}\sqrt{5 - 2\sqrt{\frac{10}{7}}}$ | $\frac{322+13\sqrt{70}}{900}$ |
|   | $\pm\frac{1}{3}\sqrt{5 + 2\sqrt{\frac{10}{7}}}$ | $\frac{322-13\sqrt{70}}{900}$ |

# IF NOT WITHIN [−1,+1]?

■ In general case an integral over **[a, b]** must be transformed into an integral over **[–1,+1]** before applying the Gaussian quadrature rule. This change of interval can be carried out as following:

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^{+1} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right) dx$$

$$\approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right)$$

*Nothing special but a simple coordination transformation would work.*

# A QUICK IMPLEMENTATION

■ Let's implement an example calculation with **21 points**:

```python
fint_exact = fint(1.2)-fint(0.)
npoints = 21                                    ⇓ see the code for the full table
weights = [[0.1460811336496904, +0.0000000000000000],
 . . . . . .
           [0.0160172282577743, +0.9937521706203895]]
area, min, max = 0., 0., 1.2

for i in range(npoints):
    x = ((max-min)*weights[i][1] + (max+min))/2.
    area += f(x)*weights[i][0]
area *= (max-min)/2.

print('Exact: %.16f, Numerical: %.16f, diff: %.16f' \
      % (fint_exact,area,abs(fint_exact-area)))
```

l202-example-05.py (partial)

```
Exact:      0.1765358676046381,
Numerical: 0.1765358676046379,
diff:       0.0000000000000002
```

*With almost full precision with only* **21 points***; Note the Simpson's rule requires 10K operations!*

# WHEN THE CODE BREAKS?

■ In order to see the **"break down" of the integration**, let's try to integrate over a **simple polynomial up to $x^n$**:

$$f(x) = 1 + x + x^2 + x^3 + \cdots + x^n$$

■ The corresponding **f(x)** and **fint(x)** are:

```python
def f(x, n):
    val = 0.
    for i in range(n+1): val += x**i
    return val

def fint(x, n):
    val = 0.
    for i in range(n+1): val += x**(i+1)/(i+1)
    return val
```

$$\sum_{i=0}^{n} x^i$$

$$\sum_{i=0}^{n} \frac{x^{i+1}}{i+1}$$

l202-example-05a.py (partial)

# WHEN THE CODE BREAKS? (CONT.)

- In order to see the effect easier — implement a
  5-point Gauss–Legendre quadrature instead of the 21-point version.

```python
npoints = 5
weights = [[0.5688888888888889,  +0.0000000000000000],
           [0.4786286704993665,  -0.5384693101056831],
           [0.4786286704993665,  +0.5384693101056831],
           [0.2369268850561891,  -0.9061798459386640],
           [0.2369268850561891,  +0.9061798459386640]]
min, max = 0., 1.

for n in range(15):
    fint_exact = fint(max,n)-fint(min,n)

    area = 0.
    for i in range(npoints):
        x = ((max-min)*weights[i][1] + (max+min))/2.
        area += f(x,n)*weights[i][0]
    area *= (max-min)/2.

    print('Power: %2d, Exact: %.16f, Numerical: %.16f, diff: %.16f' \
            % (n, fint_exact,area,abs(fint_exact-area)))
```

53

# WHEN THE CODE BREAKS? (CONT.)

- The integration does break after $x^{2n}$:

$$f(x) = \boxed{1 + x + x^2 + x^3 + \cdots + x^9} + \boxed{x^{10} + \cdots}$$

A 5-point integration rule can be exact up to here.     Not the rest…

```
Power:  0, Exact: 1.0000000000000000, Numerical: 1.0000000000000000, diff: 0.0000000000000000
Power:  1, Exact: 1.5000000000000000, Numerical: 1.5000000000000002, diff: 0.0000000000000002
. . . . . .
Power:  6, Exact: 2.5928571428571425, Numerical: 2.5928571428571430, diff: 0.0000000000000004
Power:  7, Exact: 2.7178571428571425, Numerical: 2.7178571428571425, diff: 0.0000000000000000
Power:  8, Exact: 2.8289682539682537, Numerical: 2.8289682539682537, diff: 0.0000000000000000
Power:  9, Exact: 2.9289682539682538, Numerical: 2.9289682539682538, diff: 0.0000000000000000
Power: 10, Exact: 3.0198773448773446, Numerical: 3.0198759133282937, diff: 0.0000014315490509
Power: 11, Exact: 3.1032106782106781, Numerical: 3.1032013731418489, diff: 0.0000093050688292
Power: 12, Exact: 3.1801337551337552, Numerical: 3.1800998537274987, diff: 0.0000339014062565
Power: 13, Exact: 3.2515623265623268, Numerical: 3.2514709047199117, diff: 0.0000914218424151
Power: 14, Exact: 3.3182289932289937, Numerical: 3.3180253879631731, diff: 0.0002036052658205
```

*Remark: What will you get if you still do 21-point integration?*

# NUMERICAL INTEGRATION WITH SCIPY

- You'll find there are many different integration tools in SciPy:

Scipy.org  Docs  SciPy v1.0.0 Reference Guide

## Integration and ODEs (scipy.integrate)

### Integrating functions, given function object

| | |
|---|---|
| quad(func, a, b[, args, full_output, ...]) | Compute a definite integral. |
| dblquad(func, a, b, gfun, hfun[, args, ...]) | Compute a double integral. |
| tplquad(func, a, b, gfun, hfun, qfun, rfun) | Compute a triple (definite) integral. |
| nquad(func, ranges[, args, opts, full_output]) | Integration over multiple variables. |
| fixed_quad(func, a, b[, args, n]) | Compute a definite integral using fixed-order Gaussian quadrature. |
| quadrature(func, a, b[, args, tol, rtol, ...]) | Compute a definite integral using fixed-tolerance Gaussian quadrature. |
| romberg(function, a, b[, args, tol, rtol, ...]) | Romberg integration of a callable function or method. |
| quad_explain([output]) | Print extra information about integrate.quad() parameters and returns. |
| newton_cotes(rn[, equal]) | Return weights and error coefficient for Newton-Cotes integration. |
| IntegrationWarning | Warning on issues during integration. |

### Integrating functions, given fixed samples

| | |
|---|---|
| trapz(y[, x, dx, axis]) | Integrate along the given axis using the composite trapezoidal rule. |
| cumtrapz(y[, x, dx, axis, initial]) | Cumulatively integrate y(x) using the composite trapezoidal rule. |
| simps(y[, x, dx, axis, even]) | Integrate y(x) using samples along the given axis and the composite Simpson's rule. |
| romb(y[, dx, axis, show]) | Romberg integration using samples of a function. |

> The **quad** is a general integration tool with <u>QUADPACK</u>.
> *From the name you can already guess the algorithm!*

http://docs.scipy.org/doc/scipy/reference/integrate.html#module-scipy.integrate

# INTEGRATION WITH QUAD() FUNCTION

```python
import math
import scipy.integrate as integrate

def f(x):
    return x - x**2 + x**3 - x**4 + math.sin(x*13.)/13.
def fint(x):
    return x**2/2. - x**3/3. + x**4/4. - x**5/5. -
math.cos(x*13.)/169.

fint_exact = fint(1.2)-fint(0.)

quad,quaderr = integrate.quad(f,0.,1.2,)

print('Exact: %.16f' %  fint_exact)
print('Numerical: %.16f+-%.16f, diff: %.16f' % \
      (quad,quaderr,abs(fint_exact-quad)))
```

l202-example-06.py

```
Exact:     0.1765358676046381
Numerical: 0.1765358676046380+-0.0000000000000029
diff:      0.0000000000000001
```

# FINAL REMARK

- It is very easy to use the **NumPy/SciPy** routines to do the numerical derivatives and integration: *just import the module, call the function, get your results!*

- However the limitation of these functions is not different from our homemade code: **don't use a too small stepping size!**

- You may find the integration is very precise and fast — this is due to the algorithm in the QUADPACK (based on Gaussian quadrature and written in Fortran). You can check the online document for details.

- There are few other functions provided by SciPy library for solving the problems in different cases. You can again, dig out more by yourself!

# HANDS-ON SESSION

- **Practice 1:**

  Integration rules with even higher orders can be constructed easily, for example, comparing Simpson's rule to 3/8 rule:

  **Simpson** [order 2]: $\int_0^{2h} f(x + \eta)d\eta \approx \frac{h}{3}f(x) + \frac{4h}{3}f(x+h) + \frac{h}{3}f(x+2h)$

  **3/8** [order 3]:

  $$\int_0^{3h} f(x + \eta)d\eta \approx \frac{3h}{8}f(x) + \frac{9h}{8}f(x+h) + \frac{9h}{8}f(x+2h) + \frac{3h}{8}f(x+3h)$$

  Try to modify `l202-example-04.py` to implement the **3/8 integration rule** and see how precise you can get?

# HANDS-ON SESSION

- **Practice 2:**

  The integration of cosine function is sine; let's modify the
  **1202-example-06.py** [integration with the `quad()` function]
  code to calculate the integration of a simple cosine and see how
  precise the calculation you can get, i.e.:

  ```python
  def f(x):
      return math.cos(x)
  def fint(x):
      return math.sin(x)
  ```

  by integrating $f(x)$ over the intervals of $[0,\pi]$, $[0,100\pi]$, $[0,1000\pi]$,
  $[0,100.5\pi]$, $[0,1000.5\pi]$. Is it always very precise?