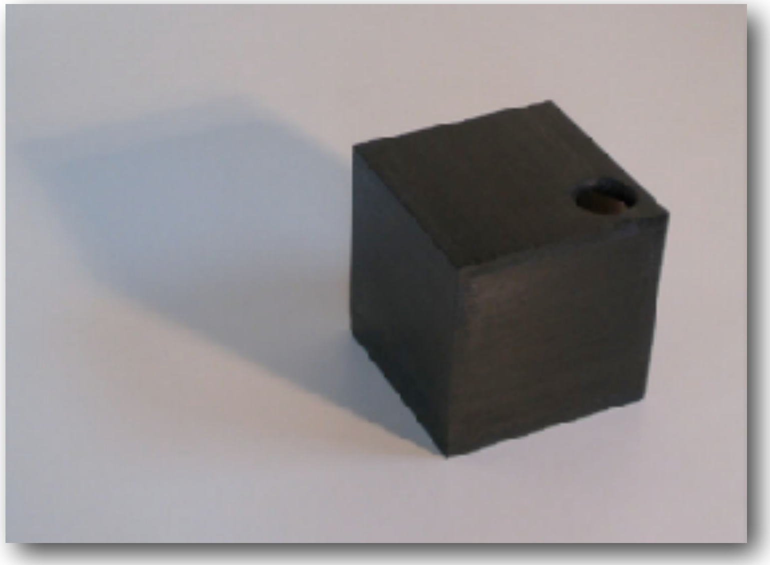
Numerical Algorithms

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).

$$f(x) =$$

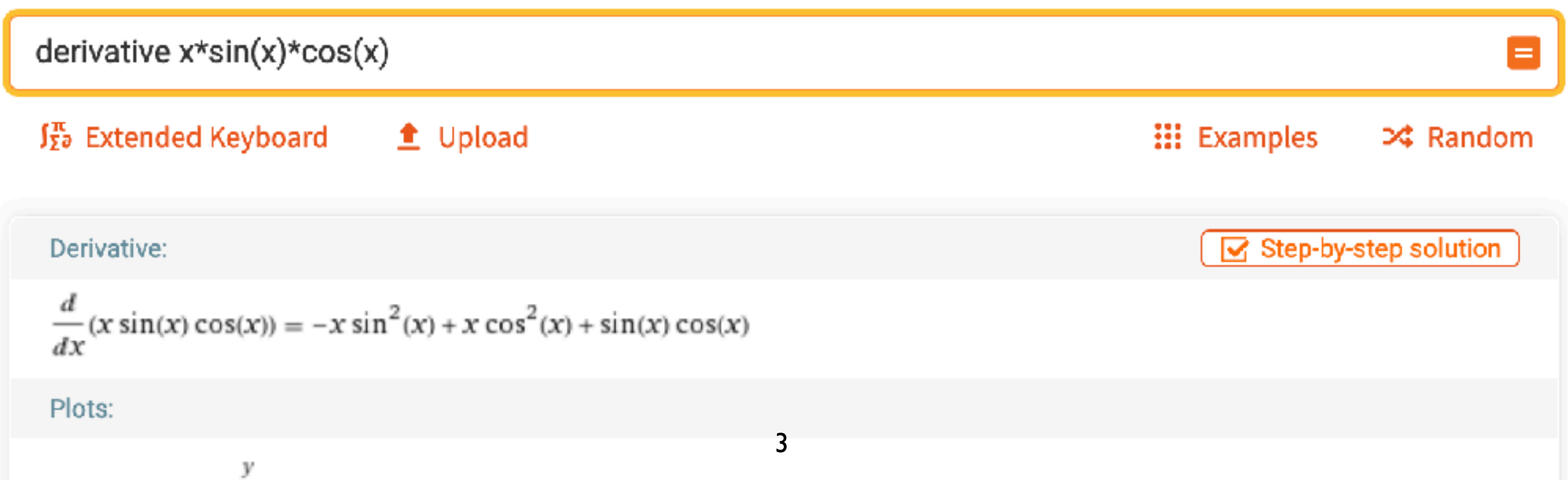


ANALYTICAL VERSUS NUMERICAL

IF THE EXACT FORM IS KNOWN…

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

WolframAlpha computational intelligence.



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?

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

- 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:

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

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

LET'S GIVE IT A QUICK TRY!

```
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 = 5.526303832590501 0

h = 1e-02, Numeric = 5.522425982064249 6, diff = 0.003877850526251 3
h = 1e-03, Numeric = 5.525891271741301 1, diff = 0.000412560849199 8
h = 1e-04, Numeric = 5.526262325323827 4, diff = 0.000041507266673 5
h = 1e-05, Numeric = 5.526299679314838 0, diff = 0.000004153275662 9
h = 1e-06, Numeric = 5.526303417324739 6, diff = 0.000000415265761 3
h = 1e-07, Numeric = 5.526303790137631 3, diff = 0.000000042452869 7
h = 1e-08, Numeric = 5.526303881175919 3, diff = 0.000000048585418 4
h = 1e-09, Numeric = 5.526303858971457 9, diff = 0.000000026380957 0
h = 1e-10, Numeric = 5.526303858971457 9, diff = 0.000000026380957 0
h = 1e-11, Numeric = 5.526312740755654 9, diff = 0.000008908165154 0
h = 1e-12, Numeric = 5.526246127374178 3, diff = 0.000057705216322 6
h = 1e-13, Numeric = 5.531131108682529 0, diff = 0.004827276092028 0
h = 1e-14, Numeric = 5.551115123125781 8, diff = 0.024811290535280 9
```

OK, WHAT'S THE PROBLEM?

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

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

This is what we are calculating:

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

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:

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

THE PROBLEM?

- So, if we account for the numerical derivatives:

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

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...

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

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

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

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!

```
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)

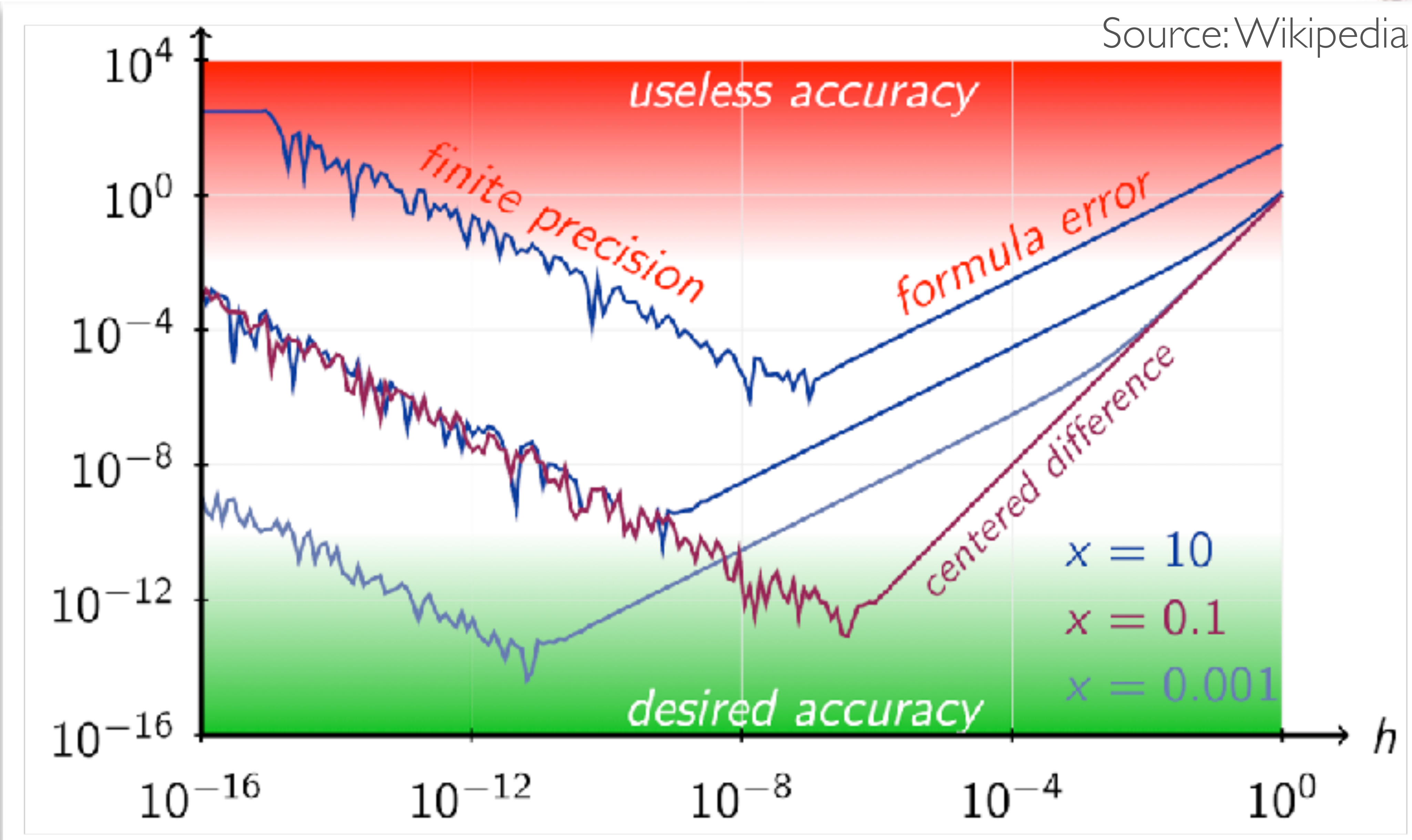
- Output:

```
          Exact = 5.5263038325905010

h = 1e-02, Numeric = 5.526373716348587l, diff = 0.000069883758086l
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"**:

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

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

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

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

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 $h\approx O(\epsilon_m^{1/5})\approx O(10^{-3})$

JUST CHANGE A LINE...

```
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 = \
    (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.
```

⇙ Update here (note: a backslash "\" can wrap a python line)

l202-example-01b.py

JUST CHANGE A LINE...(II)

- Output results:

```
Exact = 5.526303832590501O

h = 1e-02, Numeric = 5.526303831586980l, 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)

```
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.

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

- Output results:

```
Exact = -0.8307042679040748

h = 1e-02, Numeric = -0.8314867467085207, diff = 0.0007824788044459
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 = 0.0000001262572552
h = 1e-06, Numeric = -0.8304468224196168, diff = 0.0002574454844581
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

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

Working on your own research topic (TH/EXP)

Need numerical analysis for resolving some numerical problems

Write your code with standard math module

if not enough...

Adding NumPy/SciPy/etc.

still not enough...

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

```
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...

```
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 = 5.526303575382134, diff = 0.0000002572082876
h = 1e-03, Numeric = 5.5263038325648601, diff = 0.0000000000256408
h = 1e-04, Numeric = 5.5263038325881197, diff = 0.0000000000023812
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.

$$\underbrace{h'}_{\text{Updated stepping}} = \underbrace{h}_{\text{Initial stepping}} \cdot \left(\frac{\epsilon_R}{2\epsilon_T}\right)^{\frac{1}{3}}$$

ϵ_R : rounding error

ϵ_T : approximation error

➡ *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 $\mathbf{x^2}$ and $\mathbf{x^3}$. Can the calculation be 100% precise or not?
- For example you may try such a simple function:

$$f(x) = 5x^3 + 4x^2 + 3x + 2$$
$$\rightarrow f'(x) = 15x^2 + 8x + 3$$

NUMERICAL INTEGRATION

- Starting from some **super basic** integration rules: