2018

INTRODUCTION TO NUMERICAL ANALYSIS

Lecture 2-5: Root finding & minimization

Kai-Feng Chen
National Taiwan University

ROOT FINDING

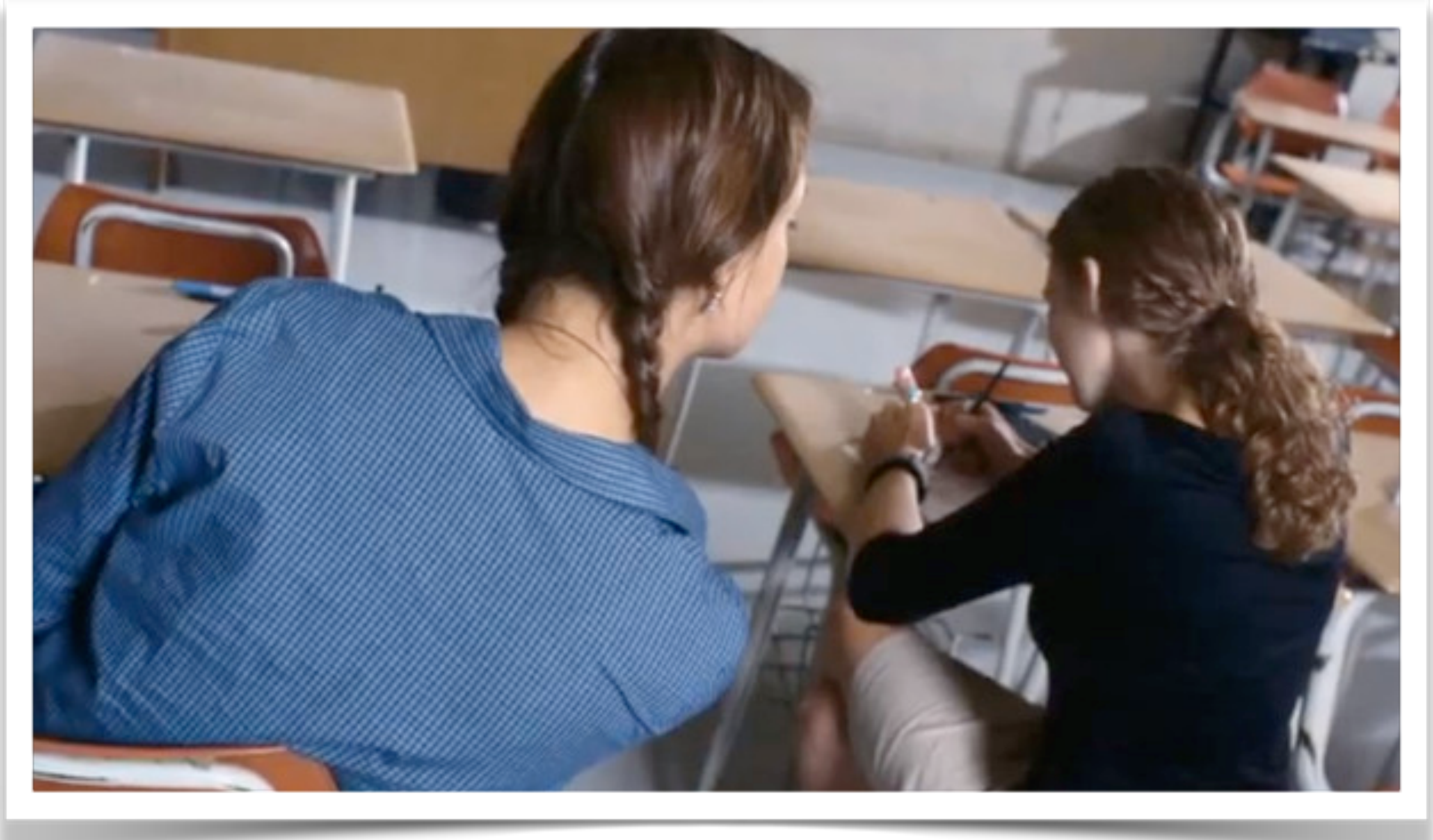
- Root finding is one of classical algebra problems since your high school times...

For a given function f(x),

if f(x) = 0, what's the x?

A CLASSICAL METHOD:
FIND THE ANSWER WITH YOUR EYES

- I'm not talking about peeking at other person's answer sheet...

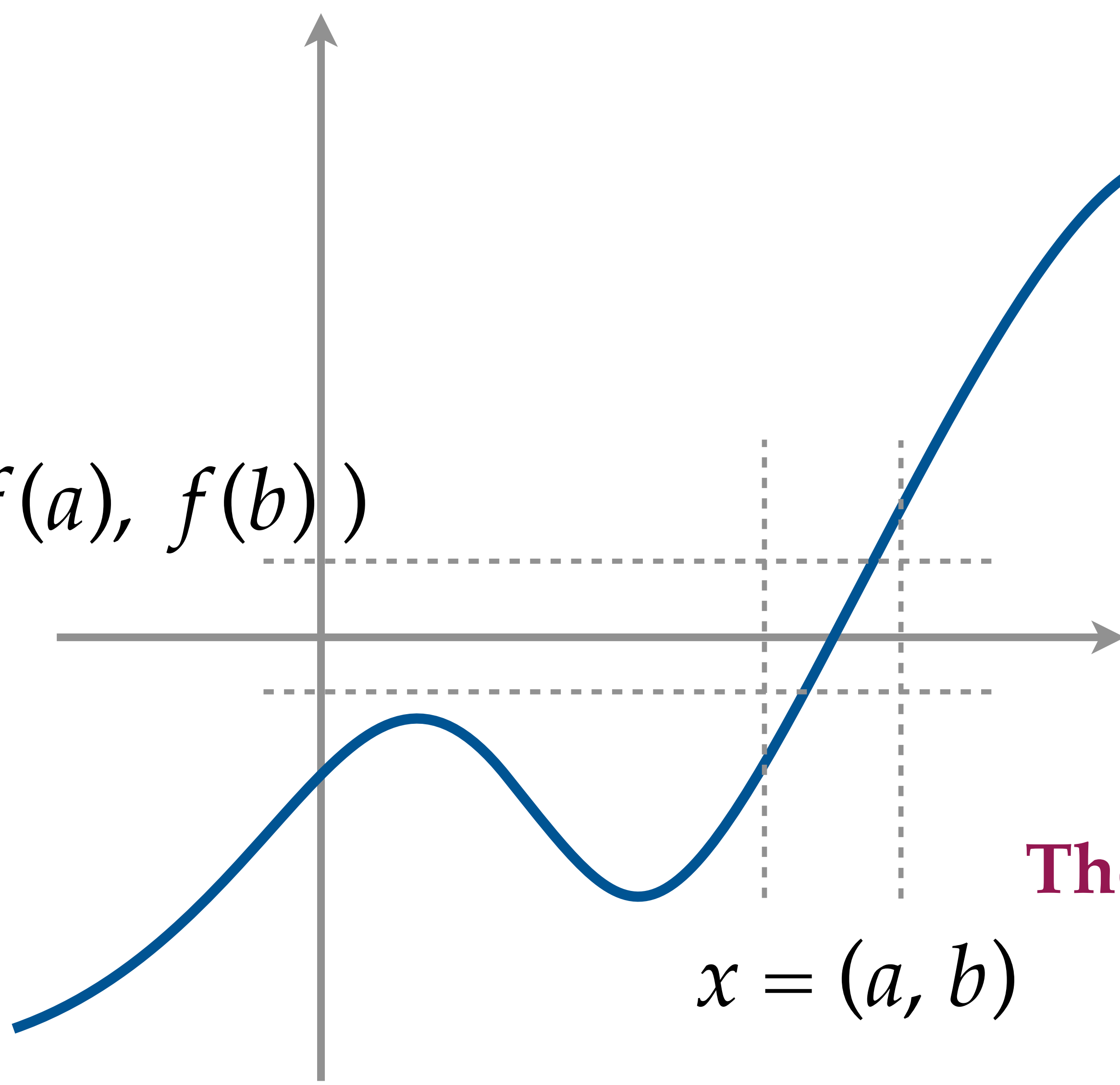


A CLASSICAL METHOD:
FIND THE ANSWER WITH YOUR EYES

First plotting the function

$y = f(x)$

$y = (\, f(a),\ f(b)\,)$



$x = (a,\, b)$

for $f(a) < 0 < f(b)$

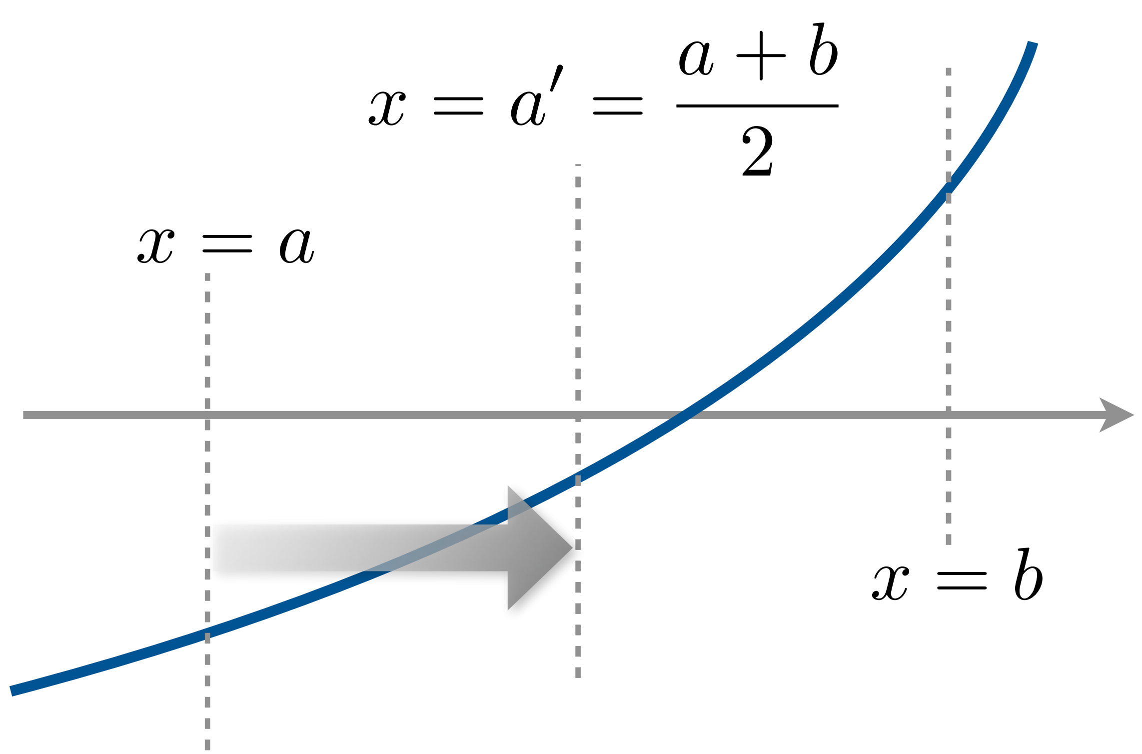
[especially if $f(a), f(b) \rightarrow 0$]

There may exist a root with $x \in (a,b)$

(Assessment: try to find an invalid example!)

LET DO SUCH A PRACTICE WITH YOUR COMPUTER

- Suppose we know that there is an solution of f(x) = 0 for x ∈(a,b), how to find the best solution by your computer?
- Surely there is an "almost" trivial algorithm: the **Bisection method**

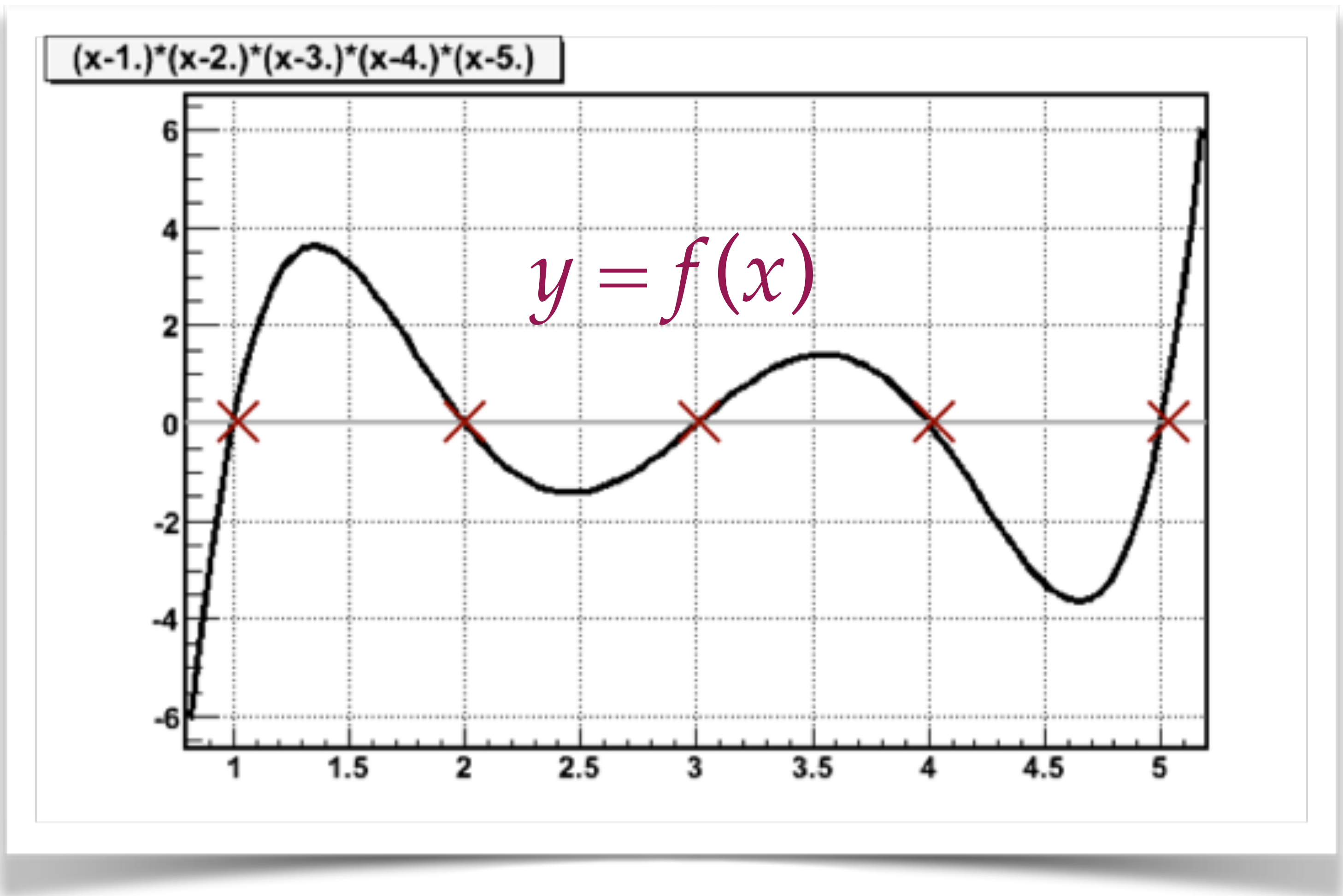


*Keep updating the boundaries with the middle point of **a** and **b**, until reaching the limited precision.*

LET'S GIVE IT A TRY!

- Suppose that we are going to solve the following equation:

$$f(x) = (x-1)\cdot(x-2)\cdot(x-3)\cdot(x-4)\cdot(x-5) = 0$$



Surely we know that there are 5 explicit solutions.

A DEMO IMPLEMENTATION

- A simple implementation of the Bisection method:

```
def f(x):
    return (x-1.)*(x-2.)*(x-3.)*(x-4.)*(x-5.)

a, b = 2.4, 3.4
fa, fb = f(a), f(b)

for step in range(50): ⇐ Let’s do maximum 50 iterations
    c = (a+b)*0.5 ⇐ Test point c – at the middle of a and b
    fc = f(c)

    print('Step: %2d, root = %.16f, diff = %.16f' % (step,c,abs(c-3.)))

    if abs(a-c)<1E-14: break ⇐ Limited precision = 10^-14

    if fc*fa>0.:
        a, fa = c, fc
    else:
        b, fb = c, fc
```

l205-example-01.py

A DEMO IMPLEMENTATION (II)

- Terminal output:

```
Step:  0, root = 2.8999999999999999, diff = 0.1000000000000001
Step:  1, root = 3.1499999999999999, diff = 0.1499999999999999
Step:  2, root = 3.0249999999999999, diff = 0.0249999999999999
Step:  3, root = 2.9624999999999999, diff = 0.0375000000000001
Step:  4, root = 2.9937499999999999, diff = 0.0062500000000001
Step:  5, root = 3.0093749999999999, diff = 0.0093749999999999
... ...
Step: 10, root = 3.0000976562499999, diff = 0.0000976562499999
... ...
Step: 20, root = 2.9999999046325683, diff = 0.0000000953674317
... ...
Step: 30, root = 3.0000000000931322, diff = 0.0000000000931322
... ...
Step: 40, root = 2.9999999999999090, diff = 0.0000000000000910
... ...
Step: 44, root = 2.9999999999999942, diff = 0.0000000000000058
Step: 45, root = 3.0000000000000084, diff = 0.0000000000000084
Step: 46, root = 3.0000000000000013, diff = 0.0000000000000013
```

HIGHER ORDER METHOD(S)

- Although this bisection algorithm sounds not so smart, but it must success (if the function is ***well behaved***).
- For higher efficiency (speed), we could go for the algorithms with an idea of higher order mathematics, e.g. **Brent's Method**:

Suppose we have three points: $(x,y) = (a, f_a), (b, f_b), (c, f_c)$

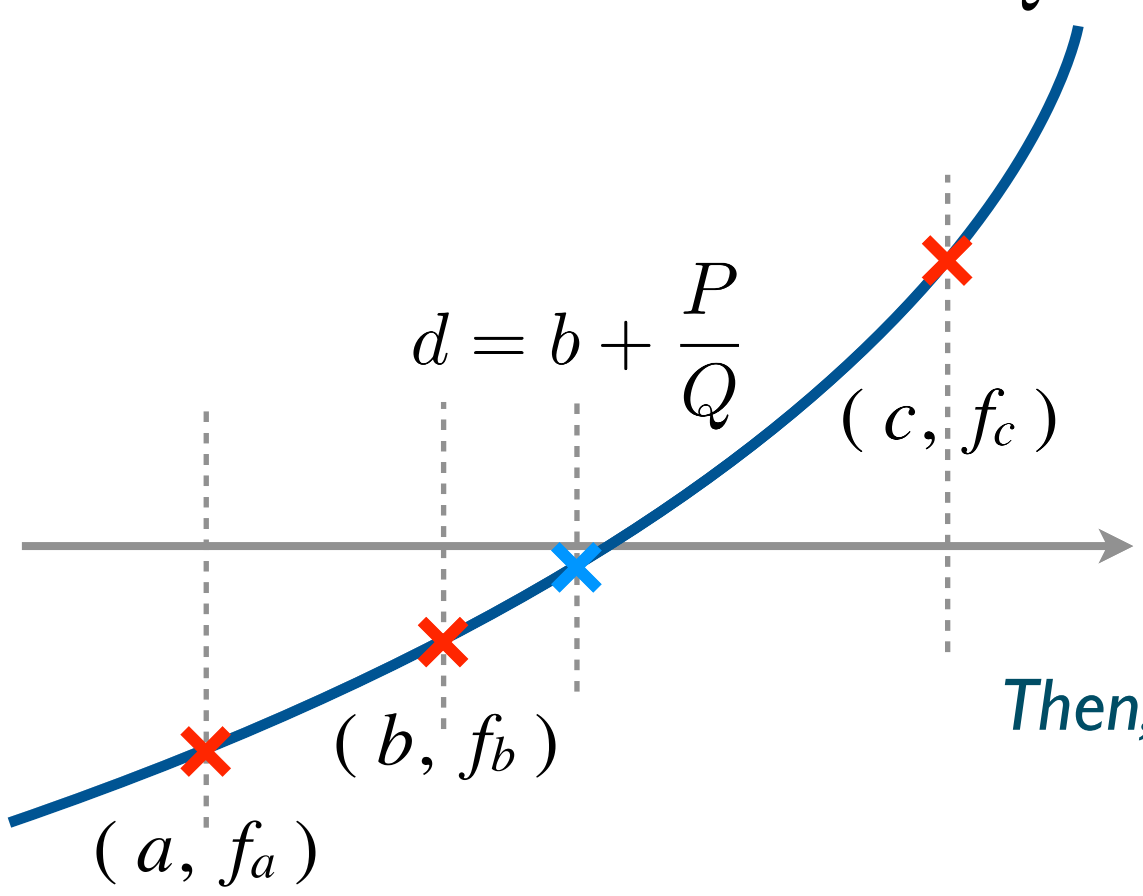
Adopt Lagrange interpolation (=3 points parabola)

$$x = \frac{(y-f_a)(y-f_b)c}{(f_c-f_a)(f_c-f_b)} + \frac{(y-f_b)(y-f_c)a}{(f_a-f_b)(f_a-f_c)} + \frac{(y-f_c)(y-f_a)b}{(f_b-f_c)(f_b-f_a)}$$

The best guess of root should be located at $\boldsymbol{y = g(x) = 0}$

BRENT'S METHOD

- Suppose **x = b** is the current best guess of root, the next best estimation is: $d = b + \frac{P}{Q}$



$$P = S[T(R-T)(c-b) - (1-R)(b-a)]$$
$$Q = (T-1)(R-1)(S-1)$$
$$R = \frac{f_b}{f_c}, \ S = \frac{f_b}{f_a}, \ T = \frac{f_a}{f_c}$$

Then, we could pick up the best three values as the new (a,b,c) for the next iteration.

LET'S TRY IT!

```
a, b, c = 2.4, 2.5, 3.4
fa, fb, fc = f(a), f(b), f(c)

for step in range(50):

    R, S, T  = fb/fc, fb/fa, fa/fc
    P  = S*(T*(R-T)*(c-b)-(1.-R)*(b-a))
    Q  = (T-1.)*(R-1.)*(S-1.)

    d  = b + P/Q
    fd = f(d)

    print('Step: %2d, root = %.16f, diff = %.16f' % (step,d,abs(d-3.)))

    if abs(b-d)<1E-14: break

    if fa*fb>0.:
        a, fa = b, fb
        b, fb = d, fd
    else:
        c, fc = b, fb
        b, fb = d, fd
```

⇐ Now we need 3 points to host the search

⇐ Simply copy the equations here!

⇐ Replace **(a, b)** with **(b, d)**

⇐ Replace **(c, b)** with **(b, d)**

l205-example-02.py (partial)

LET'S TRY IT! (II)

- Terminal output is like this:

```
Step:  0, root = -5.206462747862000O, diff = 8.206462747862000O
Step:  1, root =  2.969342622172016 3, diff = 0.030657377827983 7
Step:  2, root =  3.006679882610452 8, diff = 0.006679882610452 8
Step:  3, root =  2.999852447241841 1, diff = 0.000147552758158 9
Step:  4, root =  3.000000037829857 5, diff = 0.000000037829857 5
Step:  5, root =  2.999999999999953 4, diff = 0.000000000000046 6
Step:  6, root =  3.000000000000000 0, diff = 0.000000000000000 0
Step:  7, root =  3.000000000000000 0, diff = 0.000000000000000 0
```

- Well, it does happen: it does **NOT** guarantee the next step will always gives a better guess of the root, especially if we approximate the function by a 2nd order parabola.
- Alternative fix: replace the next guess by **Bisection method**, if the guess is bad / poor.

A FAIL-SAFE CODE

- Simply fix the value of test point `(d,fd)` with **Bisection method** if the resulting values are bad:

```
d  = b + P/Q
fd = f(d)

if (d-a)*(d-c)>0. or abs(fd)>abs(fb):
    if fa*fb>0.: d = (b+c)*0.5
    else:        d = (a+b)*0.5
    fd = f(d)

print('Step: %2d, root = %.16f, diff = %.16f' % (step,d,abs(d-3.)))
```

l205-example-02a.py (partial)

```
Step:  0, root = 2.9500000000000002, diff = 0.0499999999999998
Step:  1, root = 3.0169811828014468, diff = 0.0169811828014468
Step:  2, root = 2.9993946939327074, diff = 0.0006053060672926
Step:  3, root = 3.0000006446632410, diff = 0.0000006446632410
Step:  4, root = 2.9999999999917120, diff = 0.0000000000082880
Step:  5, root = 3.0000000000000000, diff = 0.0000000000000000
Step:  6, root = 3.0000000000000000, diff = 0.0000000000000000
```

⇐ All good!

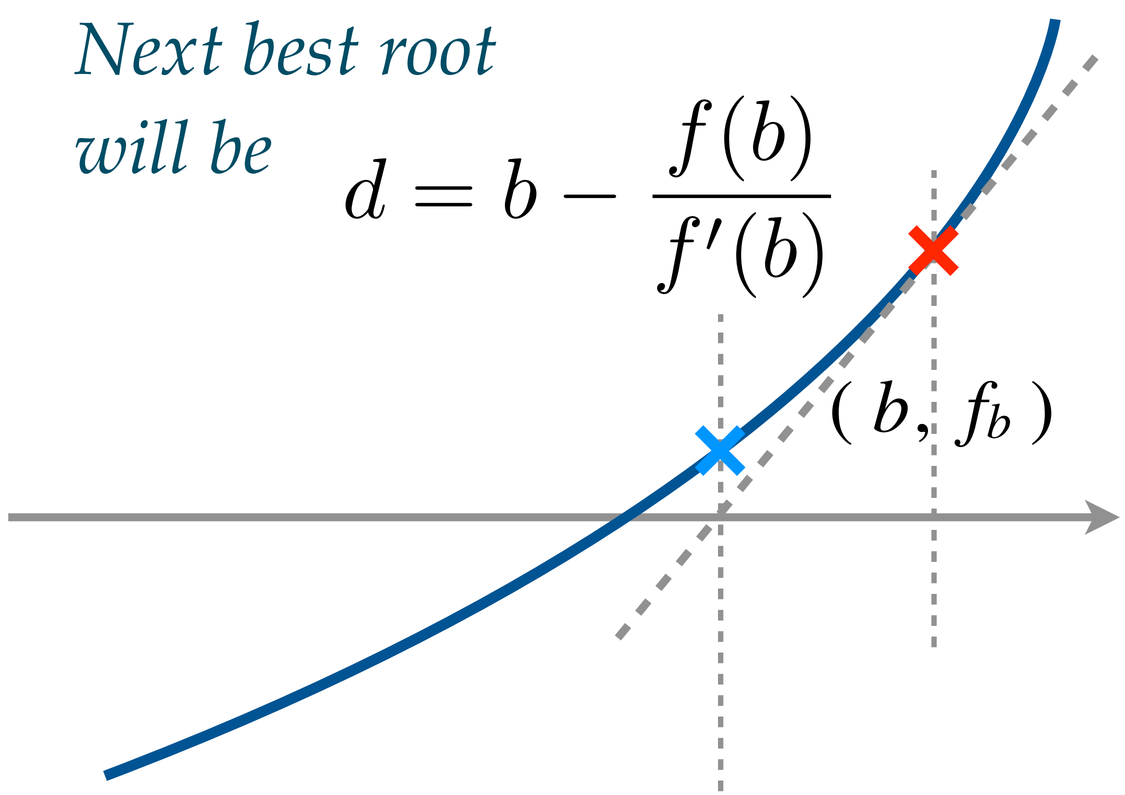
ALGORITHM WITH DERIVATIVE: NEWTON'S METHOD (NEWTON-RAPHSON)

- Well, where is the beloved method, which we have learned in calculus course?

$$f(x+\delta) \approx f(x) + f'(x)\delta + \frac{f''(x)}{2}\delta^2 + \ldots$$

Take out the 2nd order term

Next best root will be $d = b - \dfrac{f(b)}{f'(b)}$



current best root is at $x = b$

IMPLEMENTATION: NEWTON'S METHOD

```
def fp(x):
    return (x-2.)*(x-3.)*(x-4.)*(x-5.) + \
           (x-1.)*(x-3.)*(x-4.)*(x-5.) + \
           (x-1.)*(x-2.)*(x-4.)*(x-5.) + \
           (x-1.)*(x-2.)*(x-3.)*(x-5.) + \
           (x-1.)*(x-2.)*(x-3.)*(x-4.)

a, b, c = 2.4, 2.5, 3.4
fa, fb, fc = f(a), f(b), f(c)

for step in range(50):

    delta = -fb/fp(b)
    d  = b + delta
    fd = f(d)

    if (d-a)*(d-c)>0. or abs(fd)>abs(fb):
        if fa*fb>0.: d = (b+c)*0.5
        else:        d = (a+b)*0.5
        fd = f(d)

    print('Step: %2d, root = %.16f, diff = %.16f' % (step,d,abs(d-3.)))
    if abs(b-d)<1E-14: break

    b, fb = d, fd
```

⇐ Analytical solution

⇐ Keep the protection as in the Bisection method

l205-example-03.py (partial)

(SUPER-)FAST CONVERGING!

- Terminal output:

```
Step:   0, root = 2.9500000000000002, diff = 0.0499999999999998
Step:   1, root = 3.0003151394705090, diff = 0.0003151394705090
Step:   2, root = 2.9999999999217564, diff = 0.0000000000782436
Step:   3, root = 3.0000000000000000, diff = 0.0000000000000000
Step:   4, root = 3.0000000000000000, diff = 0.0000000000000000
```

⇑ Just 3–4 steps!

- **Q:** Why not to use **the numerical derivatives**?
- **A:** As we have discussed before, it's very hard to have precise numerical solution for the derivatives. In this case the solution will be limited by the best precision of the derivative calculation. It's generally not a recommended way (but still "doable").

INTERMISSION

- With Newton's method:
  - What will happen if you remove the failed safe protection (the block of using Bisection method)?
  - Try to run the calculation with numerical derivative, how good is the solution?

```python
def fp(x):    ⇐You can try this by yourself!
    h = 1E-5
    return (f(x+h/2.)-f(x-h/2.))/h
```

l205-example-03a.py (partial)

- Try to find a not-working-so-well problem!

SOME MORE PRACTICAL EXAMPLES?

- Let's implement a function with Newton's method to calculate square-root and cubic-root. This is one of the places this method can do the work easily!
- The usual square-root function is `sqrt()`, and we can only use the `pow()` function or the `**` operator to calculate cubic-root.
- If we are looking for the square-(cubic-) root of a real number R, it's equivalent to find the root of

$$f(x) = x^2 - R \quad \text{or} \quad f(x) = x^3 - R$$

The corresponding first derivatives are

$$f'(x) = 2x \quad \text{or} \quad f'(x) = 3x^2$$

The implement the code should be very easy!

QUICK & SIMPLE CODE

```
def squareroot(R):
    fsq = lambda x:x*x-R
    fsqp = lambda x:2.*x

    a, b, c = 0., R*0.5, R
    fa, fb, fc = fsq(a), fsq(b), fsq(c)
    for step in range(50):
        delta = -fb/fsqp(b)
        d  = b + delta
        fd = fsq(d)

        if abs(b-d)<1E-14: return d
        b, fb = d, fd

def cubicroot(R):
    fcb = lambda x:x*x*x-R
    fcbp = lambda x:3.*x*x

    a, b, c = 0., R*0.5, R
    fa, fb, fc = fcb(a), fcb(b), fcb(c)
    for step in range(50):
        delta = -fb/fcbp(b)
        d  = b + delta
        fd = fcb(d)

        if abs(b-d)<1E-14: return d
        b, fb = d, fd
```

⇐ local functions

l205-example-04.py (partial)

Basically the implementations are the same; the only difference are the local functions `fsq()` and `fsqp()`.

LET'S TRY THE FUNCTIONS!

- This is almost a trivial task:

```
R = 1234.

print('root = %.16f, diff = %.16f' % \
    (squareroot(R),abs(R**0.5-squareroot(R))))

print('root = %.16f, diff = %.16f' % \
    (cubicroot(R),abs(R**(1./3.)-cubicroot(R))))
```

l205-example-04.py (partial)

```
root = 35.1283361405005934, diff = 0.0000000000000000
root = 10.7260146688273235, diff = 0.0000000000000000
```

Surely this code is very slow if we compare to the standard operator, but this is a very good example that almost all the math functions can be implemented in a similar way!

USE THE FUNCTIONS FROM SCIPY

- Everything is under **scipy.optimize**:

http://docs.scipy.org/doc/scipy/reference/optimize.html

SciPy.org Sponsored By ENTHOUGHT

Scipy.org | Docs | SciPy v1.0.0 Reference Guide | index | modules | next | previous

Optimization and root finding (scipy.optimize)

Table Of Contents

- Optimization and root finding (`scipy.optimize`)

Optimization

Root finding

Scalar functions

brentq(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in a bracketing interval using Brent's method.
brenth(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of f in [a,b].
ridder(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in an interval.
bisect(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of a function within an interval.
newton(func, x0[, fprime, args, tol, ...])	Find a zero using the Newton-Raphson or secant method.

You can see some familiar names here!

USING THE SUPER EASY SCIPY FUNCTIONS

- Just import the **scipy.optimize** and call the corresponding method:

```
import scipy.optimize as opt

def squareroot(R):
    fsq = lambda x:x*x-R
    fsqp = lambda x:2.*x

    return opt.newton(fsq,R*0.5,fsqp) ⇐ Just call it!

R = 1234.

print('root = %.16f, diff = %.16f' % \
    (squareroot(R),abs(R**0.5-squareroot(R))))
```

l205-example-05.py

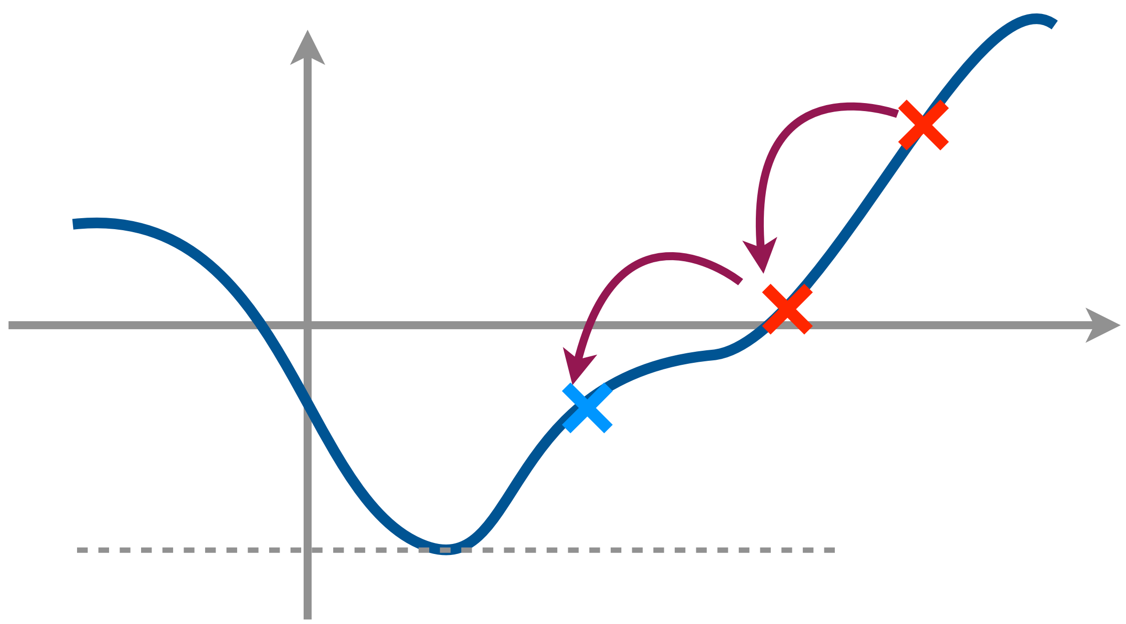
```
root = 35.1283361405005934, diff = 0.0000000000000000
```

MINIMIZATION OR MAXIMIZATION

- Method in calculus – **find the zero first derivative**:

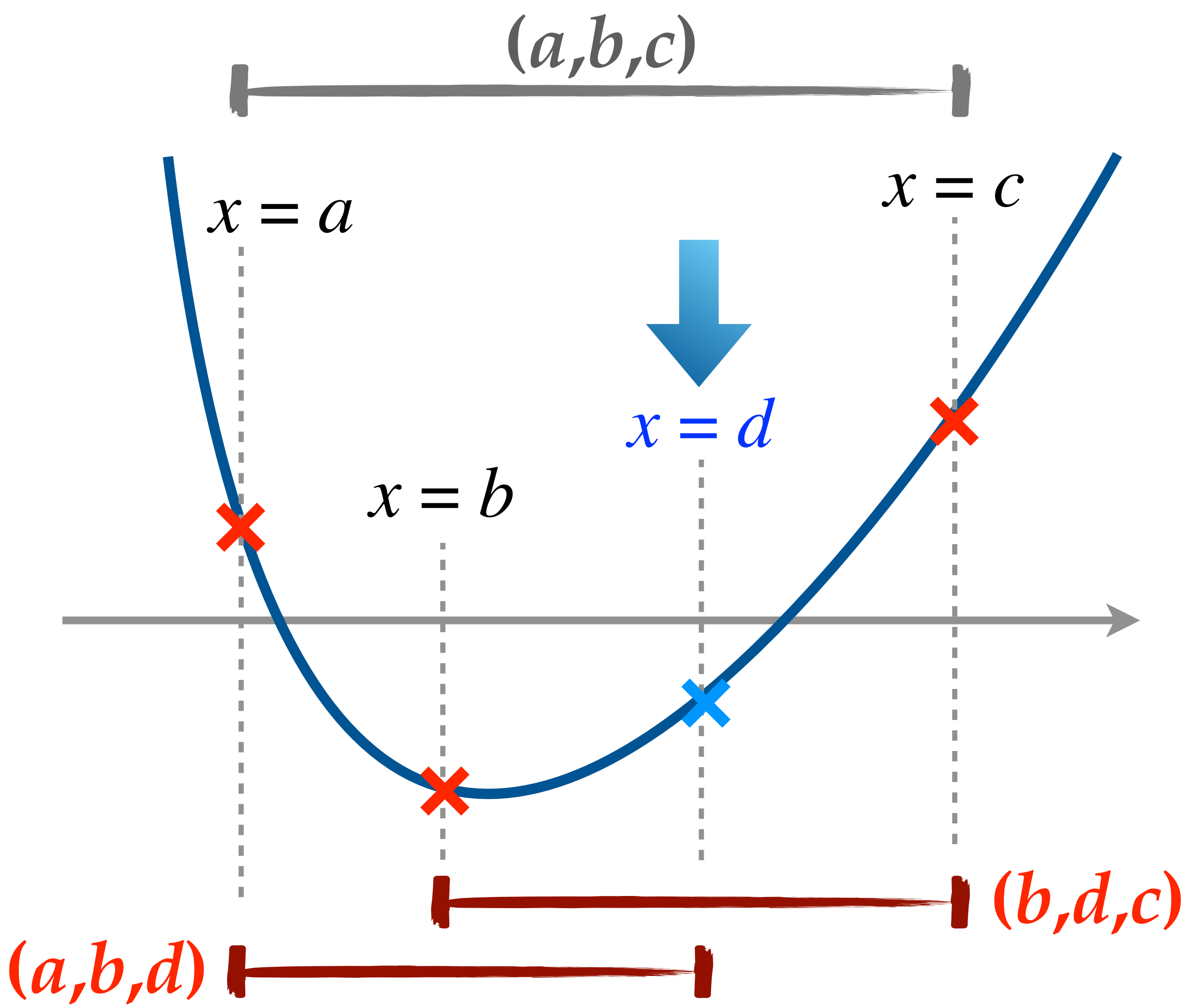
$$f'(x) = 0 \quad \rightarrow \quad x = ?$$

- How about the numerical method?
- Yep, you can probably already apply what we learned from the previous section, to find the root of $f'(x) = 0$ if we know the first derivative already.
- If not, this is what we are going to discuss now.



ONE DIMENSIONAL SEARCH IN A BRACKET

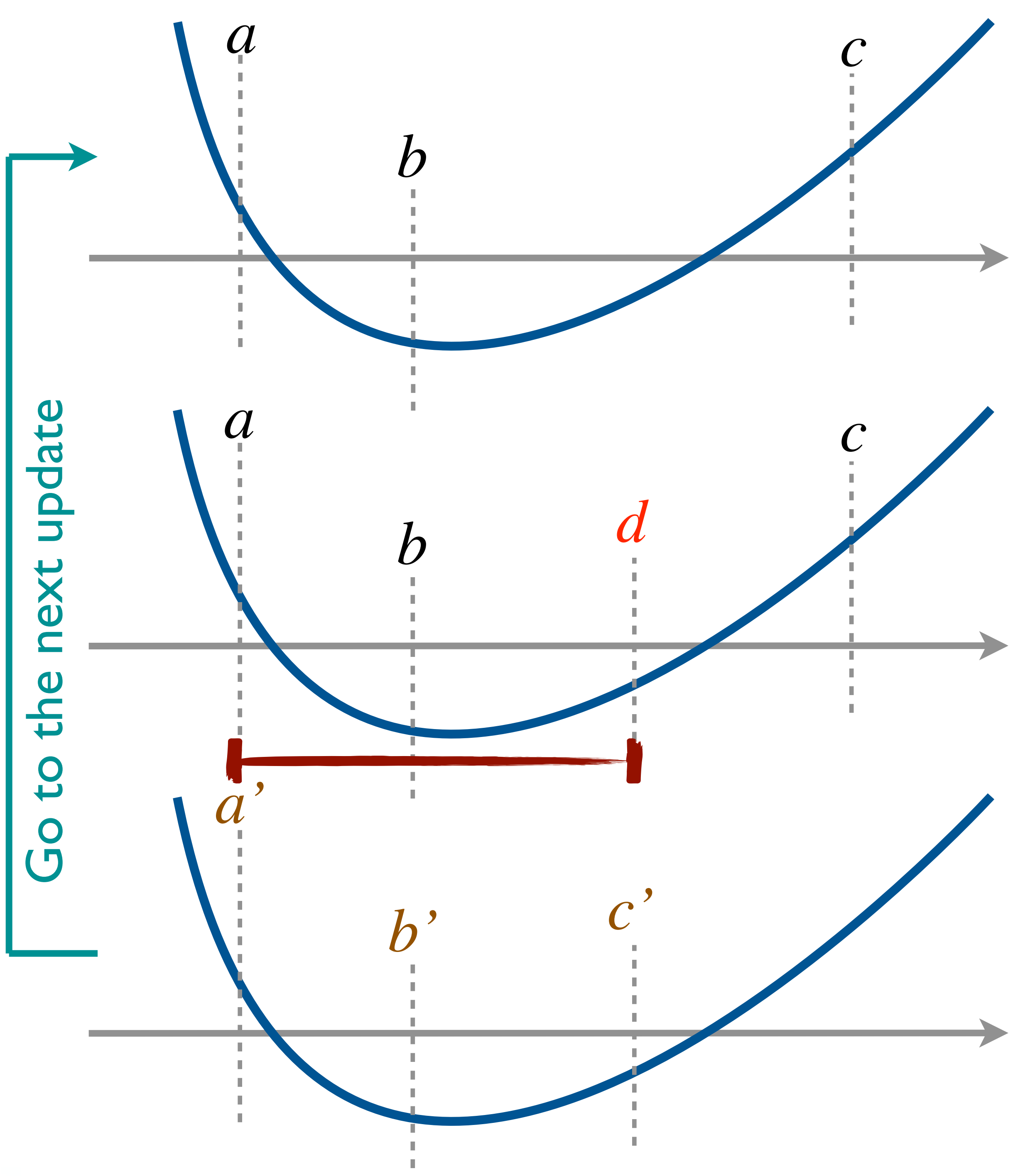
- This method is very simple: if we have a bracket (a,b,c), and $f(b) < f(a), f(c)$, and b is the current best minimum:



Keep updating the bracket by replacing (a,b,c) with (a,b,d) or (b,d,c) until a desired precision.

We always need to keep $f(b) < f(a)$ and $f(b) < f(c)$ to ensure we have at least a minimum in the interval.

1D SEARCH – THE STEPS



- Initial bracket **(a,b,c)**
- If **|b-c|>|a-b|**, find a new test point d in **[b,c]**
- If **f(b) < f(d)**, keep **b** as the current best estimation of the minimum point.
- Update the bracket accordingly: **c' = d**
- Go to the next update

A QUICK IMPLEMENTATION

```
def f(x):
    return (x-0.5)*(x-0.5)*(x-10.)*(x-10.)  ⇐ A function with 2 obvious
                                              minimal points
FRAC = 0.38197  ⇐ Magic number!
a, c = 0.0, 2.0
fa, fc = f(a), f(c)
b = a+(c-a)*FRAC
fb = f(b)

for step in range(150):

    if abs(a-b)>abs(c-b): d = b+(a-b)*FRAC  ⇐ Insert a new testing point,
    else:                 d = b+(c-b)*FRAC    between either (a,b) or (b,c)
    fd = f(d)

    print('Step: %2d, root = %.16f, diff = %.16f' % (step,d,abs(d-0.5)))
    if abs(b-d)<1E-14: break

    if fd<fb:
        b, d = d, b         ⇐ exchange b and d, keep b as the best solution as always
        fb, fd = fd, fb

    if (d-b)*(a-b)>0: a, fa = d, fd
    else:             c, fc = d, fd
```

l205-example-06.py

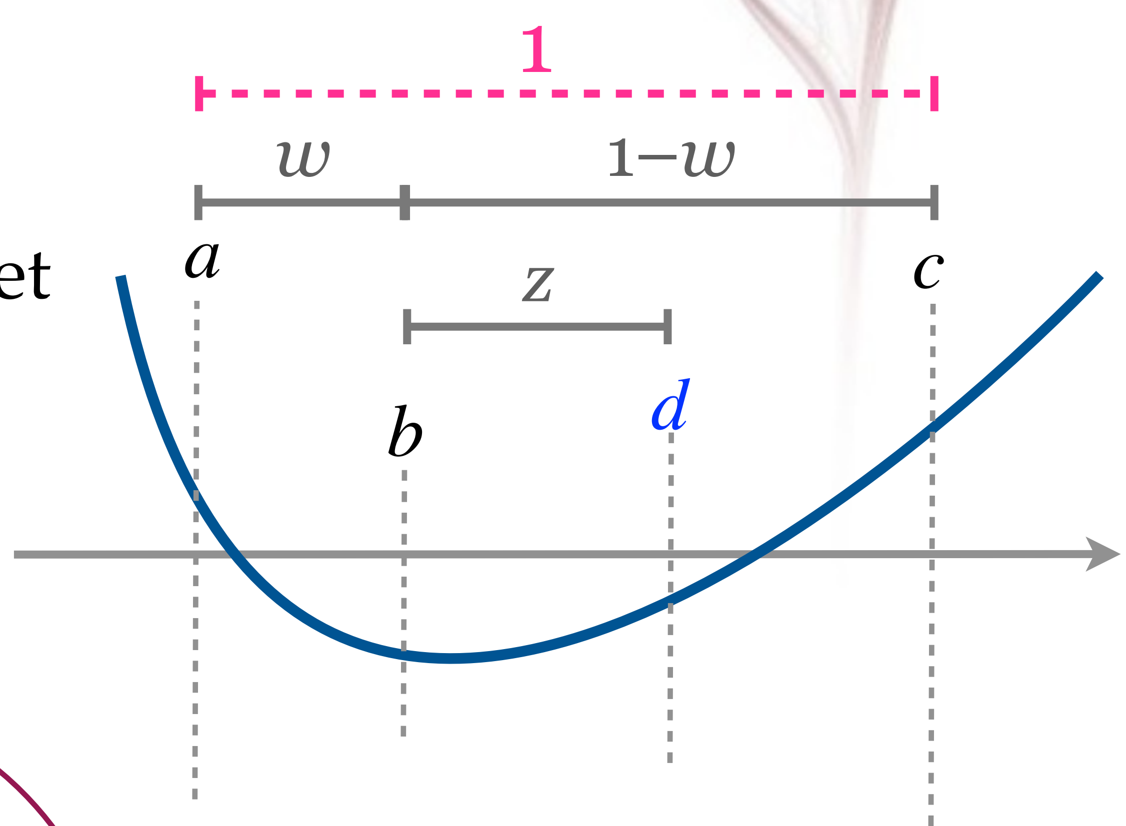
THE RESULTS

- Terminal output:

```
Step:  0, root = 1.2360778381999999, diff = 0.7360778381999999
... ...
Step: 10, root = 0.4946110292293492, diff = 0.0053889707706508
... ...
Step: 20, root = 0.4999668808722842, diff = 0.0000331191277158
... ...
Step: 30, root = 0.4999995815191064, diff = 0.0000004184808936
... ...
Step: 40, root = 0.5000000029995387, diff = 0.0000000029995387
... ...
Step: 50, root = 0.4999999999885979, diff = 0.0000000000114021
... ...
Step: 60, root = 0.4999999999997671, diff = 0.0000000000002329
Step: 61, root = 0.4999999999999878, diff = 0.0000000000000122
Step: 62, root = 0.5000000000000400, diff = 0.0000000000000400
Step: 63, root = 0.4999999999999556, diff = 0.0000000000000444
Step: 64, root = 0.5000000000000078, diff = 0.0000000000000078
Step: 65, root = 0.5000000000000201, diff = 0.0000000000000201
Step: 66, root = 0.5000000000000001, diff = 0.0000000000000001
```

WHY 0.38197?

- A funny number used in the decision of the position of **d**? Why?
- Let's look at the configuration:



- Every time, we could shrink the bracket from **1** to **(w+z)** or **(1-w)**
- In order to avoid the worst case, let's simply force them to be the same:

$$w + z = 1 - w$$

- Usually it would be the optimal if we preserve the same ***"shrinking rate"***:

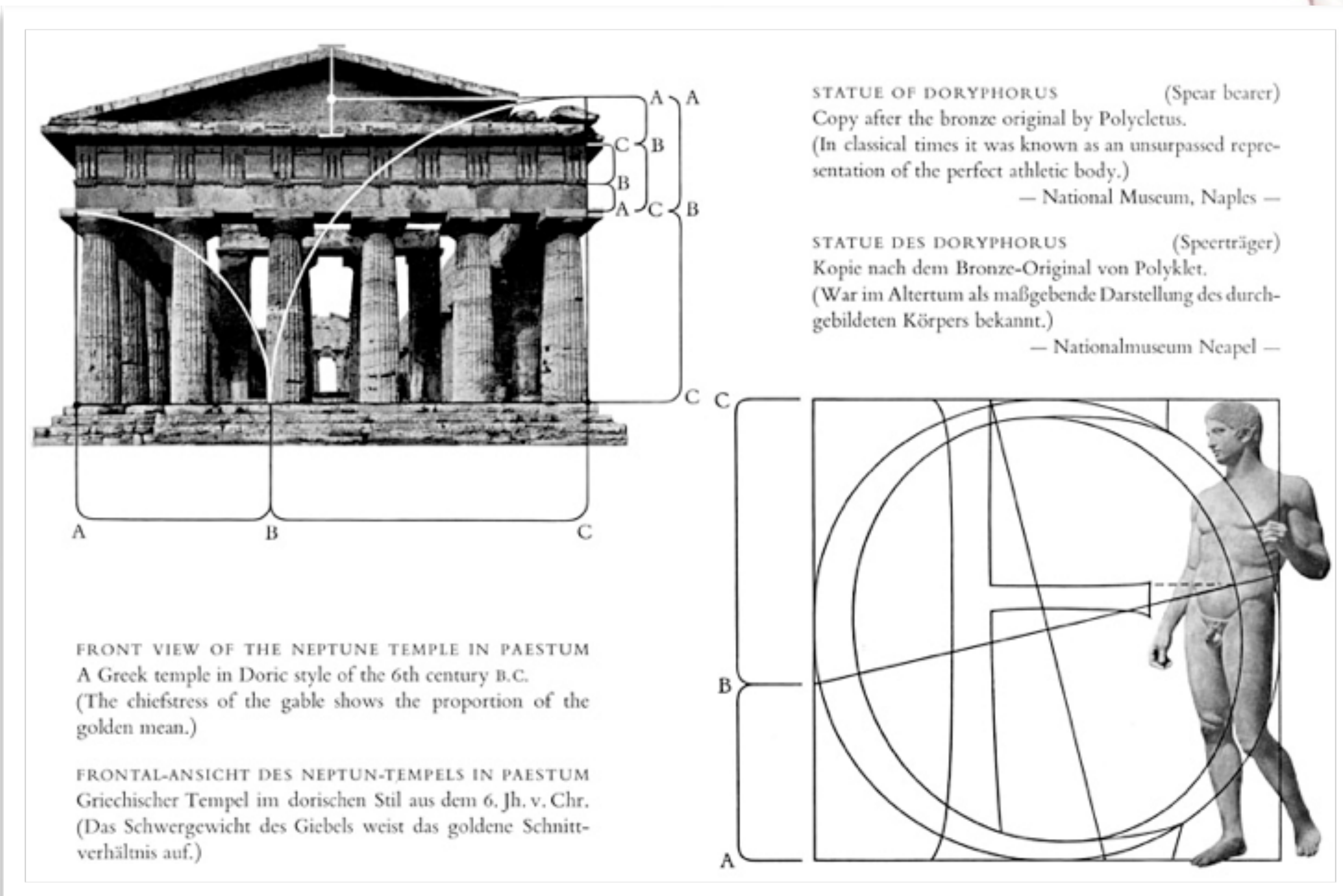
$$\frac{z}{1-w} = w$$

Then

$$w = \frac{3-\sqrt{5}}{2} \approx 0.38197$$

WHY 0.38197? (II)

- Actually, this is nothing but the **golden ratio**:



STATUE OF DORYPHORUS (Spear bearer)
Copy after the bronze original by Polycletus.
(In classical times it was known as an unsurpassed representation of the perfect athletic body.)
— National Museum, Naples —

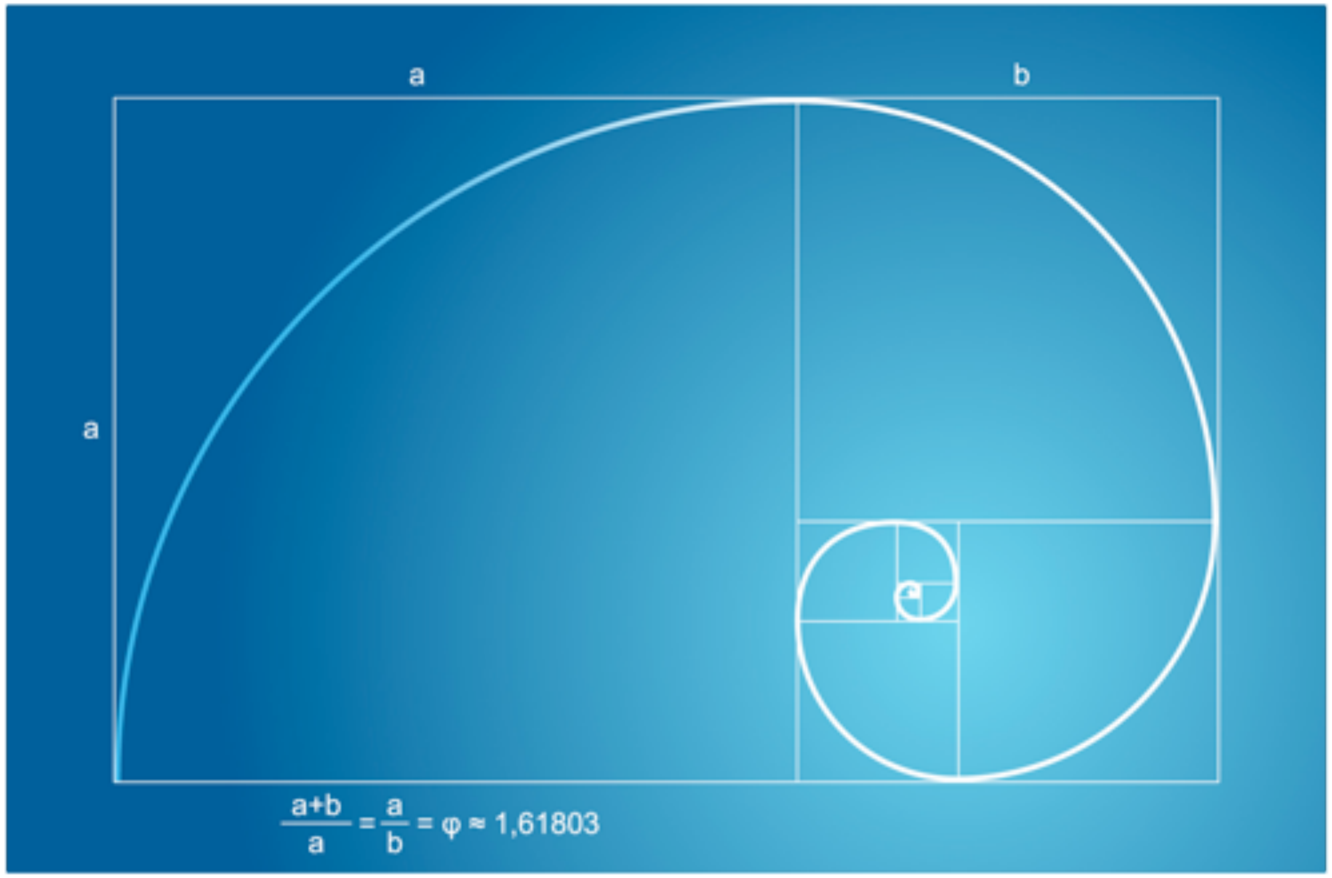
STATUE DES DORYPHORUS (Speerträger)
Kopie nach dem Bronze-Original von Polyklet.
(War im Altertum als maßgebende Darstellung des durchgebildeten Körpers bekannt.)
— Nationalmuseum Neapel —

FRONT VIEW OF THE NEPTUNE TEMPLE IN PAESTUM
A Greek temple in Doric style of the 6th century B.C.
(The chiefstress of the gable shows the proportion of the golden mean.)

FRONTAL-ANSICHT DES NEPTUN-TEMPELS IN PAESTUM
Griechischer Tempel im dorischen Stil aus dem 6. Jh. v. Chr.
(Das Schwergewicht des Giebels weist das goldene Schnittverhältnis auf.)

WHY 0.38197? (III)

- The nominal golden section is derived from



$$\phi = \frac{a+b}{a} = \frac{a}{b} \approx 1.61803$$

$$\text{And } 1 - \frac{1}{\phi} \approx 0.38197$$

So this minimum finding method is called **Golden Section Search**.

My comments: unfortunately I'm not able to prove this is the best ratio for a generic 1D minimum finding; but it's not a bad number in principle.

PARABOLIC INTERPOLATION: BRENT'S METHOD

- As we has shown in the previous half of this lecture, the parabolic interpolation (the Brent's method) shows a good solution of efficiency for 1D root finding.
- We are also able to do the same thing here:

Suppose we have three points: $(x,y) = (a,\ f_a), (b,\ f_b), (c,\ f_c)$

The minimum value of the function $f(x)$ is located at

$$d = b - \frac{1}{2} \cdot \frac{(b-a)^2[f_b - f_c] - (b-c)^2[f_b - f_a]}{(b-a)[f_b - f_c] - (b-c)[f_b - f_a]}$$

Updating term for next iteration

Current best solution

You may try to derive this formula by yourself!

EXAMPLE CODE

```
FRAC = 0.38197
a, c = 0.0, 2.0          ⇐ The same initial bracket as the golden section search
fa, fc = f(a), f(c)
b = a+(c-a)*FRAC
fb = f(b)

for step in range(150):
    P = (b-a)*(b-a)*(fb-fc) - (b-c)*(b-c)*(fb-fa)
    Q = (b-a)*(fb-fc) - (b-c)*(fb-fa)      ⇐ Estimate d with the
    d = b - 0.5*P/Q                           formula given above.

    if (d-a)*(d-c)>0.:
        if abs(a-b)>abs(c-b): d = b+(a-b)*FRAC
        else:                 d = b+(c-b)*FRAC   ⇐ Fail-safe protection

    fd = f(d)
    print('Step: %2d, root = %.16f, diff = %.16f' % (step,d,abs(d-0.5)))
    if abs(b-d)<1E-14: break

    if fd<fb:
        b, d = d, b
        fb, fd = fd, fb   ⇐ keep b as the best solution as always

    if (d-b)*(a-b)>0: a, fa = d, fd
    else:             c, fc = d, fd
```

l205-example-07.py (partial)

THE OUTPUTS

- Surely the **converging speed** is much faster than the simple golden section searches:

```
Step:  0, root = 0.564541176882796З, diff = 0.064541176882796З
```

You may notice that, finding the minimum is more difficult than finding the root!

MINIMUM FINDING WITH DERIVATIVES

- This is ***pretty tricky***: if you know the exact form of the **first derivative**, then a simply root finding code can already give you the maximum and minimum points.
- If we just want to apply the Newton's method, we need to know **the exact form of second derivative**.

Next best root is given by $d = b - \dfrac{f(b)}{f'(b)}$

Next best minimum/maximum is given by $d = b - \dfrac{f'(b)}{f''(b)}$

EXAMPLE CODE

```
def fp(x):
    return 2.*(x-0.5)*(x-10.)*(x-10.)+2.*(x-0.5)*(x-0.5)*(x-10.)
def fpp(x):
    return 2.*(x-10.)*(x-10.)+8.*(x-0.5)*(x-10.)+2.*(x-0.5)*(x-0.5)

FRAC = 0.38197
a, c = 0.0, 2.0
fa, fc = f(a), f(c)       ⇐ Again, the same initial bracket!
b = a+(c-a)*FRAC
fb = f(b)

for step in range(150):
    delta = -fp(b)/fpp(b)
    d = b + delta       ⇐ update b,d according to Newton's method

    if (d-a)*(d-c)>0.:
        if abs(a-b)>abs(c-b): d = b+(a-b)*FRAC    ⇐ Fail-safe protection
        else:                 d = b+(c-b)*FRAC

    fd = f(d)
    print('Step: %2d, root = %.16f, diff = %.16f' % (step,d,abs(d-0.5)))
    if abs(b-d)<1E-14: break

    b = d
```

l205-example-08.py (partial)

THE PERFORMANCE

- The converging speed is **VERY GOOD**. We need only~5 steps instead of 23 or 6x iterations. The second derivative is required!

```
Step:  0, root = 0.4747183508530082, diff = 0.0252816491469918
Step:  1, root = 0.4998006350485492, diff = 0.0001993649514508
Step:  2, root = 0.4999999874497394, diff = 0.0000000125502606
Step:  3, root = 0.4999999999999999, diff = 0.0000000000000001
Step:  4, root = 0.5000000000000000, diff = 0.0000000000000000
```

l205-example-08.py (output)

- Alternatively, one can adopt **Brent's method** for root finding on first derivate: *(Well, it's not too bad at all!)*

```
Step:  0, root = 0.4358830239633310, diff = 0.0641169760366690
Step:  1, root = 0.5013516961302908, diff = 0.0013516961302908
Step:  2, root = 0.4999956151890250, diff = 0.0000043848109750
Step:  3, root = 0.5000000000658166, diff = 0.0000000000658166
Step:  4, root = 0.5000000000000000, diff = 0.0000000000000000
Step:  5, root = 0.5000000000000000, diff = 0.0000000000000000
```

l205-example-08a.py (output)

INTERMISSION

- Try to use the SciPy implementation of Brent's method, `scipy.optimize.brentq()` to solve the same problem in l205-example-02.py and see what you get?
- The **golden section search** – what will happen if you do not use the "golden" ratio but a whatever number, such as 0.5? Is it better or worse in terms of converging speed?

MULTIDIMENSIONAL MINIMIZATION (COMMENTS)

- If we want to find the minimum point in multi-dimensional space, it's much harder than our those 1D examples given above.
- Many numerical algorithms have been developed in order to find the minimum point for various problems.
  (or, the best algorithm could be question dependent.)
- Some named methods: **Downhill method, Conjugate gradient, Steepest Descent, Simplex method, Quasi-Newton method,** etc.
- We will not discuss about how to write the code by yourself, instead, we are going to use **the standard tools in SciPy** directly!

BACK TO SCIPY

- The generic minimizer **scipy.optimize.minimize()** is shown below:
  http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize

Scipy.org | Docs | SciPy v1.0.0 Reference Guide | Optimization and root finding (`scipy.optimize`) | index | modules | next

scipy.optimize.minimize

scipy.optimize.minimize(*fun, x0, args=(), method=None, jac=None, hess=None, hessp=None, bounds=None, constraints=(), tol=None, callback=None, options=None*) [source]

Minimization of scalar function of one or more variables.

In general, the optimization problems are of the form:

```
minimize f(x) subject to

g_i(x) >= 0,  i = 1,...,m
h_j(x)  = 0,  j = 1,...,p
```

where x is a vector of one or more variables. `g_i(x)` are the inequality constraints. `h_j(x)` are the equality constrains.

Optionally, the lower and upper bounds for each element in x can also be specified using the *bounds* argument.

Parameters: fun : *callable*

The objective function to be minimized. Must be in the form `f(x, *args)`. The optimizing argument, `x`, is a 1-D array of points, and `args` is a tuple of any additional fixed parameters needed to completely specify the function.

Previous topic

Optimization and root fin
(`scipy.optimize`)

Next topic

scipy.optimize.minimize_

Let's see a super simple example for calling this tool!

ONE LINE TO FIND THE MINIMUM

- An example code for calling the default minimizer ("BFGS"= a quasi-Newton method by Broyden-Fletcher-Goldfarb-Shanno).

```
import numpy as np
import scipy.optimize as opt

def f(x):              ⇓ A 3D function with obvious minimal point of (1,2,3)
    return (x[0]-1.)**2+(x[1]-2.)**2+(x[2]-3.)**2

x_init = np.array([0.5,0.5,0.5]) ⇐ initial values

res = opt.minimize(f,x_init)

if res.success:
    print('The resulting vector:',res.x)
```

l205-example-09.py

```
The resulting vector:
[ 1.          1.99999991   3.00000009]
```

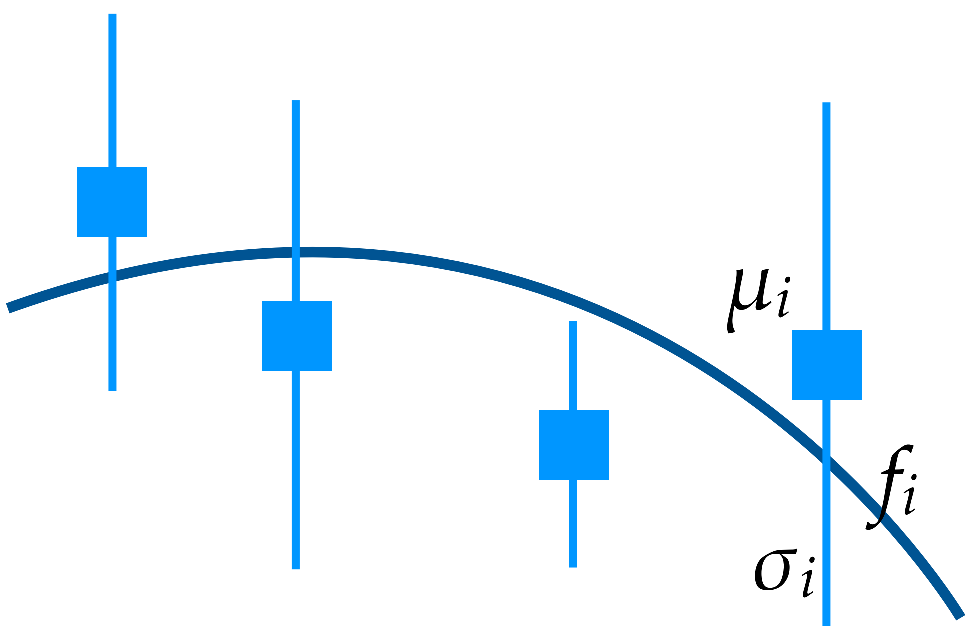
l205-example-09.py (output)

A PRACTICAL EXAMPLE: LEAST-SQUARE (χ^2) FIT

- The best results can be obtained by minimizing a $\chi 2$ value for **N independent measurements**:

$$\chi^2 = \sum_i^N \frac{(f_i - \mu_i)^2}{\sigma_i^2}$$

f_i: expected value of the model
μ_i: i^{th} measurement
σ_i: uncertainty of i^{th} measurement



Keeping updating those parameters (α,β,γ,...) until the **best (smallest)** χ^2 value is reached.

$$f_i = f(x_i; \alpha, \beta, \gamma, \ldots)$$

LET'S GET SOME REAL DATA POINTS

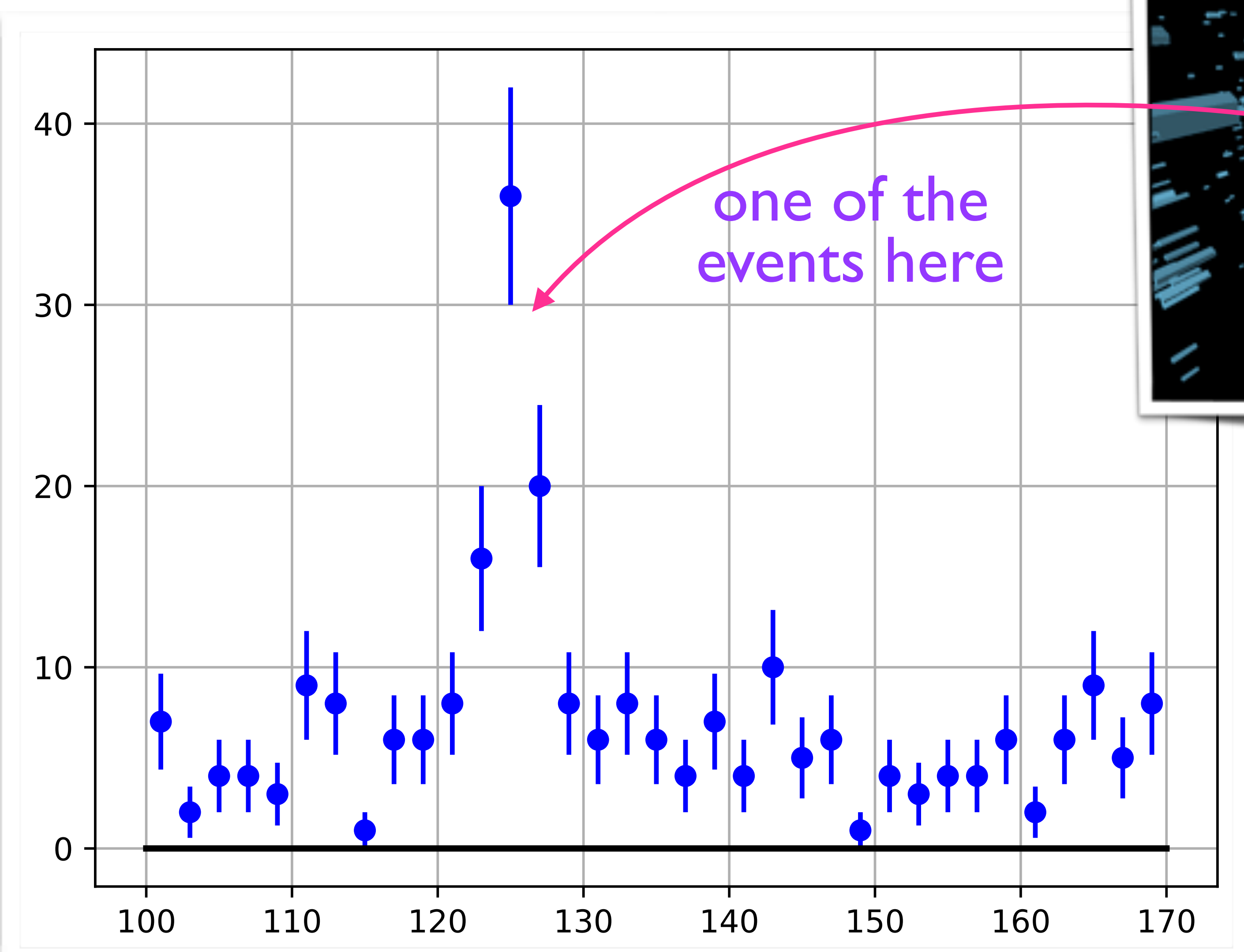
- One can start with storing the data as numpy arrays and make a simple plot with error bar:

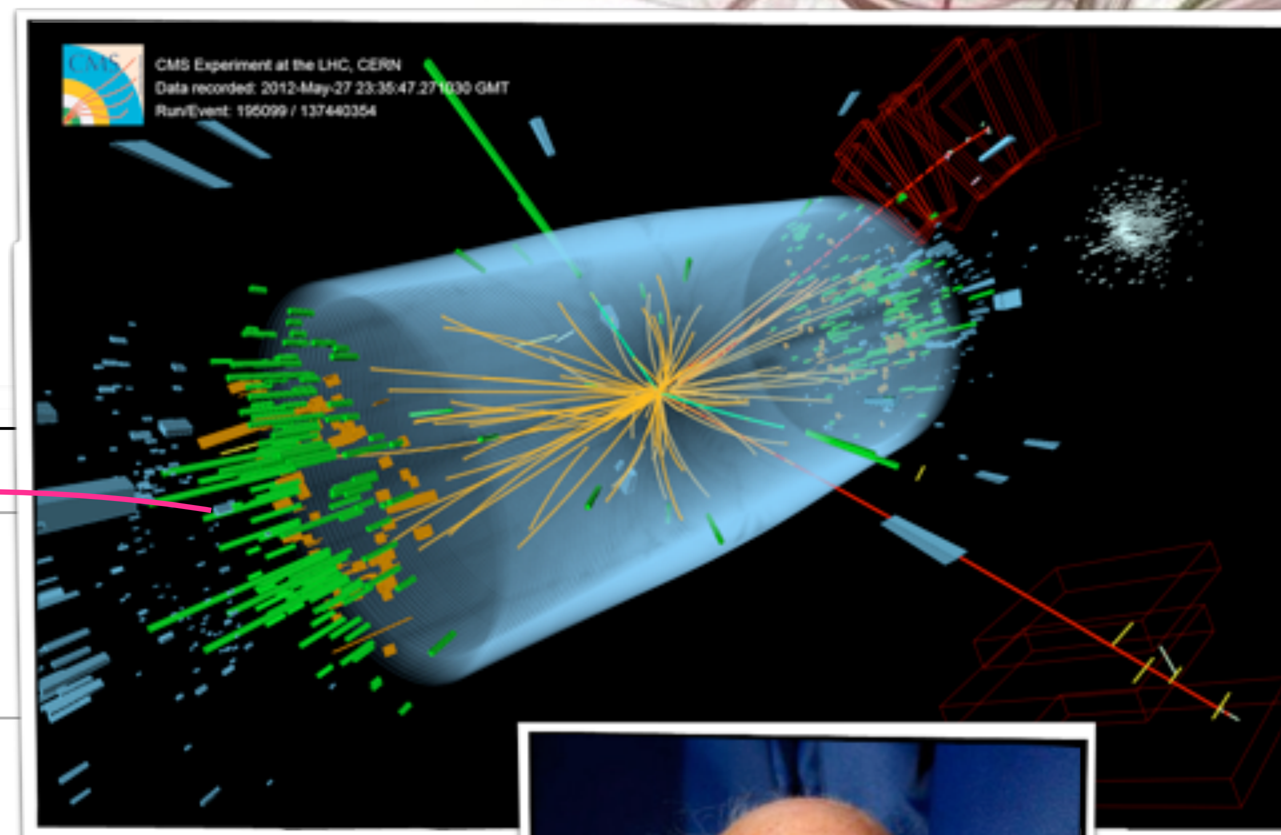
```
import numpy as np
import matplotlib.pyplot as plt

xmin, xmax, xbinwidth = 100., 170., 2.
vx = np.linspace(xmin+xbinwidth/2,xmax-xbinwidth/2,35) ⇐ x axis
vy = np.array(
[7,2,4,4,3,9,8,1,6,6,8,16,36,20,8,6,8,6,4,7,      ⇐ y axis: simple of
 4,10,5,6,1,4,3,4,4,6,2,6,9,5,8],dtype='float64') counting events in bin
vyerr = vy**0.5 ⇐ assuming Poisson standard deviation

plt.plot([xmin, xmax],[0.,0.],c='black',lw=2)
plt.errorbar(vx, vy, vyerr, c='blue', fmt = 'o')
plt.grid()
plt.show()
```

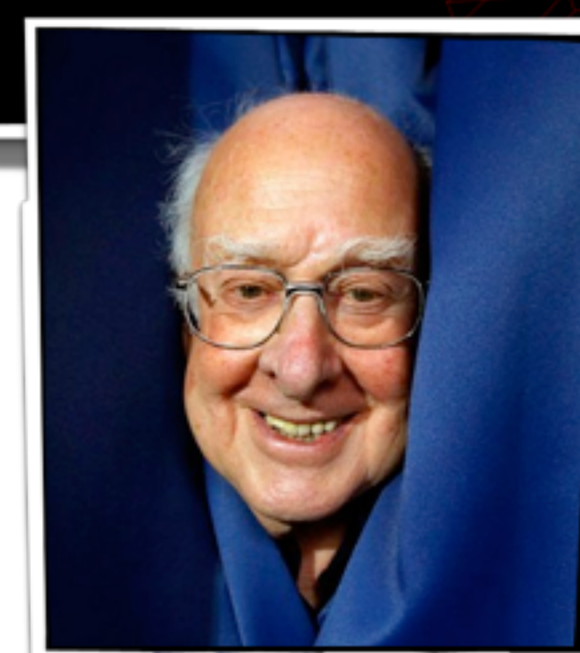
l205-example-10.py

LET'S GET SOME REAL DATA POINTS (II)

- This is the output – nothing but the (in)famous **Higgs boson**.







MODEL SETUP

- In order to perform the fit, one needs to construct a model that can describe the data. Here we simple introduce a 2nd order polynomial for the background + a Gaussian signal peak.

$$f(x) = c_0 + c_1 \cdot x + c_2 \cdot x^2$$

```python
def model(x, norm, mean, sigma, c0, c1, c2):

    xp = (x-xmin)/(xmax-xmin)
    polynomial = c0 + c1*xp + c2*xp**2

    gaussian = norm*xbinwidth/(2.*np.pi)**0.5/sigma * \
               np.exp(-0.5*((x-mean)/sigma)**2)

    return polynomial + gaussian
```

l205-example-10a.py (partial)

$$g(x) = \frac{N \cdot \Delta x}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

Δx: bin width, required for the normalization

FITTING CORE & PLOTTING

$$\chi^2 = \sum_i^N \frac{(f_i - \mu_i)^2}{\sigma_i^2}$$

Calculate χ^2 value for a given parameter set, after skipping the single zero entry bin.

```
def fcn(p):
    expt = model(vx,p[0],p[1],p[2],p[3],p[4],p[5])
    delta = (vy-expt)/vyerr
    return (delta**2).sum()

p_init = np.array([70.,125.,2.,4.,0.,0.])
r = opt.minimize(fcn,p_init)

if r.success:
    print('N(Higgs)  = %.1f events' % r.x[0])
    print('M(Higgs)  = %.1f GeV' % r.x[1])
    print('chi^2/ndf = %.2f' % (fcn(r.x)/(len(vy)-len(r.x))))
```

ndf = N(data points) −N(parameters)

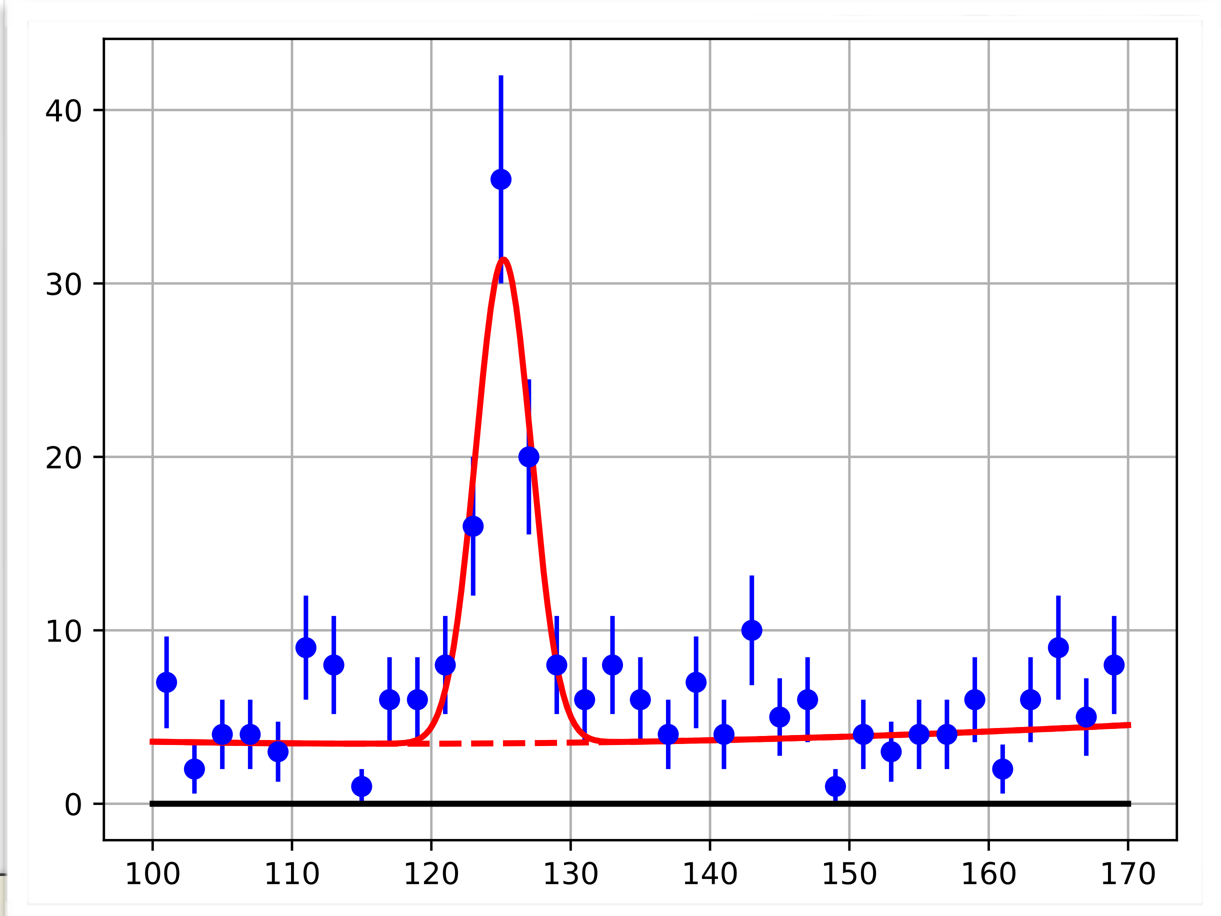
partial 110-example-10a.py

```
N(Higgs)  = 69.8 events
M(Higgs)  = 125.2 GeV
chi^2/ndf = 1.57
```

⇐ χ^2 / number of degrees of freedom ~ 1 means a good fit!

FITTING CORE & PLOTTING (II)

- Plotting – overlapping the fitting model on top of the data points.
- Generally you still have to judge/confirm the quality of fit by plotting.




```
if r.success:
    cx = np.linspace(xmin,xmax,500)
    cy = model(cx,r.x[0],r.x[1],r.x[2],r.x[3],r.x[4],r.x[5])
    cy_bkg = model(cx,0.,r.x[1],r.x[2],r.x[3],r.x[4],r.x[5])

    plt.plot(cx, cy, c='red',lw=2)
    plt.plot(cx, cy_bkg, c='red',lw=2,ls='--')
```

⇑ background curve is obtained by setting the Gaussian norm to be 0

l205-example-10a.py (partial)

ALTERNATIVE FITTING CODE

- Actually in scipy, there is a dedicated least-square fitting package, named curve_fit(). It also provides an estimation of fitting errors.

```
p_init = np.array([70.,125.,2.,4.,0.,0.])
rx,rcov = opt.curve_fit(model,vx,vy,p_init,vyerr)

if np.any(rx != p_init):
    print('N(Higgs) = %.1f +- %.1f events' % (rx[0],rcov[0,0]**0.5))
    print('M(Higgs) = %.1f +- %.1f GeV' % (rx[1],rcov[1,1]**0.5))

    cx = np.linspace(xmin,xmax,500)
    cy = model(cx,rx[0],rx[1],rx[2],rx[3],rx[4],rx[5])
    cy_bkg = model(cx,0.,rx[1],rx[2],rx[3],rx[4],rx[5])
```

⇑ No needs of calculating x^2 by ourself.

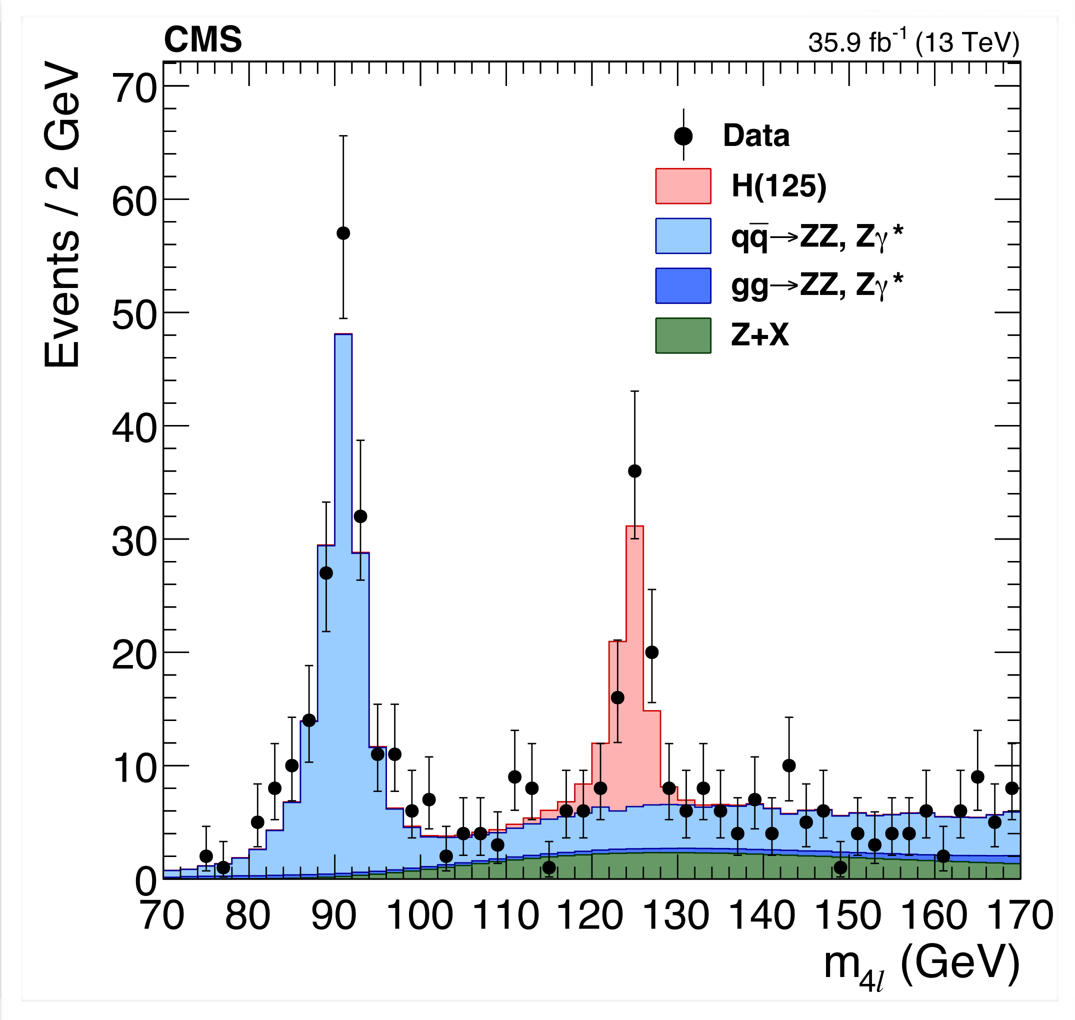
⇑ square-root of the diagonal term is the uncertainty

l205-example-10b.py (partial)

```
N(Higgs) = 18.7 +- 5.4 events
M(Higgs) = 126.3 +- 0.6 GeV
```

COMMENTS

- Surely such a simple χ^2 fit is not very professional. The real fit to the Higgs mass peak is much more difficult than just few lines.
- But this is a very good demonstration in any case!
- We will come back to this subject (statistical analysis, fitting, and modeling) again in a later lecture.



This is the real plot!

HANDS-ON SESSION

- **Practice 1:**
  Using the root function routine (Newton's method) in SciPy, implement your own **arcsine** and **arccosine** function. Please compare your own implementations and the standard routines for the following target values:

  $\sin^{-1}(0.1)$, $\sin^{-1}(0.5)$, $\sin^{-1}(0.9)$, $\sin^{-1}(1.0)$ and
  $\cos^{-1}(0.1)$, $\cos^{-1}(0.5)$, $\cos^{-1}(0.9)$, $\cos^{-1}(1.0)$

The trick: simply find the root of $\mathbf{sin(}x\mathbf{)} - \boldsymbol{R} = \mathbf{0}$ and $\mathbf{cos(}x\mathbf{)} - \boldsymbol{R} = \mathbf{0}$

HANDS-ON SESSION

- **Practice 2:**
  Produce a fit to the following data points with 2nd / 3rd / 4th / 5th order polynomial, and decide which one gives you the best quality of fit, by judging the χ^2 per number of degrees of freedom?

```
xmin, xmax, xbinwidth = 0.,1.,0.05
vx = np.linspace(0.,1.,21)
vy = np.array(
[ 0.981, 0.930, 0.900, 0.889, 0.978, 1.053, 1.000,
  0.986, 1.144, 1.188, 1.309, 1.259, 1.348, 1.435,
  1.427, 1.540, 1.426, 1.203, 0.843, 0.576, 0.060])
vyerr = np.array(
[ 0.044, 0.042, 0.037, 0.037, 0.043, 0.046, 0.038,
  0.045, 0.041, 0.041, 0.044, 0.043, 0.043, 0.041,
  0.050, 0.055, 0.052, 0.074, 0.060, 0.068, 0.082])
```

l205-practice-02.py