Lecture_02 Introduction to numerics

October 10, 2019

1 Numerical Methods

1.1 Lecture 2: Introduction to numerical methods

by: Tomasz Romaczukiewicz web: th.if.uj.edu.pl/~trom/ rm B-2-03

1.1.1 Outline

- Numerical errors

- Patriot Missile Failure (28 dead)
- Explosion of the Ariane 5
- Software errors in Aerospace (German)
- The Vancouver Stock Exchange
- Rounding error changes Parliament makeup
- The sinking of the Sleipner A offshore platform
- Tacoma bridge failure (wrong design)
- What's 77.1 x 850? Don't ask Excel 2007

Most important problems that numerical methods have to deal with are: * Accuracy what is the error of the solution? * Convergence if the method tend to the real solution * Efficiency how fast we can get the result

1.1.2 Accuracy

- Floating point numbers have finite accuracy, usually

$$x \in [x - \epsilon|x|, x + \epsilon|x|],$$

  where $\epsilon \approx 10^{-8}$ for single precision, and 10^{-16} for double precission (standard in Python).
- If not specified usually the last is the last certain digits

$$1.25 = 1.25 \pm 0.005, \quad 1.000000 = 1.000000 \pm 0.0000005,$$

  but $1 = 1 \pm 0.5$ means 50% error!
- Errors occur during calculations

$$f(a \pm \Delta a) = f(a) \pm |f'(a)|\Delta a$$

$$F(a \pm \Delta a, b \pm \Delta b) = F(a,b) \pm \left(\left| \frac{\partial F}{\partial a} \right| \Delta a + \left| \frac{\partial F}{\partial b} \right| \Delta b \right)$$

Note that euclidian norm can be used instead, which is smaller but more expensive

$$(a \pm \Delta a) + (b \pm \Delta b) = (a + b) \pm (\Delta a + \Delta b)$$

```
In [76]: import numpy as np
```

```
In [77]: c = np.float16(2.005)+np.float16(2.004)  # ~ 4 digit accuracy
         exact = 4.009
         error = abs(c-exact)
         print ("c={:.4e}, error={:.4e}, relative err={:.3f}%".format(c, error, error/c*100))
```

```
c=4.0078e+00, error=1.1875e-03, relative err=0.030%
```

$$(a \pm \Delta a) - (b \pm \Delta b) = (a - b) \pm (\Delta a + \Delta b)$$

```
In [78]: c = np.float16(2.005)-np.float16(2.004)  # ~ 4 digit accuracy
         exact = 0.001
         error = abs(c-exact)
         print ("c={:.5e}, error={:.5e}, relative err={:.3f}%".format(c, error, error/c*100))
```

```
c=1.95312e-03, error=9.53125e-04, relative err=48.800%
```

Note 49% relative error. Only the first digit is certain! Subtracting close numbers is the worst numerical operation

$$(a \pm \Delta a) - (b \pm \Delta b) = (a + b) \pm (\Delta a + \Delta b)$$

Note: Errors are added but values are subtracted. This can lead to a huge loss of accuracy (many orders of amplitude)!

$$(a \pm \Delta a) \cdot (b \pm \Delta b) = \left(a \cdot b\right) \pm \left(|a|\Delta b + \Delta a|b| + \mathcal{O}(\Delta^2)\right)$$

$$\text{Relative error:} \qquad \frac{\Delta(ab)}{|ab|} = \frac{\Delta(a/b)}{|a/b|} = \frac{\Delta a}{|a|} + \frac{\Delta b}{|b|}$$

```
In [79]: c = np.float16(2.005)*np.float16(2.004)  # ~ 4 digit accuracy
         exact = np.float64(2.005)*np.float64(2.004)
         error = abs(c-exact)
         print ("c={:.5e}, error={:.5e}, relative err={:.3f}%".format(c, error, error/c*100))
```

```
c=4.01953e+00, error=1.51125e-03, relative err=0.038%
```

Recall the example of quadratic equation

```
In [82]: a = 0.2; b=15; c=0.2
          = b**2-4*a*c
         x1 = x1_exact = (-b+np.sqrt())/(2*a)
         x2 = x2_exact = (-b-np.sqrt())/(2*a)
         print ("x1 = ", x1, "x2 =",  x2)

          = np.float16(b**2-4*a*c)
         x1 = (-b+np.sqrt())/(2*a)
         x2 = (-b-np.sqrt())/(2*a)
         print ("x1 = ", x1, "x2 =",  x2)
         print ("Relative errors: Ex_1={:.3f}%, Ex_2={:.3f}%".\
                format(100*(x1-x1_exact)/x1_exact, 100*(x2-x2_exact)/x2))
```

```
x1 =  -0.01333570454687738 x2 = -74.98666429545312
x1 =  -0.01953125 x2 = -74.98046875
Relative errors: Ex_1=46.458%, Ex_2=-0.008%
```

x_1 has a 56% error when accuracy of 10^{-4} is used wheras x_2 only 0.08%. Why? Could x_1 be calculated in a different way?

$$x_1 = \frac{\sqrt{b^2-4ac}-b}{2a} = \frac{b^2-4ac-b^2}{2a\left(\sqrt{b^2-4ac}+b\right)} = -\frac{2c}{\sqrt{b^2-4ac}+b}$$

```
In [81]:  = np.float16(b**2-4*a*c)
         x1 = -2*c/(np.sqrt() + b )
         print ("x1 = ", x1, "Exact:", x1_exact)
         print ("relative Error: {:.4f}%".format((x1-x1_exact)/x1*100))
```

```
x1 =  -0.01333680646001563 Exact: -0.01333570454687738
relative Error: 0.0083%
```

1.1.3 Miracle!

Even for Δ with lower accuracy, the final result is almost 12000 times more accurate than before! Sometimes it is enough to reformulate an equation to have better defined solution

But still:

$$\frac{1}{3}x^2 + \frac{1}{5}x + \frac{3}{100} = 0$$

```
In [68]: a, b, c = 1/3, 0.2, 0.03
         b**2-4*a*c  # test  == 0 would fail
```

```
Out[68]: 1.3877787807814457e-17
```

$$100\,x^2 + 30\,x + 9 = 0$$

```
In [71]: a, b, c = 100, 60, 9
         b**2-4*a*c  #  = 0 exactly

Out[71]: 0
```

Other examples:

$$E_0 = \frac{mc^2}{\sqrt{1-\frac{v^2}{c^2}}} - mc^2$$

$$E_1 = \frac{mv^2}{\sqrt{1-\frac{v^2}{c^2}}+1-\frac{v^2}{c^2}}$$

$$E_2 = \frac{1}{2}mv^2$$

```
In [110]: m, c, v, E = 1, 1e9, 0.5, np.empty(4) ;
          E[0] = m*c**2/np.sqrt(1-v**2/c**2)-m*c**2
          E[1] = m*v**2/(np.sqrt(1-v**2/c**2)+1-v**2/c**2)
          E[2] = m*v**2/2
          v=np.float128(v)  # check what happens when v=0.1
          E[3] = m*c**2/np.sqrt(1-v**2/c**2)-m*c**2
          print("E = ", E)

E =  [0.     0.125  0.125  0.1875]
```

1.1.4 Micro transactions

```
In [89]: T = np.float16
         a, b, c = T(0.1), T(0.2), T(0.3)
         print(1e16*(a+b-c))
         a, b, c = T(0.1), T(0.2), T(a+b)
         print(1e16*(a+b-c))

-2441406250000.0
0.0
```

1.1.5 Summation problem I

```
In [73]: """Standard summation"""
         data = np.array([1000]+1000*[0.1])
         def naive_summation(data):
             s = 0
             for d in data:
                 s += d
             return s
         print(naive_summation(data))
```

```
1099.9999999999363
```

```
In [74]: data2=np.copy(data) # data2 = data is just an alias to data (reference not a real cop
         data2.sort()
         print(naive_summation(data2))  # better  but not accurate
         print(np.sum(data))            # standard numpy summation
```

```
1099.9999999999986
1100.0000000000002
```

```
In [55]: """Kahan Summation"""
         def kahn_summation(data):
             s, c = 0, 0
             for d in data:
                 y = d-c
                 t = s+y
                 c = (t-s)-y
                 s = t
             return s
         print(kahn_summation(data))
```

```
1100.0
```

Condition number Question: how much a numerical procedure $f(x)$ multiples an input error Δx Absolute condition number: how much an absolute error can be increased

$$\lim_{\varepsilon \to 0} \sup_{\|\Delta\| x \leq \varepsilon} \frac{\|\Delta f\|}{\|\Delta x\|}$$

Relative condition number:

$$\text{cond}(f) = \lim_{\varepsilon \to 0} \sup_{\|\Delta\| x \leq \varepsilon} \frac{\|\Delta f(x)\| / \|f(x)\|}{\|\Delta x\| / \|x\|}$$

Condition number is a property of a given problem not the method.
Example from previous lecture

$$I_n = \frac{1}{e} \int_0^1 e^x x^n \, dx$$

generates the following conditions (decreasing but positive sequence)

$$0 < I_n < I_{n-1}$$

and

$$I_0 = 1 - \frac{1}{e} \qquad I_n = 1 - n I_{n-1}$$

Suppose that we calculate I_0 with ΔI_0 error

$$I_n + \Delta I_n = 1 - nI_{n-1} - n\,\Delta I_{n-1} \quad \Rightarrow \quad \Delta I_n = -n\,\Delta I_{n-1}$$

So the error of I_n is

$$\Delta I_n = (-n)(-(n-1))(-(n-2))\ldots(-2)(-1)\,\Delta I_0 = \Delta I_n = (-1)^n n!\,\Delta I_0$$

So the condition number for calculation of I_n is $n!$. Even if $\Delta I_0 = 10^{-16}$ we have $\Delta I_{20} \approx 240$ with cond $> 2.4 \cdot 10^{18}$ since $I_n < I_0$

```
In [287]: import math
          print("{:e}".format(math.factorial(20)))
```

```
2.432902e+18
```

1.2 Root finding

```
In [289]: def F(x):
              return x**4-2

          import scipy.optimize as sc
          x = sc.fsolve(F, 1)
          print(x[0], 2**0.25)
```

```
1.189207115002721 1.189207115002721
```

Python (or more precisely scipy) provides a powerful solver It is enough to provide a function definition and a starting point But

1.3 ## Root finding

Suppose that we have a continous function $f : \mathbb{R} \ni x \to \mathbb{R}$

Task: find a solution $f(x) = 0$. Two basic strategies: - bracketing methods $x \in [a, b]$ - more reliable (stable) - require sign change $f(a)f(b) < 0$ (not for even roots)
- iterations - faster convergence - usually require differentiability - can be unstable (cycles, unstable fix points etc.)

- bracketing methods
  - bisection
  - regula falsi
- iterations
  - Newton method (Warden and Master of Royal Mint)
  - secan method
  - Hayley method
- combined forces

– Brent's method (bisection + secant method, inverse quadratic interpolation)

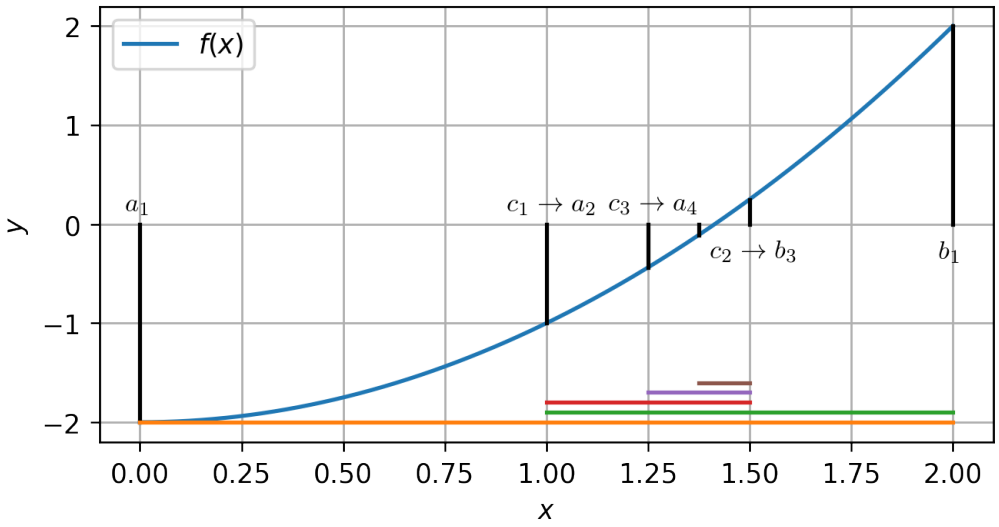
Bisection method - check if $f(a)f(b) < 0$ (zero is somewhere in between a and b): - $c = \frac{1}{2}(a+b)$ - choose a segment where $f(x)$ changes a sign $[a,c]$ or $[c,b]$ by checking the sign - if $f(a)f(c) < 0$ do the bisection for $[a,c]$ by changing $b \leftarrow c$, $f(b) \leftarrow f(c)$ - if $f(b)f(c) < 0$ do the bisection for $[a,c]$ by changing $a \leftarrow c$, $f(a) \leftarrow f(c)$

Note: count f only once for each point

```
In [321]: def f(x): return x**2-2
```

```
          import matplotlib.pyplot as plt  # ploting library
          def make_bisection_plot():
              x = np.linspace(0, 2, 1000)
              y = f(x)
              plt.figure(figsize=(6,3), dpi=150)
              plt.plot(x, y)
              plt.plot([0, 0], [0, f(0)], c='black')
              plt.plot([2, 2], [0, f(2)], c='black')
              plt.plot([1, 1], [0, f(1)], c='black')
              plt.plot([1.5, 1.5], [0, f(1.5)], c='black')
              plt.plot([1.25, 1.25], [0, f(1.25)], c='black')
              plt.plot([1.375, 1.375], [0, f(1.375)], c='black')
              plt.plot([0, 2], [-2, -2])
              plt.plot([1, 2], [-1.9, -1.9])
              plt.plot([1, 1.5], [-1.8, -1.8])
              plt.plot([1.25, 1.5], [-1.7, -1.7])
              plt.plot([1.375, 1.5], [-1.6, -1.6])
              plt.text(0, 0.1, "$a_1$", verticalalignment='bottom', horizontalalignment='cente
              plt.text(2, -0.1, "$b_1$", verticalalignment='top', horizontalalignment='center'
              plt.text(1, 0.1, '$c_1\\to a_2$', verticalalignment='bottom', horizontalalignmen
              plt.text(1.5, -0.1, '$c_2\\to b_3$', verticalalignment='top', horizontalalignmen
              plt.text(1.25, 0.1, '$c_3\\to a_4$', verticalalignment='bottom', horizontalalign
              plt.legend(['$f(x)$'])
              plt.xlabel('$x$')
              plt.ylabel("$y$")
              plt.grid(True)
              plt.show()
```

```
In [322]: make_plot()
```

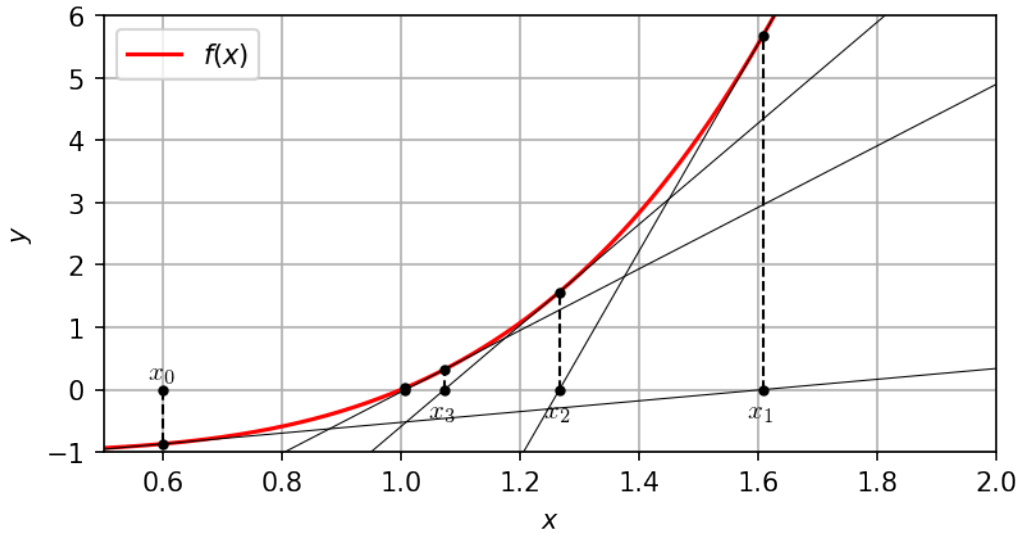


In each step we reduce twice the legth of the interval in which the root resides $\epsilon_n = \frac{1}{2}\epsilon_{n-1} = 2^{-n}(b-a)$ Example of a linear convergence $\epsilon_n \sim \epsilon^1{}_{n-1}$ Why bisection and not decasection? - 10 iterations of bisection (11 execution of f) gives accuracy $\frac{b-a}{1024}$ - 1 decasection (11 execution of f) gives accuracy $\frac{b-a}{10}$

```
In [323]: def f(x): return x**4-1

          def df(x): return 4*x**3;
          def tangent(x, x0, y0, dy):
              return y0+dy*(x-x0)

          def make_Newton_plot(x0, f, df, left=0.5, right=2, bottom=-1, top=6):
              x = np.linspace(left, right, 1000)
              y = f(x)
              plt.figure(figsize=(6,3), dpi=150)
              plt.ylim(bottom, top)
              plt.xlim(left, right)
              plt.plot(x, y, c='r')

              N = 5
              X = np.empty(N)
              X[0] = x0
              for n in range(N-1):
                  xn = X[n]
                  yn = f(xn)
                  dy = df(xn)
                  X[n+1] = xn - yn/dy
                  plt.plot([xn, xn], [0, yn], c='black', lw=1, linestyle='dashed')
                  plt.plot([xn, xn], [0, yn], 'o', c='black', ms=3)
                  plt.plot(x, yn + dy*(x-xn), c='black', lw = 0.5)
```

```
            if yn<0:
                plt.text(xn, 0.1, "$x_{:d}$".format(n), \
                         verticalalignment='bottom', \
                         horizontalalignment='center', usetex=True)
            else:
                plt.text(xn, -0.1, "$x_{:d}$".format(n), \
                         verticalalignment='top', \
                         horizontalalignment='center', usetex=True)
        xn = X[N-1]
        yn = f(xn)
        plt.plot([xn, xn], [0, yn], 'o', c='black', ms=3)
        print(X)

        plt.legend(['$f(x)$'])
        plt.xlabel('$x$')
        plt.ylabel("$y$")
        plt.grid(True)
        plt.show()

    make_Newton_plot(0.6, f, df)
```

```
[0.6        1.60740741 1.26575079 1.07259388 1.0070429 ]
```



1.3.1 Newton method

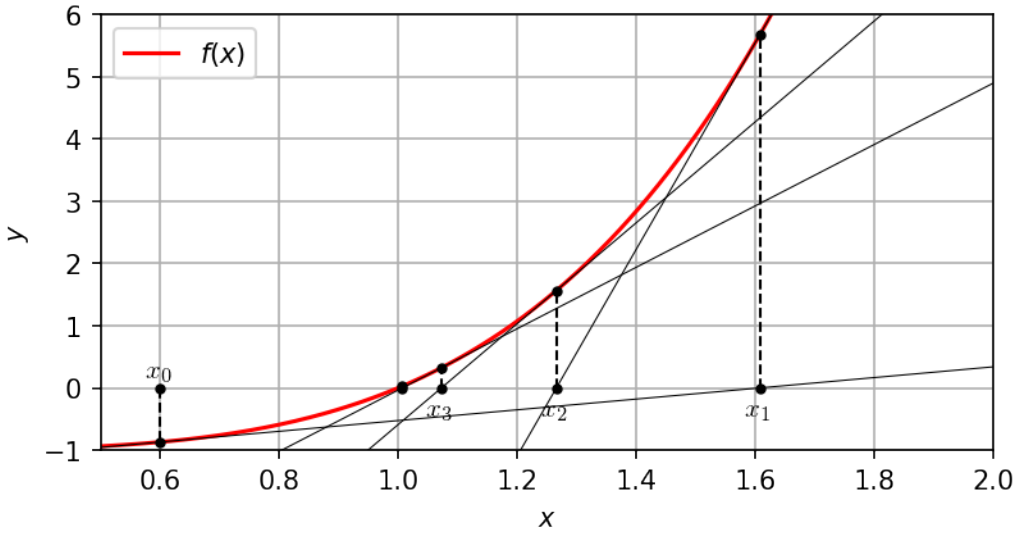
- We choose a starting point x_0
- For $n > 0$ we construct a tangent line at $(x_n, f(x))$

$$y = f'(x_n)(x - x_n) + f(x_n)$$

- and find its root

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

```
In [324]: make_Newton_plot(0.6, f, df)

[0.6        1.60740741 1.26575079 1.07259388 1.0070429 ]
```



Newton method can converge very fast $x_n = x_* + \epsilon_n : f(x_*) = 0$:

$$(x_{n+1} - x_n) f'(x_n) = -f(x_n)$$

Expanding in Taylor series up to $\mathcal{O}(\epsilon^2)$:

$$(\epsilon_{n+1} - \epsilon_n)(f'(x_*) + \epsilon_n f''(x_*)) = -f(x_*) - \epsilon_n f'(x_*) - \frac{1}{2}\epsilon_n^2 f''(x_*)$$

The convergence can be quadratic

$$\epsilon_{n+1} = \frac{f''(x_*)}{2f'(x_*)}\epsilon_n^2$$

provided that $f'(x_*) \neq 0$.

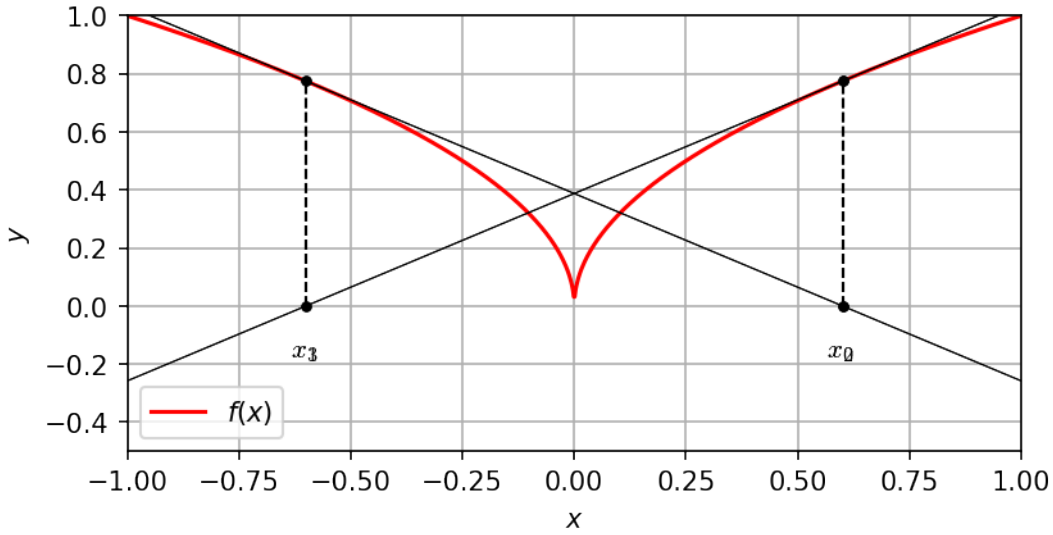
What happens when $f(x_*) = 0$: exapmle: $f(x) = |x|^k$.

$$x_{n+1} = x_n - \frac{1}{k}x_n = \left(1 - \frac{1}{k}\right) x_n = q\, x_n$$

This is a geometric series which converges linearly $|q| < 1$ or can be even divergent $|q| > 1$.
Interesting case $q = -1$ for $k = 1/2$

$$x_{n+1} = -x_n \qquad \text{two-cycle}$$

```
In [325]: make_Newton_plot(0.6, lambda x: np.sqrt(np.abs(x)), \
                           lambda x: 0.5*np.sign(x)/np.sqrt(np.abs(x)), \
                           left=-1, right=1, bottom =-0.5, top=1)

[ 0.6 -0.6  0.6 -0.6  0.6]
```



Newton method: + fast (quadratic convergence for single root, - linear convergence for highert multiplicity), - can be divergent when $|f'(x_n)| \ll 1$, - problem for $f'(x_n) = 0$, - can admit cycles - add a damping term $x_{n+1} = x_n - \alpha f(x_n)/f'(x_n)$ with $|\alpha| < 1$ - can be applied for complex numbers (beware of a Newton fractal)
- easily generalizable to higher dimensions

Similar methods: - secant method: secant line through last two points instead of a tangent line at a single point

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

- slower convergence but still superlinear $\epsilon_{n+1} \approx \epsilon_n^{1.8}$ - does not require derivatives - other problems similar to Newton method - regula falsi similar to secant but from the three points we choose a segment where the function changes sign (just like bisection) - bracketing method but usually faster than bisection - sometimes it can get stuck

- Hayley's method

  $$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2[f'(x_n)]^2 - f(x_n)f''(x_n)}$$

  is a Newton's method for

  $$g(x) = \frac{f(x)}{\sqrt{|f'(x)|}}.$$

  – faster (usually cubic) convergence

Note: both Newton and Hayley developed their methods for celestial mechanical problems to solve Kepler's problem (finding E)

$$M = E - e \sin E$$

where M - mean anomaly, E - eccentric anomaly, and e - eccentricity.

Other high order methods? Take parabola from three points $(x_n, y_n), (x_{n-1}, y_{n-1}), (x_{n-2}, y_{n-2})$ instead of a straight line?

$$Q(x) = \frac{(x - x_{n-1})(x - x_{n-2})y_n}{(x_n - x_{n-1})(x_n - x_{n-2})} + \frac{(x - x_n)(x - x_{n-2})y_{n-1}}{(x_{n-1} - x_n)(x_{n-1} - x_{n-2})} + \frac{(x - x_n)(x - x_{n-1})y_{n-2}}{(x_{n-2} - x_n)(x_{n-2} - x_{n-1})}$$

Yes, but two solutions for $Q(x) = 0$ (or one or none or infinitely many) Muller's method - some more stable algorithm

Better: inverse interpolation

$$P(y) = \frac{(y - y_{n-1})(y - y_{n-2})x_n}{(y_n - y_{n-1})(y_n - y_{n-2})} + \frac{(y - y_n)(y - y_{n-2})x_{n-1}}{(y_{n-1} - y_n)(y_{n-1} - y_{n-2})} + \frac{(y - x_n)(y - x_{n-1})x_{n-2}}{(y_{n-2} - y_n)(y_{n-2} - y_{n-1})}$$

$P(y)$ interpolates the inverse of $y = f(x)$ i.e. $x = f^{-1}(y) \approx P(y)$ so $x_{n+1} = P(y = 0)$ gives another approximation to zero.

- Unique iteration,
- three pointsmust form a monotonic series
- function f also must be monotonic in a given segment
- straightforward generalization for higher order methods.

Polynomial equations $Q(x) = 0$ can make use of some better techniques: - use deflation if $Q(x_*) = 0$ we can devive $\tilde{Q}(x) = \frac{Q(x)}{x - x_*}$ and solve easier problem - important: solution $\tilde{Q}(x) = 0$ is just an approximation and must be used as a starting point for solving full problem $Q(x) = 0$ due to numerical errors generated by divisions. - Newton's method but starting from real x_0 only real x_n can be found (if Q has real coefficients) - Better: Laguerre's method capable of finding complex solutions

1.3.2 Why use or not to use the standard scipy solver?

- For a single root it is fine and perhaps even quite optimal
- Sometimes it can be overcostly
  - we have to solve iteratively many equations implicit schemes for solving differential equations
  - general solvers can work too slowly for certain functions, sometimes we can choose better solver

1.4 Tasks:

1. Implement Newton's and bisection method (best make it library-style)
2. Use the scipy solver to solve and your own methods to solve the following equations
   - $f(x) = x - \cos(x) = 0$