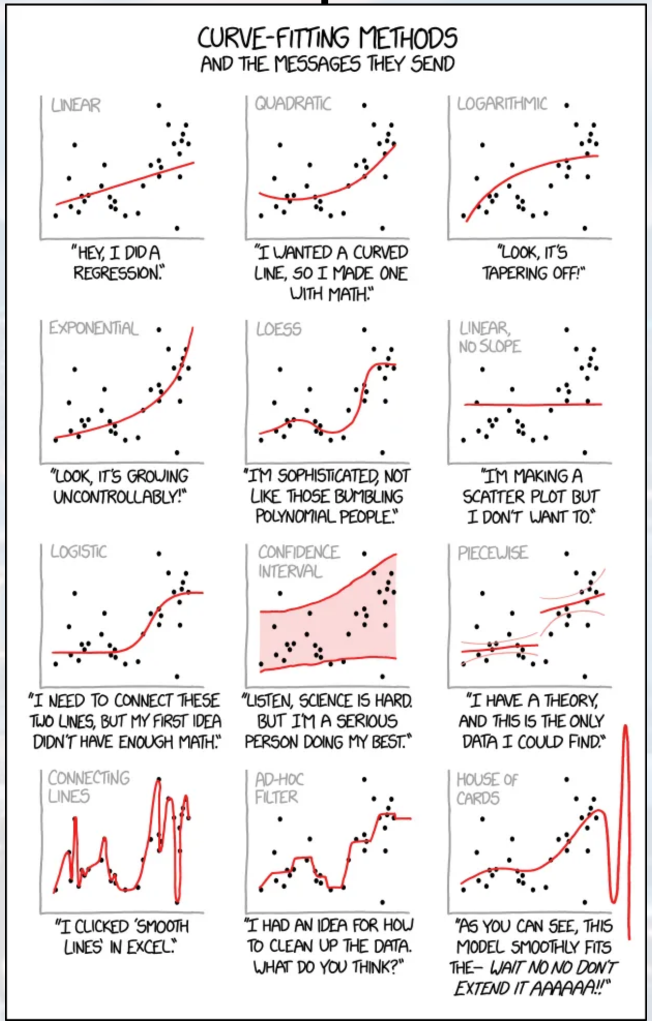
M. Hohle:

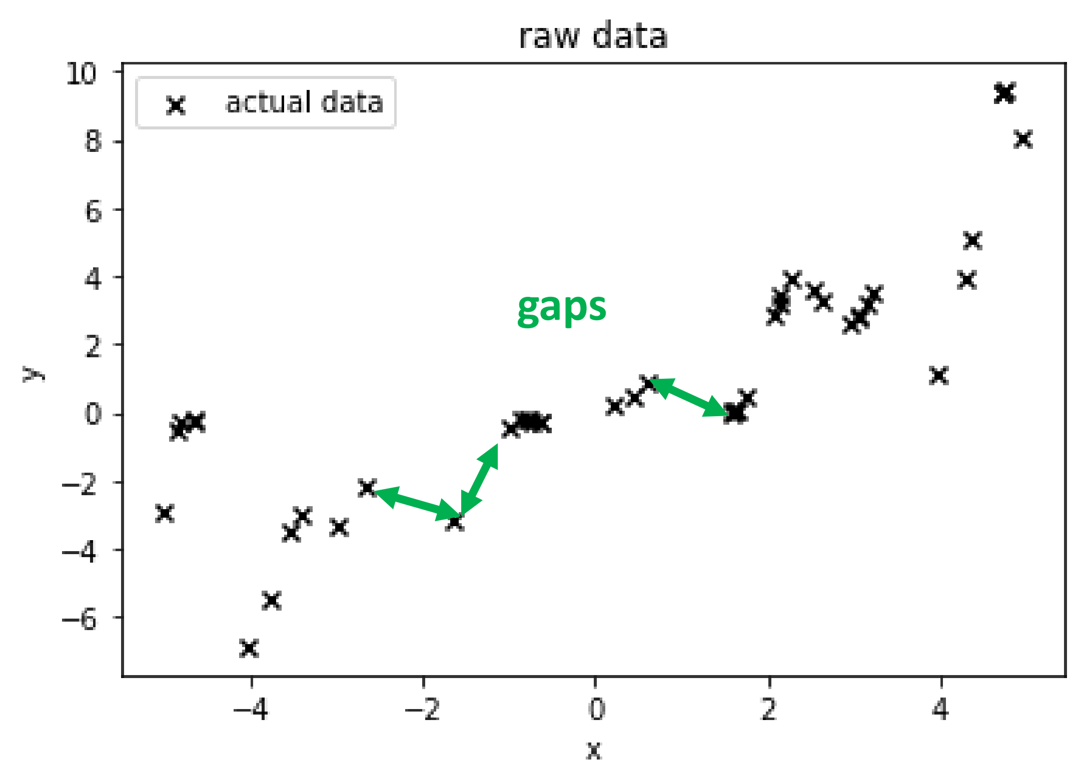
Physics 77: Introduction to Computational Techniques in Physics



syllabus:

- Introduction to Unix & Python	(week 1 - 2)
- Functions, Loops, Lists and Arrays	(week 3 - 4)
- Visualization	(week 5)
- Parsing, Data Processing and File I/O	(week 6)
- Statistics and Probability, Interpreting Measurements	(week 7 - 8)
- Random Numbers, Simulation	(week 9)
- Numerical Integration and Differentiation	(week 10)
- Root Finding, Interpolation	**(week 11)**
- Systems of Linear Equations	(week 12)
- Ordinary Differential Equations	(week 13)
- Fourier Transformation and Signal Processing	(week 14)
- Capstone Project Presentations	(week 15)

the problem:



How to interpolate?

- polynomials (1st order = linear)
- piecewise polynomials
- trigonometric functions
- exponential functions
- rational functions

called **“basis functions”**

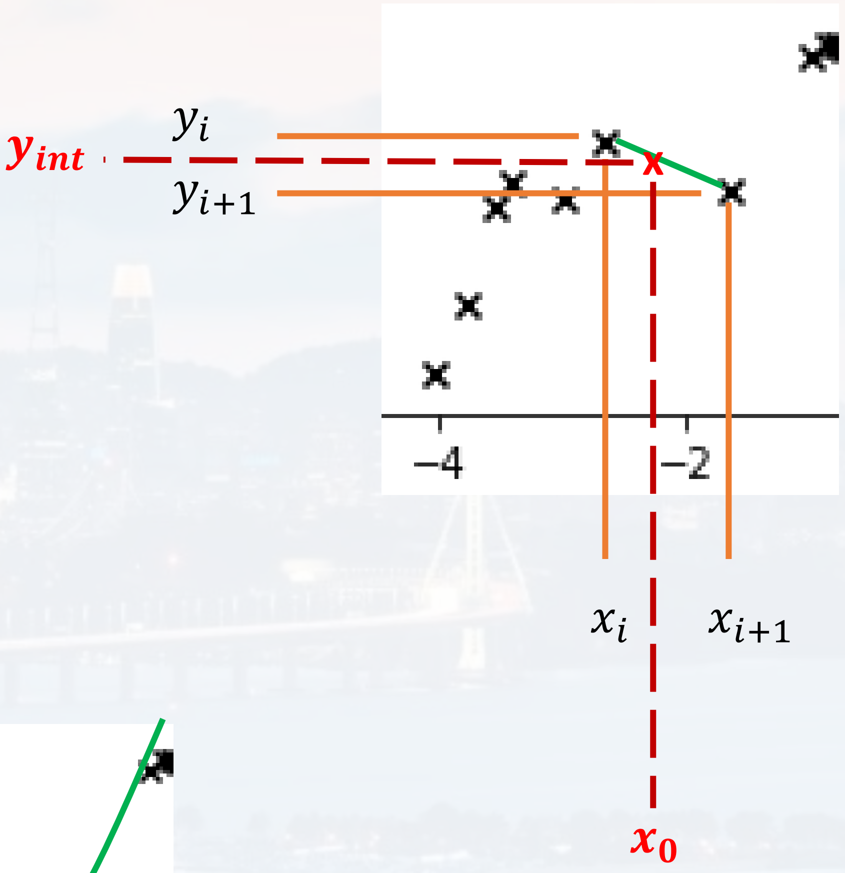
note: interpolation is not fitting!

the problem:

linear interpolation

$$y_{int} = y_i + m\,(x_0 - x_i)$$

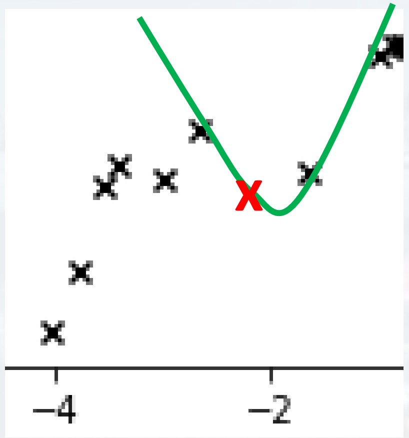
$$y_{int} = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}\,(x_0 - x_i)$$



quadratic interpolation

$$y_{int} = y_i + m\,(x_0 - x_i) + a\,(x_0 - x_i)^2$$

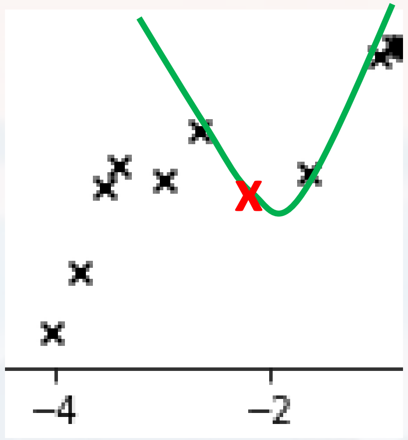
this time we need **one more** reference point for calculating ***a***



the problem:

quadratic interpolation

$$y_{int} = y_i + m\,(x_0 - x_i) + a\,(x_0 - x_i)^2$$



this time we need **one more** reference point for calculating ***a***

all three reference points need to fit the same parabola

$$y_i = c + m x_i + a\,{x_i}^2$$

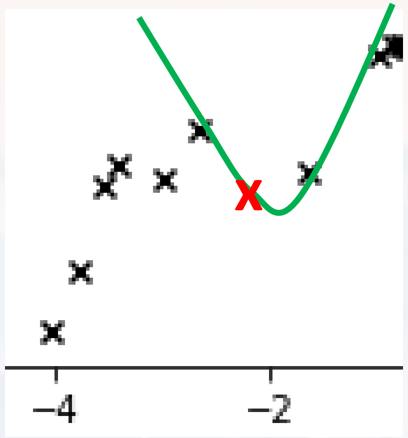
$$y_{i+1} = c + m x_{i+1} + a\,{x_{i+1}}^2$$

$$y_{i+2} = c + m x_{i+2} + a\,{x_{i+2}}^2$$

solving for c, m and a

the problem:

linear interpolation: $$y_{int} = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x_0 - x_i)$$



quadratic interpolation:

$$y_i = c + m x_i + a\, x_i^2$$
$$y_{i+1} = c + m x_{i+1} + a\, x_{i+1}^2$$
$$y_{i+2} = c + m x_{i+2} + a\, x_{i+2}^2$$

$\Rightarrow$ **solving for c, m and a**

Maybe there is a closed (= general) solution/method? → **Lagrange Polynomials**

$$y_i = y_{int} + y'_{int}(x_i - x_0) + \mathcal{O}(\Delta x^2)$$ Taylor expansion
$$y_{i+1} = y_{int} + y'_{int}(x_{i+1} - x_0) + \mathcal{O}(\Delta x^2)$$

$$y_{int} = \frac{y_i(x_{i+1} - x_0)}{(x_{i+1} - x_i)} - \frac{y_{i+1}(x_i - x_0)}{(x_{i+1} - x_i)} = \boxed{y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x_0 - x_i)}$$

Maybe there is a closed (= general) solution/method? → **Lagrange Polynomials**

$$y_i = y_{int} + y'_{int}(x_i - x_0) + \mathcal{O}(\Delta x^2)$$
$$y_{i+1} = y_{int} + y'_{int}(x_{i+1} - x_0) + \mathcal{O}(\Delta x^2)$$

Taylor expansion

$$y_{int} = \frac{y_i(x_{i+1} - x_0)}{(x_{i+1} - x_i)} - \frac{y_{i+1}(x_i - x_0)}{(x_{i+1} - x_i)} = \boxed{y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x_0 - x_i)}$$

$$y_i = y_{int} + y'_{int}(x_i - x_0) + y''_{int}(x_i - x_0)(x_i - x_0)/2 + \mathcal{O}(\Delta x^3)$$
$$y_{i+1} = y_{int} + y'_{int}(x_{i+1} - x_0) + y''_{int}(x_{i+1} - x_0)(x_{i+1} - x_0)/2 + \mathcal{O}(\Delta x^3)$$
$$y_{i+2} = y_{int} + y'_{int}(x_{i+2} - x_0) + y''_{int}(x_{i+2} - x_0)(x_{i+2} - x_0)/2 + \mathcal{O}(\Delta x^3)$$

Taylor expansion

$$y_{int} = \frac{(x_0 - x_{i+1})(x_0 - x_{i+2})}{(x_i - x_{i+1})(x_i - x_{i+2})}y_i + \frac{(x_0 - x_i)(x_0 - x_{i+2})}{(x_{i+1} - x_i)(x_{i+1} - x_{i+2})}y_{i+1} + \frac{(x_0 - x_i)(x_0 - x_{i+1})}{(x_{i+2} - x_i)(x_{i+2} - x_{i+1})}y_{i+2}$$

Maybe there is a closed (= general) solution/method? → **Lagrange Polynomials**

$$y_{int} = \frac{(x_0 - x_{i+1})(x_0 - x_{i+2})}{(x_i - x_{i+1})(x_i - x_{i+2})} y_i + \frac{(x_0 - x_i)(x_0 - x_{i+2})}{(x_{i+1} - x_i)(x_{i+1} - x_{i+2})} y_{i+1} + \frac{(x_0 - x_i)(x_0 - x_{i+1})}{(x_{i+2} - x_i)(x_{i+2} - x_{i+1})} y_{i+2}$$

<u>for any polynomial of n-th order:</u>

$$y_{int} = \frac{(x_0 - x_{i+1})(x_0 - x_{i+2}) \ldots (x_0 - x_{i+n})}{(x_i - x_{i+1})(x_i - x_{i+2}) \ldots (x_i - x_{i+n})} y_i + \frac{(x_0 - x_i)(x_0 - x_{i+2}) \ldots (x_0 - x_{i+n})}{(x_{i+1} - x_i)(x_{i+1} - x_{i+2}) \ldots (x_{i+1} - x_{i+n})} y_{i+1} +$$

$$\ldots + \frac{(x_0 - x_i)(x_0 - x_{i+2}) \ldots (x_0 - x_{i+n-1})}{(x_{i+n} - x_i)(n - x_{i+2}) \ldots (x_{i+1} - x_{i+n-1})} y_{i+n}$$

$$\boxed{y_{int} = L(x_0) = \sum_{j=0}^{n} y_j \prod_{\substack{0 \le m < n \\ m \neq j}} \frac{x_0 - x_m}{x_j - x_m}}$$

Lagrange Polynomials

$$y_{int} = L(x_0) = \sum_{j=0}^{n} y_j \prod_{\substack{0 \le m < n \\ m \ne j}} \frac{x_0 - x_m}{x_j - x_m}$$

Lagrange Polynomials

- computation is simple
- but not efficient for large n
- → only considering data points close to x_0
- → reduces approximation accuracy

Newton's Interpolating Polynomials

fitting n+1 data points to nth order polynomial

→ fitting the n+1 coefficients:

$$a_0 = y_i$$

$$a_1 = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

$$a_2 = \frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}$$

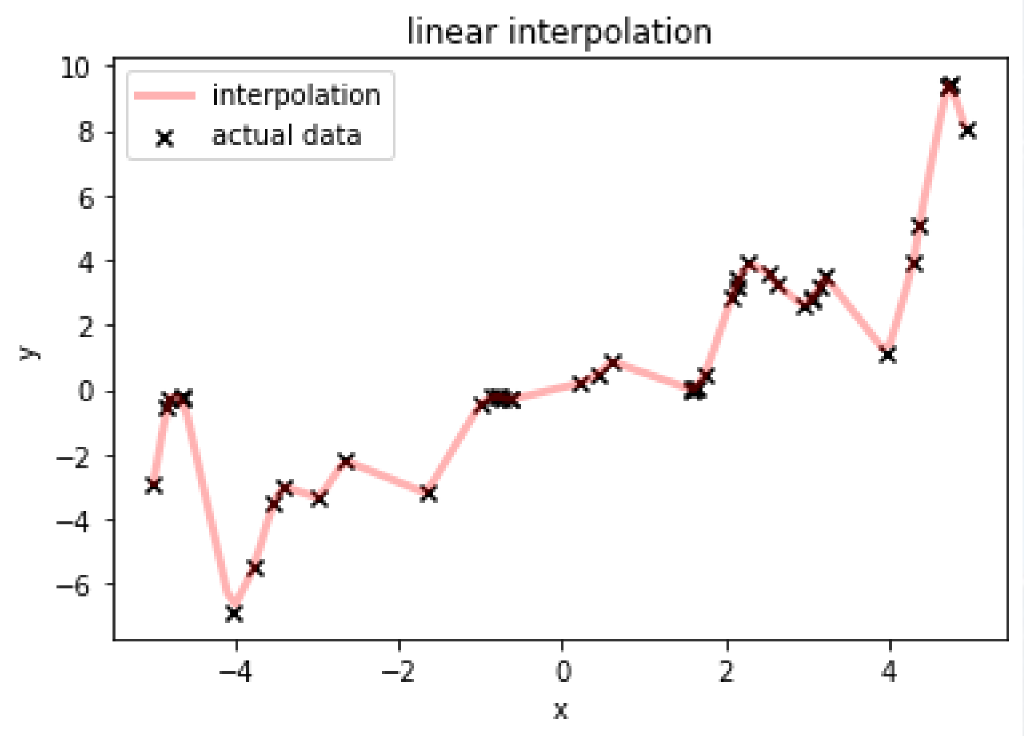
…and so on

check out InterpolateExamples.py

```
from scipy import interpolate

I    = interpolate.interp1d(x, y)

xint = np.arange(left, right, 0.1)
yint = I(xint)

plt.plot(xint, yint, c = 'r', linewidth = 3, alpha = 0.3,\
                                            label = 'interpolation')
plt.scatter(x, y, marker = 'x', c = 'k', label = 'actual data')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('linear interpolation')
plt.show()
```

check out InterpolateExamples.py

```
from scipy import interpolate

I    = interpolate.interp1d(x, y)

xint = np.arange(left, right, 0.1)
yint = I(xint)

plt.plot(xint, yint, c = 'r', linewidth =

plt.scatter(x, y, marker = 'x', c = 'k',
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('linear interpolation')
plt.show()
```



check out `InterpolateExamples.py`

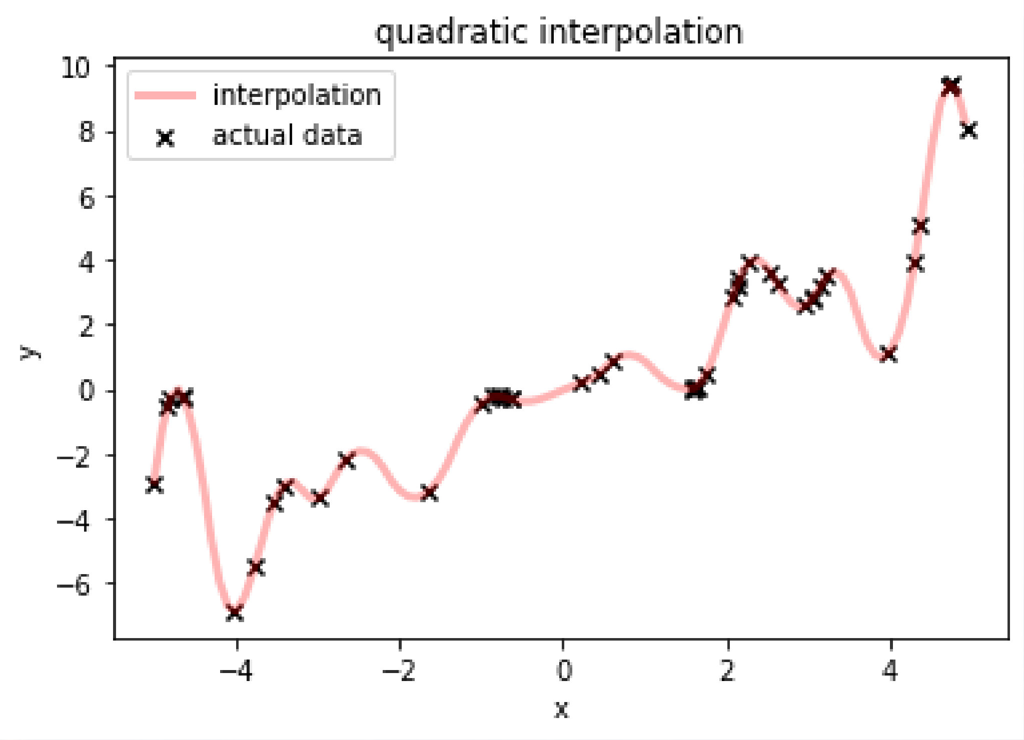
```
from scipy import interpolate

I    = interpolate.interp1d(x, y, kind = 2)

xint = np.arange(left, right, 0.1)
yint = I(xint)

plt.plot(xint, yint, c = 'r', linewidth = 3, alpha = 0.3,\
                                              label = 'interpolation')
plt.scatter(x, y, marker = 'x', c = 'k', label = 'actual data')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('linear interpolation')
plt.show()
```

quadratic interpolation

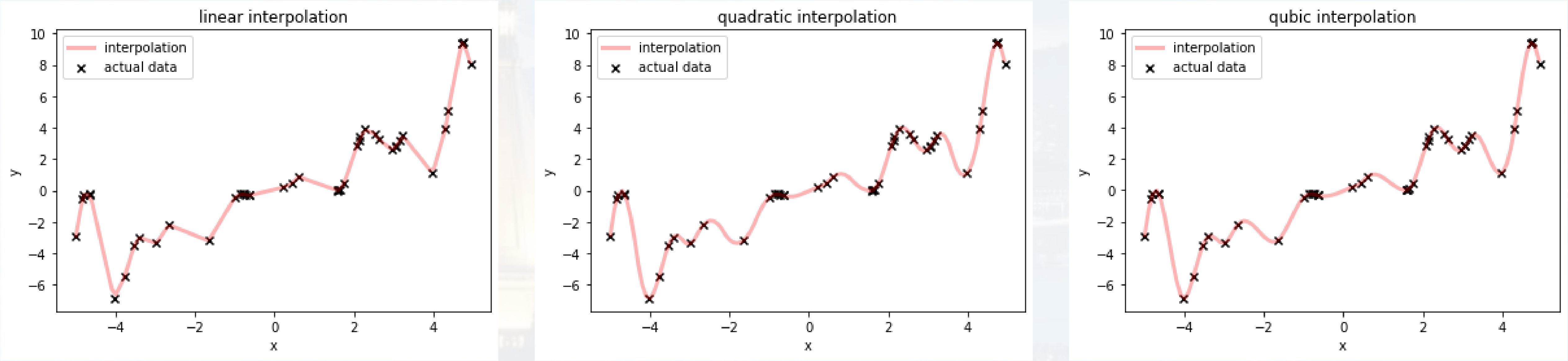
check out InterpolateExamples.py

```
from scipy import interpolate

I    = interpolate.interp1d(x, y, kind = 2)

xint = np.arange(left, right, 0.1)
yint = I(xint)

plt.plot(xint, yint, c = 'r', linewidth =

plt.scatter(x, y, marker = 'x', c = 'k',
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title('linear interpolation')
plt.show()
```

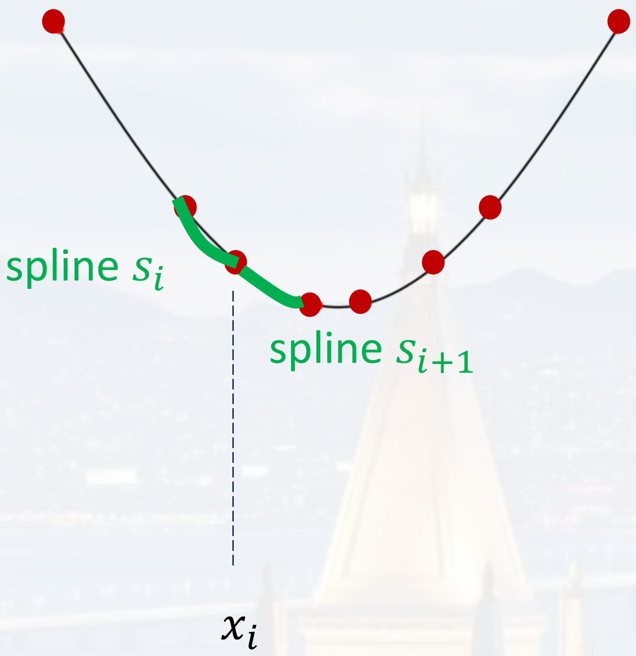
quadratic interpolation



Interpolation
Smoothing
Root Finding



spline interpolation



A shape (**piecewise** polynomials, usually cubic) that minimizes the curvature ***κ*** under the constraint of passing through all reference points

$$\kappa = \frac{\frac{d^2y}{dx^2}}{\left(1 + \left[\frac{dy}{dx}\right]^2\right)^{3/2}}$$

$$s_i(x_i) = s_{i+1}(x_i) = y_i$$

$$s_i'(x_i) = s_{i+1}'(x_i)$$

$$s_i''(x_i) = s_{i+1}''(x_i)$$

spline interpolation

A shape (**piecewise** polynomials, usually cubic) that minimizes the curvature ***κ*** under the constraint of passing through all reference points

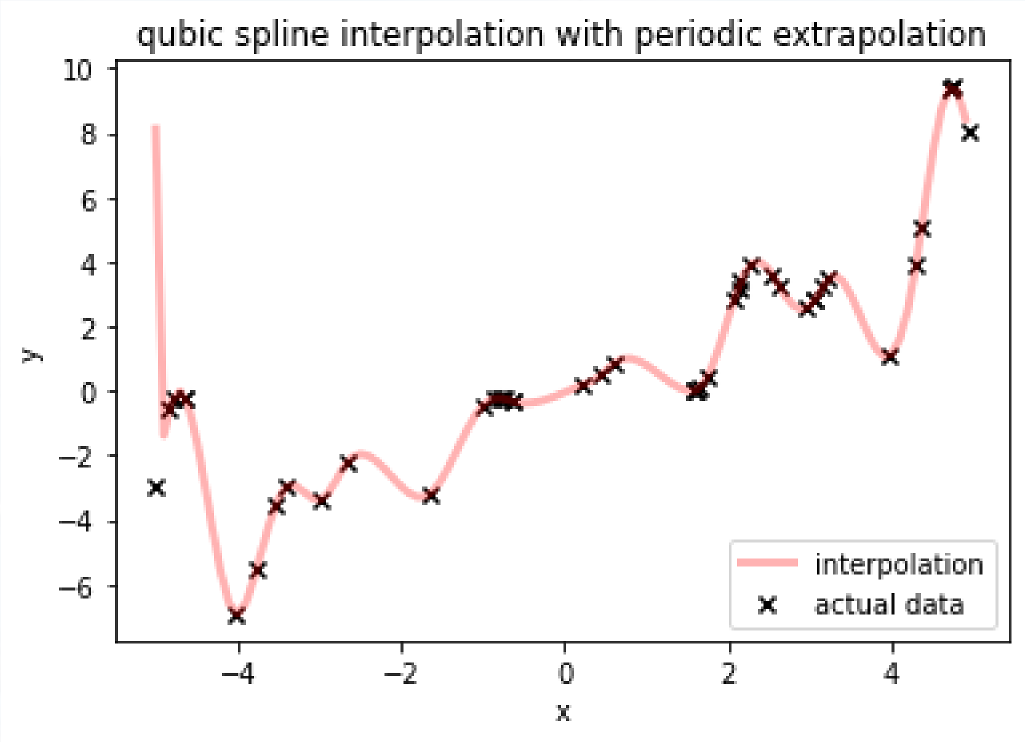
$$\kappa = \frac{\frac{d^2y}{dx^2}}{\left(1 + \left[\frac{dy}{dx}\right]^2\right)^{3/2}}$$

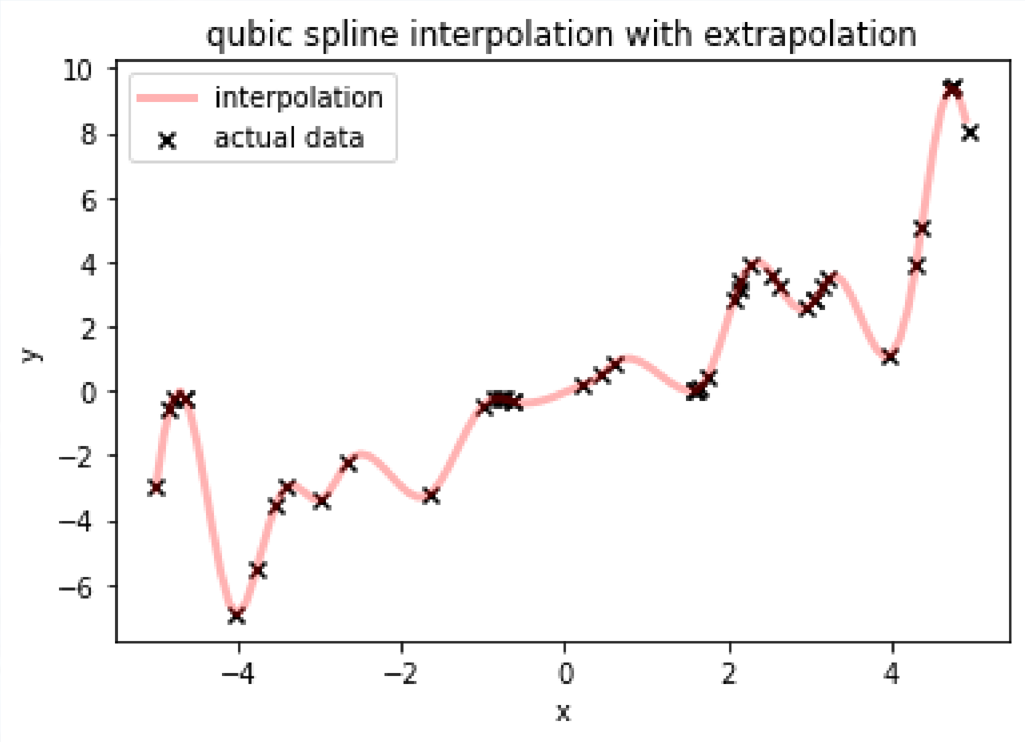
x needs to be sorted in ascending order
* stands for unpacking zipped objects

```
sorted_pairs       = sorted(zip(x, y))
x_sorted, y_sorted = zip(*sorted_pairs)
```

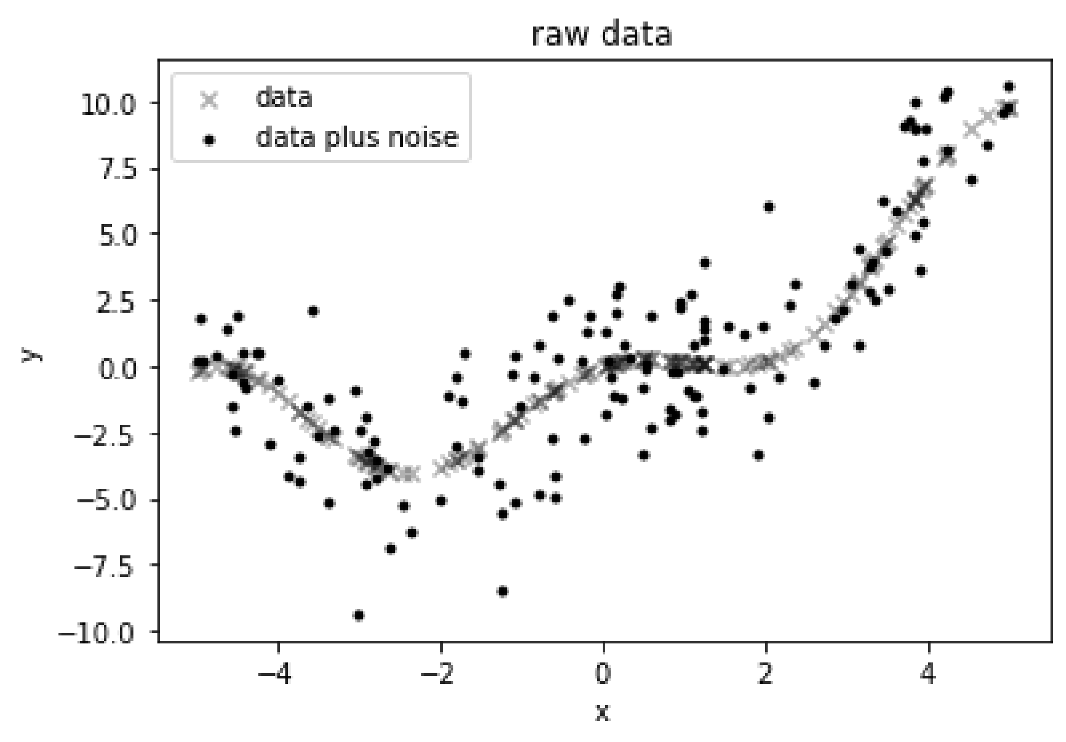
```
I = interpolate.CubicSpline(x_sorted, y_sorted,\
                              extrapolate = 'periodic')
```

Interpolation
Smoothing
Root Finding





the problem:

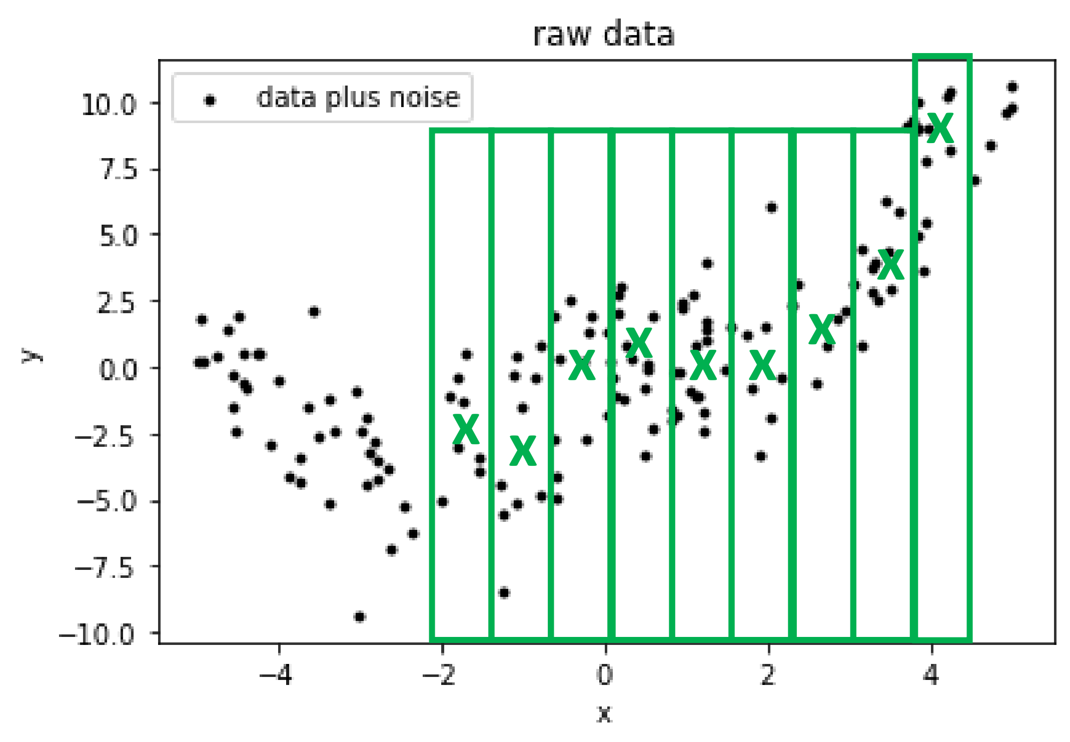


when interpolating

→ you don't want to interpolate noise

many noise filter are **low pass** filter smoothing filter:

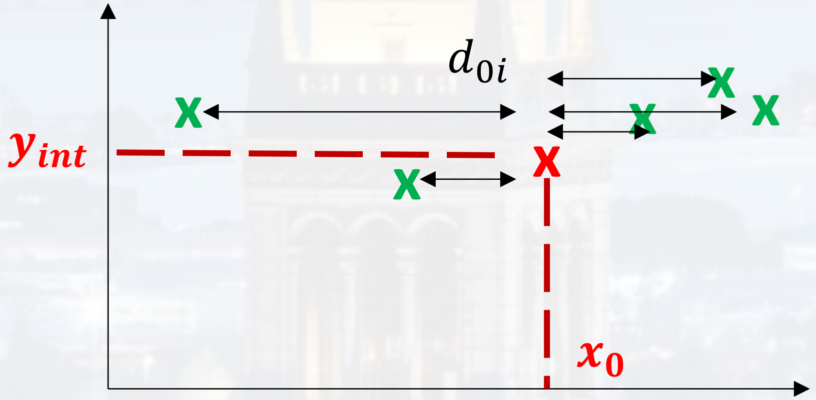
Algorithm	Overview and uses	Pros	Cons
Additive smoothing	used to smooth categorical data.		
Butterworth filter	Slower roll-off than a Chebyshev Type I/Type II filter or an elliptic filter	• More linear phase response in the passband than Chebyshev Type I/ Type II and elliptic filters can achieve. • Designed to have a frequency response as flat as possible in the passband.	• requires a higher order to implement a particular stopband specification
Chebyshev filter	Has a steeper roll-off and more passband ripple (type I) or stopband ripple (type II) than Butterworth filters.	• Minimizes the error between the idealized and the actual filter characteristic over the range of the filter	• Contains ripples in the passband.
Digital filter	Used on a sampled, discrete-time signal to reduce or enhance certain aspects of that signal		
Elliptic filter			
Exponential smoothing	• Used to reduce irregularities (random fluctuations) in time series data, thus providing a clearer view of the true underlying behaviour of the series. • Also, provides an effective means of predicting future values of the time series (forecasting).[3]		



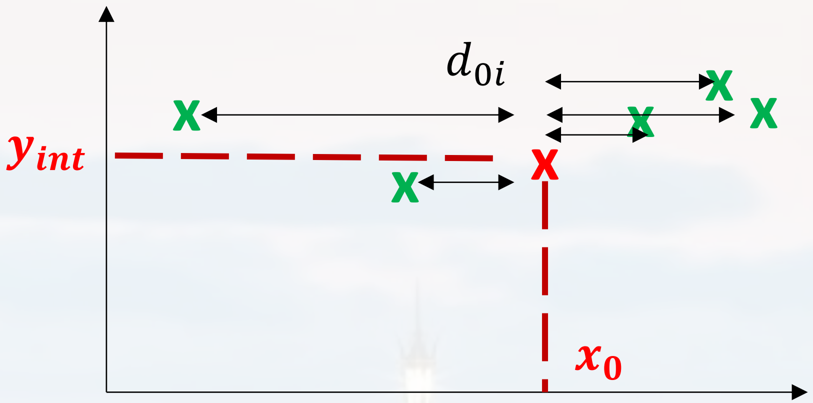
moving averages

better: → weighted average

→ data points **further away** from reference point have **lower weights** w



$$y_{int} \sim \sum_{i=1}^{I} w_i \, y_i \qquad w_i \sim \frac{1}{d_{0i}}$$

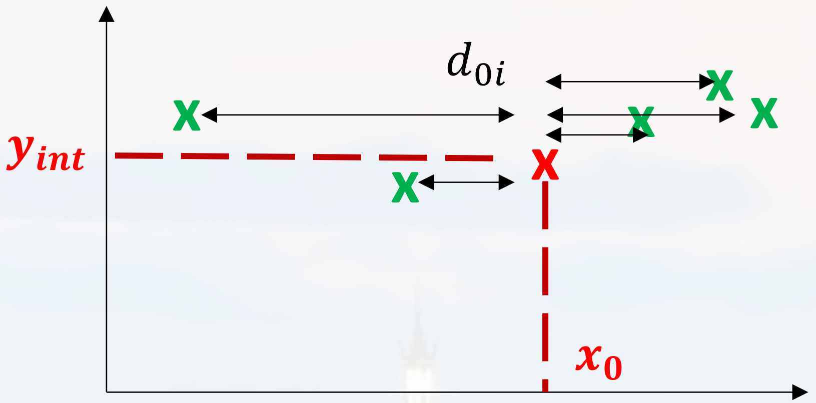


data points **further away** from reference point have **lower weights** w

$$y_{int} \sim \sum_{i=1}^{I} w_i \, y_i \qquad w_i \sim \frac{1}{\boxed{d_{0i}}}$$

```
L = np.random.uniform(0,1,(100,1))

D = np.zeros((len(L),len(L)))

for ii, i in enumerate(L):
    for jj, j in enumerate(L):
        D[ii,jj] = i - j
```

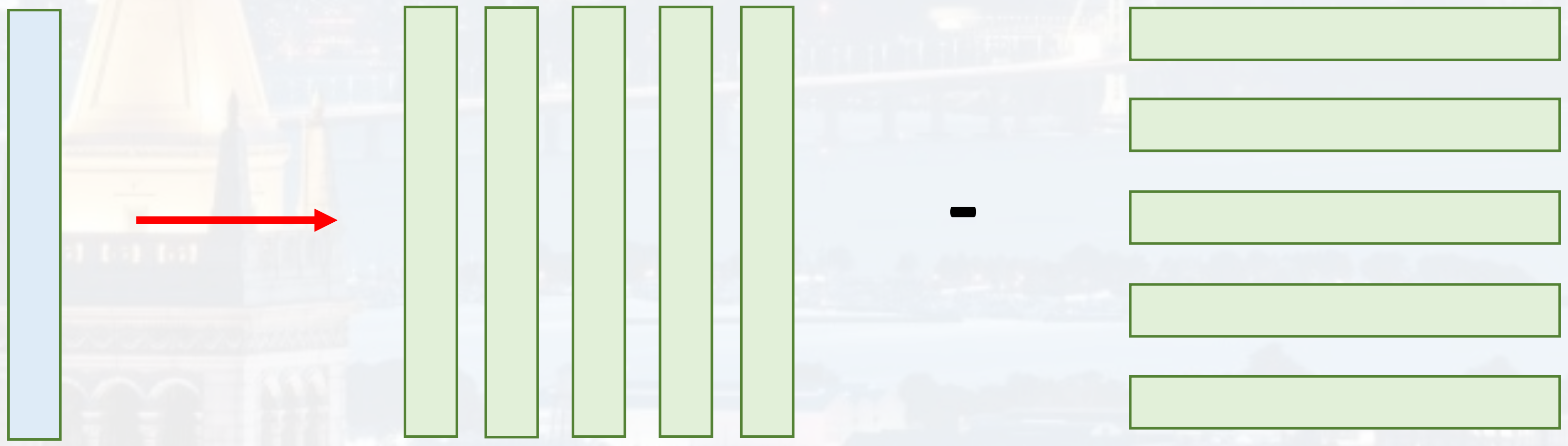
But that is very inefficient!



data points **further away** from reference point have **lower weights** w

$$y_{int} \sim \sum_{i=1}^{I} w_i \, y_i \qquad w_i \sim \frac{1}{\boxed{d_{0i}}}$$

better:

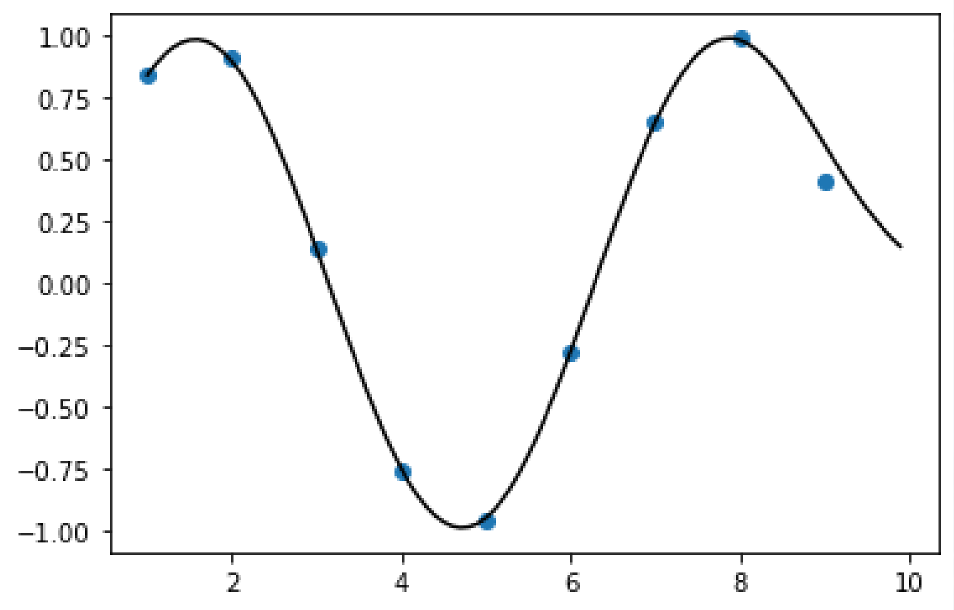


```
L = np.random.uniform(0,1,(100,1))

D = np.tile(L, (1, len(L))) - np.tile(L.transpose(), (len(L), 1))
```

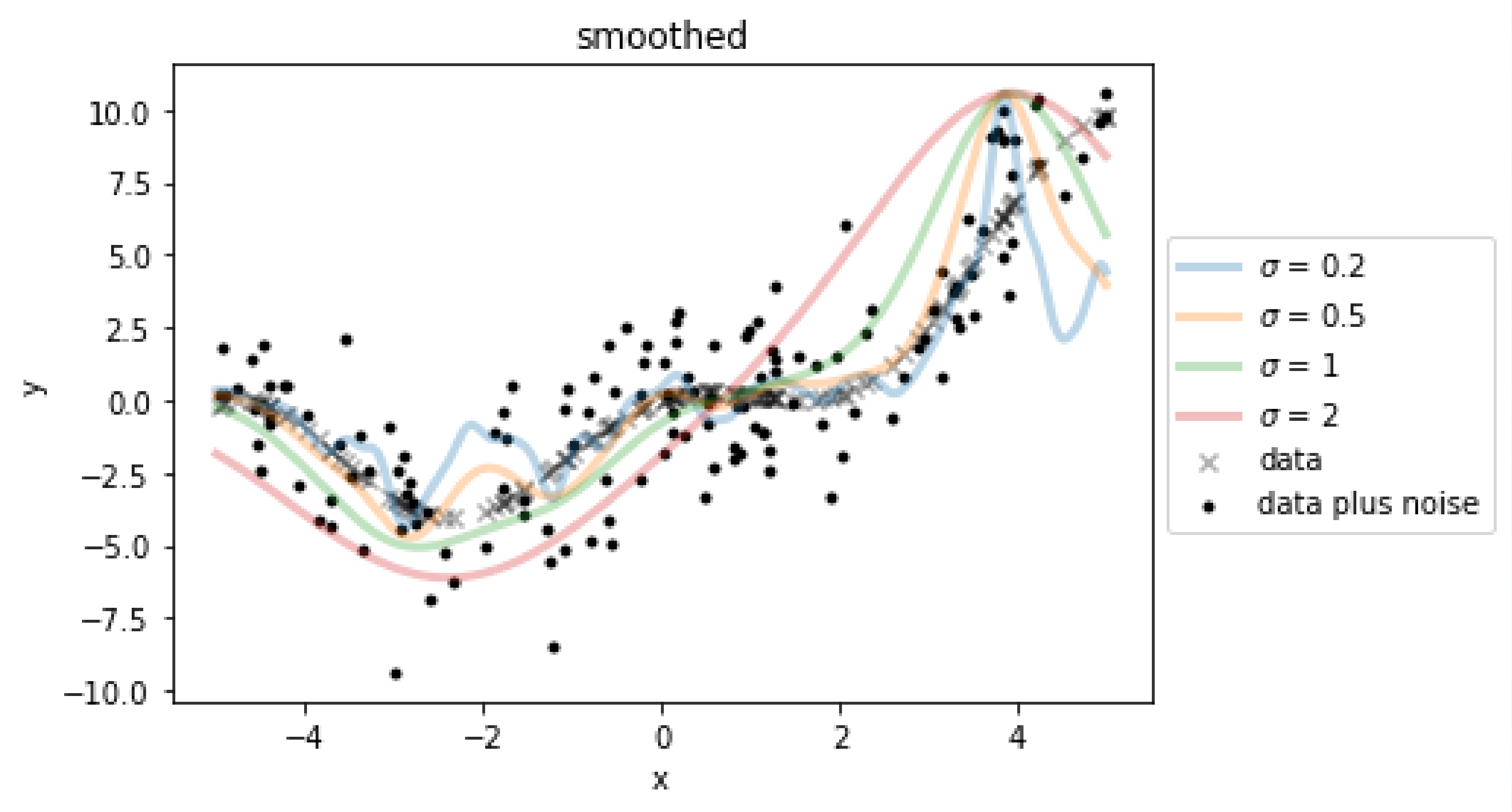
check out: SmoothGaussKernel.py
SmoothExamples.py

```
import numpy as np

def SmoothGaussKernel(x, xint, y, sigma):

    Dx    = np.tile(x.transpose(), (len(xint), 1))

    Dxint = np.tile(xint.transpose(), (len(x), 1))

    D     = Dx.transpose() - Dxint

    W     = np.exp(-(D**2)/(sigma))
    W     = W/np.sum(W)
    yint  = np.dot(W.transpose(), y)
    Scale = np.max(y)/np.max(yint)

    return yint*Scale
```

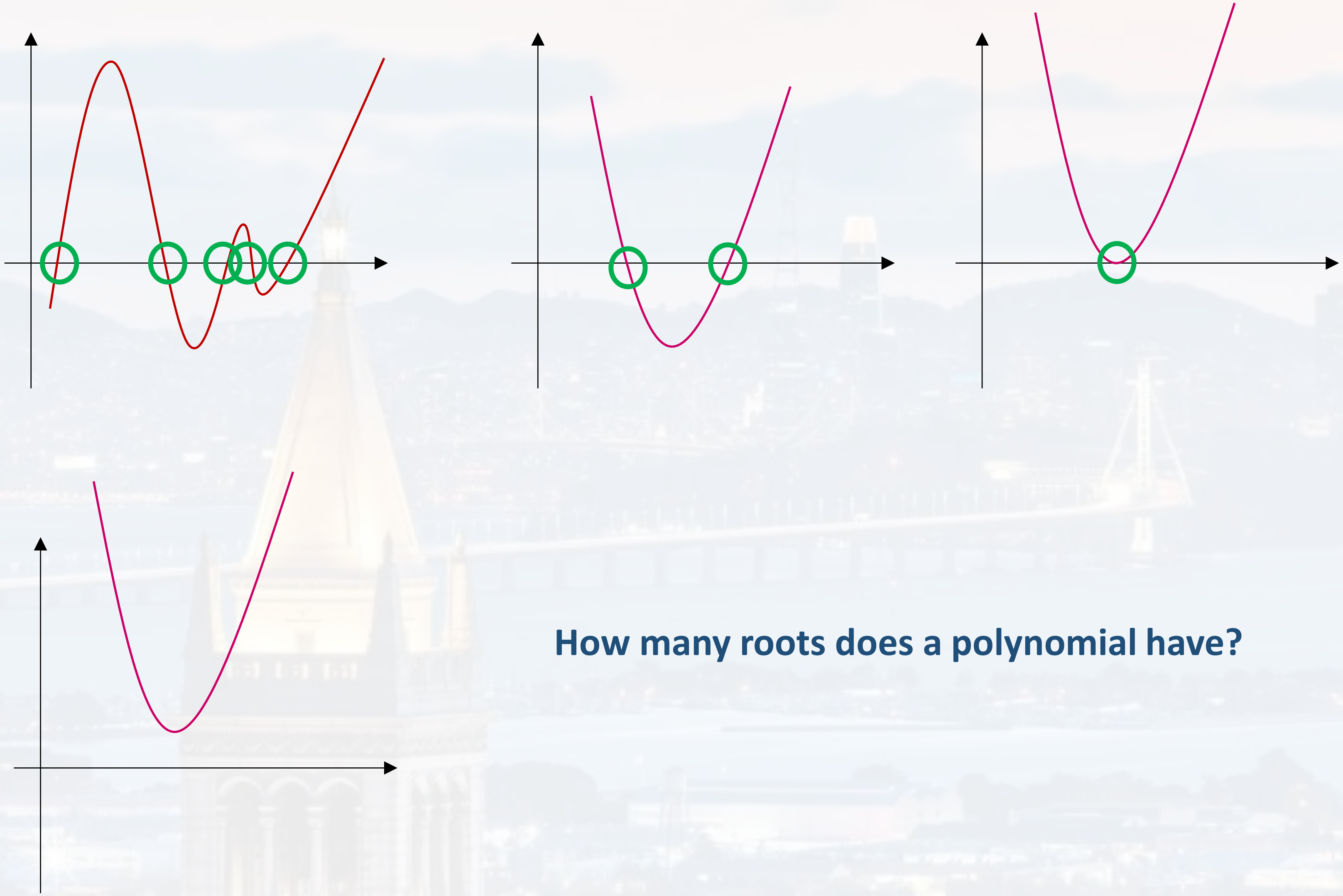
determining how distances are been weighted. Here: normal distribution aka **kernel**



`SmoothExamples.py`



root finding: finding the zeros of a polynomial



How many roots does a polynomial have?

How many roots does a polynomial have?

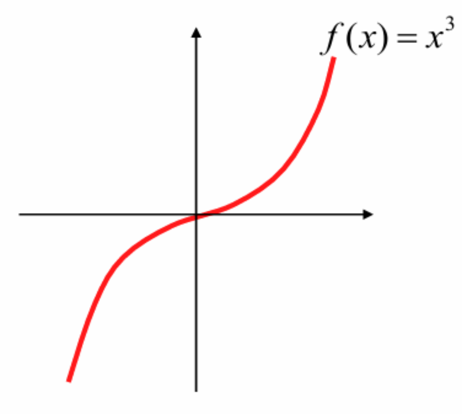
$$f_N(x) = \sum_{i=0}^{N} a_i x^i = \alpha \prod_{i=1}^{N} (x - x_i)$$

factored form

x_i : zeros

- a polynomial of **Nth order** has **N roots** (real & complex)
- for N $\geq$ 5: no analytical solutions
- for N is odd: at least one real zero

$$f(x) = x^3 = (x - x_1)(x - x_2)(x - x_3)$$



zeros: $x_1 = x_2 = x_3 = 0$

one zero with multiplicity m = 3

How many roots does a polynomial have?

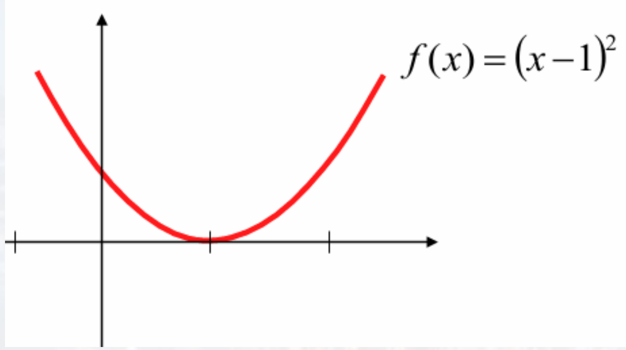
$$f_N(x) = \sum_{i=0}^{N} a_i x^i = \alpha \prod_{i=1}^{N} (x - x_i)$$

factored form

x_i : zeros

- a polynomial of **Nth order** has **N roots** (real & complex)
- for N ≥ 5: no analytical solutions
- for N is odd: at least one real zero

$$f(x) = x^2 - 2x + 1 = (x - x_1)(x - x_2)$$



zeros: $x_1 = x_2 = 1$

one zero with multiplicity m = 2

How many roots does a polynomial have?

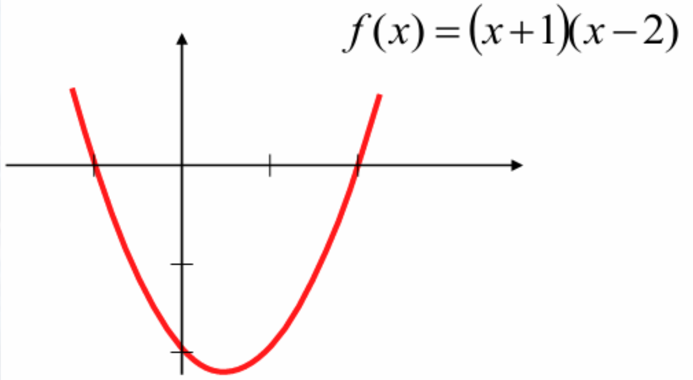
factored form

$$f_N(x) = \sum_{i=0}^{N} a_i x^i = \alpha \prod_{i=1}^{N} (x - x_i)$$

x_i : zeros

- a polynomial of **Nth order** has **N roots** (real & complex)
- for N $\geq$ 5: no analytical solutions
- for N is odd: at least one real zero

$$f(x) = x^2 - x - 2 = (x - x_1)(x - x_2)$$



zeros: $x_1 = 2, \; x_2 = -1$

two zeros with multiplicity m = 1 each

methods:

Root finding [edit]

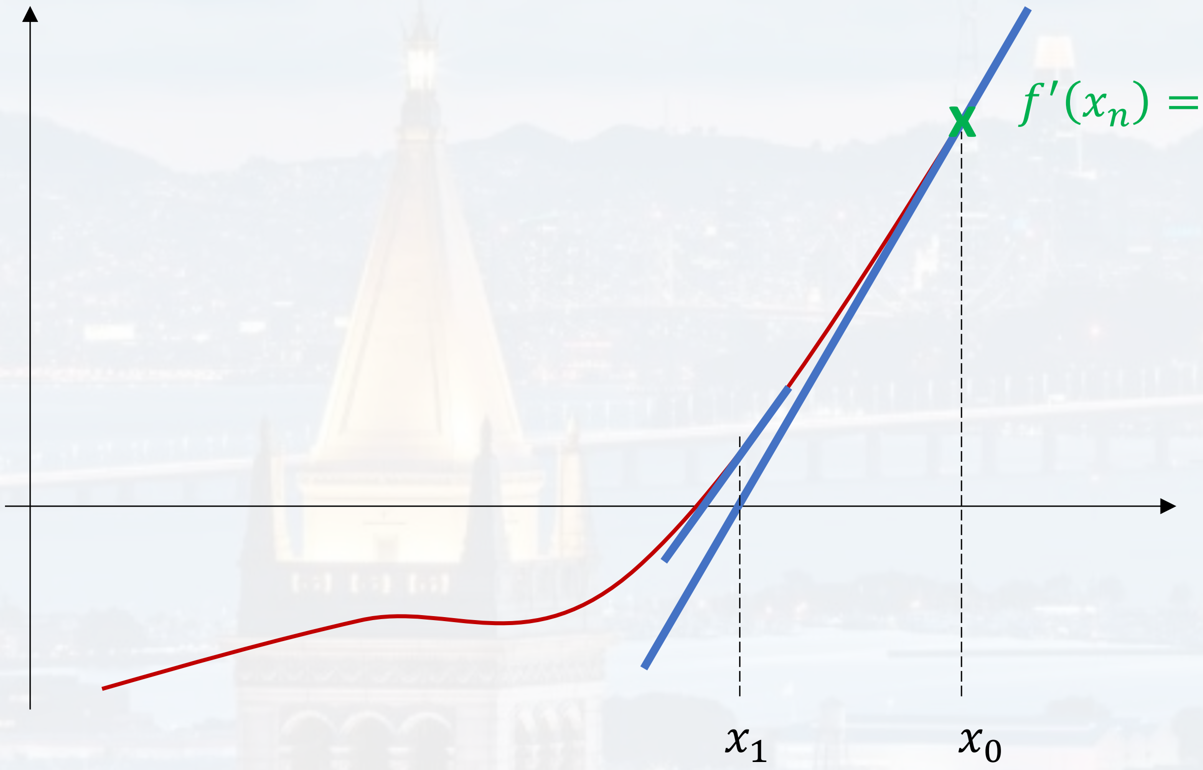
Main article: Root-finding algorithm

- Bisection method
- False position method: and Illinois method: 2-point, bracketing
- Halley's method: uses first and second derivatives
- ITP method: minmax optimal and superlinear convergence simultaneously
- Muller's method: 3-point, quadratic interpolation
- Newton's method: finds zeros of functions with calculus
- Ridder's method: 3-point, exponential scaling
- Secant method: 2-point, 1-sided

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

Newton's method:

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



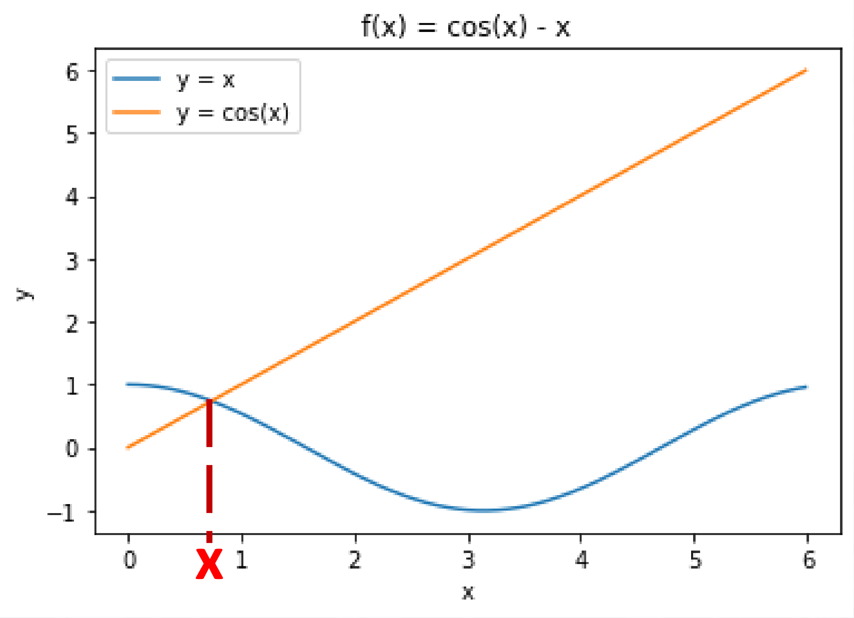
$$f'(x_n) = \frac{f(x_n) - 0}{x_n - \boxed{x_{n+1}}}$$

methods:

Root finding [edit]

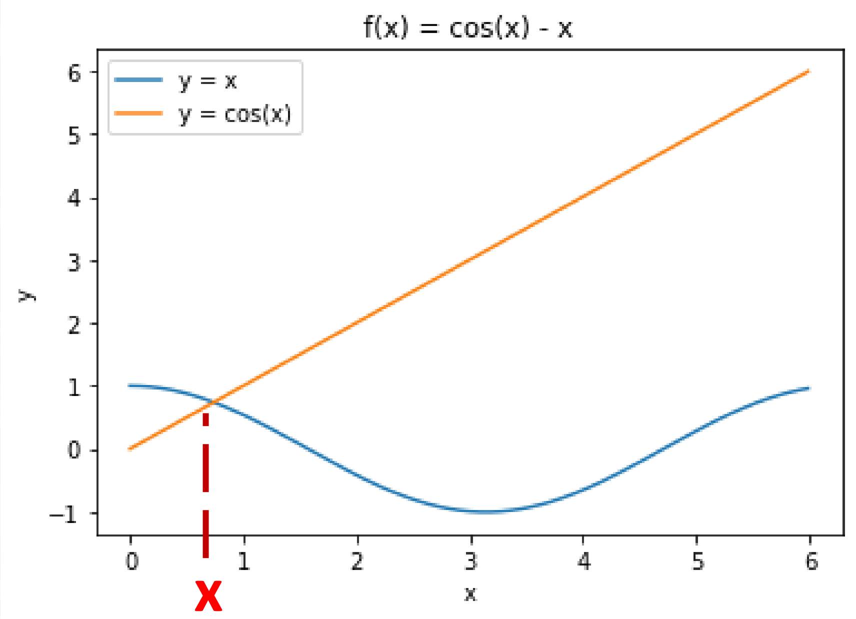
Main article: Root-finding algorithm

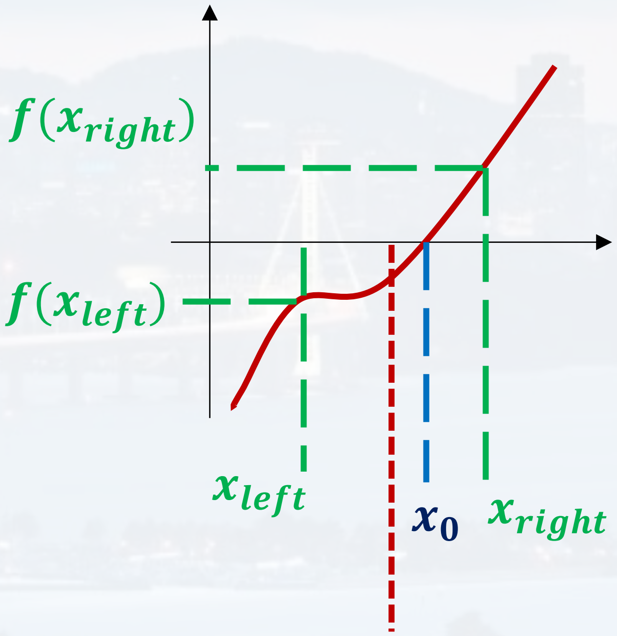
- Bisection method
- False position method: and Illinois method: 2-point, bracketing
- Halley's method: uses first and second derivatives
- ITP method: minmax optimal and superlinear convergence simultaneously
- Muller's method: 3-point, quadratic interpolation
- Newton's method: finds zeros of functions with calculus
- Ridder's method: 3-point, exponential scaling
- Secant method: 2-point, 1-sided



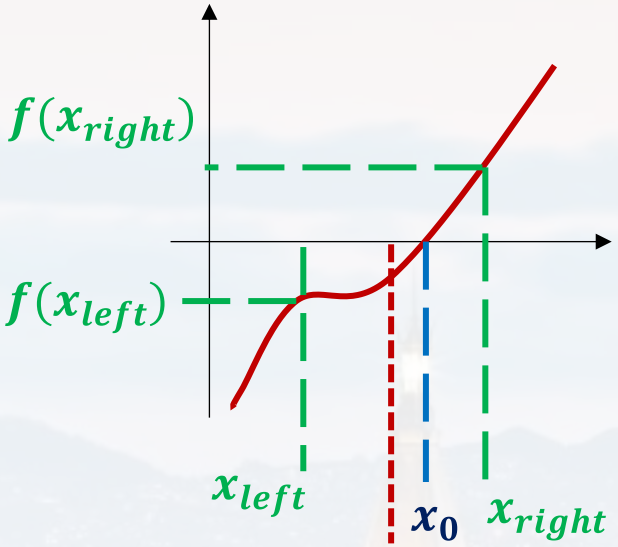
Bisection:

assumption: root is within interval $[x_{left}, x_{right}]$





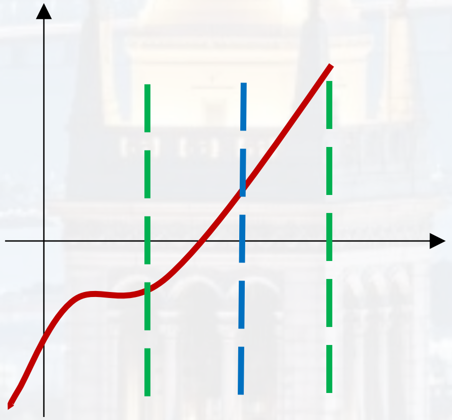
$x_{center} = 0.5\,(x_{left} + x_{right})$

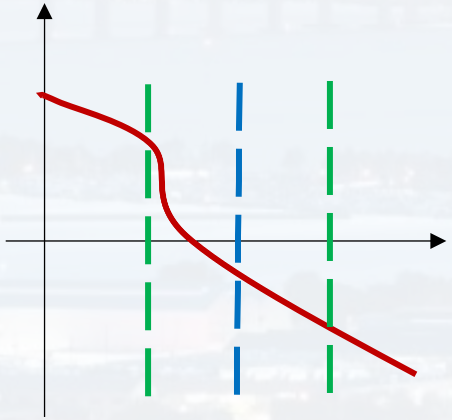


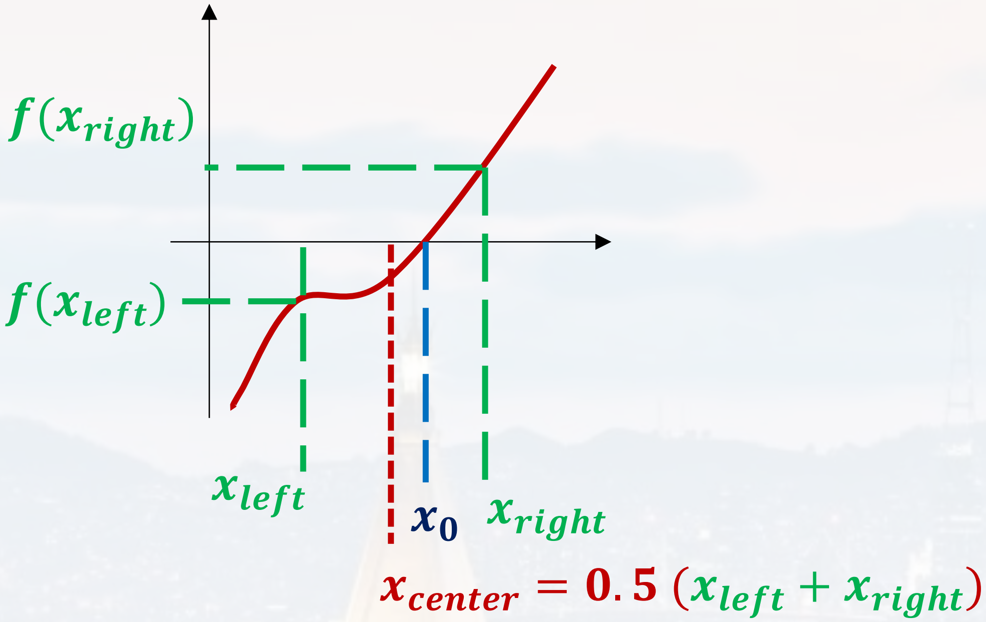
$x_{center} = 0.5\,(x_{left} + x_{right})$

if $f(x_{center}) \cdot f(x_{left}) < 0$

- $x_{left} \rightarrow x_{left}$
- set x_{right} to x_{center}
- reset $x_{center} = 0.5\,(x_{left} + x_{right})$

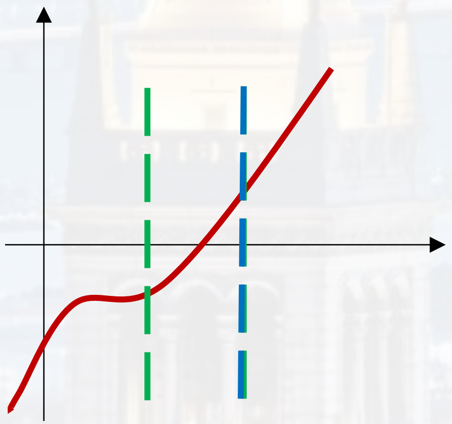


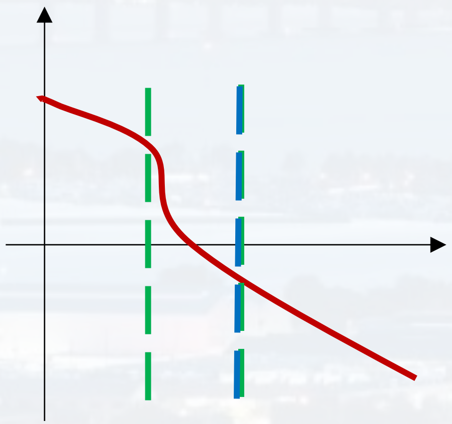


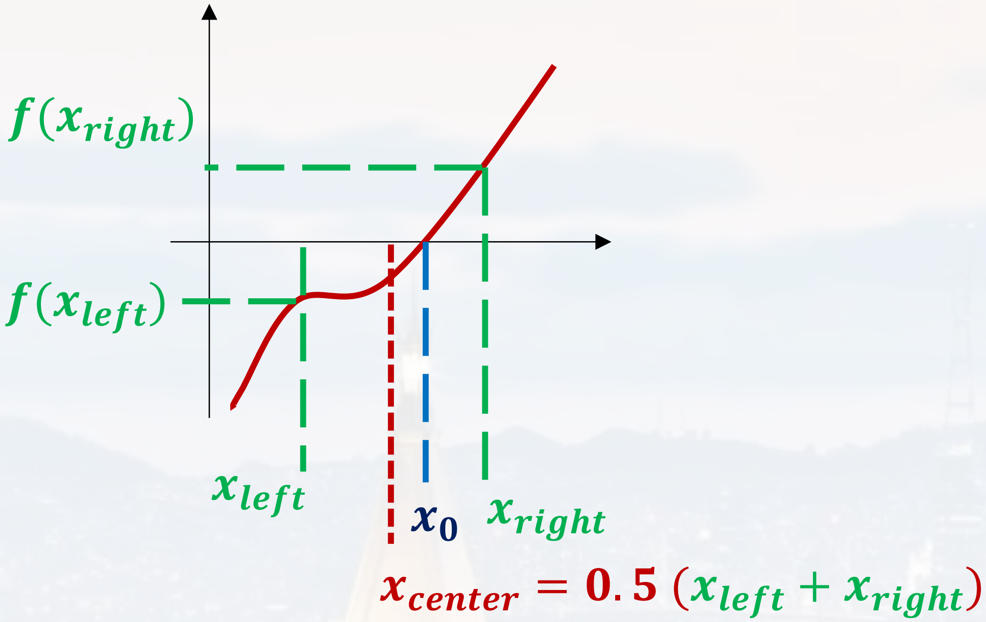


if $f(x_{center}) \cdot f(x_{left}) < 0$

- $x_{left} \rightarrow x_{left}$
- set x_{right} to x_{center}
- reset $x_{center} = 0.5\,(x_{left} + x_{right})$

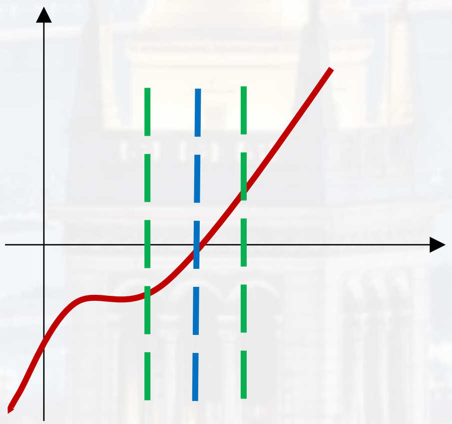


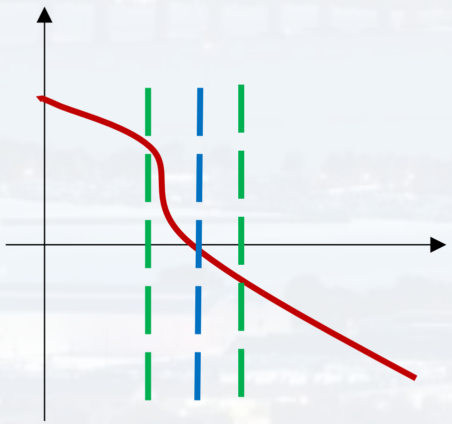




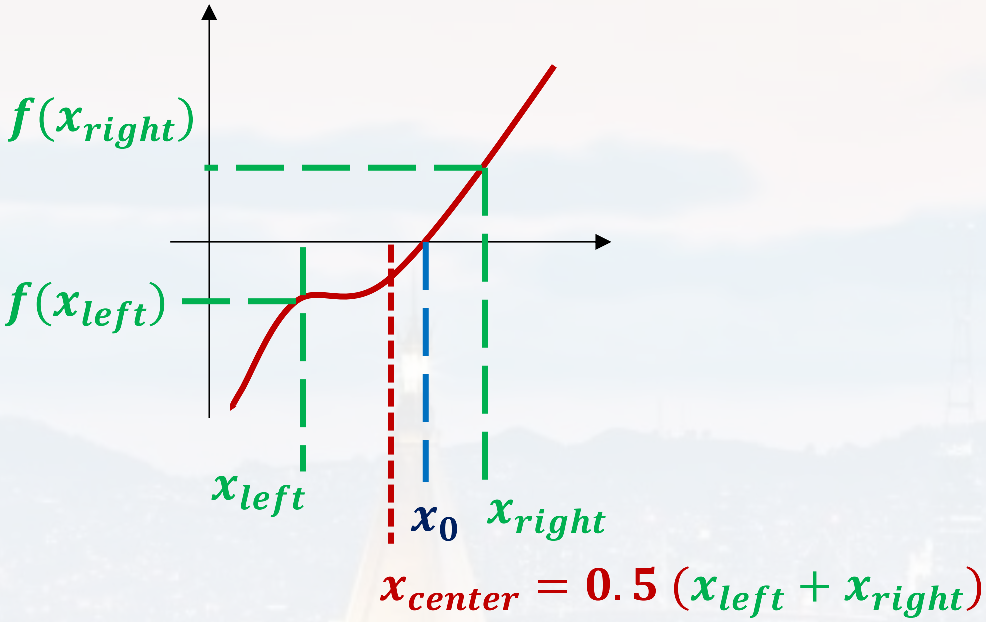
if $f(x_{center}) \cdot f(x_{left}) < 0$

- $x_{left} \rightarrow x_{left}$
- set x_{right} to x_{center}
- reset $x_{center} = 0.5\,(x_{left} + x_{right})$



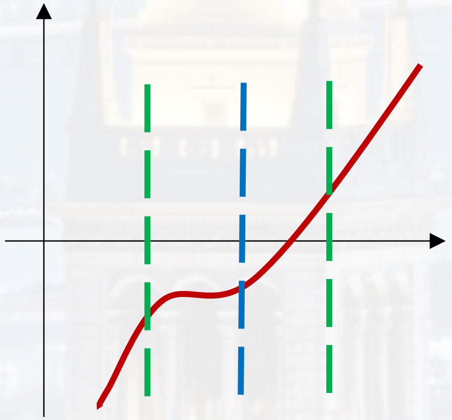


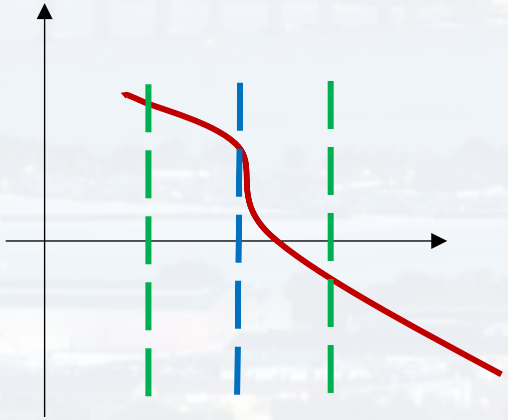
either we end up with the same situation, or…

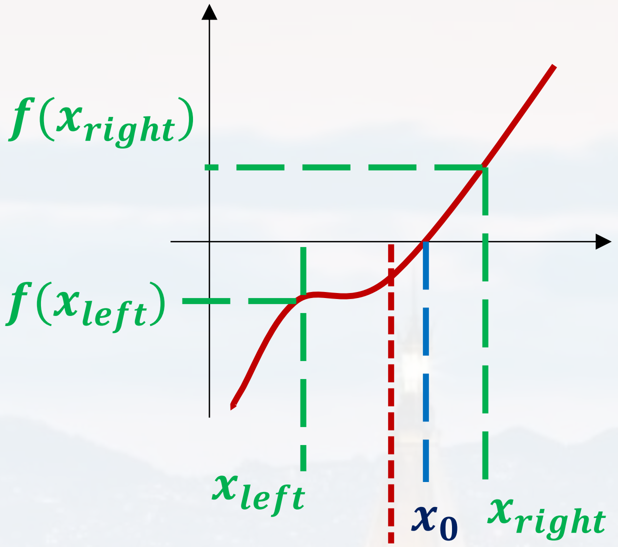


if $f(x_{center}) \cdot f(x_{left}) > 0$

- set x_{left} to x_{center}
- $x_{right} \rightarrow x_{right}$
- reset $x_{center} = 0.5\,(x_{left} + x_{right})$



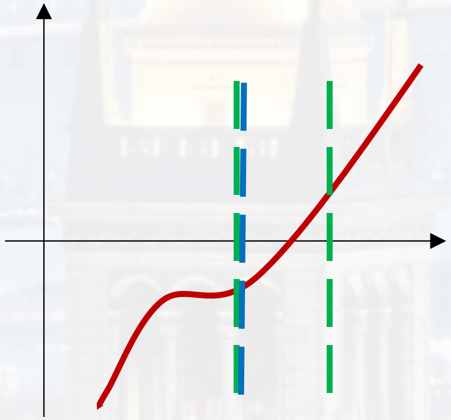


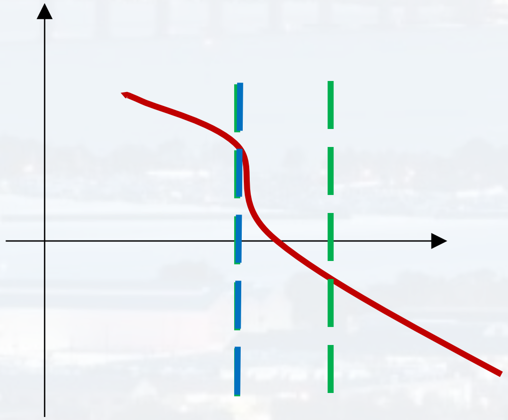


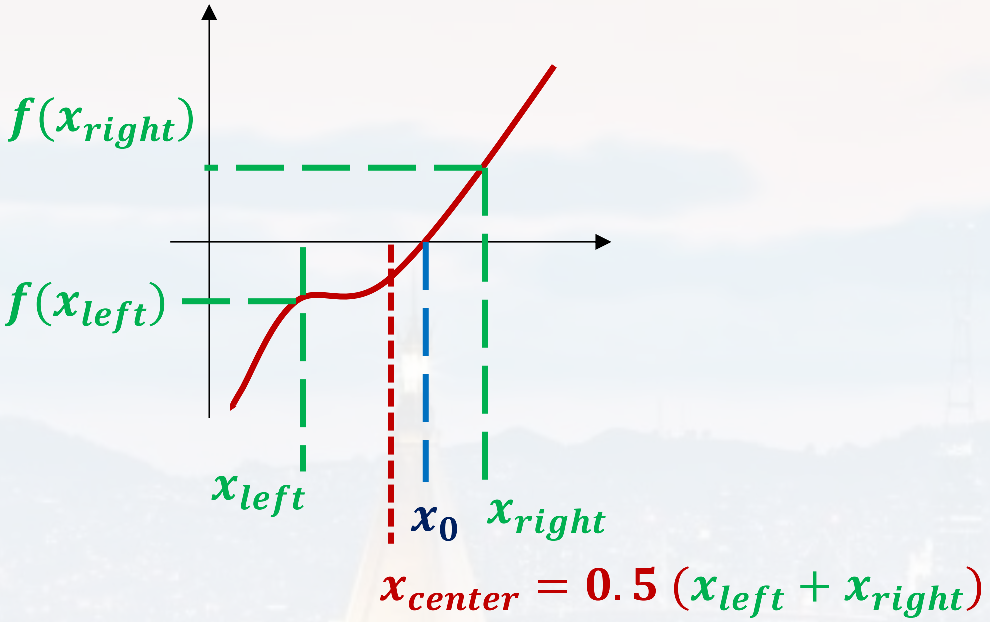
$x_{center} = \mathbf{0.5}\ (x_{left} + x_{right})$

if $f(x_{center}) \cdot f(x_{left}) > 0$

- set x_{left} to x_{center}
- $x_{right} \rightarrow x_{right}$
- reset $x_{center} = \mathbf{0.5}\ (x_{left} + x_{right})$

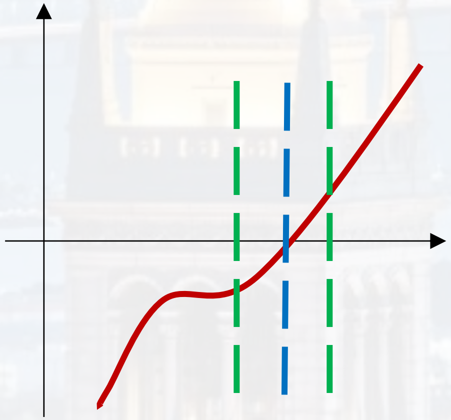


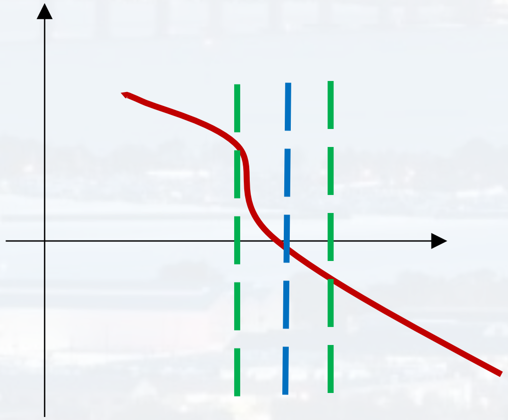




if $f(x_{center}) \cdot f(x_{left}) > 0$

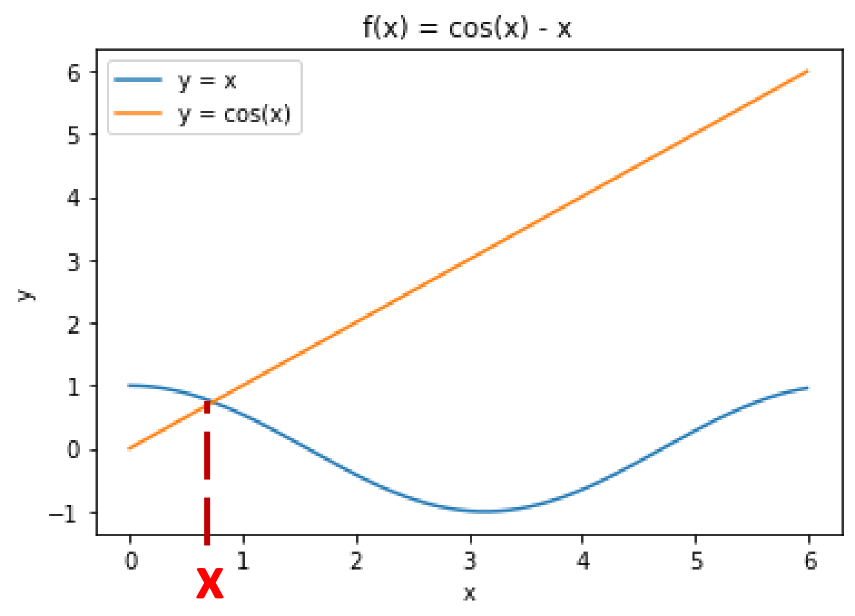
- set x_{left} to x_{center}
- $x_{right} \rightarrow x_{right}$
- reset $x_{center} = 0.5\,(x_{left} + x_{right})$

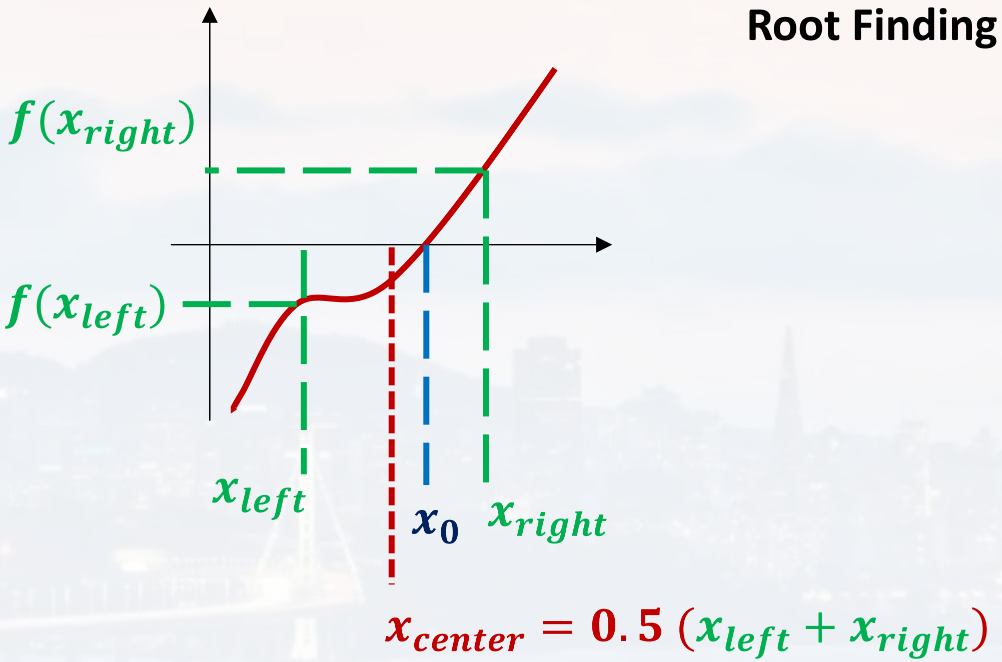




…and so on…

Bisection:





- robust: always finds a root
- easy to implement (recursion), see `Bisection.py`
- slow: converges linearly **(accuracy increases by factor of 2 for each step *n*)** with *n* required for a certain accuracy

- **Newton's method**: since slope of the function points to next x_{n+1} → converges quadratically
- needs derivative → evaluation numerically
- convergence depends on initial guess → might not converge!

methods:

Root finding [edit]

Main article: Root-finding algorithm

- Bisection method
- False position method: and Illinois method: 2-point, bracketing
- Halley's method: uses first and second derivatives
- ITP method: minmax optimal and superlinear convergence simultaneously
- Muller's method: 3-point, quadratic interpolation
- Newton's method: finds zeros of functions with calculus
- Ridder's method: 3-point, exponential scaling
- Secant method: 2-point, 1-sided

M. Hohle:

Thank you for your attention!