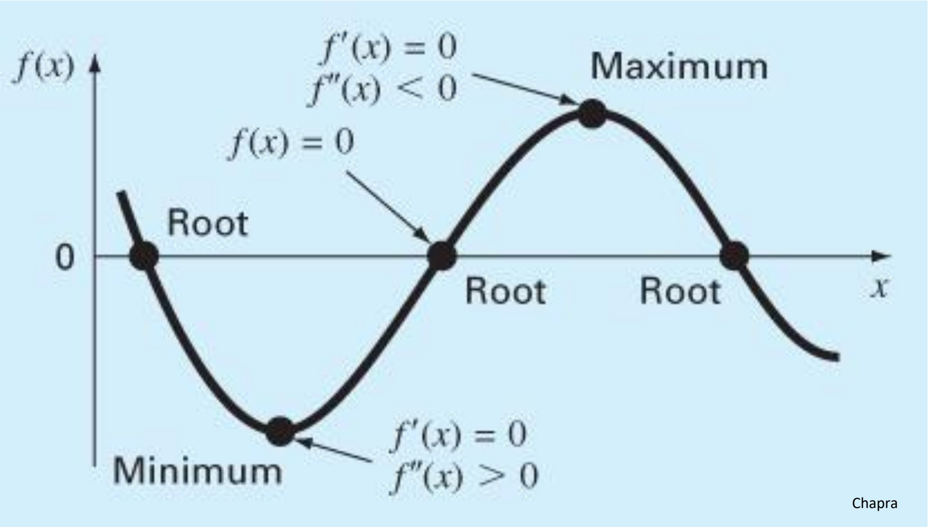
SECTION 2: ROOT FINDING AND OPTIMIZATION

ESC 440 – Computational Methods for Engineers

Root Finding & Optimization

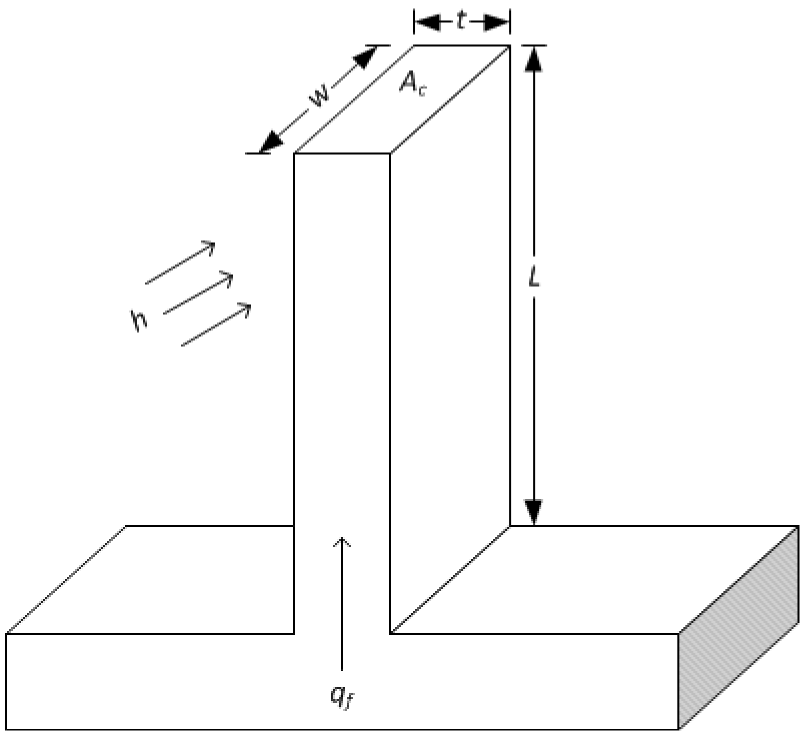
- Two closely related topics covered in this section
  - ***Root finding*** – determination of independent variable values at which the value of a function is ***zero***
  - ***Optimization*** – determination of independent variable values at which the value of a function is at its ***maximum*** or ***minimum*** (***optima***)



Root Finding

Root Finding - Example

- Determine the length, *L*, of a single-fin heat sink to remove 500mW from an electronic package, given the following:
  - **Width**: *w* = 1 cm
  - **Thickness**: *t* = 2 mm
  - **Heat transfer coeff.**:
    $h = 100\ W/(m^2K)$
  - **Aluminum**: k = 210 W/(m·K)
  - **Ambient temperature**:
    $T_\infty = 40°C$
  - **Base temperature**:
    $T_b = 100°C$



Root Finding - Example

- Fin heat transfer rate is given by:

$$q_f = M \cdot \frac{\sinh(mL) + \left(\frac{h}{mk}\right)\cosh(mL)}{\cosh(mL) + \left(\frac{h}{mk}\right)\sinh(mL)}$$

where

$$m = \sqrt{\frac{hP}{kA_c}}, \quad M = \sqrt{hPkA_c} \cdot \theta_b$$

$$A_c = w \cdot t, \quad P = 2w + 2t$$

$$\theta_b = T_b - T_\infty$$

Root Finding - Example

- Would like to set $q_f = 500mW$ and solve for L, given all other parameters
  - But, we can't isolate L – a ***transcendental equation*** – can't be solved algebraically
- Instead, subtract $500mW$ from both sides

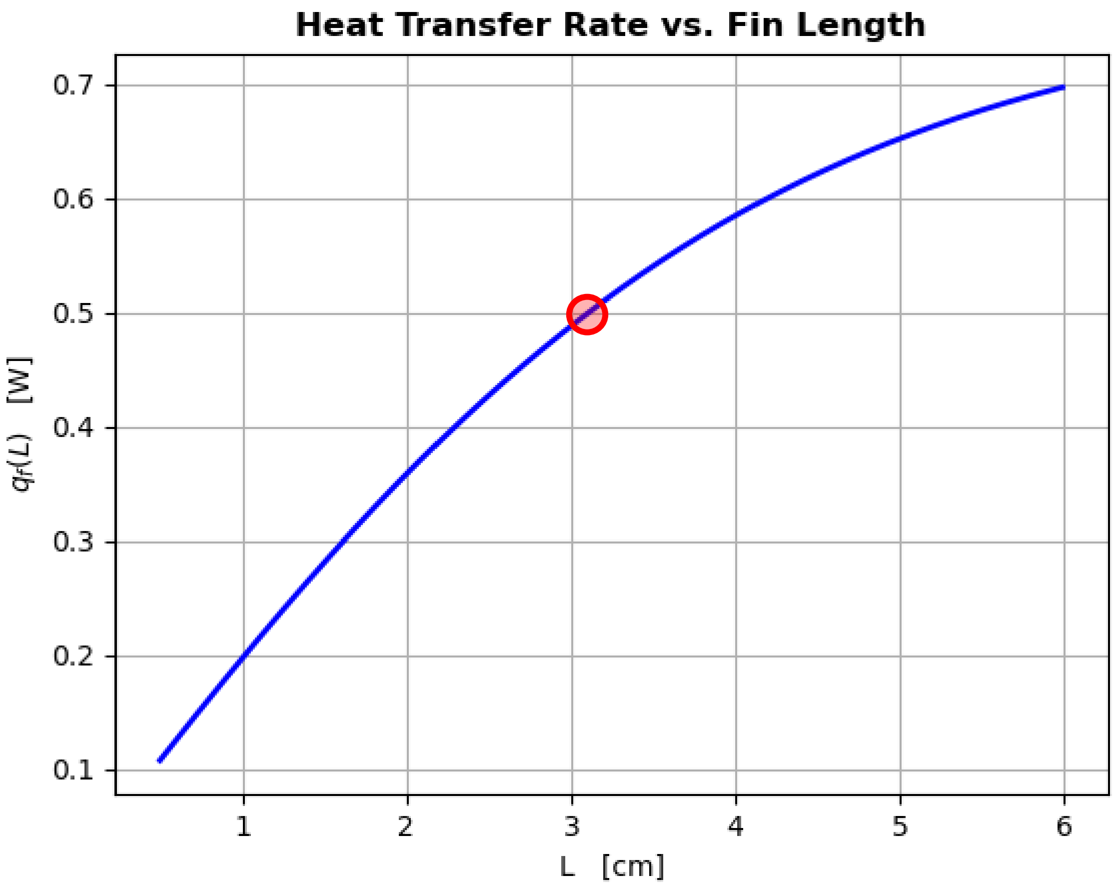
$$f(L) = q_f(L) - 500mW$$

$$f(L) = M \cdot \frac{\sinh(mL) + \left(\frac{h}{mk}\right)\cosh(mL)}{\cosh(mL) + \left(\frac{h}{mk}\right)\sinh(mL)} - 500mW = 0$$

- Now, find the value of L for which $f(L) = 0$
  - A ***<u>root-finding problem</u>***

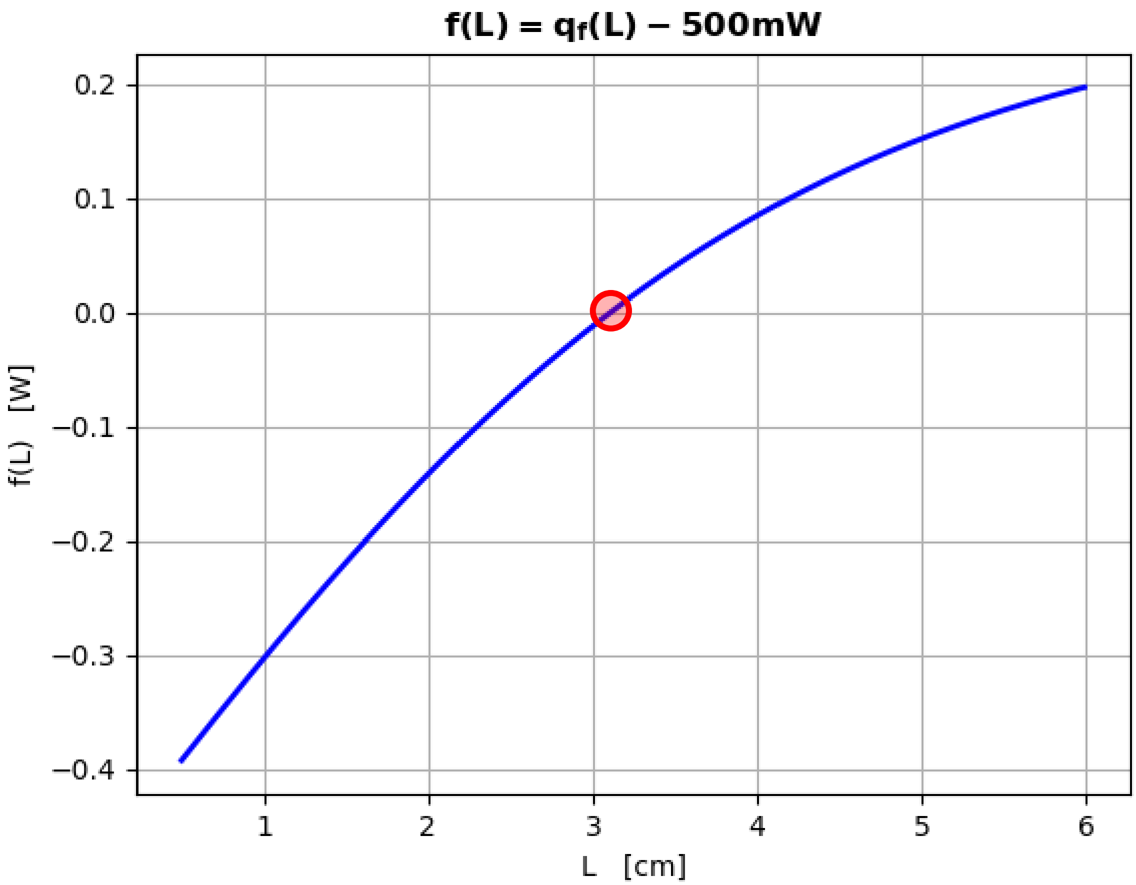
Root Finding - Example

- Looking for L such that $q_f(L) = 500mW$



Root Finding - Example

- Find the root of $f(L)$, i.e. L such that $f(L) = 0$



Root-Finding Techniques – Bracketing vs. Open

- Two categories of root-finding methods:
- ***Bracketing methods***
  - Require two initial values – must bracket (one on either side of) the root
  - Always converge
  - Can be slow
- ***Open methods***
  - Initial value(s) need not bracket the root
  - Often faster
  - May not converge

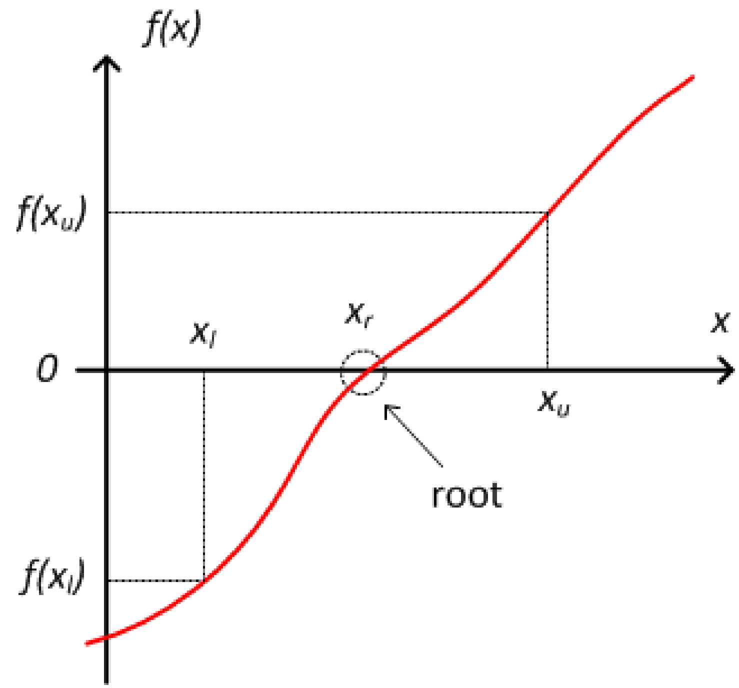
Root Finding: Basic Concepts

Presence of a Root – Sign Change

- A ***root*** is a value of x at which $f(x) = 0$
  - $f(x)$ ***crosses the x-axis***
  - $f(x)$ ***changes sign***
- If x_r is a root of $f(x)$, and $x_l < x_r < x_u$, then

$$f(x_l) \cdot f(x_u) < 0$$

- Not always true
  - e.g., ***multiple roots***
  - Won't consider multiple roots here



Error Evaluation and Tracking

- ***Approximate error,*** $|\boldsymbol{\varepsilon_a}|$
  - Don't know where the true root is, so must approximate error

$$|\boldsymbol{\varepsilon_a}| = \left|\frac{\widehat{\boldsymbol{x}}_{r,i+1} - \widehat{\boldsymbol{x}}_{r,i}}{\widehat{\boldsymbol{x}}_{r,i+1}}\right| \cdot \mathbf{100\%}$$

  - Tells us when a root has been determined to adequate precision – stop when $|\boldsymbol{\varepsilon_a}| \leq |\boldsymbol{\varepsilon_s}|$
- ***True error,*** $|\boldsymbol{\varepsilon_t}|$
  - Useful for evaluating the performance of root-finding algorithms – when we know the location of the root

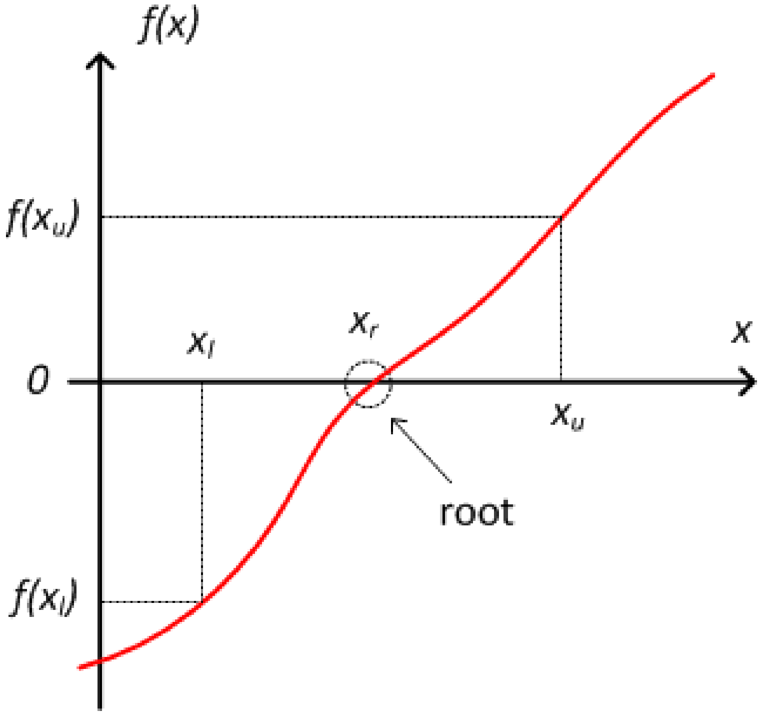
Root Finding: Bracketing Methods

Root Finding – Bracketing Methods

- We'll look at three ***bracketing methods***
  - ***Incremental search***
  - ***Bisection***
  - ***False position***
- Each require ***two initial values***
  - ***Must bracket the root***

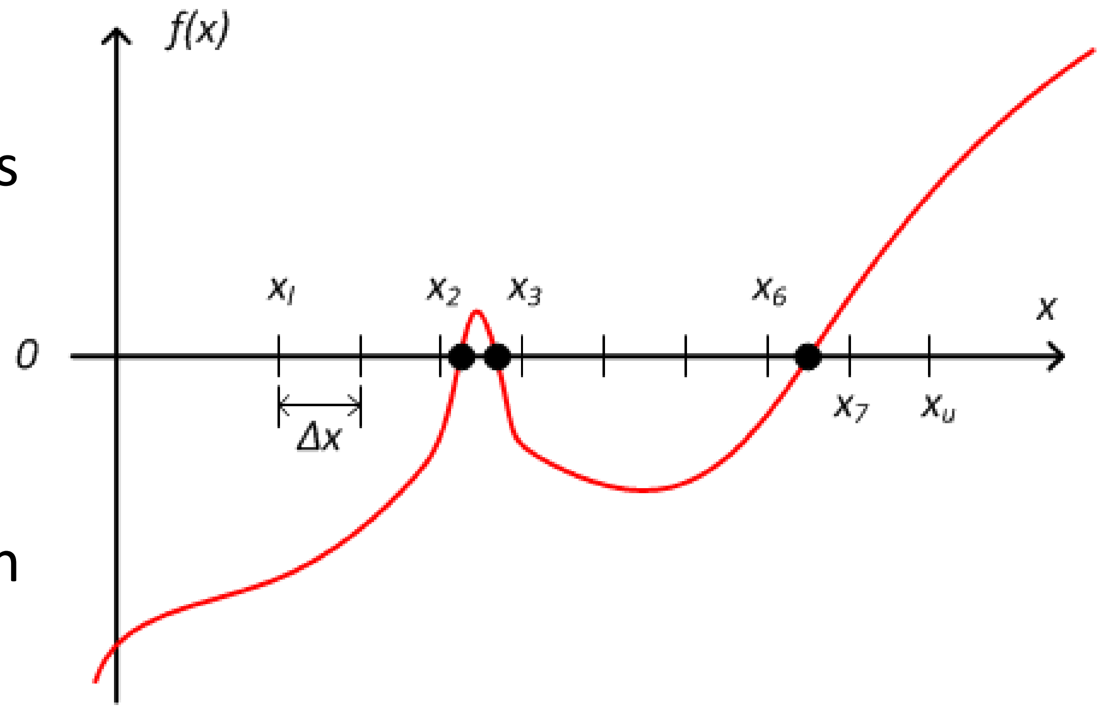
Incremental Search

- Say we want to find a root, x_r, which we know exists between x_l and x_u
- Initialize the search with bracketing values
- Starting at x_l, move incrementally toward x_u, searching for a ***sign change*** in $f(x)$
- Accuracy determined by ***increment length***
  - Too large – inaccurate – could miss closely spaced roots
  - Too small - slow



Incremental Search

- $f(x)$ has three roots on $[x_l, x_u]$
- Incremental search with increment length, Δx
- $f(x_2) \cdot f(x_3) > 0$
  - Closely-spaced roots are missed entirely
- $f(x_6) \cdot f(x_7) < 0$
  - A root is detected
  - Location only known to within Δx
  - $|E_t| < \Delta x$



Bisection

Bisection

- Search initialized with bracketing values
- Current root estimate, $\hat{x}_{r,i}$, is the midpoint of the current interval

$$\hat{x}_{r,i} = \frac{x_{l,i} + x_{u,i}}{2}$$

- At each iteration, root estimate replaces upper or lower bracketing value

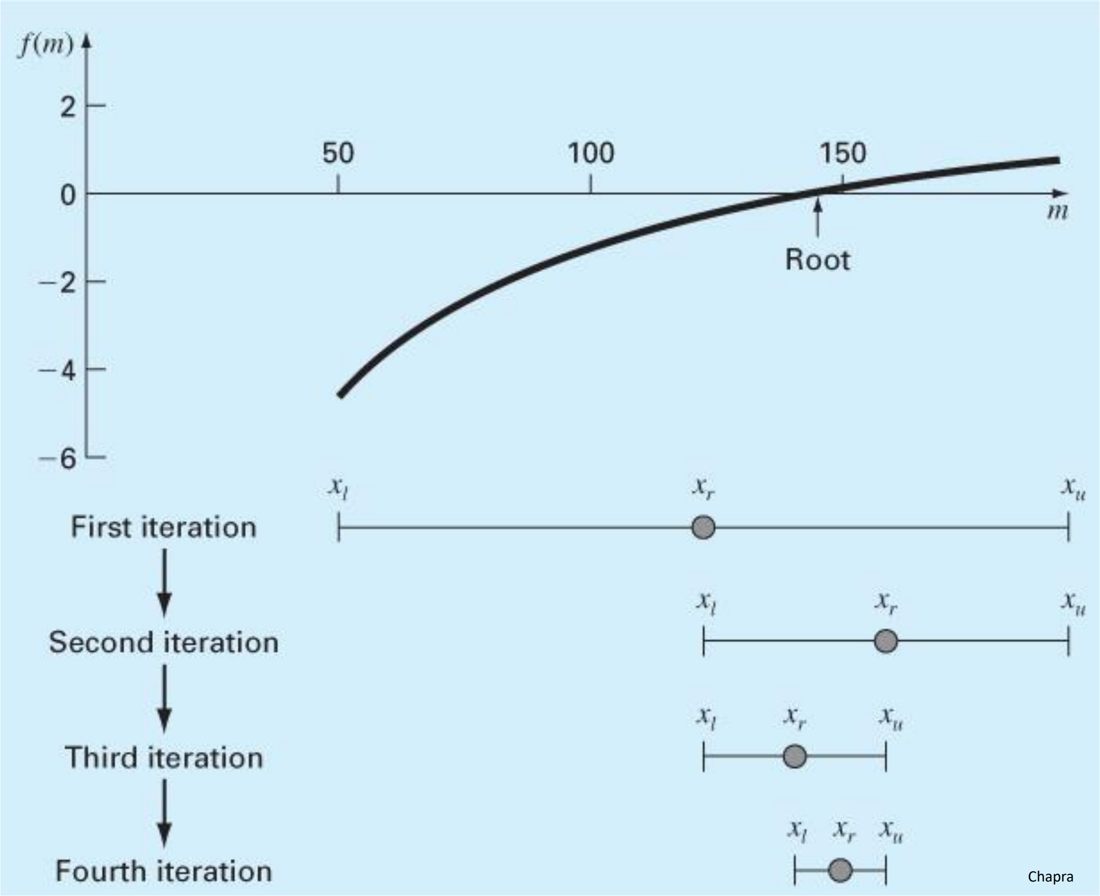
$$x_{l,i+1} = \begin{cases} x_{l,i} & f\left(x_{l,i}\right) \cdot f\left(\hat{x}_{r,i}\right) < 0 \\ \hat{x}_{r,i} & f\left(x_{l,i}\right) \cdot f\left(\hat{x}_{r,i}\right) \geq 0 \end{cases}$$

$$x_{u,i+1} = \begin{cases} x_{u,i} & f\left(x_{u,i}\right) \cdot f\left(\hat{x}_{r,i}\right) < 0 \\ \hat{x}_{r,i} & f\left(x_{u,i}\right) \cdot f\left(\hat{x}_{r,i}\right) \geq 0 \end{cases}$$

Bisection

At each iteration:

- ***Root estimate***
  - midpoint of bracketing interval
- ***New bracketing interval***
  - sub-interval containing the sign change



Bisection – Absolute Error

- Absolute error is bounded by the bracketing interval

$$|E_{t,i}| \leq \frac{\Delta x_i}{2} = \frac{(x_{u,i} - x_{l,i})}{2}$$

- Bracketing interval halved at each iteration
  - Max absolute error halved each iteration. After n iterations:

$$|E_{t,n}| \leq \frac{\Delta x_0}{2^{n+1}}$$

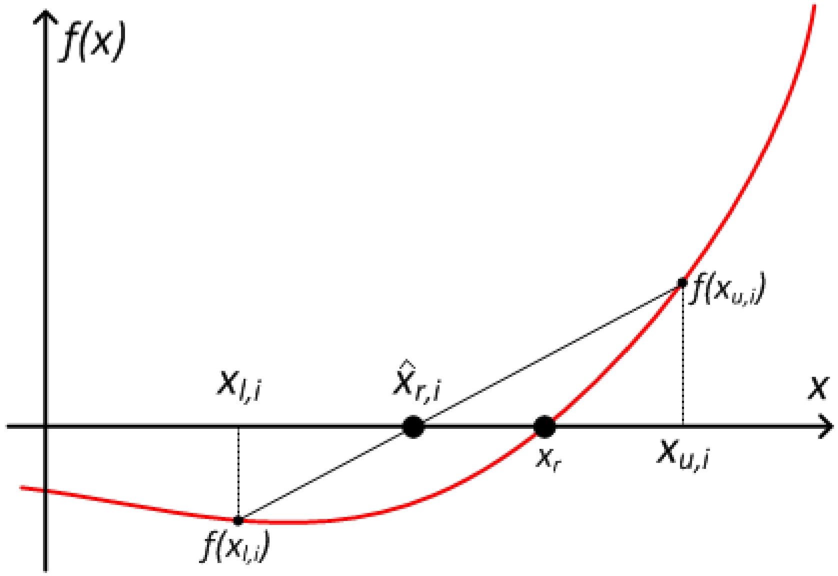
- Can calculate required iterations for a specified maximum absolute error:

$$n = \log_2\left(\frac{\Delta x_0}{E_t}\right) - 1$$

False Position

False Position – Linear Inerpolation

- Similar to bisection, but root estimate calculated differently
  - Not the midpoint of the bracketing interval
  - $\hat{x}_{r,i}$ is the ***root of the line*** connecting $f\left(x_{l,i}\right)$ and $f\left(x_{u,i}\right)$



False Position – Calculating $\hat{x}_{r,i}$

- Slope of the line:

$$\frac{\Delta y}{\Delta x} = \frac{f\left(x_{u,i}\right) - f\left(x_{l,i}\right)}{x_{u,i} - x_{l,i}}$$

- From $f\left(x_{u,i}\right)$ to zero:

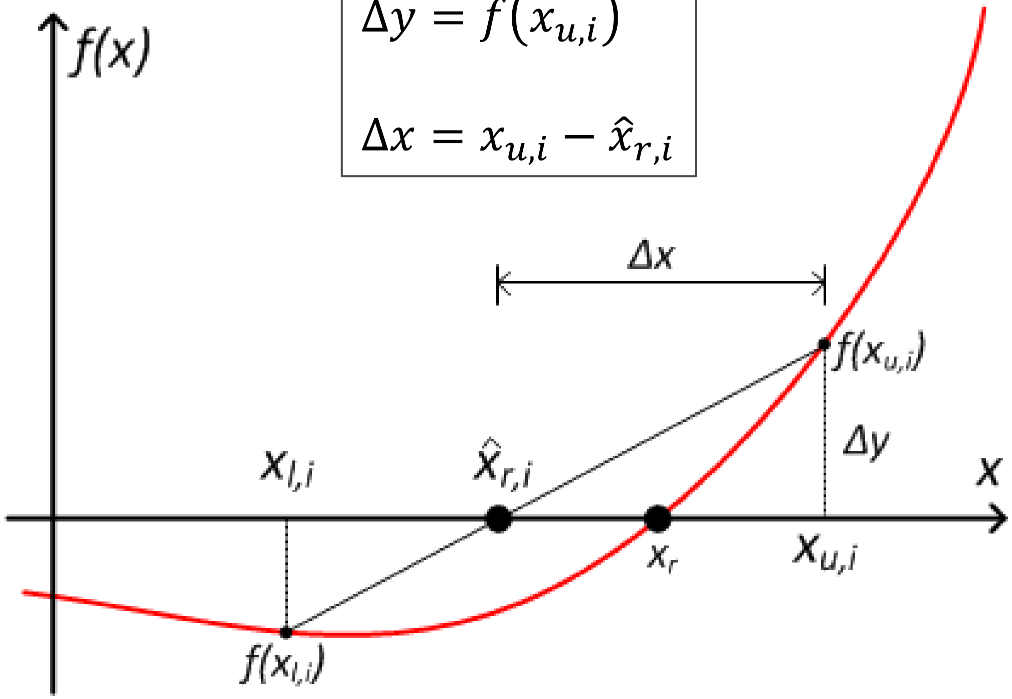
$$\Delta y = f\left(x_{u,i}\right)$$

- From $x_{u,i}$ to $\hat{x}_{r,i}$:

$$\Delta x = \frac{\Delta x}{\Delta y} \cdot f\left(x_{u,i}\right)$$

$$\Delta y = f\left(x_{u,i}\right)$$

$$\Delta x = x_{u,i} - \hat{x}_{r,i}$$



- The root estimate is:

$$\hat{x}_{r,i} = x_{u,i} - \Delta x \quad \rightarrow \quad \boxed{\hat{x}_{r,i} = x_{u,i} - f\left(x_{u,i}\right)\frac{x_{u,i} - x_{l,i}}{f\left(x_{u,i}\right) - f\left(x_{l,i}\right)}}$$

False Position – Reducing the Bracket

- As with bisection, the bracket is reduced on each iteration
  - Keep the sub-bracket containing the sign change
  - Root estimate replaces upper or lower bracketing value

$$x_{l,i+1} = \begin{cases} x_{l,i} & f(x_{l,i}) \cdot f(\hat{x}_{r,i}) < 0 \\ \hat{x}_{r,i} & f(x_{l,i}) \cdot f(\hat{x}_{r,i}) \geq 0 \end{cases}$$

$$x_{u,i+1} = \begin{cases} x_{u,i} & f(x_{u,i}) \cdot f(\hat{x}_{r,i}) < 0 \\ \hat{x}_{r,i} & f(x_{u,i}) \cdot f(\hat{x}_{r,i}) \geq 0 \end{cases}$$

Bracketing Methods - Summary

- All methods require ***two initial values that bracket the root***
- ***Always convergent***
  - ***<u>Incremental search</u>***
    - Mostly for illustrative purposes – not recommended
  - ***<u>Bisection</u>***
    - Predictable
    - Can calculate required iterations for desired absolute error - predictable
  - ***<u>False position – linear interpolation</u>***
    - Often outperforms bisection
    - May be slow for certain types of functions

Root Finding: Open Methods

Root Finding – Open Methods

- May require only a single initial value
- If two initial values are required, they need not bracket the root
- ***Often significantly faster*** than bracketing methods
- Convergence is not guaranteed
  - Dependent on function and initial values

- ***<u>Fixed-point iteration</u>***
- ***<u>Newton-Raphson</u>***
- ***<u>Secant methods</u>***
- ***<u>Inverse quadratic interpolation</u>***

Fixed Point Iteration

Fixed Point Iteration

- A ***fixed point*** of a function is a value of the independent variable that the function ***maps to itself***
- Root-finding problem is determining x, such that

$$f(x) = 0$$

- Can ***add x to both sides*** – equation is unchanged

$$x = f(x) + x$$
$$x = g(x)$$

- Value of x that satisfies the equation is ***still the root***

Fixed Point Iteration

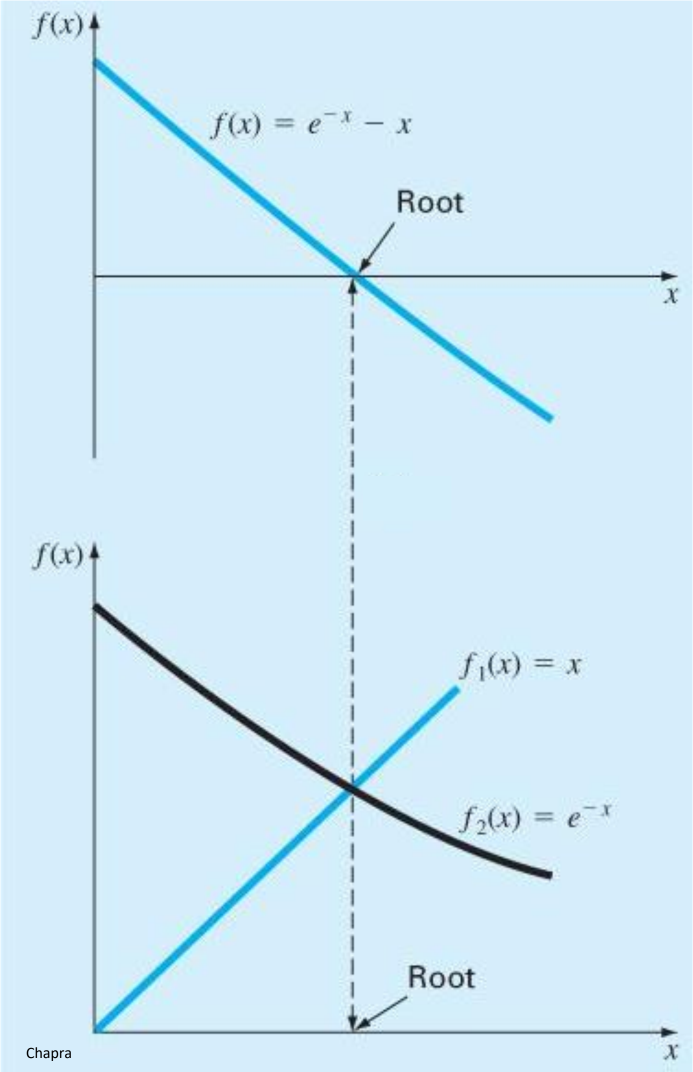
- Root is the solution to

$$x = g(x)$$

  - A ***fixed point*** of $g(x)$
- Also the solution to ***system of two equations***

$$f_1(x) = x$$
$$f_2(x) = g(x)$$

- Root is the ***intersection*** of $f_1(x)$ and $f_2(x)$
  - i.e., the intersection of $y = f(x) + x$ and $y = x$



Fixed Point Iteration

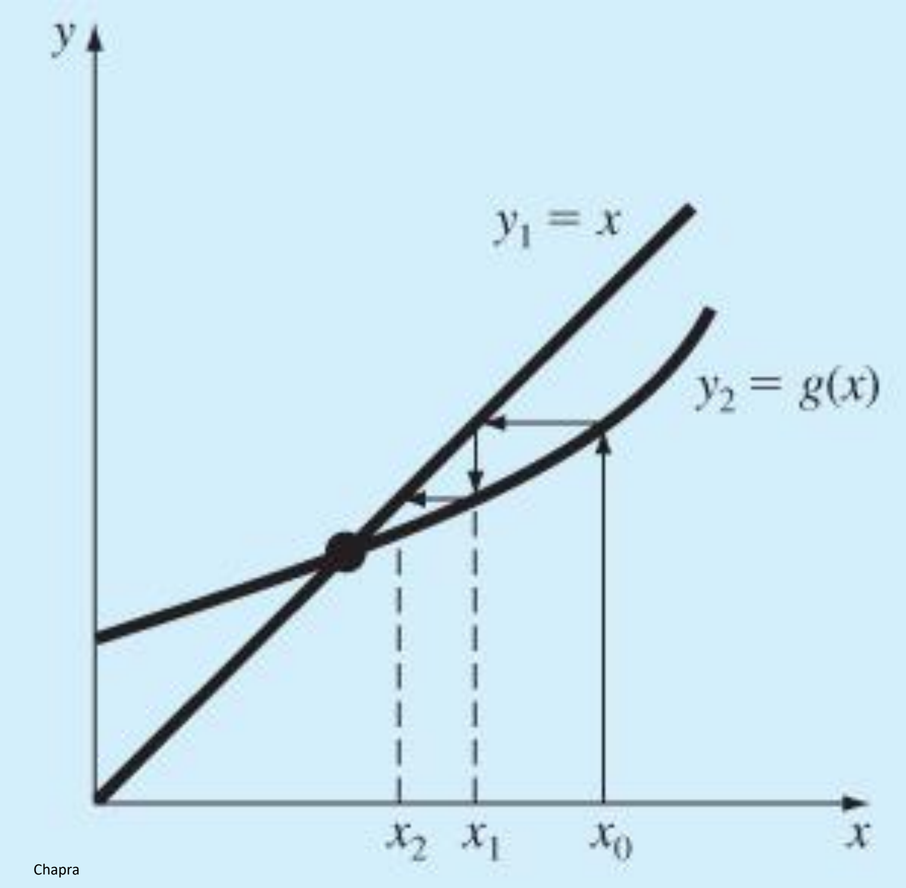
$$x = g(x)$$

- Provides an iterative formula for x:

$$x_{i+1} = g(x_i)$$

- Iterate until approximate error falls below a specified stopping criterion

$$|\varepsilon_a| \le \varepsilon_s$$

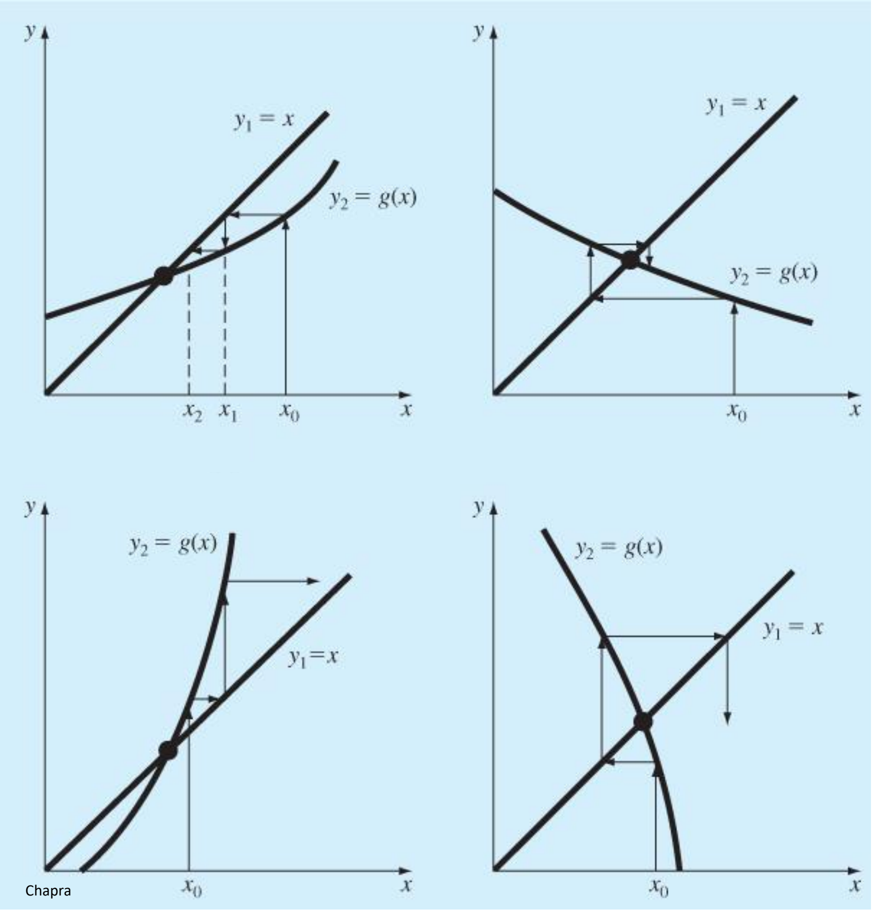


Fixed Point Iteration – Convergence

- ***Current error is proportional to the previous error times the slope of $g(x)$:***

$$E_{t,i+1} = g'(\xi) \cdot E_{t,i}$$

- If $|g'(x)| > 1$, error will grow
  - Estimate will ***diverge***
- If $|g'(x)| < 1$, error will decrease
  - Estimate will ***converge***
- If $g'(x) < 0$, sign of error will oscillate
  - ***Oscillatory***, or ***spiral*** convergence or divergence



Fixed Point Iteration – Rate of Convergence

- ***Current error is proportional to the previous error times the slope of $g(x)$***:

$$E_{t,i+1} = g'(\xi) \cdot E_{t,i}$$

- Once a convergent estimate becomes relatively close to the root, the ***slope of $g(x)$ is relatively constant***
  - $\hat{x}_r$ varies little from iteration to iteration
- Error of the current iteration is roughly ***proportional to the error from the previous iteration***
  - ***<u>Linear convergence</u>***

Newton-Raphson & Secant Methods

Newton-Raphson Method

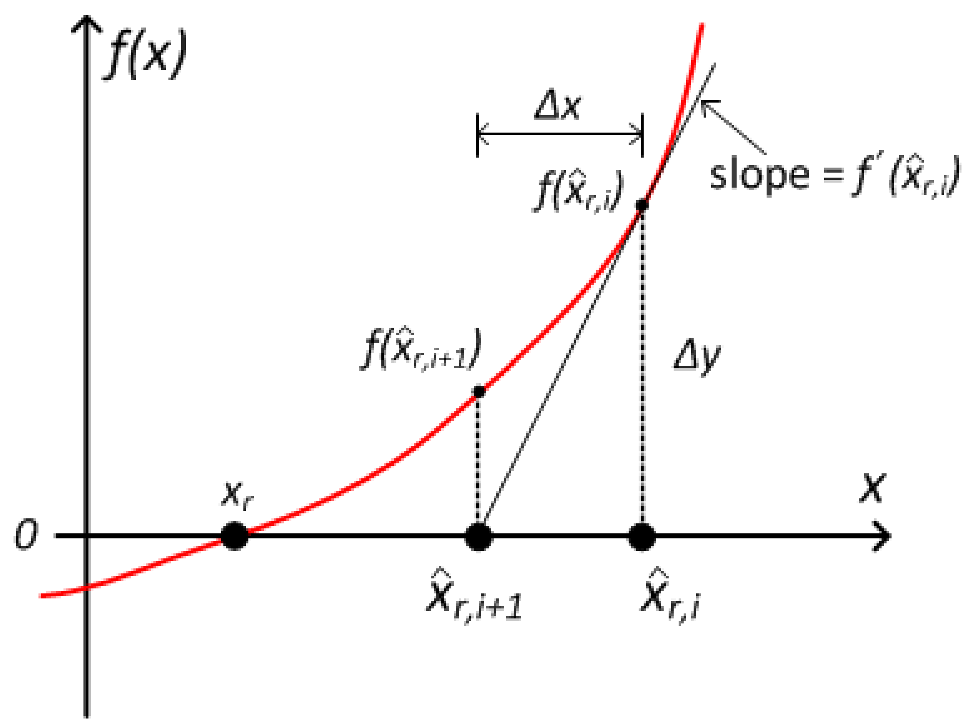
- New estimate is the root of a line tangent to $f(x)$ at $\hat{x}_{r,i}$
- Slope of $f(x)$ at $\hat{x}_{r,i}$ is the derivative at $\hat{x}_{r,i}$:

$$f'(\hat{x}_{r,i}) = \frac{\Delta y}{\Delta x} = \frac{f(\hat{x}_{r,i})}{\hat{x}_{r,i} - \hat{x}_{r,i+1}}$$

- Solving for the new root estimate:

$$\boxed{\hat{x}_{r,i+1} = \hat{x}_{r,i} - \frac{f(\hat{x}_{r,i})}{f'(\hat{x}_{r,i})}}$$

- An iterative formula for $\hat{x}_r$



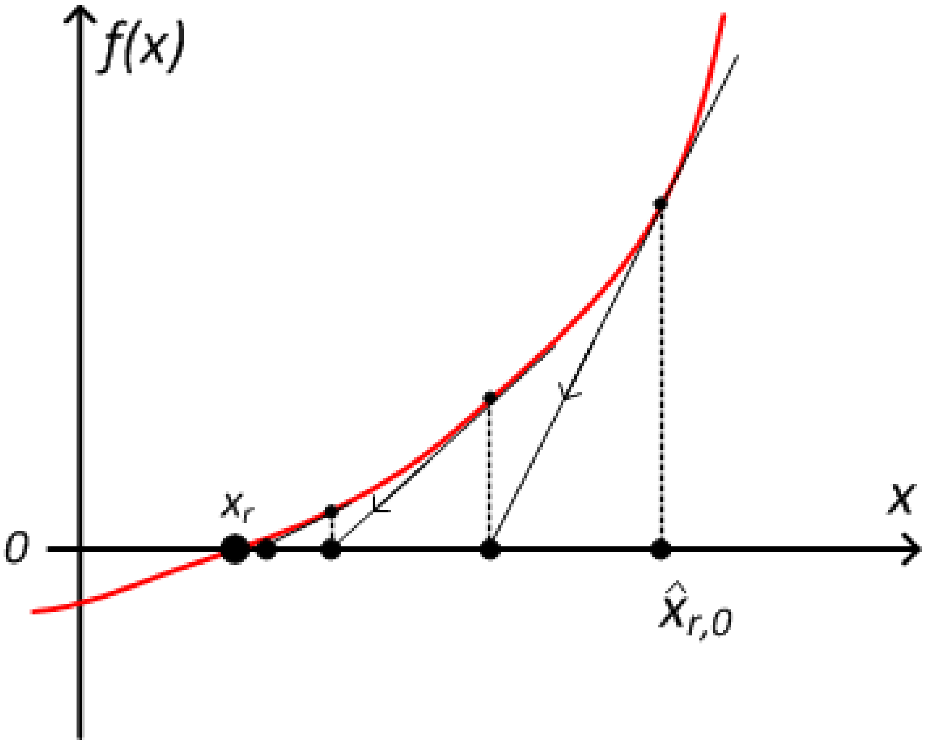
Newton-Raphson Method

- Iterate, using the ***Newton-Raphson formula***:

$$\hat{x}_{r,i+1} = \hat{x}_{r,i} - \frac{f\left(\hat{x}_{r,i}\right)}{f'\left(\hat{x}_{r,i}\right)}$$

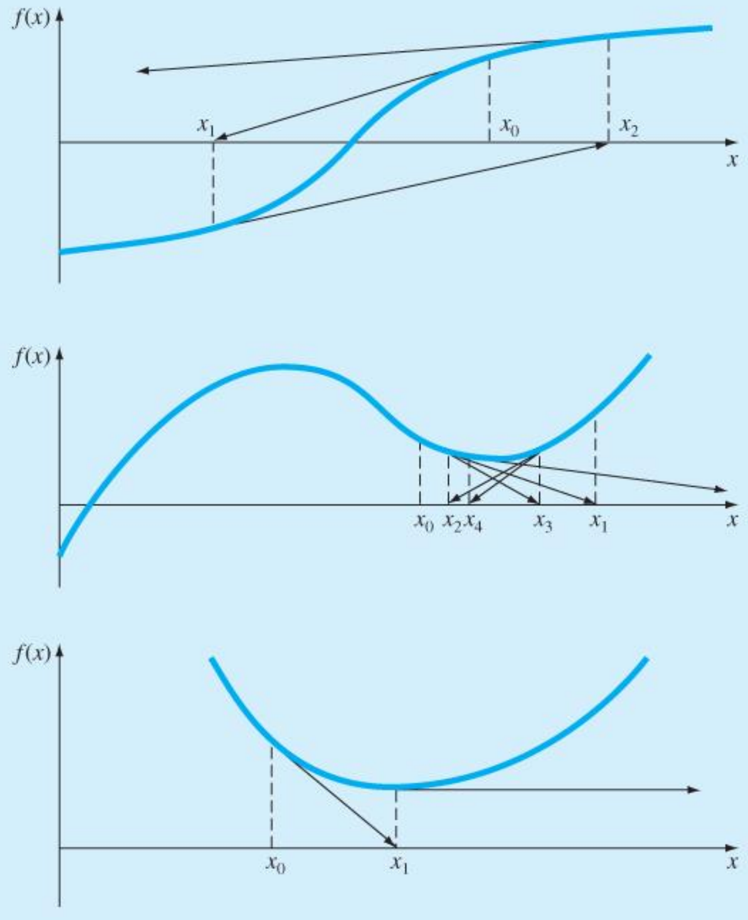
- Iterate until approximate error falls below a specified ***stopping criterion***

$$|\varepsilon_a| \leq \varepsilon_s$$



Newton-Raphson – Convergence

- Often fast, but convergence is not guaranteed
- ***Inflection point*** (constant slope) near a root causes divergence
- Areas of ***near-zero slope*** are problematic
  - Oscillation around local maximum/minimum
  - Tangent line sends estimate very far away – or to infinity for zero slope



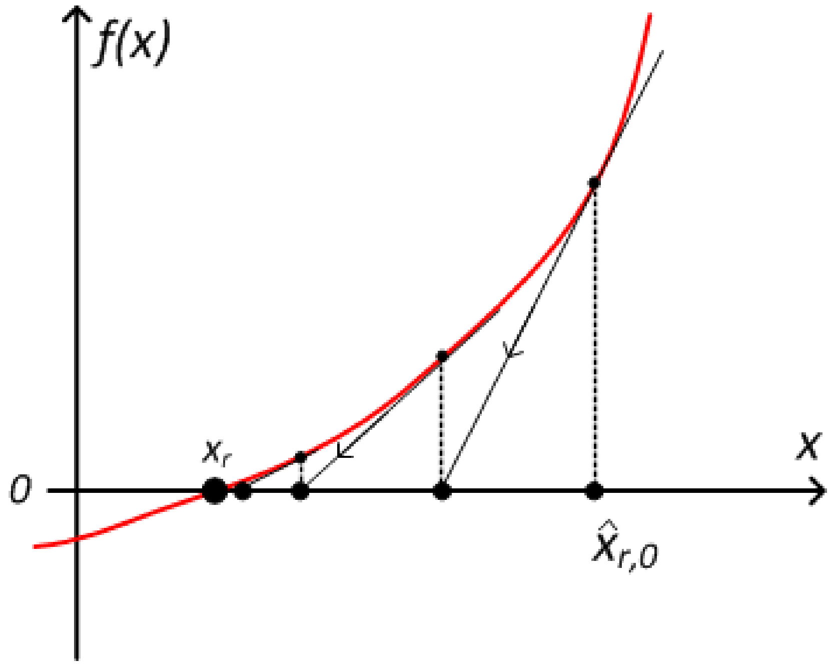
Chapra

Newton-Raphson – Rate of Convergence

- Current error is ***proportional to the square*** of the previous error

$$E_{t,i+1} = -\frac{f''(x_r)}{2f'(x_r)} E_{t,i}^2$$

- ***Quadratic convergence***
- Number of significant figures of accuracy approximately doubles each iteration



Newton-Raphson – Derivative Function

- Newton-Raphson algorithm requires two functions

$$\hat{x}_{r,i+1} = \hat{x}_{r,i} - \frac{f(\hat{x}_{r,i})}{f'(\hat{x}_{r,i})}$$

  - Function whose roots are to be found, $f(x)$
  - Derivative function, $f'(x)$
- That means $f'(x)$ must be found ***analytically***
  - Inconvenient – may be tedious for some functions
- Already performing numerical approximations
  - Why not calculate $f'(x)$ numerically? → ***Secant methods***

Secant Methods

- Same iterative formula as Newton-Raphson:

$$\hat{x}_{r,i+1} = \hat{x}_{r,i} - \frac{f(\hat{x}_{r,i})}{f'(\hat{x}_{r,i})}$$

- Now, approximate $f'(x)$ using a ***finite difference***

$$f'(x) \cong \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

- ***Secant method*** iterative formula:

$$\hat{x}_{r,i+1} = \hat{x}_{r,i} - \frac{f(\hat{x}_{r,i})(x_{i+1} - x_i)}{f(x_{i+1}) - f(x_i)}$$

- Would require ***two initial values***

- Instead, generate the second x value as a fractional perturbation of the first (the current estimate)

$$x_{i+1} = x_i + \delta x_i = \hat{x}_{r,i} + \delta \hat{x}_{r,i}$$

where δ is a very small number

- Finite difference approx. of $f'(x)$:

$$f'(x) \cong \frac{f(\hat{x}_{r,i} + \delta \hat{x}_{r,i}) - f(\hat{x}_{r,i})}{\delta \hat{x}_{r,i}}$$

- The ***modified secant*** iterative formula:

$$\boxed{\hat{x}_{r,i+1} = \hat{x}_{r,i} - \frac{\delta \hat{x}_{r,i} \cdot f(\hat{x}_{r,i})}{f(\hat{x}_{r,i} + \delta \hat{x}_{r,i}) - f(\hat{x}_{r,i})}}$$

Inverse Quadratic Interpolation

Root-Finding Methods – Interpolation

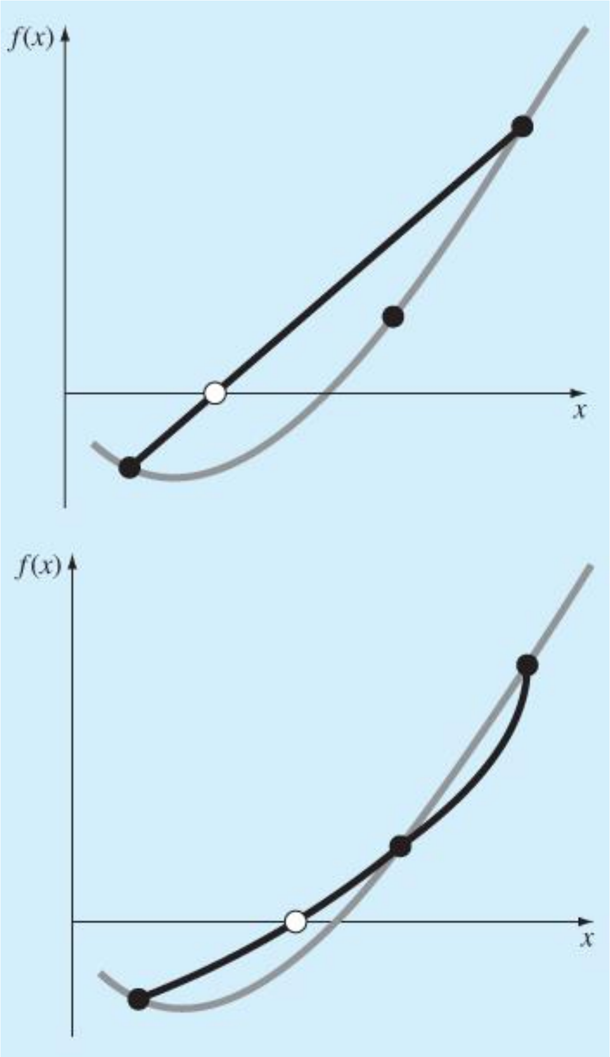
- False position and the Newton-Raphson/secant methods all use ***linear interpolation***
  - Non-linear function ***approximated as a linear function***
  - Root of the linear approximation becomes the approximation of the root
- We'll get to curve-fitting and interpolation later, but we should already suspect that a ***higher-order approximation*** for a non-linear function may be more accurate than a linear (first-order) approximation
- Increase accuracy of the root estimate by approximating our non-linear function as a ***quadratic***

Inverse Quadratic Interpolation

- Instead of using two points to approximate $f(x)$ as a line, use ***three points*** to approximate it as a ***parabola***
- Root estimate is where the parabola crosses the x-axis
- But, not all parabolas cross the x-axis – ***complex roots***
- All parabolas do cross the y-axis
  - To guarantee an x-axis crossing, ***turn the parabola on its side***

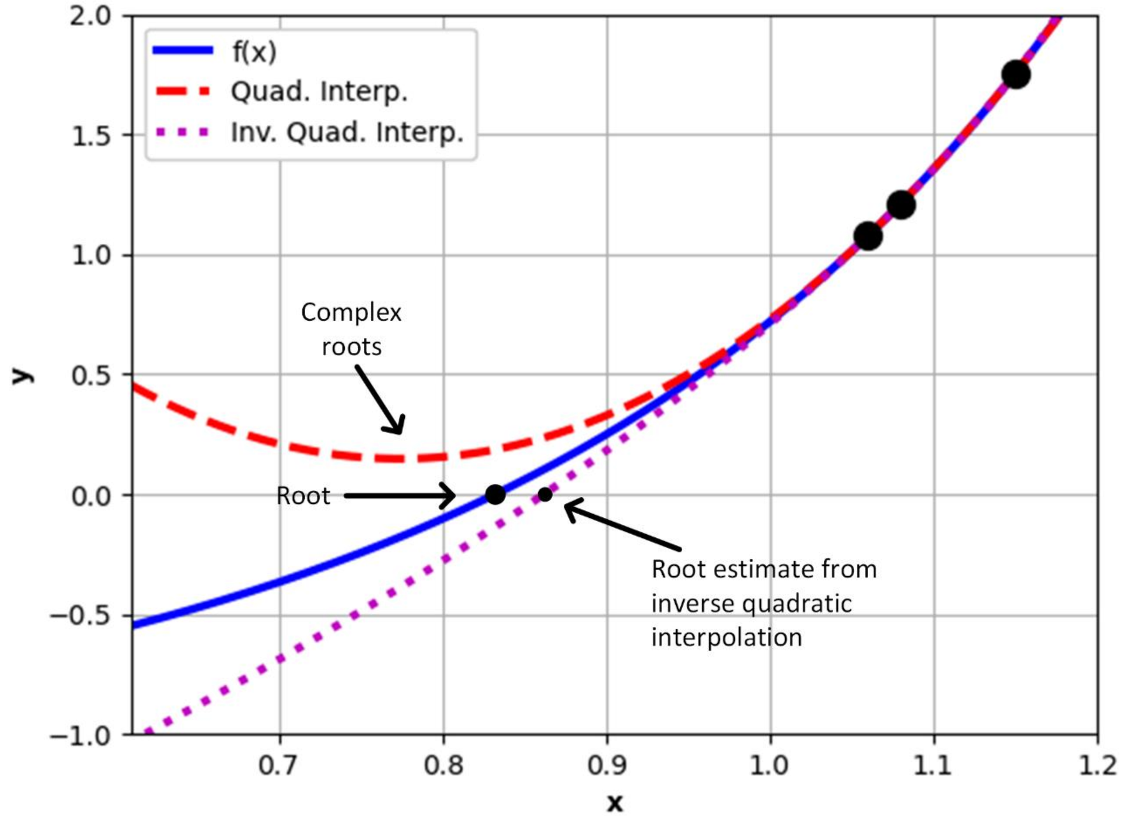
$$x = g(y)$$

  - An ***inverse quadratic*** function



Inverse Quadratic Interpolation – Example

- ***Three points*** required for quadratic approx.
  - How are they chosen?
- Inverse quadratic function ***will cross the x-axis***
  - For same three points a quadratic may not
- May be very efficient
  - ***May not converge***



Inverse Quadratic Interpolation

- Three known x and corresponding $f(x)$ values:
  - x_1, x_2, x_3, and $f(x_1), f(x_2), f(x_3)$
- Fit an inverse parabola to these three points
  - ***Lagrange polynomial*** – more on these later

$$x = g(y) = \frac{(y - y_2)(y - y_3)}{(y_1 - y_2)(y_1 - y_3)} x_1 + \frac{(y - y_1)(y - y_3)}{(y_2 - y_1)(y_1 - y_3)} x_2 + \frac{(y - y_1)(y - y_2)}{(y_3 - y_1)(y_3 - y_2)} x_3$$

- Don't actually need to calculate this parabola
- Only need its root – evaluate at $y = 0$ for new root estimate:

$$\hat{x}_{r,i+1} = \frac{y_2 y_3}{(y_1 - y_2)(y_1 - y_3)} x_1 + \frac{y_1 y_3}{(y_2 - y_1)(y_1 - y_3)} x_2 + \frac{y_1 y_2}{(y_3 - y_1)(y_3 - y_2)} x_3$$

Inverse Quadratic Interpolation

- Determining $\hat{x}_{r,i+1}$ from the three points is only part of the algorithm
  - Algorithm initialized with one or two x values
    - Need to determine the other one or two initial x values
  - Must update x_1, x_2, and x_3 on each iteration
- We won't get into these details here

- Will fail if any two $f(x_i)$ are equal
  - Revert to another open method (e.g. secant)
- May diverge
  - Revert to a bracketing method (e.g. bisection)

Brent's Method

Brent's Method – `brentq()`

- `brentq()` from SciPy's optimize package is based on ***Brent's method***
  - A bracketing method
  - Uses ***inverse quadratic interpolation*** to generate root estimates ***when possible***
  - In case of convergence issues reverts to ***bisection***
  - Always tries ***faster method first***, then uses ***bisection only if necessary***
- To use, first import the function:

```
from scipy.optimize import brentq
```

`scipy.optimize.brentq()`


```
root = brentq(func, a, b)
```

- `func`: function whose root you are looking for
- a: lower bracketing value
- b: upper bracketing value
- `root`: approximate root value returned

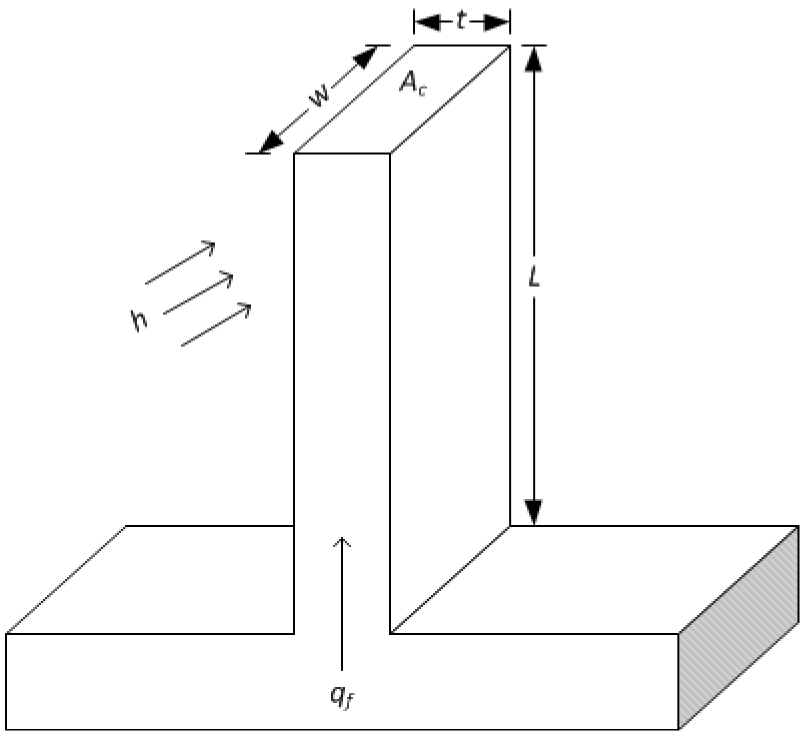
- Alternatively, we can control the output type:

```
r = brentq(func, a, b, full_output=True)
```

- r: `(root, robj)` – a tuple
  - `root`: approximate root value returned
  - `robj`: a `RootResults` object including convergence information

Example – brentq()

- Returning to our ***heat sink fin design problem***
- Want to know the length of the fin required for a heat transfer rate of $q_f = 500mW$, given the other specified parameters:
  - **Width**: *w* = 1 cm
  - **Thickness**: *t* = 2 mm
  - **Heat transfer coeff.**:
    h = 100 W/(m^{2}K)
  - **Aluminum**: k = 210 W/(m·K)
  - **Ambient temperature**:
    $T_\infty = 40°C$
  - **Base temperature**:
    $T_b = 100°C$



Example – brentq()

- We'll now use `brentq()` to find the root of $f(L)$

$$f(L) = M \cdot \frac{\sinh(mL) + \left(\frac{h}{mk}\right)\cosh(mL)}{\cosh(mL) + \left(\frac{h}{mk}\right)\sinh(mL)} - 500mW = 0$$

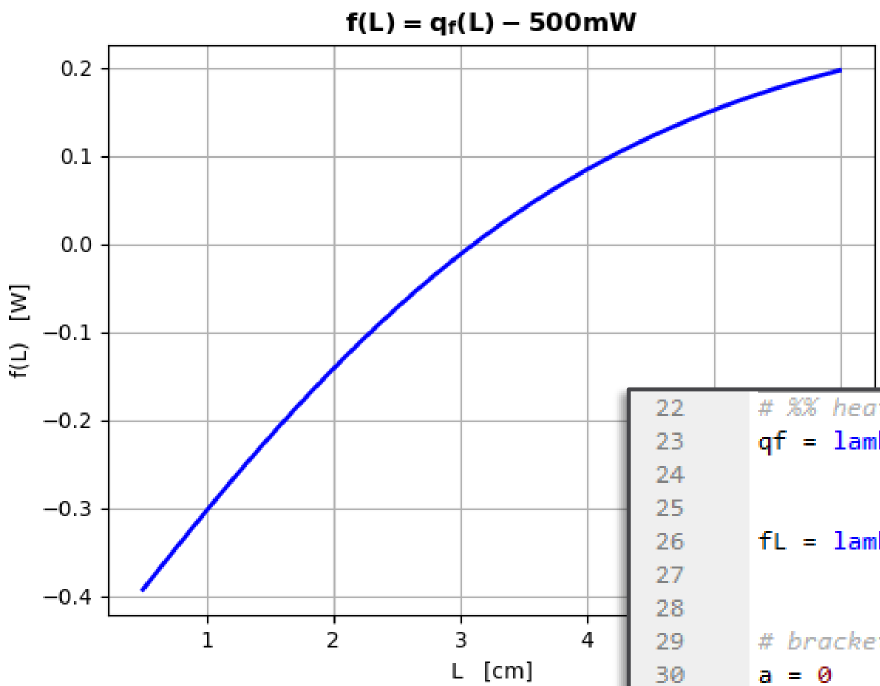
where

$$m = \sqrt{\frac{hP}{kA_c}}, \quad M = \sqrt{hPkA_c} \cdot \theta_b$$

$$A_c = w \cdot t, \quad P = 2w + 2t$$

$$\theta_b = T_b - T_\infty$$

Example – brentq()



- Define the function whose root we want to find
- Define the bracket
  - Values must bracket a sign change

```python
# %% heat transfer equation to be solved
qf = lambda L: M*(np.sinh(m*L) + h/(m*k)*np.cosh(m*L))/(np.cosh(m*L)
                  + h/(m*k)*np.sinh(m*L))

fL = lambda L: M*(np.sinh(m*L) + h/(m*k)*np.cosh(m*L))/(np.cosh(m*L)
                  + h/(m*k)*np.sinh(m*L)) - 0.5

# bracketing values [m]
a = 0
b = 0.2
(root, r) = brentq(fL, a, b, full_output=True)
L500 = root
print(f'\nqf(L) = {qf(L500)}\n')
display(r)
```

- Pass the function object, bracketing values, and other arguments to brentq()

Example – brentq()

```
In [86]: runfile('C:/Users/webbky/Box/KWebb/Cl
finDesign.py', wdir='C:/Users/webbky/Box/KWebb

qf(L) = 0.49999999999995176

      converged: True
           flag: 'converged'
 function_calls: 10
     iterations: 9
           root: 0.031091553634813824
```

- Convergence achieved in nine iterations
- Root is at 0.031 m
  - A 3.1 cm fin

Roots of Polynomials

Roots of Polynomials

- Polynomials are linear (first order) or nonlinear (second and higher order) functions of the form

$$f(x) = a_1 x^n + a_2 x^{n-1} + \cdots + a_n x + a_{n+1}$$

- An ***n^{th}-order polynomial has n roots***
  - Often, we'd like to find all n roots at once
  - Methods described thus far find only one root at a time
- For 2^{nd}-order,the ***quadratic formula*** yields both roots at once:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Roots of Polynomials – `np.roots()`

- To find all n roots of a polynomial:

```
x = np.roots(c)
```

  - c: (n+1)-vector of polynomial coefficients, i.e., the a_i's from the previous slide:

$$f(x) = c[0]x^n + c[1]x^{n-1} + \cdots + c[n-1]x + c[n]$$

  - x: n-vector of roots

- `np.roots()` works by treating the root-finding problem as an ***eigenvalue problem***

Roots of Polynomials – `np.poly()`

- Polynomials are an important class of functions
  - Curve-fitting and interpolation
  - Linear system theory and controls
- Often, we may want to generate the n^{th}-order polynomial corresponding to a given set of n roots

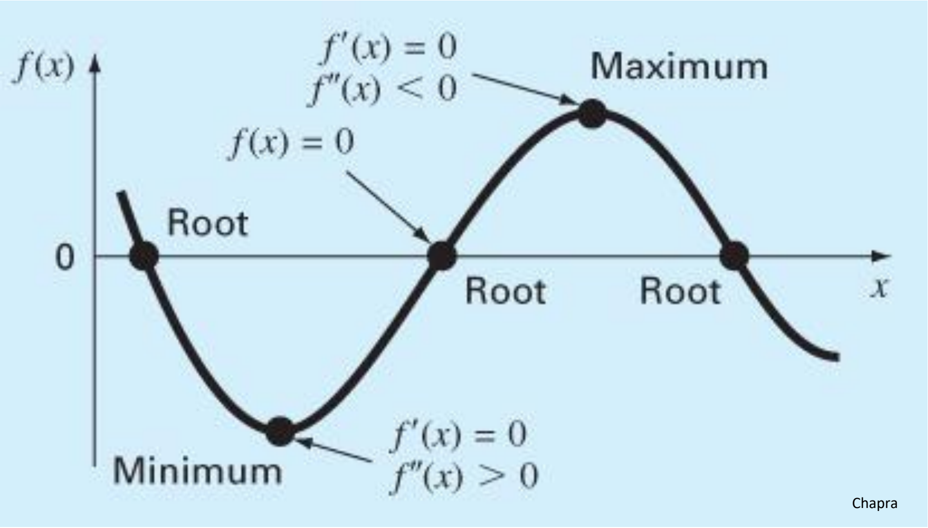
```
c = np.poly(x)
```

  - x: n-vector of roots
  - c: (n+1)-vector of polynomial coefficients

Optimization

Optimization

- Optimization is very important to engineers
  - Adjusting parameters to maximize some measure of performance of a system
- Process of finding ***maxima*** and ***minima*** (optima) of functions



Maxima and Minima

- An optimum point of a function occurs where the first derivative (***slope***) of the function is zero

$$f'(x) = 0$$

- An optimum point is a ***maximum*** if the second derivative (***curvature***) of the function is ***negative***

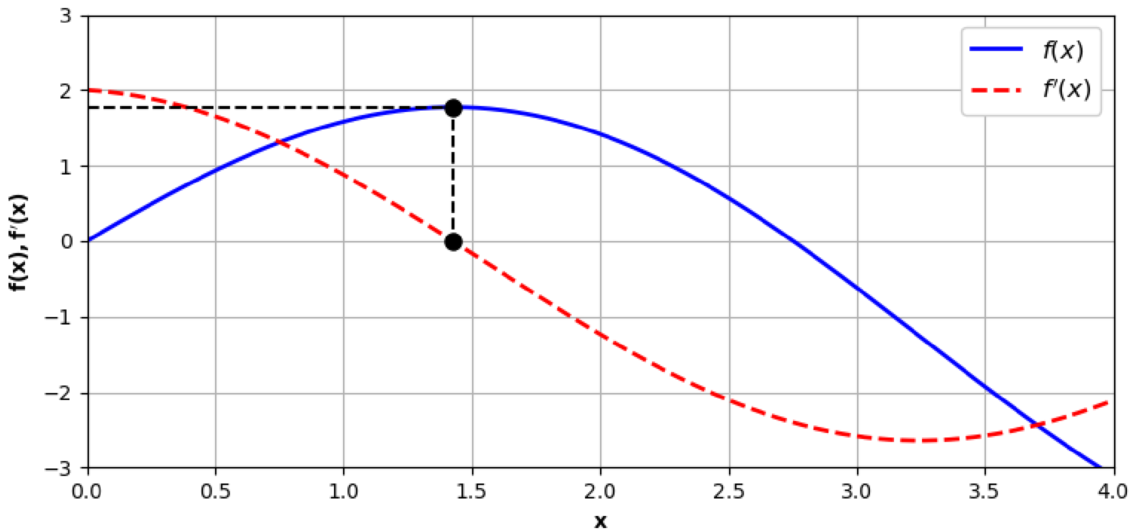
$$f''(x) < 0$$

- An optimum point is a ***minimum*** if the second derivative (***curvature***) of the function is ***positive***

$$f''(x) > 0$$

Optimization as a Root-Finding Problem

- Optima occur where $f'(x) = 0$
  - Could find optima of $f(x)$ by finding roots of $f'(x)$



- Requires calculation of the derivative, either analytically or numerically
- Direct (non-derivative) methods are often faster and more reliable

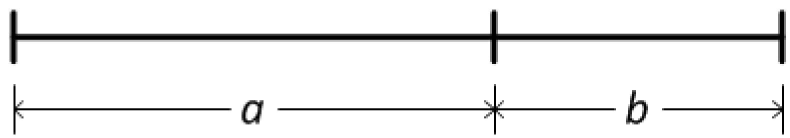
Optimization

- Optimization methods exist for ***one-dimensional*** and ***multi-dimensional*** functions
- As with root-finding, both ***bracketing*** and ***open methods*** exist
- Here, we'll look at:
  - **One dimensional optimization**
    - Golden-section search
    - Parabolic interpolation
    - Use of `scipy.optimize.minimize_scalar()`
  - **Multi-dimensional optimization**
    - Use of `scipy.optimize.minimize()`

Golden-Section Search

The Golden Ratio – ϕ

- Divide a value into two parts, a and b,



such that the ratio of the larger part to the smaller part is equal to the ratio of the whole to the larger part

$$\frac{a}{b} = \frac{a+b}{a}$$

- The ratio a/b is the ***golden ratio***

$$\phi = \frac{1+\sqrt{5}}{2} = 1.618033988\ldots$$

The Golden Ratio – ϕ

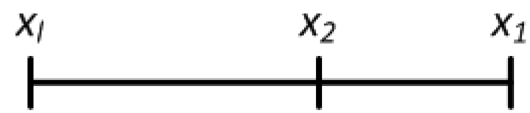
- Given an interval $[x_l, x_u]$, subdivide it from both ends according to the golden ratio

$$\frac{x_1 - x_l}{x_u - x_1} = \frac{x_u - x_l}{x_1 - x_l} = \phi$$

and

$$\frac{x_u - x_2}{x_2 - x_l} = \frac{x_u - x_l}{x_u - x_2} = \phi$$

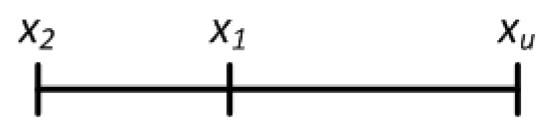
- If we discard the upper portion of the interval



we're left with a smaller interval, itself divided according to ϕ

$$\frac{x_2 - x_l}{x_1 - x_2} = \frac{x_1 - x_l}{x_2 - x_l} = \phi$$

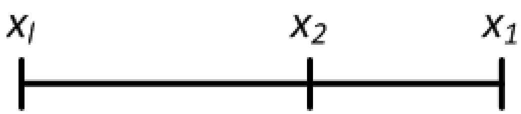
- The same is true if we discard the lower subinterval



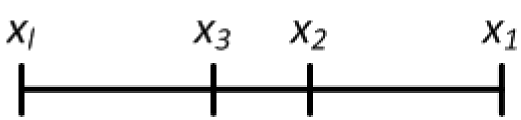
$$\frac{x_u - x_1}{x_1 - x_2} = \frac{x_u - x_2}{x_u - x_1} = \phi$$

The Golden Ratio – ϕ

- Starting from one of the subintervals (the lower one, here)



we can further subdivide it according to the golden ratio, starting from the upper bound on the interval

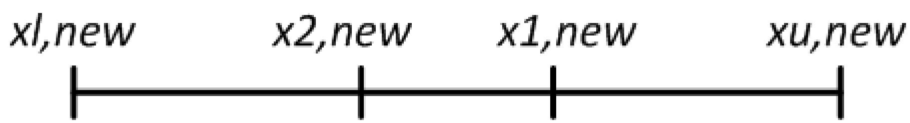


$$\frac{x_1 - x_3}{x_3 - x_l} = \frac{x_1 - x_l}{x_1 - x_3} = \phi$$

- If we reassign the variable names

$$x_l \rightarrow x_{l,new}$$
$$x_1 \rightarrow x_{u,new}$$
$$x_2 \rightarrow x_{1,new}$$
$$x_3 \rightarrow x_{2,new}$$

we‘re back where we started



- But now, the overall ***interval size has been reduced by a factor of ϕ***
- This process is the basis for the ***golden-section search algorithm***

Golden-Section Search

- A ***bracketing*** optimization method
  - Two initial values must bracket an optimum point
- Looks for a ***minimum***
  - To find a maximum use $-f(x)$
- Only one minimum point (local or global) in the bracketing interval
  - ***Unimodal***
- Very ***similar to bisection***
  - Now looking for a minimum, instead of a zero-crossing
  - ***Need two intermediate points***

Golden-Section Search

- Start with two initial values, x_l and x_u, that bracket a minimum point of the function, $f(x)$

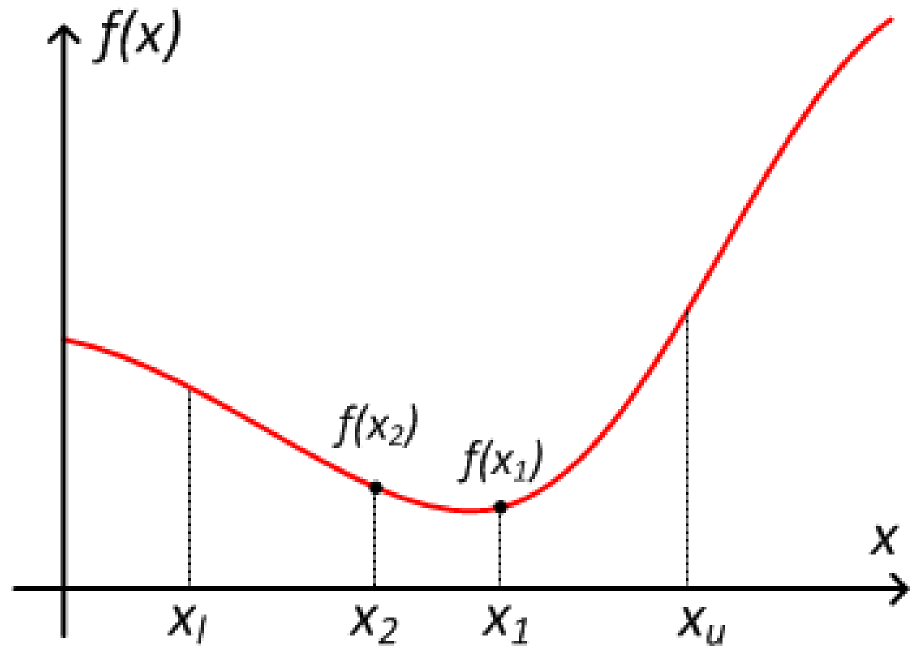
- Subdivide the interval according to the golden ratio with two intermediate points x_1 and x_2

$$x_1 = x_l + \frac{x_u - x_l}{\phi}$$

$$x_2 = x_u - \frac{x_u - x_l}{\phi}$$

- Evaluate the function at each of the intermediate points

$$f(x_1) \text{ and } f(x_2)$$



- Compare values of $f(x_1)$ and $f(x_2)$
- Two possibilities
  - $f(x_1) > f(x_2)$ or
  - $f(x_1) < f(x_2)$

Golden-Section Search – $f(x_1) < f(x_2)$

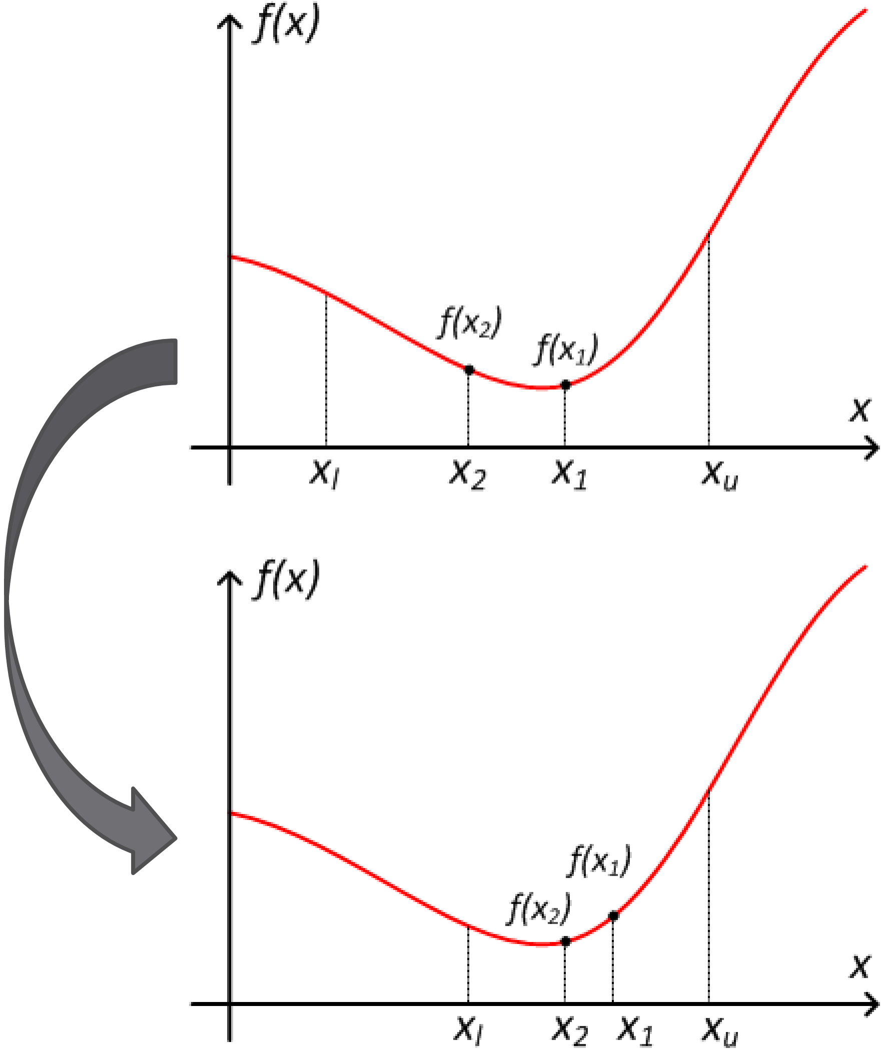
If $f(x_1) < f(x_2)$

- x_1 is the current estimate for the minimum point of $f(x)$, $\hat{x}_{opt}$
- True minimum cannot lie in the range of $[x_l, x_2]$
- Discard the lower subinterval
- Reassign variable names

$$x_2 \rightarrow x_l$$
$$x_1 \rightarrow x_2$$
$$x_u \rightarrow x_u$$

- Using new x_l, x_u, and x_2 values, calculate a new x_1

$$x_1 = x_l + \frac{x_u - x_l}{\phi}$$



Golden-Section Search – $f(x_1) > f(x_2)$

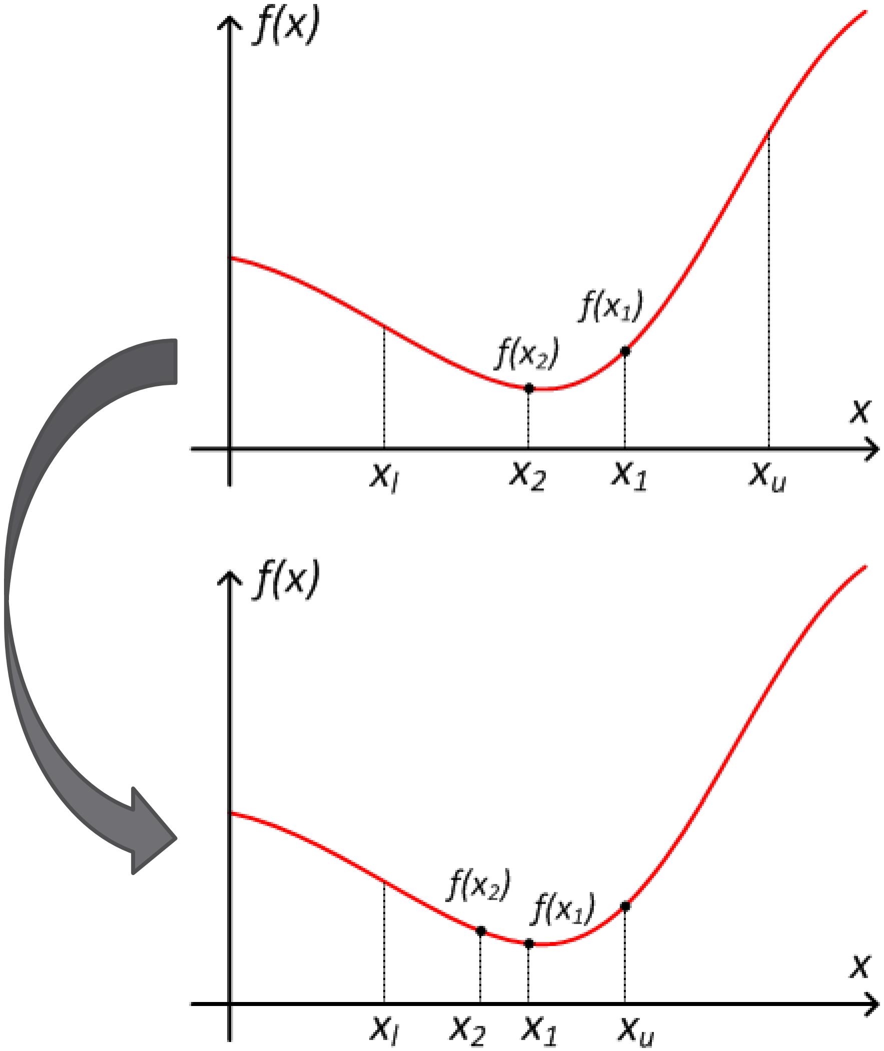
If $f(x_1) > f(x_2)$

- x_2 is the current estimate for the minimum point of $f(x)$, $\hat{x}_{opt}$
- True minimum cannot lie in the range of $[x_1, x_u]$
- Discard the upper subinterval
- Reassign variable names

$$x_l \rightarrow x_l$$
$$x_2 \rightarrow x_1$$
$$x_1 \rightarrow x_u$$

- Using new x_l, x_u, and x_1 values, calculate a new x_2

$$x_2 = x_u - \frac{x_u - x_l}{\phi}$$



Golden-Section Search

- Continue iterating and updating the $\hat{x}_{opt}$, the estimate of the minimizing value for $f(x)$
  - ***Only one new point needs to be calculated at each iteration***
    - This is the beauty of using the golden ratio
    - Very efficient
- ***Size of the bracketing interval decreases by a factor of $\boldsymbol{\phi = 1.618}$ … with each iteration***
- Continue to iterate until error estimate satisfies a stopping criterion

Golden-Section Search – Error

- Consider the case where $x_{opt} = x_u$
- Lower subinterval, $[x_l, x_2]$, is discarded
- Optimum point estimate is x_1

$$\hat{x}_{opt} = x_1$$

- This scenario represent the ***worst-case error***

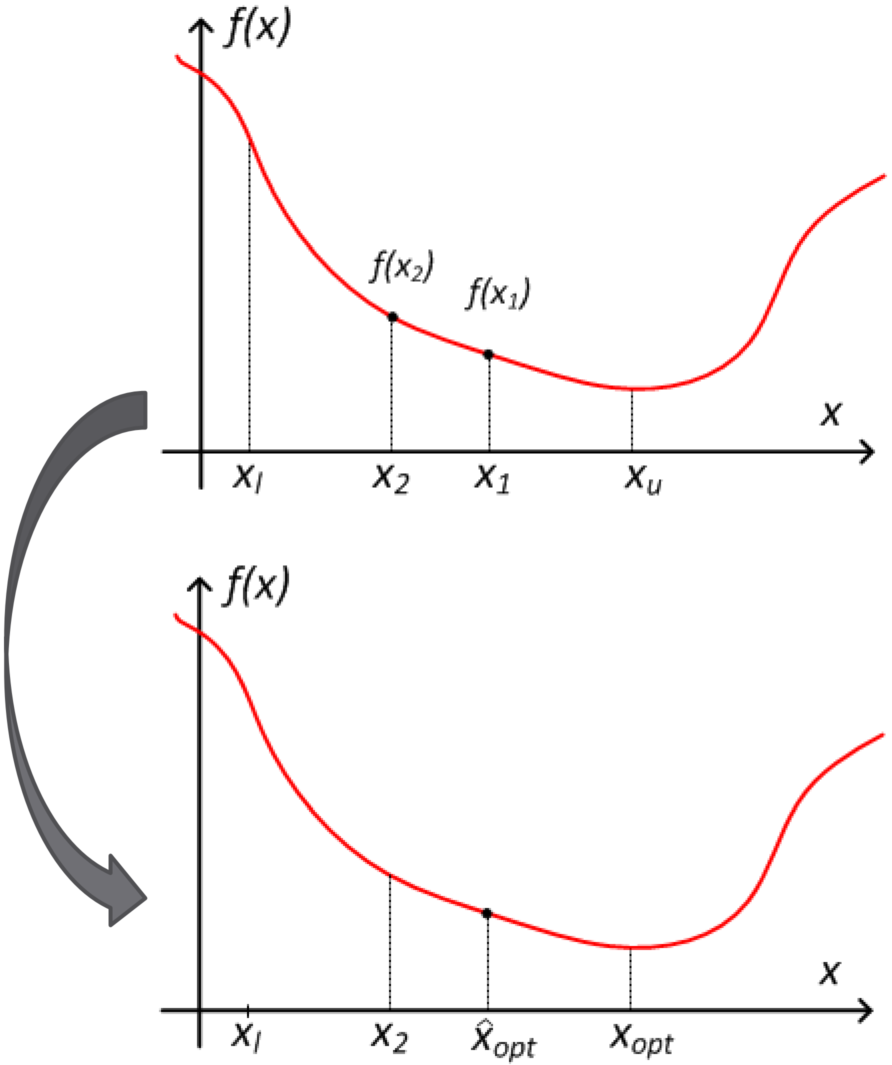
$$|E_{max}| = |\hat{x}_{opt} - x_{opt}| = |x_1 - x_u|$$

$$= \left|\left(x_l + \frac{x_u - x_l}{\phi}\right) - x_u\right|$$

$$= (x_u - x_l)\left(1 - \frac{1}{\phi}\right)$$

and

$$\frac{1}{\phi} = \phi - 1$$



Golden-Section Search – Error

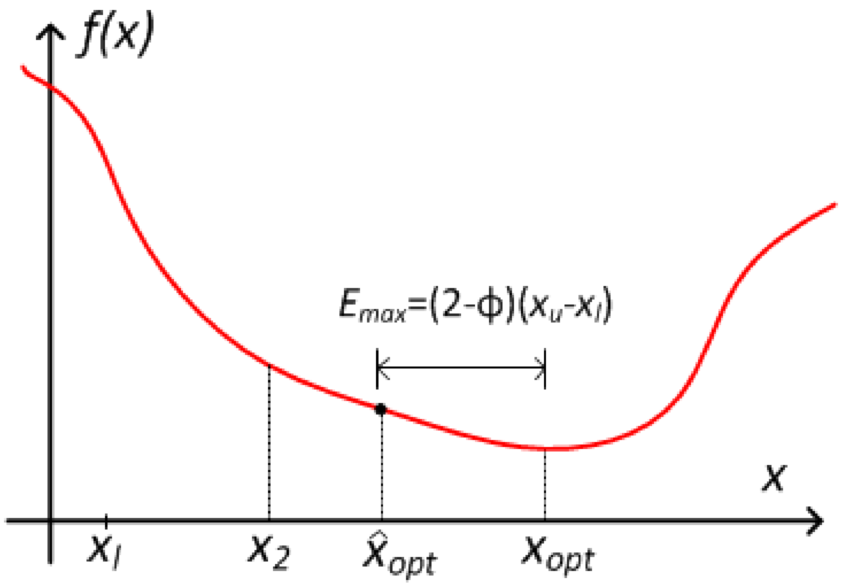
- The ***worst-case error*** is

$$|E_{max}| = (2-\phi)(x_u - x_l)$$

- Normalize to the current estimate
  - Convert from absolute to ***relative error***
- Use worst-case value as our ***approximate error***

$$\boxed{\varepsilon_a = (2-\phi)\left|\frac{x_u - x_l}{\hat{x}_{opt}}\right| \cdot 100\%}$$

- Calculate ε_a each iteration
  - Continue until stopping criterion is satisfied



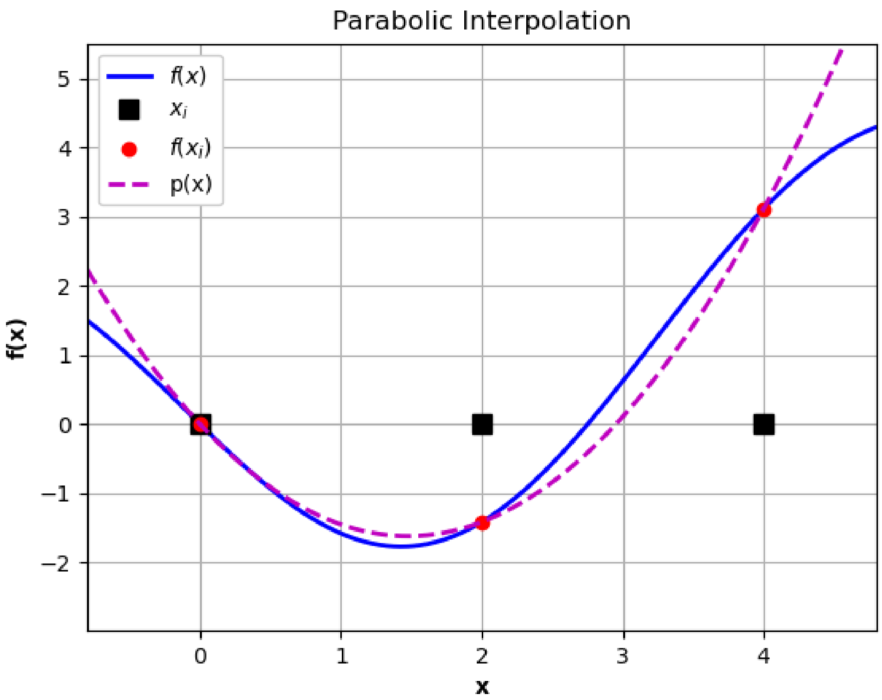
Parabolic Interpolation

Parabolic Interpolation

- Near an optimum point, many functions can be satisfactorily ***approximated with a quadratic***
- ***Three points*** define a unique parabola
  - Two points define the bracketing interval
  - A third intermediate point somewhere within the bracket
- Optimum point of the parabolic approximation becomes current estimate of the optimum point
- Evaluate $f(x)$ at $\hat{x}_{opt}$
- Retain the subinterval containing the optimum point, discard one of the bracketing points, and iterate
- $f(x)$ must be ***unimodal***
- Looking for a ***minimum***, but algorithm can easily be modified to look for a ***maximum***

Parabolic Interpolation

- Start with three points, which bracket the optimum
- Evaluate the $f(x)$ at these points
- Fit a parabola to the three points
  - Can use a Lagrange polynomial
  - Not necessary to actually calculate the parabola – can jump to finding its optimum point



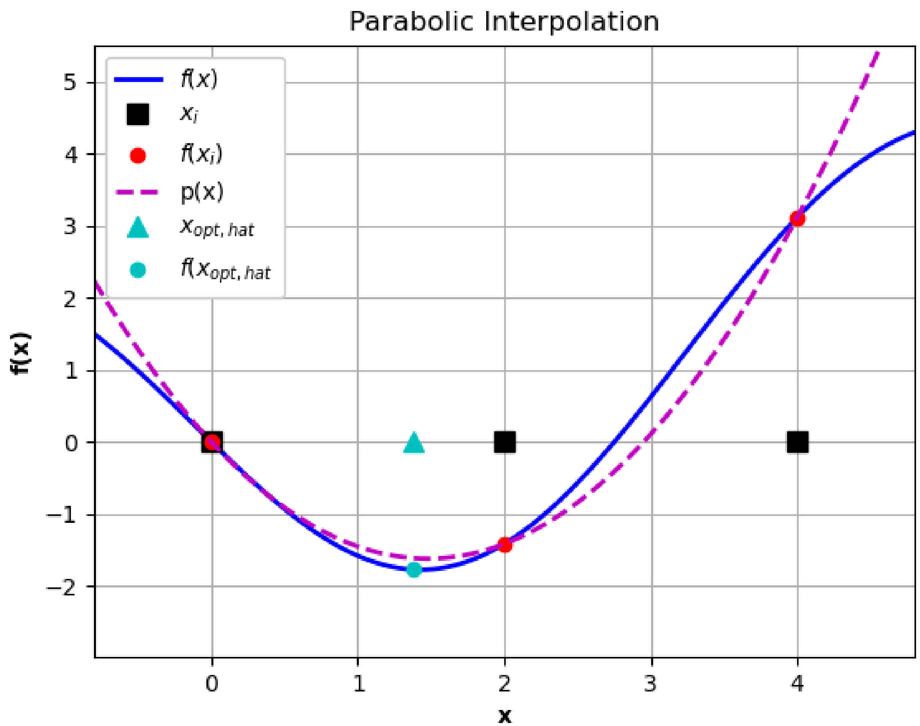
$$p(x) = \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} f(x_1) + \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} f(x_2) + \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)} f(x_3)$$

Parabolic Interpolation

- Calculate the ***optimum point of the parabolic approximation***

$$x_4 = x_2 - \frac{1}{2} \cdot \frac{(x_2 - x_1)^2[f(x_2) - f(x_3)] - (x_2 - x_3)^2[f(x_2) - f(x_1)]}{(x_2 - x_1)[f(x_2) - f(x_3)] - (x_2 - x_3)[f(x_2) - f(x_1)]}$$

- Expression for x_4 derived by solving $\frac{dp}{dx} = 0$
- x_4 becomes the current ***estimate for the optimum point***, $\hat{x}_{opt}$
- Evaluate $f(\hat{x}_{opt})$
  - Use values of $\hat{x}_{opt}$ and $f(\hat{x}_{opt})$ to appropriately ***reduce the bracketing interval***



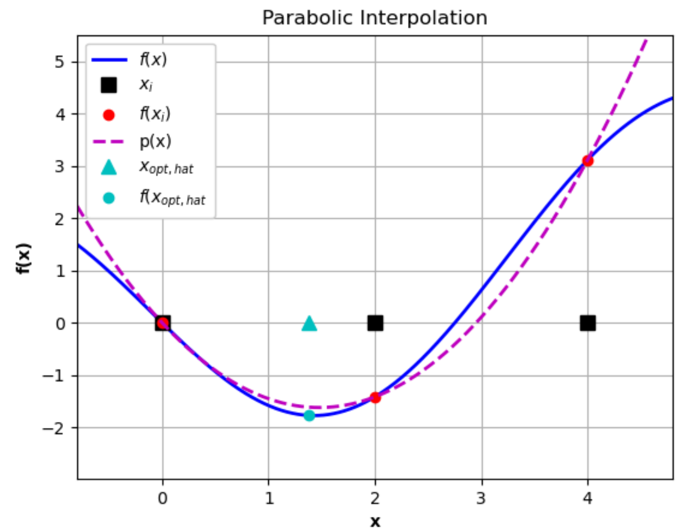
Parabolic Interpolation – Reducing the Bracket

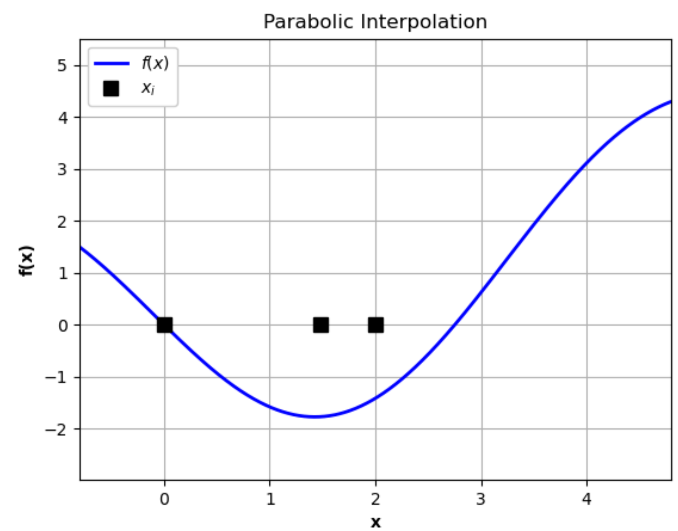
- If $x_4 < x_2$
  - If $f(x_4) < f(x_2)$ (shown here)
    - x_{opt} is in the lower subinterval
    - Discard the upper subinterval

$$x_{1,i+1} = x_{1,i}$$
$$x_{2,i+1} = x_{4,i}$$
$$x_{3,i+1} = x_{2,i}$$

  - If $f(x_4) > f(x_2)$
    - x_{opt} is in the upper subinterval
    - Discard the lower subinterval

$$x_{1,i+1} = x_{4,i}$$
$$x_{2,i+1} = x_{2,i}$$
$$x_{3,i+1} = x_{3,i}$$





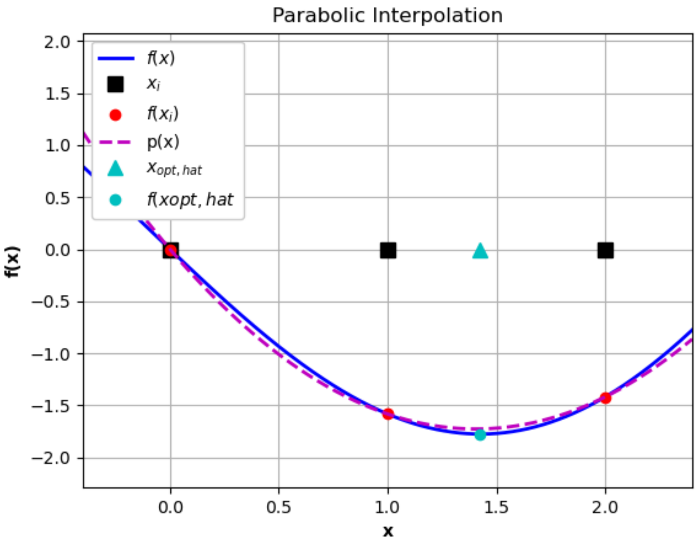
Parabolic Interpolation – Reducing the Bracket

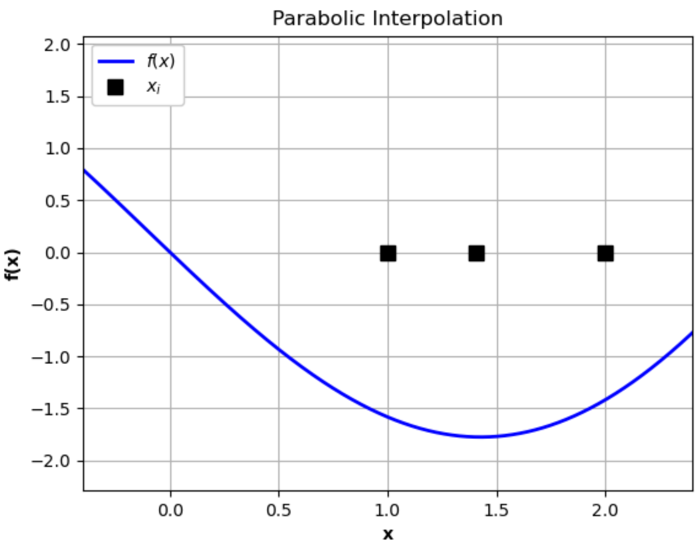
- If $x_4 > x_2$
  - If $f(x_4) < f(x_2)$ (shown here)
    - x_{opt} is in the upper subinterval
    - Discard the lower subinterval

$$x_{1,i+1} = x_{2,i}$$
$$x_{2,i+1} = x_{4,i}$$
$$x_{3,i+1} = x_{3,i}$$

  - If $f(x_4) > f(x_2)$
    - x_{opt} is in the lower subinterval
    - Discard the upper subinterval

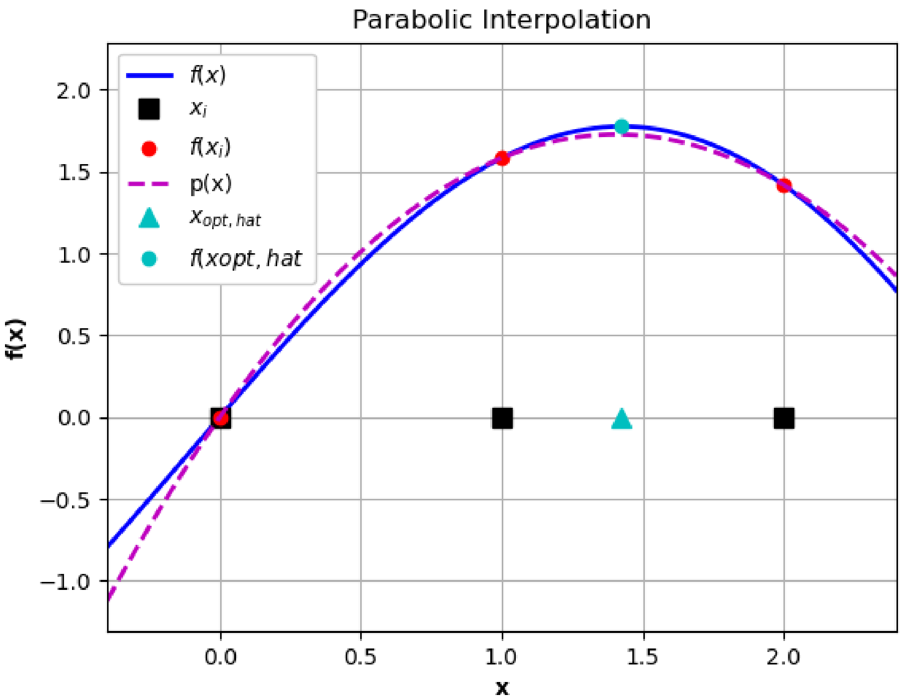
$$x_{1,i+1} = x_{1,i}$$
$$x_{2,i+1} = x_{2,i}$$
$$x_{3,i+1} = x_{4,i}$$





Parabolic Interpolation – Finding a Maximum

- Can also use parabolic interpolation to ***locate a maximum*** point
  - Parabola fit to the three points may open up or down
  - Need to ***adjust bracket reduction algorithm*** depending on whether a maximum or minimum point is sought



Optimization in Python

One-Dimensional Optimization – `minimize_scalar()`

- Parabolic interpolation is efficient, but may not converge
  - `minimize_scalar()` uses a ***parabolic interpolation*** when possible and ***golden-section search*** when necessary
- Finds the ***minimum*** of a function over an interval

```
opt = minimize_scalar(f, bracket=(x0, x1))
```

  - `f`: function to be optimized
  - `x0, x1`: bracketing values
  - `opt`: `optimizeResult` object returned – includes:
    - `opt.x:` the solution of the optimization (i.e., x_{opt})
    - `opt.fun:` value of objective function at the optimum (i.e., $f\left(x_{opt}\right)$)
    - `opt.nit:` number of iterations

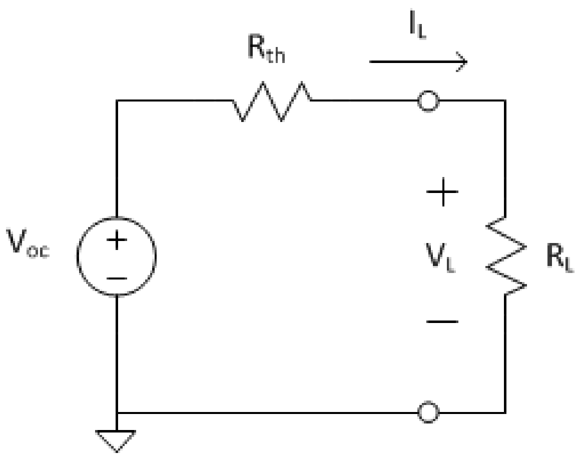
One-Dimensional Optimization – Example

- Determine the load resistance of an electrical circuit that ***maximizes power delivered to the load***
  - Normalize to source resistance and open-circuit voltage
    - $R_{th} = 1\Omega, V_{oc} = 1V$
  - Power delivered to the load is

$$P_L = I_L V_L$$

$$P_L = \frac{V_{oc}}{R_{th} + R_L} \cdot V_{oc} \frac{R_L}{R_{th} + R_L}$$

$$P_L = \frac{V_{oc}^2 R_L}{(R_{th} + R_L)^2}$$



  - Determine R_L to maximize P_L

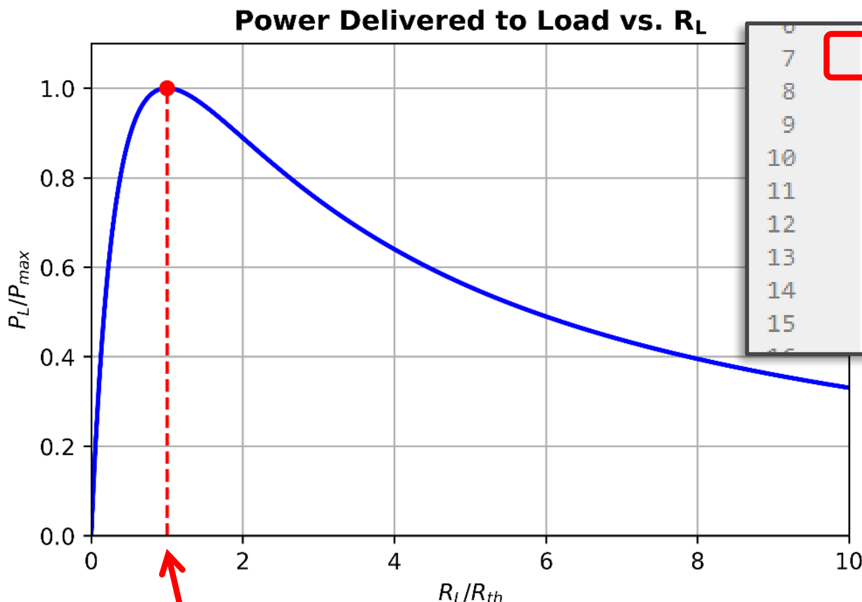
One-Dimensional Optimization – Example

- Negate function to find maximum

```
f = lambda RL: -(RL/(1+RL)**2)
R0 = 0
R1 = 100

opt_opts = {'maxiter': 1000, 'xtol': 1e-9, 'disp': True}
opt = minimize_scalar(f, bracket=(R0, R1), options=opt_opts)
RLmax = opt.x
Pmax = opt.fun
Pmax = -Pmax          # found min of -f(RL)
```

- Use options `dict` to set solver options



- Max Power occurs at

$$\boxed{\frac{R_L}{R_{th}} = 1 \rightarrow R_L = R_{th}}$$

```
Optimization terminated successfully;
The returned value satisfies the termination criteria
(using xtol = 1e-09 )

      fun: -0.25000000000000006
  message: '\nOptimization terminated successfully;\nThe returned value satisfies the termination criteria\n(using xtol = 1e-09 )'
     nfev: 28
      nit: 24
  success: True
        x: 0.999999997614594
```

Multi-Dimensional Optimization – `minimize()`

- Find the minimum of a function of two or more variables

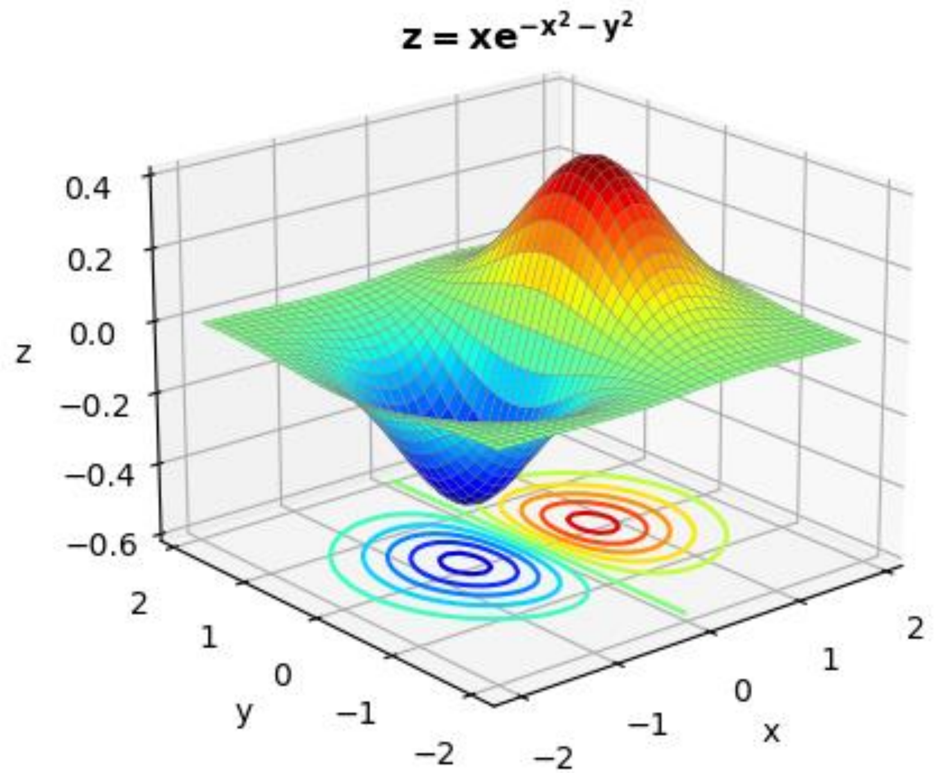
```
opt = minimize(f, x0)
```

  - `f`: function to be optimized
  - `x0`: array of initial values
  - `opt`: `optimizeResult` object returned – includes:
    - `opt.x`: the solution of the optimization (i.e., x_{opt})
    - `opt.fun`: value of objective function at the optimum (i.e., $f(x_{opt})$)
    - `opt.nit`: number of iterations

Multi-Dimensional Optimization – Example

- Find the minimum of a function of two variables

$$f(x, y) = x \cdot e^{-x^2-y^2}$$

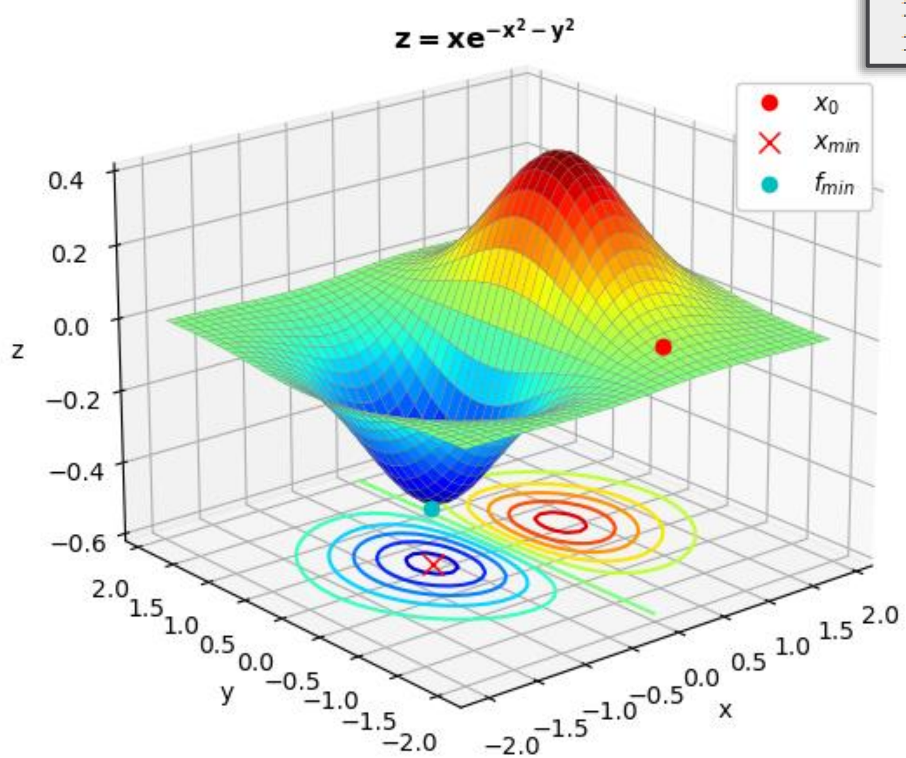


Multi-Dimensional Optimization – Example

- Use options dict to set solver options

```python
# function to be minimized
f = lambda x: x[0]*np.exp(-x[0]**2-x[1]**2)

# setup and run optimization
x0 = [0.5, -1.5]
opt_opts = {'disp': True, 'maxiter': 1000}
opt = minimize(f, x0, tol=1e-6, options=opt_opts)
xmin = opt.x[0]
ymin = opt.x[1]
zmin = opt.fun
```



- Set tolerance, if desired

```
Section2/twoDoptim.py', wdir='C:/Users/webbky/E
Python/Section2')
Optimization terminated successfully.
         Current function value: -0.428882
         Iterations: 30
         Function evaluations: 144
         Gradient evaluations: 48
```

- Convergence for this example depends on choice of x_0