Roots of Equations

by Loïc Quertenmont, PhD

LINFO1113 - 2019-2020

Programme

Cours	1	Librairies mathématiques, représentation des nombres en Python et erreurs liées
Cours	2,3	Résolution des systèmes linéaires
Cours	4,5	Interpolation et Régression Linéaires
Cours	**6,7**	**Racines d'équations**
Cours	8,9	Différentiation numérique
Cours	10	Intégration numérique
Cours	11,12	Introduction à l'optimisation
Cours	13	Rappel / Répétition

Outline

- **Introduction**
- **Incremental Search Method**
- **Method of Bisection**
- **Method based on linear interpolation**
- **Newton-Raphson Method**
- **Systems of equations**

Introduction

Introduction

- Goal:
  - Find x such that $f(x) = 0$
  - With f known

- Terminology:
  - Roots of the equation $f(x) = 0$
  - Zeros of f

Function

- Reminder for $y = f(x)$
  - x is the input
  - y is the output
  - f is the rule for computing y given x
    - f could be a set of mathematical operations
    - f could be an algorithm
- At the end of the day, in numerical analysis, f is always an algorithm going from x to y
- f is known if we can run the algorithm for any x

Roots

- The roots of an equation can be real or complex
  - In general complex roots are ignored as they have no physical importance.
  - An exception is the roots of polynomial equations

    $$a_0 + a_1 x + a_1 x^2 + \cdots + a_n x^n = 0$$

    where complex roots might be meaningful (vibration study). Not discussed here.
  - We will mostly focus on **real roots**
- An equation may have **many** roots or **none**
  - $\sin x - x = 0$ 1 root → x=0
  - $\tan x - x = 0$ ∞ roots → x=0, ±4.493, ±7.725, …

Algorithms

- All algorithms for finding roots are iterative procedures
- They require a starting point (an estimate of the root)
- **The root estimation is crucial**
  - a bad starting value may fail to converge, or it may converge to the "wrong" root (a root different from the one thought).
- There is no universal recipe for estimating the value
  - If the equation is associated with a physical problem, then the context of the problem (physical insight) might suggest the approximate location of the root
  - Systematic numerical search for the roots
  - Plotting the function and visually inspecting it
- **Bracketing the root before using root finding algorithms**

Incremental Search Method

Principle

- Useful tool for detecting and bracketing roots
- It can also be used for computing roots
  - but other methods are more efficient for that task

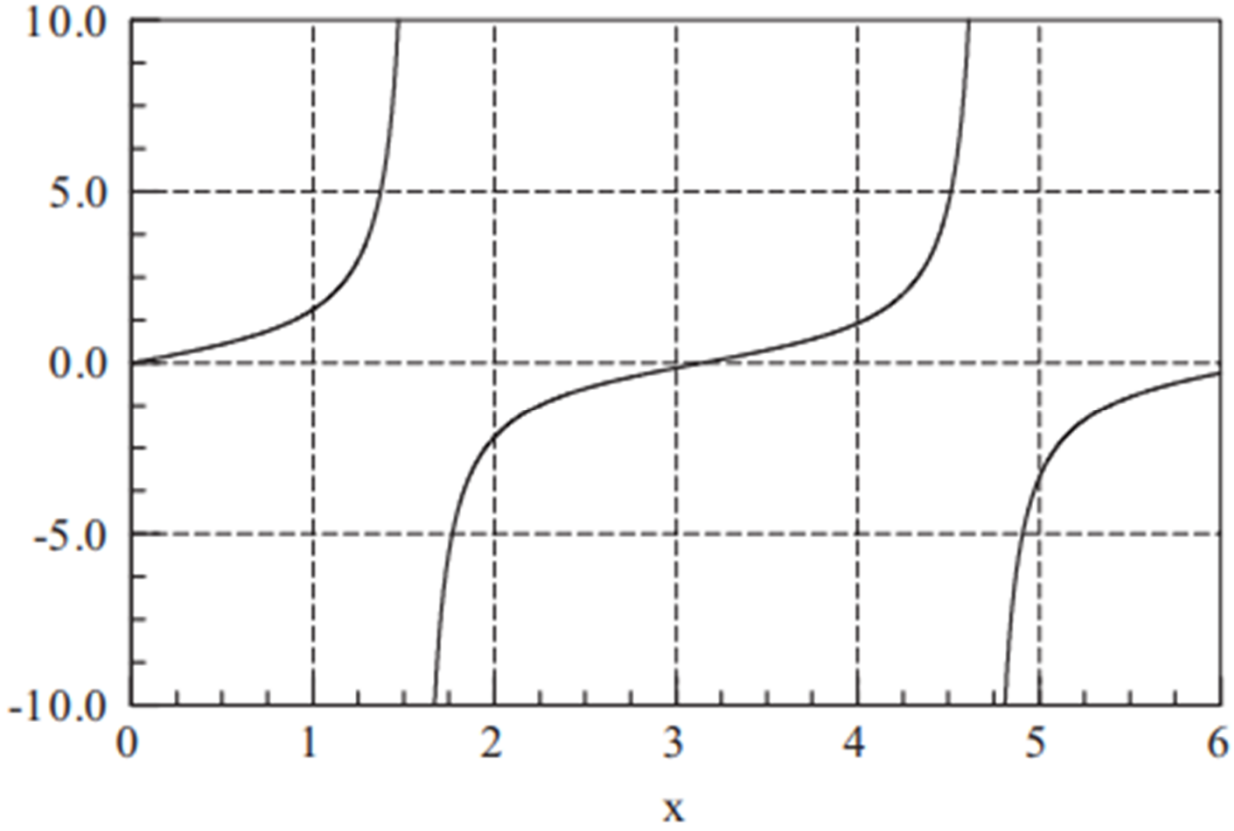
- Principle:
  - If $f(x_1)$ and $f(x_2)$ have opposite signs, there is a least one root in the interval $[x_1, x_2]$
  - If the interval is small enough, there is likely just a single root
  - Zeros of $f(x)$ can be detected by evaluating the function at interval Δx and looking for a change of sign

Problems (1)

- It is possible to miss two closely spaced roots if the search increment x is larger than the spacing of the roots.
- A double root (two roots that coincide) will not be detected.
- Certain singularities (poles) of f(x) can be mistaken for roots.

Problems (2)

- f(x) = tan x changes sign at $x = \pm\frac{1}{2}n\pi$, n = 1, 3, 5, ...
  However, these locations are not true zeroes, because the function does not cross the x-axis.



Algorithm

Can you implement the algorithm ?

```
def rootsearch(f,a,b,dx):
    x1 = a; f1 = f(a)
    x2 = a + dx; f2 = f(x2)
    while sign(f1) == sign(f2):
        if x1  >=  b: return None,None
        x1 = x2; f1 = f2
        x2 = x1 + dx; f2 = f(x2)
    else:
        return x1,x2
```

- Search in [a,b]
- Steps of dx
- Stops at the first change of sign detected
  - Return the bracketing
- If no root detected,
  - Return None, None

Example

- A root of $x^3 - 10x^2 + 5 = 0$ lies in [0, 1].
- Use the root search algorithm to compute this root with four-digit accuracy.
  - Root = (0.7346 + 0.7347)/2
- How many function evaluation will be performed at most ?
  - 10k iterations
- How can we reduce this number ?
  - Progressively increase the accuracy
  - 4 steps → 40 iterations at most

Method of Bisection

Principle

- A root is bracket in the interval (x_1, x_2)
- The bisection method is halving the interval at each iteration until it gets sufficiently small
  - Also known as “interval halving method
- We check if the root is on the left half or on the right half of the interval.

Principle

- $f(x_1)$ and $f(x_2)$ have opposite signs
- Evaluate $f(x_3)$, where $x_3 = \frac{1}{2}(x_1 + x_2)$
- If $f(x_1)$ and $f(x_3)$ have opposite signs
  - The root is in (x_1, x_3)
- Otherwise
  - The root is in (x_3, x_2)

- Repeat the algorithm until the interval is small enough: $|x_2 - x_1| \leq \varepsilon$

Number of iteration required

- The number of iteration required to achieved a given precision is easy to compute

- Bracket length (Δx) is reduced
  - to $\frac{\Delta x}{2}$ after 1 iteration
  - to $\frac{\Delta x}{4}$ after 2 iterations
  - to $\frac{\Delta x}{2^n}$ after n iterations
- $n = \frac{\ln(\Delta x / \varepsilon)}{\ln(2)}$ (ceiled to an integer value)

Additional checks

- For smooth and easy functions, we expect
  - $|f(x_3)| \leq |f(x_1)|$ and $|f(x_3)| \leq |f(x_2)|$

- If this is not the case, that would mean that something is wrong OR that we have bracket a pole and not a root.

Algorithm

Can you implement the algorithm ?

```
def bisection(f,x1,x2,switch=1,tol=1.0e-9):
    f1 = f(x1)
    if f1 == 0.0: return x1
    f2 = f(x2)
    if f2 == 0.0: return x2
    if sign(f1) == sign(f2):
        error.err('Root is not bracketed')
    n = int(math.ceil(math.log(abs(x2 - x1)/tol)/math.log(2.0)))

    for i in range(n):
        x3 = 0.5*(x1 + x2); f3 = f(x3)
        if (switch == 1) and (abs(f3) > abs(f1)) \
                         and (abs(f3) > abs(f2)):
            return None
        if f3 == 0.0: return x3
        if sign(f2)!= sign(f3): x1 = x3; f1 = f3
        else: x2 = x3; f2 = f3
    return (x1 + x2)/2.0
```

- Initial checks
- Compute n
- Halving algorithm
  - Check on f3 reduction or not (according to switch parameter)
- Return the center of the final bracketing interval

Example

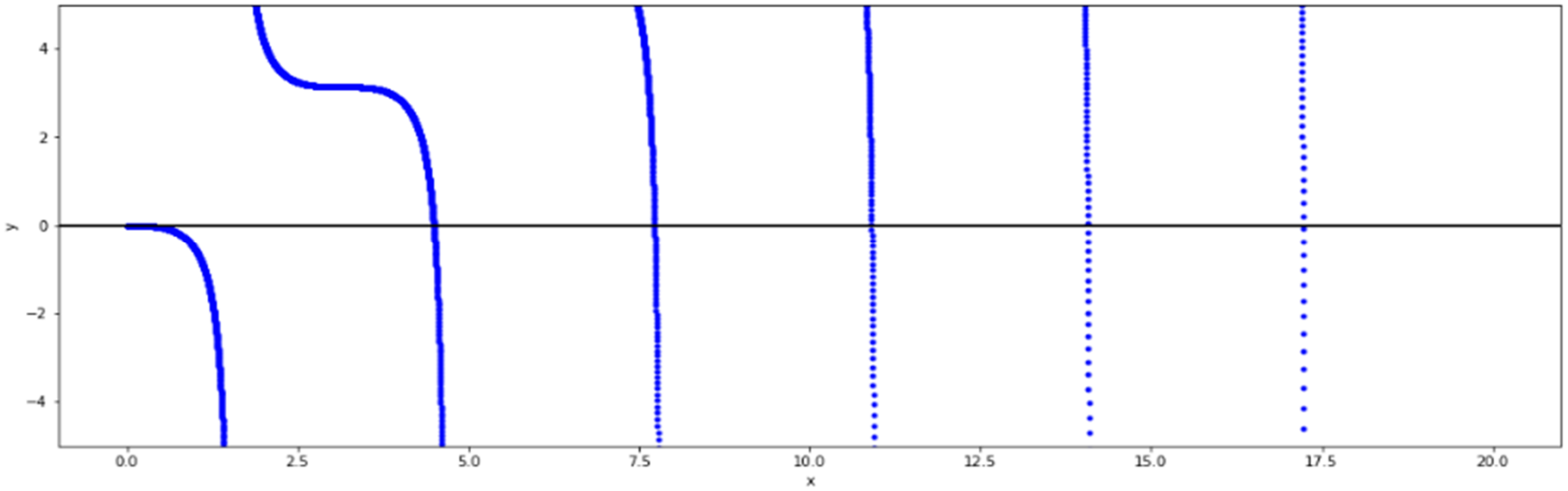
- A root of $x^3 - 10x^2 + 5 = 0$ lies in [0, 1].

- Use the bisection algorithm to compute this root with four-digit accuracy.
  - Root = 0.7346

- How many function evaluation will be performed at most ?
  - n=14 (+2 evaluation at start)

Example 2

- Find ALL roots of $x - \tan(x) = 0$ in [0, 20].
  - They are poles and distance between roots > 0.01
- We use both root search and bisection algorithms
- We make sure f at half point decrease to avoid poles



Example 2

```
a,b,dx = (0.0, 20.0, 0.01)
print("The roots are:")
while True:
    x1,x2 = rootsearch(f,a,b,dx)
    if x1 != None:
        a = x2
        root = bisection(f,x1,x2,1)
        if root != None: print(root)
    else:
        print("\nDone")
        break
```

executed in 29ms, finished 13:34:29 2019-10-24

```
The roots are:
0.0
4.493409458100745
7.725251837074637
10.904121659695917
14.066193912923 08
17.220755272209537

Done
```

Methods Based on Linear Interpolation

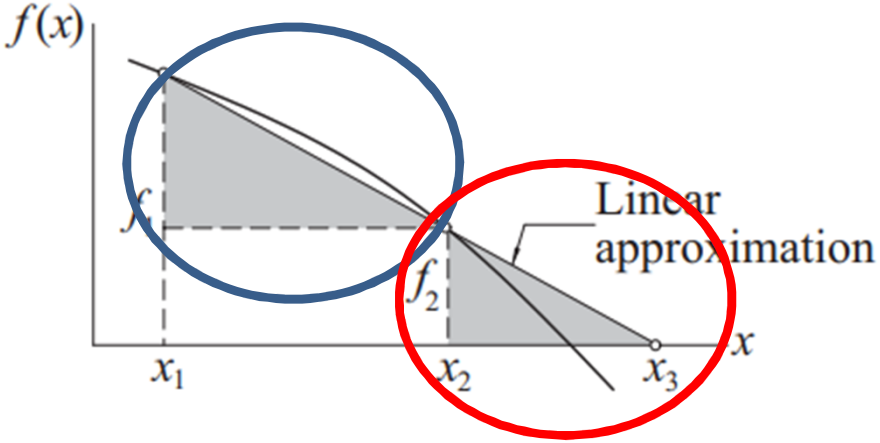
Secant and False Position Methods (1)

- Secant and False Position Methods are closely related methods
- The function $f(x)$ is assumed to be approximately linear near the root
- The improved value x_3 of the root can be estimated by linear interpolation

Secant and False Position Methods (2)

- With the notation

$$f_i = f(x_i)$$



- Similar triangles lead to:

$$\frac{f_2}{x_3 - x_2} = \frac{f_1 - f_2}{x_2 - x_1}$$

- Therefore:

$$x_3 = x_2 - f_2 \frac{x_2 - x_1}{f_2 - f_1}$$

The False Position Method

- The false position method (aka Regula Falsi) requires x_1 and x_2 to bracket the root
- After x_3 has been computed, x_1 or x_2 is replaced by x_3 according to the sign of f_1, f_2 and f_3
- The procedure is repeated until convergence

The secant Method

- The secant method does not requires x_1 and x_2 to bracket the root

- After x_3 has been computed
  - x_1 is replaced by x_2
  - x_2 is replaced by x_3

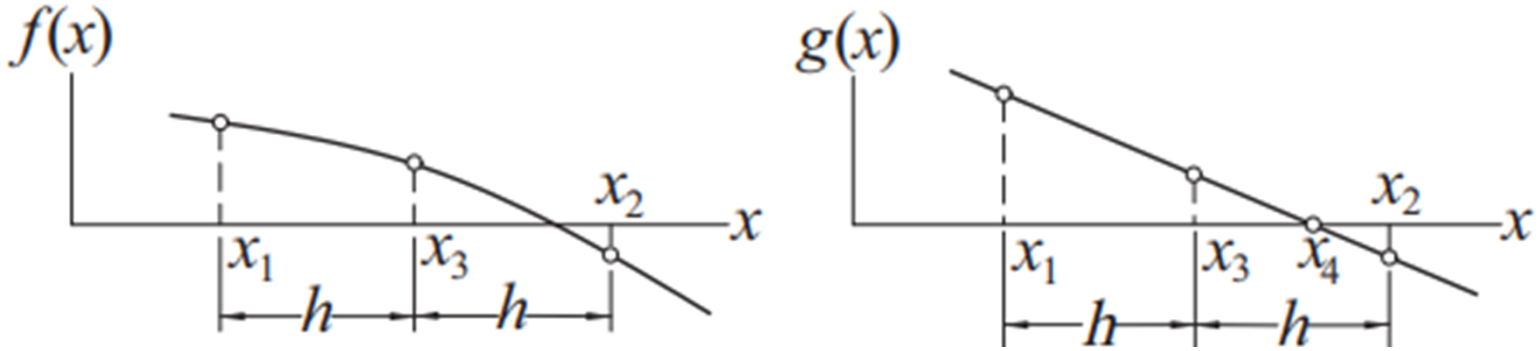
- The procedure is repeated until convergence

Comparison

- Convergence
  - Secant: has superlinear convergence
    - Error evolves has $E_{k+1} = cE_k^{1.618}$ (1.618 = golden ratio)
  - Regula Falsi: Impossible to calculate
    - generally a bit better than linear, but not so much
- Reliability
  - Regula Falsi always bracket the root, so it is more reliable
  - Secant might be diverging
- Both methods are superseded by Ridder's

Ridder's method (1)

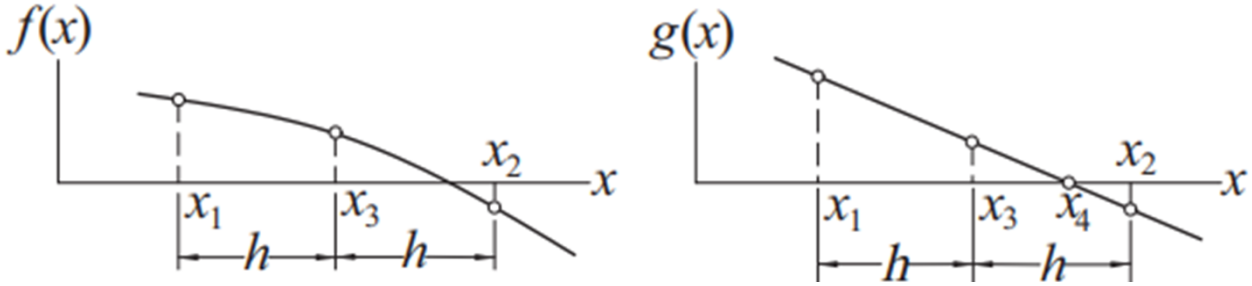
- Improvement on Regula Falsi
  - Compute f_3 with x_3 as the midpoint in $[x_1, x_2]$
- Introduce $g(x) = f(x)e^{(x-x_1)Q}$
  - Where Q is such that (x_1, g_1), $(x_2, g_2)$and (x_3, g_3) lies on a straight line



- The improved value of the root is then obtained by linear interpolation of g(x) rather than f(x).

Ridder's method (2)

- $g_1 = f_1$,
- $g_2 = f_2 e^{2hQ}$
- $g_3 = f_3 e^{hQ}$



- Straight line requirement → $g_3 = (g_1 + g_2)/2$

quadratic equation in e^{hQ}.

$$f_3 e^{hQ} = \frac{1}{2}(f_1 + f_2 e^{2hQ}) \qquad e^{hQ} = \frac{f_3 \pm \sqrt{f_3^2 - f_1 f_2}}{f_2}$$

Ridder's method (3)

- Linear interpolation based on (x_1, g_1) and (x_3, g_3) to compute x_4

$$x_4 = x_3 - g_3 \frac{x_3 - x_1}{g_3 - g_1} = x_3 - f_3 e^{hQ} \frac{x_3 - x_1}{f_3 e^{hQ} - f_1}$$

- Then we can substitute e^{hQ} by it's value

$$x_4 = x_3 \pm (x_3 - x_1) \frac{f_3}{\sqrt{f_3^2 - f_1 f_2}}$$

- We get the correct results if we choose
  - + sign for $f_1 - f_2 > 0$
  - - sign for $f_1 - f_2 < 0$
- After x_4 is computed, new bracket are determined for the root and the algorithm is repeated until convergence

Ridder's method (4)

- Properties
  - x_4 is always within the interval $[x_1, x_2]$
  - Each iteration requires two function evaluations
    - They are method that require only one function evaluation but they are more complex and requires bookeeping
  - Convergence is quadratic, which makes it more efficient than both regula falsi or secant methods
  - It is the method to use if the derivative of $f(x)$ is impossible or difficult to compute

Ridder's algorithm

```
def ridder(f,a,b,tol=1.0e-9):
    fa = f(a)
    if fa == 0.0: return a
    fb = f(b)
    if fb == 0.0: return b
    if sign(fa) == sign(fb): error.err('Root is not bracketed')
    for i in range(30):
      # Compute the improved root x from Ridder's formula
        c = 0.5*(a + b); fc = f(c)
        s = math.sqrt(fc**2 - fa*fb)
        if s == 0.0: return None
        dx = (c - a)*fc/s
        if (fa - fb) < 0.0: dx = -dx
        x = c + dx; fx = f(x)
      # Test for convergence
        if i > 0:
            if abs(x - xOld) < tol*max(abs(x),1.0): return x
        xOld = x
      # Re-bracket the root as tightly as possible
        if sign(fc) == sign(fx):
            if sign(fa)!= sign(fx): b = x; fb = fx
            else: a = x; fa = fx
        else:
            a = c; b = x; fa = fc; fb = fx
    return None
    print('Too many iterations')
```

Example

- A root of $x^3 - 10x^2 + 5 = 0$ lies in [0.6, 0.8].

- How many function evaluation will be performed at most ?

Solution. The starting points are

$$x_1 = 0.6 \qquad f_1 = 0.6^3 - 10(0.6)^2 + 5 = 1.6160$$

$$x_2 = 0.8 \qquad f_2 = 0.8^3 - 10(0.8)^2 + 5 = -0.8880$$

Solution taken from the reference book (Numerical Methods in Engineering With Python 3)

Example

First Iteration. Bisection yields the point

$$x_3 = 0.7 \qquad f_3 = 0.7^3 - 10(0.7)^2 + 5 = 0.4430$$

The improved estimate of the root can now be computed with Ridder's formula:

$$s = \sqrt{f_3^2 - f_1 f_2} = \sqrt{0.4330^2 - 1.6160(-0.8880)} = 1.2738$$

$$x_4 = x_3 \pm (x_3 - x_1)\frac{f_3}{s}$$

Because $f_1 > f_2$ we must use the plus sign. Therefore,

$$x_4 = 0.7 + (0.7 - 0.6)\frac{0.4430}{1.2738} = 0.7348$$

$$f_4 = 0.7348^3 - 10(0.7348)^2 + 5 = -0.0026$$

As the root clearly lies in the interval (x_3, x_4), we let

$$x_1 \leftarrow x_3 = 0.7 \qquad f_1 \leftarrow f_3 = 0.4430$$

$$x_2 \leftarrow x_4 = 0.7348 \qquad f_2 \leftarrow f_4 = -0.0026$$

Solution taken from the reference book (Numerical Methods in Engineering With Python 3)

Example

Second Iteration

$$x_3 = 0.5(x_1 + x_2) = 0.5(0.7 + 0.7348) = 0.717\,4$$

$$f_3 = 0.717\,4^3 - 10(0.717\,4)^2 + 5 = 0.2226$$

$$s = \sqrt{f_3^2 - f_1 f_2} = \sqrt{0.2226^2 - 0.4430(-0.0026)} = 0.2252$$

$$x_4 = x_3 \pm (x_3 - x_1)\frac{f_3}{s}$$

Since $f_1 > f_2$, we again use the plus sign, so that

$$x_4 = 0.717\,4 + (0.717\,4 - 0.7)\frac{0.2226}{0.2252} = 0.7346$$

$$f_4 = 0.7346^3 - 10(0.7346)^2 + 5 = 0.0000$$

Thus the root is $x = 0.7346$, accurate to at least four decimal places.

Solution taken from the reference book (Numerical Methods in Engineering With Python 3)

Newton-Raphson Method

Introduction

- Best known root finding method
  - Simple and Fast

- BUT
  - It uses both $f(x)$ and its derivative $f'(x)$
  - Can only be used if $f'(x)$ can easily be computed

- Derived from Taylor expansion of $f(x)$ about x
  - $f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + O(x_{i+1} - x_i)^2$

Introduction

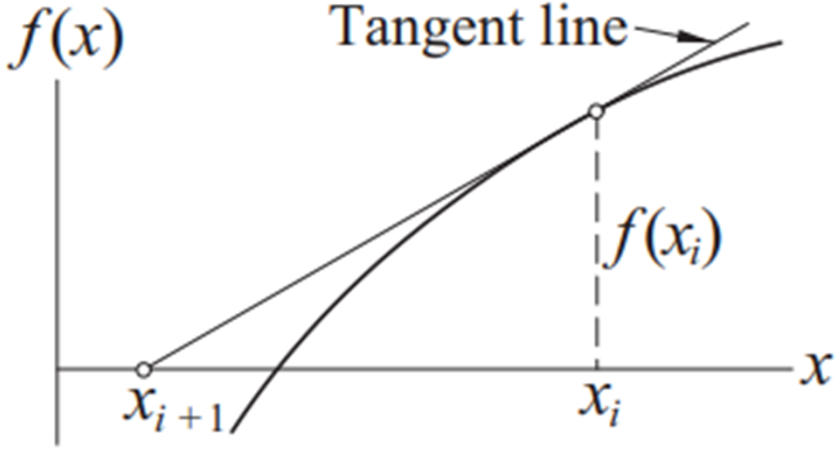
- Assuming that we are close to the solution, we can neglect the last term $O(x_{i+1} - x_i)^2$

- Taylor expansion around 0 become :

$$0 = f(x_i) + f'(x_i)(x_{i+1} - x_i)$$

- This lead to the solution

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



Algorithm

- Choose an initial value

  x_0

- Repeatedly apply:

  $$x_{i+1} = x_i - \frac{f(x_i)}{f\prime(x_i)}$$

- Stop when convergence is achieved:

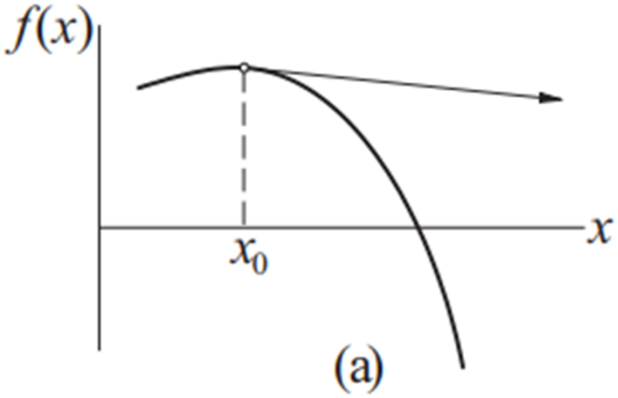
  $$|x_{i+1} - x_i| < \varepsilon$$

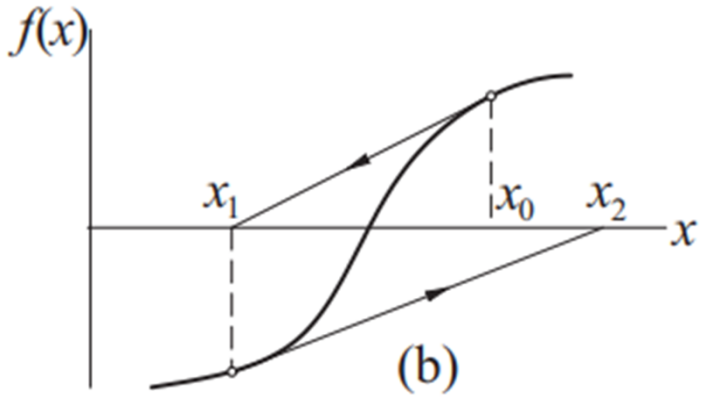
Remarks

- Convergence:
  - The error evolves has:
  - Convergence is quadratic

$$E_{i+1} = -\frac{f''(x)}{2f'(x)} E_i^2$$

- Newton-Raphson converges fast close to the root, but it's global convergence characteristic is poor
  - Tangent line is not always an acceptable approximation of the function. → Can even diverge in some cases
  - Combining it with bisection method can be a solution





Algorithm

Advanced algorithm that combines NR with the bisection method

```
def newtonRaphson(f,df,a,b,tol=1.0e-9):
    import error
    from numpy import sign

    fa = f(a)
    if fa == 0.0: return a
    fb = f(b)
    if fb == 0.0: return b
    if sign(fa) == sign(fb): error.err('Root is not bracketed')
    x = 0.5*(a + b)
    for i in range(30):
        fx = f(x)
        if fx == 0.0: return x
      # Tighten the brackets on the root
        if sign(fa) != sign(fx): b = x
        else: a = x
      # Try a Newton-Raphson step
        dfx = df(x)
      # If division by zero, push x out of bounds
        try: dx = -fx/dfx
        except ZeroDivisionError: dx = b - a
        x = x + dx
      # If the result is outside the brackets, use bisection
        if (b - x)*(x - a) < 0.0:
            dx = 0.5*(b - a)
            x = a + dx
      # Check for convergence
        if abs(dx) < tol*max(abs(b),1.0): return x
    print('Too many iterations in Newton-Raphson')
```

Example

- Compute sqrt(2) as the solution of $x^2 - 2 = 0$

- Using Newton-Raphson properties, we have:

$$x \leftarrow x - \frac{f(x)}{f'(x)} = x - \frac{x^2 - 2}{2x} = \frac{x^2 + 2}{2x}$$

- We can start with x=1

$$x \leftarrow \frac{(1)^2 + 2}{2(1)} = \frac{3}{2}$$

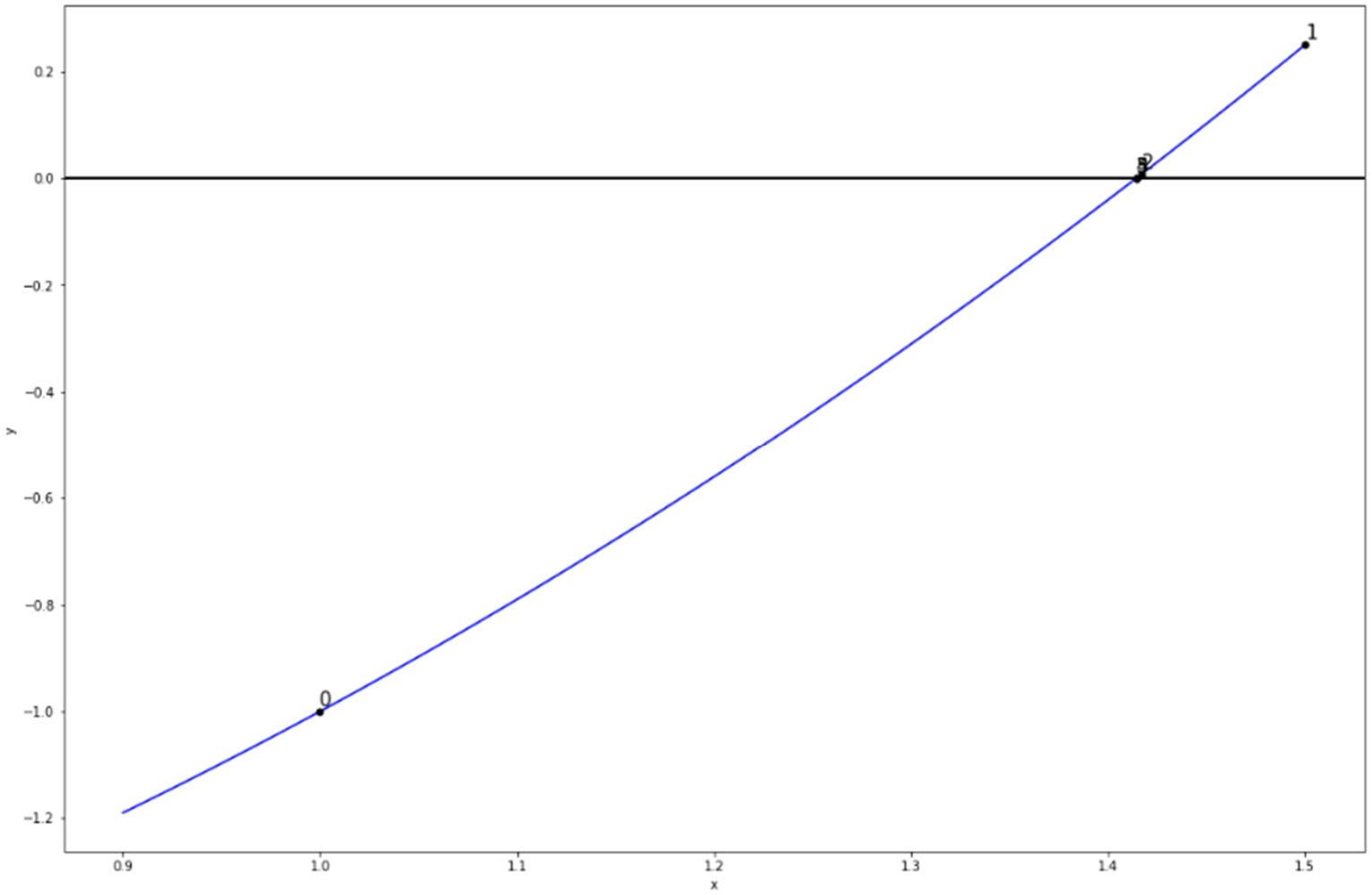
$$x \leftarrow \frac{(3/2)^2 + 2}{2(3/2)} = \frac{17}{12}$$

$$x \leftarrow \frac{(17/12)^2 + 2}{2(17/12)} = \frac{577}{408}$$

x = 577/408 = 1.1414216
that's already pretty close to
√2 = 1.1414214.

Example

- Visually checking the sequence of points



Systems of Equations

Introduction

- So far we focused on solving the single equation

$$f(x) = 0$$

- Let's now consider the n-dimensionnal system:

$$\boldsymbol{f}(\boldsymbol{x}) = \mathbf{0}$$

Or in scalar notation:

$$f_1(x_1, x_2, \dots, x_n) = 0$$
$$\vdots$$
$$f_n(x_1, x_2, \dots, x_n) = 0$$

Introduction

- Solving n simultaneous, nonlinear equations sounds more interesting than finding the root of one dimensional function.

- The trouble is there is no a reliable method for bracketing the solution vector **x**

- We cannot always provide the solution algorithm with a good starting value of **x**, unless such a value is suggested by the physics of the problem

Newton-Raphson Method (1)

- Newton-Raphson Method works well with simultaneous equations provided that it is supplied with a good starting point.

- Again, we start from Taylor expansion

$$f_i(\boldsymbol{x} + \boldsymbol{\Delta x}) = f_i(\boldsymbol{x}) + \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j}(\boldsymbol{x})\boldsymbol{\Delta x} + O(\boldsymbol{\Delta x}^2)$$

- Neglecting the term in $\boldsymbol{\Delta x}^2$, we get:

$$\boldsymbol{f}(\boldsymbol{x} + \boldsymbol{\Delta x}) = \boldsymbol{f}(\boldsymbol{x}) + \mathbf{J}(\boldsymbol{x})\boldsymbol{\Delta x} \quad \text{with} \quad \mathrm{J}_{ij} = \frac{\partial f_i}{\partial x_j}$$

Newton-Raphson Method (2)

- $\mathrm{J}_{ij} = \frac{\partial f_i}{\partial x_j}$ is the **Jacobian** Matrix of size (n x n)
- The jacobian matrix is made up of all partial derivatives

- Let's now assume that we are close to the root:

$$\boldsymbol{f}(\boldsymbol{x} + \boldsymbol{\Delta x}) = 0 \rightarrow \mathbf{J}(\boldsymbol{x})\boldsymbol{\Delta x} = -\boldsymbol{f}(\boldsymbol{x})$$

- It's hard for the computer to compute $\frac{\partial f_i}{\partial x_j}$, it is easier to make the computation based on the finite difference approximation:

$$\frac{\partial f_i}{\partial x_j}(\boldsymbol{x}) \approx \frac{f_i(\boldsymbol{x} + \mathbf{e_j} h) - f_i(\boldsymbol{x})}{h}$$

- Where $\mathbf{e_j}$ represents a unit vector in the direction of $\boldsymbol{x_j}$ and where h is a small increment

Newton-Raphson Method (3)

- We can approximate the **Jacobian** Matrix by the finite difference because Newton-Raphson is rather insensitive to error in $\mathbf{J}(\boldsymbol{x})$
- This has also the advantage to avoid the boring work of typing all the $\frac{\partial f_i}{\partial x_j}$ into the computer code

- **The Algorithm is therefore:**
  - Take a starting point $\boldsymbol{x_0}$
  - Do until$|\ \Delta x\ | < \varepsilon$:
    - Compute $\mathbf{J}(\boldsymbol{x})$
    - Solve $\mathbf{J}(\boldsymbol{x})\boldsymbol{\Delta}\mathbf{x} = -\boldsymbol{f}(\boldsymbol{x})$ for $\boldsymbol{\Delta}\mathbf{x}$
    - Let $\mathbf{x} \leftarrow \mathbf{x} + \boldsymbol{\Delta}\mathbf{x}$

Algorithm

```
def newtonRaphson2(f,x,tol=1.0e-9):

    def jacobian(f,x):
        h = 1.0e-4
        n = len(x)
        jac = np.zeros((n,n))
        f0 = f(x)
        for i in range(n):
            temp = x[i]
            x[i] = temp + h
            f1 = f(x)
            x[i] = temp
            jac[:,i] = (f1 - f0)/h
        return jac,f0

    for i in range(30):
        jac,f0 = jacobian(f,x)
        if math.sqrt(np.dot(f0,f0)/len(x)) < tol:
            return x
        dx = gaussPivot(jac,-f0)
        x = x + dx
        if math.sqrt(np.dot(dx,dx)) < tol*max(max(abs(x)),1.0): return x
    print('Too many iterations')
```

Example

- Determine the points of intersection between
  - the circle $x^2 + y^2 = 3$
  - the hyperbola $xy = 1$.

Solution. The equations to be solved are

$$f_1(x, y) = x^2 + y^2 - 3 = 0 \qquad \text{(a)}$$

$$f_2(x, y) = xy - 1 = 0 \qquad \text{(b)}$$

The Jacobian matrix is

$$\mathbf{J}(x, y) = \begin{bmatrix} \partial f_1/\partial x & \partial f_1/\partial y \\ \partial f_2/\partial x & \partial f_2/\partial y \end{bmatrix} = \begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix}$$

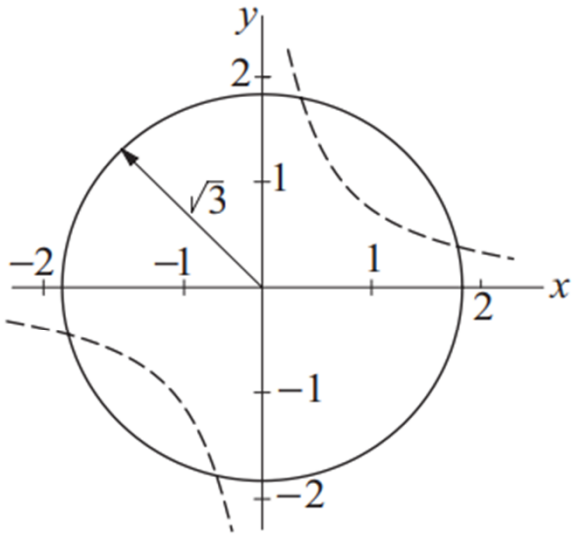
Thus the linear equations $\mathbf{J}(\mathbf{x})\boldsymbol{\Delta}\mathbf{x} = -\mathbf{f}(\mathbf{x})$ associated with the Newton-Raphson method are

$$\begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -x^2 - y^2 + 3 \\ -xy + 1 \end{bmatrix} \qquad \text{(c)}$$

Solution taken from the reference book (Numerical Methods in Engineering With Python 3)

Example

By plotting the circle and the hyperbola, we see that there are four points of intersection. It is sufficient, however, to find only one of these points, because the others can be deduced from symmetry. From the plot we also get a rough estimate of the coordinates of an intersection point, $x = 0.5$, $y = 1.5$, which we use as the starting values.



The computations then proceed as follows.

Solution taken from the reference book (Numerical Methods in Engineering With Python 3)

Example

First Iteration. Substituting $x = 0.5$, $y = 1.5$ in Eq. (c), we get

$$\begin{bmatrix} 1.0 & 3.0 \\ 1.5 & 0.5 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} 0.50 \\ 0.25 \end{bmatrix}$$

the solution of which is $\Delta x = \Delta y = 0.125$. Therefore, the improved coordinates of the intersection point are

$$x = 0.5 + 0.125 = 0.625 \qquad y = 1.5 + 0.125 = 1.625$$

Second Iteration. Repeating the procedure using the latest values of x and y, we obtain

$$\begin{bmatrix} 1.250 & 3.250 \\ 1.625 & 0.625 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -0.031250 \\ -0.015625 \end{bmatrix}$$

which yields $\Delta x = \Delta y = -0.00694$. Thus

$$x = 0.625 - 0.006\,94 = 0.618\,06 \qquad y = 1.625 - 0.006\,94 = 1.618\,06$$

Solution taken from the reference book (Numerical Methods in Engineering With Python 3)

Example

Third Iteration. Substitution of the latest x and y into Eq. (c) yields

$$\begin{bmatrix} 1.236\,12 & 3.23612 \\ 1.618\,06 & 0.61806 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -0.000\,116 \\ -0.000\,058 \end{bmatrix}$$

The solution is $\Delta x = \Delta y = -0.00003$, so that

$$x = 0.618\,06 - 0.000\,03 = 0.618\,03$$

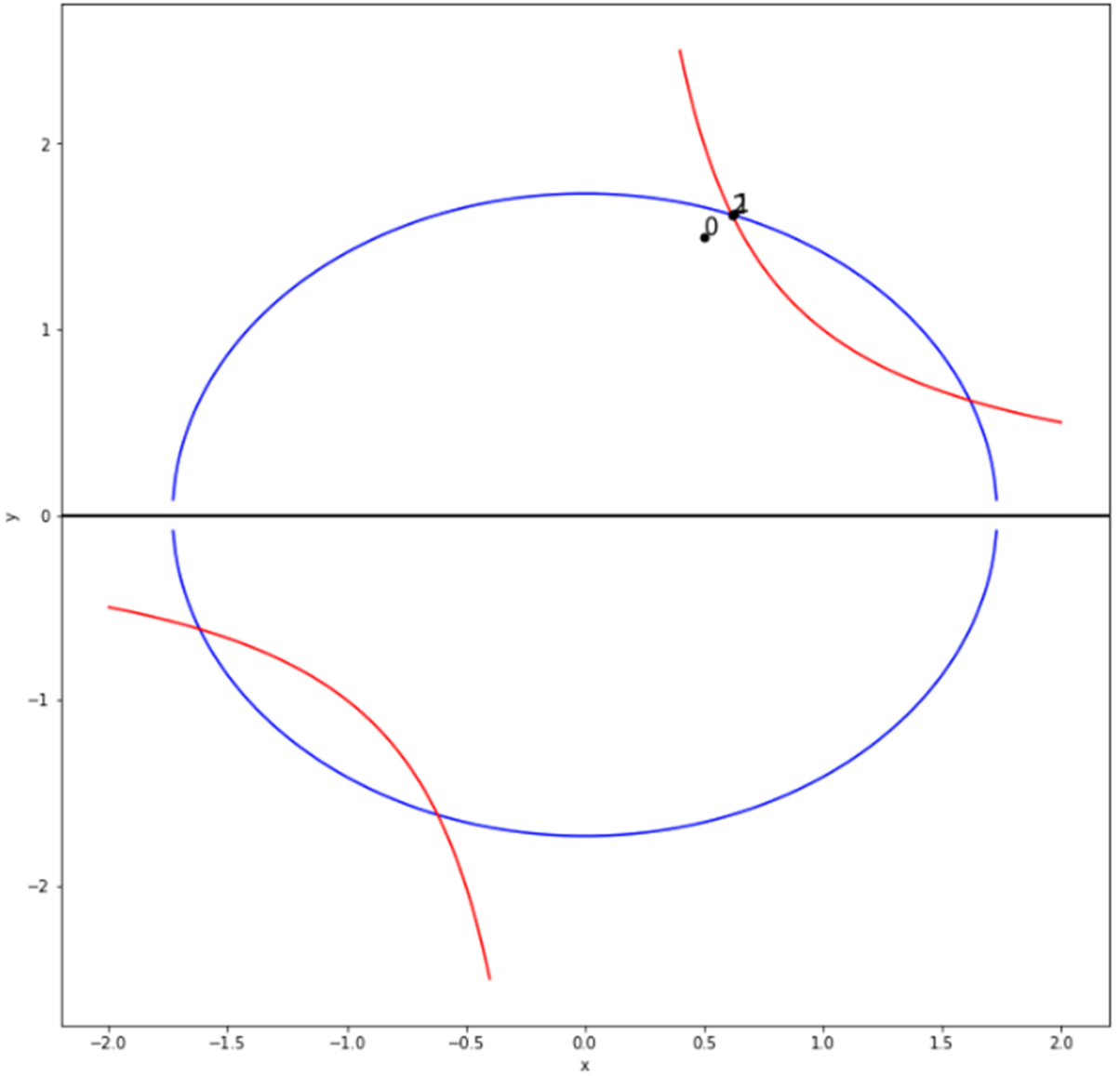
$$y = 1.618\,06 - 0.000\,03 = 1.618\,03$$

Subsequent iterations would not change the results within five significant figures. Therefore, the coordinates of the four intersection points are

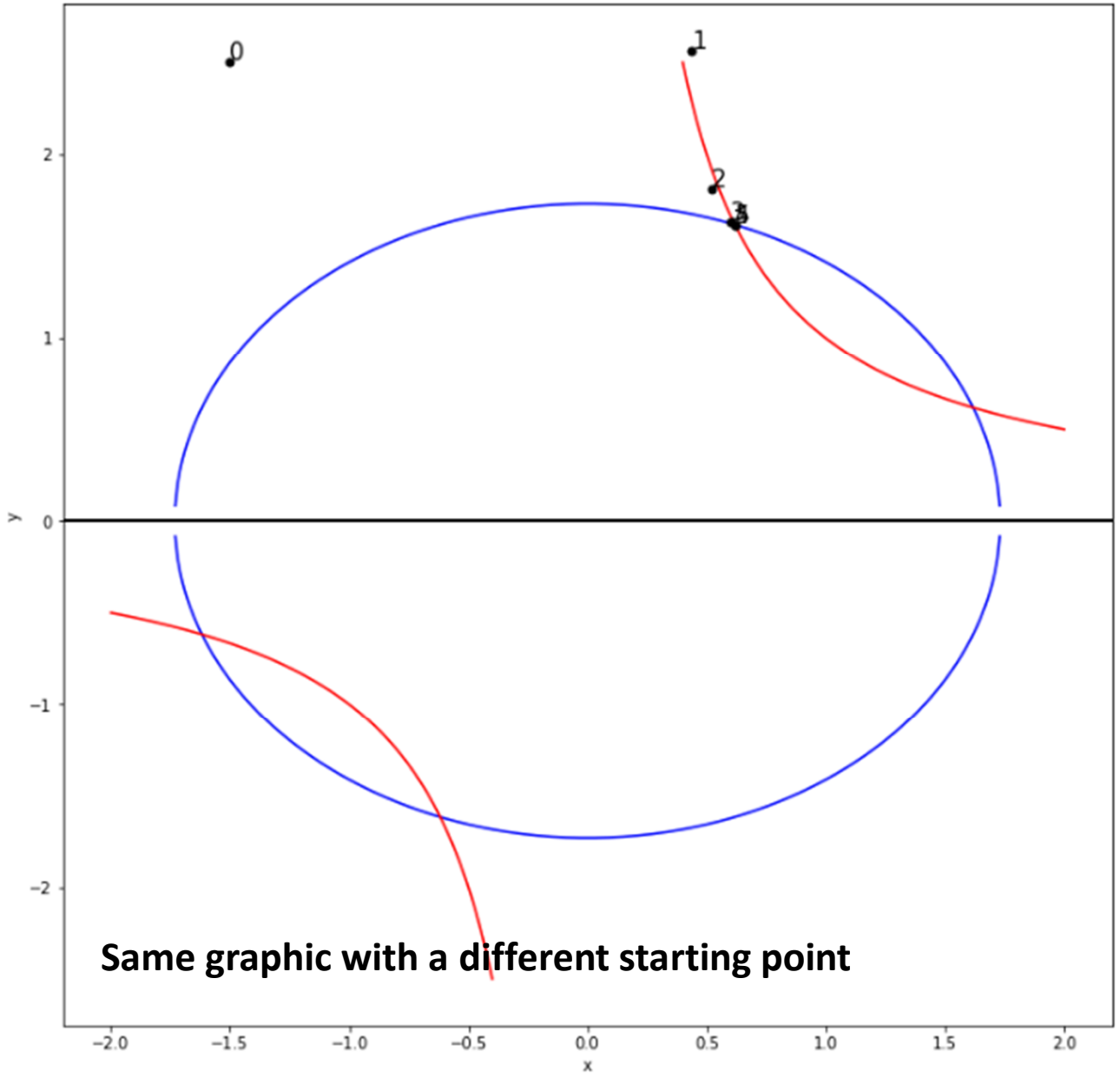
$$\pm(0.618\,03,\ 1.618\,03) \text{ and } \pm\,(1.618\,03,\ 0.618\,03)$$

Solution taken from the reference book (Numerical Methods in Engineering With Python 3)

Visual representation of the convergence



Visual representation of the convergence



Same graphic with a different starting point