Roots of Equations

Mark A. Austin

University of Maryland

austin@umd.edu

ENCE 201, Fall Semester 2023

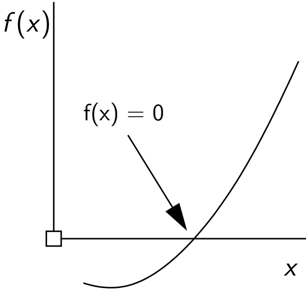
September 30, 2023

Overview

Numerical Solution of Equations

Numerical Solution of Equations

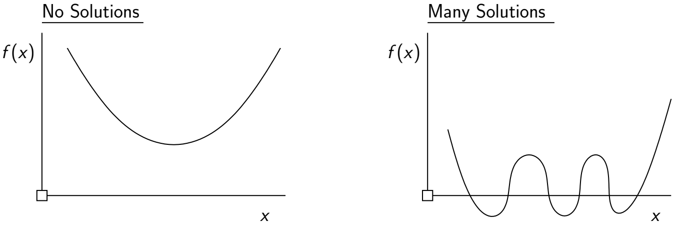
Math Problem. Given f(x), find a value of x such that f(x) = g(x), f(x) = constant, or f(x) = 0.



All forms may be put in the format F(x) = 0.

Numerical Solution of Equations

Mathematical Difficulties.



Quality of a Solution

Several possibilities exist:

- Solution x^* is good if $f(x^*) \approx 0.0$
- Solution x^* is good if it is close to the exact answer.

Easy to find functions that satisfy one criteria, but not both.

Numerical Solution of Equations

Example 1. Consider the equation:

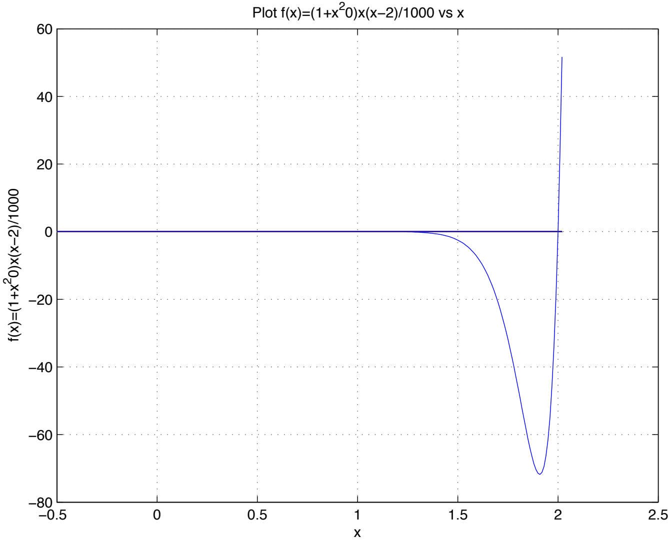
$$f(x) = \left[\frac{(x^{20}+1)x(x-2)}{1000}\right] \qquad (1)$$

We know x = 0 and x = 2 are roots, but:

- x = 0.123 satisfies (i) but not (ii).
- x = 2.001 satisfies (ii) but not (i).

x	F(x)
0.123	-2.31 $\times$ 10^{-4}
2.001	2.1200
0.000	0.0000
2.000	0.0000

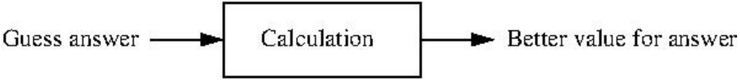
Numerical Solution of Equations



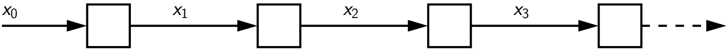
Iterative Methods

Iterative Methods

Procedure. Solve problem through a sequence of approximations:



Apply process iteratively:



Ideally, $x_0, x_1, \cdots, x_n$ will converge to the true answer.

Potential problems:

- Sequence may not converge.
- Convergence may be slow.

Iterative Methods

Example 1. Divide-and-average method for computing $\sqrt{A}$ is equivalent to solving:

$$x^2 = A \Longrightarrow x = \frac{A}{x} \Longrightarrow \frac{1}{2}\left[x + \frac{A}{x}\right] \Longrightarrow x_{n+1} = \frac{1}{2}\left[x_n + \frac{A}{x_n}\right]. \quad (2)$$

Let A $= 4$. Use initial guess $x_1 = 1 \approx \sqrt{4}$.

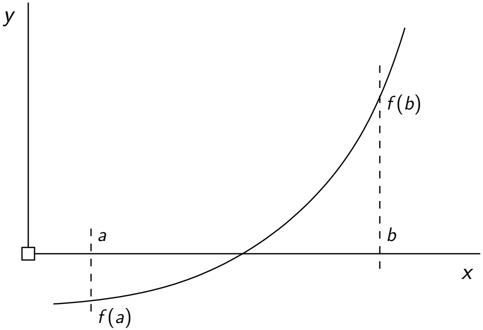
n	x_n	x_{n+1}
1	1.0000	2.5000
2	2.5000	2.0500
3	2.0500	2.0060
4	2.0060	2.0000

Problem Solving

Strategies

Problem Solving Strategies

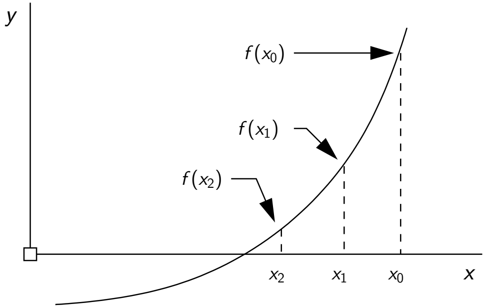
Bracketing Methods: Requires two initial guesses that bracket the solution.



- Various algorithms for computing estimates to $f(x) = 0$, e.g, Bisection, Secant stiffness.

Problem Solving Strategies

Open Methods: Methods may involve one or more initial guesses, but no need to bracket a solution.



- Algorithms are designed to provide updates: Newton Raphson Iteration, Modified Newton Raphson.

Method of Bisection

Method of Bisection

A reliable method for solving $f(x) = 0$.

Fact. Suppose we have continuous function $f(x)$. If $f(a) < 0$ and $f(b) > 0$ then there exists a point c in $[a, b]$ such that $f(c) = 0$.

Numerical Procedure. Find initial points a and b such that $f(a)$ and $f(b)$ have opposite signs. Let $x_{left} = a$ and $x_{right} = b$.

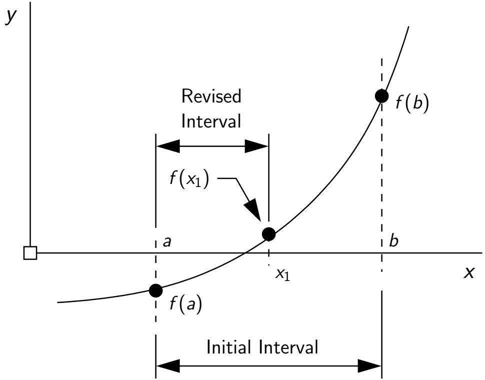
- Evaluate at mid-point: $x_{new} = \frac{1}{2}[x_{left} + x_{right}]$.
- Look for change in sign in function evaluation.

  Keep $f(x_{left})$ if $f(x_{new}).f(x_{left}) < 0$.

  Otherwise, keep $f(x_{right})$ if $f(x_{new}).f(x_{right}) < 0$.
- Repeat until solution converges.

Method of Bisection

Schematic: One iteration of Bisection:



For iteration 2, we set $x_{left} = f(a)$ and $x_{right} = f(x_1)$.

Method of Bisection

Example 1. Demonstrate use of bisection method to compute roots of the quadratic.

$$f(x) = (x-3)*(x-3) - 2 = 0; \qquad (3)$$

Analytic Solution: From equation 3:

$$(x-3)^2 = 2 \implies [x_1, x_2] = \left[3 - \sqrt{2}, 3 + \sqrt{2}\right]. \qquad (4)$$

Source Code:

- TestBisection01.py: Test program and functions for bisection algorithm ...
- Solutions.py: Python code for bisection algorithm.

Method of Bisection

Test Program Source Code:

```
# ===========================================================================
# TestBisection01.py: Use bisection algorithm to compute roots of equations.
#
# Written By: Mark Austin                                     February 2023
# ===========================================================================

import math;
import Solutions;

# Define mathematical functions ...

def f1(x):
    return (x-3)*(x-3)-2;

# main method ...

def main():
    print("--- Enter TestBisection01.main()                  ... ");
    print("--- ======================================== ... ");

    print("--- ");
    print("--- Case Study 1: Solve (x-3)*(x-3)-2 = 0 ... ");
    print("--- ==================================== ... ");

    # Initialize problem setup ...
```

Method of Bisection

Test Program Source Code: Continued ...

```
    a =  -1.0;
    b =   2.0
    tolerance     = 0.01
    maxiterations = 100

    print("--- Inputs:")
    print("---   a = {:5.2f} ...".format(a) )
    print("---   b = {:5.2f} ...".format(b) )
    print("---   tolerance      = {:8.5f} ...".format(tolerance) )
    print("---   max iterations = {:8.2f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.bisection(f1, a, b, tolerance, maxiterations )

    # Summary of computations ...

    print("--- Output:")
    print("---   root = {:10.5f} ...".format(root) )
    print("---   f(root) --> {:12.5e} ...".format( f1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

    print("--- ");
    print("--- Case Study 2: Solve 2x^3 - cos(x+1) - 3 = 0 ... ");
    print("--- ========================================== ... ");
```

Method of Bisection

Test Program Source Code: Continued ...

```
    # Initialize problem setup ...

    a = -1.0;
    b =  2.0
    tolerance     = 0.01
    maxiterations = 100

    print("--- Inputs:")
    print("---   a = {:5.2f} ...".format(a) )
    print("---   b = {:5.2f} ...".format(b) )
    print("---   tolerance      = {:8.5f} ...".format(tolerance) )
```

Abbreviated Output:

```
--- Case Study 1: Solve (x-3)*(x-3)-2 = 0 ...
--- ===================================== ...
--- Inputs:
---   a = -1.00 ...
---   b =  2.00 ...
---   tolerance      =  0.01000 ...
---   max iterations =   100.00 ...
```

Method of Bisection

Abbreviated Output: Continued ...

```
--- Execution:
---   Initial Conditions:
---   f(a) -->  1.40000e+01 ...
---   f(b) --> -1.00000e+00 ...
---   Main Loop for Root Computation:
---   Iteration 00: dx = 1.50000e+00, x =  5.00000e-01, f(x) ->  4.25000e+00
---   Iteration 01: dx = 7.50000e-01, x =  1.25000e+00, f(x) ->  1.06250e+00
---   Iteration 02: dx = 3.75000e-01, x =  1.62500e+00, f(x) -> -1.09375e-01
---   Iteration 03: dx = 1.87500e-01, x =  1.43750e+00, f(x) ->  4.41406e-01
---   Iteration 04: dx = 9.37500e-02, x =  1.53125e+00, f(x) ->  1.57227e-01
---   Iteration 05: dx = 4.68750e-02, x =  1.57812e+00, f(x) ->  2.17285e-02
---   Iteration 06: dx = 2.34375e-02, x =  1.60156e+00, f(x) -> -4.43726e-02
---   Iteration 07: dx = 1.17188e-02, x =  1.58984e+00, f(x) -> -1.14594e-02
---   Iteration 08: dx = 5.85938e-03, x =  1.58398e+00, f(x) ->  5.10025e-03
--- Output:
---   root =    1.58398 ...
---   f(root) -->  5.10025e-03 ...
---   no iterations = 8 ...
---   converged: True ...
```

Method of Bisection

Example 2. The test function

$$f(x) = \left[\frac{(x^{20}+1)x(x-2)}{1000}\right] \tag{5}$$

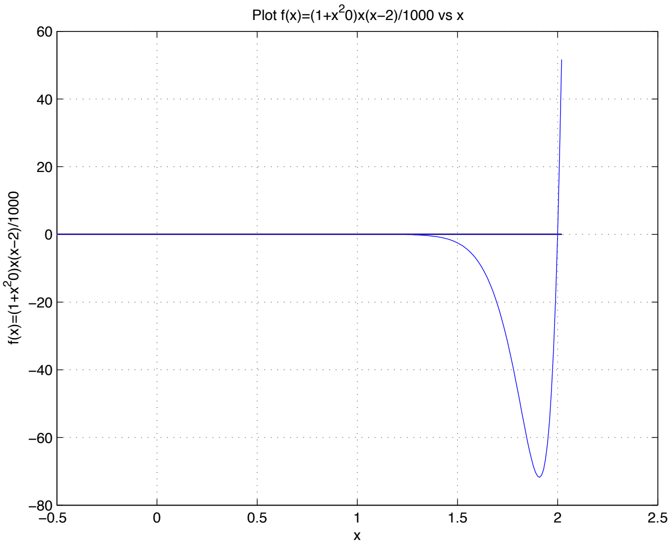
has two roots within the interval $[-1, 3]$.

From a numerical standpoint, this problem is challenging:

- In the neighborhood of $x = 0$, the test function values and slope are very close to zero.
- In the neighborhood of $x = 2$, the test function slope is extremely high.

We can break the solution into blocks:

Method of Bisection



Method of Bisection

Test Program Source Code:

```
# ============================================================================
# TestBisection02.py: Use bisection algorithm to compute roots of equations:
#
# Written By: Mark Austin                                        February 2023
# ============================================================================

import math;
import Solutions;

# Define mathematical functions ...

def f1(x):
    return (x**20 + 1)*x*(x-2)/1000.0;

# main method ...

def main():
    print("--- ");
    print("--- Case Study 1: Solve f(x) = ((x^20 + 1)x(x-2))/1000 = 0 ... ");
    print("--- ====================================================== ... ");

    # Initialize problem setup ...

    a =  0.5;
    b =  2.5
    tolerance     = 0.0001
```

Method of Bisection

Test Program Source Code: Continued ...

```
    maxiterations = 100

    print("--- Inputs:")
    print("---   a = {:5.2f} ...".format(a) )
    print("---   b = {:5.2f} ...".format(b) )
    print("---   tolerance      = {:8.5f} ...".format(tolerance) )
    print("---   max iterations = {:8.2f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.bisection(f1, a, b, tolerance, maxiterations )

    # Summary of computations ...

    print("--- Output:")
    print("---   root = {:12.7f} ...".format(root) )
    print("---   f(root) --> {:14.7e} ...".format( f1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

# call the main method ...

main()
```

Method of Bisection

Abbreviated Output: Solve f(x) = $((x^{20} + 1)x(x-2))/1000 = 0$

```
--- Inputs:
---   a =  0.50 ...
---   b =  2.50 ...
---   tolerance      =  0.00010 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---   f(a) --> -7.50001e-04 ...
---   f(b) -->  1.13687e+05 ...
---   Main Loop for Root Computation:
---   Iteration 00: dx = 1.0000e+00, x = 1.500000e+00, f(x) -> -2.494692e+00
---   Iteration 01: dx = 5.0000e-01, x = 2.000000e+00, f(x) ->  0.000000e+00
...
---   Iteration 24: dx = 5.9605e-08, x = 1.999999e+00, f(x) -> -1.250000e-04
---   Iteration 25: dx = 2.9802e-08, x = 2.000000e+00, f(x) -> -6.250004e-05
--- Output:
---   root =    2.0000000 ...
---   f(root) --> -6.2500040e-05 ...
---   no iterations = 25 ...
---   converged: True ...
```

Method of Bisection

Summary

- A reliable method for solving $f(x) = 0$.

Limitations

- Need to find two bracketing points before iteration can begin.
- Convergence can be slow.

Newton Raphson Iteration

Newton-Raphson Iteration

Derivation of Numerical Procedure. Starting point $(x_0, f(x_0))$.

We wish to find a steplength $h = x_1$ - x_0 that will provide an improved estimate of the root.

Using first-order Taylor's expansion:

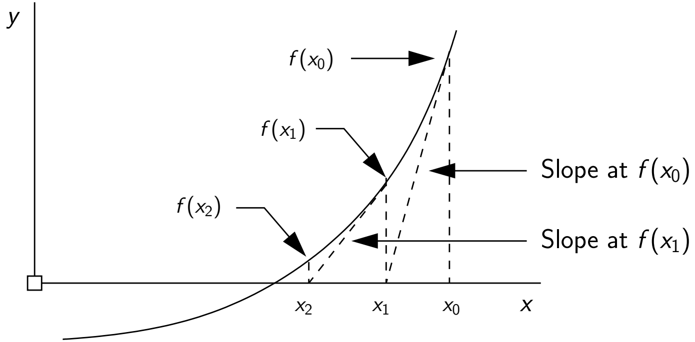
$$f(x_1) = f(x_0) + hf^{'}(x_0) + O(h^2) = 0.0 \tag{6}$$

Next, neglect $O(h^2)$ terms, rarrange, and generalize:

$$x_{n+1} = x_n - \left[\frac{f(x_n)}{f'(x_n)} \right]. \tag{7}$$

Newton-Raphson Iteration

Schematic: Two iterations of Newton-Raphson.



Sequence of estimates is: x_0, x_1, x_3, ...

Newton-Raphson Iteration

Example 1. Solve $f(x) = 0$ where

$$f(x) = xsin(x) - 3cos(x) \quad (8)$$

Differentiating,

$$f^{'}(x) = sin(x) + xcos(x) + 3sin(x) \quad (9)$$

The Newton-Raphson update is:

$$x_{n+1} = x_n - \left[\frac{x_n sin(x_n) - 3cos(x_n)}{sin(x_n) + x_n cos(x_n) + 3sin(x_n)}\right]. \quad (10)$$

This gives: $x_0 = 0.8$, x_1 - 1.24, $x_2 = 1.1927$ ....

Newton-Raphson Iteration

Example 2. Demonstrate use of newton-raphson algorithm by computing roots of the quadratic equation

$$f(x) = (x - 3) * (x - 3) - 2; \tag{11}$$

The derivative is given by:

$$df(x)/dx = 2x - 6. \tag{12}$$

The source code is partitioned into two Python:

1. Solutions.py: Contains function for newton raphson algorithm.
2. TestNewtonRaphson.py. main test program + f1(x) and df1(x).

Program Source Code

```
# ===========================================================================
# TestNewtonRaphson01.py: Use newton raphson algorithm to compute roots of
# equations.
#
# Written By: Mark Austin                                     February 2023
# ===========================================================================

import math;
import Solutions;

# Mathematical functions: (x-3)*(x-3) - 2 = 0 ...

def f1(x):
    return (x-3)*(x-3)-2;

def df1(x):
    return 2*(x-3);

# main method ...

def main():
    print("--- Enter TestNewtonRaphson01.main()          ... ");
    print("--- ======================================== ... ");

    print("--- ");
    print("--- Case Study 1: Solve (x-3)*(x-3)-2 = 0, Initial guess: x0 = -10 ... ");
    print("--- ========================================================= ... ");
```

Program Source Code

```
    # Initialize problem setup ...

    x0 = -10.0;
    tolerance     = 0.001
    maxiterations = 100

    print("--- Inputs:")
    print("---   x0 = {:5.2f} ...".format(x0) )
    print("---   tolerance      = {:8.5f} ...".format(tolerance) )
    print("---   max iterations = {:8.2f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )

    # Summary of computations ...

    print("--- Output:")
    print("---   root = {:10.5f} ...".format(root) )
    print("---   f(root) --> {:12.5e} ...".format( f1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

    print("--- ");
    print("--- Case Study 2: Solve (x-3)*(x-3)-2 = 0, Initial guess: x0 = 10 ... ");
    print("--- ============================================================ ... ");

    # Initialize problem setup ...
```

Program Source Code

```
    x0 = 10.0;
    tolerance     = 0.001
    maxiterations = 100

    print("--- Inputs:")
    print("---   x0 = {:5.2f} ...".format(x0) )
    print("---   tolerance      = {:8.5f} ...".format(tolerance) )
    print("---   max iterations = {:8.2f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )

    # Summary of computations ...

    print("--- Output:")
    print("---   root = {:10.5f} ...".format(root) )
    print("---   f(root) --> {:12.5e} ...".format( f1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

    print("--- ======================================== ... ");
    print("--- Leave TestNewtonRaphson01.main()         ... ");

# call the main method ...
```

Newton-Raphson Iteration

Abbreviated Output: Case Study 1, x0 = -10.

```
--- Inputs:
---   x0 = -10.00 ...
---   tolerance      =  0.00100 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---     x0     --> -1.00000e+01 ...
---     f(x0)  -->  1.67000e+02 ...
---     df(x0) --> -2.60000e+01 ...
---   Main Loop for Newton Raphson Iteration:
---   Iteration 01: dx = 6.42308e+00, x = -3.57692e+00, f(x) -> 4.12559e+01
---   Iteration 02: dx = 3.13641e+00, x = -4.40508e-01, f(x) -> 9.83710e+00
...
---   Iteration 06: dx = 2.60526e-03, x =  1.58578e+00, f(x) -> 6.78739e-06
---   Iteration 07: dx = 2.39970e-06, x =  1.58579e+00, f(x) -> 5.75895e-12
--- Output:
---   root =    1.58579 ...
---   f(root) -->  5.75895e-12 ...
---   no iterations = 7 ...
---   converged: True ...
```

Newton-Raphson Iteration

Abbreviated Output: Case Study 2, x0 = 10.

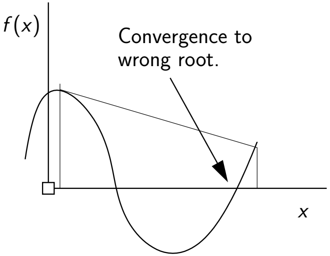
```
--- Inputs:
---   x0 = 10.00 ...
---   tolerance      =  0.00100 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---     x0     -->  1.00000e+01 ...
---     f(x0)  -->  4.70000e+01 ...
---     df(x0) -->  1.40000e+01 ...
---   Main Loop for Newton Raphson Iteration:
---   Iteration 01: dx = -3.35714e+00, x = 6.64286e+00, f(x) -> 1.12704e+01
---   Iteration 02: dx = -1.54692e+00, x = 5.09594e+00, f(x) -> 2.39296e+00
...
---   Iteration 05: dx = -4.02419e-03, x = 4.41422e+00, f(x) -> 1.61941e-05
---   Iteration 06: dx = -5.72546e-06, x = 4.41421e+00, f(x) -> 3.27804e-11
--- Output:
---   root =    4.41421 ...
---   f(root) -->  3.27804e-11 ...
---   no iterations = 6 ...
---   converged: True ...
```

Modified

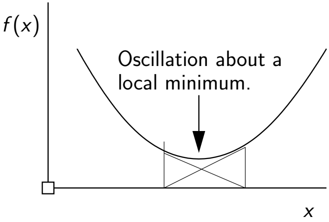
Newton Raphson Iteration

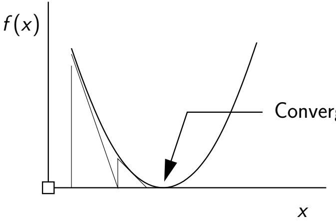
Limitations of Newton-Raphson Iteration

No Solutions



Many Solutions





Convergence to multiple roots: $\left[\frac{f(x)}{f'(x)}\right] = 0/0.$

Modified Newton-Raphson Iteration

Derivation of Numerical Procedure. In the case of multiple roots, we can improve on N-R by solving an equivalent problem:

$$F(x) = \left[\frac{f(x_n)}{f'(x_n)}\right] = 0. \tag{13}$$

Same solutions as f(x) = 0, but they occur as single roots.

Differentiating,

$$\frac{d}{dx}[F(x)] = \left[\frac{f(x_n)}{f'(x_n)}\right] = \left[\frac{(f'(x_n))^2 - f(x)f''(x)}{[f'(x_n)]^2}\right]. \tag{14}$$

Modified Newton-Raphson Iteration

Substituting into N-R formula:

$$x_{n+1} = x_n - \left[\frac{f(x_n)f'(x_n)}{(f'(x_n))^2 - f(x)f''(x)} \right]. \qquad (15)$$

Example 1. The function

$$f(x) = x^2 - 4x + 4, \;\; f'(x) = 2x - 4, \;\; f''(x) = 2. \qquad (16)$$

has a double root at $x = 2$. Using Newton-Raphson:

$$x_0 = 3.0$$

$$x_1 = 3.0 - \left[\frac{f(3.0)}{f'(3.0)} \right] = 3 - 1/2 = 2.5.$$

Modified Newton-Raphson Iteration

$$x_2 = 2.50 - \left[\frac{f(2.5)}{f'(2.5)}\right] = 2.25.$$

$$x_3 = 2.25 - \left[\frac{f(2.25)}{f'(2.25)}\right] = 2.125.$$

Using Modified Newton-Raphson:

$$x_0 = 3.0$$

$$x_1 = 3.0 - \left[\frac{f(3.0)f'(3.0)}{(f'(3.0))^2 - f(3.0)f''(3.0)}\right]$$

$$= 3.0 - \left[\frac{2}{2}\right] = 2.0. \quad \text{Exact answer in one step!}$$

Modified Newton-Raphson Iteration

Test Program Source Code:

```
# ============================================================================
# TestModifiedNewtonRaphson01.py: Use modified newton raphson algorithm to
# compute solutions to equations having double roots.
#
# Written By: Mark Austin                                        February 2023
# ============================================================================

import math;
import Solutions;

# Mathematical functions: (x-2)*(x-2) = 0 ...

def f1(x):
    return (x-2)*(x-2);

def df1(x):
    return 2*(x-2);

def ddf1(x):
    return 2;

# main method ...

def main():
    print("--- Enter TestModifiedNewtonRaphson01.main()  ... ");
    print("--- ========================================= ... ");
```

Modified Newton-Raphson Iteration

Test Program Source Code: Continued ...

```

    print("--- ");
    print("--- Case Study 1: Solve (x-2)*(x-2) = 0, Initial guess: x0 = 3      ... ");
    print("--- ============================================================ ... ");

    # Initialize problem setup ...

    x0 = 3.0;
    tolerance     = 0.001
    maxiterations = 100

    print("--- Inputs:")
    print("---   x0 = {:5.2f} ...".format(x0) )
    print("---   tolerance      = {:8.5f} ...".format(tolerance) )
    print("---   max iterations = {:8.2f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.modifiednewtonraphson(f1, df1, ddf1, x0, tolerance, m

    # Summary of computations ...

    print("--- Output:")
    print("---   root = {:10.5f} ...".format(root) )
    print("---   f(root)   --> {:12.5e} ...".format(   f1(root)) )
```

Modified Newton-Raphson Iteration

Test Program Source Code: Continued ...

```
    print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
    print("---   ddf(root) --> {:16.8e} ...".format( ddf1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

    print("--- ");
    print("--- Case Study 2: Solve (x-2)*(x-2) = 0, Initial guess: x0 = -3   ... ");
    print("--- ============================================================= ... ");

    # Initialize problem setup ...

    x0 = -3.0;
    tolerance     = 0.001
    maxiterations = 100

    ... lines of source code removed ...
    ... details are identical to case study 1 ...

    print("--- ======================================== ... ");
    print("--- Leave TestModifiedNewtonRaphson01.main()  ... ");

# call the main method ...

main()
```

Modified Newton-Raphson Iteration

Abbreviated Output: Case Study 1: Initial guess: x0 = 3

```
--- Inputs:
---   x0 =  3.00 ...
---   tolerance      =  0.00100 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---     x0     -->  3.00000e+00 ...
---     f(x0)  -->  1.00000e+00 ...
---   Main Loop for Modified Newton Raphson Iteration:
---   Iteration 01: dx = -1.00000e+00, x = 2.00000e+00, f(x) -> 0.00000e+00
--- Output:
---   root =    2.00000 ...
---   f(root)    -->   0.00000000e+00 ...
---   df(root)   -->   0.00000000e+00 ...
---   ddf(root) -->   2.00000000e+00 ...
---   no iterations = 1 ...
---   converged: True ...
```

Modified Newton-Raphson Iteration

Abbreviated Output: Case Study 2: Initial guess: x0 = -3

```
--- Inputs:
---   x0 = -3.00 ...
---   tolerance      =  0.00100 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---     x0     --> -3.00000e+00 ...
---     f(x0)  -->  2.50000e+01 ...
---   Main Loop for Modified Newton Raphson Iteration:
---   Iteration 01: dx = 5.00000e+00, x = 2.00000e+00, f(x) -> 0.00000e+00
--- Output:
---   root =    2.00000 ...
---   f(root)    -->   0.00000000e+00 ...
---   df(root)   -->   0.00000000e+00 ...
---   ddf(root) -->   2.00000000e+00 ...
---   no iterations = 1 ...
---   converged: True ...
```

Python Code Listings

Code 1: Method of Bisection

```
# ==========================================================================
# Solutions.bisection(): Compute Roots of an equation by the Bisection method.
#
# Args: f (function): equation f(x).
#       a (float): lower limit.
#       b (float): upper limit.
#       toler (float): tolerance (stopping criterion).
#       iter_max (int): maximum number of iterations (stopping criterion).
#
# Returns:
#       root (float): root value.
#       iter (int): number of iterations used by the method.
#       converged (boolean): flag to indicate if the root was found.
# ==========================================================================

import math

def bisection(f, a, b, toler, iter_max):

    fa = f(a)
    fb = f(b)

    # Check that the function changes sign ....

    print("---   Initial Conditions: ")
    print("---   f(a) --> {:12.5e} ...".format( f(a) ) );
    print("---   f(b) --> {:12.5e} ...".format( f(b) ) );
```

Code 1: Method of Bisection

```
    if fa * fb > 0:
        raise ValueError("--- The function does not change signal at \
              the ends of the given interval.")

    delta_x = math.fabs(b - a) / 2

    # Main loop for bisection iteration ..

    print("---   Main Loop for Root Computation: ")

    x = 0
    converged = False
    for i in range(0, iter_max + 1):
        x   = (a + b) / 2
        fx = f(x)

        print("---   Iteration {:03d}: dx = {:10.5e}, x = {:14.7e}, f(x) --> {:14.7e} ..

        if delta_x <= toler and math.fabs(fx) <= toler:
            converged = True
            break

        if fa * fx > 0:
            a = x
            fa = fx
        else:
            b = x

        delta_x = delta_x / 2
```

Code 2: Newton Raphson Algorithm

```
# ==============================================================================
# Calculate the root of an equation by the Newton Raphson method.
#
# Args: f (function): equation f(x).
#       df (function): derivative of quation f(x).
#       x0 (float): initial guess.
#       toler (float): tolerance (stopping criterion).
#       iter_max (int): maximum number of iterations (stopping criterion).
#
# Returns:
#       root (float): root value.
#       iter (int): number of iterations used by the method.
#       converged (boolean): flag to indicate if the root was found.
# ==============================================================================

import math

def newtonraphson(f, df, x0, toler, iter_max):

    fx  = f(x0)
    dfx = df(x0)
    x   = x0

    print("---   Initial Conditions: ")
    print("---     x0     --> {:12.5e} ...".format( x0 ) );
    print("---     f(x0)  --> {:12.5e} ...".format( f(x0) ) );
    print("---     df(x0) --> {:12.5e} ...".format( df(x0) ) );
```

Code 2: Newton Raphson Algorithm

```
    print("---   Main Loop for Newton Raphson Iteration: ")

    converged = False
    for i in range(1, iter_max + 1):

        # Compute update to root estimate ...

        delta_x = -fx / dfx
        x += delta_x
        fx  = f(x)
        dfx = df(x)

        print("---   Iteration {:03d}: dx = {:12.5e}, x = {:12.5e}, f(x) --> {:12.5e} ..

        # Check for convergence ...

        if math.fabs(delta_x) <= toler and math.fabs(fx) <= toler or dfx == 0:
            converged = True
            break

    root = x
    return root, i, converged
```

Code 3: Modified Newton Raphson

```
# ==========================================================================
# Calculate the root of an equation by the Modified Newton Raphson method.
#
# Args: f (function): equation f(x).
#       df (function): derivative of f(x).
#       ddf (function): second derivative of f(x).
#       x0 (float): initial guess.
#       toler (float): tolerance (stopping criterion).
#       iter_max (int): maximum number of iterations (stopping criterion).
#
# Returns:
#       root (float): root value.
#       iter (int): number of iterations used by the method.
#       converged (boolean): flag to indicate if the root was found.
# ==========================================================================

import math

def modifiednewtonraphson(f, df, ddf, x0, toler, iter_max):

    fx   = f(x0)
    dfx  = df(x0)
    ddfx = ddf(x0)
    x = x0

    print("---   Initial Conditions: ")
    print("---     x0     --> {:12.5e} ...".format( x0 ) );
    print("---     f(x0)  --> {:12.5e} ...".format( f(x0) ) );
```

Code 3: Modified Newton Raphson

```
    print("---   Main Loop for Modified Newton Raphson Iteration: ")

    converged = False
    for i in range(1, iter_max + 1):

        # Compute update to root estimate ...

        delta_x = -((fx*dfx)/(dfx*dfx - fx*ddfx))
        x       = x + delta_x
        fx   = f(x)
        dfx  = df(x)
        ddfx = ddf(x)

        print("---   Iteration {:03d}: dx = {:12.5e}, x = {:12.5e}, f(x) --> {:12.5e} ..

        # Check for convergence ...

        if math.fabs(delta_x) <= toler and math.fabs(fx) <= toler or dfx == 0:
            converged = True
            break

    root = x
    return root, i, converged
```