

Roots of Equations

Mark A. Austin

University of Maryland

austin@umd.edu

ENCE 201, Fall Semester 2025

October 20, 2025

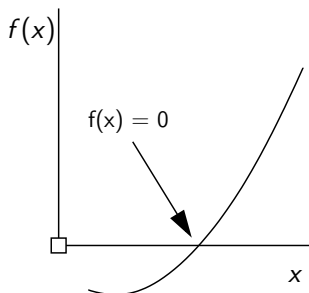
Overview

- 1 Numerical Solution of Equations
- 2 Iterative Methods
- 3 Method of Bisection
 - Numerical Procedure, Examples
- 4 Newton Raphson Iteration
- 5 Modified Newton Raphson Iteration
- 6 Python Code Listings
 - Method of Bisection
 - Newton Raphson Algorithm
 - Modified Newton Raphson

Numerical Solution of Equations

Numerical Solution of Equations

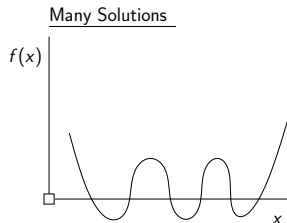
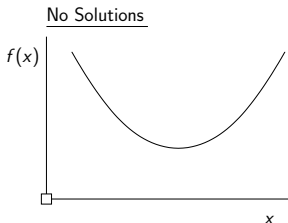
Math Problem. Given $f(x)$, find a value of x such that $f(x) = g(x)$, $f(x) = \text{constant}$, or $f(x) = 0$.



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:

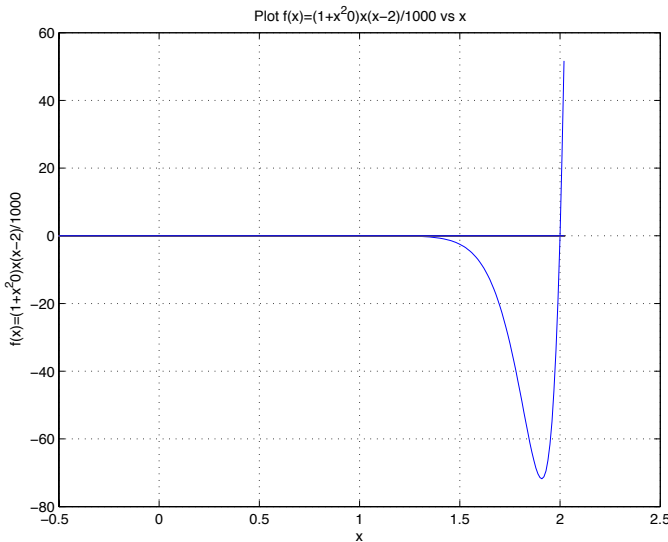
$$f(x) = \left[\frac{(x^{20} + 1)x(x - 2)}{1000} \right] \quad (1)$$

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	-2.31×10^{-4}
2.001	2.1200
0.000	0.0000
2.000	0.0000

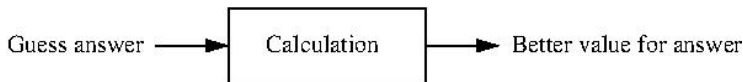
Numerical Solution of Equations



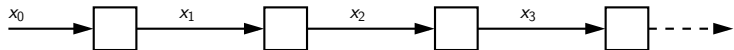
Iterative Methods

Iterative Methods

Procedure. Solve problem through a sequence of approximations:



Apply process iteratively:



Ideally, x_0, x_1, \dots, 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:

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

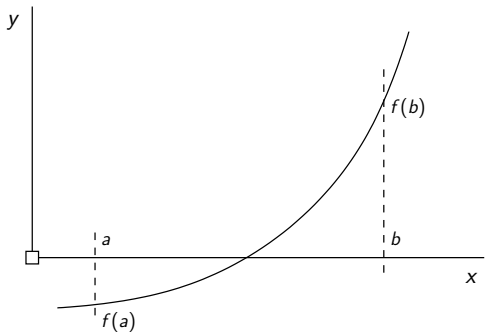
Let $A = 4$. Use initial guess $x_1 = 1 \approx \sqrt{4}$.

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

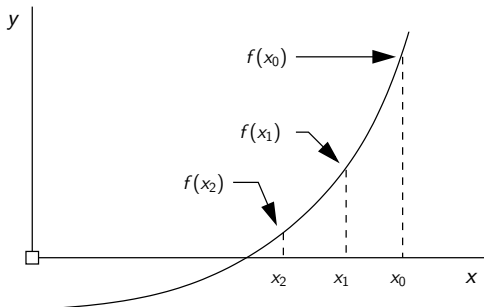
Bracketing Methods: Requires two initial guesses that bracket the solution.



- 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.

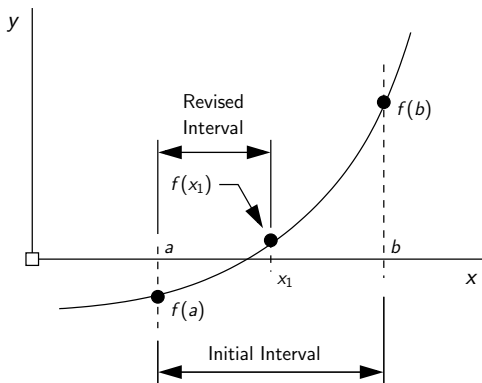
Keep $f(x_{left})$ if $f(x_{new}) \cdot f(x_{left}) < 0$.

Otherwise, keep $f(x_{right})$ if $f(x_{new}) \cdot f(x_{right}) < 0$.

- 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.

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

Analytic Solution: From equation 3:

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

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:

```
1 # =====
2 # TestBisection01.py: Use bisection algorithm to compute roots of equations.
3 #
4 # Written By: Mark Austin February 2023
5 # =====
6
7 import math;
8 import Solutions;
9
10 # Define mathematical functions ...
11
12 def f1(x):
13     return (x-3)*(x-3)-2;
14
15 # main method ...
16
17 def main():
18     print("--- Enter TestBisection01.main() ... ");
19     print("--- ===== ... ");
20
21     print("--- ");
22     print("--- Case Study 1: Solve (x-3)*(x-3)-2 = 0 ... ");
23     print("--- ===== ... ");
24
25     # Initialize problem setup ...
```

Method of Bisection

Test Program Source Code: Continued ...

```
27     a = -1.0;
28     b = 2.0
29     tolerance      = 0.01
30     maxiterations  = 100
31
32     print("--- Inputs:")
33     print("---   a = {:5.2f} ...".format(a) )
34     print("---   b = {:5.2f} ...".format(b) )
35     print("---   tolerance      = {:8.5f} ...".format(tolerance) )
36     print("---   max iterations = {:8.2f} ...".format(maxiterations) )
37
38     # Compute roots to equation ...
39
40     print("--- Execution:")
41     root, i, converged = Solutions.bisection(f1, a, b, tolerance, maxiterations )
42
43     # Summary of computations ...
44
45     print("--- Output:")
46     print("---   root = {:10.5f} ...".format(root) )
47     print("---   f(root) --> {:12.5e} ...".format( f1(root) ) )
48     print("---   no iterations = {:d} ...".format(i) )
49     print("---   converged: {:s} ...".format( str(converged) ) )
50
51     print("--- ");
52     print("--- Case Study 2: Solve  $2x^3 - \cos(x+1) - 3 = 0$  ... ");
53     print("--- ===== ... ");
```

Method of Bisection

Test Program Source Code: Continued ...

```
54
55     # Initialize problem setup ...
56
57     a = -1.0;
58     b = 2.0
59     tolerance      = 0.01
60     maxiterations  = 100
61
62     print("--- Inputs:")
63     print("--- a = {:5.2f} ...".format(a) )
64     print("--- b = {:5.2f} ...".format(b) )
65     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

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

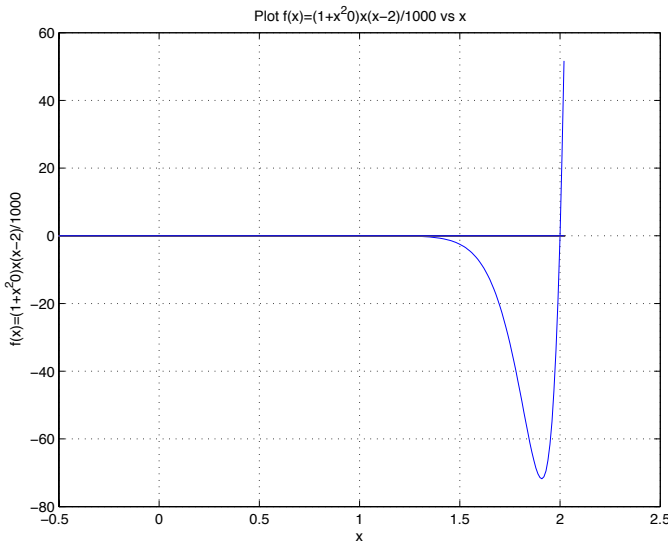
has two roots within the interval $[-1, 3]$.

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:

```
1 # =====
2 # TestBisection02.py: Use bisection algorithm to compute roots of equations:
3 #
4 # Written By: Mark Austin February 2023
5 # =====
6
7 import math;
8 import Solutions;
9
10 # Define mathematical functions ...
11
12 def f1(x):
13     return (x**20 + 1)*x*(x-2)/1000.0;
14
15 # main method ...
16
17 def main():
18     print("--- ");
19     print("--- Case Study 1: Solve f(x) = ((x^20 + 1)x(x-2))/1000 = 0 ... ");
20     print("--- ===== ... ");
21
22     # Initialize problem setup ...
23
24     a = 0.5;
25     b = 2.5
26     tolerance = 0.0001
```

Method of Bisection

Test Program Source Code: Continued ...

```
27     maxiterations = 100
28
29     print("--- Inputs:")
30     print("---   a = {:.2f} ...".format(a) )
31     print("---   b = {:.2f} ...".format(b) )
32     print("---   tolerance       = {:.5f} ...".format(tolerance) )
33     print("---   max iterations = {:.2f} ...".format(maxiterations) )
34
35     # Compute roots to equation ...
36
37     print("--- Execution:")
38     root, i, converged = Solutions.bisection(f1, a, b, tolerance, maxiterations )
39
40     # Summary of computations ...
41
42     print("--- Output:")
43     print("---   root = {:.2f} ...".format(root) )
44     print("---   f(root) --> {:.4e} ...".format( f1(root) ) )
45     print("---   no iterations = {d} ...".format(i) )
46     print("---   converged: {s} ...".format( str(converged) ) )
47
48     # call the main method ...
49
50     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:

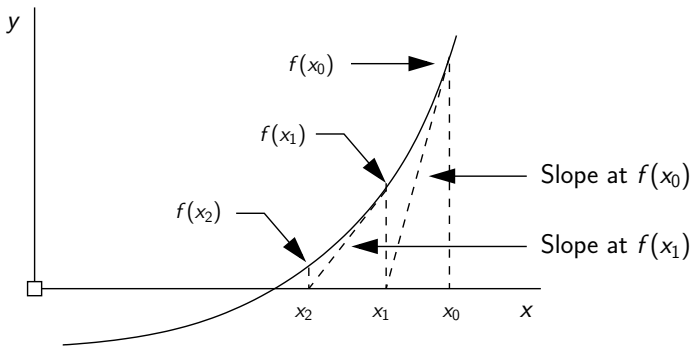
$$f(x_1) = f(x_0) + hf'(x_0) + O(h^2) = 0.0 \quad (6)$$

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

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

Newton-Raphson Iteration

Schematic: Two iterations of Newton-Raphson.



Sequence of estimates is: x_0, x_1, x_3, \dots

Newton-Raphson Iteration

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

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

Differentiating,

$$f'(x) = \sin(x) + x\cos(x) + 3\sin(x) \quad (9)$$

The Newton-Raphson update is:

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

This gives: $x_0 = 0.8$, $x_1 = 1.24$, $x_2 = 1.1927 \dots$

Newton-Raphson Iteration

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

$$f(x) = (x - 3) * (x - 3) - 2; \quad (11)$$

The derivative is given by:

$$df(x)/dx = 2x - 6. \quad (12)$$

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

```
1 # =====
2 # TestNewtonRaphson01.py: Use newton raphson algorithm to compute roots of
3 # equations.
4 #
5 # Written By: Mark Austin February 2023
6 # =====
7
8 import math;
9 import Solutions;
10
11 # Mathematical functions:  $(x-3)*(x-3) - 2 = 0$  ...
12
13 def f1(x):
14     return (x-3)*(x-3)-2;
15
16 def df1(x):
17     return 2*(x-3);
18
19 # main method ...
20
21 def main():
22     print("--- Enter TestNewtonRaphson01.main() ... ");
23     print("--- ===== ... ");
24
25     print("--- ");
26     print("--- Case Study 1: Solve  $(x-3)*(x-3)-2 = 0$ , Initial guess:  $x_0 = -10$  ... ");
27     print("--- ===== ... ");
```

Program Source Code

```
29      # Initialize problem setup ...
30
31      x0 = -10.0;
32      tolerance      = 0.001
33      maxiterations  = 100
34
35      print("--- Inputs:")
36      print("---- x0 = {:.52f} ...".format(x0) )
37      print("---- tolerance      = {:.85f} ...".format(tolerance) )
38      print("---- max iterations = {:.82f} ...".format(maxiterations) )
39
40      # Compute roots to equation ...
41
42      print("---- Execution:")
43      root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )
44
45      # Summary of computations ...
46
47      print("---- Output:")
48      print("---- root = {:.105f} ...".format(root) )
49      print("---- f(root) --> {:.125e} ...".format( f1(root) ) )
50      print("---- no iterations = {:d} ...".format(i) )
51      print("---- converged: {:s} ...".format( str(converged) ) )
52
53      print("---- ");
54      print("---- Case Study 2: Solve (x-3)*(x-3)-2 = 0, Initial guess: x0 = 10 ... ");
55      print("---- ===== ... ");
56
57      # Initialize problem setup ...
```

Program Source Code

```
59     x0 = 10.0;
60     tolerance      = 0.001
61     maxiterations  = 100
62
63     print("--- Inputs:")
64     print("---     x0 = {:5.2f} ...".format(x0) )
65     print("---     tolerance      = {:8.5f} ...".format(tolerance) )
66     print("---     max iterations = {:8.2f} ...".format(maxiterations) )
67
68     # Compute roots to equation ...
69
70     print("--- Execution:")
71     root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )
72
73     # Summary of computations ...
74
75     print("--- Output:")
76     print("---     root = {:10.5f} ...".format(root) )
77     print("---     f(root) --> {:12.5e} ...".format( f1(root)) )
78     print("---     no iterations = {:d} ...".format(i) )
79     print("---     converged: {:s} ...".format( str(converged) ) )
80
81     print("--- ===== ... ");
82     print("--- Leave TestNewtonRaphson01.main()           ... ");
83
84     # call the main method ...
```

Newton-Raphson Iteration

Abbreviated Output: Case Study 1, $x_0 = -10$.

```
--- 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

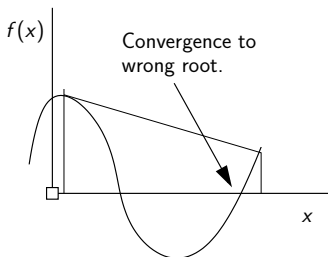
Abbreviated Output: Case Study 2, $x_0 = 10$.

```
--- 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 ...
```

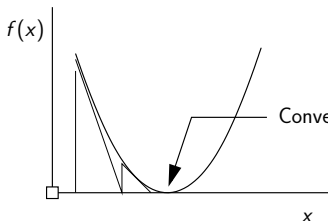
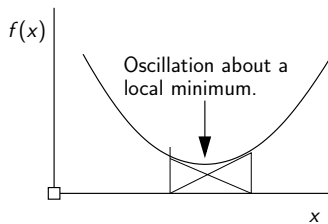
Modified Newton Raphson Iteration

Limitations of Newton-Raphson Iteration

No Solutions



Many Solutions



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:

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

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

Differentiating,

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

Modified Newton-Raphson Iteration

Substituting into N-R formula:

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

Example 1. The function

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

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:

$$x_0 = 3.0$$

$$\begin{aligned} x_1 &= 3.0 - \left[\frac{f(3.0)f'(3.0)}{(f'(3.0))^2 - f(3.0)f''(3.0)} \right] \\ &= 3.0 - \left[\frac{2}{2} \right] = 2.0. \quad \text{Exact answer in one step!} \end{aligned}$$

Modified Newton-Raphson Iteration

Test Program Source Code:

```
1 # =====
2 # TestModifiedNewtonRaphson01.py: Use modified newton raphson algorithm to
3 # compute solutions to equations having double roots.
4 #
5 # Written By: Mark Austin February 2023
6 # =====
7
8 import math;
9 import Solutions;
10
11 # Mathematical functions: (x-2)*(x-2) = 0 ...
12
13 def f1(x):
14     return (x-2)*(x-2);
15
16 def df1(x):
17     return 2*(x-2);
18
19 def ddf1(x):
20     return 2;
21
22 # main method ...
23
24 def main():
25     print("--- Enter TestModifiedNewtonRaphson01.main() ... ");
26     print("--- ===== ... ");
```

Modified Newton-Raphson Iteration

Test Program Source Code: Continued ...

```
27
28     print("--- ");
29     print("--- Case Study 1: Solve (x-2)*(x-2) = 0, Initial guess: x0 = 3      ... ");
30     print("--- ===== ... ");
31
32     # Initialize problem setup ...
33
34     x0 = 3.0;
35     tolerance = 0.001
36     maxiterations = 100
37
38     print("--- Inputs:")
39     print("---     x0 = {:.5.2f} ...".format(x0) )
40     print("---     tolerance = {:.8.5f} ...".format(tolerance) )
41     print("---     max iterations = {:.8.2f} ...".format(maxiterations) )
42
43     # Compute roots to equation ...
44
45     print("--- Execution:")
46     root, i, converged = Solutions.modifiednewtonraphson(f1, df1, ddf1, x0, tolerance, m
47
48     # Summary of computations ...
49
50     print("--- Output:")
51     print("---     root = {:.10.5f} ...".format(root) )
52     print("---     f(root) --> {:.12.5e} ...".format( f1(root)) )
```

Modified Newton-Raphson Iteration

Test Program Source Code: Continued ...

```
53     print("--- df(root) --> {:16.8e} ...".format( df1(root)) )
54     print("--- ddf(root) --> {:16.8e} ...".format( ddf1(root)) )
55     print("--- no iterations = {:d} ...".format(i) )
56     print("--- converged: {:s} ...".format( str(converged) ) )
57
58     print("--- ");
59     print("--- Case Study 2: Solve (x-2)*(x-2) = 0, Initial guess: x0 = -3    ... ");
60     print("--- ===== ... ");
61
62     # Initialize problem setup ...
63
64     x0 = -3.0;
65     tolerance      = 0.001
66     maxiterations  = 100
67
68     ... lines of source code removed ...
69     ... details are identical to case study 1 ...
70
71     print("--- ===== ... ");
72     print("--- Leave TestModifiedNewtonRaphson01.main() ... ");
73
74     # call the main method ...
75
76     main()
```

Modified Newton-Raphson Iteration

Abbreviated Output: Case Study 1: Initial guess: $x_0 = 3$

```
--- 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: $x_0 = -3$

```
--- 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

```
1  # =====
2  # Solutions.bisection(): Compute Roots of an equation by the Bisection method.
3  #
4  # Args: f (function): equation f(x).
5  #       a (float): lower limit.
6  #       b (float): upper limit.
7  #       toler (float): tolerance (stopping criterion).
8  #       iter_max (int): maximum number of iterations (stopping criterion).
9  #
10 # Returns:
11 #       root (float): root value.
12 #       iter (int): number of iterations used by the method.
13 #       converged (boolean): flag to indicate if the root was found.
14 # =====
15
16 import math
17
18 def bisection(f, a, b, toler, iter_max):
19
20     fa = f(a)
21     fb = f(b)
22
23     # Check that the function changes sign ...
24
25     print("--- Initial Conditions: ")
26     print("--- f(a) --> {:.12.5e} ...".format( f(a) ) );
27     print("--- f(b) --> {:.12.5e} ...".format( f(b) ) );
```

Code 1: Method of Bisection

```
29     if fa * fb > 0:
30         raise ValueError("--- The function does not change signal at \
31             the ends of the given interval.")
32
33     delta_x = math.fabs(b - a) / 2
34
35     # Main loop for bisection iteration ..
36
37     print("---    Main Loop for Root Computation: ")
38
39     x = 0
40     converged = False
41     for i in range(0, iter_max + 1):
42         x = (a + b) / 2
43         fx = f(x)
44
45         print("---    Iteration {:03d}: dx = {:10.5e}, x = {:14.7e}, f(x) --> {:14.7e} ..
46
47         if delta_x <= toler and math.fabs(fx) <= toler:
48             converged = True
49             break
50
51         if fa * fx > 0:
52             a = x
53             fa = fx
54         else:
55             b = x
56
57     delta_x = delta_x / 2
```

Code 2: Newton Raphson Algorithm

```
1 # =====
2 # Calculate the root of an equation by the Newton Raphson method.
3 #
4 # Args: f (function): equation f(x).
5 #       df (function): derivative of quation f(x).
6 #       x0 (float): initial guess.
7 #       toler (float): tolerance (stopping criterion).
8 #       iter_max (int): maximum number of iterations (stopping criterion).
9 #
10 # Returns:
11 #         root (float): root value.
12 #         iter (int): number of iterations used by the method.
13 #         converged (boolean): flag to indicate if the root was found.
14 # =====
15
16 import math
17
18 def newtonraphson(f, df, x0, toler, iter_max):
19
20     fx = f(x0)
21     dfx = df(x0)
22     x = x0
23
24     print("--- Initial Conditions: ")
25     print("--- x0 --> {:.125e} ...".format( x0 ) );
26     print("--- f(x0) --> {:.125e} ...".format( f(x0) ) );
27     print("--- df(x0) --> {:.125e} ...".format( df(x0) ) );
```

Code 2: Newton Raphson Algorithm

```
29     print("---    Main Loop for Newton Raphson Iteration: ")
30
31     converged = False
32     for i in range(1, iter_max + 1):
33
34         # Compute update to root estimate ...
35
36         delta_x = -fx / dfx
37         x += delta_x
38         fx = f(x)
39         dfx = df(x)
40
41         print("---    Iteration {:03d}: dx = {:12.5e}, x = {:12.5e}, f(x) --> {:12.5e} ..
42
43         # Check for convergence ...
44
45         if math.fabs(delta_x) <= toler and math.fabs(fx) <= toler or dfx == 0:
46             converged = True
47             break
48
49     root = x
50     return root, i, converged
```

Code 3: Modified Newton Raphson

```
1  # =====
2  # Calculate the root of an equation by the Modified Newton Raphson method.
3  #
4  # Args: f (function): equation f(x).
5  #       df (function): derivative of f(x).
6  #       ddf (function): second derivative of f(x).
7  #       x0 (float): initial guess.
8  #       toler (float): tolerance (stopping criterion).
9  #       iter_max (int): maximum number of iterations (stopping criterion).
10 #
11 # Returns:
12 #       root (float): root value.
13 #       iter (int): number of iterations used by the method.
14 #       converged (boolean): flag to indicate if the root was found.
15 # =====
16
17 import math
18
19 def modifiednewtonraphson(f, df, ddf, x0, toler, iter_max):
20
21     fx = f(x0)
22     dfx = df(x0)
23     ddfx = ddf(x0)
24     x = x0
25
26     print("--- Initial Conditions: ")
27     print("--- x0 --> {:.125e} ...".format( x0 ) );
28     print("--- f(x0) --> {:.125e} ...".format( f(x0) ) );
```

Code 3: Modified Newton Raphson

```
29
30 print("--- Main Loop for Modified Newton Raphson Iteration: ")
31
32 converged = False
33 for i in range(1, iter_max + 1):
34
35     # Compute update to root estimate ...
36
37     delta_x = -((fx*dfx)/(dfx*dfx - fx*ddf))
38     x      = x + delta_x
39     fx     = f(x)
40     dfx    = df(x)
41     ddfx   = ddf(x)
42
43     print("--- Iteration {:03d}: dx = {:12.5e}, x = {:12.5e}, f(x) --> {:12.5e} ..
44
45     # Check for convergence ...
46
47     if math.fabs(delta_x) <= toler and math.fabs(fx) <= toler or dfx == 0:
48         converged = True
49         break
50
51 root = x
52 return root, i, converged
```