

NUMERICAL METHODS

In this report, I tried to investigate some of the iterative methods used in finding solutions to the scalar, nonlinear equations. Since every method has its own strengths and weaknesses most of the time it is hard to choose the most appropriate one to solve the problem. For this reason, I tried to come up with a road map to help finding the correct method. Every method is tested with different equations in order to see their capabilities clearly.

Methods covered in this report:

1. Bisection Method
2. Newton's Method
3. Secant Method
4. Fixed Point Iteration

1 Bisection Method

Bisection method is a closed method used in finding roots. An interval which includes the root is picked ($[a, b]$). This interval is narrowed so that the value of the function changes from plus (+) to minus (-) or minus (-) to plus (+). By continuing this process, the result is approached. You can see the code I have written in order to use this method and generate results below.

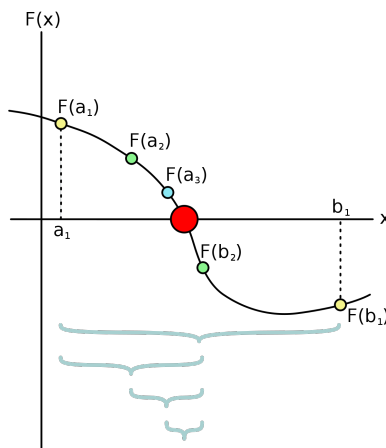


Figure 1: Image from <https://www.wikipedia.org/>

```
1 from scipy import poly1d
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import time
5 import math
6 import plotly.graph_objects as go
7
8 # a = lower bound of the iteration
9 # b = upper bound of the iteration
10 # f = non-linear equation
11 # error = absolute tolerance
12 # n = number of iterations
13 # polynom = f(x)
14
15 def bisection(a,b,f,error):
16
17     print(f)
18     print("Starting interval:[" ,a, " , " ,b,"]")
19     print("Error tolerance:",error)
20
21     #plotting the fuction graph
22     pol_x = np.arange(0.5,3.5,0.01)
23     pol_y = polynom(pol_x)
24     plt.plot(pol_x, pol_y, 'g' , label = "f(x)")
25     plt.xlabel('x - axis')
26     plt.ylabel('y - axis')
27     plt.title('Bisection Method')
28     plt.grid(True)
29
30     x = []
31     y = []
32
33     n = 1
34     if f(a)*f(b) >= 0:
35         print("These interval points can't be used")
36         return None
37
38     while((b-a)/(2**n) > error):
39
40         mean = (a+b)/2
41
42         #saving the iterations
43         x.append(mean)
44         y.append(f(mean))
45
46         if f(mean)*f(a) < 0:
```

```

47         b = mean
48     elif f(mean)*f(b) < 0:
49         a = mean
50     elif f(mean) == 0:
51         print("Found exact solution")
52         return f(mean)
53     else:
54         print("Method failed")
55         return None
56
57     n += 1
58
59     print("Numer of iterations :", n)
60
61     #marking the iterations
62     plt.plot(x, y,color='blue',linestyle='dashed', linewidth = 1, marker= ↵
        "o" , markerfacecolor='blue', markersize=5, label = "iteration ↵
        points")
63     plt.legend()
64     return (a+b)/2
65
66 t = time.process_time()
67
68 #generating a polynomial function
69 polynom = poly1d([1,-6,11,-6])
70
71 print("Root found:",bisection(1.5,2.25,polynom,10**-3))
72
73 #time measure
74 elapsed_time = time.process_time() - t
75
76 print("Elapsed time:",elapsed_time)

```

You can see the outputs of the code using different scenarios below.

Two different equations used:

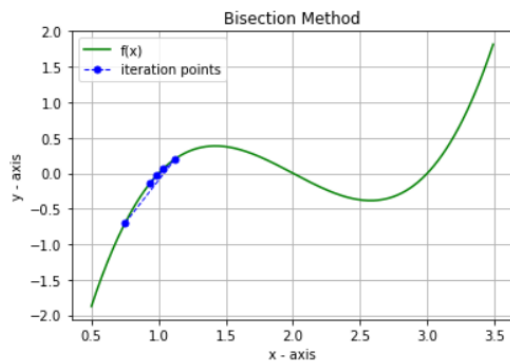
$$f(x) : x^3 - 6x^2 + 11x - 6 \quad (1)$$

$$g(x) : -2x^4 + 2x^3 - 4x^2 - 10x + 200 \quad (2)$$

```

      3      2
1 x - 6 x + 11 x - 6
Starting interval:[ 0 , 1.5 ]
Error tolerance: 0.001
Nuner of iterations : 6
Root found: 1.0078125
Elapsed time: 0.03125

```

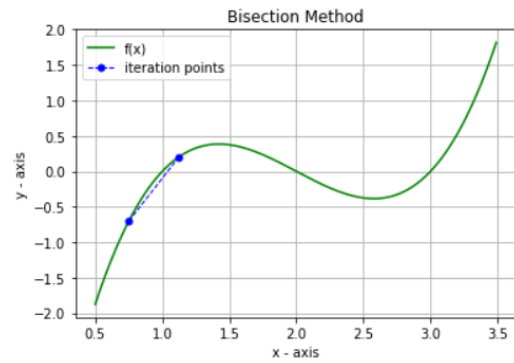


(a) $f(x)$

```

      3      2
1 x - 6 x + 11 x - 6
Starting interval:[ 0 , 1.5 ]
Error tolerance: 0.1
Nuner of iterations : 3
Root found: 0.9375
Elapsed time: 0.03125

```

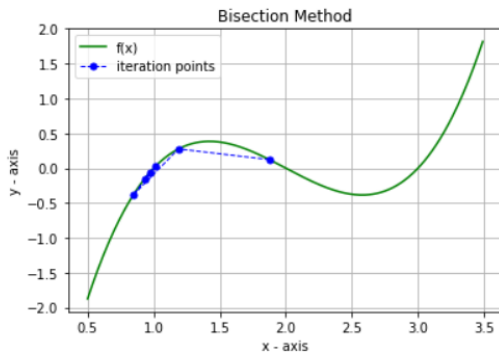


(b) $f(x)$ with different tolerance

```

      3      2
1 x - 6 x + 11 x - 6
Starting interval:[ 0.5 , 3.25 ]
Error tolerance: 0.001
Nuner of iterations : 7
Root found: 0.994140625
Elapsed time: 0.015625

```

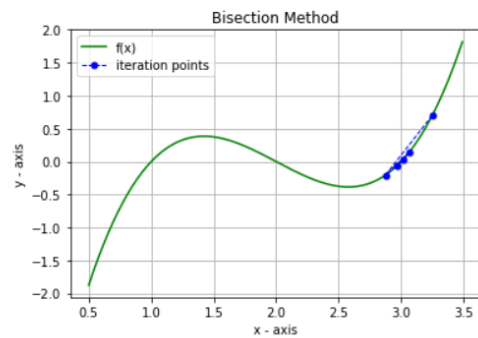


(c) $f(x)$ with different interval

```

      3      2
1 x - 6 x + 11 x - 6
Starting interval:[ 2.5 , 4 ]
Error tolerance: 0.001
Nuner of iterations : 6
Root found: 2.9921875
Elapsed time: 0.03125

```



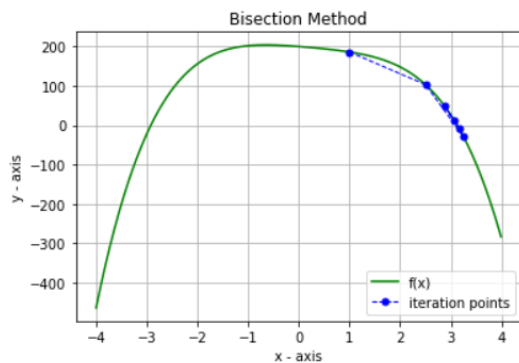
(d) $f(x)$ with different interval

Figure 2: Bisection method used on $f(x)$

```

      4      3      2
-2 x + 2 x - 4 x - 10 x + 200
Starting interval:[ -2 , 4 ]
Error tolerance: 0.001
Nuner of iterations : 7
Root found: 3.109375
Elapsed time: 0.015625

```

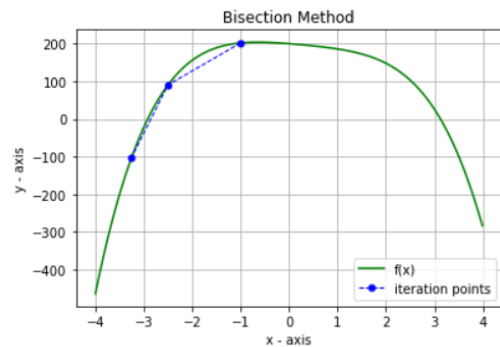


(a) $g(x)$

```

      4      3      2
-2 x + 2 x - 4 x - 10 x + 200
Starting interval:[ -4 , 2 ]
Error tolerance: 0.1
Nuner of iterations : 4
Root found: -2.875
Elapsed time: 0.015625

```

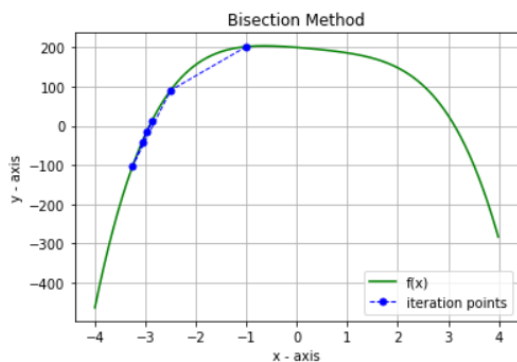


(b) $g(x)$ with different tolerance

```

      4      3      2
-2 x + 2 x - 4 x - 10 x + 200
Starting interval:[ -4 , 2 ]
Error tolerance: 0.001
Nuner of iterations : 7
Root found: -2.921875
Elapsed time: 0.015625

```

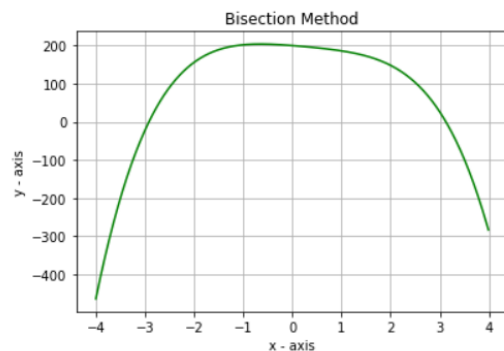


(c) $g(x)$ with different interval

```

      4      3      2
-2 x + 2 x - 4 x - 10 x + 200
Starting interval:[ -2 , 3 ]
Error tolerance: 0.1
These interval points can't be used
Root found: None
Elapsed time: 0.015625

```



(d) $g(x)$ with different interval

Figure 3: Bisection method used on $g(x)$

By looking at these results:

Pros:

1. Always converges

Cons:

1. Slow convergence rate
2. Doesn't work with every starting interval (case **(d)** on **g(x)**)
3. In some cases it is hard to find every root (finding the root **x = 2** on **f(x)**)

2 Newton's method

Newton's method, also known as Newton-Raphson method is an open method used in finding roots. An initial starting point is picked. Starting this point, new points found by using the derivative of the function on the point. With iterating, actual root is obtained.

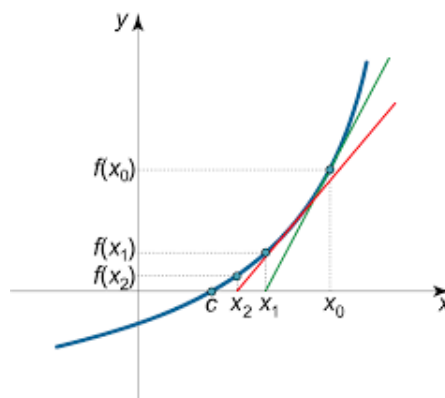


Figure 4: Image from <https://www.math24.net/>

You can see the code I have written in order to use this method and generate results below.

Listing 2: Python code – Newton method implemented

```
1 from scipy import poly1d
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import time
5 import math
6 import plotly.graph_objects as go
7
8 # a = initial point
9 # xn = next iteration point
10 # fx = next iteration value
```

```

11 # f = non-linear equation
12 # deriv = derivative of f
13 # error = absolute tolerance
14 # n = number of iterations
15
16 def newton(a,f,error):
17
18     print(f)
19     deriv = f.deriv()
20     print("Taking derivative:")
21     print(deriv)
22     print("Starting point:",a)
23     print("Error tolerance:",error)
24
25     #plotting the fuction graph
26     pol_x = np.arange(0,4,0.01)
27     pol_y = f(pol_x)
28     plt.plot(pol_x, pol_y, 'g' , label = "f(x)")
29     plt.xlabel('x - axis')
30     plt.ylabel('y - axis')
31     plt.title("Newton's Method")
32     plt.grid(True)
33
34     x = []
35     y = []
36
37     n = 1
38     xn = a
39
40     while(True):
41         fx = f(xn)
42
43         #saving the iterations
44         x.append(xn)
45         y.append(fx)
46         deriv_xn = deriv(xn)
47
48         if deriv_xn == 0:
49             print("No solution found")
50             return None
51
52         old = xn
53         xn = xn - (fx/deriv_xn)
54         err = abs(xn - old)
55         if (err < error):
56             break
57         n += 1

```

```

58
59
60     print("Numer of iterations :", n)
61
62     #marking the iterations
63     plt.plot(x, y,color='blue',linestyle='dashed', linewidth = 1, marker= ←
        "o" , markerfacecolor='blue', markersize=5, label = "iteration ←
        points")
64     plt.legend()
65     return xn
66
67 t = time.process_time()
68
69 #generating a polynomial function
70 polynom = poly1d([1,-6,11,-6])
71
72 print("Root found:",newton(0.5,polynom,10**-5))
73
74 #time measure
75 elapsed_time = time.process_time() - t
76
77 print("Elapsed time:",elapsed_time)

```

You can see the outputs of the code using different scenarios below.

Two different equations used:

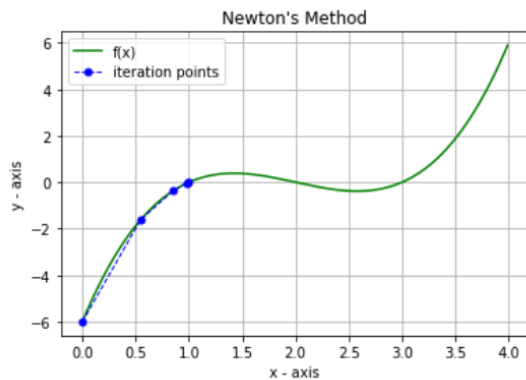
$$f(x) : x^3 - 6x^2 + 11x - 6 \quad (3)$$

$$g(x) : x^6 - x^5 - 6x^4 - x^2 + x + 10 \quad (4)$$


```

3      2
1 x - 6 x + 11 x - 6
Taking derivative:
2
3 x - 12 x + 11
Starting point: 0
Error tolerance: 0.001
Number of iterations : 5
Root found: 0.9999987646910548
Elapsed time: 0.03125

```

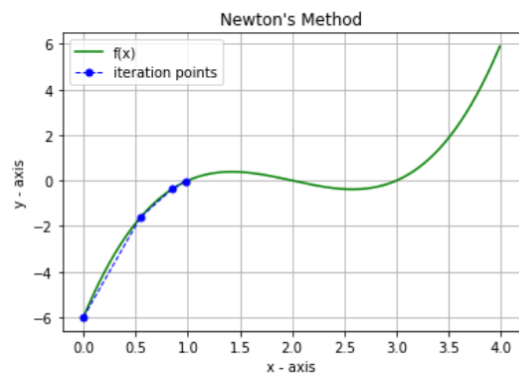


(a) $f(x)$

```

3      2
1 x - 6 x + 11 x - 6
Taking derivative:
2
3 x - 12 x + 11
Starting point: 0
Error tolerance: 0.1
Number of iterations : 4
Root found: 0.9990915480569487
Elapsed time: 0.03125

```

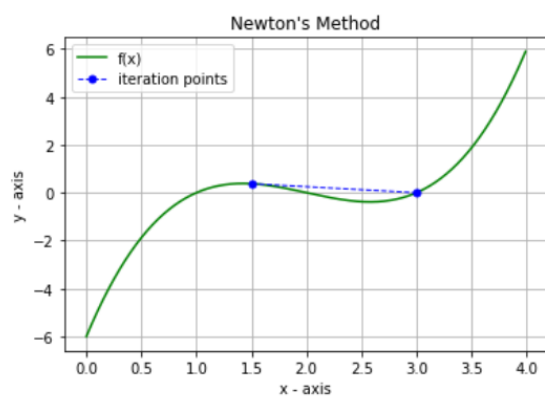


(b) $f(x)$ with different tolerance

```

3      2
1 x - 6 x + 11 x - 6
Taking derivative:
2
3 x - 12 x + 11
Starting point: 1.5
Error tolerance: 0.001
Number of iterations : 2
Root found: 3.0
Elapsed time: 0.015625

```

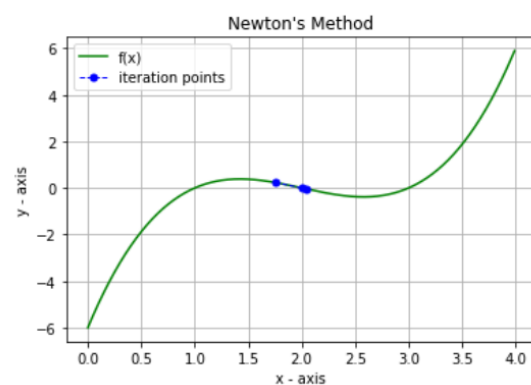


(c) $f(x)$ with different starting point

```

3      2
1 x - 6 x + 11 x - 6
Taking derivative:
2
3 x - 12 x + 11
Starting point: 1.75
Error tolerance: 0.001
Number of iterations : 3
Root found: 2.000000000029865
Elapsed time: 0.015625

```



(d) $f(x)$ with different starting point

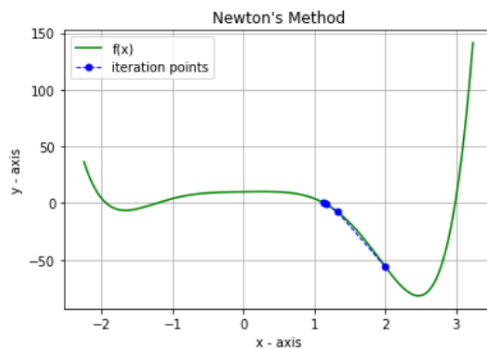
Figure 5: Newton's method used on $f(x)$

$$1x^6 - 1x^5 - 6x^4 - 1x^3 + 1x^2 + 10$$

Taking derivative:

$$6x^5 - 5x^4 - 24x^3 - 2x^2 + 1$$

Starting point: 2
 Error tolerance: 0.001
 Numer of iterations : 4
 Root found: 1.1392945911225982
 Elapsed time: 0.03125



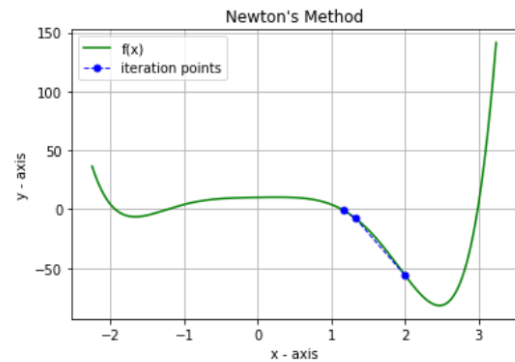
(a) $g(x)$

$$1x^6 - 1x^5 - 6x^4 - 1x^3 + 1x^2 + 10$$

Taking derivative:

$$6x^5 - 5x^4 - 24x^3 - 2x^2 + 1$$

Starting point: 2
 Error tolerance: 0.1
 Numer of iterations : 3
 Root found: 1.1401685608920944
 Elapsed time: 0.03125



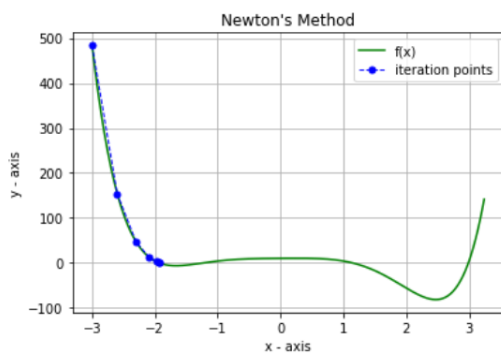
(b) $g(x)$ with different tolerance

$$1x^6 - 1x^5 - 6x^4 - 1x^3 + 1x^2 + 10$$

Taking derivative:

$$6x^5 - 5x^4 - 24x^3 - 2x^2 + 1$$

Starting point: -3
 Error tolerance: 0.001
 Numer of iterations : 7
 Root found: -1.9382288406304586
 Elapsed time: 0.015625



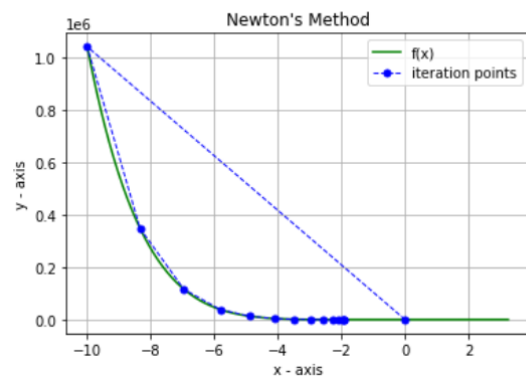
(c) $g(x)$ with different starting point

$$1x^6 - 1x^5 - 6x^4 - 1x^3 + 1x^2 + 10$$

Taking derivative:

$$6x^5 - 5x^4 - 24x^3 - 2x^2 + 1$$

Starting point: 0
 Error tolerance: 0.001
 Numer of iterations : 15
 Root found: -1.9382288375857275
 Elapsed time: 0.015625



(d) $g(x)$ with different starting point

Figure 6: Newton's method used on $g(x)$

By looking at these results:

Pros:

1. Less iterations on $f(x)$ compared to the **Bisection Method**
2. Requires one initial guess
3. Has quadratic convergence (the error is squared at each iteration)

Cons:

1. Depending on the initial point, it may diverge (case **(d)** on $g(x)$)
2. It requires additional data (derivative of the function)

3 Secant method

Secant method is an open method used in finding roots. Similar to the Newton's method, Secant method uses tangent lines. Since calculation of the some derivatives is hard, Secant method works with finite difference.

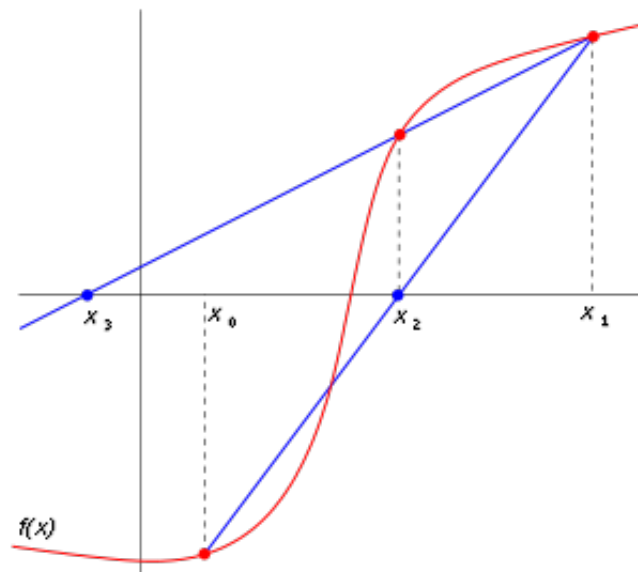


Figure 7: Image from <https://www.wikipedia.org/>

You can see the code I have written in order to use this method and generate results below.

Listing 3: Python code –Secant method implemented

```
1 from scipy import poly1d
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import time
```

```

5 import math
6 import plotly.graph_objects as go
7
8 # a = lower bound of the iteration
9 # b = upper bound of the iteration
10 # xn = next iteration point
11 # old = previous iteration to calculate error
12 # f = non-linear equation
13 # error = absolute tolerance
14 # n = number of iterations
15
16
17 def secant(a,b,f,error):
18
19     print(f)
20     print("Starting interval:[" ,a, " , " ,b,"]")
21     print("Error tolerance:",error)
22
23     #plotting the fuction graph
24     pol_x = np.arange(0,4,0.01)
25     pol_y = f(pol_x)
26     plt.plot(pol_x, pol_y, 'g' , label = "f(x)")
27     plt.xlabel('x - axis')
28     plt.ylabel('y - axis')
29     plt.title('Secant Method')
30     plt.grid(True)
31
32     x = []
33     y = []
34
35     n = 1
36
37     if f(a)*f(b) >= 0:
38         print("These interval points can't be used")
39         return None
40
41
42     while(True):
43         old = a
44         xn = a - f(a)*(b-a)/(f(b) - f(a))
45         err = abs(xn - old)
46         if (err < error):
47             break
48
49         #saving the iterations
50         x.append(xn)
51         y.append(f(xn))

```

```

52
53     if f(xn)*f(a) < 0:
54         a = a
55         b = xn
56     elif f(xn)*f(b) < 0:
57         a = xn
58         b = b
59     elif f(xn) == 0:
60         print("Found exact solution")
61         return xn
62     else:
63         print("Method failed")
64         return None
65
66     n += 1
67
68     print("Numer of iterations :", n)
69
70     #marking the iterations
71     plt.plot(x, y,color='blue',linestyle='dashed', linewidth = 1, marker= '↔'
72             "o" , markerfacecolor='blue', markersize=5, label = "iteration ↔
73             points")
74     plt.legend()
75     return a - f(a)*(b - a)/(f(b) - f(a))
76
77 t = time.process_time()
78
79 #generating a polynomial function
80 poly1d([1,-6,11,-6])
81
82 print("Root found:",secant(0,2.5,poly1d,10**-3))
83
84 #time measure
85 elapsed_time = time.process_time() - t
86
87 print("Elapsed time:",elapsed_time)

```

You can see the outputs of the code using different scenarios below.

Two different equations used:

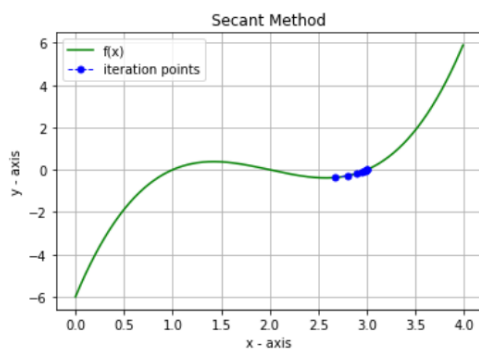
$$f(x) : x^3 - 6x^2 + 11x - 6 \quad (5)$$

$$g(x) : -x^4 + 2x^3 \quad (6)$$

```

3      2
1 x - 6 x + 11 x - 6
Starting interval:[ 0 , 3.5 ]
Error tolerance: 0.001
Nuner of iterations : 10
Root found: 2.9994255968071664
Elapsed time: 0.015625

```

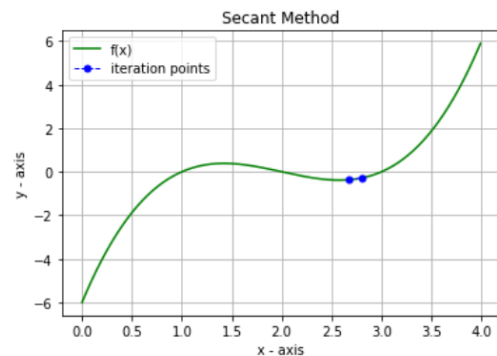


(a) $f(x)$

```

3      2
1 x - 6 x + 11 x - 6
Starting interval:[ 0 , 3.5 ]
Error tolerance: 0.1
Nuner of iterations : 3
Root found: 2.895707202616954
Elapsed time: 0.015625

```

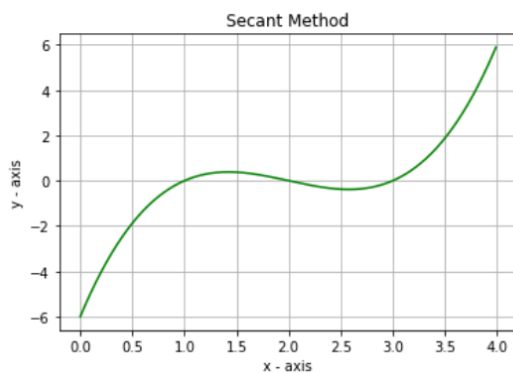


(b) $f(x)$ with different tolerance

```

3      2
1 x - 6 x + 11 x - 6
Starting interval:[ 0 , 2.5 ]
Error tolerance: 0.001
These interval points can't be used
Root found: None
Elapsed time: 0.015625

```

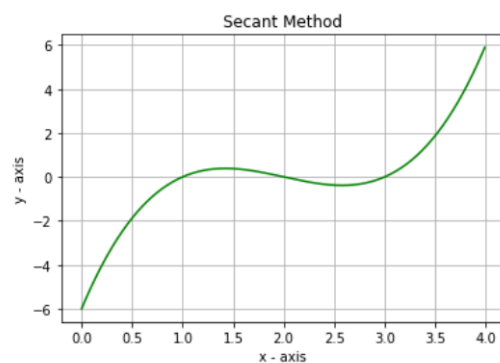


(c) $f(x)$ with different interval

```

3      2
1 x - 6 x + 11 x - 6
Starting interval:[ 0.5 , 1.75 ]
Error tolerance: 0.001
Found exact solution
Root found: 1.0000000000000002
Elapsed time: 0.015625

```



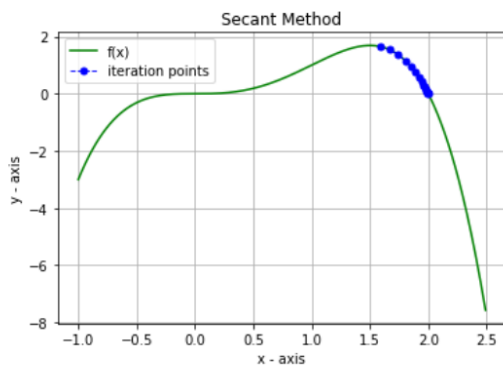
(d) $f(x)$ with different interval

Figure 8: Secant method used on $f(x)$

```

      4      3
    -1 x + 2 x
Starting interval:[ 1.5 , 3 ]
Error tolerance: 0.001
Nuner of iterations : 18
Root found: 1.9981552346349738
Elapsed time: 0.015625

```

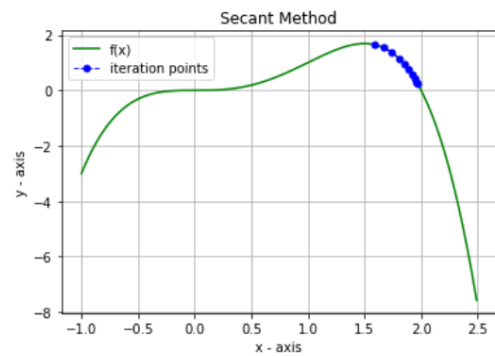


(a) $g(x)$

```

      4      3
    -1 x + 2 x
Starting interval:[ 1.5 , 3 ]
Error tolerance: 0.01
Nuner of iterations : 11
Root found: 1.9788822142102067
Elapsed time: 0.015625

```

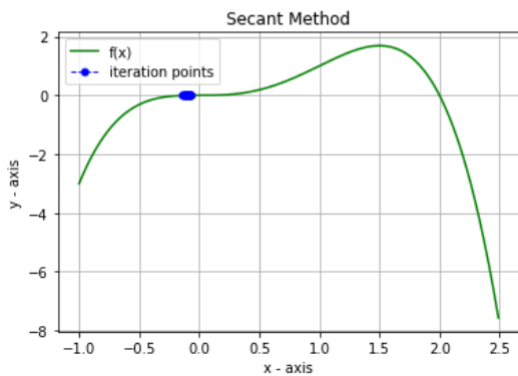


(b) $g(x)$ with different tolerance

```

      4      3
    -1 x + 2 x
Starting interval:[ -0.5 , 1 ]
Error tolerance: 0.001
Nuner of iterations : 28
Root found: -0.07497672895566532
Elapsed time: 0.015625

```

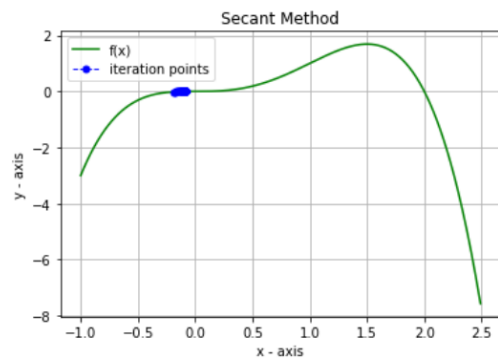


(c) $g(x)$ with different interval

```

      4      3
    -1 x + 2 x
Starting interval:[ -0.5 , 1.5 ]
Error tolerance: 0.001
Nuner of iterations : 33
Root found: -0.07840281105328663
Elapsed time: 0.015625

```



(d) $g(x)$ with different interval

Figure 9: Secant method used on $g(x)$

By looking at these results:

Pros:

1. It does not require use of the derivative of the function
2. It may converge faster or slower than **Newton's Method** depending on how far from the root your initial guesses are.

Cons:

1. It may not converge
2. Doesn't work with every starting interval (case **(c)** on **f(x)**)
3. Iteration number increases rapidly with lower error tolerance (comparing case **(a)** and **(b)** on **f(x)** or **g(x)**)

4 Fixed Point Iteration

Fixed point iteration is an open method computing fixed points of iterated functions.

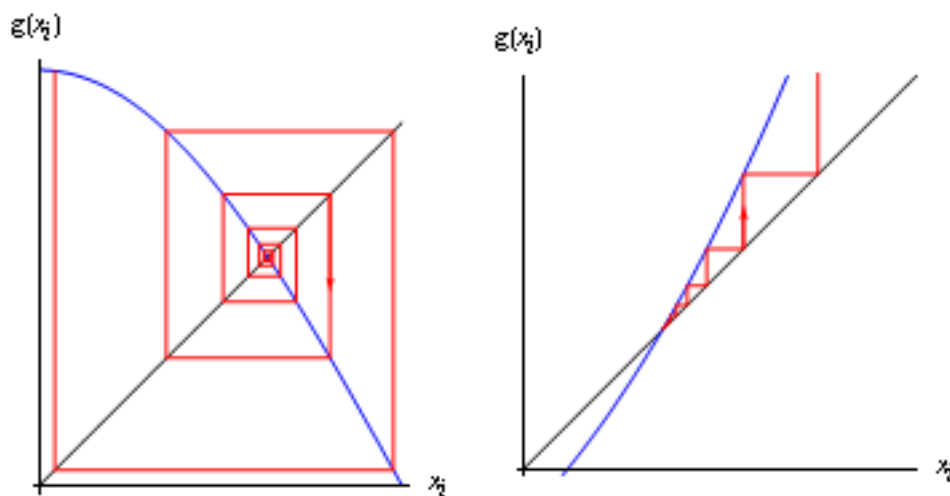


Figure 10: Image from <http://wwwf.imperial.ac.uk/>

You can see the code I have written in order to use this method and generate results below.

Listing 4: Python code – Fixed Point Iteration implemented

```
1 from scipy import poly1d
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import time
5 import math
```



```

6  import plotly.graph_objects as go
7
8  # xn = starting point
9  # f = non-linear equation
10 # g = rewriting f(x)=0 to x = g(x)
11 # error = absolute tolerance
12 # n = number of iterations
13
14
15 def fixed(xn,f,g,error):
16
17     print(f)
18     print("Leaving x alone:")
19     print(g)
20     print("Starting point:",xn)
21     print("Error tolerance:",error)
22
23     #plotting the fuction graph
24     pol_x = np.arange(0,4,0.01)
25     pol_y = f(pol_x)
26     plt.plot(pol_x, pol_y, 'g' , label = "f(x)")
27     plt.xlabel('x - axis')
28     plt.ylabel('y - axis')
29     plt.title('Fixed Point Iteration')
30     plt.grid(True)
31
32     x = []
33     y = []
34
35     n = 1
36
37     while(True):
38
39         #saving the iterations
40         x.append(xn)
41         y.append(f(xn))
42
43         g_x = g(xn)
44
45         xn = g_x
46
47         if (abs(f(g_x)) < error):
48             break
49
50         n += 1
51
52     print("Nuner of iterations :", n)

```

```

53
54     #marking the iterations
55     plt.plot(x, y,color='blue',linestyle='dashed', linewidth = 1, marker= ←
        "o" , markerfacecolor='blue', markersize=5, label = "iteration ←
        points")
56     plt.legend()
57     return g_x
58
59 t = time.process_time()
60
61 #generating a polynomial function
62 polynom = poly1d([1,-6,11,-6])
63 g = poly1d([-1/11, 6/11, 0, 6/11])
64 print("Root found:",fixed(1.9,polynom,g,10**-3))
65
66 #time measure
67 elapsed_time = time.process_time() - t
68
69 print("Elapsed time:",elapsed_time)

```

You can see the outputs of the code using different scenarios below.

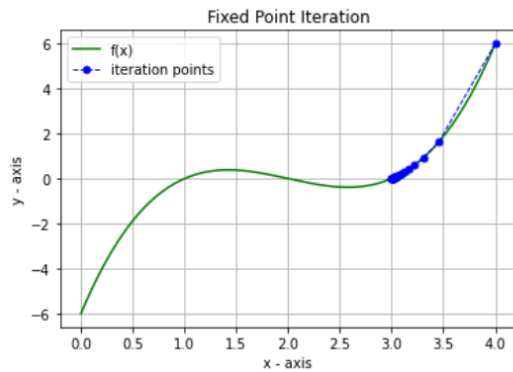
One equation used:

$$f(x) : x^3 - 6x^2 + 11x - 6 \quad (7)$$

```

      3      2
1 x - 6 x + 11 x - 6
Leaving x alone:
      3      2
-0.09091 x + 0.5455 x + 0.5455
Starting point: 4
Error tolerance: 0.001
Nuner of iterations : 32
Root found: 3.0004595266248626
Elapsed time: 0.046875

```

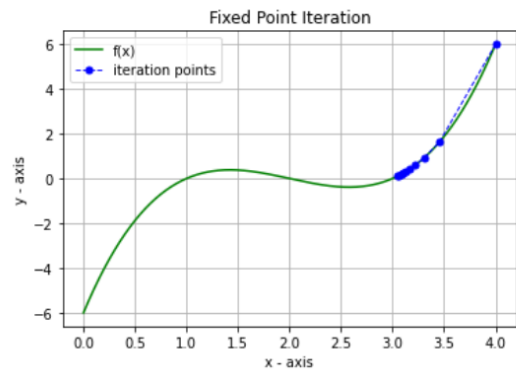


(a) $f(x)$

```

      3      2
1 x - 6 x + 11 x - 6
Leaving x alone:
      3      2
-0.09091 x + 0.5455 x + 0.5455
Starting point: 4
Error tolerance: 0.1
Nuner of iterations : 10
Root found: 3.0408370439288523
Elapsed time: 0.015625

```

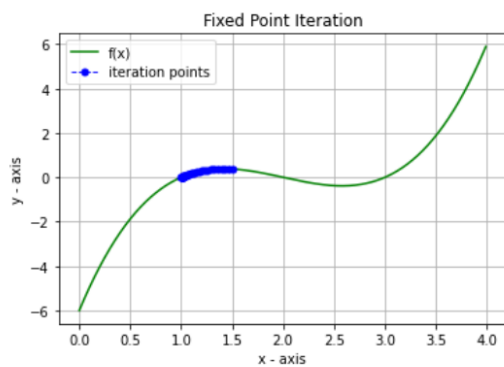


(b) $f(x)$ with different tolerance

```

      3      2
1 x - 6 x + 11 x - 6
Leaving x alone:
      3      2
-0.09091 x + 0.5455 x + 0.5455
Starting point: 1.5
Error tolerance: 0.001
Nuner of iterations : 41
Root found: 1.0004998058880492
Elapsed time: 0.015625

```

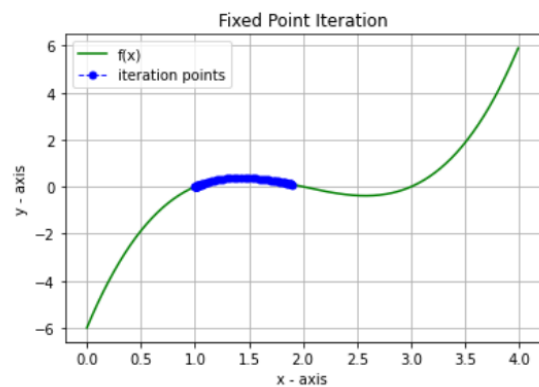


(c) $f(x)$ with different starting point

```

      3      2
1 x - 6 x + 11 x - 6
Leaving x alone:
      3      2
-0.09091 x + 0.5455 x + 0.5455
Starting point: 1.9
Error tolerance: 0.001
Nuner of iterations : 61
Root found: 1.0004892909783027
Elapsed time: 0.03125

```



(d) $f(x)$ with different starting point

Figure 11: Fixed Point Iteration used on $f(x)$

By looking at these results:

Pros:

1. Converges fast
2. Requires one initial guess
3. Easy to program

Cons:

1. It may converge very slowly due to the starting point
2. Many iterations
3. It can't find the root in some cases (on $f(x)$, root $x = 2$ can't be found even with a very close starting point (case **(d)**))
4. Requires at least 2 function evaluations

4.1 Road Map

With all these experiments and results, we can compare these methods in order to choose one for a problem.

- If you want your method to always converge, **Bisection Method** is your only option. The convergence of this method is slow so if you have concerns about speed, you should consider using other methods. In addition, if your equation has more than one root you have to provide multiple appropriate intervals to the method which is may not be very easy.
- If finding the derivative of the function you are dealing with is not a difficult process, you may consider using **Newton's Method**. This convergence of this method is fast compared to the **Bisection Method**. While using **Newton's Method**, you should pick suitable starting points to prevent diversion.
- If you don't want to use the derivative of the function. You can use **Secant Method**. This method requires only one function input but two initial guesses. Similar to the **Bisection Method**, you should provide suitable intervals for every root.
- When appropriate conditions are met, **Fixed Point Iteration** converges fast and it requires only one initial guess. On the other hand, it may converge very slowly due to the starting point. This method requires at least two function evaluations and in some cases it is not *Robust* as I've shown before.