Numerical Methods

Berdan Çağlar Aydın, Istanbul Technical University 26/04/2020

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 it's 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.

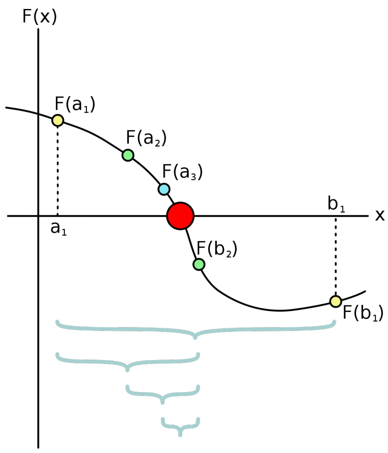


Figure 1: Image from *https://www.wikipedia.org/*

Listing 1: Python code – Bisection method implemented

```python
from scipy import poly1d
import numpy as np
import matplotlib.pyplot as plt
import time
import math
import plotly.graph_objects as go

# a = lower bound of the iteration
# b = upper bound of the iteration
# f = non-linear equation
# error = absolute tolerance
# n = number of iterations
# polynom = f(x)

def bisection(a,b,f,error):

    print(f)
    print("Starting interval:[",a, " , ",b,"]")
    print("Error tolerance:",error)

    #plotting the fuction graph
    pol_x = np.arange(0.5,3.5,0.01)
    pol_y = polynom(pol_x)
    plt.plot(pol_x, pol_y, 'g' , label = "f(x)")
    plt.xlabel('x - axis')
    plt.ylabel('y - axis')
    plt.title('Bisection Method')
    plt.grid(True)

    x = []
    y = []

    n = 1
    if f(a)*f(b) >= 0:
        print("These interval points can't be used")
        return None

    while((b-a)/(2**n) > error):

        mean = (a+b)/2

        #saving the iterations
        x.append(mean)
        y.append(f(mean))

        if f(mean)*f(a) < 0:
```

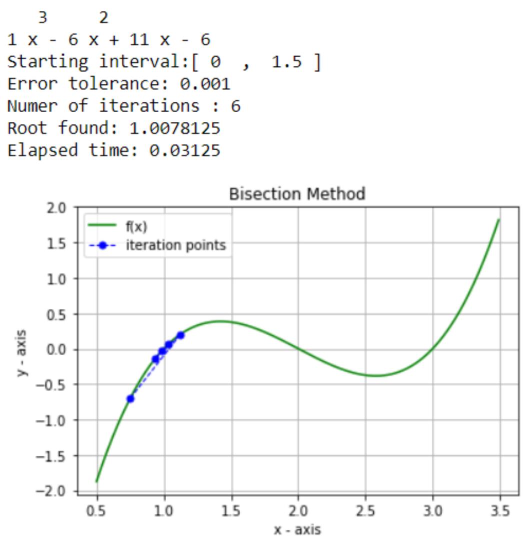
```
            b = mean
        elif f(mean)*f(b) < 0:
            a = mean
        elif f(mean) == 0:
            print("Found exact solution")
            return f(mean)
        else:
            print("Method failed")
            return None

        n += 1

    print("Numer of iterations :", n)

    #marking the iterations
    plt.plot(x, y,color='blue',linestyle='dashed', linewidth = 1, marker= "o" , markerfacecolor='blue', markersize=5, label = "iteration points")
    plt.legend()
    return (a+b)/2

t = time.process_time()

#generating a polynomial function
polynom = poly1d([1,-6,11,-6])

print("Root found:",bisection(1.5,2.25,polynom,10**-3))

#time measure
elapsed_time = time.process_time() - t

print("Elapsed time:",elapsed_time)
```

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

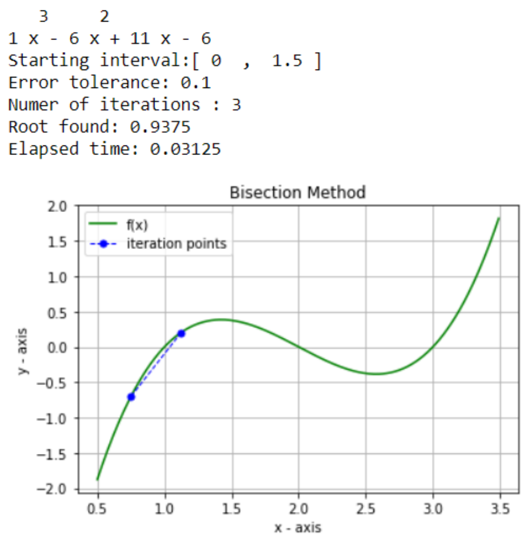
Two different equations used:

$$f(x) : x^3 - 6x^2 + 11x - 6 \tag{1}$$

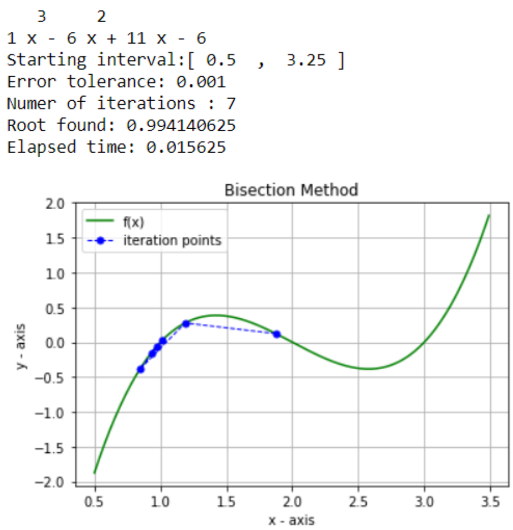
$$g(x) : -2x^4 + 2x^3 - 4x^2 - 10x + 200 \tag{2}$$



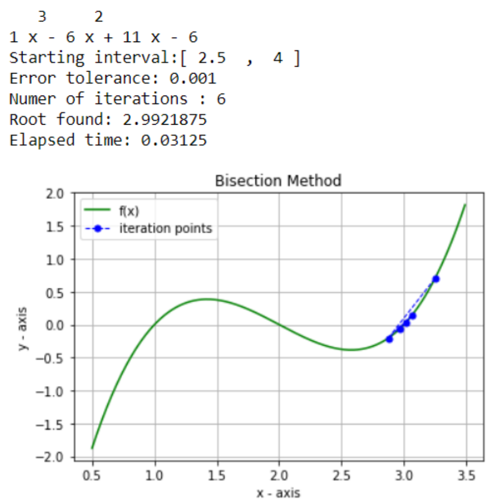
(a) f(x)



(b) f(x) with different tolerance

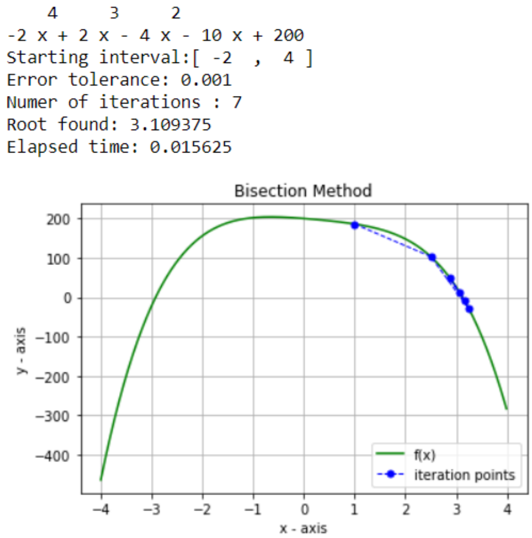


(c) f(x) with different interval

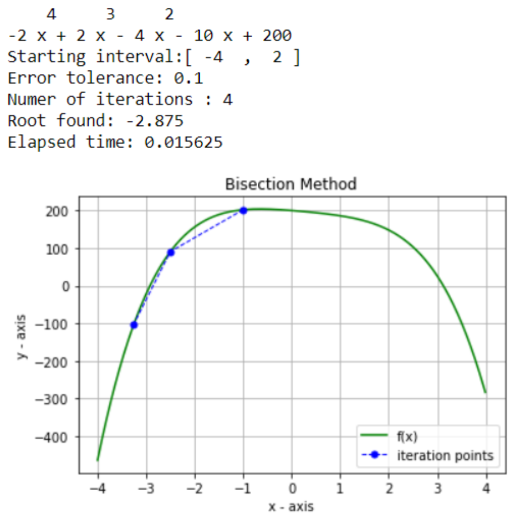


(d) f(x) with different interval

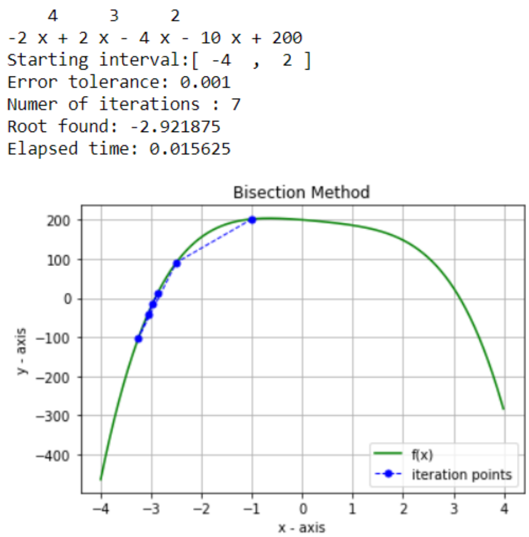
Figure 2: Bisection method used on f(x)



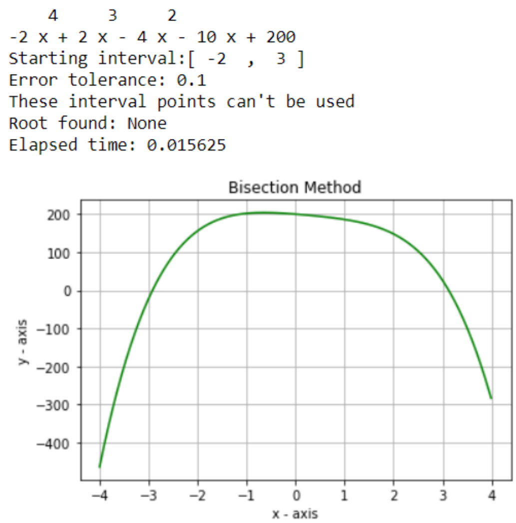
(a) g(x)



(b) g(x) with different tolerance



(c) g(x) with different interval



(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.

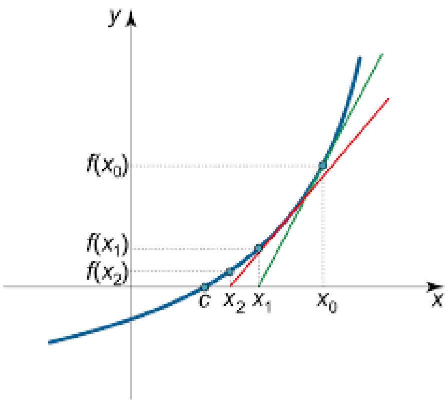


Figure 4: Image from *https://www.math24.net/*

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

Listing 2: Python code – Newthon method implemented

```
from scipy import poly1d
import numpy as np
import matplotlib.pyplot as plt
import time
import math
import plotly.graph_objects as go

# a = initial point
# xn = next itaration point
# fx = next iteration value
```

```python
# f = non-linear equation
# deriv = derivative of f
# error = absolute tolerance
# n = number of iterations

def newton(a,f,error):

    print(f)
    deriv = f.deriv()
    print("Taking derivative:")
    print(deriv)
    print("Starting point:",a)
    print("Error tolerance:",error)

    #plotting the fuction graph
    pol_x = np.arange(0,4,0.01)
    pol_y = f(pol_x)
    plt.plot(pol_x, pol_y, 'g' , label = "f(x)")
    plt.xlabel('x - axis')
    plt.ylabel('y - axis')
    plt.title("Newton's Method")
    plt.grid(True)

    x = []
    y = []

    n = 1
    xn = a

    while(True):
        fx = f(xn)

        #saving the iterations
        x.append(xn)
        y.append(fx)
        deriv_xn = deriv(xn)

        if deriv_xn == 0:
            print("No solution found")
            return None

        old = xn
        xn = xn - (fx/deriv_xn)
        err = abs(xn - old)
        if (err < error):
            break
        n += 1
```

```python


    print("Numer of iterations :", n)

    #marking the iterations
    plt.plot(x, y,color='blue',linestyle='dashed', linewidth = 1, marker= "o" , markerfacecolor='blue', markersize=5, label = "iteration points")
    plt.legend()
    return xn

t = time.process_time()

#generating a polynomial function
polynom = poly1d([1,-6,11,-6])

print("Root found:",newton(0.5,polynom,10**-5))

#time measure
elapsed_time = time.process_time() - t

print("Elapsed time:",elapsed_time)
```

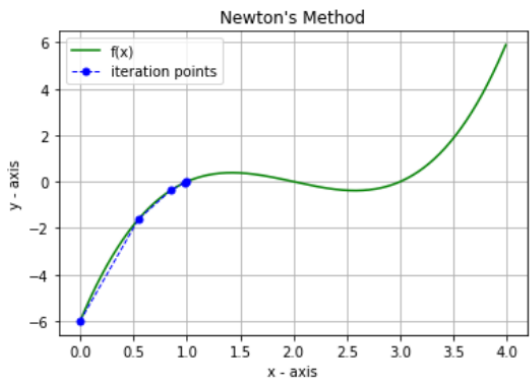
You can see the outputs of the code using different scenarios below.

Two different equations used:

$$f(x) : x^3 - 6x^2 + 11x - 6 \tag{3}$$

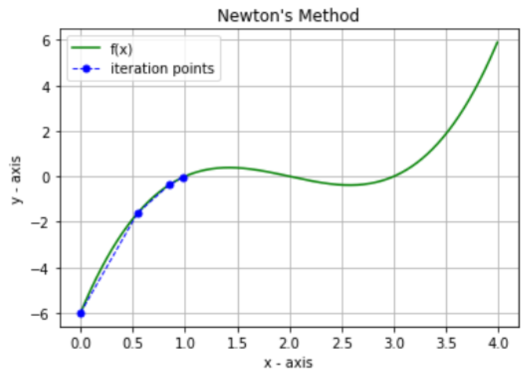
$$g(x) : x^6 - x^5 - 6x^4 - x^2 + x + 10 \tag{4}$$

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



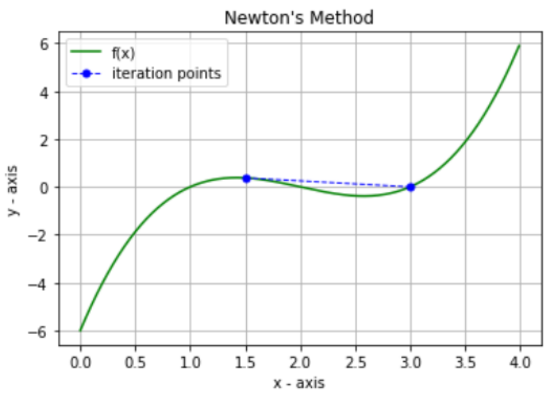
(a) f(x)

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



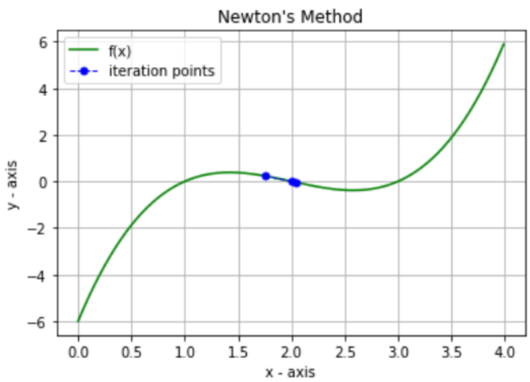
(b) f(x) with different tolerance

```
   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
Numer of iterations : 2
Root found: 3.0
Elapsed time: 0.015625
```



(c) f(x) with different starting point

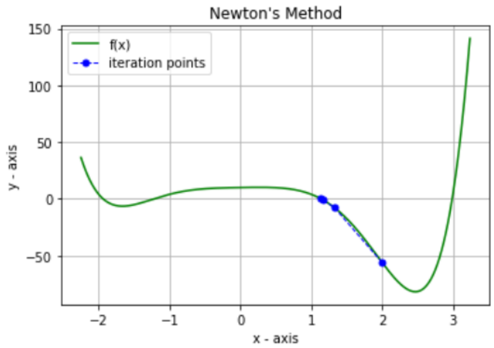
```
   3     2
1 x - 6 x + 11 x - 6
Taking derivative:
   2
3 x - 12 x + 11
Starting point: 1.75
Error tolerance: 0.001
Numer of iterations : 3
Root found: 2.0000000000029865
Elapsed time: 0.015625
```



(d) f(x) with different starting point

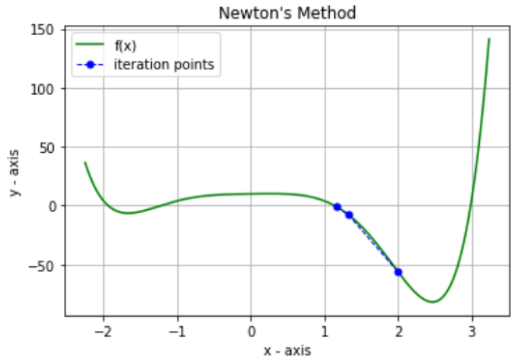
Figure 5: Newton's method used on f(x)

```
   6     5     4     2
1 x - 1 x - 6 x - 1 x + 1 x + 10
Taking derivative:
   5     4      3
6 x - 5 x - 24 x - 2 x + 1
Starting point: 2
Error tolerance: 0.001
Numer of iterations : 4
Root found: 1.1392945911225982
Elapsed time: 0.03125
```



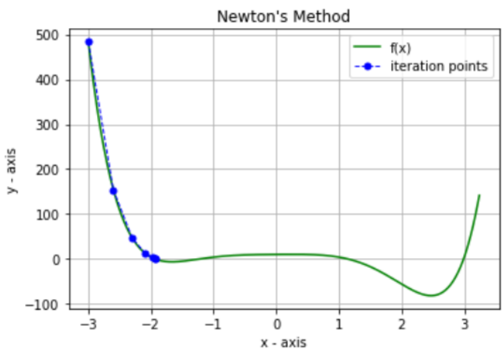
(a) g(x)

```
   6     5     4     2
1 x - 1 x - 6 x - 1 x + 1 x + 10
Taking derivative:
   5     4      3
6 x - 5 x - 24 x - 2 x + 1
Starting point: 2
Error tolerance: 0.1
Numer of iterations : 3
Root found: 1.1401685608920944
Elapsed time: 0.03125
```



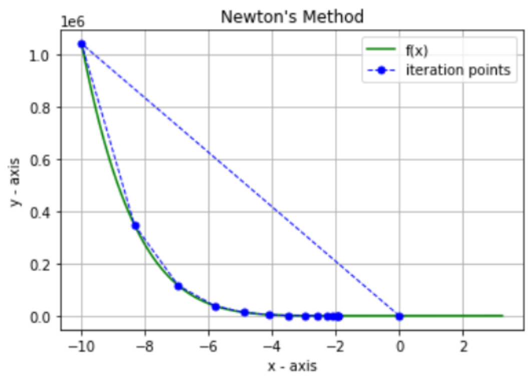
(b) g(x) with different tolerance

```
   6     5     4     2
1 x - 1 x - 6 x - 1 x + 1 x + 10
Taking derivative:
   5     4      3
6 x - 5 x - 24 x - 2 x + 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

```
   6     5     4     2
1 x - 1 x - 6 x - 1 x + 1 x + 10
Taking derivative:
   5     4      3
6 x - 5 x - 24 x - 2 x + 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.

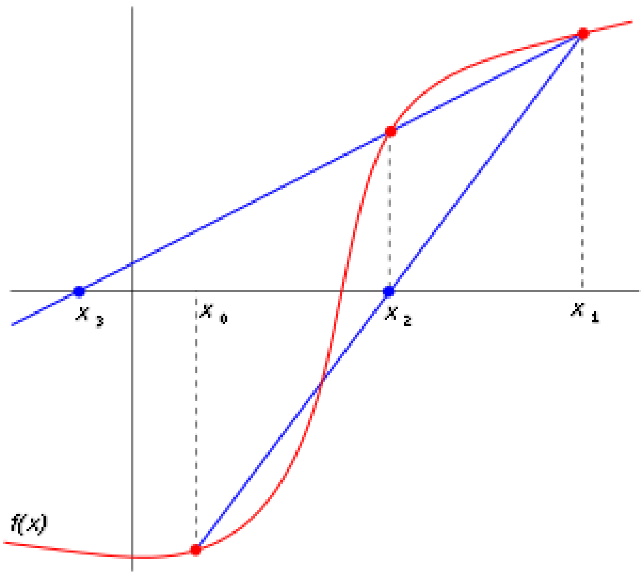


Figure 7: Image from *https://www.wikipedia.org/*

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

```
from scipy import poly1d
import numpy as np
import matplotlib.pyplot as plt
import time
```

```python
import math
import plotly.graph_objects as go

# a = lower bound of the iteration
# b = upper bound of the iteration
# xn = next iteration point
# old = previous iteration to calculate error
# f = non-linear equation
# error = absolute tolerance
# n = number of iterations


def secant(a,b,f,error):

    print(f)
    print("Starting interval:[",a, " , ",b,"]")
    print("Error tolerance:",error)

    #plotting the fuction graph
    pol_x = np.arange(0,4,0.01)
    pol_y = f(pol_x)
    plt.plot(pol_x, pol_y, 'g' , label = "f(x)")
    plt.xlabel('x - axis')
    plt.ylabel('y - axis')
    plt.title('Secant Method')
    plt.grid(True)

    x = []
    y = []

    n = 1

    if f(a)*f(b) >= 0:
        print("These interval points can't be used")
        return None


    while(True):
        old = a
        xn = a - f(a)*(b-a)/(f(b) - f(a))
        err = abs(xn - old)
        if (err < error):
            break

        #saving the iterations
        x.append(xn)
        y.append(f(xn))
```

```python

        if f(xn)*f(a) < 0:
            a = a
            b = xn
        elif f(xn)*f(b) < 0:
            a = xn
            b = b
        elif f(xn) == 0:
            print("Found exact solution")
            return xn
        else:
            print("Method failed")
            return None

        n += 1

    print("Numer of iterations :", n)

    #marking the iterations
    plt.plot(x, y,color='blue',linestyle='dashed', linewidth = 1, marker= "o" , markerfacecolor='blue', markersize=5, label = "iteration points")
    plt.legend()
    return a - f(a)*(b - a)/(f(b) - f(a))

t = time.process_time()

#generating a polynomial function
polynom = poly1d([1,-6,11,-6])

print("Root found:",secant(0,2.5,polynom,10**-3))

#time measure
elapsed_time = time.process_time() - t

print("Elapsed time:",elapsed_time)
```

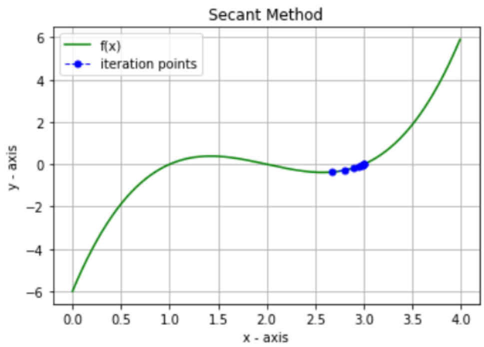
You can see the outputs of the code using different scenarios below.

Two different equations used:

$$f(x) : x^3 - 6x^2 + 11x - 6 \tag{5}$$

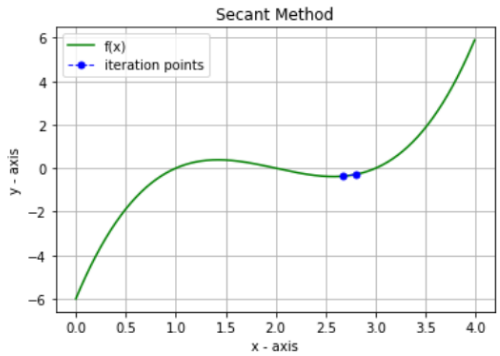
$$g(x) : -x^4 + 2x^3 \tag{6}$$

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



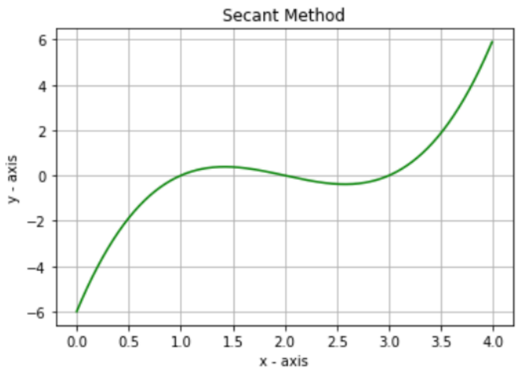
(a) f(x)

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



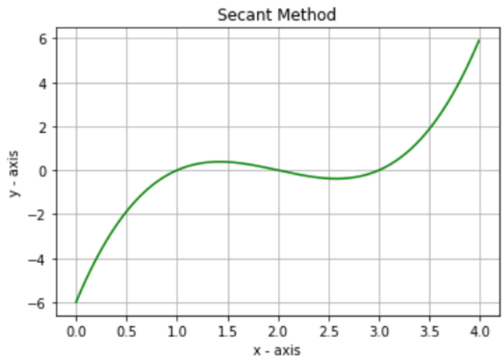
(b) f(x) with different tolerance

```
   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
```



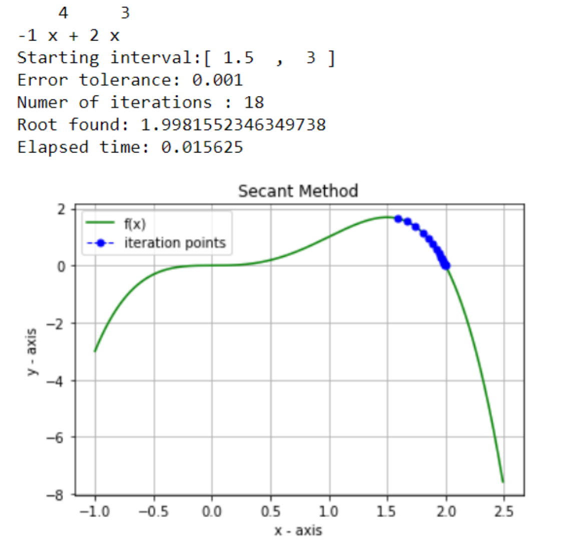
(c) f(x) with different interval

```
   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
```

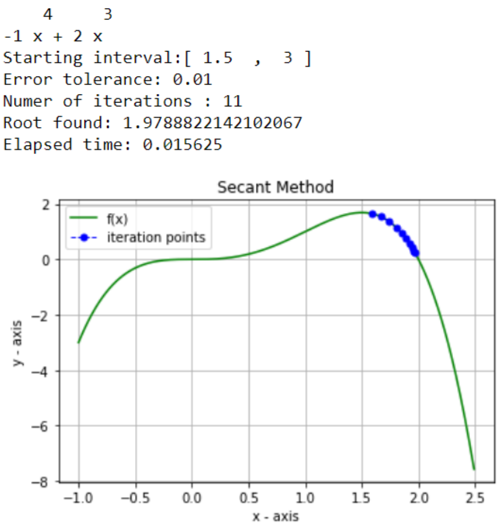


(d) f(x) with different interval

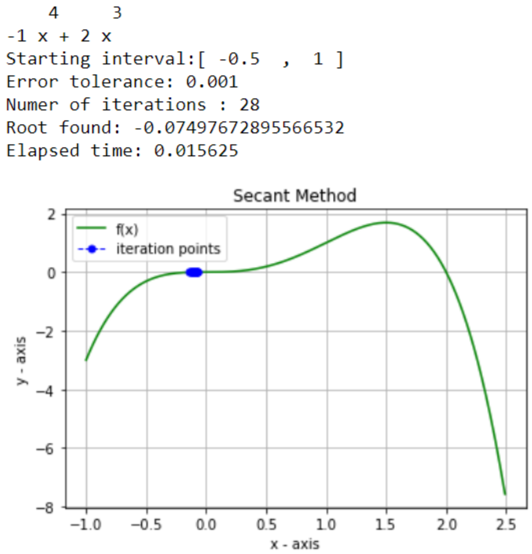
Figure 8: Secant method used on f(x)



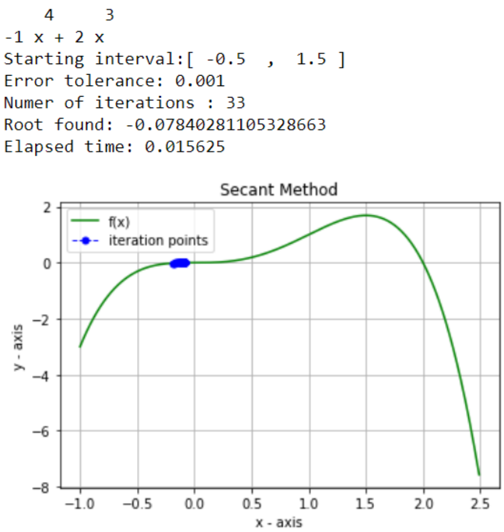
(a) g(x)



(b) g(x) with different tolerance



(c) g(x) with different interval



(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.

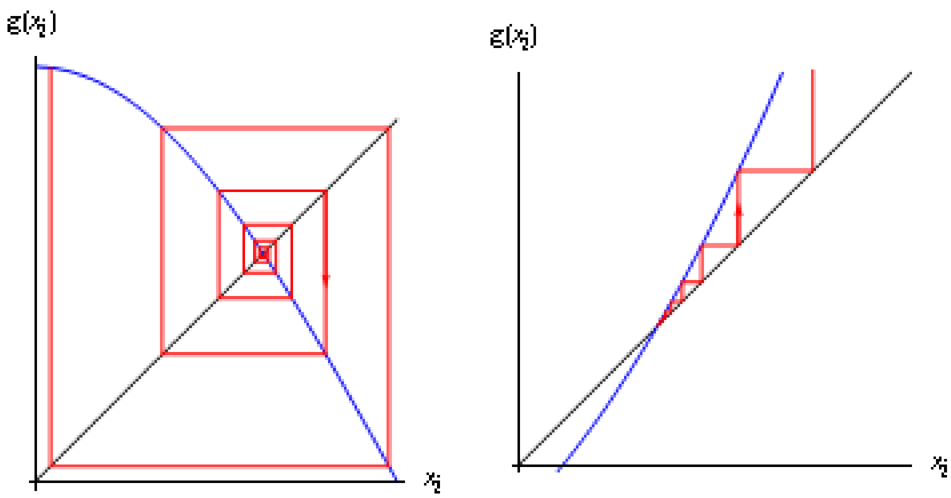


Figure 10: Image from *http://wwwf.imperial.ac.uk/*

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

```
from scipy import poly1d
import numpy as np
import matplotlib.pyplot as plt
import time
import math
```

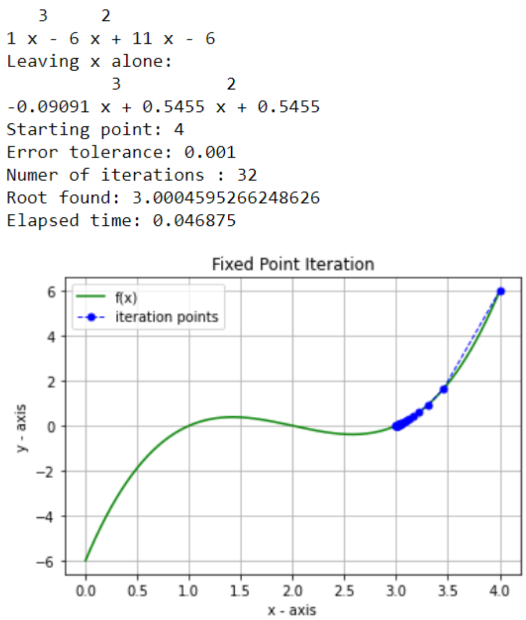
```python
import plotly.graph_objects as go

# xn = starting point
# f = non-linear equation
# g = rewriting f(x)=0 to x = g(x)
# error = absolute tolerance
# n = number of iterations


def fixed(xn,f,g,error):

    print(f)
    print("Leaving x alone:")
    print(g)
    print("Starting point:",xn)
    print("Error tolerance:",error)

    #plotting the fuction graph
    pol_x = np.arange(0,4,0.01)
    pol_y = f(pol_x)
    plt.plot(pol_x, pol_y, 'g' , label = "f(x)")
    plt.xlabel('x - axis')
    plt.ylabel('y - axis')
    plt.title('Fixed Point Iteration')
    plt.grid(True)

    x = []
    y = []

    n = 1

    while(True):

        #saving the iterations
        x.append(xn)
        y.append(f(xn))

        g_x = g(xn)

        xn = g_x

        if (abs(f(g_x)) < error):
            break

        n += 1

    print("Numer of iterations :", n)
```

```

    #marking the iterations
    plt.plot(x, y,color='blue',linestyle='dashed', linewidth = 1, marker= "o" , markerfacecolor='blue', markersize=5, label = "iteration points")
    plt.legend()
    return g_x

t = time.process_time()

#generating a polynomial function
polynom = poly1d([1,-6,11,-6])
g = poly1d([-1/11, 6/11, 0, 6/11])
print("Root found:",fixed(1.9,polynom,g,10**-3))

#time measure
elapsed_time = time.process_time() - t

print("Elapsed time:",elapsed_time)
```

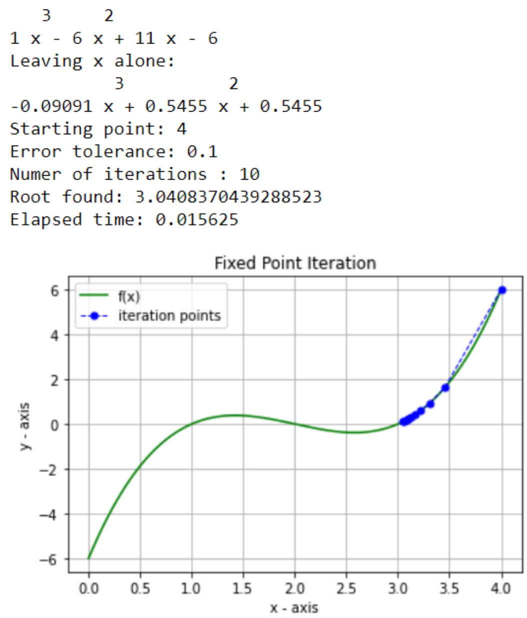
You can see the outputs of the code using different scenarios below.

One equation used:

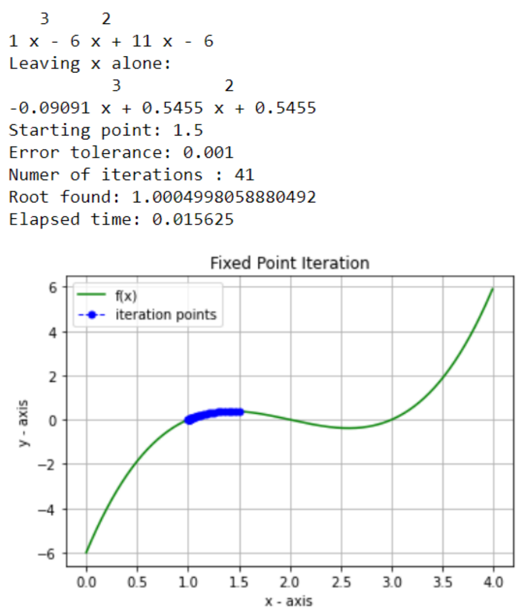
$$f(x) : x^3 - 6x^2 + 11x - 6 \tag{7}$$



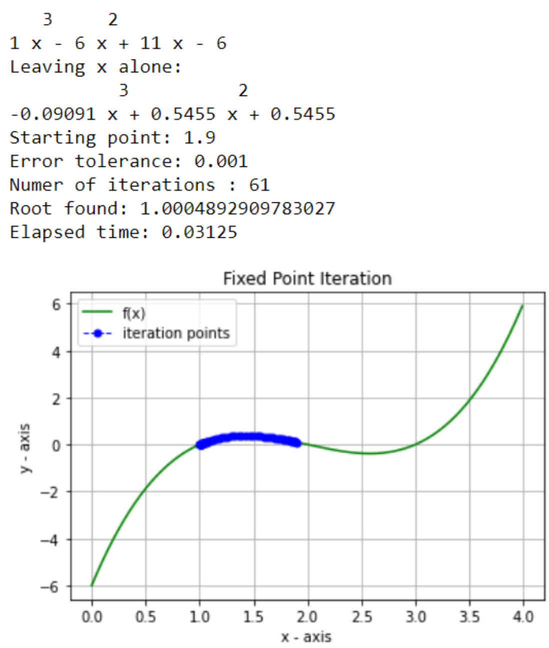
(a) f(x)



(b) f(x) with different tolerance



(c) f(x) with different starting point



(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.