# Numerical Methods Homework № 2

Başar Demir, Istanbul Technical University, 150180080 25/04/2020

## Problem 1

### 0.1 Bisection Method

Bisection method is the simplest method to understand. It needs a and b values which have different function signs. By the Intermediate Value Theorem, if any $a$ and $b$ provides $f(a).f(b) \leq 0$ and $a < b$, there is a root $x$ such that $a \leq x \leq b$. Based on this principle, function narrows the searching interval by checking sign of $f((a+b)/2)$ and updating $a$ and $b$ values until the termination criteria is met. Python implementation is given below.

Listing 1: Python code – Bisection Method

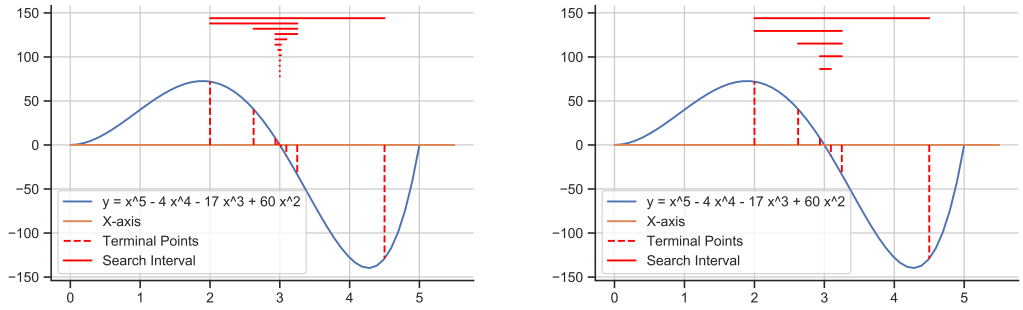
```
def bisection(a,b,f,atol, arr_a=[] , arr_b=[]):
    if a>=b or atol<=0 or f(a)*f(b)>0:
        print("Check your inputs")
        return None
    if f(a)==0:
        return f(a)
    if f(b)==0:
        return f(b)
    n = math.ceil(math.log((b-a)/(2*atol),2))
    i=0
    while i<n:
        arr_a.append(a)
        arr_b.append(b)
        mid_point = (b+a)/2
        if(f(mid_point)==0):
            return mid_point ,arr_a,arr_b,i+1
        elif f(a)*f(mid_point)<0:
            b = mid_point
        elif f(b)*f(mid_point)<0:
            a= mid_point
        else:
            print("Fail")
            return None
        i+=1
    return (b+a)/2 ,arr_a,arr_b,i
```

### 0.1.1 Experiment 1: Iteration-Error Relationship

In this experiment, behaviour changing of same bisection function calls with adjustments in error tolerance is observed. Graphic at the bottom of the page helps us to examine relationship between iteration number and error tolerance in case of error tolerance is dividing by 10 in every step. Nearly, linear dependence is observed and after a certain point function becomes constant because of the limited number representation capability of computer. In short, if we want to get higher result correction, we should do more calculations which is linearly dependent to correction rate.

Listing 2: Function Call and Results

```
1 def f(x):
2     return (x-3)*(x-5)*(x+4)*(x**2)
3 res, arr_a, arr_b, iteration = bisection(2,4.5,f, 0.0005)
4 print(res) #3.00006103515625
5 print(iteration) #12
6 res, arr_a, arr_b, iteration = bisection(2,4.5,f, 0.05) #With 100x ↩
    higher error value
7 print(res) #2.9765625
8 print(iteration) #5
```




(a) 0.0005 Absolute Tolerance

(b) 0.05 Absolute Tolerance

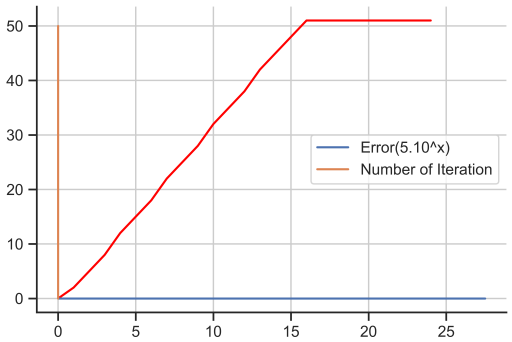
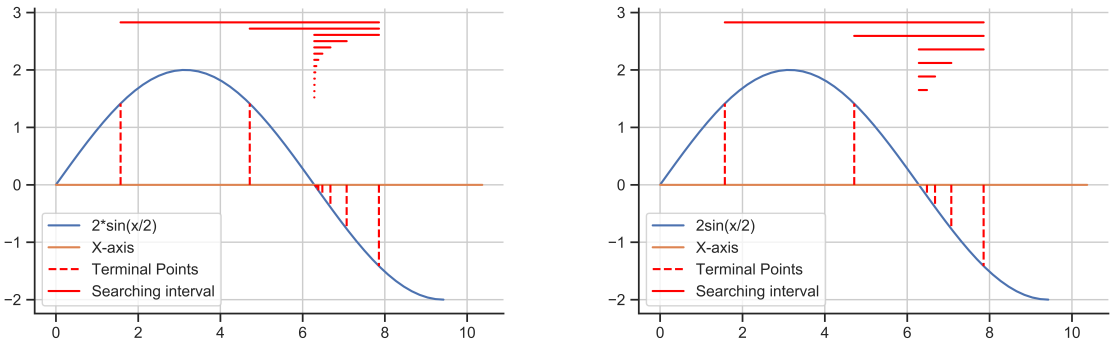



Figure 2: Error-Number of Iterations Relationship

Listing 3: Function Call and Results

```
def f(x):
    return 2*math.sin(x/2)
res, arr_a, arr_b, iteration =bisection(math.pi/2,5*math.pi/2,f, 0.0005)
print(res) #6.283568802376558
print(iteration) #13
print(res) #6.332272692391927
res, arr_a, arr_b, iteration =bisection(math.pi/2,5*math.pi/2,f, 0.05)
print(iteration) #6
```




(a) 0.0005 Absolute Tolerance

(b) 0.05 Absolute Tolerance

Figure 3: Comparison of Effects of Error Tolerance

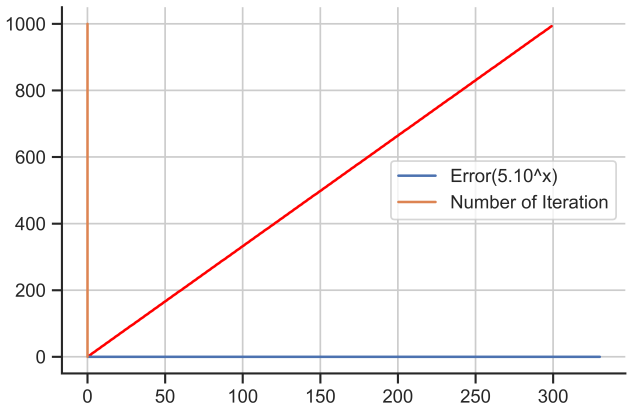


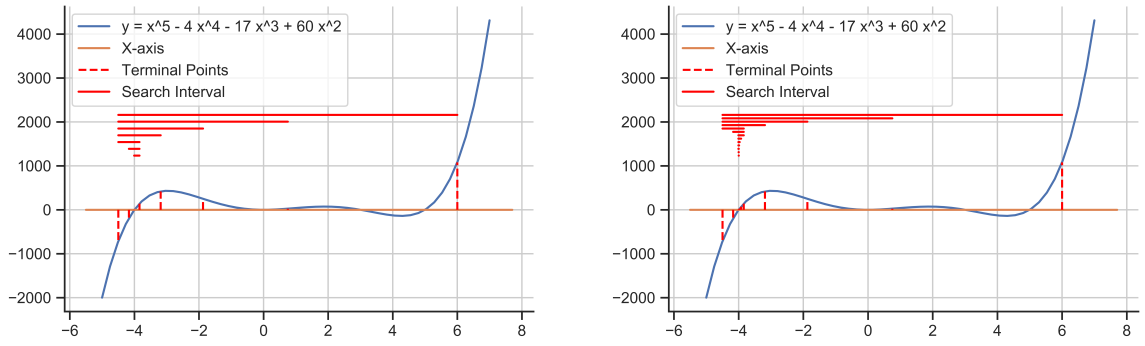

Figure 4: Error-Number of Iterations Relationship

### 0.1.2 Experiment 2: Behaviour in Multiple Root Interval

In this experiment, large interval which contains more than one roots is chosen as a search parameters for function. One of the disadvantages of Bisection Method, it converges to only one root and information is not given as a output for missing roots. It complicates to find all roots of the function. It is independent from the error rate.

Listing 4: Function Call and Results

```
def f(x):
    return (x-3)*(x-5)*(x+4)*(x**2)
res, arr_a, arr_b, iteration = bisection(-4.5,6,f, 0.0005)
print(res)#-3.966796875
print(iteration)#14
res, arr_a, arr_b, iteration = bisection(-4.5,6,f, 0.05)
print(res) #-3.9998016357421875
print(iteration) #7
def h(x):
    return 2*math.sin(x/2)
res, arr_a, arr_b, iteration =bisection(-1,13,h, 0.0005)
print(res) #0.00018310546875
print(iteration) #14
```



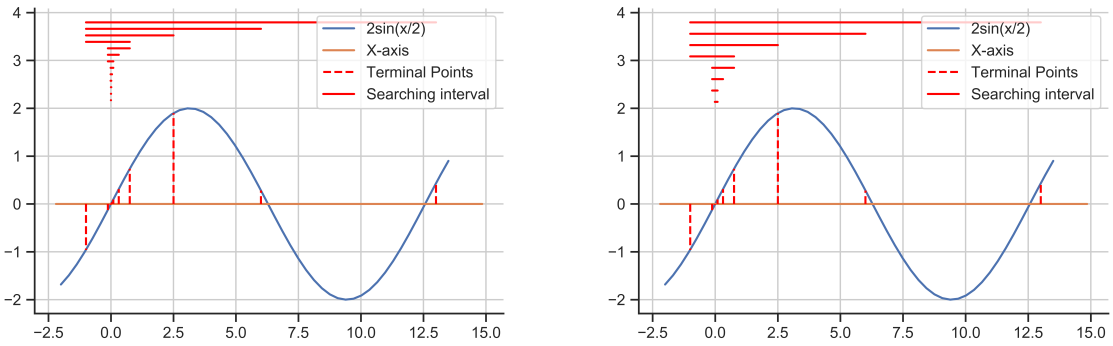

(a) 0.0005 Absolute Tolerance

(b) 0.05 Absolute Tolerance

Figure 5: Behaviour in Multiple Root Interval

Listing 5: Function Call and Results

```
def h(x):
    return 2*math.sin(x/2)
res, arr_a, arr_b, iteration =bisection(-1,13,h, 0.0005)
print(res) #0.00018310546875
print(iteration) #14
res, arr_a, arr_b, iteration =bisection(-1,13,h, 0.05)
print(res) #0.01171875
print(iteration) #8
```




(a) 0.0005 Absolute Tolerance

(b) 0.05 Absolute Tolerance

Figure 6: Behaviour in Multiple Root Interval

### 0.1.3 Experiment 3: Finding Roots of Tangent Functions

To demonstrate one of the disadvantage of the Bisection method, this experiment was set. If there is a root which is on tangent point of a function to x-axis, bisection method does not converge to this root. Because of it is a robust method, it gives error message. It is able to find only secant roots.

Listing 6: Function Call and Results

```
def f(x):
    return (x-3)*(x-5)*(x+4)*(x**2)
res, arr_a, arr_b, iteration = bisection(-2,2,f, 0.0005)
#Check your inputs
```

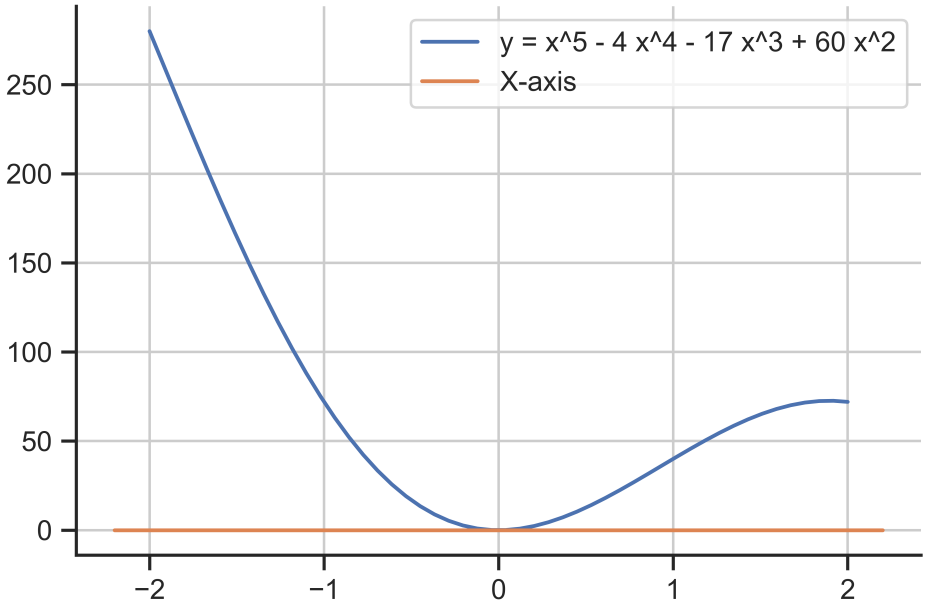


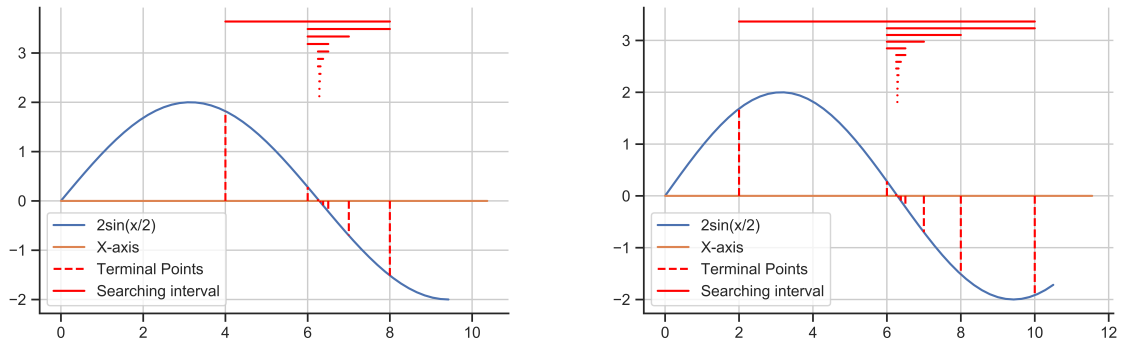

Figure 7: Tangent Root Finding with Bisection Method

### 0.1.4 Experiment 4: Expanding Interval By Keeping Other Variables Constant

By this experiment, observing iteration number changes which depends to searching interval is aimed. Firstly, I started to increment searching interval by 0.1 units in each iteration and I got a small raise in graph. To generalize my hypothesis, I set second experiment which doubles searching interval in each iteration. I got logarithmic ascending graph. This means that, search interval and iteration number is dependent as a logarithmic relation.

Listing 7: Function Call and Results

```
1 def f(x):
2     return 2*math.sin(x/2)
3 res, arr_a, arr_b, iteration =bisection(4,8,f, 0.0005)
4 print(res) #6.28271484375
5 print(iteration) #12
6 res, arr_a, arr_b, iteration =bisection(2,10,f, 0.0005)
7 print(res) #6.28271484375
8 print(iteration) #13
```




(a) [4,8] Interval

(b) [2,10] Interval

Figure 8: Comparison of Effects of Error Tolerance

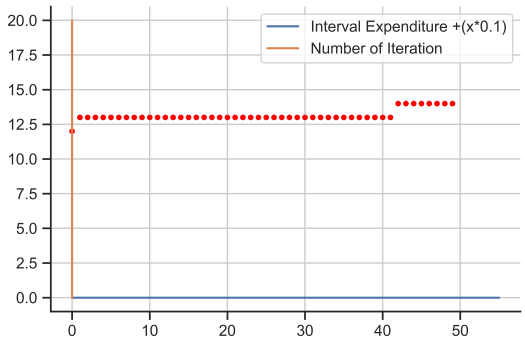



Figure 9: Search Interval-Iteration Relationship

Listing 8: Function Call and Results

```
def h(x):
    return math.tanh(x)
arr= []
a = -1.5
b = 1
diff = (b-a)/2
for i in range(10000):
    res, arr_a, arr_b, iteration = bisection(a,b,h, 0.0005)
    arr.append(iteration)
    a-=diff
    b+=diff
```

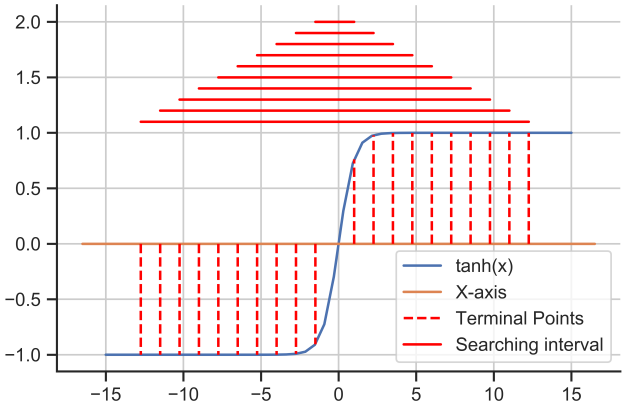



Figure 10: Some Search Intervals

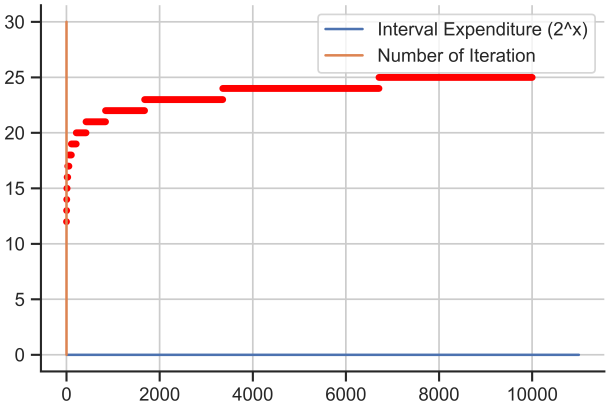



Figure 11: Search Interval-Iteration Relationship

### 0.1.5 Experiment 5: Searching Interval Which Does Not Contain Root

To test the robustness of the Bisection Method, I tried to set a search interval which does not contains any root. Function returns a error message.

Listing 9: Function Call and Results

```
def f(x):
    return 2*math.sin(x/2)
res, arr_a, arr_b, iteration =bisection(3,4,f, 0.0005)
#Check your inputs
def h(x):
    return (x-3)*(x-5)*(x+4)*(x**2)
res, arr_a, arr_b, iteration =bisection(3.2,4.9,h, 0.0005)
#Check your inputs
```

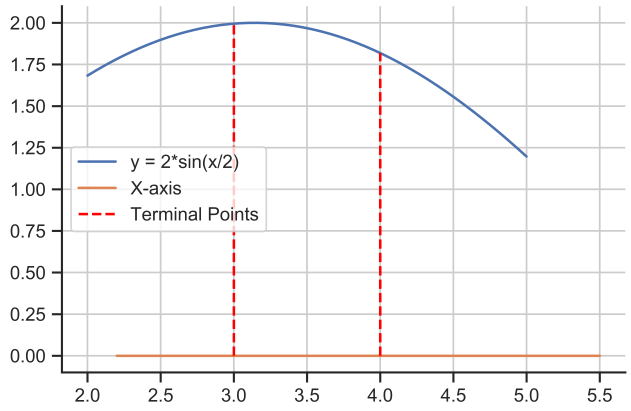



Figure 12: No Root Test Case

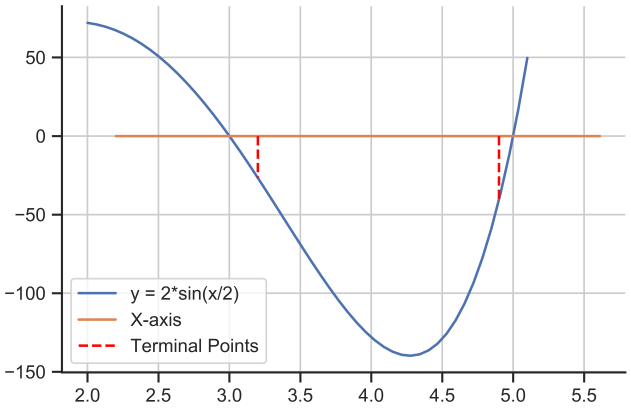



Figure 13: No Root Test Case

### 0.1.6 Experiment 6: Mid-Point Shifting and Iteration Relationship

Bisection Method is based on finding middle point and updating interval corresponding to this value. Therefore, I set an experiment to understand the behaviour behind it. Firstly, I shifted mid-point to the right and it got away from root. Then, in other experiment I shifted right middle point in different equation but in this test, it was getting closer to the real root. Both graphs show us, middle point selection before function call is important to reduce iteration numbers and they are in logarithmic dependence.

Listing 10: Function Call and Results

```
def h(x):
    return math.tanh(x)
a = -100000
b = 999999
diff = 1
mid_point =[]
arr_iteration = []
for i in range(99999):
    mid_point.append(a/2+b/2)
    res, arr_a, arr_b, iteration = bisection(a,b,h, 0.0005)
    arr_iteration.append(iteration)
    a+=diff
```

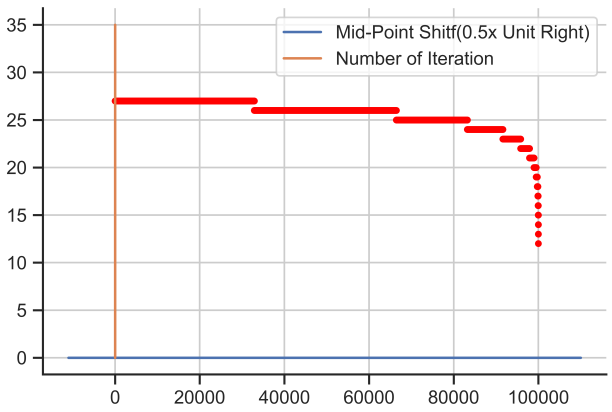



Figure 14: Mid-point Shifting-Iteration Number Relation

Listing 11: Function Call and Results

```
def f(x):
    return (x-3)*(x-5)*(x+4)*(x**2)
a = 4.9
b = 5.2
diff = 1
arr= []
mid_point =[]
for i in range(99999):
    mid_point.append(a/2+b/2)
    res, arr_a, arr_b, iteration = bisection(a,b,f, 0.0005)
    arr.append(iteration)
    b+=diff
```

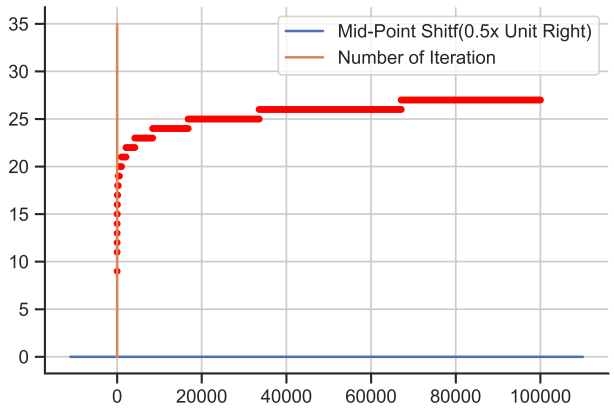



Figure 15: Mid-point Shifting-Iteration Number Relation

### 0.1.7 Experiment 7: Piece-wise Functions Without Root and Non-Continues Functions

To determine Bisection method's required smoothness property, I tested to solve piece-wise function. It supposed that, 0 point is the root of the function and iterates normally. This test case shows that, bisection method requires continuous function. In theoretically, bisection method scans lots of point and terminal point must be defined over the function. Otherwise, it can give an error or find wrong answer.

Listing 12: Function Call and Results

```
1 def piece_wise(x):
2     if(x!= 0):
3         return 1/x
4 res, arr_a, arr_b, iteration = bisection(-10,11,piece_wise, 0.0005)
5 print(res)#-0.0001983642578125
```

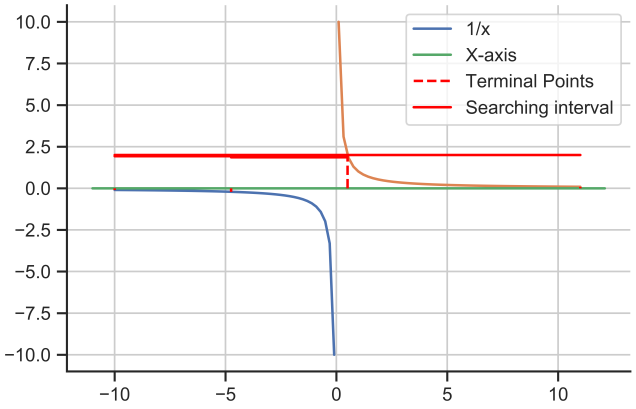



Figure 16: Behaviour in Discontinuous Functions

### 0.1.8 Conclusion

Bisection method is easy and reliable method. It is very recommended in case of when user have high power RAM and CPU, and wants to fail rarely. But, user must draw the graph of the function and find two points which are opposite signed in function before calling method.

**Advantages**

- It is faster than regular searching.
- It is robust. Fails rarely and if it fails, gives an error message
- It always convergent.
- Overflow occurring rate is low. Only exponential functions might cause it.
- Requires only a function and two starting point information.

**Disadvantages**

- It can only find secant roots. Tangent roots cannot find with this method.
- It can find only one root in the interval.
- It is slower than other functions.
- It needs a smooth function. Discontinuities causes problems.

**Rate of Convergence**

Firstly we should find error decreasing rate of every iteration. Therefore, we assume that, there is a root which is $x^*$. When we observe, we cant get $2 * x_{n+1} = x_n + xn - 1$ Then, there is a middle point $x$, $x_n = error_n + x^*$ and $x_{n-1} = error_{n-1} + x^*$. We know that from the observations $errorn + 1 + x^* = ((error_n + x^*) + (error_{n-1} + x^*))/2)$. When we simply this equation, we get $error_{n+1} = error_n/2$. It means that, each iteration error decreases to half.

By the formula of the rate of convergence, this error fraction satisfies $|x_{k+1} - x^*| \leq p|x_k - x^*|$. It means that it is **Linearly Convergent**.

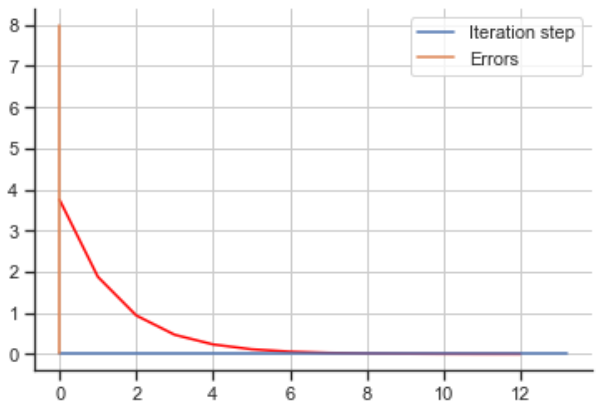



Figure 17: 2*math.sin(x/2) Convergence Graph

## 0.2 Fixed Point Iteration

Fixed point iteration is based on finding the root of $f(x) = 0$ by solving $g(x) = x$ function. At each iteration, function converges to the result and updates x value. Speed, success properties of this method is dependent on external factors. Choosing right $g(x) = x$ and starting point is the most important external factors. Python implementation is given below.

Listing 13: Python code – Fixed Point Iteration

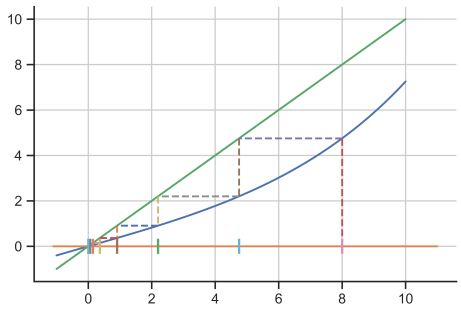
```
1  def fixed_point(g, initial_x, max_error, points = []):
2      points.clear()
3      iteration = 0
4      x = initial_x
5      current_error=sys.maxsize
6      while max_error < current_error:
7          temp = x
8          points.append(x)
9          x = g(temp)
10         current_error = abs(x-temp)
11         iteration+=1
12     return x , points, iteration
```

### 0.2.1 Experiment 1: Iteration-Error Relationship

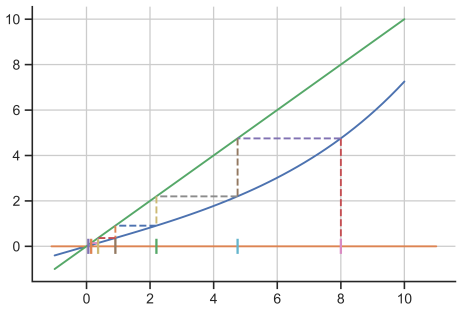
In this experiment, behaviour changing of same bisection function calls with adjustments in error tolerance is observed. Graphic at the bottom of the page helps us to examine relationship between iteration number and error tolerance in case of error tolerance is dividing by 10 in every step. Nearly, linear dependence is observed and after a certain point function becomes constant. In short, if we want to get higher result correction, we should do more calculations which is linearly dependent to correction rate.

Listing 14: Function Call and Results

```
1 def g(x):
2     return 2*math.sinh(x/5)
3 init = 8
4 res = fixed_point(g, init, 0.0005)
5 #iteration number:12
6 draw(g,res,-1,10)
7 res = fixed_point(g, init, 0.05)
8 #iteration number:12
9 draw(g,res,-1,10)
```




(a) 0.0005 Absolute Tolerance




(b) 0.05 Absolute Tolerance

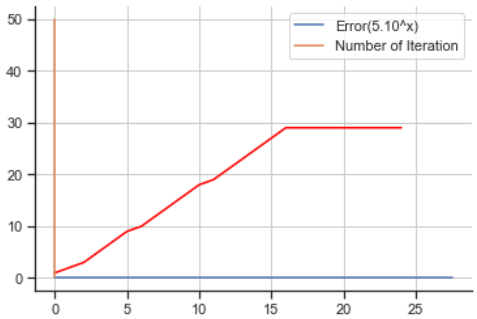
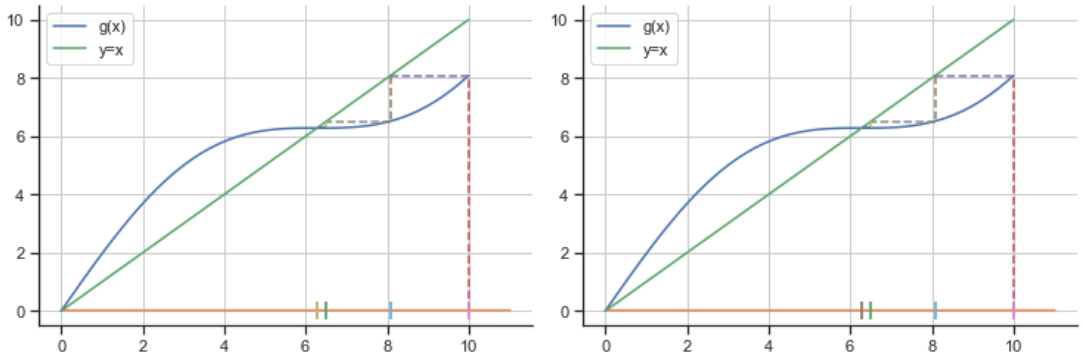



Figure 19: Error-Number of Iterations Relationship

Listing 15: Function Call and Results

```
def g_h(x):
    return 2*math.sin(x/2)+x
init = 10
res = fixed_point(g_h, init, 0.0005)
#iteration number:5
draw(g_h,res,0,10)
res = fixed_point(g_h, init, 0.05)
#iteration number:4
draw(g_h,res,0,10)
```




(a) 0.0005 Absolute Tolerance

(b) 0.05 Absolute Tolerance

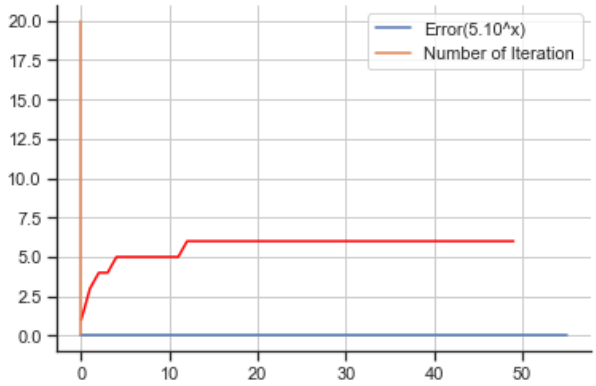



Figure 21: Error-Number of Iterations Relationship

### 0.2.2 Experiment 2: Finding Roots of Tangent Functions

To observe, tangent root finding capability of the Fixed Point Iteration this experiment is set. I chose simple equation which is $f(x) = x^2$. Then I found $g(x) = 0$ function by using $g(x) = x - (f(x)/f^{'}(x))$. After function call, it found the result successfully. It proves that, if proper $g(x)$ can be found, Fixed Point Iteration is able to find tangent roots.

Listing 16: Function Call and Results

```
1 def f(x):
2     return x**2
3 def g__f(x):
4     return x-(x**2/(2*x))
5 init = 8
6 res = fixed_point(g__f, init, 0.0005)
```

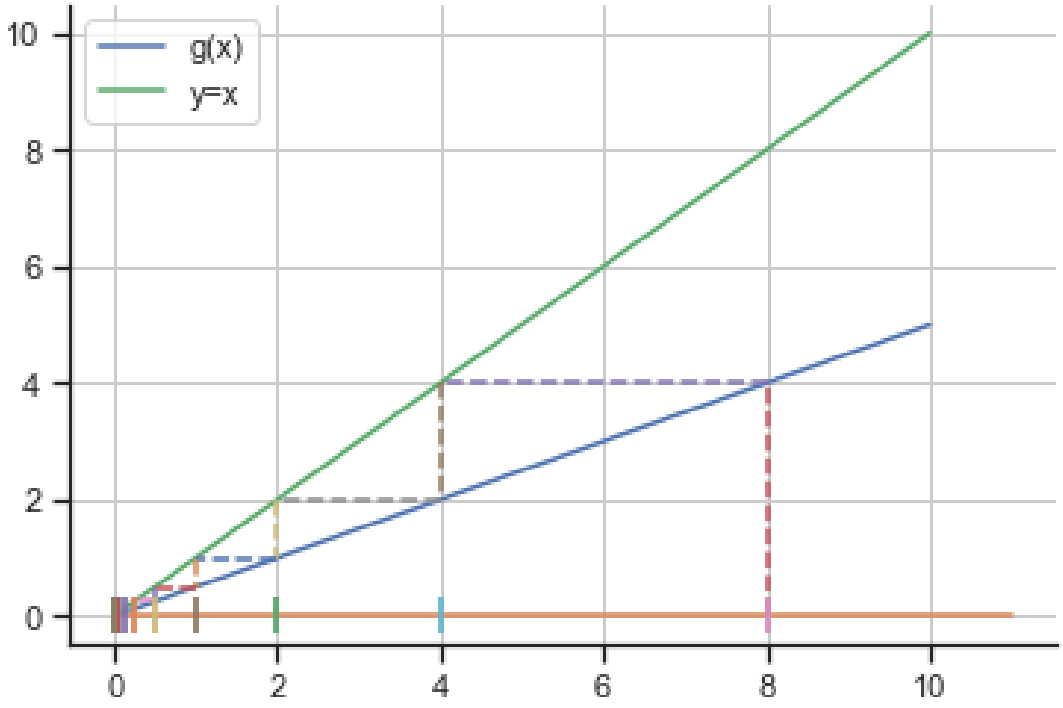



Figure 22: Root Finding with Fixed Point Iteration

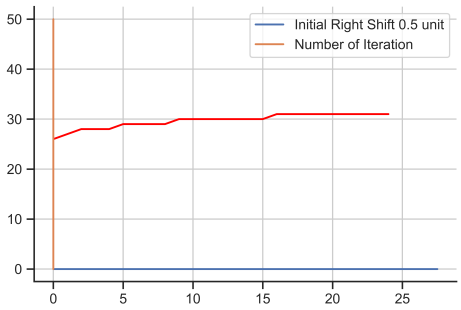
### 0.2.3 Experiment 3: Effects of $g(x)$

In fixed point iteration, finding $g(x)$ is the most critical point. In this experiment, I observed the effects of $g$ function. Therefore, we should do calculations before function call. Choosing wrong $g$ function can cause problems that are given below.

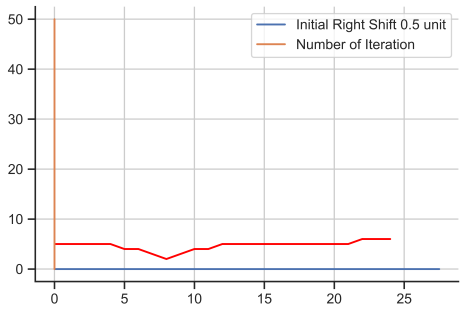
- Problems in converging
- Overflow occurrence
- Slower calculations

Listing 17: Function Call and Results

```
#Root Finding for (x-3)*(x-5)*(x+4)*(x**2)
def a(x):
    return math.pow((4*(x**4)+17*(x**3)-60*(x**2)),1/5)
def b(x):
    return 4+17/x+60/x**2
def c(x):
    return 4*x**5-12*x**4-34*x**3-60*x**2
init = 8
res_a = fixed_point(a, init, 0.0005)#Converges
res_b = fixed_point(b, init, 0.0005)#Converges
res_c = fixed_point(c, init, 0.0005)#Does not Converge
```




(a) Iteration Number of a(x)




(b) Iteration Number of b(x)

Listing 18: Function Call and Results

```
#Root Finding for 2*sinh(x/5)
def d(x):
    return 2/5*math.cosh(x/5)
def e(x):
    return x-2*math.sinh(x/5)
def f(x):
    return x + 4*math.sinh(x/5)
init = 8
res_d = fixed_point(d, init, 0.0005)#Converges
res_e = fixed_point(e, init, 0.0005)#Overflow
res_f = fixed_point(f, init, 0.0005)#Overflow
```

### 0.2.4 Experiment 4: Starting Slope and Overflow Relationship

To visualize, relation between starting slope and overflow relationship this experiment was set. If the slope of the root point is bigger than 1, function cannot diverges and overflow might be occur. If the starting point is locate upside of the $y = x$ curve. Then overflow occurs very quickly. We always check the graph and then determine starting point.

Listing 19: Function Call and Results

```
1 init = 12.76
2 res = fixed_point(g, init, 0.0000005)
3 #res =1.5710828877390292e-07
4 init = 12.78
5 res = fixed_point(g, init, 0.0000005)
6 #Overflow occurs
```

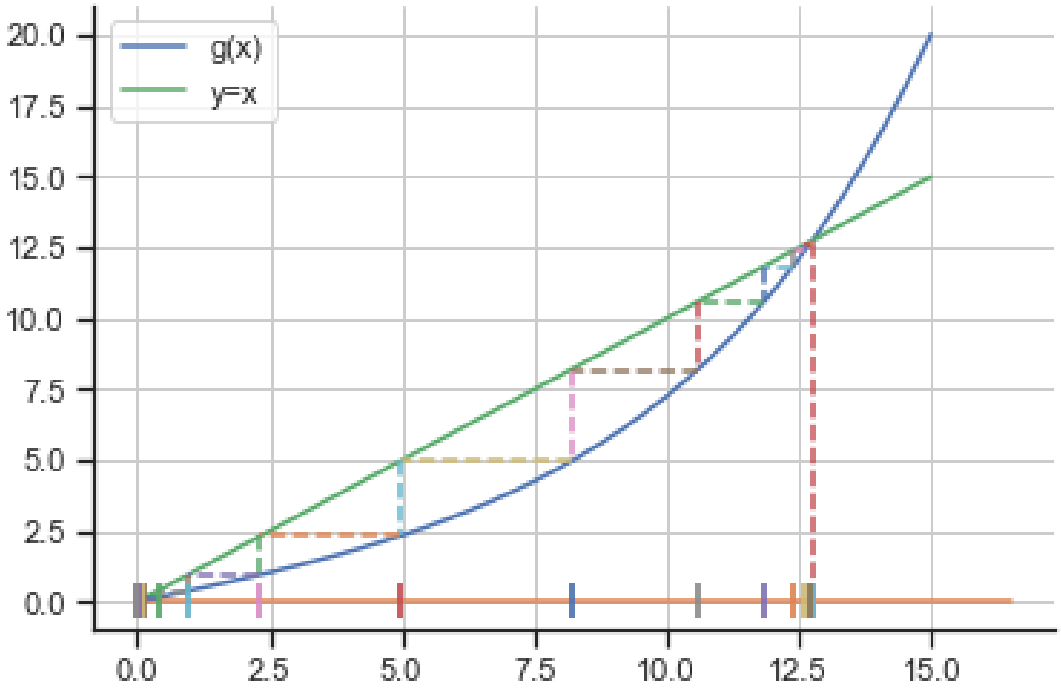



Figure 24: Fixed Point at 12.76

### 0.2.5 Conclusion

Fixed point iteration method has easy implementation and if all rules are provided to function, it works reliable. But, generally it is not easy to find optimized g function and making calculations about slope of the roots. In my opinion, it is hard to handle all things.

**Advantages**

- It is faster than bisection method.
- Easy to implement.
- Needs few calculations.
- Requires only one initial guess
- It can find tangent roots

**Disadvantages**

- It is not robust. Many conditions can make function fail.
- It can only find secant roots. Tangent roots cannot find with this method.
- It needs additional g function and it is hard to find optimal one.
- Many cases causes divergence and overflow problems.
- It needs a smooth function. Discontinuities causes problems.
- Initial guess should provide many rules.

**Rate of Convergence**

If we consider, $x_{n+1} = g(x_n)$ and the $g(x) = x - (f(x)/f^{'}(x))$. Using Taylor's Series, for a fixed point r we can get $g(x) = g(r) + g^{'}(r)(x-r) + 0.5g^{''}(q)(x-r)^2$. If we simplify the equation with limit n goes to infinity. We obtain, $\lim_{x\to\infty} |x_{n+1} - r|/|x_{n-r} - r| = |g^{'}(r)|$. This means that, Fixed point iteration **linearly** converges and its speed is dependent to first derivative of the starting point.

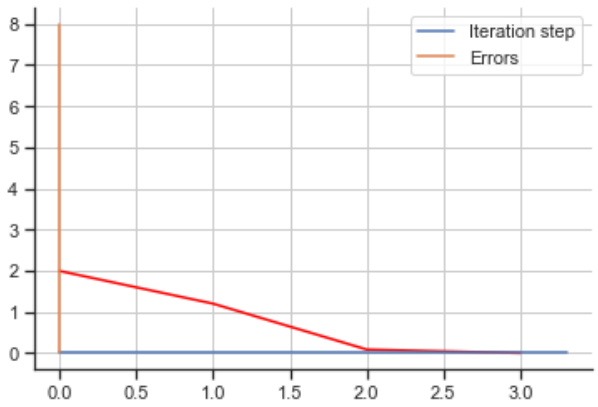



Figure 25: 2*math.sin(x/2) Convergence Graph

### 0.3 Newton's Method

Newton's Method is a fast root finding algorithm which constructed based on $x_n$ and $x_n + 1$ relation in Taylor's Expansion. According to Taylor's Series, functions satisfy $x_{k+1} = xk - (f(xk)/f^{'}(xk))$. Every step of iteration, it calculates $x_{k+1}$ and converges to real root.

Listing 20: Python code – Newton's Method

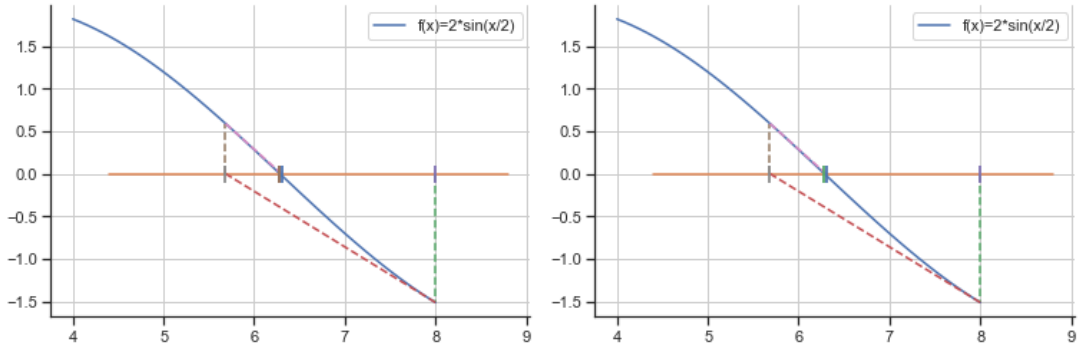
```
def NewtonsMethod(initial_x, f, diff_f, max_error, values=[]):
    x = initial_x
    iteration=0
    current_error=sys.maxsize
    while current_error>max_error:
        values.append(x)
        temp = x
        x = x - (f(x)/diff_f(x))
        current_error = abs(x-temp)
        iteration+=1
    values.append(x)
    print(iteration)
    return x, values
```

### 0.3.1 Experiment 1: Iteration-Error Relationship

In this experiment, behaviour changing of same bisection function calls with adjustments in error tolerance is observed. Graphic at the bottom of the page helps us to examine relationship between iteration number and error tolerance in case of error tolerance is dividing by 10 in every step.

Listing 21: Function Call and Results

```
1 def f(x):
2     return 2*math.sin(x/2)
3 def diff_f(x):
4     return math.cos(x/2)
5 res = NewtonsMethod(8,f,diff_f,0.0005)
6 #iteration:4
7 res = NewtonsMethod(8,f,diff_f,0.05)
8 #iteration:3
```




(a) 0.0005 Absolute Tolerance (b) 0.05 Absolute Tolerance

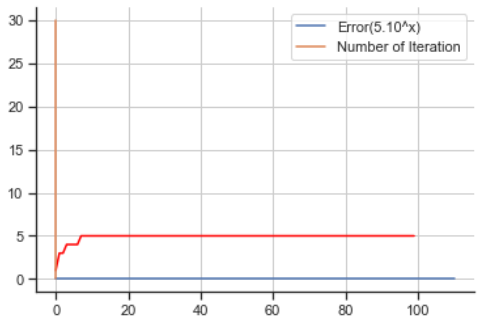
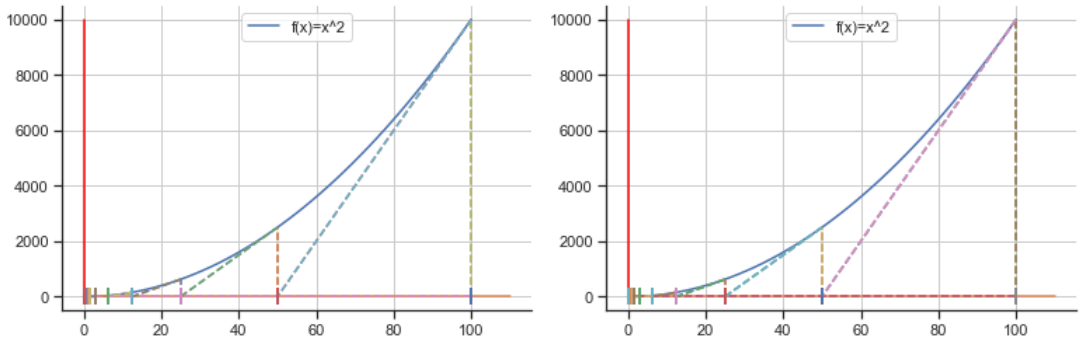



Figure 27: Error-Number of Iterations Relationship

Listing 22: Function Call and Results

```
def f(x):
    return x**2
def diff_f(x):
    return 2*x
res = NewtonsMethod(100,f,diff_f,0.0005)
#iteration:18
res = NewtonsMethod(100,f,diff_f,0.05)
#iteration:11
```




(a) 0.0005 Absolute Tolerance

(b) 0.05 Absolute Tolerance

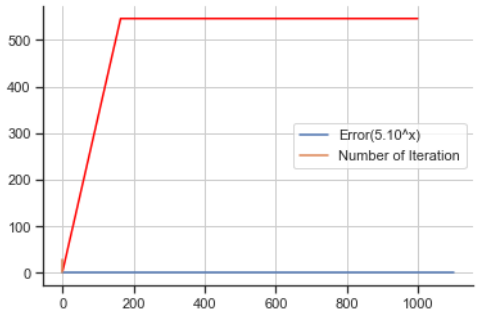



Figure 29: Error-Number of Iterations Relationship

### 0.3.2 Experiment 2: Finding Roots of Tangent Functions

In the first experiment, we observed that Newton's Method can converge to secant roots. In this experiment, root of $f(x) = x^2$ was tried to find. This method is able to converge tangent roots.

Listing 23: Function Call and Results

```
def f(x):
    return x**2
def diff_f(x):
    return 2*x
res = NewtonsMethod(8,f,diff_f,0.0005)
#iteration:14
#res:0.00048828125
```

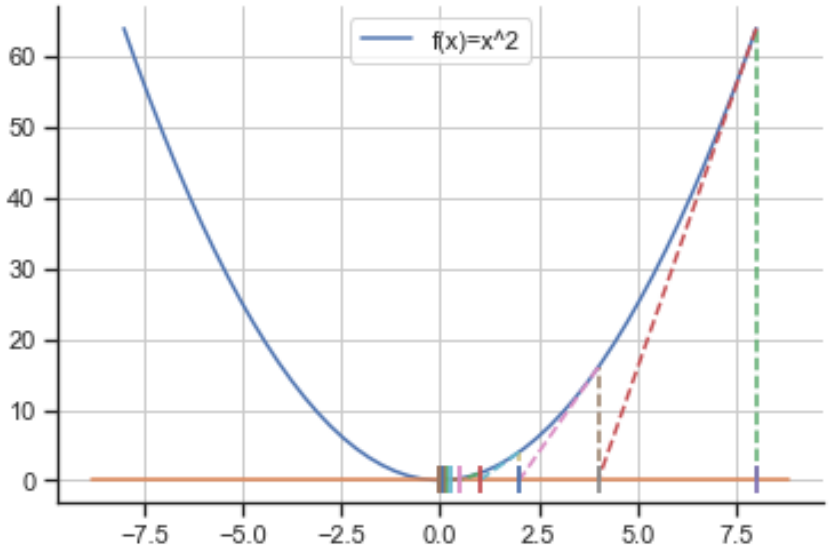



Figure 30: Tangent Root Finding

### 0.3.3 Experiment 3: Discontinuity-Convergence Relationship

In the first experiment, we aimed to determine behaviour of Newton's method with discontinuous function. Method, does not need to smooth functions but function must satisfy every point is defined. In addition, for every part of function, derivative must be calculated.

Listing 24: Function Call and Results

```
def f(x):
    if(x<=0):
        return -x**2+x+10
    if(x>0 and x<5):
        return x**2+5*x
    if(x>=5 and x<10):
        return 5*x
    else:
        return -x
def diff_f(x):
    if(x<=0):
        return -2*x+1
    if(x>0 and x<5):
        return 2*x+5
    if(x>=5 and x<10):
        return 5
    else:
        return -1
res = NewtonsMethod(8,f,diff_f,0.0005)
#Root: -2.7015621300171557
#Iteration:7
```

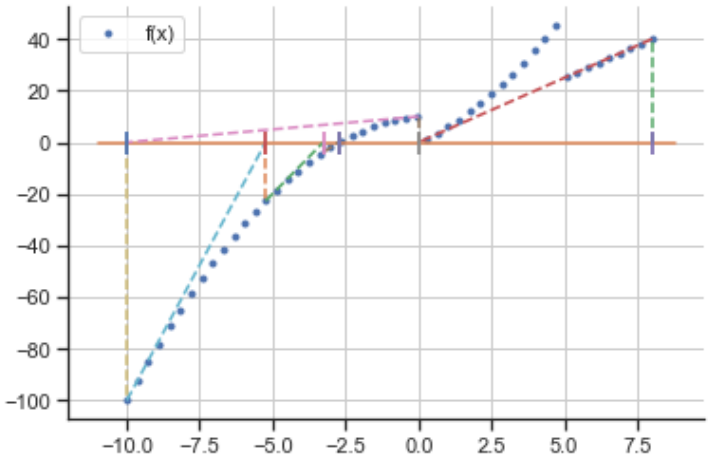



Figure 31: Behaviour in Discontinuous Function

Listing 25: Function Call and Results

```
def f(x):
    if(x<0):
        return x**3-x**2+5
    if(x>=0):
        return math.exp(x)
def diff_f(x):
    if(x<0):
        return 3*x**2-2*x
    if(x>=0):
        return math.exp(x)
res = NewtonsMethod(2,f,diff_f,0.0005)
#Root: -1.4334276722808474
#Iteration:7
```

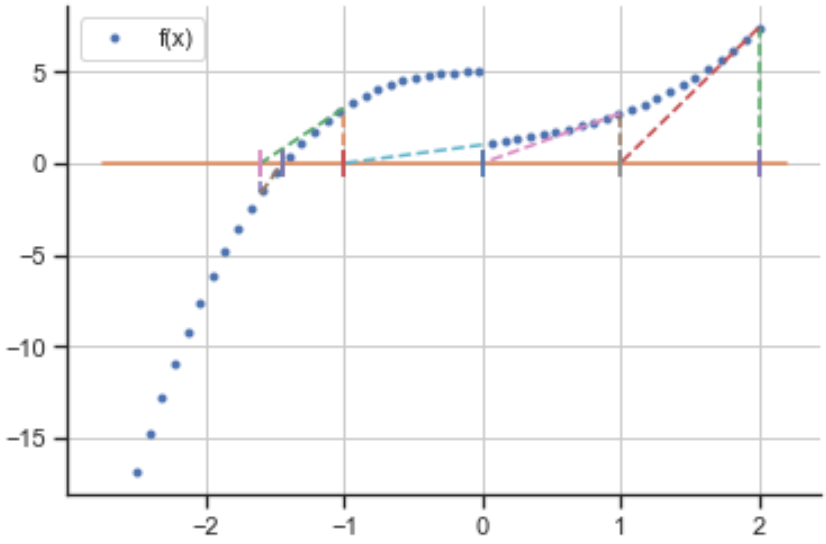



Figure 32: Behaviour in Discontinuous Function

### 0.3.4 Experiment 4: Not Convergent Cases

Initial guess and determining behaviour of the function has a important role in Newton's Method. If proper initial guess is not selected, function cannot even iterate. Two cases is given below:

- If method comes any $x$ which has $f^{'}(x) = 0$, Newton's method gives an zero division error.
- If function has a $x$ which does not defined $f^{'}(x)$, Newton's method cannot converge.

Listing 26: Function Call and Results

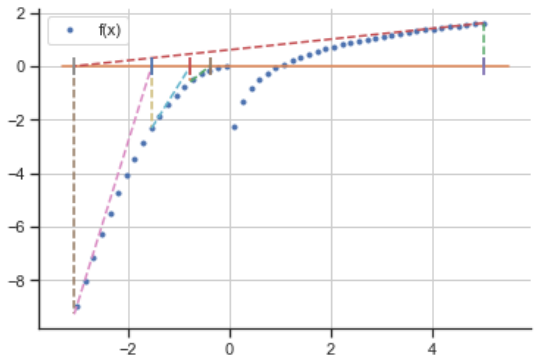
```
def f(x):
    return x**2-5
def diff_f(x):
    return 2*x
res = NewtonsMethod(0,f,diff_f,0.0005)#At x=0, slope is 0.
#ZeroDivisionError: division by zero
def f(x):
    return math.pow(x,1/3)
def diff_f(x):
    return (1/3)*math.pow(x,-2/3)
res = NewtonsMethod(5,f,diff_f,0.0005)#At x=0, derivative does not exist
#ValueError: math domain error
```

### 0.3.5 Experiment 5: Initial Guess-Iteration Relationship

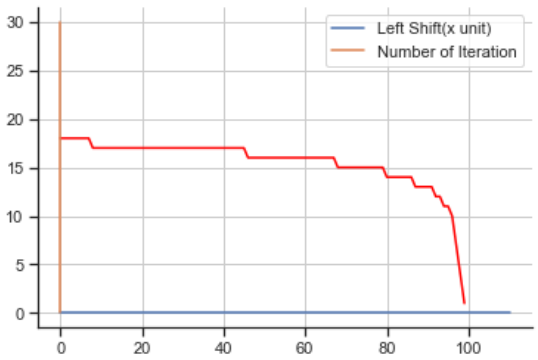
I set this experiment to determine effect of distance between initial guess and root. This method is based on slope operations and therefore, it is not guaranteed that it gets faster as you get closer. First observation shows that closer initial guess has small number of iteration. But the second one, refutes this observation. To summarize, it is not getting faster, when we guess closer points for initialization.

Listing 27: Function Call and Results

```
def f(x):
    if(x<=0):
        return -x**2
    else:
        return math.log(x)
def diff_f(x):
    if(x<=0):
        return -2*x
    else:
        return 1/x
res = NewtonsMethod(5,f,diff_f,0.5) #for observation of function graph
start = 100
multiplier=1
arr= []
for i in range(100):
    res, point,iteration = NewtonsMethod(start,f,diff_f,0.005)
    arr.append(iteration)
    start-=1
```



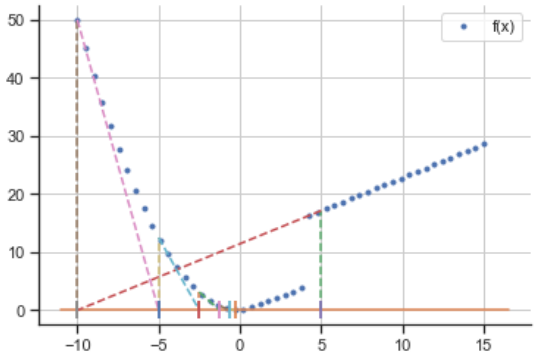

(a) Initial Guess x=5



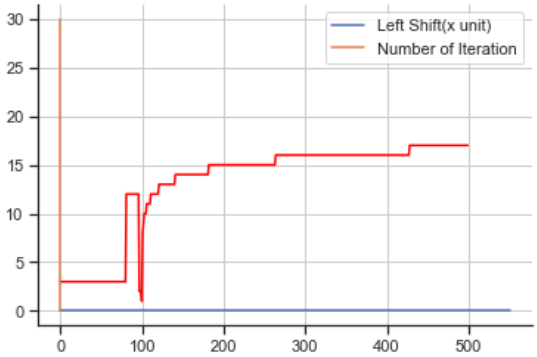

(b) Shifting-Iteration Number Relationship

Listing 28: Function Call and Results

```
def f(x):
    if(x<0):
         return 1/2*x**2
    if(x<4):
        return x
    if(x>=20):
        return 35/18*(x-20)+35
    else:
        return 16/14*(x-4)+16
def diff_f(x):
    if(x<0):
         return x
    if(x<4):
        return 1
    if(x>=20):
        return 35/18
    else:
        return 16/14
res = NewtonsMethod(5,f,diff_f,0.5) #for observation of function graph
start = 100
multiplier=1
arr= []
for i in range(500):
    res, point,iteration = NewtonsMethod(start,f,diff_f,0.005)
    arr.append(iteration)
    start-=1
```




(a) Initial Guess x=5




(b) Shifting-Iteration Number Relationship

### 0.3.6 Conclusion

Newton's method is useful, fast and has a reliable approach to the roots when rules are applied. Generally, it can handle many function solutions. If we have differentiated form of our function, it might be the best method to use.

**Advantages**

- It converges fast.
- Needs only one guess.
- Requires only one initial guess
- Can work with discontinuous functions.

**Disadvantages**

- It is not robust.
- It needs additional $f^{'}(x)$
- It needs a differential function.
- Initial guess should be good. It might cause errors.
- It calls two functions in every step.

**Rate of Convergence**

If there is a root which is $f(x^*) = 0$. By the Taylor's expansion, $0 = f(x^*) = f(x_n) + f^{'}(x_n)((x^* - x_n) + 0.5f^{''}(q)(x^* - x_n)^2$, Subtracting $f(x_n) + (x_{n+1} - x_n)f^{'}(x_n) = 0$ and solving for $error_(n + 1) = x_{n+1} - x^*$. Then we will get, $error_{n+1} = 0.5 * error^2_{n+1} * f^{''}(q)/f^{'}(x_n)$. Finally, we get $(x^* - x_{n+1})/(-x_n)^2 = -f^{''}(x^*)/f^{'}(x*) = error_{n+1}/error^2_n$. This means that, Newton's method **Quadratic** converges .

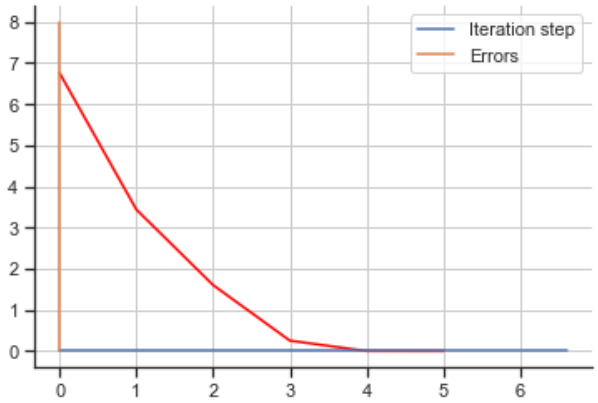



Figure 35: 2*math.sin(x/2) Convergence Graph

## 0.4 Secant Method

Secant Method is a similar method to the Newton's Method. Secant method removes the requirement of $f^{'}(x)$. Instead of this, it uses slope between to points. It starts with two initial guess and tries to converge the root. In every step, it calculates $f(x_n) * (x_n - x_{n-1})/(f(x_n) - f(x_{n-1})$ and assigns to $x_{n+1}$. Finally it terminates when criteria is provided.

Listing 29: Python code – Secant Method

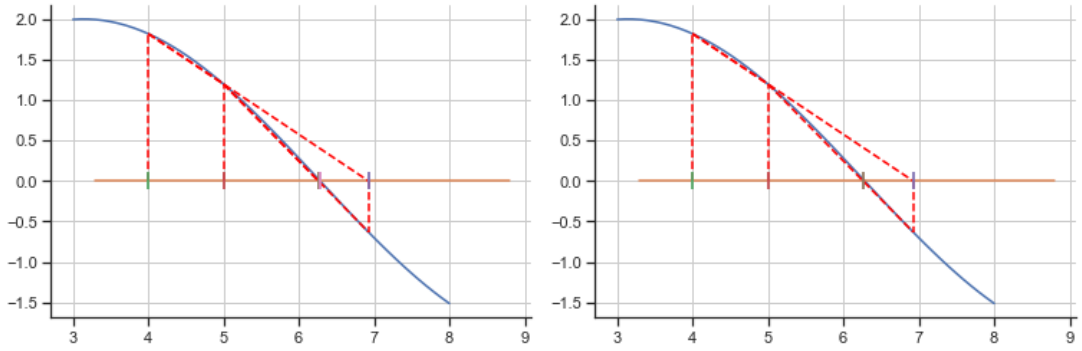
```
1  def SecantMethod(initial_x0,initial_x1,  f, max_error, values=[]):
2      iteration=0
3      current_error=sys.maxsize
4      last_x = initial_x0
5      current_x= initial_x1
6      while current_error>max_error:
7          values.append(last_x)
8          next_x = current_x - (f(current_x)*(current_x-last_x)/(f(↩
               current_x)-f(last_x)))
9          current_error = abs(next_x-current_x)
10         last_x = current_x
11         current_x = next_x
12         iteration+=1
13     values.append(last_x)
14     return current_x, values
```

### 0.4.1 Experiment 1: Iteration-Error Relationship

In this experiment, behaviour changing of same bisection function calls with adjustments in error tolerance is observed. Graphic at the bottom of the page helps us to examine relationship between iteration number and error tolerance in case of error tolerance is dividing by 10 in every step.

Listing 30: Function Call and Results

```
def f(x):
    return 2*math.sin(x/2)
res = SecantMethod(4,5,f,0.0005)
#iteration:5
#res=6.283185299259055
res = SecantMethod(4,5,f,0.0005)
#iteration:3
#res:6.283564205089842
```



(a) 0.0005 Absolute Tolerance

(b) 0.05 Absolute Tolerance

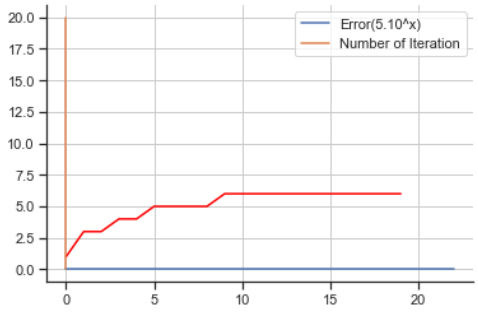
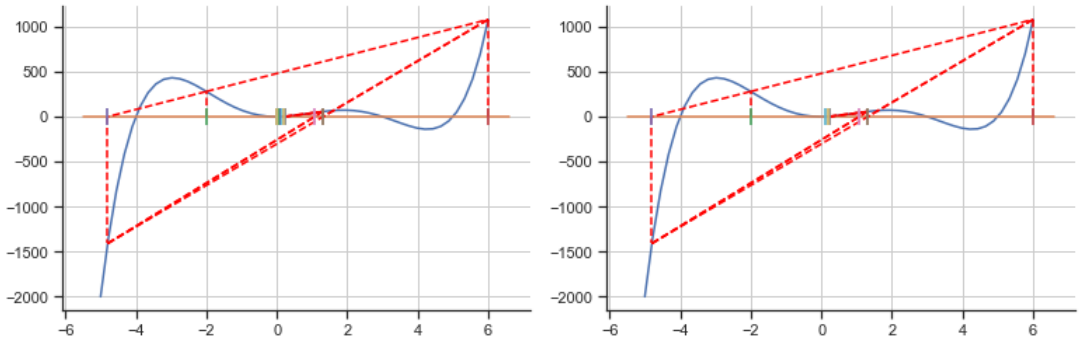



Figure 37: Error-Number of Iterations Relationship

Listing 31: Function Call and Results

```
def f(x):
    return (x-3)*(x-5)*(x+4)*(x**2)
res = SecantMethod(-2,6,f,0.0005)
#iteration:16
#res:0.0007907105180212549
res = SecantMethod(-2,6,f,0.05)
#iteration:7
#res:0.061668242961135315
```




(a) 0.0005 Absolute Tolerance

(b) 0.05 Absolute Tolerance

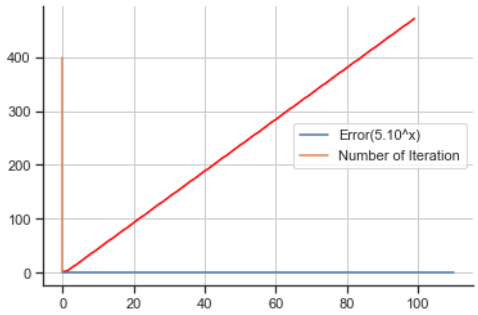



Figure 39: Error-Number of Iterations Relationship

### 0.4.2 Experiment 2: Finding Roots of Tangent Functions

In the first experiment, we observed that Secant Method can converge to secant roots. In this experiment, root of $f(x) = x^2$ was tried to find. This method is able to converge tangent roots.

Listing 32: Function Call and Results

```
1 def f(x):
2     return x**2
3 res = SecantMethod(8,10,f,0.0005)
4 #iteration:20
5 #res:0.0005154041412722751
```

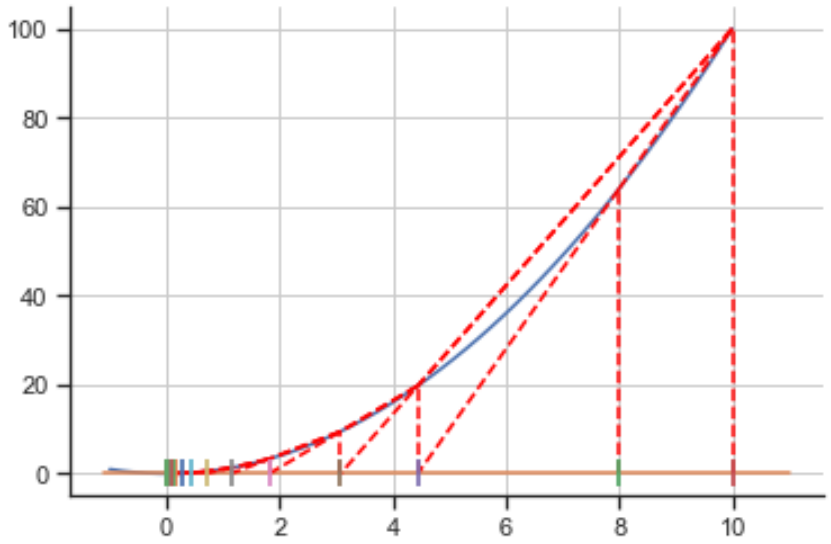



Figure 40: Tangent Root Finding

### 0.4.3 Experiment 3: Discontinuity-Convergence Relationship

In the first experiment, we aimed to determine behaviour of secant method with discontinuous function. Method, does not need to smooth functions but function must satisfy every point is defined. In addition, for every part of function, derivative must be calculated.

Listing 33: Function Call and Results

```
def f(x):
    if(x<=0):
        return -x**2+x+10
    if(x>0 and x<5):
        return x**2+5*x
    if(x>=5 and x<10):
        return 5*x
    else:
        return -x
res = SecantMethod(6,8,f,0.0005)
#Root: -2.7015615575363583
#Iteration:5
```

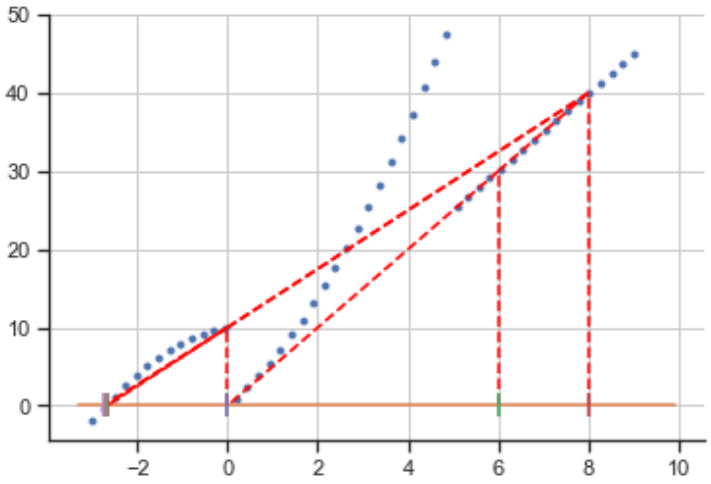



Figure 41: Behaviour in Discontinuous Function

Listing 34: Function Call and Results

```
def f(x):
    if(x<0):
        return x**3-x**2+5
    if(x>=0):
        return math.exp(x)
res = SecantMethod(2,5,f,0.0005)
#Root: -1.433427675463692
#Iteration:27
```

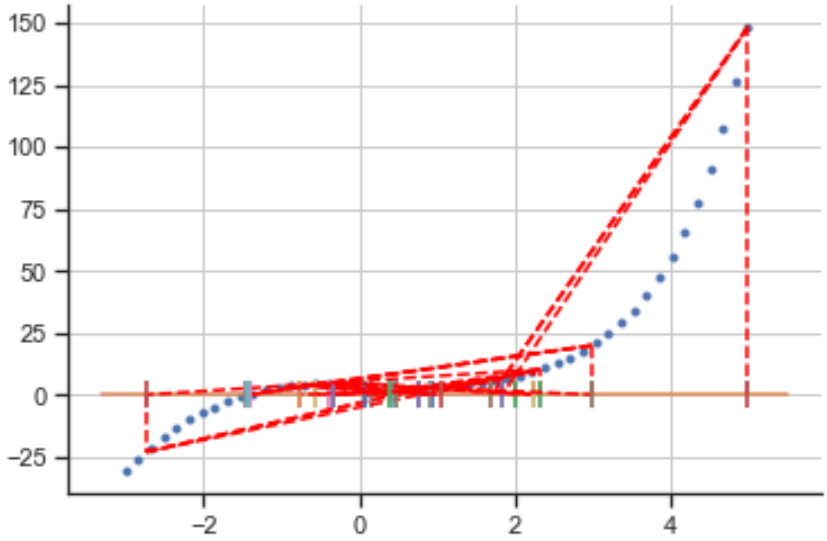



Figure 42: Behaviour in Discontinuous Function

### 0.4.4 Experiment 4: Behaviour in Not Smooth cases

Initial guess and determining behaviour of the function has a important role in Secant Method. If proper initial guess is not selected, function cannot even iterate. Two cases is given below:

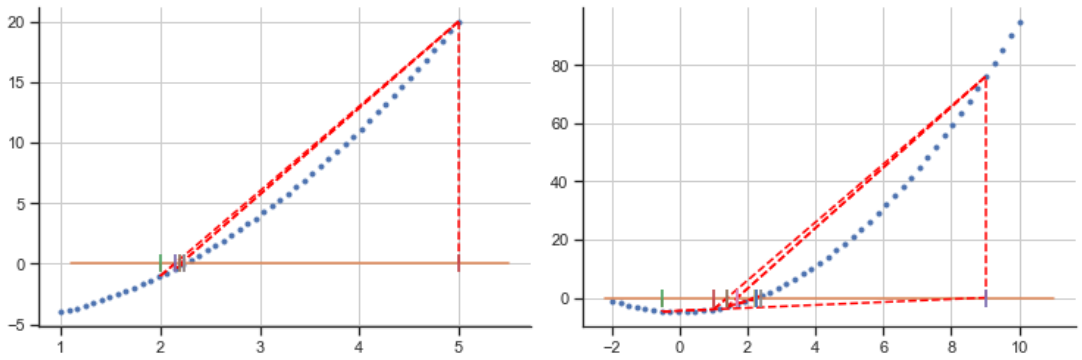
- If values of two initial guesses is equal on function, theoretically slope of line between this two point is 0. This means that function cannot converge and gives zero division error.
- Initial points should not converge to undefined function parts. Then, user might get ValueError.

Listing 35: Function Call and Results

```
def f(x):
    return x**2-5
res = SecantMethod(2,5,f,0.0005)
#res:2.2360679764106504
#iteration:5

res = SecantMethod(-0.5,1,f,0.0005)
#res:2.2360679771124263
#iteration:8

res = SecantMethod(-0.5,-0.5,f,0.0005)
#ZeroDivisionError: float division by zero
```




(a) 0.0005 Absolute Tolerance

(b) 0.05 Absolute Tolerance

Listing 36: Function Call and Results

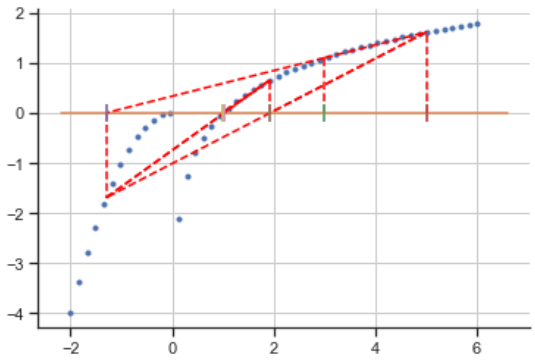
```
def f(x):
    return math.pow(x,1/3)
res = SecantMethod(0,5,f,0.0005)#At x=0, derivative does not exist
#ValueError: math domain error
```

### 0.4.5 Experiment 5: Initial Guesses-Iteration Relationship

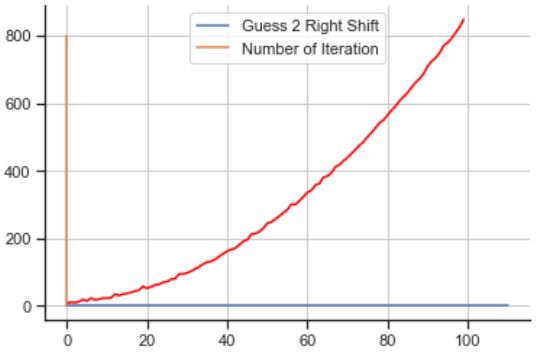
I set this experiment for observing behaviour of method in case of shifting one guess to one unit right. Generalizing method behaviour cannot be possible. It is not guaranteed that iteration number changes by making changes in initial guesses or increasing distance between two guesses. Experiments are given below.

Listing 37: Function Call and Results

```
def f(x):
    if(x<=0):
        return -x**2
    else:
        return math.log(x)
res = SecantMethod(3,5,f,0.0005) #for observation of function graph
arr= []
g1=-2
g2=6
for i in range(n):
    res, point,iteration = SecantMethod(g1,g2,f,0.005)
    arr.append(iteration)
    g2+=1
```



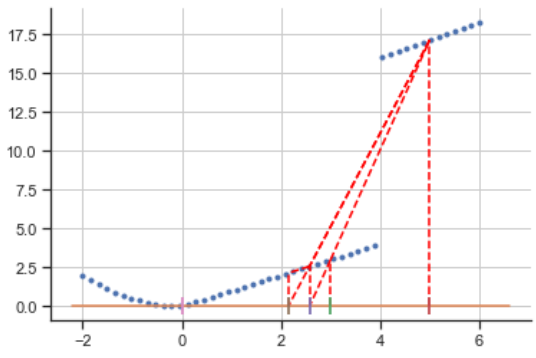

(a) 0.0005 Absolute Tolerance



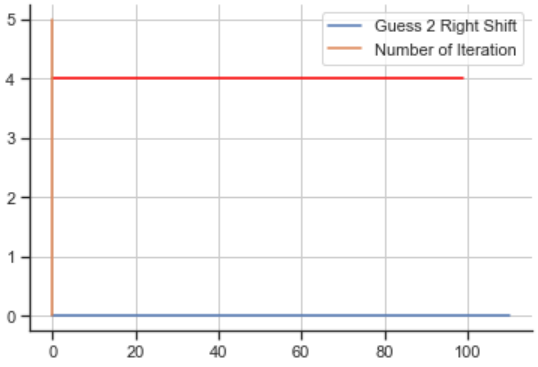

(b) Initial Guess Shifting-Iteration Relationship

Listing 38: Function Call and Results

```
def f(x):
    if(x<0):
         return 1/2*x**2
    if(x<4):
         return x
    if(x>=20):
        return 35/18*(x-20)+35
    else:
        return 16/14*(x-4)+16
res = SecantMethod(3,5,f,0.0005) #for observation of function graph
arr= []
g1=3
g2=5
for i in range(n):
    res, point,iteration = SecantMethod(g1,g2,f,0.005)
    arr.append(iteration)
    g2+=1
```




(a) 0.0005 Absolute Tolerance




(b) Initial Guess Shifting-Iteration Relationship

### 0.4.6 Conclusion

Secant method is based on Newton's Method approach but it does not need the derivative of the function. This eliminates necessity of derivative calculation. On the other hand, it is slower than Newton's Method in many cases. It is ideal for solving complex functions which have hard derivative calculations with higher speed than normal approaches.

**Advantages**

- It converges fast.
- It need only one function call.
- Can work with discontinuous functions.
- Can work with non-smooth functions. **Disadvantages**
- It is not robust.
- Needs two initial guesses.
- Initial guess should be good. It might cause errors.

**Rate of Convergence**

We know that $x_{n+1} = f(x_n) * (x_n - x_{n-1})/(f(x_n) - f(x_{n-1})$. If we consider $x^*$ is the root of $f(x) = 0$. Then, $error_{n+1} = x_{n+1} - x^*, error_n = x_n - x^*$ and $error_{n-1} = x_{n-1} - x^*$. If we write this expression in the first equation, we get $error_{n+1} = (f^{''}(x^*)/(2 * f^{'}(x^*))) * error_{n-1}error_n$. By the based on $error_{n+1} = K * error_n^p$, we can write $K = f^{''}(x^*)/(2 * f^{'}(x^*) * K^{1/p})$. When we solve for $p$, $p = (1 + \sqrt{5})/2$. This means that, its order of convergence is lower than Newton's method and it converges **Super-linearly**.

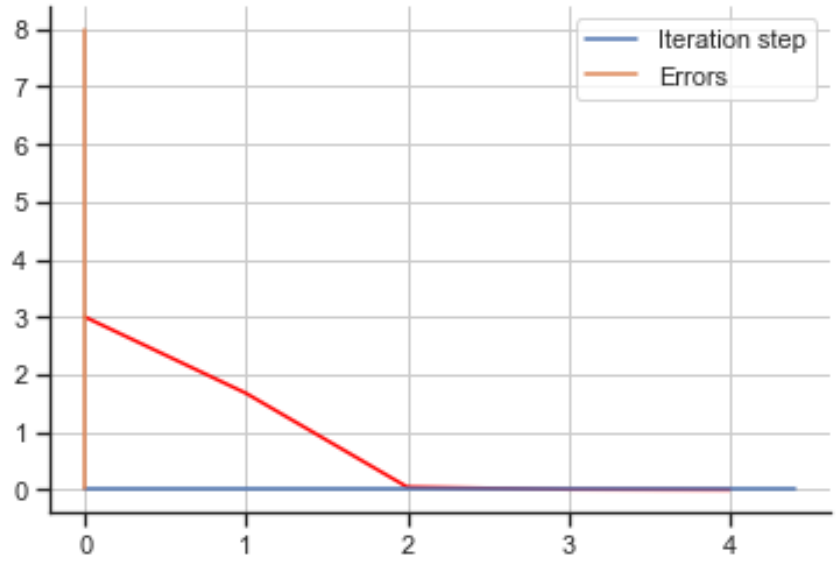



Figure 46: 2*math.sin(x/2) Convergence Graph

## 0.5 Conclusion

In this first problem, we observed different cases for four methods. Each method has many advantages and disadvantages. Therefore, every problem needs different approaches and methods. In this conclusion part, we will compare all methods and we will get a rough road map for each of them.

| Performance Comparison | | | | | |
|---|---|---|---|---|---|
| Method | **Convergence Rate** | **Robustness** | **Additional Needs** | **Smoothness Need** | **Generalization Capability** |
| **Bisection** | Linear | Yes | Two initial guesses | Yes | No |
| **Fixed Point** | Linear | No | g Function, one initial guess | Yes | Yes |
| **Newton's** | Quadratic | No | Derivative, one initial guess | Yes | Yes |
| **Secant** | Super-linear | No | Two initial guesses | No | No |

### 0.5.1 Rough Road Map

#### Bisection Method

- If speed is not important
- If we cannot calculate derivative or any $g(x)$ function
- If we know two point which have opposite signs on function
- If we need robust behaviour

#### Fixed Point Iteration

- If we want simple implementation
- If we are confident enough to find a optimal $g(x)$ function
- If we do not know the graph of the function

#### Newton's Method

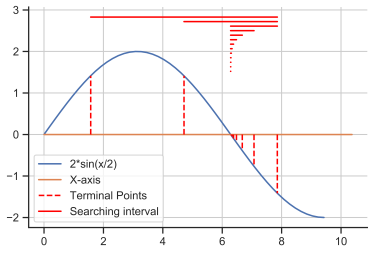
- If we want fast convergence
- If we know the derivative of the function.
- If we have differentiable function

#### Secant Method

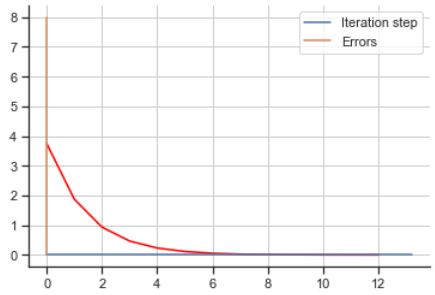
- If we want fast convergence
- If we have complex and hard to differentiate function
- If we do not have smooth function
- If we have good two initial guesses

### 0.5.2 Same Function Behaviours

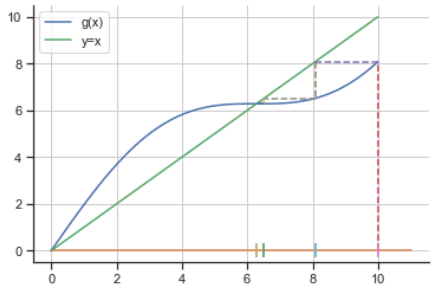
To understand this methods better, I put graphs of methods that are solve same function. We can easily compare solving styles and convarge speeds. Root finding of $f(x) = 2sin(x/2)$ with 0.0005 absolute error criteria:



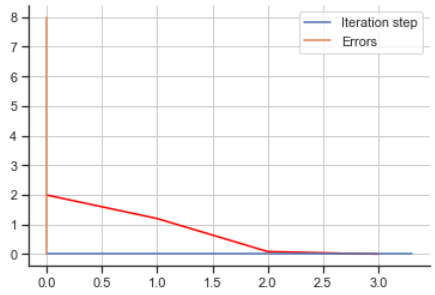

(a) Bisection Method Visualization



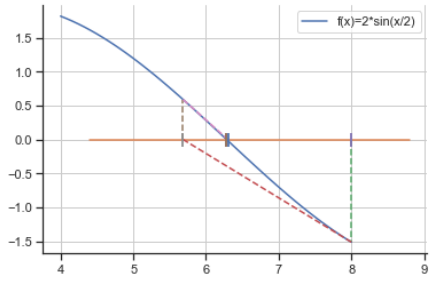

(b) Absolute Error-Iteration Relationship



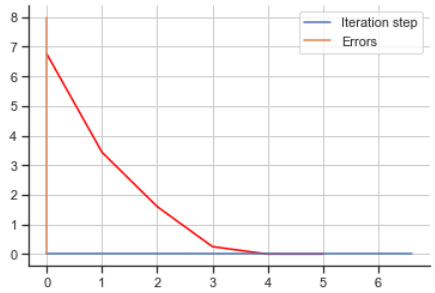

(a) Fixed Point Iteration Visualization



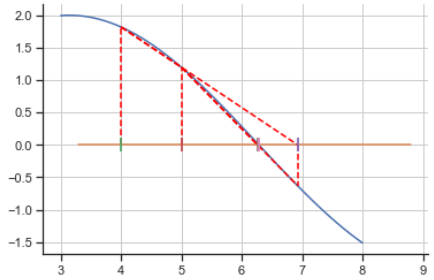

(b) Absolute Error-Iteration Relationship



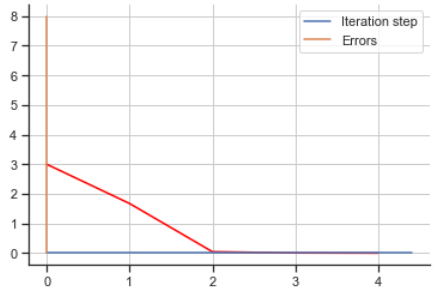

(a) Newtons Method Visualization




(b) Absolute Error-Iteration Relationship



(a) Secant Method Visualization




(b) Absolute Error-Iteration Relationship

# Problem 2

## 0.6 PA=LU Decomposition

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & -1 & 2 & -1 \\ -1 & 2 & -1 & 0 \end{pmatrix}$$

**Column 1** We do not need to interchange any rows. Because 2 is the biggest item of first column.

$$P^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & 0 & 0 & 1 \end{bmatrix}$$

$$L^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\frac{1}{2} & 0 & 0 & 1 \end{bmatrix} M^{(1)}P^{(1)}A = U^{(1)} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & -1 & 2 & -1 \\ 0 & \frac{3}{2} & -1 & 0 \end{bmatrix}$$

**Column 2** We need to interchange second and fourth rows. Because biggest value is $\frac{3}{2}$ in [2,4] rows interval.

$$P^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} M^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{3}{2} & 1 & 0 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

$$L^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & -\frac{2}{3} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M^{(2)}P^{(2)}U^{(1)} = U^{(2)} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & 0 & \frac{4}{3} & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

**Column 3** We do not need to interchange any rows. Because $\frac{4}{3}$ is the biggest item of third column.

$$P^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{3}{4} & 1 \end{bmatrix}$$

$$L^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & -\frac{2}{3} & 1 & 0 \\ 0 & 0 & -\frac{3}{4} & 1 \end{bmatrix} U = M^{(3)}P^{(3)}U^{(2)} = U^{(3)} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & 0 & \frac{4}{3} & -1 \\ 0 & 0 & 0 & \frac{1}{4} \end{bmatrix}$$

**Final P Form**

$$P = P^{(3)}P^{(2)}P^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

**PA=LU Decomposed Form**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & -1 & 2 & -1 \\ -1 & 2 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & -\frac{2}{3} & 1 & 0 \\ 0 & 0 & -\frac{3}{4} & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & 0 & \frac{4}{3} & -1 \\ 0 & 0 & 0 & \frac{1}{4} \end{bmatrix}$$

## 0.7 Finding X

To solve LU Decomposed Linear Systems, Forward and Backward Substitution should be performed.

### 0.7.1 Forward Substitution (Ly=Pb)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 0 & 0 \\ 0 & -\frac{2}{3} & 1 & 0 \\ 0 & 0 & -\frac{3}{4} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

If we transform this linear equation to the simple equation, We can solve for y values easily.

$$y_1 = 1$$
$$-\frac{1}{2}y_1 + y_2 = 0 \rightarrow y_2 = \frac{1}{2}$$
$$-\frac{2}{3}y_2 + y_3 = 0 \rightarrow y_3 = \frac{1}{3}$$
$$-\frac{3}{4}y_3 + y_4 = 0 \rightarrow y_4 = \frac{1}{4}$$

$$y = \begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{3} \\ \frac{1}{4} \end{bmatrix}$$

### 0.7.2 Backward Substitution (Ux=y)

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ 0 & \frac{3}{2} & -1 & 0 \\ 0 & 0 & \frac{4}{3} & -1 \\ 0 & 0 & 0 & \frac{1}{4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ \frac{1}{2} \\ \frac{1}{3} \\ \frac{1}{4} \end{bmatrix}$$

If we transform this linear equation to the simple equation from backward, We can solve for x values easily.

$$\begin{aligned} \frac{1}{4}x_4 = \frac{1}{4} &\rightarrow x_4 = 1 \\ \frac{4}{3}x_3 - x_4 = \frac{1}{3} &\rightarrow x_3 = 1 \\ \frac{3}{2}x_2 - x_3 = \frac{1}{2} &\rightarrow x_2 = 1 \\ 2x_1 - x_2 = 0 &\rightarrow x_1 = 1 \end{aligned}$$

The solution of the linear equation is:

$$x = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

## Problem 3

In this problem we are expected to find most influential 10 papers. To find them, I used PageRank algorithm which is based on Power Method. Implementation steps are given below.

1. File which contains citations is preprocessed by deleting redundant lines.
2. File is read by genfromtxt and it gives edge list.
3. Adjacency matrix is constructed.
4. While reading values from edge list, I give Id for indexing them from 0 to 27770, I record citation number of each paper and I determine papers which do not contain citation to any paper.
5. Adjacency matrix is filled from edge list by 1 over citation number. This is the main idea of the algorithm. Increase in citation number of paper causes decrease in worth of the citation.
6. The columns which have only zeros are filled by 1/27770. Because, reader can read another paper from the database with same probability.
7. I implemented power method which takes one initial guess and it predicts the importance of each paper which based on eigenvalue principles.
8. I gave same initial guesses for every paper which is 1/27770.
9. I iterate the method for 20 times.
10. I printed paper numbers of the biggest 10 values in the vector.

Listing 39: Python Implementation of Algorithm

```
import numpy as np
import pandas as pd
#File reading
data = np.genfromtxt('cit-HepTh.txt', delimiter = "\t", dtype="int")

#Nodes: 27770 Edges: 352807

#Adjacency Matrix
adjacency_matrix = np.zeros((27770,27770), dtype=float)

#List for citation numbers and 0 citation checking
zero_citation_check=[True for i in range(27770)]
citation_number=[0 for i in range(27770)]

#For indexing papers from 0 to 27770
id_dict = {}
```

```
name_dict={}
#Current index which can be given
dictionary_counter=0

#Reading Data
for i in range(352807):
    #From and to paper numbers
    frm = data[i,0]
    to =  data[i,1]
    #Check for already assigned indices
    if frm not in id_dict:
        #Assigning index id
        id_dict.update( {frm: dictionary_counter} )
        name_dict.update({dictionary_counter: frm})
        dictionary_counter+=1

    if to not in id_dict:
        #Assigning index id
        id_dict.update({to: dictionary_counter})
        name_dict.update({dictionary_counter: to})
        dictionary_counter+=1

    from_id = id_dict[frm]
    to_id = id_dict[to]

    #Incrementing cite number of paper
    citation_number[from_id]+=1
    zero_citation_check[from_id]=False

#Filling adjacency matrix with proper values
for i in range(352807):
    frm = data[i,0]
    to =  data[i,1]

    from_id = id_dict[frm]
    to_id = id_dict[to]

    adjacency_matrix[to_id,from_id]=1/citation_number[from_id]

filler = 1/27770

#Fiiling columns which contain full zeros
for i in range(len(zero_citation_check)-1):
    if zero_citation_check[i]==True:
        adjacency_matrix[:,i]=filler

#Initial guess
```

```
vector = np.full((27770,1), filler)

#Power Method
for i in range(20):
    vector = np.matmul(adjacency_matrix,vector)

#For finding biggest 10 values
a = vector.transpose()
a= a[0,:]
lst = pd.Series(a)
i = lst.nlargest(10)

res = i.index.values.tolist()
#Printing paper number of corresponding 10 papers
for i in res:
    print(name_dict[i])
```

**Most Influential 10 Papers**

1. Noncompact Symmetries in String Theory (9207016)
2. An Algorithm to Generate Classical Solutions for String Effective Action (9201015)
3. Monopole Condensation, And Confinement In N=2 Supersymmetric Yang-Mills Theory (9407087)
4. String Theory Dynamics In Various Dimensions (9503124)
5. Exact Results on the Space of Vacua of Four Dimensional SUSY Gauge Theories (9402044)
6. Dirichlet-Branes and Ramond-Ramond Charges (9510017)
7. Unity of Superstring Dualities (9410167)
8. Strong-Weak Coupling Duality in Four Dimensional String Theory (9402002)
9. Supersymmetry as a Cosmic Censor (9205027)
10. One-Loop Threshold Effects in String Unification (9205068)

## 0.8 Most Influential Twitter Accounts

Nowadays,Twitter follower bots become popular and follower numbers are getting insignificant factor for account importance evaluation. I decided to find more important Twitter accounts using algorithm. I compared accounts of four scientists.

1. Roger Freedman
2. Chris Hadfield
3. Michio Kaku
4. Bill Nye

If I transform their connections to adjacency matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

I suggest that Bill Nye will be chosen as most influential account. Algorithm is given below.
1. I fill adjacency matrix by 1 over number of follows.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 & 0 \end{bmatrix}$$

2. I fill zero column with 1/4.

$$\begin{bmatrix} 0 & 0 & 0 & \frac{1}{4} \\ \frac{1}{2} & 0 & 0 & \frac{1}{4} \\ 0 & \frac{1}{2} & 0 & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{2} & 1 & \frac{1}{4} \end{bmatrix}$$

3. I give initial guess as 1/4 for all accounts and make 2 iterations.

$$\begin{bmatrix} \frac{1}{16} \\ \frac{3}{16} \\ \frac{3}{16} \\ \frac{1}{2} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \frac{1}{4} \\ \frac{1}{2} & 0 & 0 & \frac{1}{4} \\ 0 & \frac{1}{2} & 0 & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{2} & 1 & \frac{1}{4} \end{bmatrix} \begin{bmatrix} \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \\ \frac{1}{4} \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{8} \\ \frac{5}{32} \\ \frac{7}{32} \\ \frac{13}{32} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & \frac{1}{4} \\ \frac{1}{2} & 0 & 0 & \frac{1}{4} \\ 0 & \frac{1}{2} & 0 & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{2} & 1 & \frac{1}{4} \end{bmatrix} \begin{bmatrix} \frac{1}{16} \\ \frac{3}{16} \\ \frac{3}{16} \\ \frac{1}{2} \end{bmatrix}$$

As we see in the result the biggest value is in row 4 and it corresponds to Bill Nye. My guess and mathematical result are same.