

Report on Chapter 2—Solutions of Equations in One Variable

Abstract

In this report, we briefly introduce three algorithms about finding zeros of functions and use these algorithm to find zeros of several functions.

Contents

1	Introduction on Algorithms	1
1.1	Newton's Method	1
1.1.1	Motivation	1
1.1.2	Algorithm	2
1.2	Steffensen's Method	2
1.2.1	Motivation	2
1.2.2	Algorithm	3
1.3	Müller's Method	3
1.3.1	Motivation	3
1.3.2	Algorithm	5
2	Codes and Numerical Experiments	6
2.1	Newton's Method	6
2.2	Steffensen's Method	9
2.3	Müller's Method	13
3	Discussions and Conclusions	20
A	Table of convergence of the sequence generated by bisection method	21

1 Introduction on Algorithms

1.1 Newton's Method

1.1.1 Motivation

Suppose that $f \in C^2[a, b]$ and p is a root, $\bar{x} \in [a, b]$ to be an approximation to p such that $f'(\bar{x}) \neq 0$ and $|\bar{x} - p|$ is small enough. Consider the first Taylor polynomial for $f(x)$ expanded about \bar{x} :

$$f(x) = f(\bar{x}) + (x - \bar{x})f'(\bar{x}) + \frac{(x - \bar{x})^2}{2}f''(\xi(x)).$$

where $\xi(x)$ lies between x and \bar{x} . Since $f(p) = 0$, this equation, with $x = p$, gives

$$0 = f(p) = f(\bar{x}) + (p - \bar{x})f'(\bar{x}) + \frac{(p - \bar{x})^2}{2}f''(\xi(p)).$$

Newton's method is then derived by assuming that $|p - \bar{x}|$ is small, so the term involving $(p - \bar{x})^2$ is much smaller and that

$$0 = f(p) \approx f(\bar{x}) + (p - \bar{x})f'(\bar{x}),$$

Solving for p in this equation gives

$$p \approx \bar{x} - \frac{f(\bar{x})}{f'(\bar{x})}.$$

The formula above generates a sequence $\{p_n\}$ by defining $p_n = p_{n-1} - \frac{f(p_{n-1})}{f'(p_{n-1})}$, $\forall n \geq 1$. We hope this sequence to converge to the root p .

1.1.2 Algorithm

INPUT: initial approximation p_0 ; tolerance TOL ; number of iterations N_0 .

OUTPUT: approximate solution p or message of failure.

Step 1 Let $i = 1$.

Step 2 While $i \leq N_0$, do step 3-5.

Step 3 Set $p = p_0 - f(p_0)/f'(p_0)$. (Compute p_i)

Step 4 If $|p - p_0| < TOL$ then OUTPUT (p); (Procedure completes.) STOP.

Step 5 Set $i = i + 1$, $p_0 = p$, goto Step 2.

Step 6 OUTPUT (Method failed after N_0 iterations, ' $N_0 = '$, N_0); (Procedure completes.) STOP.

The following theorem shows that if the initial approximation p_0 is chosen to be sufficiently close to the actual root, then our iterative sequence will converge.

Theorem 1.1.1. Let $f \in C^2[a, b]$ and $p \in (a, b)$ such that $f(p) = 0$, $f'(p) \neq 0$, then there exists a $\delta > 0$ such that the sequence $\{p_n\}$ generated above converges to p for any $p_0 \in [p - \delta, p + \delta]$.

However, the theorem above just shows the existence of such δ without telling us how to determine it. In practical application, we can use other algorithms (e.g. Bisection Method) to get a great initial approximation to make our sequence converge.

1.2 Steffensen's Method

1.2.1 Motivation

For a fixed iteration problem: $p = g(p)$, given initial approximation p_0 , let $p_1 = g(p_0)$, $p_2 = g(p_1)$, and then $\hat{p}_0 = p_0 - (p_1 - p_0)^2 / (p_2 - 2p_1 + p_0)$. Assume that \hat{p}_0 is a better approximation than p_2 , and apply fixed point iteration to \hat{p}_0 instead of p_2 .

1.2.2 Algorithm

INPUT: initial approximation p_0 ; tolerance TOL; maximum number of iterations N_0 .

OUTPUT: approximate solution p , or message of failure.

Step 1 Set $i=1$.

Step 2 While $i \leq N_0$, do Step 3-6.

Step 3 Set

$$\begin{aligned} p_1 &= g(p_0), p_2 = g(p_1), \\ p &= p_0 - (p_1 - p_0)^2 / (p_2 - 2p_1 + p_0). \end{aligned}$$

Step 4 If $|p - p_0| < TOL$, then OUTPUT p , STOP.

Step 5 Set $i = i + 1$.

Step 6 Set $p_0 = p$.

Step 7 OUTPUT (Method failed after N_0 iterations, 'N₀ =', N_0), STOP.

The procedure is described as follow:

k	$p_0^{(k)}$	$p_1^{(k)}$	$p_2^{(k)}$
0	$p_0^{(0)}$	$p_1^{(0)} = g(p_0^{(0)})$	$p_2^{(0)} = g(p_1^{(0)})$
1	$p_0^{(1)} = p_0^{(0)} - \frac{(p_1^{(0)} - p_0^{(0)})^2}{p_2^{(0)} - 2p_1^{(0)} + p_0^{(0)}}$	$p_1^{(1)} = g(p_0^{(1)})$	$p_2^{(1)} = g(p_1^{(1)})$
2	$p_0^{(2)} = p_0^{(1)} - \frac{(p_1^{(1)} - p_0^{(1)})^2}{p_2^{(1)} - 2p_1^{(1)} + p_0^{(1)}}$	$p_1^{(2)} = g(p_0^{(2)})$	$p_2^{(2)} = g(p_1^{(2)})$
\vdots	\vdots	\vdots	\vdots

The following theorem guarantees our iterative sequence will converge.

Theorem 1.2.1. Suppose that $x = g(x)$ has the solution p with $g'(p) \neq 1$. If there exists a $\delta > 0$ such that $g \in C^3[p - \delta, p + \delta]$, then Steffensen's method gives a quadratic convergencing sequence for any $p_0 \in [p - \delta, p + \delta]$.

1.3 Müller's Method

1.3.1 Motivation

Müller's method is firstly presented by D.E.Müller in 1956, and can be thought as an extension of the Secant method. Using three initial approximations, x_0, x_1 and x_2 , it determines the

next approximation x_3 by considering the intersection of the x-axis with the parabola through $(x_0, f(x_0))$, $(x_1, f(x_1))$ and $(x_2, f(x_2))$. It is obvious that three point can only determine one quadratic polynomial $P(x)$. Suppose that $P(x)$ has the form

$$P(x) = a(x - x_2)^2 + b(x - x_2) + c$$

that passes through $(x_0, f(x_0))$, $(x_1, f(x_1))$ and $(x_2, f(x_2))$. Then we have

$$\begin{cases} f(x_0) = a(x_0 - x_2)^2 + b(x_0 - x_2) + c, \\ f(x_1) = a(x_1 - x_2)^2 + b(x_1 - x_2) + c, \\ f(x_2) = a \times 0 + b \times 0 + c = c, \end{cases}$$

By solving this equations, we can get coefficients a, b, c of $P(x)$.

$$\begin{aligned} a &= \frac{\frac{f(x_0)-f(x_2)}{x_0-x_2} - \frac{f(x_1)-f(x_2)}{x_1-x_2}}{x_0 - x_1}, \\ &= \frac{\frac{f(x_0)-f(x_1)+f(x_1)-f(x_2)}{x_0-x_2} - \frac{f(x_1)-f(x_2)}{x_1-x_2}}{x_0 - x_1} \\ &= \frac{\frac{f(x_0)-f(x_1)}{x_0-x_2} + (\frac{1}{x_0-x_2} - \frac{1}{x_1-x_2})(f(x_1) - f(x_2))}{x_0 - x_1} \\ &= \frac{\frac{x_0-x_1}{x_0-x_2} \frac{f(x_1)-f(x_0)}{x_1-x_0} + \frac{x_1-x_0}{x_0-x_2} \frac{f(x_2)-f(x_1)}{x_2-x_1}}{x_0 - x_1} \\ &= \frac{\frac{f(x_2)-f(x_1)}{x_2-x_1} - \frac{f(x_1)-f(x_0)}{x_1-x_0}}{x_2 - x_0} \\ b &= \frac{f(x_2) - f(x_1)}{x_2 - x_1} + (x_2 - x_1)a, \\ c &= f(x_2). \end{aligned}$$

To determine the intersection x_3 , or a zero of quadratic polynomial $P(x)$, we apply the quadratic formula to $P(x) = 0$, and get

$$\begin{aligned} x - x_2 &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ &= \frac{(-b \pm \sqrt{b^2 - 4ac})(-b \mp \sqrt{b^2 - 4ac})}{2a(-b \mp \sqrt{b^2 - 4ac})}. \end{aligned}$$

thus

$$x - x_2 = \frac{-2c}{b \pm \sqrt{b^2 - 4ac}}$$

Let $x = x_3$, thus above formula gives two solutions or possibilities for the approximation x_3 . In Müller's method, the sign is chosen to agree with the sign of b .

$$x_3 = x_2 - \frac{2c}{b + \text{sign}(b)\sqrt{b^2 - 4ac}}$$

Chosen in this manner, the denominator will be the largest in magnitude and will result in x_3 being selected as the closest zero of P to x_2 .

Once x_3 is determined, the procedure is reinitialized using x_1, x_2, x_3 in place of x_0, x_1, x_2 to determine next approximation x_4 . The method continues until satisfactory conclusion is obtained.

In each step, the method involves the radical $\sqrt{b^2 - 4ac}$, so the method gives approximate complex roots when $b^2 - 4ac < 0$.

1.3.2 Algorithm

INPUT: x_0, x_1, x_2 ; tolerance TOL ; maximum number of iterations N_0 .

OUTPUT: approximate solution p , or message of failure.

Step 1 Set

$$\begin{aligned} h_1 &= x_1 - x_0, h_2 = x_2 - x_1; \\ \delta_1 &= (f(x_1) - f(x_0))/h_1, \delta_2 = (f(x_2) - f(x_1))/h_2; \\ a &= (\delta_2 - \delta_1)/(h_2 + h_1), \\ i &= 3. \end{aligned}$$

Step 2 While $i \leq N_0$, do Step 3-7.

Step 3 $b = \delta_2 + h_2a, d = (b^2 - 4 * a * f(x_2))^{1/2}$. (Note: maybe complex arithmetic.)

Step 4 If $|b - d| < |b + d|$, then $e = b + d$, else $e = b - d$.

Step 5 Let $h = -2f(x_2)/e; p = x_2 + h$.

Step 6 If $|h| < TOL$, then OUTPUT p (Procedure completed successfully), STOP.

Step 7 Set (To prepare next iteration)

$$\begin{aligned} x_0 &= x_1, x_1 = x_2, x_2 = p; \\ h_1 &= x_1 - x_0, h_2 = x_2 - x_1; \\ \delta_1 &= (f(x_1) - f(x_0))/h_1, \delta_2 = (f(x_2) - f(x_1))/h_2; \\ a &= (\delta_2 - \delta_1)/(h_2 + h_1), \\ i &= i + 1. \end{aligned}$$

Step 8 OUTPUT (Method failed after N_0 iterations, ' $N_0 =$ ', N_0), STOP.

2 Codes and Numerical Experiments

2.1 Newton's Method

In this section, we firstly use Newton's method to calculate the root of $f(x) = e^{3x} - x^3 - 2$ between 0 and 1. Setting $N_0 = 100$, $TOL = 10^{-6}$ and $p_0 = 1$. The Matlab code is as follow:

```
1 clear
2 clc
3 format long
4 N0=100;%maximal number of iterations
5 TOL=1e-6;%tolerence of error
6 p0=1;%initial value
7 for i=1:N0+1
8     if i<=N0
9         p=p0-f(p0)/df(p0);%next value
10        if abs(p-p0)<TOL%reach the tolerance of error
11            disp(['The root is ',num2str(p)])%output the root
12            break%break the loop
13        else
14            p0=p;%set new p0
15        end
16    else
17        disp(['Method failed after ',num2str(N0),' iterations'])
18        %the method failed
19    end
20 end
```

```

21 function y=f(x)
22 y=exp(3*x)-x^3-2;
23 end
24 function y=df(x)%the derivative of f
25 y=3*exp(3*x)-3*x^2;
26 end

```

The procedure completes after 7 iterations,the process is as follow:

k	x_k	$f(x_k)$
0	1.000000000000000	17.085536923187668
1	0.701597130994171	17.085536923187668
2	0.448348414442025	5.860037985503131
3	0.288136741027671	1.748234857143000
4	0.237249570606668	0.349683861484862
5	0.233178586157927	0.024197083286938
6	0.233154823825949	0.000139611718049
7	0.233154823022655	0
\vdots	\vdots	\vdots

So the root is about 0.233155.

Now,to show the efficiency of Newton's method,we use bisection method to calculate the root of the equation and compare its efficiency with Newton's method's.Set $N_0 = 100$, $TOL = 10^{-6}$ and initial interval $[a, b] = [0, 1]$.

```

1 clear
2 clc
3 format long

```

```

4  a=0;

5  b=1;%initial interval

6  N0=100;%maximal number of iterations

7  TOL=1e-6;%tolerence of error

8  for i=1:N0+1

9  if i<=N0

10     c=(a+b)/2;%set the mid point

11     if abs(c-a)<TOL%reach the tolerence of error

12         disp(['the root is ',num2str(c)])%output the root

13         break%break the loop

14     else if f(c)>0

15         b=c;

16     else

17         a=c;

18     end%set new interval

19 end

20 else

21     disp(['Method failed after ',num2str(N0),' iterations'])

22     %the method failed

23 end

24 end

25

26 function y=f(x)

27 y=exp(3*x)-x^3-2;

28 end

```


We see that bisection method need 20 iterations to get the same accuracy as Newton's Method. The table of the sequence generated by bisection method is too long to be put here, I will show x_1, x_2, \dots, x_8 here and leave the whole table to the appendix. We can see from the table of x_1, x_2, \dots, x_8 that after 8 iterations, the result of bisection method is not accurate at all, while the result of Newton's method is a lot more accurate.

k	x_k	$f(x_k)$
1	0.5000000000000000	2.356689070338065
2	0.2500000000000000	0.101375016612675
3	0.1250000000000000	-0.546961710381799
4	0.1875000000000000	-0.251537139914702
5	0.2187500000000000	-0.082917079129330
6	0.2343750000000000	0.007180924437212
7	0.2265625000000000	-0.038368589641581
8	0.2304687500000000	-0.015720374378063
\vdots	\vdots	\vdots

2.2 Steffensen's Method

In this section, we will use fixed-point iteration method and Steffensen's method to calculate the fixed point of $g(x) = \frac{x + \sin(x) - x^2 + 1}{2}$ and compare the efficiency of them by comparing their times of iterations. Firstly, we use fixed-point iteration method to calculate the fixed point of $g(x)$. Setting $N_0 = 100$, $TOL = 10^{-6}$ and $p_0 = 1$, the Matlab code is as follows:

```

1 clear
2 clc
3 format long
4 N0=100;%maximal number of iterations
5 TOL=1e-6;%tolerance of error

```

```

6  p0=1;%initial approximation
7  for i=1:N0+1
8      if i<=N0
9          p=g(p0);%new point
10         if abs(p-p0)<TOL%reach the tollerance of error
11             disp(['the root is ',num2str(p)])%output the root
12             break%break the loop
13         else
14             p0=p;%set p to be the new initial approximation
15         end
16     else
17         disp(['Method failed after ',num2str(N0),' iterations'])
18         %the method failed
19     end
20 end
21
22
23 function y=g(x)
24 y=(sin(x)-x^2+1)/2;
25 end

```

The procedure completes after 9 iterations,the process is as follow:

k	x_k	$g(x_k)$
0	1.0000000000000000	0.420735492403948
1	0.420735492403948	0.615706779060772
2	0.615706779060772	0.599220409359827
3	0.599220409359827	0.602466889481239
4	0.602466889481239	0.601855206039177
5	0.601855206039177	0.601971489703701
6	0.601971489703701	0.601949420636978
7	0.601949420636978	0.601953610381290
8	0.601953610381290	0.601952815019303
9	0.601952815019303	0.601952966008934
\vdots	\vdots	\vdots

Secondly, we use Steffensen's method to accelerate the fixed point iteration. Setting $N_0 = 100$, $TOL = 10^{-6}$ and $p_0 = 1$, the Matlab code is as follow:

```

1  clear
2  clc
3  format long
4  N0=100;%maximal number of iterations
5  TOL=1e-6;%tolerance of error
6  p0=1;%initial approximation
7  for i=1:N0+1
8      if i<=N0
9          p1=g(p0);
10         p2=g(p1);

```

```

11     p = p0-(p1-p0)^2/(p2-2*p1+p0);%new p
12     if abs(p-p0)<TOL%reach the tollerance of error
13         disp(['the fixed point is ',num2str(p)])
14         %output the fixed point
15         break%break the loop
16     else
17         p0=p;%set p to be the new initial approximation
18     end
19 else
20     disp(['Method failed after ',num2str(N0)', ' iterations'])
21     %the method failed
22 end
23 end
24
25
26 function y=g(x)
27 y=(sin(x)-x^2+1)/2;
28 end

```

The procedure completes after 4 iterations,the process is as follow:

k	x_k	$g(x_k)$
0	1.0000000000000000	0.420735492403948
1	0.566608296527683	0.607864255093552
2	0.601838838004853	0.601974594742265
3	0.601952940586461	0.601952942171617
4	0.601952941918707	0.601952941918707
\vdots	\vdots	\vdots

Both of methods above provide the fixed point is about 0.601953. But we see that Steffensen's method completed after only 4 iterations while the fixed-point method completed after 9 iterations. Hence we can conclude that our accelerating method worked.

2.3 Müller's Method

In this section, we use Müller's method to calculate all roots of the polynomial function $f(x) = 2x^5 - 5x^4 + 2x - 3$. Setting $N_0 = 100$, $TOL = 10^{-10}$.

Firstly, we set the initial value $x_0 = 0$, $x_1 = 1$, $x_2 = 2$, and calculate one root of $f(x)$ to show that Müller's method is rapid. The Matlab code is as follow:

```

1 clear
2 clc
3 format long
4 N0=100;%maximal number of iterations
5 TOL=1e-10;%tolerance of error
6 x0=0;
7 x1=1;
8 x2=2;%initial value
9 h1=x1-x0;
10 h2=x2-x1;
```

```

11 delta1=(f(x1)-f(x0))/h1;
12 delta2=(f(x2)-f(x1))/h2;
13 a=(delta2-delta1)/(h2+h1);
14 for i=3:N0+1
15     if i<=N0
16         b=delta2+h2*a;
17         d=sqrt(b^2-4*a*f(x2));
18         if abs(b-d)<abs(b+d)
19             e=b+d;
20         else
21             e=b-d;
22         end
23         h=-2*f(x2)/e;
24         p=x2+h;%set new point
25         if abs(h)<TOL%reach the tolerance of error
26             disp(['the root is ',num2str(p)])%output the root
27             break%break the loop
28         else
29             x0=x1;
30             x1=x2;
31             x2=p;
32             h1=x1-x0;
33             h2=x2-x1;
34             delta1=(f(x1)-f(x0))/h1;
35             delta2=(f(x2)-f(x1))/h2;

```

```

36         a=(delta2-delta1)/(h2+h1);%set new value
37     end
38 else
39     disp(['Method failed after ',num2str(N0),' iterations'])
40     %the method failed
41 end
42 end
43 function y=f(x)
44 y=2*x^5-5*x^4+2*x-3;
45 end

```

The procedure completes after 8 iterations,the process is as follow:

k	x_k	$f(x_k)$
3	0.4000000000000000 - 0.663324958071080i	-0.9523200000000001 - 2.305186894288617i
4	0.668244172140612 - 0.713573073154541i	1.824615451168063 - 0.572422538132202i
5	0.587608740024608 - 0.535773339902369i	-0.399957712802112 - 0.369883074284750i
5	0.641548360446264 - 0.560034273086612i	0.009014006032035 - 0.029256712215316i
6	0.643979636713453 - 0.558047752696774i	0.000242292643412 - 0.000269235277105i
7	0.643996032877509 - 0.558013934264273i	0.000000009844697 - 0.000000080032884i
8	0.643996040186258 - 0.558013930187542i	0.000000000000010 - 0.000000000000006i
9	0.643996040186258 - 0.558013930187541i	-0.000000000000000 + 0.000000000000000i
\vdots	\vdots	\vdots

So,the root is about 0.6439960402 - 0.55801393019i.And the method only needs 9 iterations to get the accuracy of 10^{-10} .Hence we conclude that the method is rapid.

Now,to find all zeros of $f(x)$,we package Müller's method as a function of the coefficients of the polynomial and the initial value x_0, x_1, x_2 .The input is c_0 and $x = [x_0, x_1, x_2]$,the output is

the corresponding root of the input. The Matlab code is as follow:

```
1 function p=mul(c,x)
2 format long
3 N0=100;%maximal number of iterations
4 TOL=1e-10;%tolerance of error
5 x0=x(1);
6 x1=x(2);
7 x2=x(3);%initial value
8 h1=x1-x0;
9 h2=x2-x1;
10 delta1=(f(x1)-f(x0))/h1;
11 delta2=(f(x2)-f(x1))/h2;
12 a=(delta2-delta1)/(h2+h1);
13 for i=3:N0+1
14     if i<=N0
15         b=delta2+h2*a;
16         d=sqrt(b^2-4*a*f(x2));
17         if abs(b-d)<abs(b+d)
18             e=b+d;
19         else
20             e=b-d;
21         end
22         h=-2*f(x2)/e;
23         p=x2+h;
24         if abs(h)<TOL%reach the tolerance of error
```



```

25         break%break the loop
26     else
27         x0=x1;
28         x1=x2;
29         x2=p;
30         h1=x1-x0;
31         h2=x2-x1;
32         delta1=(f(x1)-f(x0))/h1;
33         delta2=(f(x2)-f(x1))/h2;
34         a=(delta2-delta1)/(h2+h1);%set new value
35     end
36 else
37     disp(['Method failed after ',num2str(N0),' iterations'])
38     %the method failed
39 end
40 end
41 function y=f(x)%use Horner's method to calculate f(x)
42 n=size(c',1);
43 y=c(1);
44 z=c(1);
45 for k=2:n-1
46     y=x*y+c(k);
47     z=x*z+y;
48 end
49 y=x*y+c(n);

```

```
50 end
```

```
51 end
```

Then we use our function above to calculate all roots of the polynomial using the following procedure:

INPUT: coefficients of the aim polynomial c_0 ;initial value $x = [x_0, x_1, x_2]$.

OUTPUT: all roots of the polynomial x_1, \dots, x_n .

Step 1 Set $n = (\text{degree of } f(x)) + 1$

Step 2 For $t = 1, \dots, n - 1$, do Step 3-5.

Step 3 Set $r_t = \text{mul}(c_0, x_0)$.

Step 4 Set $c(1) = c_0(1)$;

for $k = 2, \dots, n - t$, set $c(k) = c_0(k) + c(k - 1)r_t$. (Use Horner's method to get coefficients of the new polynomial)

Step 5 Set $c_0 = c$;

clear c .

Step 6 OUTPUT r (Procedure completed), STOP.

Setting $x = [0, 1, 2]$, $c_0 = [2, -5, 0, 0, 2, -3]$, the Matlab code of the procedure above is as follow:

```
1 clear
2 clc
3 format long
4 c0=[2,-5,0,0,2,-3];%coefficients of polynomial
5 x=[0,1,2];%initial value
6 n=size(c0',1);%(the degree of f(x)) +1
7 for t=1:n-1
8     r(t)=mul(c0,x);%calculate the root
9     c(1)=c0(1);
10 for k=2:n-t
```

```

11      c(k)=c0(k)+c(k-1)*r(t);
12  end
13  c0=c;%set new coefficients
14  clear c
15  end
16  disp(['all roots are ',num2str(r)])%output all roots

```

Then we get that all roots of $f(x)$ are as follow:

x_1	x_2	x_3	x_4	x_5
2.474001	0.643996 - 0.558014i	0.643996+0.558014i	-0.630996+0.660945i	-0.630996-0.660945i

3 Discussions and Conclusions

Theoretically, Newton's method, which is quadratically convergent, is the most efficient in this three algorithm. However, it needs numerical derivative of the function, which is difficult to get. At the same time, the initial approximation is tremendously important in Newton's method. If the initial approximation is not close enough to the root, the iterative sequence will rapidly diverge. For this reason, we have to choose a good initial value. The bisection method does help to find a good initial value.

Steffensen's method is derived from the trick of accelerating convergence and helps to accelerate the fixed-point iteration algorithm.

Müller's method, which is not so rapid as Newton's method, is the most convenient one in this three method. It will converge to a root of the equation no matter what initial value we use, so there's no need for us to choose the initial value carefully.

Appendix

A Table of convergence of the sequence generated by bisection method

k	x_k	$f(x_k)$
1	0.5000000000000000	2.356689070338065
2	0.2500000000000000	0.101375016612675
3	0.1250000000000000	-0.546961710381799
4	0.1875000000000000	-0.251537139914702
5	0.2187500000000000	-0.082917079129330
6	0.2343750000000000	0.007180924437212
7	0.2265625000000000	-0.038368589641581
8	0.2304687500000000	-0.015720374378063
9	0.2324218750000000	-0.004301539174307
10	0.2333984375000000	0.001431716606418
11	0.2329101562500000	-0.001436902475161
12	0.2331542968750000	-0.000003091083694
13	0.2332763671875000	0.000714188180060
14	0.2332153320312500	0.000355517408355
15	0.2331848144531250	0.000176205378061
16	0.2331695556640630	0.000086555201202
17	0.2331619262695310	0.000041731572269
18	0.2331581115722660	0.000019320122668
19	0.2331562042236330	0.000008114489082
20	0.2331552505493160	0.000002511695093
\vdots	\vdots	\vdots

References

- [1] Richard L. Burden, J. Douglas Faires, Annette M. Burden, Numerical Analysis (Tenth Edition), Cengage Learning, 2014.