EE1103: Numerical Methods

Programming assignment # 2

Lakshmiram S - EE22B117

November 22, 2022

Contents

List of Figures

1 Problem 1: deflection in a rod

Given is a uniform beam subject to a linearly increasing distributed load. The equation for the resulting elastic curve is

$$y = \frac{w_0}{120EIL}(-x^5 + 2L^2x^3 - L^4x) \tag{1}$$

Use bisection to determine the point of maximum deflection (that is, the value of x where $\frac{dy}{dx} = 0$). Then substitute this value into Eq. 1 to determine the value of the maximum deflection. Use the following parameter values in your computation:
L = 600 cm,
E = 50,000 kN/cm^2,
I = 30,000 cm^4,
w_0 = 2.5 kN/cm.

1.1 Approach

Bisection method:

- Bisection is a popular bracketing method for root finding.The core idea is to first pick a bracket (a lower value say x_l and and an upper value x_u) manually such that the bracket encloses the root. This can be done by first plotting the function on a graphing software and later roughly estimating where the root lies. if $f(x_u)f(x_l) < 0$, then we have a desirable bracket.

- Now we devise an algorithm to shrink the bracket (halve it to be precise) repeatedly while still ensuring that the root lies inside it.To accomplish this, we define a parameter $x_r = \frac{x_l + x_u}{2}$. Now we test which half of the two contain the root. if $f(x_l)f(x_r) < 0$,then the root lies between x_l and x_r.By doing this , we just halved our interval while still flanking the root on either sides.We can recursively apply the the process for the new interval x_l to x_r.By a similar reasoning, the root can be found on the other half if $f(x_r)f(x_u) < 0$ and the same logic as the previous case holds here.

- Well, we should stop executing the iterations at some point. A good indication to whether we have reached satisfactory accuracy or not is the true error percentage that can be found out using the expression:

$$true\ error = |\frac{x_r - actual\ root}{x_r}| * 100\% \tag{2}$$

  But, this is not that useful because it's evaluation requires the actual root which is what all this hustle is for! So we introduce the notion of relative error:

$$relative\ error = |\frac{x_r new - x_r old}{x_r new}| * 100\% \tag{3}$$

  We know for a fact that the relative error is always greater than the actual error in magnitude and also that the bisection method guarantees the convergence of the relative error, which implies the convergence of the true error. We allow for a maximum tolerance of about 0.05% relative error in the entire process.

1.2 Algorithm

define a function that returns f(x)
define f'(x)
funcBISECT(xL,xu,tolerance)\\
iter=0,xr=(xl+xu)/2,error=|(xu-xl)/(2*xr)|*100
While{error<tolerance}{
if f(xr)=0
break
else if f(xr)*f(xl)<0}
xu=xr
xr=(xl+xu)/2
iter++
error=|(xu-xl)/(2*xr)|*100
else
xl=xr
xr=(xl+xu)/2
iter++
error=|(xu-xl)/(2*xr)|*100
return xr

1.3 Code

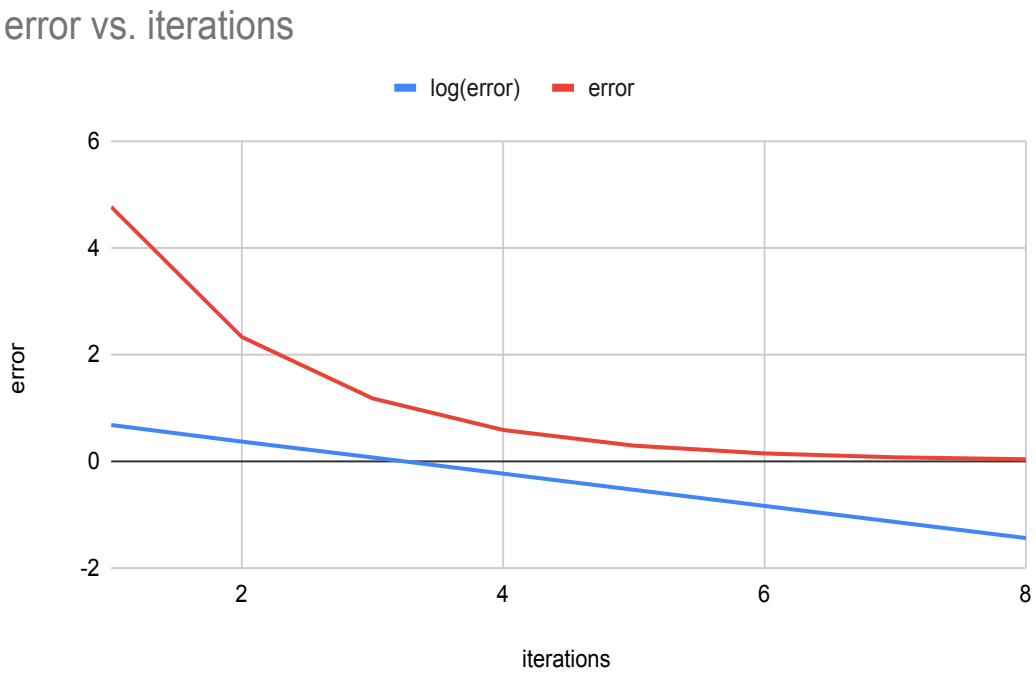
```
#include<stdio.h>
#include<math.h>
//function for function
float function(float x)
{
    float w=2500,e=50000000,i=30000,l=600;
    return (w/(120*e*i*l))*(-pow(x,5)+2*pow(l,2)*pow(x,3)-pow(l,4)*x);
}
//function for derivative
float y(float x)
{
    int l=600,e=50000,i=30000,w=2500;
    return w*(-pow(x,4)*5 + 6*pow(l,2)*pow(x,2)-pow(l,4))/(120*e*i*l);
}
float bisect(float xl,float xu,float tolerance)
{

    float xr=(xl+xu)/2;
    float error=fabs((xl-xu)/(2*xr))*100;
    int iter=0;
    while(error>tolerance)
    {
        if(y(xr)==0)
        {
            break;
        }
        else if(y(xr)*y(xl)<0)
        {
```

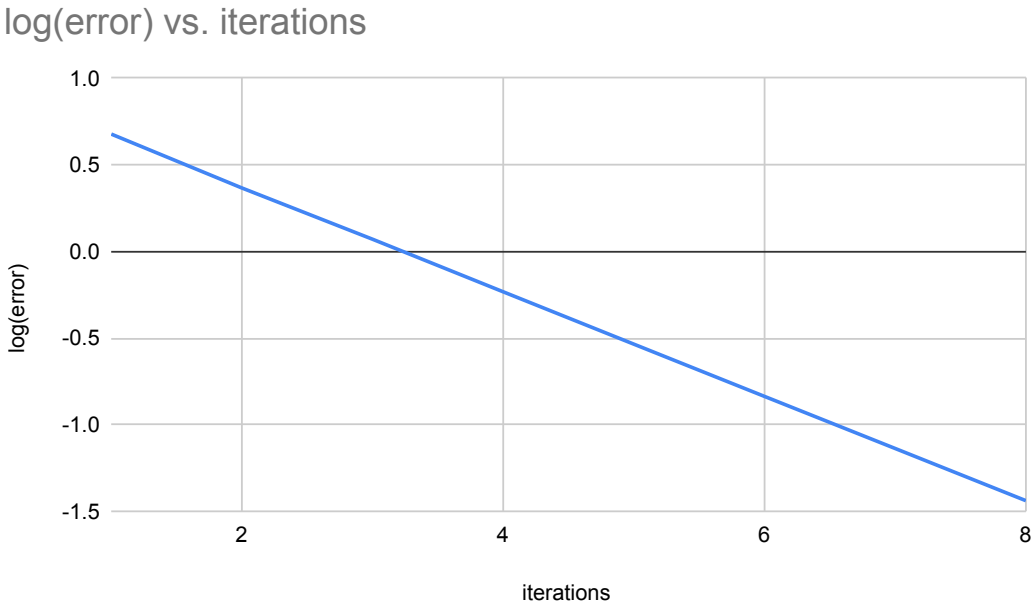
```
            xu=xr;
            xr=(xl+xu)/2;
            iter++;
            error=fabs((xl-xu)/(2*xr))*100;

        }
        else{
            xl=xr;
            xr=(xl+xu)/2;
            iter++;
            error=fabs((xl-xu)/(2*xr))*100;

        }
        printf("iteration %d   error %f\n", iter, error);
    }
    return xr;
}
int  main()
{
    float root=bisect(250, 300, 0.05);
    printf("root is %f\n",root);
    printf("the maximum elongation is: %f",function(root));
}
```

1.4 Results

We observe that the maximum deflection is -0.515190, caused at x = 268.261719 .Bisection exhibits linear convergence.The only advantage of bisection, and in general , bracketing methods is that they guarantee convergence no matter how slow they are.

1.5 Graphs





2 Problem 2: Population estimate

Many fields of engineering require accurate population estimates. For example, transportation engineers might find it necessary to determine separately the population growth trends of a city and adjacent suburb. The population of the urban area is declining with time according to

$$P_u(t) = P_{u,max}e^{-k_u t} + P_{u,min} \tag{4}$$

while the suburban population is growing, as in

$$P_s(t) = \frac{P_{s,max}}{1 + [\frac{P_{s,max}}{P_0} - 1]e^{-k_s t}} \tag{5}$$

where $P_{u,max}$, k_u, $P_{s,max}$, P_0, and k_s are empirically derived parameters. Determine the time and corresponding values of $P_u(t)$ and $P_s(t)$ when the suburbs are 20% larger than the city. The parameter values are
$P_{u,max}$ = 75,000,
k_u = 0.045/yr,
$P_{u,min}$ = 100,000 people,
$P_{s,max}$ = 300,000 people,
P_0 = 10, 000 people,
k_s = 0.08/yr.
To obtain your solution, use the False Position Method.

2.1 Approach

- The false position method is a yet another bracketing approach to root finding.It is very similar to bisection but differs in the way x_r is computed. This method represents an attempt towards making bisection more efficient and is backed upon the philosophy that which ever of the two-$f(x_l)$ or $f(x_u)$ is closer to 0, then the actual root is also supposed to be closer to the corresponding abscissa. This works very well with some functions but it also fails in many cases .
- So here's what we do: we first draw a line connecting $f(x_l)$ and $f(x_u)$ .now we assign the value of x_r as the x coordinate of the point where the line intersects the x-axis.
- Note that the line will always intersect the axis because we have chosen the limits of the bracket in a way so as to have opposite signs.

2.2 Algorithm

```
funcFALSEPOSITION(xl,xu,tolerance)
    iter=0,xr=xu-((y(xu)*(xl-xu))/(y(xl)-y(xu))),error=1000(some number)
    while{error<tolerance}{
        if f(xr)=0
            break
        else if f(xr)*f(xl)<0}
            xu=xr
            xr=xu-((y(xu)*(xl-xu))/(y(xl)-y(xu)));
            iter++
            error=|(xu-xl)/(2*xr)|*100
        else
            xl=xr
            xr=xu-((y(xu)*(xl-xu))/(y(xl)-y(xu)));
            iter++
            error=|(xu-xl)/(2*xr)|*100
    return xr
```

2.3 Code

```
#include <stdio.h>
#include <math.h>
float pu(float t)//function for urban population
{
    //data from question
```

```
    float pu_max=75000,pu_min=100000,ku=0.045;
    return pu_max*exp(-ku*t)+pu_min;
}
float ps(float t)//function for sub-urban population
{
    //data from question
    float ps_max=300000,p0=10000,ks=0.08;
    return ps_max/(1+((ps_max/p0)-1)*exp(-ks*t));
}
float y(float t)//function whose root is to be found
{
    return ps(t)-1.2*pu(t);
}
//fucntion for deploying the false position method
//almost everything about this method is the same as bisection except for the
↪  way we calculate xr
float falsepoint(float xl,float xu,float tolerance)
{

    float xr=xu-((y(xu)*(xl-xu))/(y(xl)-y(xu)));
    float xr_old=xr;
    float error=10;
    int iter=0;
    while(error>tolerance)
    {
        if(y(xr)==0)
        {
            error=0;
            break;
        }
        else if(y(xr)*y(xl)<0)
        {
            xu=xr;
            xr=xu-((y(xu)*(xl-xu))/(y(xl)-y(xu)));
            xr_old=xu;
            float xr_new=xr;
            iter++;
            error=fabs((xr_new-xr_old)/xr_new)*100;

        }
        else{
            xl=xr;
            xr=xu-((y(xu)*(xl-xu))/(y(xl)-y(xu)));
            xr_old=xl;
            float xr_new=xr;
            iter++;
            error=fabs((xr_new-xr_old)/xr_new)*100;

        }
        printf("iteration %d   error %f\n", iter, error);
```

```
    }
    return xr;
}
int  main()
{
    float root=falsepoint(10, 100, 0.05);
    printf("root is %f",root);
    printf("urban population is : %f\n",pu(root));
    printf("sub-urban population is : %f\n",ps(root));
}
```

2.4 Results :

We observe that the population of the sub-urbs will exceed the urban population by 20% by the end of 39.606797 years.
At that time the urban population will be 112618.726562 and the sub-urban population will be 135142.50000.

2.5 Graphs

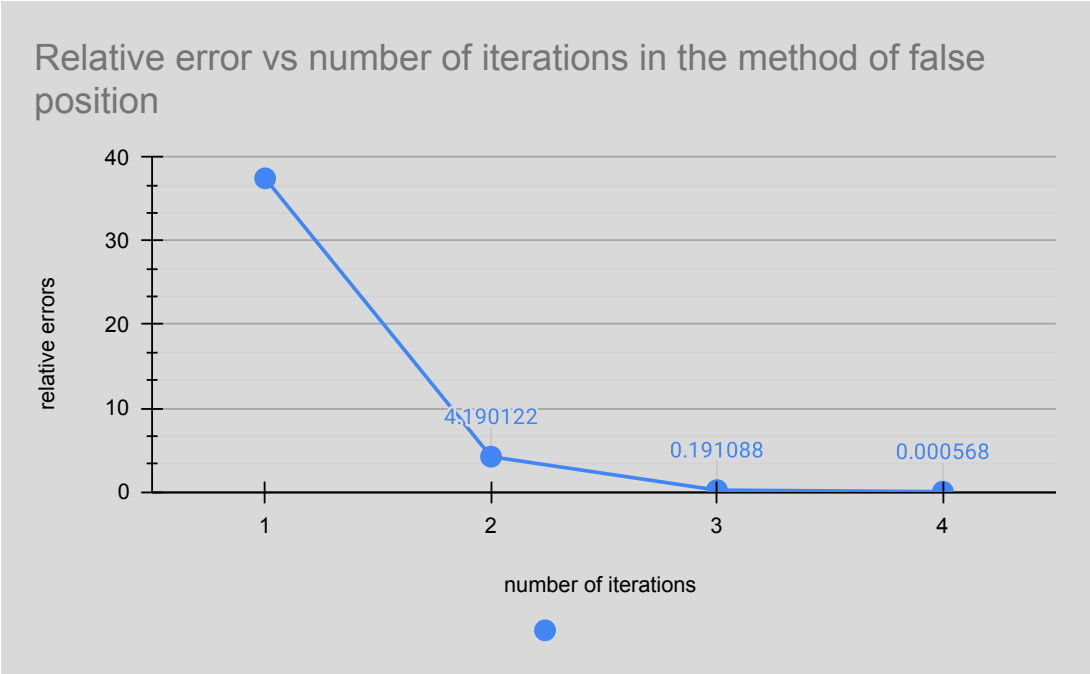


Figure 1: Percentage error Q2

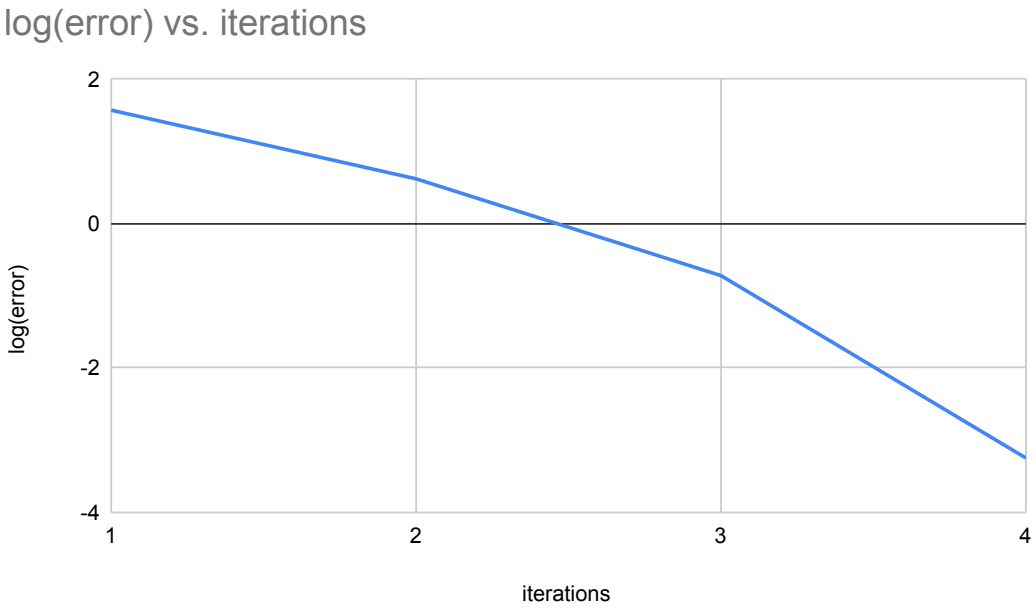


Figure 2: logarithmic error Q2

3 Problem 3: The Newton-Raphson method

Use the Newton-Raphson method to find the root of

$$f(x) = e^{-0.5x} * (4 - x) - 2 \tag{6}$$

Employ initial guesses of (a) 2, (b) 6, and (c) 8. Explain your results.

3.1 Approach

- The Newton-Raphson method falls under the category of open methods for root finding. Here we start off with one random value say x, draw a tangent to the function at the point $(x, f(x))$ and rename the the point of intersection of this tangent with the x-axis as the new x. This is repeated until the relative error is below the maximum permissible limit.
- Open methods are generally fast. This method converges quadratically, if it does converge at all. Open methods do not guarantee convergence,unlike bracketing techniques.
- To keep a check on divergence, we set a limit on the number of iterations.
- The program is so designed that it terminates on the first instance of the relative error falling below the max error percentage or if the number of iteration exceeds the mentioned value, whichever occurs first.
- One more noteworthy point is that, because this method works with the derivative of a given function, care needs to be taken to avoid division by 0.

3.2 Algorithm

```
y(x) for function
y'(x) for derivative of function
iter=0
funcNR(x,iter,tolerance)
    iter_max=10000
    if iter<iter_max
        if y(x)=0
            return x
        else
            if y'(x)=0
                print "division by 0 error"
            else
                x_new=x-(y(x)/y'(x))
                iter++
                error=|(x_new-x)/x_new|*100%
                if error<tolerance
                    return x_new
                else
                    return funcNR(x_new,iter,tolerance)
    else
        print "divergent"
```

3.3 Code

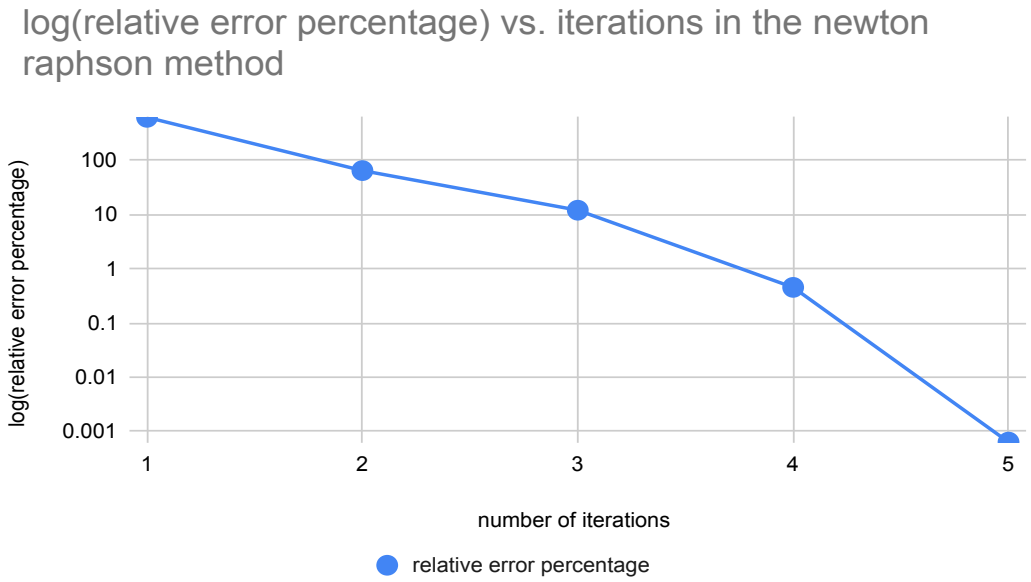
```c
#include<stdio.h>
#include<math.h>
float y(float x)//function oracle
{
    return ((4-x)*exp(-0.5*x))-2;
}
float y_(float x)//derivative oracle
{
    return 0.5*(x-6)*exp(-0.5*x);
}
float nr(float x,int iter,float tolerance)//nr stands for newton raphson
{
   int iter_max=10000;
   if(iter<iter_max)//ensuring my computer doesnt fry in case the sequence diverges!
   {
       float xr_old=x;//this notation makes it easier for understanding the error formula
       if(y(x)==0)
       {
           return xr_old;
       }
       else
       {
           //avoiding division by 0
```

```
            if(y_(xr_old)==0)
            {
                printf("division by 0 error\n");
                return -1;
            }
            else
            {
                float xr_new=xr_old-(y(xr_old)/y_(xr_old));
                iter++;
                float error=fabs((xr_new-xr_old)/xr_new)*100;
                printf("iteration: %d root: %f error: %f\n",iter,xr_new,error);
                if(error<=tolerance)
                {
                    return xr_new;
                }
                else{
                   return nr(xr_new, iter, tolerance);
                }

            }
        }
    }
    else{
        printf("the sequence is divergent");
        return -1;
    }


}
int main()
{
    printf("the most approximate root when n=2 is : %f\n", nr(2,0,0.05));
    printf("the most approximate root when n=6 is : %f\n", nr(6,0,0.05));
    printf("the most approximate root when n=8 is : %f\n", nr(8,0,0.05));
}
```

3.4 Results

We observe that when x=2, the most approximate root is 0.885709 and we get this in 5 iterations. When n=6 and 8, the derivative value becomes so low that division by 0 error is raised and hence the root cannot be computed this way- a clear case of divergence. Geometrically, this means that the tangent to the curve at some iteration became nearly flat and the point where it intersects the x-axis becomes undefined.

3.5 Graphs



4 Problem 4:Pollutant Concentration

A mass balance for a pollutant in a well-mixed lake can be written as

$$V\frac{dc}{dt} = W - Qc - kV\sqrt{c} \tag{7}$$

Given the parameter values:
V = 1×10^6 m^3,
Q = 1×10^5 m^3/yr,
W = 1×10^6 g/yr and
k = $0.25m^{0.5}/g^{0.5}/yr$
Use the modified secant method to solve for the steady-state concentration. Employ an initial guess of c = 4 g/m^3 and $\delta = 0.5$. Perform three iterations and determine the percent relative error after the third iteration.

4.1 Approach

The modified secant method is another open method for root finding.This is similar to Newton-Raphson except for one aspect-instead of providing the derivative function beforehand which might involve heavy calculations for certain functions, we include a function that provides for a close approximate of the derivative. All the other operations are identical to the Newton-Raphson method. the approximate for the derivative is:

$$f'(x) = \frac{f(x+\delta) - f(x)}{\delta} \tag{8}$$

4.2 Algorithm

```
y(x) for function
y'(x)=(y(x+d)-y(x))/d
iter=0
funcNR(x,iter,tolerance)
    iter_max=10000
    if iter<iter_max
        if y(x)=0
            return x
        else
            if y'(x)=0
                print "division by 0 error"
            else
                x_new=x-(y(x)/y'(x))
                iter++
                error=|(x_new-x)/x_new|*100%
                if error<tolerance
                    return x_new
                else
                    return funcNR(x_new,iter,tolerance)
    else
        print "divergent"
```

4.3 Code :

```c
//A mass balance for a pollutant in a well-mixed lake can be
//written as
//Vdc =WQckVc dt
//Given the parameter values V = 1 × 106m3, Q = 1 × 105 m3/yr, W = 1 × 106
↪ g/yr, and k = 0.25 m0.5/g0.5/yr, use the modified secant method to solve
↪ for the steady-state concentration. Employ an initial guess of c = 4 g/m3
↪ and  = 0.5. Perform three iterations and determine the percent relative
↪ error after the third iteration.
#include<stdio.h>
#include<math.h>
float y(float c)//function oracle
{
    float v=pow(10,6),q=v/10,w=v,k=0.25;
    return w-q*c-k*v*(sqrt(c));
}
float ms(float c,int iter,float tolerance)//ms stands for modified secant
{
    float del=0.5;
    float iter_max=1000;//again, precaution for the case of divergence.
    float xr_old=c;
    if(y(xr_old)==0)
    {
        return xr_old;
    }
    else
```

```
    {
        if(iter<iter_max)
        {
            float temp=y(xr_old+del)-y(xr_old);//introducing a temporary
↪   variable after realising that this quantity is required quite a few times.
            if(temp!=0)
            {
                float xr_new=xr_old-((y(xr_old))/((temp)/del));
                iter++;
                float error=fabs((xr_new-xr_old)/xr_new)*100;
                printf("iteration:  %d root : %f error:
↪   %f\n",iter,xr_new,error);
                if(error<tolerance)
                {
                    return xr_new;
                }
                else{
                    return ms(xr_new, iter, tolerance);
                }
            }
            else{
                printf("divergent");
                return -1;
            }
        }
        else{
            printf("divergent\n");
            return -1;
        }
    }
}
int main()
{
    printf("the most approximate root is : %f", ms(4,0,0.005));
}
```

4.4 Results

The output is :
iteration: 1 root : 4.622432 error: 13.465461
iteration: 2 root : 4.624096 error: 0.035999
iteration: 3 root : 4.624081 error: 0.000340
the most approximate root is : 4.624081
We observe that this method converges pretty quickly and we got a root within an error percentage of 0.05% in just 3 steps.

4.5 Graphs

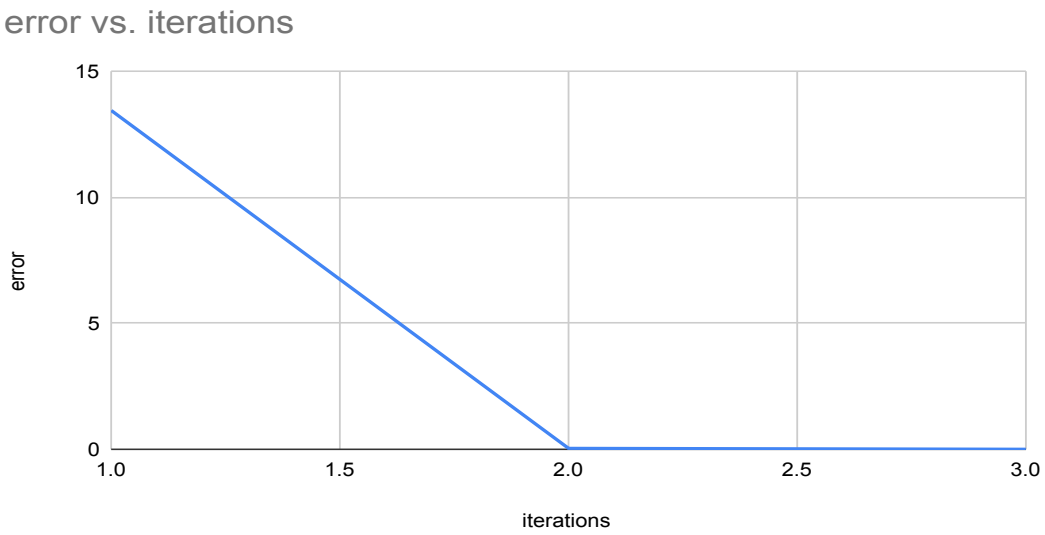


Figure 3: Percentage error after every iteration