Analysis of Numerical Methods - Project 1

Tej Sai Kakumanu

September 2024

1 Bisection Method

The Bisection method is typically used for finding the Zeroes of a function. It is used on continuous functions that have a zero in the given domain [a,b].

Firstly, we make sure that a given function $f \in C[a, b]$ and that $f(a) * f(b) < 0$. This is to ensure that, before we begin, there exists a solution (x^*) for which $f(x^*) = 0$ by the Intermediate value theorem(IVT).

Now, we define two types of tolerances to arrive at a particular solution that is very close to x^*. They are Error tolerance Tol_e and Residual tolerance Tol_r. These are the values that the user estimates as close enough for the algorithm to stop. Error Tolerance expresses how close the x_n should be to the x^* and the residual tolerance shows how close $f(x_n)$ needs to be to $f(x^*)$

Furthermore, we can use these values to make the following conditions for our algorithm to stop when it finds an appropriate result:

$$Tol_e > |x_n - x^*|$$
$$Tol_r > |f(x_n) - f(x^*)|$$

Though we don't know x^* and we don't use error tolerance directly embedded in the algorithm. It really helps in knowing the maximum amount of iterations the algorithm will need to get closest to the position. we know the maximum level of error in the domain [a,b] is b-a when we are in the initial iteration and the domain cuts by half each time the iteration goes through. we could guarantee that $\frac{b-a}{(2^n)} \leqslant Tol_e$ where n = the number of iterations possible. Now, we manipulate this expression to get a formula for the maximum number of steps needed as $n >= log_2(\frac{b-a}{Tol_e})$. This is just to make sure the program doesn't get to an infinite loop when we choose a wrong domain where there exists no $x^* s.t. f(x^*) = 0$.

However, it is not always very efficient and reliable to depend on estimation of error tolerance as it can be a bad estimation where the algorithm is using exponentially more time than it actually needs. Therefore, in cases when the nature of function is known better and the estimation of how close $f(x_n)$ needs to be from $f(x^*)$ is simpler, we use the condition $Tol_r > |f(x_n) - f(x^*)|$.

With all these conditions in place, we try to loop inside the domain of the functions until one of these conditions are met by cutting the domain by half in each iteration. We start with $x_n = \frac{b+a}{2}$ in the domain [a,b] and keep changing the domain of function in each iteration by assigning the current x_n value to a or b. This is determined based on the condition $f(x_n) * f(a) < 0$ If the condition is true, it means x^* is in upper half of the domain so we set a $a = x_n$ and otherwise, it means it is in lower half of the domain so we set $b = x_n$

For Example, The number of iterations to guarantee the answer within 10^{-6} and 10^{-10} can be derived from the formula $n = log_2(\frac{b-a}{Tol_e})$ and round up the result to get the maximum number of steps needed. Therefore by using calculator and substituting the respective ErrTol, we get Nsteps = 24 for ErrTol = 10^{-6} and Nsteps = 38 for ErrTol = 10^{-10}

In summary, we put all these concepts together to make the following algorithm and find the $x^* \in [a, b]$

Step 1: Define Function, Domain, ResTol, ErrTol, and Nsteps using respective formulae.
Step 2: Check if(f(a)*f(b) ¡ 0), Proceed to next step if true or end with exit code 1.
Step 3: Start loop for Nsteps iterations.
Step 4: Assign x with the midpoint of the domain, $x = \frac{b+a}{2}$
Step 5: Check if(abs(f(x)) $\leqslant$ ResTol), Go to Step 10 if true or continue otherwise.
Step 6: Check if(f(x) * f(a) < 0), Go to Step 7 if true, or Step 8 if false.
Step 7: Assign b = x.
Step 8: Assign a = x.
Step 9: Go to step 3 if the current iteration is not equal to Nsteps, otherwise continue to next step.
Step 10: Assign xStar = x.
Step 11: End.

Keeping aside how efficient this algorithm is, the answer to whether or not this algorithm will work is just yes while the pre-conditions are met i.e. $f \in C[a, b]$ and $f(a) * f(b) < 0$ are both true by Intermediate Value Theorem. These conditions assure that the function that we are working with is defined at every spot in the domain [a,b] and that there exists $x^* \in C[a, b] s.t. f(x^*) = 0$. The algorithm won't work if these conditions are not met as they open up the algorithm for more chances of crashing and makes it unreliable.

Here is the snippet of the code that performs 5 iterations of the bisection method to find the zero so the function $f(x) = x^2 - 3$ $x \in [a, b]$

```
% MTH361 Fall 2024
% Bisection method for Project #1
% Tej Sai Kakumanu

clear all
clc

f = @(x) x^2 - 3;           % Define the Function

a = 1;                      % Left endpoint
b = 2;                      % Right endpoint

% Check that function has an opposite points at
   the left and right endpoints.
if (f(a)*f(b) < 0)
        for i=1:5           % Take up to Nsteps

            x = (b+a)/2;    % Find the midpoint
% Check which half has the value that we are
   looking for
            if(f(x) * f(a) < 0)

            b = x;
            else
            a = x;
            end
        end
end
xstar = x;
f(xstar)
```

This code results in $x^* = 1.7188$ where $f(x^*) = -0.0459$ and close to 0 and the $|sqrt(3) - x^*|$ gives how close to the result is to the actual values, rounded to three decimal places. So we get 0.0133 as the difference.

2 Fixed Point Iteration Method

The Fixed point iteration method is known for its efficiency when the nature of a function in a particular domain is well known. To prove that function g has a fixed point $p \in [0,1]$ when $g(x) \in [0,1] and x \in [0,1]$, we can first check if the function has a fixed point at the endpoints by checking if $g(p) = p$ when $p = 0, 1$ if not, then we know $g(0) > 0$ and $g(1) < 1$ so we can use the following explanation for proving that function g has a fixed point.

We can assume the function g has a fixed point at $g(p) = p$ where $p \in [0,1]$ and consider a function $h(x) = g(x) - x \in C[0,1]$ since we already know $g(x) \in C[0,1]$. Now, at p, we have $h(p) = g(p) - p = 0 \therefore g(p) - p = 0 \implies g(p) = p$ This means our assumption was correct and the function g has at least one fixed point in the domain [0,1].
Furthermore, we can prove that this point is unique by using the Mean Value Theorem(MVT) and proof by contradiction. Assuming p and q as two different fixed points in [0,1] for the function $g \in [0,1]$ we know that by MVT there exists a point x between p and q where $g'(x) = \frac{g(p)-g(q)}{p-q}$ This gives us the following expression:

$$|p - q| = |g(p) - g(q)| = |g'(x)||(p - q)|$$

Since we have $|g'(x)| \leqslant \frac{1}{3}$ and the equation $|p - q| = |g'(x)||p - q|$ can only be true when $p = q$ this assumption is incorrect and by proof of contradiction we can conclude that we have a unique fixed point for $g \in [0,1]$
Now, let's assume $x_{k+1} = g(x_k)$ is true and by MVT we have the following:

$$|x_{k+1} - p| = |g(x_k) - g(q)| = |g'(\eta)||(x_{k+1} - p)|$$

We already know $|g'(\eta)| < 1$ and by taking $|g'(\eta)| \leqslant n = \frac{1}{3}$ we get the inequality

$$n|x_k - p| \leqslant n^2|x_{k-1} - p|$$

And by applying this inductively we get(1):

$$|x_{k+1} - p| \leqslant n|x_k - p| \leqslant n^2|x_{k-1} - p| \leqslant \ldots \leqslant n^k|x_0 - p|$$

Now we check what happens when k $\rightarrow \infty$,

$$\lim_{k\to\infty} |x_{k+1} - p| \leqslant \lim_{k\to\infty} n^k|x_0 - p| \leqslant 0$$

$$\lim_{k\to\infty} |x_{k+1} - p| \leqslant 0$$

$$\lim_{k\to\infty} |x_{k+1}| \leqslant \lim_{k\to\infty} |p|$$

$$\lim_{k\to\infty} |x_{k+1}| \leqslant p$$

$$\therefore \{x_{k+1}\}_{n=0}^{\infty} \rightarrow \infty$$

From (1) we know that $|x_{k+1} - p| \leqslant n^k |x_0 - p|$ and because the function is in the domain [0,1] we get $|x_0 - p| \leqslant 1$ and therefore we can write the equation as $|x_{k+1} - p| \leqslant (\frac{1}{3})^k$ when we substitute $n = \frac{1}{3}$.

To find the fixed points of the function $f(x) = x - \frac{x^2-3}{6}$ in [1,2] we can construct and use the algorithm in Matlab using the theory above. The algorithm ran successfully and gave the anticipated results for the same function and domain. The anticipated maximum number of iterations for this function for tolerance 10^{-6} and 10^{-10} should be 35 and 57, respectively. This hypothesis can be derived by using the following formula that uses the maximum value of $f'(x) \in [1, 2]$ which is $\frac{2}{3}$ in this case and the appropriate tolerance:

$$Nsteps = \log_{\frac{2}{3}} Tol$$

However, running the algorithm with tolerance 10^{-10} took 25 iterations whereas tolerance 10^{-6} took 14 iterations. Using higher tolerance did not give off any drastic differences in the results as both the answers were the same when rounded to three decimal places.

Furthermore, considering the equation $f(x) = x - \frac{x^2-3}{100}$ in the domain [1,2], the algorithm ran successfully without errors and resulted with the exact same answers as last time. We can use the same formula as last time but with different base for the log, which is $\frac{49}{50}$, since the function and the maximum value of derivative changed in the domain. By doing so we get the maximum anticipated iterations as 684 and 1140 for tolerance 10^{-6} and 10^{-10} respectively. However, it did not actually take those many iterations since the algorithm stopped for 519 and 258 iterations for tolerance 10^{-10} and 10^{-6}, respectively.

Lastly, To find the most appropriate value of ϵ for the function $f(x) = x - \epsilon * (x^2 - 3)$ we need to evaluate the value of ϵ when the derivative of the function is equated to zero at a fixed point. This is because this derivative gives the rate of change of function at the fixed point and the closer it is to 0, the faster it tends to converge.

$$f'(x) = 1 - \epsilon * (2x)$$
$$f'(\sqrt{3}) = 1 - \epsilon * (2\sqrt{3})$$
$$\text{if} f'(\sqrt{3}) = 0 \rightarrow 1 - \epsilon * (2\sqrt{3}) = 0$$

Therefore, the fastest convergence will occur at $\epsilon = \frac{1}{2\sqrt{3}}$. With this value the algorithm meets the tolerance in just 3 and 4 steps for 10^{-6} and 10^{-10} respectively.

This way could be used to optimise the ϵ value when we know the root of the equation. However, in typical scenario when we use the fixed point iteration, we don't really know the root of the function. Therefore, we should perform a trail and error method in which we test out a few values and choose the best guess from those for getting the fastest convergence. For example, in this case, we can take 10 guesses between 0 and 1 since we know $f'(x)$ won't map into the

same domain(Pre-condition for fixed point iteration) if it is less than or equal to 0 or greater than or equal to 1. //
When compared to the bisection method, finding the value of $\sqrt{3}$ is less reliable in the fixed point iteration method as we have to make an appropriate combination of functions to make it converge properly but the bisection method is more reliable and easier to deal with in this case. However, the fixed point iteration method would be way more efficient and cheap than the bisection method if you choose the correct domain and combination for the function.

The following is the Matlab code that I used to implement the algorithm, all the results presented in this section are based on how this code executed (I did change the function and tolerance accordingly when required) and the graphs of error convergence for functions $f(x) = x - \frac{x^2-3}{6}$ and $f(x) = x - \frac{x^2-3}{100}$:

```
% MTH361 Fall 2024
% Project 1 Fixed Point Iteration Method
% Tej Sai Kakumanu

clear all
clc

eps = 1/(2*sqrt(3));
g = @(x) x - eps*(x^2 - 3); % Define the Function

a = 1;                      % Left endpoint
b = 2;                      % Right endpoint


ResTol = 1.e-6;             % Residual Tolerance
Nsteps = 1000;
x = (b+a) / 2;
for i=1:Nsteps
    x = g(x);               % Fixed point operation
    xguess(i) = x;
    if(abs(g(x) - x) < ResTol)
        break
    end
end

xstar = x
```

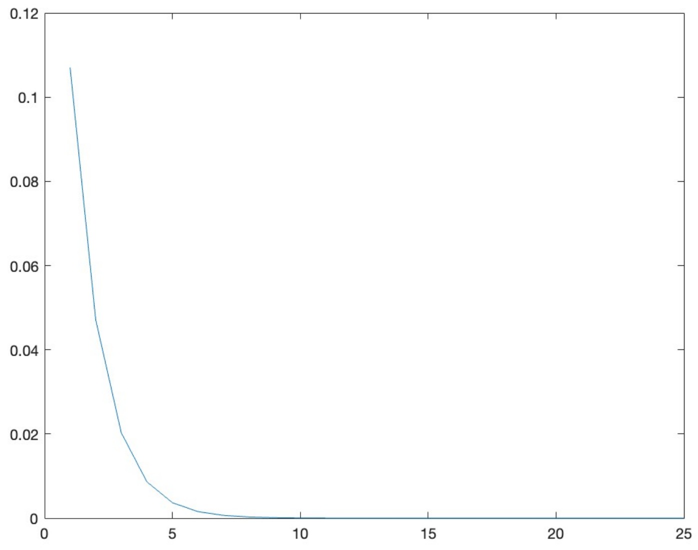

Figure 1: $f(x) = x - \frac{x^2-3}{6}$

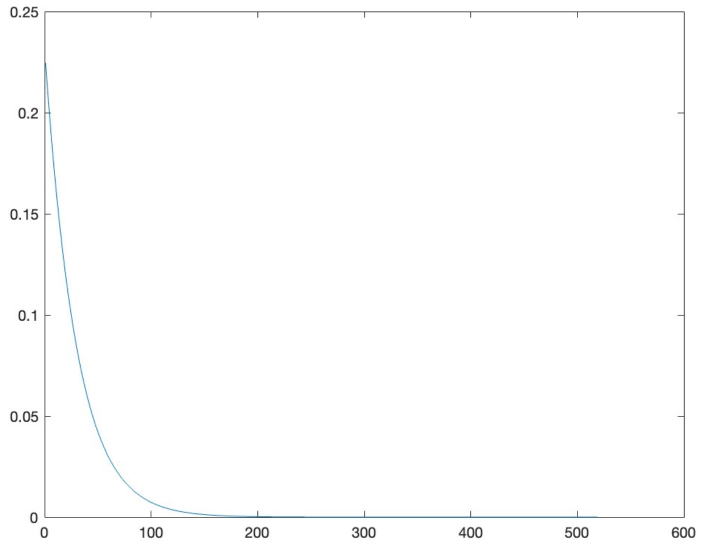

Figure 2: $f(x) = x - \frac{x^2-3}{100}$

3 Newton Iteration

The Newton's Iteration work based on the same principle as the Fixed point iteration but it uses the following newton's formula to get a quadratic convergence.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

To prove the convergence of this equation, we use the tailor series expansion for $f(x^*)$ at x_k.

$$f(x^*) = f(x_k) + (x^* - x_k)f'(x_k) + \frac{1}{2}(x^* - x_k)^2 f"(x_k)...$$

By using the remainder theorm, we get:

$$f(x^*) = f(x_k) + (x^* - x_k)f'(x_k) + \frac{1}{2}(x^* - x_k)^2 f"(\eta)$$

By dividing this equation by $f'(x_k)$

$$f(x^*) = \frac{f(x_k)}{f'(x_k)} + (x^* - x_k) + \frac{1}{2}(x^* - x_k)^2 \frac{f"(\eta)}{f'(x_k)}$$

Since we know $f(x^*) = 0$ and $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ we can rearrange and simplify our equation:

$$0 = x^* - (x_k - \frac{f(x_k)}{f'(x_k)}) + \frac{1}{2}(x^* - x_k)^2 \frac{f"(\eta)}{f'(x_k)}$$

$$0 = x^* - x_{k+1} + \frac{1}{2}(x^* - x_k)^2 \frac{f"(\eta)}{f'(x_k)}$$

$$|x^* - x_{k+1}| = \frac{1}{2}(x^* - x_k)^2 \frac{f"(\eta)}{f'(x_k)}$$

Substituting error $e_k = x^* - x_k$ and $C = \frac{f"(\eta)}{2f'(x_k)}$ gives us

$$e_{k+1} = C(e_k)^2$$

Therefore, we can see how our error has a quadratic convergence when picked correct value of C i.e appropriate function and domain that doesn't have a minimum or maximum at the root that we're trying to find. This is because our function will fail if we $f'(x_k) = 0$. This could also be identified as a disadvantage of Newton's method and the deciding factor that we should consider when thinking of using newton's method for a particular function in a particular domain.

Furthermore, using this method for the function $f(x) = x^2 - 3 \in [1,2]$ would definitely work since $f'(x) \neq 0 \in [1,2]$. It will take 3 and 4 iterations for tolerances 10^{-6} and 10^{-10}, respectively and result in exact same result correct to 4 decimal places.

When compared to bisection and fixed point iterations, this method is exponentially faster since it works by an algorithm with quadratic convergence. However, it is less reliable due to its condition of $f'(x) \neq 0$ in the domain and is also more expensive than both of them in terms of the required computational power since it is evaluating two functions $f(x)$ and $f'(x)$ compared to just one in the other two algorithms.

The following is the Matlab code that I used to implement the algorithm, all the results presented in this section are based on how this code executed (I did change the tolerance accordingly when required):

```
% MTH361 Fall 2024
% Project 1 Newton's Method
% Tej Sai Kakumanu

clear all
clc


a = 1;                          % Left endpoint
b = 2;                          % Right endpoint
f = @(x) x^2 - 3;
fp = @(x) 2*x;


ResTol = 1.e-6;                 % Residual Tolerance
Nsteps = 100;
x = (b+a) / 2;
for i=1:Nsteps
    x = x - f(x)/fp(x);         % Newton's Method
    xguess(i) = x;
    if(abs(f(x)) < ResTol)
        break
    end
end

xstar = x
```

4 Secant Method

The secant method is a better version of Newton's method in terms of computational power. This method introduces a new iterative formula that uses two previous points to evaluate the zero of a function. That said, it is clear that this method uses more memory than Newton's iteration.

Since Newton's Method uses $f'(p_{n-1})$ to evaluate the value of p_n we use the definition of the secant slope and equate

$$f'(p_{n-1}) = \frac{f(p_{n-1}) - f(p_{n-2})}{p_{n-1} - p_{n-2}}$$

Substituting this into Newton's Method gives us the following:

$$p_n = p_{n-1} - \frac{f(p_{n-1})(p_{n-1} - p_{n-2})}{f(p_{n-1}) - f(p_{n-2})}$$

Having the algorithm iterate through this formula until $f(p_n)$ meets the tolerance level. you can get a good approximate of where the zero of the function is. Now, using this in the algorithm with the function $f(x) = x^2 - 3 \in [1, 2]$ gives the required result and takes 4 and 5 iterations for tolerance 10^{-6} and 10^{-10} respectively.

These results are very similar to Newton's method but highly cost-effective due to the less computational power required. When compared to other methods, in terms of iterations it took, again, it is close to Newton's Method with ± 1 difference. However, like newton's method, it showed a drastic difference when compared to the number of iterations in bisection and fixed point iteration method.

The following is the Matlab code that I used to implement the algorithm, all the results presented in this section are based on how this code executed (I did change the tolerance accordingly when required):

```
% MTH361 Fall 2024
% Project 1 Secant Method
% Tej Sai Kakumanu

clear all
clc


a = 1;                                          % Left
    endpoint
b = 2;                                          % Right
    endpoint
f = @(x) x^2 - 3;


ResTol = 1.e-10;                                % Residual
     Tolerance
Nsteps = 100;
x_old = a;
x = (b+a) / 2;
for i=1:Nsteps
     xtemp = x;
     x = x - f(x)*(x-x_old)/(f(x)-f(x_old));
                             % Secant Iteration
     x_old = xtemp;
     xguess(i) = x;
     if(abs(f(x)) < ResTol)
          break
     end
end

xstar = x;
f(xstar);
```

5 Method of False Position

The Method of false iteration uses a similar technique as bisection method as it works by bracketing the domain and makes sure the solution lies somewhere inside the domain. However, it uses a very different logic than bisection method as it works based on the secant line formula similar to the one in secant method. It works by making the domain shorter in each iteration based on the point where secant line is meeting the x-axis or its function is 0 until the error meets the tolerance.

We can use the following iterative formula for doing the same:

$$x_{new} = \frac{(x_{old} * f(x)) - (x * f(x_{old}))}{(f(x) - f(x_{old}))}$$

Here, x and x_{old} just represent the co-ordinates of the current domain and x_{new} is the new guess that we get from the formula which will be replaced with x in every iteration. x_{old} gets replaced with x if the condition $f(x_{new}) * f(x) < 0$ is true.

Executing this algorithm with the function $f(x) = x^2 - 3 \in [1, 2]$ gives the exact same result correct to 4 decimal places for both 10^{-6} and 10^{-10} tolerances but they take 6 and 9 iterations each, respectively.

When compared to the other methods, this method took fewer iterations than Fixed Point Method and Bisection Method but a little more than Newton's and Secant Method. It, again has the same advantages as secant method over Newton's method of not needing the extra computational power for evaluating a second function. Lastly, This is also a good alternative for the case where secant method is undefined.

The following is the Matlab code that I used to implement the algorithm, all the results presented in this section are based on how this code executed (I did change the tolerance accordingly when required):

```
% MTH361 Fall 2024
% Project 1 Method of False position
% Tej Sai Kakumanu

clear all
clc

f = @(x) x^2 - 3;                    % Define the
    Function
```

```
a = 1;                                    % Left
    endpoint
b = 2;                                    % Right
    endpoint


ResTol = 1.e-10;                          % Residual
    Tolerance
Nsteps = 100;

% Check that function has an opposite points at
    the left and right endpoints.
if (f(a)*f(b) < 0)
          tic
          x = (b+a)/2;                    % Find the
              midpoint
          xguess(1) = x
          % Check which half has the value that we
              are looking for
          if(f(x) * f(a) < 0)
               xold = a;
          else
               xold = b;
          end
          for i=1:Nsteps
               xnew = (xold * f(x) - x*f(xold))/(f(x)
                   -f(xold));
               xguess(i) = x
               % Check which side has the value that
                   we are looking for
               if(f(xnew) * f(x) < 0)
                    xold = x;
               end
               x = xnew;

               if (abs(f(x)) < ResTol)  % Check that
                   my residual is not too small
                    break;               % Break if it
                         is small enough
               end

     end
  end
     toc
xstar = x;
f(xstar)
```

The following is the summary of how many iterations each method took with respective functions and tolerances.

Method	Function	Iterations (10^{-6})	Iterations (10^{-10})
Bisection	N/A	Max: 24	Max: 38
Fixed Point Iteration	$f(x) = x - \frac{1}{6}(x^2 - 3)$	14	25
Fixed Point Iteration	$f(x) = x - \frac{1}{100}(x^2 - 3)$	258	519
Newton's Method	$f(x) = x^2 - 3$	3	4
Secant Method	$f(x) = x^2 - 3$	4	5
Method of False Position	$f(x) = x^2 - 3$	6	9

6 Resources used

• Burden, Richard L, J Douglas Faires. Numerical Analysis. 1 Jan. 2015.
• Matlab