Problem 1.

(a) Show that $x - (\ln x)^x = 0$ has at least one solution on $[4, 5]$.

(b) Use the Intermediate Value Theorem and Rolle's Theorem to show that the graph of $f(x) = x^3 + 2x + k$ crosses the x-axis exactly once, regardless of the value of the constant k.

Solution

Part (a)

Proof. Note that the function is continuous, in particular, it is continuous over $[4, 5]$. Further as we have that $f(4) = 4 - ln(4)^4 = 0.306638422$ and
$f(5) = 5 - ln(5)^5 = -5.798691578$, and $-5.79869 < 0 < 0.306638422$, then by the Intermediate Value Theorem, $\exists x \in [4, 5]$ such that $f(x) = 0$. □

Part (b)

Proof. Note that as f is a polynomial, thus it is continuous everywhere. In particular, it is continuous on any interval (i, j) where $i, j \in \mathbb{R}$.

If $x < 0$ then $x^3 + 2x + k < 2x + k$ further, if $x < -\frac{k}{2}$ then $x^3 + 2x + k < 2x + k < 0$.
Choose $a \in \mathbb{R}$ such that $a < -\frac{k}{2}$.

If $x > 0$ then $x^3 + 2x + k > 2x + k$ further, if $x > -\frac{k}{2}$ then $x^3 + 2x + k > 2x + k > 0$.
Choose $b \in \mathbb{R}$ such that $b > -\frac{k}{2}$.

We know that f is continuous on (a, b), we also know that $f(a) < 0 < f(b)$.
Thus, by the Intermediate Value Theorem, $\exists c \in (a, b)$ such that $f(c) = 0$. □

Problem 2.

Let $f(x) = 2x\cos(2x) - (x-2)^2$ and $x_0 = 0$.

(a) Find the third Taylor polynomial $P_3(x)$, and use it to approximate $f(0.4)$

(b) Use the error formula in Taylor's Theorem to find an upper bound for the error $|f(0.4) - P_3(0.4)|$. Compute the actual error.

Solution

Part (a)

$P_3(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2}(x-x_0)^2 + \frac{f^{(3)}(x_0)}{6}(x-x_0)^3$

$$\begin{aligned} f(0) &= 2 \times 0\cos(0) - (-2)^2 = -4 \\ f'(0) &= -2x - 4x\sin(2x) + 2\cos(2x) + 4 = 2\cos 0 + 4 = 2 + 4 = 6 \\ f''(0) &= -2(4\sin(2x) + 4x\cos(2x) + 1) = -2(0+0+1) = -2 \\ f^{(3)}(0) &= 8(2x\sin(2x) - 3\cos(2x)) = 8(0-3) = -24 \end{aligned}$$

Thus, the third Taylor polynomial about $x_0 = 0$ is given by

$$P_3(x) = -4 + 6x + -x^2 + -4x^3$$

Then

$$f(0.4) \approx P_3(0.4) = -2.016$$

Part (b)

We have that $f^{(4)}(x) = 32(2\sin(2x) + x\cos(2x))$.
The error is then given by the following (for some $c_x \in (0, 0.4)$):

$$\mathcal{E}(x) = \frac{f^{(4)}(c_x)}{24}x^4$$

Thus, to find an upper bound of the error, we must find an upper bound for $f^{(4)}(c_x)$. This is found at $f^{(4)}(0.4) = 54.8286$. Giving us an error of $\frac{54.8286}{24}(0.4)^4 = 0.05848384$.
Actual error is given by $|f(0.4) - P_3(0.4)| = |-2.00263 - -2.016| = 0.0133654$.

Problem 3.

Let $f \in C[a,b]$ be a function whose derivative exists on (a,b). Suppose f is to be evaluated at x_0 in (a,b), but instead of computing the actual value $f(x_0)$, the approximation value, $\widetilde{f}(x_0)$, is the actual value of f at $x_0+\epsilon$, that is $\widetilde{f}(x_0) = f(x_0+\epsilon)$.

(a) Use the Mean Value Theorem to estimate the absolute error $|f(x_0) - \widetilde{f}(x_0)|$ and the relative error $|f(x_0) - \widetilde{f}(x_0)|/|f(x_0)|$, assuming $f(x_0) \neq 0$.

(b) If $\epsilon = 5 \times 10^{-6}$ and $x_0 = 1$, find bounds for the absolute and relative errors for

(i) $f(x) = e^x$

(ii) $f(x) = \sin x$

Solution

Part (a)

For absolute error, we have $|f(x_0) - \widetilde{f}(x_0)| = |f(x_0) - f(x_0+\epsilon)|$ by the mean value theorem it follows that $|f(x_0) - f(x_0+\epsilon)| = |f'(c)\epsilon|, c \in (x_0, x_0+\epsilon)$.
Therefore, our bound for absolute error is given by $\epsilon \times \sup|\{f'(c) : c \in (x_0, x_0+\epsilon)\}|$.
Similarly, the bound for relative error is thus given by
$\epsilon \times \sup|\{f'(c) : c \in (x_0, x_0+\epsilon)\}|/|f(x_0)|$.

Part (b)

(i) $f(x) = e^x$
To find absolute error by the formula given in part (a), we simply need to find the maximum derivative of $f(x)$ over $(1, 1+5\times 10^{-6})$. As e^x is non-decreasing, and $e^{x\prime} = e^x$ the maximum is given by $e^{(1+5\times 10^{-6})} \approx 2.7182954199$ we then multiply this by ϵ to get the absolute error $= e^{1+5\times 10^{-6}} 5 \times 10^{-6} \approx 0.000013591477$.

Relative error is then $(e^{1+5\times 10^{-6}} 5 \times 10^{-6})/(e^1) \approx 5.000025 \times 10^{-6}$.

(ii) $f(x) = \sin x$
$f'(x) = \cos(x)$, thus to determine bound for absolute error, I need the maximum value of $\cos(x)$ over $(1, 1+5\times 10^{-6})$. Between 0 and $\pi/2$ $\cos(x)$ is positive and decreasing. As we have $0 < 1 < 1 + 5 \times 10^{-6} < \pi/2$ we will find our maximum value at $cos(1) \approx 0.5403023$. We then multiply this by ϵ to get the absolute error $= \cos(1) \times \epsilon \approx 2.701511529 \times 10^{-6}$.

Relative error is then $\cos(1) \times \epsilon / \sin(1) \approx 3.210463 \times 10^{-6}$.

Problem 4.

The equation $4x^2 - e^x - e^{-x} = 0$ has four solutions $\pm x_1$ and $\pm x_2$. Use Newton's method to approximate the solution to within 10^-5 with the following values of y_0 : (a)$p_0 = -10$, (b) $p_0 = -5$, (c) $p_0 = -3$, (d) $p_0 = -1$, (c) $p_0 = 1$, (g) $p_0 = 3$, (h) $p_0 = 5$, (i) $p_0 = 10$. Comment on the results.

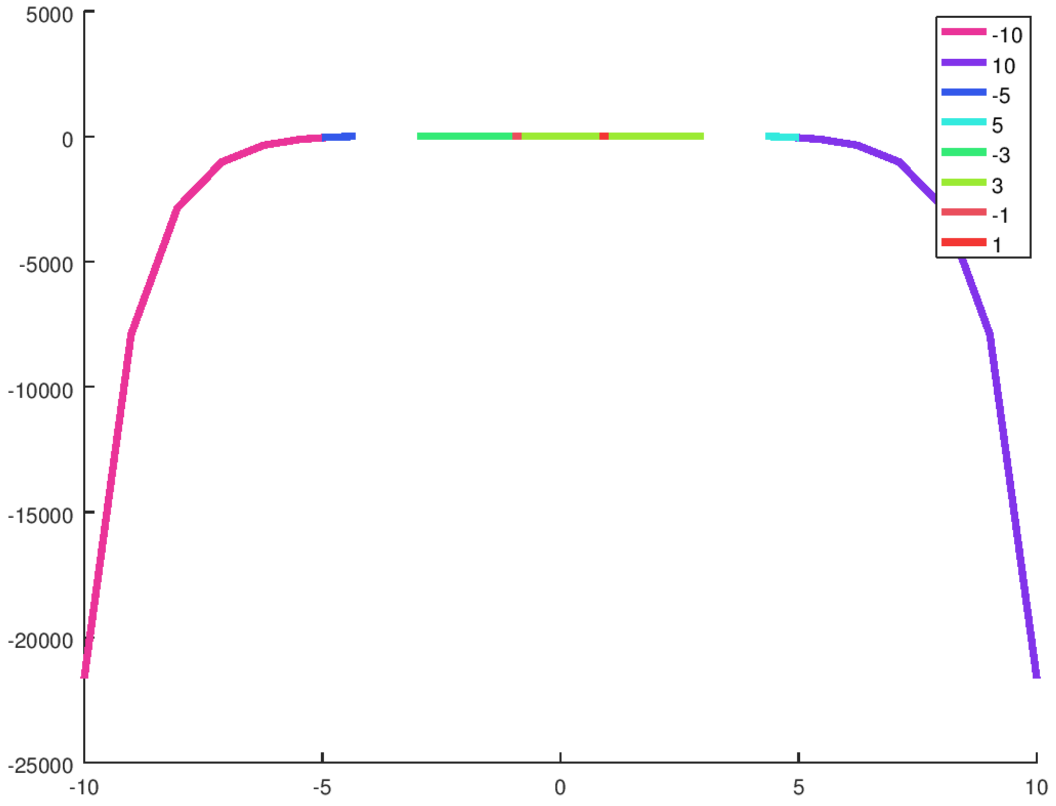
Solution

```
function p4
clear all;
close all;
hold on;

% The function to find 0's for...
f = @(t) (4 * t.^2) - e.^t - e.^(-t);
df = @(t) (8 * t) - e.^t + e.^(-t);
% Iterate via Newton's method until we are within 'err' of 0.
ERR = 10^-(5);
% List of starting points to use with Newton's method.
% Change the order so they show up better when I plot the results.
_y0 = [-10, 10, -5, 5, -3, 3, -1, 1];
% Use these colors for plotting Newton's method for each _y0.
COL = [
        [243 54 51] ./ 255,
        [234 78 92] ./ 255,
        [155 234 51] ./ 255,
        [51 234 118] ./ 255,
        [51 234 222] ./ 255,
        [51 88 234] ./ 255,
        [130 51 234] ./ 255,
        [234 51 152] ./ 255
]

% Loop through each starting point to run Newton's method.
for y0 = _y0

        % Steps will contain a list of each point in the iteration.
        % We can plot this, as well as examine it's size to see
        % how well each initial point is performing.
        steps = [y0];

        % loop while we still have too much error
        % infinite loop if we don't converge... but we do
        % for these y, so it's cool
        while abs(f(steps(end))) > ERR
                tmp = steps(end);
                nxt = tmp - f(tmp)/df(tmp);
                steps = [steps nxt];
        end

        y0
        steps
        plot(steps, f(steps), 'LineWidth', 3, 'Color', COL(end, :),
                'DisplayName', int2str(y0));
        COL = COL(1:end-1, :);

end

legend('show');
end
```

Below I have formatted the text output from that code, we can immediately see some symmetry about the axis $x = 0$. Using the initial y_0 values, I found a total of 4 zeros at $x = \{\pm 4.3062, \pm 0.82450\}$. The closer I start to the guess, the faster (less steps) Newton's method will find a result. Interestingly, $x = \pm 3$ overshoot the zero on their side of the axis $x = 0$, that is $+3$ finds -0.82450 and -3 finds $+0.82450$. All others initial values find the zero nearest to them.

```
-10 => [-10.0000, -9.0146, -8.0456, -7.1093, -6.2339, -5.4634,
        -4.8572, -4.4752, -4.3269, -4.3066,  -4.3062]

-5 => [-5.0000  -4.5533  -4.3478  -4.3076  -4.3062  -4.3062]

-3 => [-3.00000   1.00194   0.83852   0.82461   0.82450]

-1 => [-1.00000  -0.83825  -0.82460  -0.82450]

1 => [1.00000   0.83825   0.82460   0.82450]

3 => [3.00000  -1.00194  -0.83852  -0.82461  -0.82450]

5 => [5.0000   4.5533   4.3478   4.3076   4.3062   4.3062]

10 => [10.0000, 9.0146, 8.0456, 7.1093, 6.2339, 5.4634,
        4.8572, 4.4752, 4.3269, 4.3066, 4.3062]
```

And here is a graph to visualize what Newton's method is doing for each starting point. Note there is some overlap for initial values of ± 3 as they approach opposite zeros.



Problem 5.

Use Newton's method and Modified Newton's method to find the solution accurate to within 10^{-5} for

$$1 - 4x\cos x + 2x^2 + \cos 2x = 0, \text{ for } 0 \leq x \leq 1.$$

Comment on the performance of the methods.

Solution

```
function p5
clear all;
close all;
hold on;

% The function to find 0's for...
f = @(t) 1 - 4.*t .* cos(t) + 2 .* (t.^2) + cos(2.*t);
df = @(t) 4.*t + 4.*t.*sin(t) - 2.*sin(2.*t) - 4.*cos(t);
% We'll need this for Modified Newton's Method.
dff = @(t) 8.*sin(t) + 4.*t.*sin(t) - 4.*cos(2.*t) + 4
% Use the midpoint as our initial guess.
MID = (0.0 + 1.0) / 2.0
% Want a zero within this error.
ERR = 10^-(5);

%% Newton's Method

steps = [MID];
while abs(f(steps(end))) > ERR
        tmp = steps(end);
        nxt = tmp - f(tmp)/df(tmp);
        steps = [steps nxt];
end

% Output the results for Newton's Method
steps
f(steps(end))
plot(steps, f(steps), 'LineWidth', 3, 'DisplayName', 'Newtons Method');

%% Modified Newton's Method

% Use the midpoint as the initial step.
steps = [0.5];

mu = @(t) f(t) ./ df(t);
% Use the quotient rule from Calculus to get the derivative of mu.
```

```
dmu = @(t) (df(t) .* df(t) - f(t) .* dff(t)) ./ (df(t).^2);

while abs(mu(steps(end))) > ERR
        tmp = steps(end);
        nxt = tmp - mu(tmp)/dmu(tmp);
        steps = [steps nxt];
end

% Output the results for Newton's Method
steps
f(steps(end))
plot(steps, f(steps), 'LineWidth', 3, 'DisplayName', 'Modified Newtons Method');

legend('show');
end
```

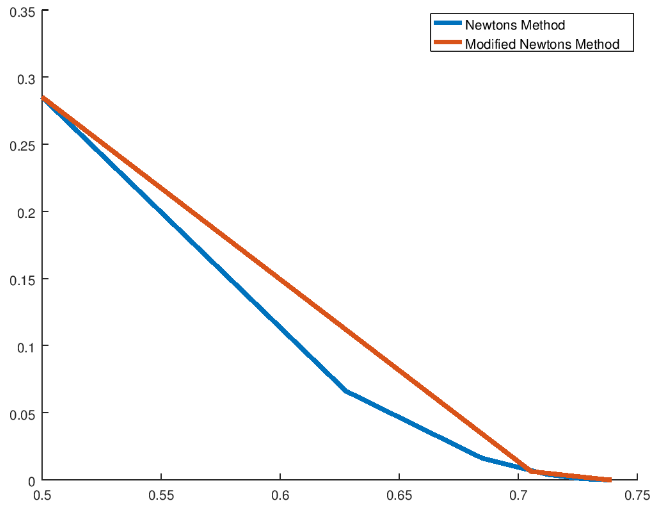
Here are the results we get from running regular Newton's Method to find the zero. From the steps used to find the solution, we can see that Newton's method took 9 steps when using the midpoint to find the zero.

```
steps = [ 0.50000, 0.62761, 0.68489, 0.71233, 0.72579, 0.73246,
        0.73578, 0.73743, 0.73826 ]
zero =  3.8246e-06
```

And here are the results when we use Modified Newton's Method to find the zero. We can see that this method found a result much faster than regular Newton's Method, taking only 5 steps to find the (more accurate) solution.

```
steps = [ 0.50000, 0.70543, 0.73788, 0.73906, 0.73908 ]
ans = 7.2775e-13
```

In the graph below, we can clearly see Modified Newton's Method jumping straight away from 0.5 to 0.7 in one step, something that regular Newton's Method took about 3 steps.



Problem 6.

Use each of the following methods to find a solution in $[0.1, 1]$ accurate to within 10^-4 for

$$600x^4 - 550x^3 + 200x^2 - 20x - 1 = 0.$$

(a) Bisection method

(b) Newton's method

(c) Secant method

(d) Müller's method

Comment on the performance of these methods.

Solution

```
function p6
clear all;
close all;
hold on;

% The function to find 0's for...
f = @(t) 600.*t.^4 - 550.*t.^3 + 200.*t.^2 - 20.*t - 1;
df = @(t) 2400.*t.^3 - 1650.*t.^2 + 400.*t - 20;
% Use the midpoint as our initial guess.
MID = (0.1 + 1.0) / 2.0;
% Want a zero within this error.
ERR = 10^-(4);

%% (a) Bisection Method

int = [0.1 1.0];
steps = [(sum(int)/2.0)];

while abs(f(steps(end))) > ERR
        % Shrink our interval to approach the actual result.
        if ( sign(f(steps(end))) == sign(f(int(1))) )
                int(1) = steps(end);
        else
                int(2) = steps(end);
        end

        % Update the midpoint.
        steps = [steps (sum(int)/2.0)];
end
```

```
% Output the results for the Bisection Method.
steps
f(steps(end))
plot(steps, f(steps), 'LineWidth', 3, 'DisplayName', 'Bisection Method');

%% (b) Newton's Method

steps = [MID];
while abs(f(steps(end))) > ERR
        tmp = steps(end);
        nxt = tmp - f(tmp)/df(tmp);
        steps = [steps nxt];
end

% Output the results for Newton's Method
steps
f(steps(end))
plot(steps, f(steps), 'LineWidth', 3, 'DisplayName', 'Newtons Method');

%% (c) Secant Method

% Initial guess is the midpoint,
% second guess is taken from newtons method
steps = [MID (MID - f(MID)/df(MID))];

while abs(f(steps(end))) > ERR
        tmp = steps(end);
        p1 = steps(end);
        p2 = steps(length(steps)-1);
        nxt = steps(end) - (f(p1) * (p1-p2)) / (f(p1)- f(p2));
        steps = [steps nxt];
end

% Output the results for Secant Method
steps
f(steps(end))
plot(steps, f(steps), 'LineWidth', 3, 'DisplayName', 'Secant Method');

%% (d) Muller's Method

% Initial guess is the midpoint,
% Second & Third guesses are taken from Newtons Method
p1 = MID;
p2 = p1 - f(p1)/df(p1);
```

```
p3 = p2 - f(p2)/df(p2);

% Run Muller's Method.
[steps fval] = muller(f, p1, p2, p3, ERR, 100);

% Output the results for Muller's Method
steps
f(steps(end))
plot(steps, f(steps), 'LineWidth', 3, 'DisplayName', 'Mullers Method');

legend('show');
end
```

The Bisection Method is the slowest solution of these 4 methods, taking a total of 15 steps to find a solution. Further, this solution only barely breaks the required error, indicative of the small jumps taken by this method.

```
steps = [0.55000, 0.32500, 0.21250, 0.26875, 0.24063, 0.22656, 0.23359, 0.23008,
        0.23184, 0.23271, 0.23228, 0.23250, 0.23239, 0.23233, 0.23236]
ans = 6.7369e-05
```

Newton's Method is about twice as fast as the Bisection Method in this example, converging in just 6 steps. Likely due to the simplicity of finding the zero of a polynomial, with an initial guess that was relatively close to what we were looking for, that is, more complex methods were likely not needed in this scenario.

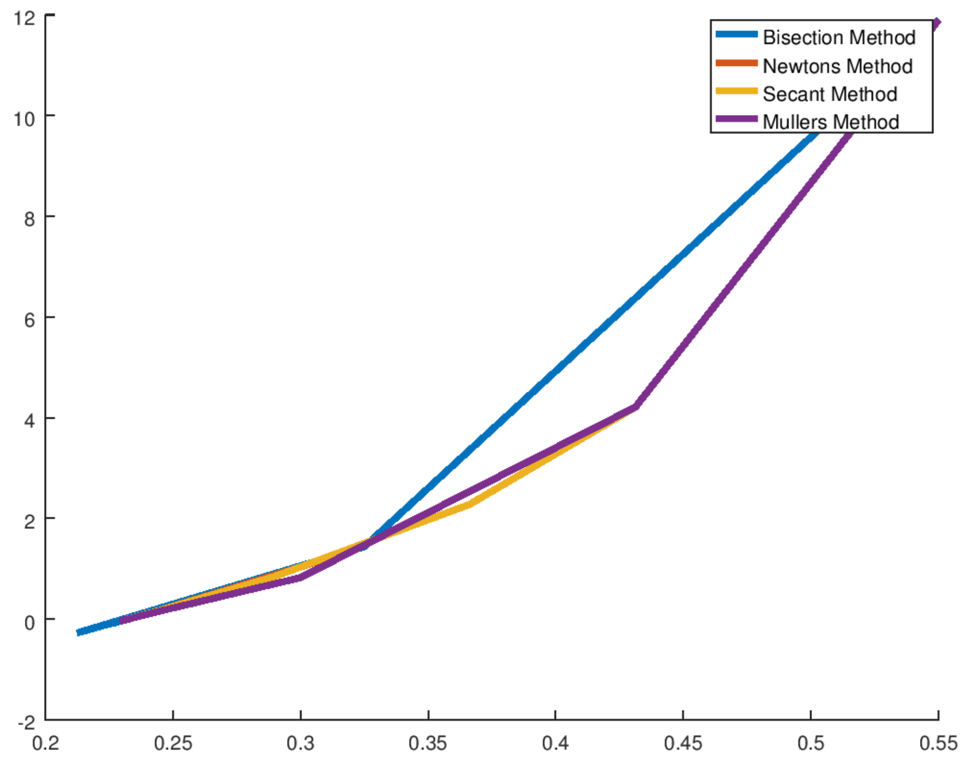
```
steps = [0.55000, 0.43123, 0.32077, 0.24398, 0.23246, 0.23235]
ans = 1.2871e-07
```

The Secant Method found a solution slightly slower than Newton's, taking an additional step, for a total of 7 steps. This does however, include both of the initial guesses, one of which was derived from Newton's method.

```
steps = [0.55000, 0.43123, 0.36617, 0.28961, 0.24453, 0.23301, 0.23236]
ans = 8.8042e-05
```

Müller's Method found it's result in 8 steps, which included 3 initial guesses, 2 of which were found using Newton's method. If these Initial guesses are excluded from the count, only 5 steps of iteration were actually needed. Which would bring tie it with Newton's method. Regardless, this method was not necessarily needed in finding the desired 0, although this function does have complex 0's, we did not actually find one using these initial parameters.

```
steps = [0.55000 + 0.00000i, 0.43123 + 0.00000i, 0.32077 + 0.00000i,
        0.30011 - 0.08715i, 0.24307 + 0.03357i, 0.22940 - 0.00000i,
        0.23240 - 0.00001i, 0.23235 + 0.00000i]
ans = -6.3042e-08 + 1.1083e-07i
```



Problem 7.

An object falling vertically through the air is subjected to viscous resistance as well as the force of gravity. Assume that an object with mass m is dropped from a height s_0 and that the height of the object after t seconds is

$$s(t) = s_0 - \frac{mg}{k}t + \frac{m^2 g}{k^2}(1 - e^{-kt/m}),$$

where $g = 32.17 ft/s^2$ and k represents the coefficient of air resistance in lb-s/ft. Suppose s_0=300ft, $m = 0.25$lb, and $k = 0.1$ lb-s/ft. Find to within 0.01s the time it takes this quarter-pounder to hit the ground. Use Fixed-point iteration, Steffensen's method and Newton's method to find the solution.

Solution

```
function p7
clear all;
close all;
hold on;

%% Initial Parameters
g = 32.17;
m = 0.25;
k = 0.1;
s0 = 300;
ERR = 0.01;

% Initial guess of 5 seconds. Turned out to be super close.
GUESS = 5;

%% Find the 0 of f, using function f,df, and/or g.
f = @(t) s0 - (m*g/k).*t + (m*m*g/(k*k)).*(1 - exp(-k.*t./m));
df = @(t) (m*g*(exp(-k.*t./m)-1)) ./ k;
g = @(t) (s0 + m^2*g/(k^2)*(1-exp(-k*t/m))) * k/(m*g);

%% Fixed-Point Iteration

steps = [GUESS];
while abs(f(steps(end))) > ERR
        tmp = steps(end);
        steps = [steps g(tmp)];
end

% Output the results for Fixed-Point Iteration
steps
```

```
f(steps(end))
plot(steps, f(steps), 'LineWidth', 3, 'DisplayName', 'Fixed-Point Iteration');

%% Steffensen's Method

steps = [GUESS];
while abs(f(steps(end))) > ERR
        p0 = steps(end);
        p1 = g(p0);
        p2 = g(p1);
        p = p0 - (p1-p0)^2 / (p2 - 2*p1 + p0);
        steps = [steps p];
end

% Output the results for Steffensen's Method
steps
f(steps(end))
plot(steps, f(steps), 'LineWidth', 3, 'DisplayName', 'Steffensens Method');

%% Newton's Method

steps = [GUESS];
while abs(f(steps(end))) > ERR
        tmp = steps(end);
        nxt = tmp - f(tmp)/df(tmp);
        steps = [steps nxt];
end

% Output the results for Newton's Method
steps
f(steps(end))
plot(steps, f(steps), 'LineWidth', 3, 'DisplayName', 'Newtons Method');

legend('show');
end
```

Fixed-Point Iteration was the slowest of the three methods I used, but thanks to a decent initial guess, and a relatively large error, none of the methods took particularly long to find a solution. In this case, only 5 steps were needed.

```
steps = [5.0000, 5.8918, 5.9934, 6.0028, 6.0036]
ans = 0.0062343
```

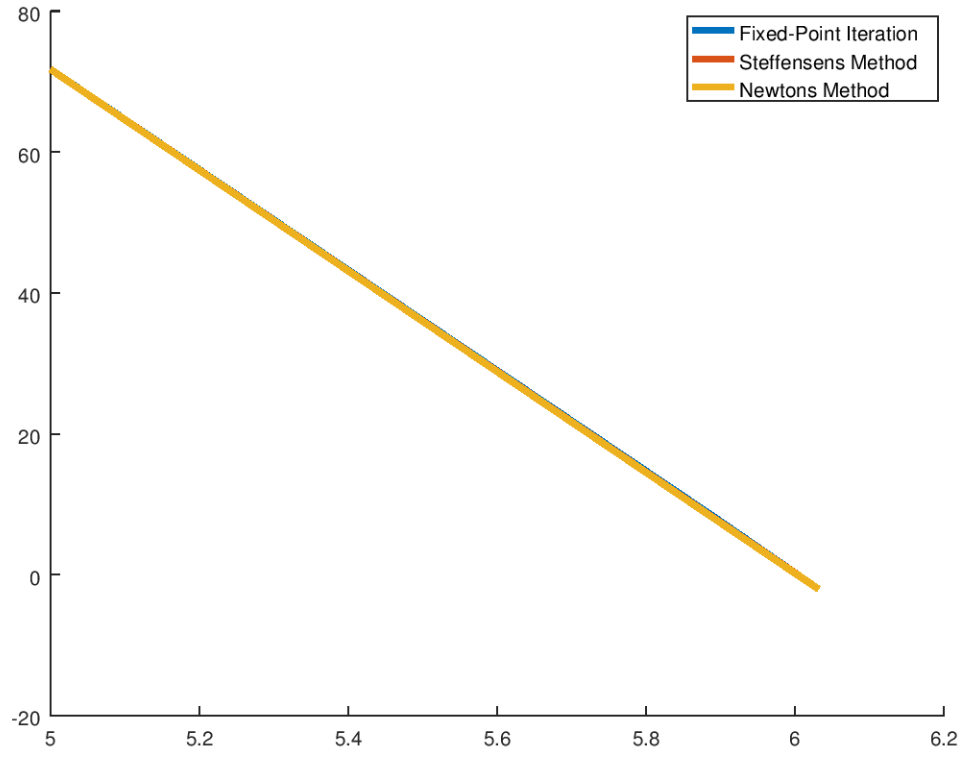
Steffensen's Method found a solution in only 3 steps, just as fast as Newton's method. However, Steffensen's method produced a notably much closer result, and had much greater accuracy than the other two methods.

```
steps = [5.0000, 6.0064, 6.0037]
ans = -9.4342e-07
```

Like Steffensen's Method, Newton's Method found a solution in only 3 steps, however as noted above the result was not as accurate as the one found by Steffensen's Method, although the result was still better than Fixed-Point Iteration. This may have to do with the fact that the solution was found so quickly, and may not be indicative of long term behavior.

```
steps = [5.0000, 6.0314, 6.0037]
ans = -0.0011092
```

In the graph below, we can see that although the number of steps varied, the path took in all these methods was roughly the same, this again is likely due to the initial guess being reasonably close to the desired solution.


Fixed-Point Iteration
Steffensens Method
Newtons Method