CX 4640 Assignment 2

Wesley Ford

August 29th, 2020

1. Chapter 3, question 7, on page 60. For part (a), show the result analytically. For part (b),you can define efficiency in terms of convergence per evaluation of the function f.

   Steffensens's method uses the following formula for finding roots:

$$x_{n+1} = x_n - \frac{f(x_n)}{g(x_n)}$$

   where

$$g(x) = \frac{f(x + f(x)) - f(x)}{f(x)}$$

   Expressing x_{n+1} using only $f(x)$, we get:

$$x_{n+1} = x_n - \frac{f(x_n)^2}{f(x_n + f(x_n)) - f(x_n)}. \tag{1}$$

   The error for the n^{th} term is $e_n = x_n - r$, where r is the root of the equation $f(x)$. Applying this to the $n^{th} + 1$ term:

$$e_{n+1} = x_{n+1} - r = x_n - \frac{f(x_n)^2}{f(x_n + f(x_n)) - f(x_n)} - r$$

$$e_{n+1} = e_n - \frac{f(x_n)^2}{f(x_n + f(x_n)) - f(x_n)}$$

   Using Taylor's series to approximate $f(x_n + f(x_n))$ centered at the point x_n

$$f(x_n+f(x_n)) = f(x_n)+f'(x_n)(x_n+f(x_n)-x_n)+f''(\zeta_1)\frac{(x_n + f(x_n) - x_n)^2}{2} = f(x_n)+f'(x_n)f(x_n)+f''(\zeta_1)\frac{f(x_n)^2}{2}$$

   with $x_n \leq \zeta_1 \leq x_n + f(x)$. Substituting this back into equation (1), we get

$$e_{n+1} = e_n - \frac{f(x_n)^2}{f(x_n)f'(x_n) + \frac{1}{2}f(x_n)^2 f''(\zeta_1)}$$

$$e_{n+1} = e_n - \frac{f(x_n)}{f'(x_n) + \frac{1}{2}f(x_n)f''(\zeta_1)}$$

$$e_{n+1} = \frac{e_n[f'(x_n) + \frac{1}{2}f(x_n)f''(\zeta_1)] - f(x_n)}{f'(x_n) + \frac{1}{2}f(x_n)f''(\zeta_1)} \tag{2}$$

   Using Taylor's series to approximate $f(r)$ at point x_n:

$$f(r) = 0 = f(x_n) + f'(x_n)(r - x_n) + \frac{1}{2}(r - x_n)^2 f''(\zeta_2)$$

$$\frac{1}{2}e_n^2 f''(\zeta_2) = e_n f'(x_n) - f(x_n)$$

where $r \leq \zeta_2 \leq x_n$ Substituting into equation (2), we have

$$e_{n+1} = \frac{\frac{1}{2}e_n^2 f''(\zeta_2)) + \frac{1}{2}f(x_n)f''(\zeta_1)}{f'(x_n) + \frac{1}{2}f(x_n)f''(\zeta_1))}$$

When x_n is near r,

$$e_{n+1} = \frac{\frac{1}{2}e_n^2 f''(r) + \frac{1}{2}f(r)f''(\zeta_1)}{f'(r) + \frac{1}{2}f(r)f''(\zeta_1))}$$

$$e_{n+1} = \frac{1}{2}\frac{f''(r)}{f'(r)}e_n^2$$

As long as $f'(r) \neq 0$, so that we don't have $f'(r) = 0$, which near the root would produce large coefficients and overcome squaring the previous error, we see that this method does have quadratic convergance.

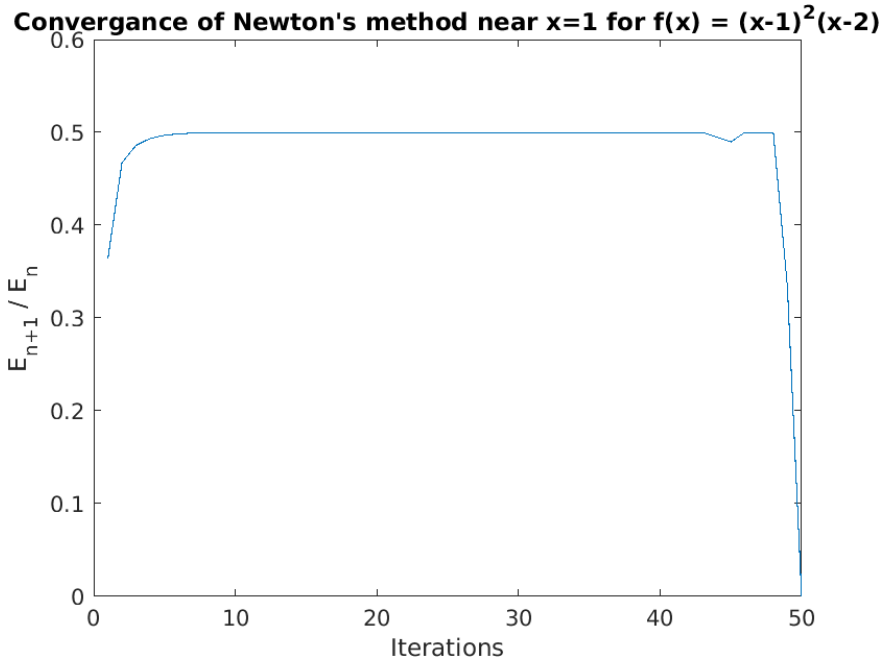
The secant method only takes 1 function evaluation per step, while Steffensen's method requires two, $f(x_n)$ and $f(x_n + f(x_n))$. In lecture, we calculated the error for two function evaluations of $f(x)$ using the secant method to be $e_{n+2} = c^{2.618}e_n^{2.618}$. The same number of function evaluations for Steffensen's method corresponds to $e_{n+1} = ce_n^2$. So while Steffensen's method may converge faster per step, the secant method is more efficient when counting function evaluations.

2. What happens in Newton's method when $f'(x) = 0$ at the root, i.e., the derivative is also zero? This question explores this case.

   i. Consider $f(x) = 0$ for $f(x) = (x-1)^2 * (x-2)$ which has a double root at $x = 1$. Note that $f'(1) = 0$. Implement Newton's method to solve for the root at $x = 1$ using a nearby initial guess. What is the order of convergence, q? If convergence is linear, what is the reduction factor, c, per iteration?

```
root = 1;
initial_x = 1.3
x1 = [initial_x];
errors = [abs(x1-root)];
iterations = 1e5;
%Using Newton's method to find root near x=1
convergance = [];
for i = 1:iterations
    x = x1(i);
    if f(x) == 0
        break;
    end
    x = x-f(x)/f_prime(x);
    errors = [errors, abs(x-root)];
    x1 = [x1, x];
    convergance = [convergance, errors(end) / errors(end-1)];
end
```

   Using an initial guess of $x = 1.3$, we can see that it takes approximately 50 iterations for Newton's method to converge onto the correct root $x = 1$. Plotting the ratio between e_{n+1} and e_n, we notice that the order of convergence is liner with a reduction factor of $c = 0.5$. In other words, around $x = 1$, the error between x_n and the root $x = 1$ is halved each iteration.



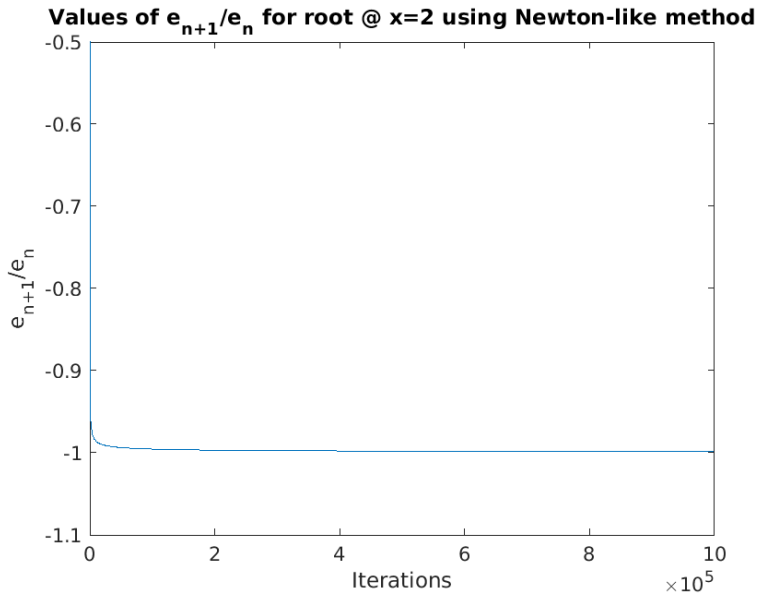
ii. Consider the Newton-like method $x_{n+1} = x_n - 2f(x)/f'(x)$ which differs from Newton's method by a factor of 2 on $f(x)$. Implement this method and solve for the double root in $f(x)$ above. Show that the convergence is quadratic.

Using an initial guess of $x_0 = 0$, and the Newton-like method describe above, we even with a larger error in our initial guess this Newton-like method converges to the root using $\frac{1}{10}$ the number of iterations as Newton's method. We can compute the value of q for error convergence. We know that $e_{n+1} \leq Me_n^q$. Solving for q, we get $\frac{\ln e_{n+1}}{\ln Me_n} = q$. Using matlab to compute values for $\frac{\ln e_{n+1}}{\ln e_n}$, we see that q gets closer to 2 after each iteration until the function actually converges, implying that the function does follow quadratic convergence.

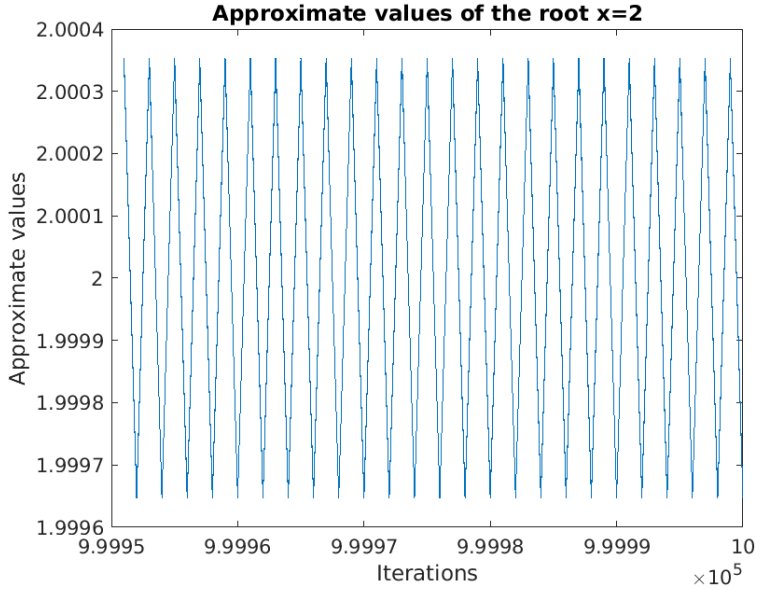
n	$\frac{\ln e_{n+1}}{\ln Me_n}$
0	2.2304
1	2.1367
2	2.0632
3	Inf

iii. Use your implementation of this Newton-like method to find the simple root near $x = 2$. What is the order of convergence?

Using the Newton-like method for the root $x = 2$, we find that the order of convergence is again linear, however with a c value approaching -1. Plotting values of e_{n+1}/e_n. we can see that the error convergence very quickly approaches -1.



This implies that as the number of iterations increases, the error in the guess stabilizes and our approximation will oscillate around the root. Plotting the last 50 iterations of our approximation (10^6 iterations were used), we can see that this is indeed what happens. Our approximation oscillates between 2.00035 and 1.99965.

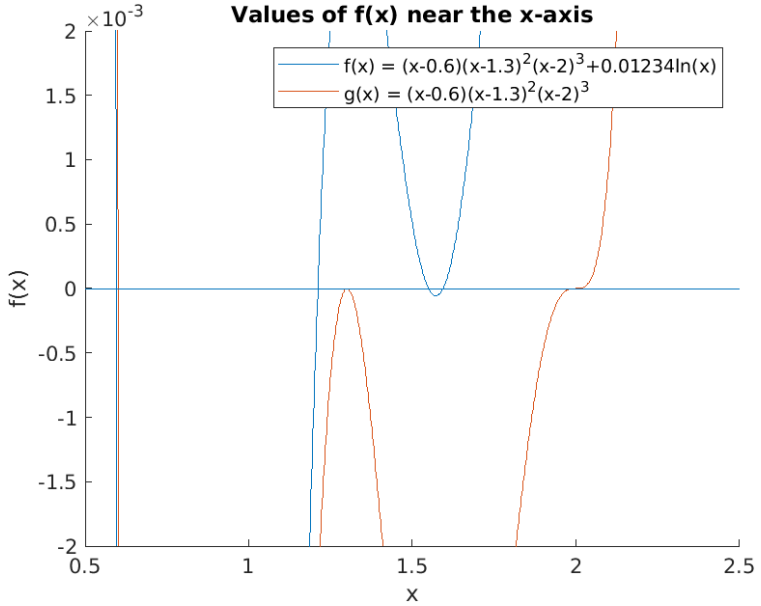


From this we can see that while the approximation does not converge, it does have linear convergence, with a convergence factor of -1 neat the root $x = 2$.

3. This question will require you to use Matlab's fzero function. Type help fzero or search Matlab documentation on the web for information on fzero.

   i. Use Matlab's fzero function to find all the roots of $(x-0.6)(x-1.3)^2(x-2)^3 + 0.01234 \ln x = 0$. Note that you can set the relative error tolerance on x and to turn on the display of each iteration.

   We can figure out how many real roots f(x) should have by looking at the function $g(x) = (x-0.6)(x-1.3)^2(x-2)^3$. $g(x)$ clearly has 3 distinct roots at $x = 0.6, x = 1.3$ and $x = 2$. Near the roots $x = 0.6$ and $x = 2$, the function behaves linearly and cubical, respectively. This is important because by adding a small offset to $g(x)$, we may change the value of those roots, but we don't actually change the number of roots near those values. Near $x = 1.3$ however, the function behaves like a parabola, where the function is negative on both sides of the root. When adding a small positive offset ends up creating 4 distinct roots instead of 3. Plotting both $f(x)$ and $g(x)$, we see that this is the case. The addition of $0.01234 \ln(x)$ is enough to cause an additional root to appear, but small enough not to push the entire function above the x-axis.



   Looking at the plot for $f(x)$, we can use initial points $x_0 = 0.5, x_0 = 1.2. x_0 = 1.5, x_0 = 1.6$ and the fzero function with a tolerance of $1e-6$ to generate four distinct roots.

Root Number	X_0	Approximate Value
1	0.5	0.59534
2	1.2	1.21121
3	1.5	1.54950
4	1.6	1.59331

   ii. Compute the value of the root near 0.5 as accurately as possible. What is this value printed with 15 significant digits? Type help format to see how to display numbers using 15 digits. Explain why you think this is the most accurate value of the root.

   We can use fzero to again approximate the root near 0.5. We get $5.953494869353895e-01$. To test if this is the most accurate approximation of the root, we can add the smallest value of precision available. When we do this, we see that even a single addition of machine epsilon causes the function evaluation to switch signs, and produce a value further away from zero then the guess provided by fzero.

```
%2b
%Use fzero to get an approximate value
root = fzero(f, 0.5)
f1 = f(root)
%Add the smallest possible value at this level of prescion
f2 = f(root + eps(root))
```

```
root =
     5.953494869353895e-01
f1 =
     2.949029909160572e-17
f2 =
    -1.231653667943533e-16
```

While there are clearly smaller numbers which are closer to the actual root than $5.953494869353895e-01$, IEEE double precision is unable to represent those numbers in an accurate way, and so this value is the best we can approximate the root near 0.5 using double precision.

iii. Find the root near 0.5 using different values of the tolerance tolX and fill in the following table. To compute the relative error, assume the value computed in part (b) is the exact value of the root.

tolX	Root Value	number of function evals.	relative error
default	5.953494869353895e-01	22	0
1e-6	5.953496821178298e-01	20	3.27845e-05%
1e-3	5.945149299255330e-01	18	1.40179e-01 %
1e-1	6.131370849898476e-01	15	6.13137e-01 %

iv. Comment on how the number of function evaluations varies with the relative error, based on what you know of how the fzero method works.

The fzero method works by first finding a bracket which contains the zero, then uses the secant method to find a zero in that bracket. When the tolerance is set to $1e-1$, the fzero functions does not use interpolation methods at all. For all tolerance levels, at least 15 function calls are made to find the correct bracket. After this though, only 1 is required to verify that the f(x) value is within the tolerance limit.

```
Search for an interval around 0.5 containing a sign change:
 Func-count    a          f(a)             b          f(b)        Procedur
     1            0.5      0.207447           0.5      0.207447   initial
     3       0.485858      0.253724      0.514142      0.165731   search
     5           0.48      0.274304          0.52      0.149715   search
     7       0.471716      0.304883      0.528284       0.12827   search
     9           0.46      0.351203          0.54       0.10025   search
    11       0.443431       0.42321      0.556569     0.0649588   search
    13           0.42      0.539101          0.58     0.0229646   search
    15       0.386863      0.734291      0.613137    -0.0225689   search

Search for a zero in the interval [0.386863, 0.613137]:
 Func-count    x          f(x)             Procedure
    15       0.613137    -0.0225689        initial

Zero found in the interval [0.386863, 0.613137]

root4 =
     6.131370849898476e-01
```

When requiring a tighter tolerance, more function evaluations are required. These function evaluations are for the secant method though, not to establish a bracket. Because of this, as the number of function evaluations grow, the error becomes reduced significantly, because for any function evaluation past 15, the error in our estimation is being reduced superlinearly. If we look at the difference between using a tolerance of $1e-3$ and $1e-6$, we notice that there are only 2 more function evaluations, but because fzero is now using the secant method, those 2 function evaluations drastically improve the relative error of the approximate value.

Matlab Code

```
%Code for problem 2
%f(x) and f'(x)
f = @(x) (x-1).^2 .* (x-2);
f_prime = @(x) 2.*(x-1) .* (x-2) + (x-1).^2;

root = 1;
initial_x = 1.3
x1 = [initial_x];
errors = [abs(x1-root)];
iterations = 1e5;
%Using Newton's method to find root.
convergance = [];
for i = 1:iterations
    x = x1(i);
    if f(x) == 0
        break;
    end
    x = x-f(x)/f_prime(x);
    errors = [errors, abs(x-root)];
    x1 = [x1, x];
    convergance = [convergance, errors(end) / errors(end-1)];
end

plot(1:length(convergance), convergance);
title("Convergance of Newton's method near x=1 for f(x) = (x-1)^2(x-2)")
xlabel("Iterations")
ylabel("E_{n+1} / E_{n}")
ylim([0,.6]);

newton_like1 = newton_like(1.1, f, f_prime);
newton_like2 = newton_like(2.3, f, f_prime);

error_newton_like1 = abs(newton_like1 - 1);
error_newton_like2 = newton_like2 -2;

convergance_newton_like1 = error_newton_like1(2:end) ./ error_newton_like1(1:end-1);
convergance_newton_like2 = error_newton_like2(2:end) ./ error_newton_like2(1:end-1);

q1 =  log(error_newton_like1(2:end)) ./ log(error_newton_like1(1:end-1));



x=1:length(convergance_newton_like1);
p = polyfit(x, convergance_newton_like1,2);
x1 = linspace(1,length(convergance_newton_like1));
y1 = polyval(p, x1);

%make plots
hold off
plot(1:length(convergance_newton_like1), convergance_newton_like1);
title(["Convergance of Newton-like method near x=1 for f(x)=(x-1)^2(x-2)"])
```

```
xlabel("Iterations")
ylabel("e_{n+1}  / e_{n}");
hold on
plot(x1, y1);
legend("e_{n+1} / e_{n}", "Quadratic fit")
hold off;

x = 1:1e3:length(convergance_newton_like2);
plot(x, convergance_newton_like2(x))
title("Values of e_{n+1}/e_{n} for root @ x=2 using Newton-like method")
xlabel("Iterations")
ylabel("e_{n+1}/e_{n}")
ylim([-1.1, -.5]);
last50 = length(convergance_newton_like2)-49:length(convergance_newton_like2);
plot(last50,newton_like2(last50));
title("Approximate values of the root x=2");
xlabel("Iterations")
ylabel("Approximate values")


function xs = newton_like(guess, f, f_prime)
%Implementation of Newton-like method with 1e6 iterations.
    iterations = 1e6;
    xs = [guess];
    for i=1:iterations
        x = xs(i);
        if f(x) == 0
            break
        end
        x = x-2*f(x)./f_prime(x);
        xs = [xs, x];
    end
end
```

```
%Code for problem 3

%functions

f = @(x) (x-0.6).*(x-1.3).^2.*(x-2).^3+0.01234.*log(x);
g = @(x) (x-0.6).*(x-1.3).^2.*(x-2).^3
%Plotting the function
hold on
x = linspace(0.3, 2.2, 1000);
plot(x, f(x))
plot(x, g(x));
ylim([-2e-3, 2e-3])
line(xlim, [0 0])
title("Values of f(x) near the x-axis")
ylabel("f(x)");
xlabel("x");
legend("f(x) = (x-0.6)(x-1.3)^2(x-2)^3+0.01234ln(x)", "g(x) = (x-0.6)(x-1.3)^2(x-2)^3 ")

%2a, find 4 roots near 0.5, 1.2, 1.5, 1.6
r1 = fzero(f, 0.5, optimset('Tolx', 1e-6));
r2 = fzero(f, 1.2, optimset('Tolx', 1e-6));
r3 = fzero(f, 1.5, optimset('Tolx', 1e-6));
r4 = fzero(f, 1.6, optimset('Tolx', 1e-6));

r = [r1,r2,r3,r4]

ftmp = @(x) (x.^2 -2.6.*x + 1.2).*(x.^2-2.6.*x+1.69).*(x^2-4.*x+4)+0.01234.*log(x);
%3b
%Use fzero to get an approximate value, using eps as the tolerance
root = fzero(f, 0.5, optimset('TolX', eps))
f1 = f(root)
%Add one machine epsilon
f2 = f(root + eps(root))

root1 = fzero(f, 0.5, optimset('display', 'iter'))
root2 = fzero(f, 0.5, optimset('display', 'iter', 'tolX', 1e-6))
root3 = fzero(f, 0.5, optimset('display', 'iter', 'tolX', 1e-3))
root4 = fzero(f, 0.5, optimset('display', 'iter', 'tolX', 1e-1))

roots = [root1, root2, root3, root4];
rel_err = abs(roots - root) / root
```