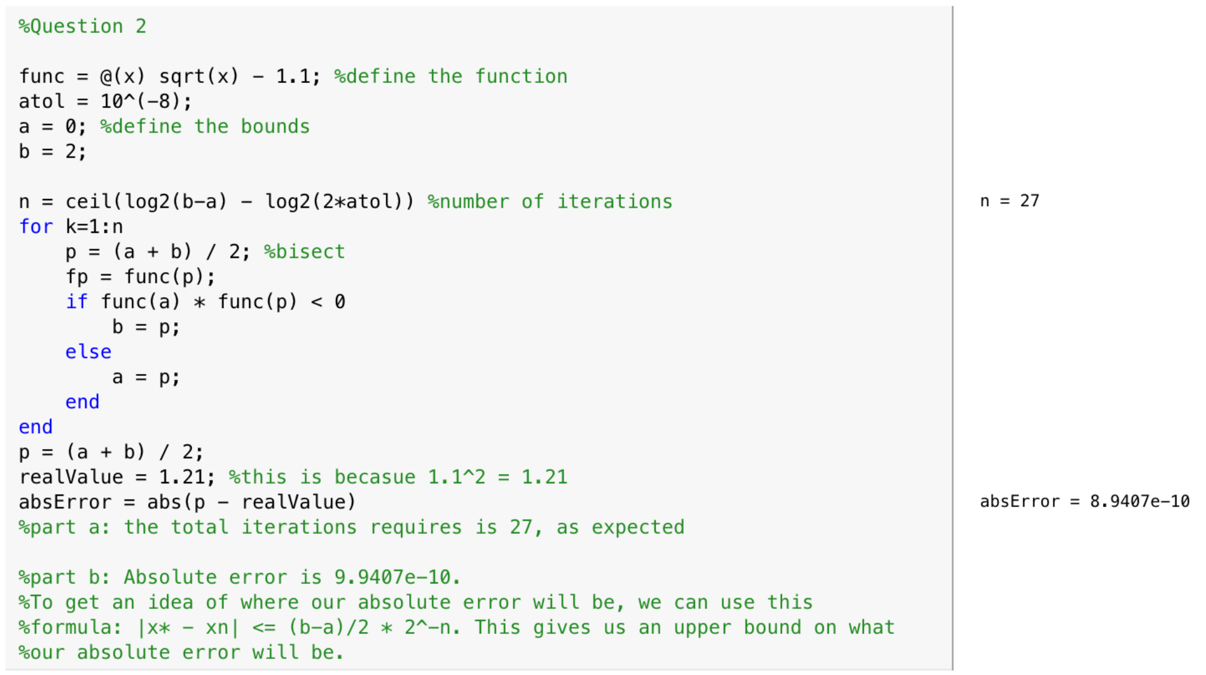
Nathanael Jo, Edward Holappa, Griffin Weinhold

MATH 458 Homework 3

1

a) i) The bisection method is not nearly as efficient as other algorithms, it is linear. The total iterations needed is the log base 2 of (b-a)/(2*atol). This is because the interval, which starts as [a,b] shrinks by a factor of 2 each iteration. ii) The method is robust for functions in relatively lower dimensions, and as long as the function is continuous. iii) All the knowledge it requires is the function itself, f(x), and a confirmation of its continuity. iv) The bisection method is difficult to generalize into higher dimensions
b) **Newton:** i) It is efficient, in fact, it converges quadratically. ii) Newton's method is relatively robust, provided that the function satisfies the proper criteria and the randomly selected point, x0, it close to a root. iii) This does not require a minimal amount of knowledge. We need to know how to evaluate the function and its derivative, and we also need the function to be continuous and smooth. iv) Yes, newton's method simply requires the function to be smooth and continuous. v) Newton's method can be applied to higher dimensions.
   Secant: i) This method converges superlinearly, so it is efficient. ii) It is most robust than newton's method, because we don't need to know how to evaluate the derivative of the function f. iii) It requires the function to be continuous and differentiable. iv) The function has to be differentiable. v) The secant method can also be applied to higher dimensions.
c) Advantages: In the case of one root, Newton's method converges quadratically. It can be generalized to higher dimensions. Disadvantages: we need to know the derivative of f exists and also how to evaluate it. The method's convergence is relatively local.
d) The order of convergence describes the "magnitude" of the convergence rate, if the function is linearly, quadratically, or superlinearly convergent. This can more generally define a method's efficiency. The rate of convergence is more specific, and applies to certain problems. It describes exactly how quickly a method can converge on a certain problem.
e) Newton's method converges linearly in the case of multiple roots, i.e, f'(x*) = 0.
f) Roundoff errors are generally negligible so long as the termination criteria of the iterative method is well above the rounding unit. If the convergence error is very small, however, rounding errors may no longer be negligible.

2



```
%Question 2

func = @(x) sqrt(x) - 1.1; %define the function
atol = 10^(-8);
a = 0; %define the bounds
b = 2;

n = ceil(log2(b-a) - log2(2*atol)) %number of iterations
for k=1:n
    p = (a + b) / 2; %bisect
    fp = func(p);
    if func(a) * func(p) < 0
        b = p;
    else
        a = p;
    end
end
p = (a + b) / 2;
realValue = 1.21; %this is becasue 1.1^2 = 1.21
absError = abs(p - realValue)
%part a: the total iterations requires is 27, as expected

%part b: Absolute error is 9.9407e-10.
%To get an idea of where our absolute error will be, we can use this
%formula: |x* - xn| <= (b-a)/2 * 2^-n. This gives us an upper bound on what
%our absolute error will be.
```

```
n = 27
```

```
absError = 8.9407e-10
```

3.

a. The fixed points are ¼, and ¾.

b. We can be sure that the iterations will converge to the fixed point ¼ . We can conclude this since g'(¼) = ½ which is less than 1. Which shows that ¼ is attracting. On the other hand, g'(¾) = 3/2 which is greater than 1 therefore it does not converge. Furthermore, as k increases for an x_k with a starting guess x_0 as ½ we get that x proceeds to get smaller and smaller. Which then makes the x^2 portion of g(x) gets quadratically smaller and converges at 1/16. This then leads $g(x_k)$ to converge to 4/16 which is equivalent to 1/16.

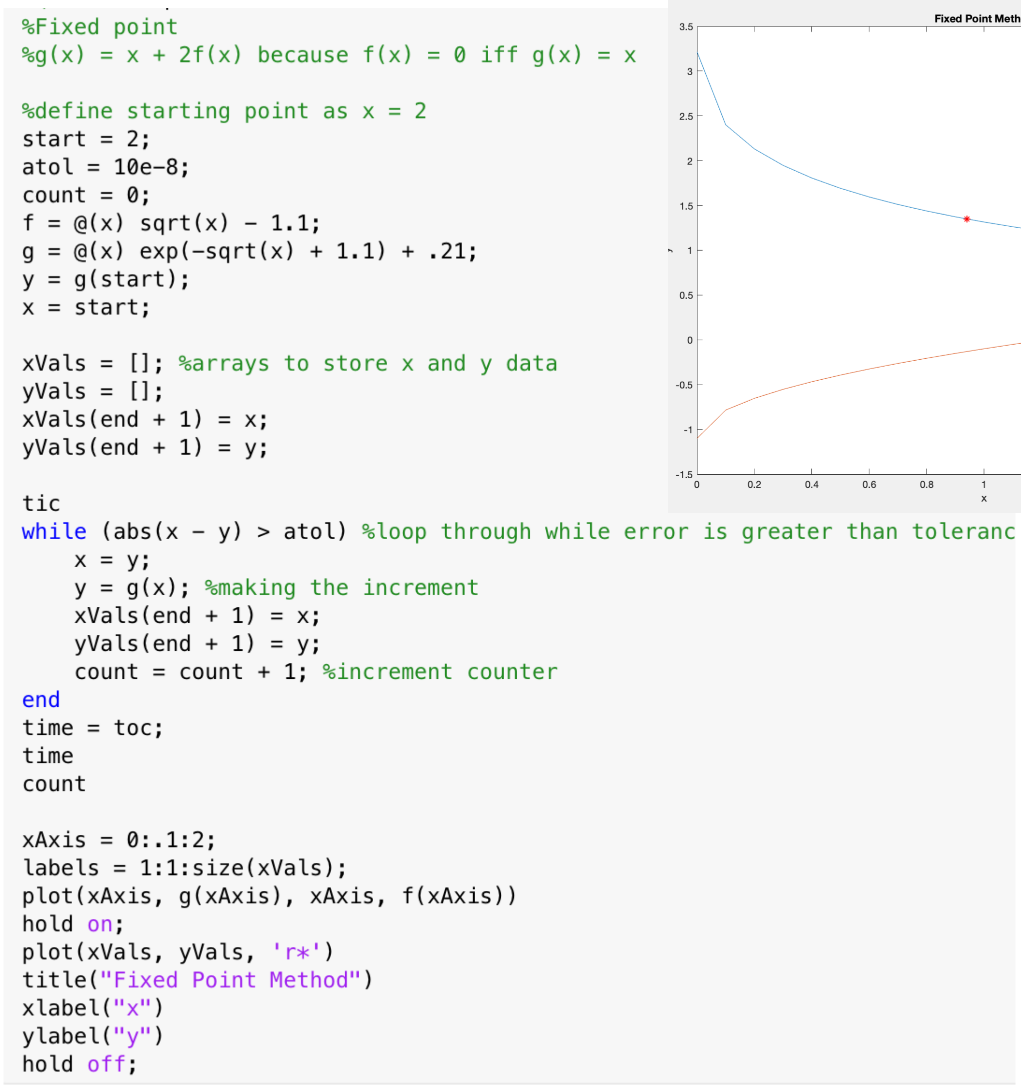
c. Code for 3

```
x = 0.5; %guess
t = 1; %tolerance


xi = x;
count = 0;
while t > 1e-8
   x = fun(xi);
   t = abs(x - xi);
   xi = x;
   count = count + 1;
end

disp(x) %fixed point
disp(count) %count of iterations


function f = fun(l)
   f = l^2 + (3/16);
end
```



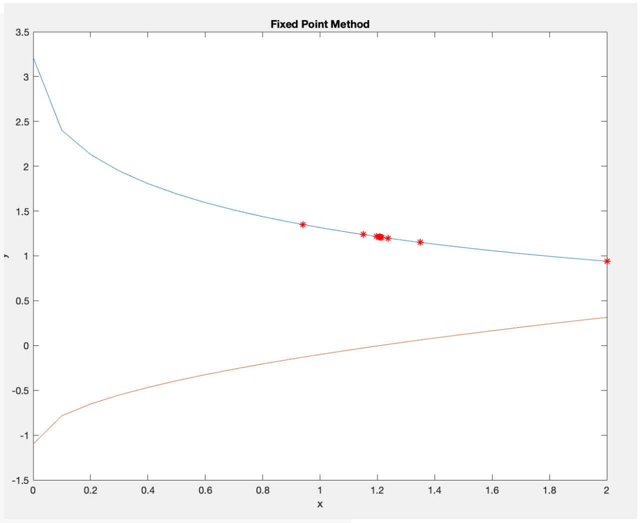
```
%Fixed point
%g(x) = x + 2f(x) because f(x) = 0 iff g(x) = x

%define starting point as x = 2
start = 2;
atol = 10e-8;
count = 0;
f = @(x) sqrt(x) - 1.1;
g = @(x) exp(-sqrt(x) + 1.1) + .21;
y = g(start);
x = start;

xVals = []; %arrays to store x and y data
yVals = [];
xVals(end + 1) = x;
yVals(end + 1) = y;

tic
while (abs(x - y) > atol) %loop through while error is greater than toleranc
    x = y;
    y = g(x); %making the increment
    xVals(end + 1) = x;
    yVals(end + 1) = y;
    count = count + 1; %increment counter
end
time = toc;
time
count

xAxis = 0:.1:2;
labels = 1:1:size(xVals);
plot(xAxis, g(xAxis), xAxis, f(xAxis))
hold on;
plot(xVals, yVals, 'r*')
title("Fixed Point Method")
xlabel("x")
ylabel("y")
hold off;
```



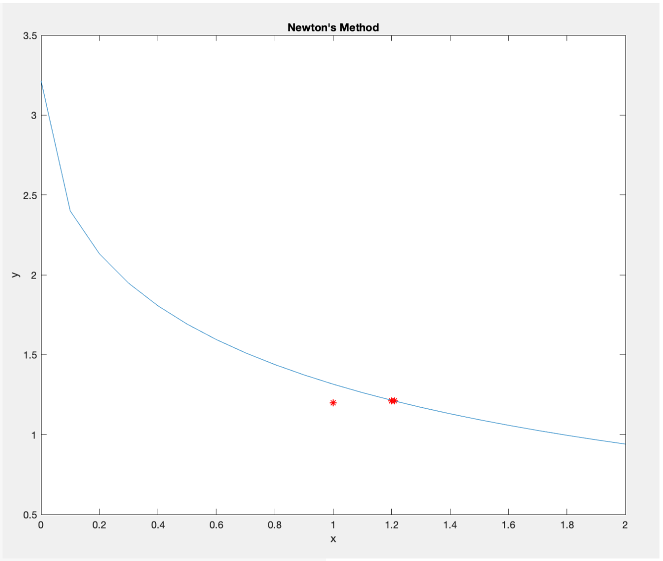
```
%Newton's method

xVal = [];
yVal = [];

tol = 1;
n = 0;
nmax = 25;
x = 2.0;
tic;
while tol >= 1e-8 & n < nmax
    y = x - (x^0.5 - 1.1)/(0.5*x^(-0.5));
    xVal(end+1) = x;
    yVal(end+1) = y;
    tol = abs(y-x);
    x = y;
    n = n+1;
end
time = toc;
time
x

n
plot(xAxis, g(xAxis));
title("Newton's Method");
xlabel("x");
ylabel("y");
hold on;
plot(xVal, yVal, "r*");

%The bisection method has 27 iteration, the fixed point had 21 iterations,
%and netwton's method had 4 iterations. The bisection converges linearly,
%the fixed point also converged linearly, while newton's method converged
%quadratically.
```



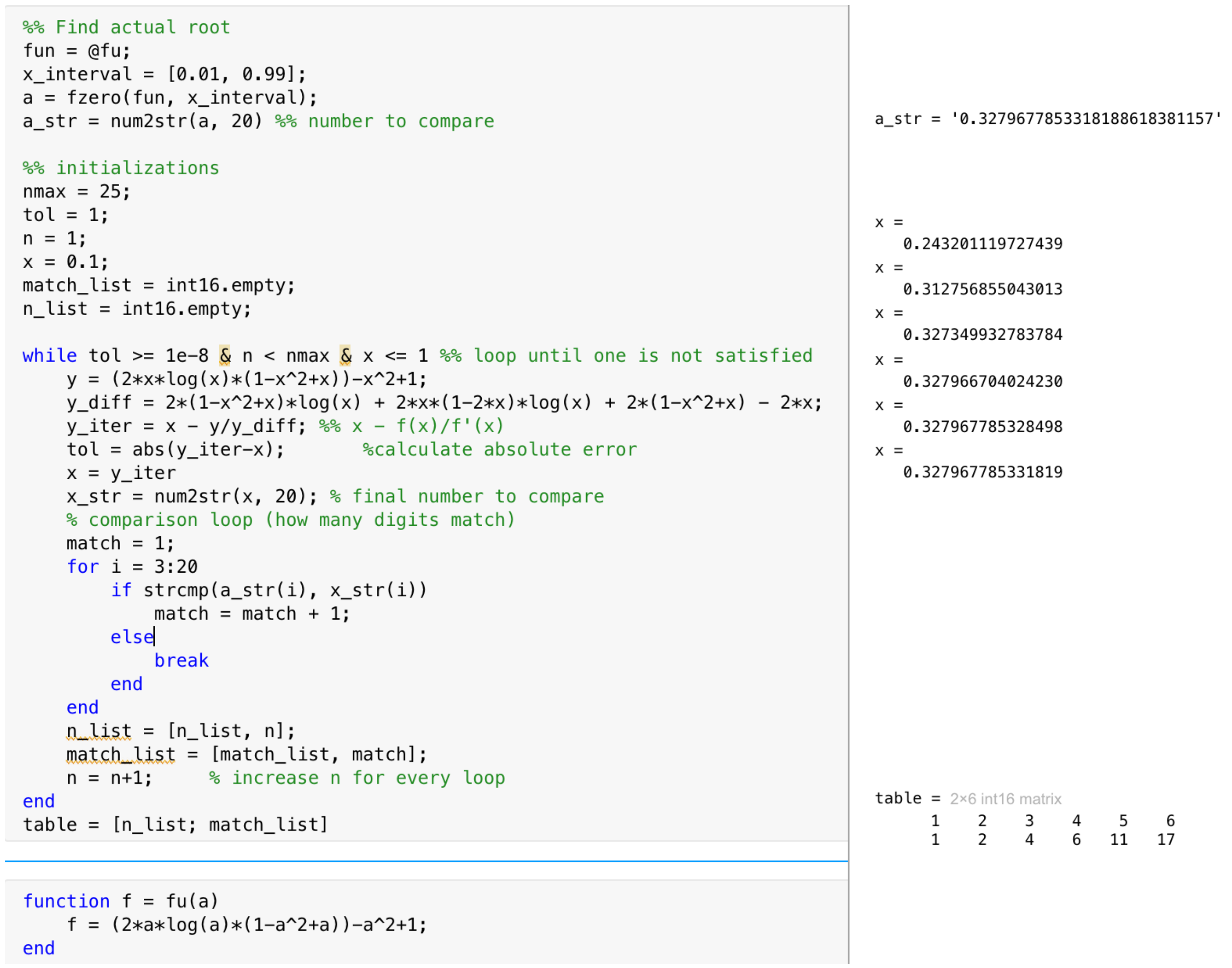
5.

```
nmax=20;        % max number of iterations
tol=1;          % initializing tolerance/error
n=0;            % initializing # of iterations
xvals =  double.empty;

for x = 0.7:1:2.0 % start from a reasonable point knowing the shape of graph
    x_temp = x;
    while tol >= 1e-8 & n < nmax     % loop until tolerance is lower than 1e-8 or nmax
        y = x-((x^4+2*(x^3)-7*(x^2)+3)/(4*(x^3)+6*(x^2)-14*x)); % x_k - f(x)/f'(x)
        tol = abs(y-x);       %calculate absolute error
        x_temp = y;
        n = n+1;     % increase n for every loop
    end
    xvals = [xvals, x_temp];    % append x roots to array of values
end
round(xvals, 2, 'significant') %round to 2 significant figures
```

```
             1                    2
1   0.790000000000000    1.700000000000000
```

6.



```
%% Find actual root
fun = @fu;
x_interval = [0.01, 0.99];
a = fzero(fun, x_interval);
a_str = num2str(a, 20) %% number to compare

%% initializations
nmax = 25;
tol = 1;
n = 1;
x = 0.1;
match_list = int16.empty;
n_list = int16.empty;

while tol >= 1e-8 & n < nmax & x <= 1 %% loop until one is not satisfied
    y = (2*x*log(x)*(1-x^2+x))-x^2+1;
    y_diff = 2*(1-x^2+x)*log(x) + 2*x*(1-2*x)*log(x) + 2*(1-x^2+x) - 2*x;
    y_iter = x - y/y_diff; %% x - f(x)/f'(x)
    tol = abs(y_iter-x);       %calculate absolute error
    x = y_iter
    x_str = num2str(x, 20); % final number to compare
    % comparison loop (how many digits match)
    match = 1;
    for i = 3:20
        if strcmp(a_str(i), x_str(i))
            match = match + 1;
        else
            break
        end
    end
    n_list = [n_list, n];
    match_list = [match_list, match];
    n = n+1;      % increase n for every loop
end
table = [n_list; match_list]
```

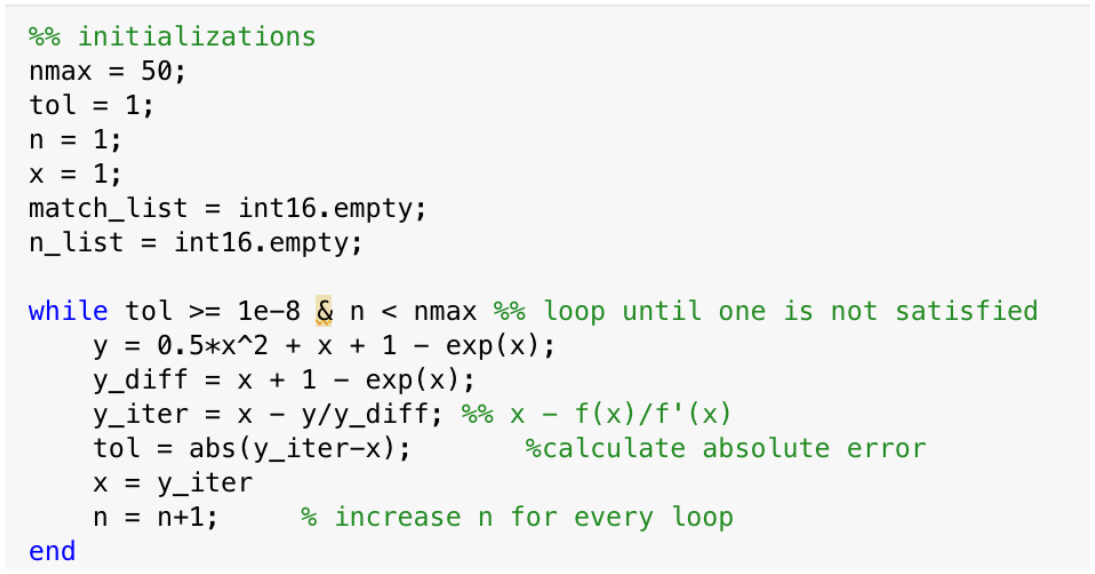
```
function f = fu(a)
    f = (2*a*log(a)*(1-a^2+a))-a^2+1;
end
```

```
a_str = '0.32796778533181886181381157'

x =
   0.243201119727439
x =
   0.312756855043013
x =
   0.327349932783784
x =
   0.327966704024230
x =
   0.327967785328498
x =
   0.327967785331819

table = 2×6 int16 matrix
     1     2     3     4     5     6
     1     2     4     6    11    17
```

7.



```
%% initializations
nmax = 50;
tol = 1;
n = 1;
x = 1;
match_list = int16.empty;
n_list = int16.empty;

while tol >= 1e-8 & n < nmax %% loop until one is not satisfied
    y = 0.5*x^2 + x + 1 - exp(x);
    y_diff = x + 1 - exp(x);
    y_iter = x - y/y_diff; %% x - f(x)/f'(x)
    tol = abs(y_iter-x);       %calculate absolute error
    x = y_iter
    n = n+1;      % increase n for every loop
end
```

```
x =
   0.696105595588666
x =
   0.478112902284116
x =
   0.325285123718804
x =
   0.219857803679102
x =
   0.147933877018087
x =
   0.099236404328835
x =
   0.066432949095211
x =
   0.044411765309888
x =
   0.029662794176641
x =
   0.019799685479325
x =
   0.013210694324681
x =
   0.008811981661324
x =
   0.005876812675109
x =
   0.003918834857523
x =
   0.002612983297712
x =
   0.001742178573622
x =
   0.001161536691921
```

```
x =
     7.743951005436301e-04
x =
     5.162804713201806e-04
x =
     3.441945169104800e-04
x =
     2.294648253299403e-04
x =
     1.529819577579298e-04
x =
     1.019977926009599e-04
x =
     6.802064061364651e-05
x =
     4.527347611488700e-05
x =
     3.010737261619692e-05
x =
     1.981917572558397e-05
x =
     1.303577425675143e-05
x =
     1.042244050528470e-05
x =
     6.334266774253827e-06
x =
     6.334266774253827e-06
```

Since the graph's gradient is decreasing going towards the root (the root is the function's inflexion point), it took more iterations than usual. In total the MATLAB loop ran through 36 iterations to reach the 1e-08 tolerance.

8. Notice the tic and toc calls in the algorithms in #4. From that, the execution times were:
   - Fixed point: 0.0337
   - Newton's method: 0.0140
   - Fzero function: 0.0120

```
f = @(x) sqrt(x) - 1.1;
tic;
root = fzero(f, 1)
time = toc;

time

%fzero is just as accurate as the other functions, returning the correct
%value, 1.21. It is faster than all other functions as well.
```

9.
   - **Use a hybrid bisection/newton algorithm**
   - Start with the lowest root, then iterate up every 0.3 to find other roots (it's completely possible that the algorithm will produce the same number in some iterations, which is fine because the algorithm should remove duplicates). The choice of 0.3 is a small enough increment to catch many roots that are possibly close to each other.

- Execute the newton algorithm indefinitely, but if it is converging slower than

  $$|f(x_{k+1})| < 0.5|f(x_k)|$$

  then switch to bisection for the next iteration
- Repeat the newton algorithm with the same condition as above. This helps speed up convergence, and is especially helpful when we start at one root and iterate blindly to find the next.

```matlab
tol=1;          % initializing tolerance/error
n=0;            % initializing # of iterations
xvals =  double.empty;
y_iter = double.empty;

for x = -1:0.3:1.5 % start from a reasonable point knowing the shape of graph
    x_temp = x;
    newton = true; %newton's method is the default starting methoc
    while tol >= 1e-8      % loop until tolerance is lower than 1e-8 or nmax
        y = x^20 - 3*x^10 + 4*x^9 - 3*x^15;
        if newton
            y_diff = 20*x^19 - 30*x^9 + 36*x^8 - 45*x^14;
            y_iter = x - y/y_diff; % x_k - f(x)/f'(x)
        else
            %% code for bisection (semi-psuedo, just a rough idea)
            b = x_temp + 0.3;
            y_b = b^20 - 3*b^10 + 4*b^9 - 3*b^15;
            p = (x_temp + b)/2;
            if y * y_b < 0
                b = p;
            else
                x_temp = p;
            end
        end

        if abs(y) > 0.5*abs(x_temp) % if iteration doesn't sufficiently improve
            newton = false; % switch to bisection in the next iteration
        end

        tol = abs(y_iter-x);       %calculate absolute error
        x_temp = y;
        n = n+1;     % increase n for every loop
    end
    xvals = [xvals, x_temp];    % append x roots to array of values
end
```

Possible problems:

- The code above ran with a function that was sufficiently spaced out, but it does not guarantee catching all roots when the function has smaller coefficients
  - But there is a tradeoff in efficiency when $\Delta x \to 0$as more iterations is needed, and many found roots are duplicates
- Deciding on the bound for the bisection method is slightly arbitrary and might even mislead the recursion algorithm even more