Table of Contents

.... 1
1. 1
2. 2
3. 3
4. 3


```
%%Lab 3
% David Merrick
% 2012-10-25

lib = make_lib();

% make_lib:
%function lib = make_lib()
%    lib.f = '(x ^ 5) - (x ^ 4) + x - 1';
%    lib.fp = '(5 * (x ^ 4)) - (4 * (x ^ 3)) + 1';
%    lib.the_root = 1;
%    lib.accuracy = 5E-6;
%end
```

1.

```
disp(sprintf('\nBisection Method Table:'));
disp(sprintf('Initial Interval \t Approximation \t\t\t Error \t\t Iterations'));
x0 = 0;
x1 = 3;
[it_count, root, xn] = bisect('x^5-x^4+x-1',x0,x1,lib.accuracy,100);
disp(sprintf('%g to %g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, x1, root, lib.the_r

x0 = 0.5;
x1 = 2;
[it_count, root, xn] = bisect('x^5-x^4+x-1',x0,x1,lib.accuracy,100);
disp(sprintf('%g to %g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, x1, root, lib.the_r

x0 = 0.9;
x1 = 1.2;
[it_count, root, xn] = bisect('x^5-x^4+x-1',x0,x1,lib.accuracy,100);
disp(sprintf('%g to %g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, x1, root, lib.the_r

%a. The second interval needs one fewer iteration because, by starting at
%0.5, it reduces the size of the initial interval and makes the initial midpoint c

%b. Yes, there is an advantage to having the root closer to the midpoint of
%the interval. This is because the bisection method works by splitting the
%interval in two and evaluating the function at that midpoint. As
%demonstrated by running the bisect() function, when the midpoint of the
%initial interval was 1.5, 19 iterations were required. When this midpoint
%was 1.25, 18 iterations were required. When this midpoint was reduced to
```

```
%1.25, only 15 iterations were required.
```

Bisection Method Table:
Initial Interval   Approximation   Error   Iterations
0 to 3   0.9999990463   0.0000009537   19
0.5 to 2   1.0000009537   -0.0000009537   18
0.9 to 1.2   0.9999984741   0.0000015259   15

2.

```
disp(sprintf('\nNewton Method Table:'));
disp(sprintf('Initial Iterate \t Approximation \t\t\t Error \t\t Iterations'));
x0 = -100;
[it_count,root,xn] = newton('x^5-x^4+x-1','5*x^4-4*x^3+1',x0,1.0e-6,100);
disp(sprintf('%g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, root, lib.the_root - root
x0 = 0;
[it_count,root,xn] = newton('x^5-x^4+x-1','5*x^4-4*x^3+1',x0,1.0e-6,100);
disp(sprintf('%g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, root, lib.the_root - root
x0 = 0.9;
[it_count,root,xn] = newton('x^5-x^4+x-1','5*x^4-4*x^3+1',x0,1.0e-6,100);
disp(sprintf('%g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, root, lib.the_root - root
x0 = 0.99;
[it_count,root,xn] = newton('x^5-x^4+x-1','5*x^4-4*x^3+1',x0,1.0e-6,100);
disp(sprintf('%g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, root, lib.the_root - root
x0 = 1.1;
[it_count,root,xn] = newton('x^5-x^4+x-1','5*x^4-4*x^3+1',x0,1.0e-6,100);
disp(sprintf('%g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, root, lib.the_root - root
x0 = 1.4;
[it_count,root,xn] = newton('x^5-x^4+x-1','5*x^4-4*x^3+1',x0,1.0e-6,100);
disp(sprintf('%g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, root, lib.the_root - root
x0 = 1000000;
[it_count,root,xn] = newton('x^5-x^4+x-1','5*x^4-4*x^3+1',x0,1.0e-6,100);
disp(sprintf('%g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, root, lib.the_root - root


% a. The Newton Method is very efficient compared to the Bisection method
% when the initial iterate is close to the root. When it was within .01 of
% the actual root, it only required 3 iterations to find it to the required
% accuracy.

% b. The errors are less than 10^-12 because the Newton method converges
% quadratically, so the number of correct digits roughly at least doubles
% in every step.

% c. For bisection, numIterations >= log((b-a)/epsilon)/log(2).
% This comes to 60 when calculated. This is more efficient than Newton's
% Method, which takes 67 iterations for an initial iterate of 1000000.
```

Newton Method Table:
Initial Iterate   Approximation   Error   Iterations
-100   1.0000000000   0.0000000000   28
0   1.0000000000   0.0000000000   2

```
0.9      1.0000000000    0.0000000000    5
0.99      1.0000000000    -0.0000000000    3
1.1      1.0000000000    -0.0000000000    4
1.4      1.0000000000    -0.0000000000    6
1e+06      1.0000000000    -0.0000000000    67
```

3.

```
disp(sprintf('\nSecant Method Table:'));
disp(sprintf('Interval \t Approximation \t\t\t Error \t\t Iterations'));
x0 = 0;
x1 = 3;
[it_count,root,xn] = secant('x^5-x^4+x-1',x0,x1,1.0e-6,100);
disp(sprintf('%g to %g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, x1, root, lib.the_r
x0 = 0.5;
x1 = 2.0;
[it_count,root,xn] = secant('x^5-x^4+x-1',x0,x1,1.0e-6,100);
disp(sprintf('%g to %g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, x1, root, lib.the_r
x0 = 0.9;
x1 = 1.2;
[it_count,root,xn] = secant('x^5-x^4+x-1',x0,x1,1.0e-6,100);
disp(sprintf('%g to %g \t\t\t %0.10f \t\t %0.10f \t\t %d', x0, x1, root, lib.the_r

% a. The Secant Method is more efficient than the Bisection Method when the
% initial interval is closer to the root, but less efficient than the
% Newton Method.

% b. No. The interval from 0-3 and .9-1.2 both take 6 iterations to find
% the root in the Secant Method where the Bisection Method took 19 and 15
% iterations on the respective intervals.
```

```
Secant Method Table:
Interval   Approximation     Error    Iterations
0 to 3     1.0000000000    -0.0000000000    6
0.5 to 2     1.0000000000    0.0000000000    10
0.9 to 1.2     1.0000000000    -0.0000000000    6
```

4.

```
disp(roots(poly(1:21)));
% The poly() function creates a polynomial whose coefficients correspond to
% the input values. In this case, this polynomial is (x-1)(x-2)...(x-21).
% Roots finds the roots of this polynomial. It finds the first 6 roots
% correctly, but then gradually becomes less and less accurate. This is
% because at the lower-order variables of the polynomial, the function and
% the derivative will be close together, producing low error. At the higher
% order variables, the function and derivative will be further apart,
% producing high error when evaluated using Newton's Method.
```

```
20.9968
20.0300
18.8029
```

```
  18.3718
  16.5702 + 0.5780i
  16.5702 - 0.5780i
  14.4596 + 0.6308i
  14.4596 - 0.6308i
  12.4147 + 0.3075i
  12.4147 - 0.3075i
  10.8665
  10.0537
   8.9869
   8.0028
   6.9996
   6.0000
   5.0000
   4.0000
   3.0000
   2.0000
   1.0000
```

Published with MATLAB® 8.0