Table of Contents

MTH 351 LAB 3 John Waczak

```
clear all;
format long;
```

1 Using the bisection method

```
xT = 1;
tol = 10e-6;
maxIter = 100;

roots = zeros(1,3) ;
iterations = zeros(1,3) ;
trueVal = [1, 1, 1];
interval = {'[0,3]', '[0.5,2]', '[0.9,1.2]'};

[itCount, root, xn] = bisect('x^5-x^4+x-1', 0, 3, tol, maxIter);
roots(1) = root;
iterations(1) = itCount;
[itCount, root, xn] = bisect('x^5-x^4+x-1', 0.5,2.0, tol, maxIter);
roots(2) = root;
iterations(2) = itCount;
[itCount, root, xn] = bisect('x^5-x^4+x-1', 0.9, 1.2, tol, maxIter);
roots(3) = root;
iterations(3) = itCount;

error = abs(trueVal-roots);

T =table(interval.', roots.', error.', iterations.');
T.Properties.VariableNames = {'Interval', 'root', 'error', 'n'};
display(T);
```

```
T =

  3×4 table

     Interval            root                   error             n
    ___________    ________________    ____________________    __

    '[0,3]'         1.00000190734863    1.9073486328125e-06     18
    '[0.5,2]'      0.999998092651367    1.9073486328125e-06     17
    '[0.9,1.2]'     1.00000305175781    3.05175781267764e-06    14
```

(a) The second interval takes exactly one step less because the interval is exactly half as wide as the first i.e. 2-0.5 = 1.5 where as 3-0 = 3

(b) I don't think there will be an advantage either way as the code doesn't presuppose the value of the root. No matter what the interval gets halved each time and so where the root is located shouldn't really matter because all the program is doing is to check when the length of the interval is within your specified tolerance. I will check this by increasing the tolerance and trying the bisection with two different intervals

```
[itCount, root, xn] = bisect('x^5-x^4+x-1', 0.5, 1.5, 1e-10, 300);
itCount
root
[itCount, root, xn] = bisect('x^5-x^4+x-1', 0.9, 1.9, 1e-10, 300);
itCount
root
```

```

itCount =

    33


root =

   1.000000000058208


itCount =

    33


root =

   1.000000000034925
```

as we can see, both intervals took the same number of iterations so there is no clear advantage either way.

2 Using Newton's Method

```
clear all;
format long;
xT = 1;
tol = 10e-6;
maxIter = 100;

roots = zeros(1,7) ;
iterations = zeros(1,7) ;
trueVal = [1, 1, 1,1,1,1,1];
iterates =[-100, 0, 0.9, 0.99, 1.1, 1.4, 1000000];
```

```
[itCount, root, xn] = newton('x^5-x^4+x-1', '5*x^4-4*x^3+1',
 iterates(1), tol, maxIter);
roots(1) = root;
iterations(1) = itCount;
[itCount, root, xn] =newton('x^5-
x^4+x-1', '5*x^4-4*x^3+1',iterates(2), tol, maxIter);
roots(2) = root;
iterations(2) = itCount;
[itCount, root, xn] = newton('x^5-
x^4+x-1','5*x^4-4*x^3+1',iterates(3), tol, maxIter);
roots(3) = root;
iterations(3) = itCount;
[itCount, root, xn] = newton('x^5-
x^4+x-1','5*x^4-4*x^3+1',iterates(4), tol, maxIter);
roots(4) = root;
iterations(4) = itCount;
[itCount, root, xn] = newton('x^5-
x^4+x-1','5*x^4-4*x^3+1',iterates(5), tol, maxIter);
roots(5) = root;
iterations(5) = itCount;
[itCount, root, xn] = newton('x^5-
x^4+x-1','5*x^4-4*x^3+1',iterates(6), tol, maxIter);
roots(6) = root;
iterations(6) = itCount;
[itCount, root, xn] = newton('x^5-
x^4+x-1','5*x^4-4*x^3+1',iterates(7), tol, maxIter);
roots(7) = root;
iterations(7) = itCount;

error = abs(trueVal-roots);

T =table(iterates.', roots.', error.', iterations.');
T.Properties.VariableNames = {'Iterate', 'root', 'error', 'n'};
display(T);
```

```
T =

  7×4 table

    Iterate          root                    error             n 
    _______    ________________    ____________________    __

       -100                   1                       0    28
          0                   1                       0     2
        0.9    1.00000000000508    5.08126873910442e-12     4
       0.99    1.00000000000001    1.33226762955019e-14     3
        1.1    1.00000000000118    1.17905685215192e-12     4
        1.4                   1    2.22044604925031e-16     6
    1000000    1.00000000000013    1.27009514017118e-13    67
```

(a) When the initial iterate is close to the true root, Newton is MUCH more efficient. For example when the inital iterate was a value of 1.1 it took 4 iterations versus the 17 it took bisection given an interval of

[0.5, 2]. For these examples, Newtons method was 4 times faster than Bisection however we must already know the derivative before hand.

(b) This is likely because the iteration that takes us below the tolerance is extremely precise and cuts down multiple orders of magnitude. From class we know for values close to the root the error goes like the square of the error for the last iteration. So if the previous error was near 10^-4 for example, then one more step would put us well below our tolerance.

(c) We know that the error for bisection goes like 1/2 the previous error so we need to figure out how many times we need to halve 2000000 in order to get to the epsilon of 1.3*10^-12. If we solve the equation 2^n = 2000000/(epsilon) we get roughly 60 iterations. This is actually fewer than was required for using Newton's method which took 67 iterations.

Using the secant method

```
clear all;
xT = 1;
tol = 10e-6;
maxIter = 100;

roots = zeros(1,3) ;
iterations = zeros(1,3) ;
trueVal = [1, 1, 1];
iterates = {'[0,3]', '[0.5,2]', '[0.9,1.2]'};

[itCount, root, xn] = secant('x^5-x^4+x-1', 0, 3, tol, maxIter);
roots(1) = root;
iterations(1) = itCount;
[itCount, root, xn] = secant('x^5-x^4+x-1', 0.5,2.0, tol, maxIter);
roots(2) = root;
iterations(2) = itCount;
[itCount, root, xn] = secant('x^5-x^4+x-1', 0.9, 1.2, tol, maxIter);
roots(3) = root;
iterations(3) = itCount;

error = abs(trueVal-roots);

T =table(iterates.', roots.', error.', iterations.');
T.Properties.VariableNames = {'Iterates', 'root', 'error', 'n'};
display(T);
```

```
T =

  3×4 table

    Iterates             root                  error              n
    ___________    __________________    ____________________    __

    '[0,3]'          1.00000000000229    2.29172236743125e-12     6
    '[0.5,2]'       0.999999999997459    2.54085641415713e-12    10
    '[0.9,1.2]'      1.00000000000237    2.37077024678456e-12     6
```

(a) The method is comparable to Newton's method (a tad slower) and still significantly faster than the bisection method.

(b) The inital iterate distance doesn't seem to impact the convergence as much as the bisection method. In fact for the shorter interval of [0.5, 2] the secant method was worse than for the interval [0,3].

4 Explain the output of roots(poly(1:21)).

```
clear all;
roots(poly(1:21))
```

```
ans =

  21.000146192273885
  19.998017068635978
  19.011750010545637
  17.954655450609799
  17.097857021160284
  15.801113088928858
  15.229242857621225
  13.803816272988852
  13.145982959109434
  11.938851148084597
  11.023009245586470
   9.994999290540639
   9.000514077105224
   8.000077854753531
   6.999961425136084
   6.000006586248248
   4.999999430414557
   4.000000020021613
   3.000000000258564
   1.999999999975864
   1.000000000000150
```

the poly(1:21) generates a polynomial with zeros at positions 1,2,3..21. If we are to take 2.3 problem 11 as an example this polynomial could be easily constructed by simply writing out the factored form. Then the roots() function takes that polynomial and finds all of its roots again which we can verify are at 1,2,3,...21 with some numerical error.

Published with MATLAB® R2017b