CS 8, Winter 2015
Homework Assignment #? (draft)

Assignment Overview

This assignment focuses on implement two algorithms for finding roots numerically.

Assignment Specifications

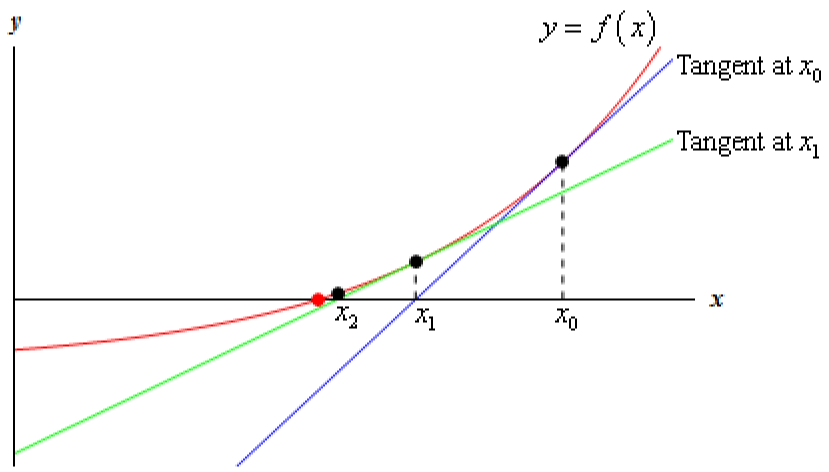
We all learned how to find root (*f(x)=0*) of 1st order (*f(x) = ax+b*) and 2nd order (*fx)= ax²+bx+c*) polynomials. Things get much harder for higher-order polynomials as no simple close-form solutions exist for polynomials of degrees 4 or higher. Often, roots are located using a numerical method.

In this assignment, you will implement two numerical methods for root finding and compare their accuracy and efficiency: a simpler bisection method and a more sophisticated Newton-Raphson method. The goal is to find the roots of a polynomial in one variable. We will assume that the polynomial is of the 3rd order, or $f(x) = x^3 + ax^2 + bx + c$. Note that the coefficient of the highest-order term (x^3) is normalized to one, but try to convince yourself that it does not invalid your solution if that coefficient is not 1, (as $sf(x)=0$ has the same roots as $f(x)=0$ as long as s is not zero).

For this normalized representation, we know that f(-infinity)->-infinity and f(infinity)->infinity. Because a polynomial function is continuous, the function must cross over from negative to positive, when *x* goes from -infinity to infinity. So a root *x* is "boxed" in between x_{upper} and x_{lower} if $f(x_{upper})$ and $f(x_{lower})$ are of opposite signs. In fact, multiple roots can be boxed in between x_{upper} and x_{lower}, but we will be satisfied with any one of them.

You are given the three coefficients of *a, b,* and *c,* and a desired accuracy. Your task initially is to find two x values, x_{upper} and x_{lower}, so that $f(x_{upper})>0$ and $f(x_{lower})<0$. Then we know that there must be at least one root in the interval $[x_{lower}, x_{upper}]$. The way we zero in on that root in the bisection method is this: We perform a bisection of the interval to compute $x_{mid} = (x_{upper} + x_{lower})/2$. There are only three possible outcomes: (1) $f(x_{mid})=0$, we found a root directly, (2) if $f(x_{mid})$ is positive, we know there must be a root in between $[x_{lower}, x_{mid}]$, and (3) if (x_{mid}) is negative, we know there must be a root in between $[x_{mid}, x_{upper}]$. So either we can stop in case (1), or we half the size of the search interval in cases (2) and (3). We repeat this process until we narrow down the search interval to be less than a given accuracy requirement, that is, the estimated function value differs from 0 by less than the specified margin.

Next you will implement the Newton-Raphson root finding algorithm for a 3rd-order polynomial. Newton-Raphson method starts from an initial guess (x_o in the following figure) and follows the tangent line to refine the root to x_1. The process is then repeated (i.e., iteratively follows tangent to refine the previous estimate) until again the function value is less than a threshold from 0.



The tangent line direction (slope) for $f(x) = x^3 + ax^2 + bx + c$ is

$$f'(x) = 3x^2 + 2ax + b$$

Now compare bisection and Newton-Raphson root finding programs on the same set of input data, what do you observe? Which one runs faster (requiring less number of iterations to get the same precision)?

Assignment Deliverables

The deliverable for this assignment is the following file:

root.py – the source code for your Python program

Be sure to use the specified file name and submit it for grading via the **turnin** system before the project deadline.

Assignment Notes

1. Your program should comprise two callable functions (and other internal functions):
   - bs(coef, f, df, eps) (bisection)
   - nr(coef, f, df, eps, initGuess = 0.0) (Newton-Raphson)
     a. coef is a list of three elements [a, b, c] that specifies the coefficient of a normalized 3rd order polynomial,
     b. *f* is a function f(coef, x) that evaluates the function value given the coefficient and *x* location
     c. *df* is a function df(coef, x) that evaluates the derivative (tangent) given the coefficient and *x* location (note: coef in *f* and *df* are that given in the BS and NR calls)
     d. *eps*: is the required precision of the root finding process
     e. *initGuess* is an optional parameter that gives the initial guess to the NR method.

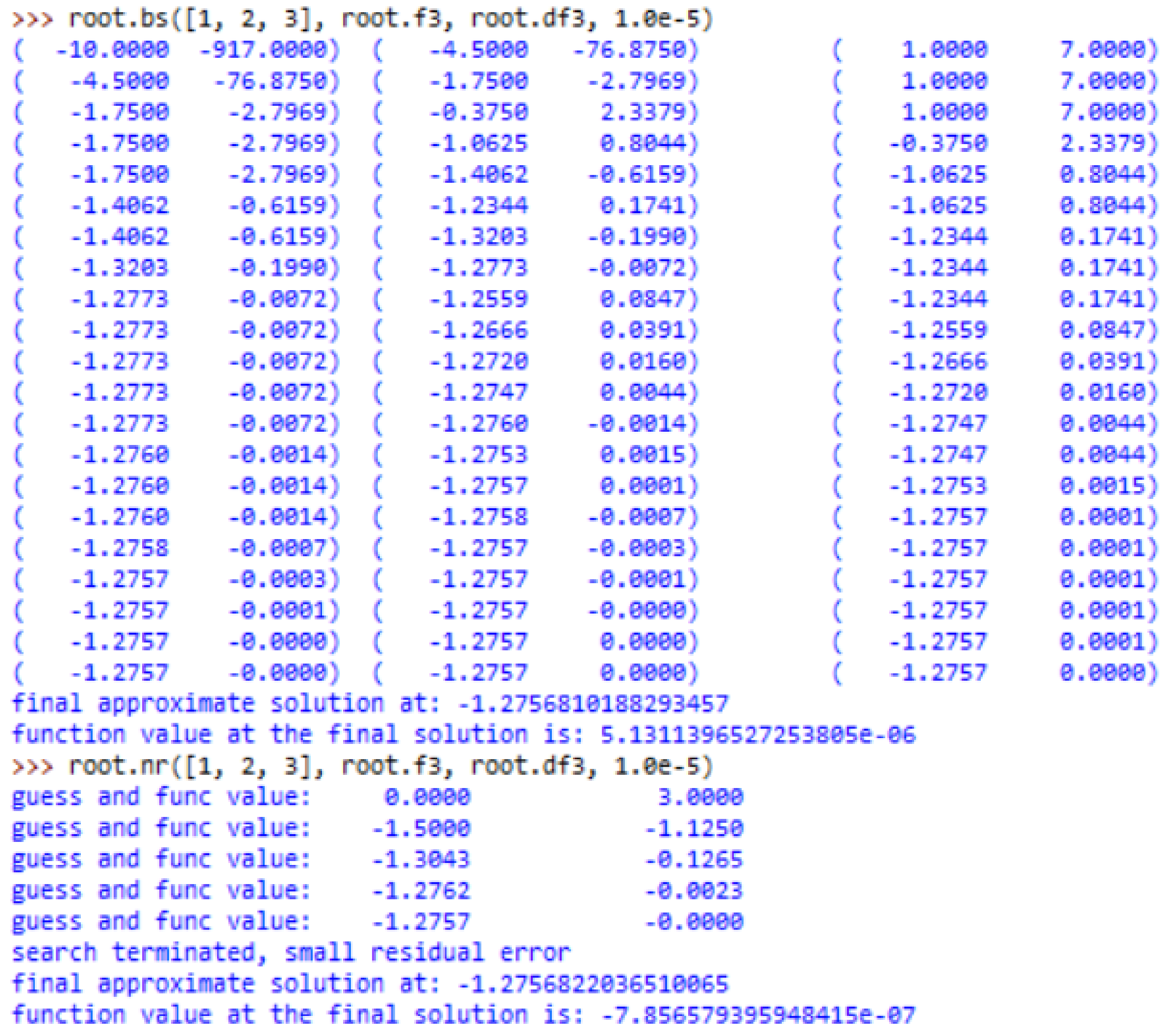
2. To establish an initial search range for bisection, you can start with x_{upper} =1 and x_{lower} =-1. If f(x_{upper}) is positive and f(x_{lower}) is negative, you have established an initial range. If not, move x_{upper} to the right and/or x_{lower} to the left until you have a positive negative pair.

3. For Newton-Raphson, you should allow the user to specify an initial guess, but if such an initial guess was not given, you should use a default initial guess of 0.

4. Finally, as BS and NR methods are iterative, there is a chance that they might iterate forever and never stop (never get close to zero). To protect against such a case, hard code a maximum iteration count in your program (a number between 50 and 100 is fine).

Sample Output:

For bisection, you should print out three tuples per line (x_{lower}, f(x_{lower})), (x_{mid}, f(x_{mid})), (x_{upper}, f(x_{upper})) for each approximation step. For Newton-Raphson, you should print out the x location and function value for each step. For both methods, print out the final root location and function value at the end.



```
>>> root.bs([1, 2, 3], root.f3, root.df3, 1.0e-5)
(  -10.0000  -917.0000)  (   -4.5000   -76.8750)        (    1.0000     7.0000)
(   -4.5000   -76.8750)  (   -1.7500    -2.7969)        (    1.0000     7.0000)
(   -1.7500    -2.7969)  (   -0.3750     2.3379)        (    1.0000     7.0000)
(   -1.7500    -2.7969)  (   -1.0625     0.8044)        (   -0.3750     2.3379)
(   -1.7500    -2.7969)  (   -1.4062    -0.6159)        (   -1.0625     0.8044)
(   -1.4062    -0.6159)  (   -1.2344     0.1741)        (   -1.0625     0.8044)
(   -1.4062    -0.6159)  (   -1.3203    -0.1990)        (   -1.2344     0.1741)
(   -1.3203    -0.1990)  (   -1.2773    -0.0072)        (   -1.2344     0.1741)
(   -1.2773    -0.0072)  (   -1.2559     0.0847)        (   -1.2344     0.1741)
(   -1.2773    -0.0072)  (   -1.2666     0.0391)        (   -1.2559     0.0847)
(   -1.2773    -0.0072)  (   -1.2720     0.0160)        (   -1.2666     0.0391)
(   -1.2773    -0.0072)  (   -1.2747     0.0044)        (   -1.2720     0.0160)
(   -1.2773    -0.0072)  (   -1.2760    -0.0014)        (   -1.2747     0.0044)
(   -1.2760    -0.0014)  (   -1.2753     0.0015)        (   -1.2747     0.0044)
(   -1.2760    -0.0014)  (   -1.2757     0.0001)        (   -1.2753     0.0015)
(   -1.2760    -0.0014)  (   -1.2758    -0.0007)        (   -1.2757     0.0001)
(   -1.2758    -0.0007)  (   -1.2757    -0.0003)        (   -1.2757     0.0001)
(   -1.2757    -0.0003)  (   -1.2757    -0.0001)        (   -1.2757     0.0001)
(   -1.2757    -0.0001)  (   -1.2757    -0.0000)        (   -1.2757     0.0001)
(   -1.2757    -0.0000)  (   -1.2757     0.0000)        (   -1.2757     0.0001)
(   -1.2757    -0.0000)  (   -1.2757     0.0000)        (   -1.2757     0.0000)
final approximate solution at: -1.2756810188293457
function value at the final solution is: 5.1311396527253805e-06
>>> root.nr([1, 2, 3], root.f3, root.df3, 1.0e-5)
guess and func value:     0.0000              3.0000
guess and func value:    -1.5000             -1.1250
guess and func value:    -1.3043             -0.1265
guess and func value:    -1.2762             -0.0023
guess and func value:    -1.2757             -0.0000
search terminated, small residual error
final approximate solution at: -1.275682203651065
function value at the final solution is: -7.856579395948415e-07
```