FFR105: Introductory programming exercise

Applied artificial intelligence group

September 2, 2020

The aim of this exercise is for you to learn how to write well-structured program code. It is very important to be able to do so, partly for your own sake (well-written program code is easier to debug, modify, and extend) and partly for the sake of others (well-written code is usually easy to read). This problem is a complement to the Matlab introduction (during which you implement a genetic algorithm), the main difference being that, here, the focus is on the actual *programming*, rather than the task solved by the program. When writing, keep in mind to write as simple and clear code as possible, using appropriate variable names, etc. You *should* follow the coding standard, available on the course web page.

The problem that you will solve is as follows: *Write a Matlab program, following the coding standard, that takes as input (the coefficients of) an n^{th}-degree polynomial and a starting point, and which then uses Newton-Raphson's method to find the nearest stationary point of the polynomial.*

The program should consist of precisely the following (not more, not less):

- A function `NewtonRaphson` (in the file `NewtonRaphson.m`) that takes as input (i) a vector containing the coefficients of the polynomial considered, (ii) a starting point, and (iii) a tolerance parameter (for checking convergence), in that order, and then executes the main iteration loop while storing the iterates x_j, $j = 0, 1, \ldots$. After converging (i.e. when $|x_{j+1} - x_j| < T$, where T is the tolerance parameter) the function should return a vector containing the iterates with the stationary point as the last element in that vector.

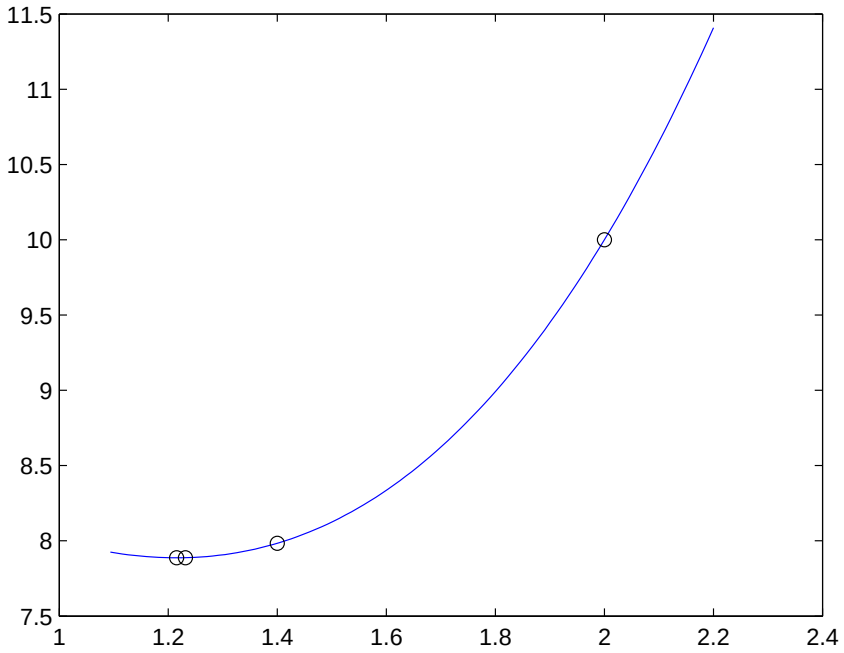
- A function `Polynomial` (in the file `Polynomial.m`) which takes x and a vector with the $n + 1$ coefficients of an n^{th}-degree polynomial $a_0, a_1, \ldots, a_n$ as input (in that order), and returns the value $f(x) = a_0 + a_1 x + \ldots + a_n x^n$.

- A function `PolynomialDifferentiation` (in the file `PolynomialDifferentiation.m`) which takes, in order, a vector with the $n+1$ coefficients of an n^{th}-degree polynomial $a_0, a_1, \ldots, a_n$ and the order k of the derivative as input and returns the $n + 1 - k$ coefficients of the k^{th} derivative of the polynomial.
  Example: if the input polynomial is $1+2x+3x^2$ and $k = 1$, the input coefficients are $(1, 2, 3)$, and the output coefficients should be $(2, 6)$ If, instead, $k = 2$, the output coefficients should instead be (6) (a vector with one element). If $k \geq 3$, an empty vector should be returned.

  Note: In this problem, you may *not* make use of any built-in polynomial or differentiation functions in Matlab!

- A function `NewtonRaphsonStep` (in the file `NewtonRaphsonStep.m`) which carries out a Newton-Raphson iteration step as in Algorithm 2.3 in the course book (p. 22), taking three inputs, x_j, $f'(x_j)$, and $f''(x_j)$, (in that order), and returning x_{j+1}.
- A function `PlotIterations` (in the file `PlotIterations.m`), which should be called after the end of the main loop, and which plots the polynomial (in an appropriate range, both in the horizontal and vertical directions) as well as the iterates (as circles) x_j, $j = 0, 1, \ldots$. A plot example is shown below where the function considered in Example 2.4 in the course text book is plotted along with the values obtained by each step of Newton-Raphson, starting at point $x_0 = 2$.
- A main script `Main.m` that defines the polynomial coefficients and starting point, calls the `NewtonRaphson` function with these parameters and finally calls `PlotIterations` as in the following example:

```
polynomialCoefficients = [10 -2 -1 1]; % Defines the polynomial 10 - 2x - x^2 + x^3
startingPoint = 2;
tolerance = 0.0001;
iterationValues = NewtonRaphson(polynomialCoefficients, startingPoint, tolerance);
PlotIterations(polynomialCoefficients,iterationValues);
```



Note that `PolynomialDifferentiation` should be able to handle *any* non-negative integer values of n and k (including the cases $n = 0$, $k = 0$, $k > n$ etc.), and that the overall program should be able to handle any non-negative value of n. If the iterates cannot be computed (for example, if $n < 2$ or $f''(x_j) = 0$ for some other reason), the program should give a suitable error message (in the form of a text string printed to the screen, clearly describing the error, for example "*the degree of the polynomial must be 2 or larger*") and then terminate. Note also that all computations of polynomials (including, for example, $f'(x)$) should be carried out using your function `Polynomial`, with appropriate inputs (obtained, in the case of $f'(x)$, by a call to `PolynomialDifferentiation`).