FFR105: Introductory programming problem

Applied artificial intelligence group

August 24, 2023

Problem description

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.

In this assignment, starting from a set of skeleton files, you will write a well-structured Matlab program for running the Newton-Raphson method in order to find stationary points of an arbitrary polynomial. You must make use of the six skeleton files provided on Canvas for this assignment. The file `Wrapper.m` is the main file for running the Newton-Raphson method. This file, which hard-codes the coefficients of a polynomial as well as a starting point for the iteration, should be *handed in* as it is, but during your work you should edit it to try many different polynomials and starting points, to make sure that your program works in all cases. For the other five files, you should write the necessary program code as described below. The skeleton files define the interfaces of the functions (i.e. the inputs and outputs) which may *not* be modified.

Functions to implement

`RunNewtonRaphson.m` This function should take 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 execute 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 (within the given tolerance) as the last element in that vector.

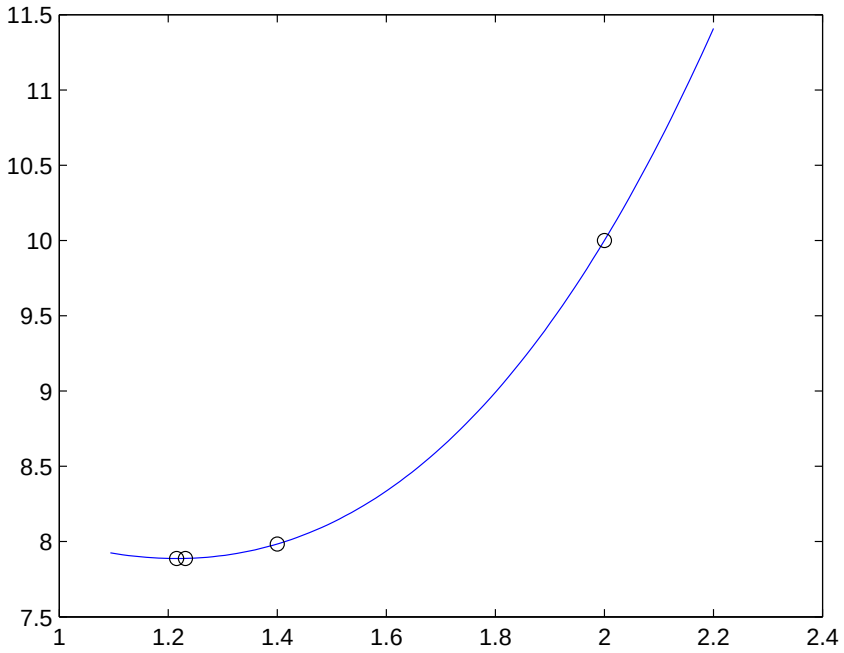
`GetPolynomialValue.m` This function should take (i) x and (ii) a vector with the $n+1$ coefficients of an n^{th}-degree polynomial $a_0, a_1, \ldots, a_n$ as input (in that order), and should return the (scalar) value $f(x) = a_0 + a_1 x + \ldots + a_n x^n$.

`DifferentiatePolynomial.m` This function should take as input (i) a vector with the $n+1$ coefficients of an n^{th}-degree polynomial $a_0, a_1, \ldots, a_n$ and (ii) the order k of the derivative and should return 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 (for this example).

Note: In this problem, you may *not* make use of any built-in polynomial or differentiation functions in Matlab! You must write the code yourself, implementing the required steps for computing a polynomial or for taking its derivative.

`StepNewtonRaphson.m` This function should carry out a Newton-Raphson iteration step as in Algorithm 2.3 in the course book (p. 22), taking three (scalar) inputs, x_j, (the value of) $f'(x_j)$, and (the value of) $f''(x_j)$, (in that order), and return x_{j+1}. If the result cannot be computed (e.g. if $f''(x_j) = 0$) the method should (a) print a suitable, clear error message and (b) return NaN (not-a-number), so that the main program can terminate, i.e. such that `RunNewtonRaphson.m` stops execution in the (rare) cases where it receives NaN from `StepNewtonRaphson.m`.

`PlotIterations.m` This function, which should be called after the end of the main loop, should neatly plot 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 book is plotted along with the values obtained by each step of Newton-Raphson, starting at the point $x_0 = 2$.



Important notes

- `DifferentiatePolynomial` should be able to handle *any* non-negative integer values of n and k (including the cases $n = 0$, $k = 0$, $k > n$ etc.). Moreover, the main program should be able to handle any non-negative value of n.
- All computations of polynomials (including, for example, $f'(x)$) should be carried out using your function `GetPolynomialValue`, with appropriate inputs (first obtained, in the case of $f'(x)$, by a call to `DifferentiatePolynomial`).
- 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 as described above. In such cases, no plot is needed (or the program may give an empty plot).

What to hand in

You should only hand in the six files, not more not less, collected in a single .zip or .7z file by uploading this single compressed file in Canvas. Do *not* include any report. Make sure also to remove any other files, e.g temporary files, test versions etc. Before correction, the compressed file (that, again, must be either a .zip file or a .7z file) will be scanned automatically, to check that the number of files (inside the compressed file) is correct, that the names of the files are unchanged, and that the interfaces are unchanged. Submitted solutions that do not fulfil those criteria will be returned (unseen) for revision.