Final Project: Polynomial Equation Root Solver

Mariano Arce Jop Programming II: C++

**Objectives**

1. Find and understand the best algorithm for finding the roots of a polynomial.
2. Design and code a parser for a C program to interpret a written function.
3. Develop a C program to find the roots of a polynomial in a given range, or the nearest root if no roots are found within the range.

**Inputs**

1. A function of x as a string in the form CoefficientxPower+CoefficientxPower+Numberc;

Where all coefficients and the constant must have two decimal digits.

The function can have as many terms as can fit inside the string of length S, a macro defined in the program currently set to 100.

The maximum value for any coefficient or the constant is 9999.99.

The highest power of x allowed is 10.

The input is entered through a parser that uses getchar() saving every character into the string.

1. The lower and upper bounds where the search for the roots will take place

These bounds are of type double, input through a scanf.

**Outputs**

The outputs of the system are the roots found, in the form “Root at x = root. “

The program will print as many roots inside the limit as it finds, if it finds no roots it will print the nearest root in the form:

“No real roots were found inside the interval, the nearest root is at x = root”.

If the function has no real roots, the program will print:

“The function has no real roots”.

**Functions**

instructions:

This function is meant to give the user the instructions for the program, and the capabilities it has. It should print the user the necessary knowledge to input the function correctly, giving him an example of one. It also specifies the program’s capacity, and the limits of computation.

The function is basically a set of printfs stating the instructions for the use of the program.

The function returns no value, nor does it have any inputs as it is just a function to print.

getString:

This function obtains the function of x from the user as a string. It acts like a scanf but for the entire string, parsing from character to character and adding the null termination at the end.

The function is limited by the Macro ‘S’, which defines the length of the string.

The function returns void, it has no ordinary outputs. However, it fills in the string “dst” (short for destination) with user input, obtained with getchar().

validateString:

This function ensures that the string input by the user is validate. It traverses the string checking that every coefficient has a decimal point, that every decimal point has two decimal digits, that every coefficient is a maximum of 9999.99, and that there is at least one x.

The function’s parameters are essentially the factors being checked for: The maximum value for the coefficients (9999.99), the two decimal digits, the decimal point, and the variable x.

The function outputs an integer, it is 1 if the function is valid, and -1 if it is not. It also prints the reason why the function input was invalid in the case that it was.

initializeArray:

This function initializes the array of coefficients by making sure there are no unwanted values inside it. It basically traverses through the array setting every double to the value 0.

The function’s only parameter is the size of the coefficient array, 11.

The function returns void, but it edits the coefficient array leaving it blank (filled with 0s).

buildArray:

This function converts the function of x input by the user into an array containing the coefficients of every x, ordered from the highest power of (10) to the lowest (0, being a constant number). This is done by searching through the string for the character ‘x’ or a ‘c’, when the ‘x’ is found the exponent is used to determine the position to be saved in the array, and the location of the last digit of its coefficient is input into the function “findCoefficient”; which outputs back the coefficient as a double. The function then saves this value into the designated location of the coefficient array, starting from x^10 until x^0, being the constant number followed by a ‘c’, whose coefficient is found using the same function.

The function uses ‘x’ and ‘c’ as parameters to find the coefficients. It also uses the exponent of every x term to find the location in the array in which the coefficient must be saved.

The function returns void, it has no real outputs, but it fills the coefficient array with the coefficients of x in descending order of powers of x.

findCoefficient:

This function finds the number to the left of the last decimal digit it is given. This number turns out to be the coefficient of some x, which is found by computing the sum of its components, each digit. To do this, the function traverses backwards starting at the hundredths place, casting the character from the string into an integer and multiplying it by its magnitude, in this case its 0.01. It skips the decimal point and keeps adding the digits into the count, until a‘+’ or ‘-‘ sign is found or the end of the string is reached, having saved digits until the thousands place.

The function uses the positions of where it finds the digits as parameters to find their weight into the sum. It also uses the first position it is given, which is the last digit of the coefficient to know where to start traversing left to find the number.

The function outputs the coefficient of x as a double with two decimal places, being a maximum of 9999.99.

findDerivative:

This function is given an array of coefficients representing a function of f, and it inputs the coefficients of the derivative of f into another array.

The function uses the common derivative technology available, multiplying the coefficient by the exponent and then subtracting one from the exponent.

The function has no real outputs, but it fills in the array of doubles “derivative” with the coefficients of the derivative of the function contained in the original coefficient array.

evaluateFunction:

This function computes the value of f given a parameter for x and the coefficient array. It simply adds up the value of every x term elevated to the power of its position in the array, multiplied by its coefficient.

The function’s only parameters are the coefficient array and the value for x being tested.

The function’s output is a double that holds the value of f(x).

add\_node\_at\_end:

This function adds a node at the end of the linked list and fills it with the lower and upper limits of type double it is given. If the list is empty, it creates the first node.

The function’s parameters are the pointer to the first node, which is equal to NULL if there is no node. And, the values for the lower and upper bound of the interval being saved.

The function has no real output, but it adds a new node and saves the inputted limits into this new node.

getRoots:

This function uses the function bisectionMethod to find and print the roots inside every range saved into the linked list. If there are no saved ranges in the list, it means that the limits inputted by the user contained no roots. If this is the case, the function uses newtonsMethod to find and print the nearest root. If newtonsMethod can’t find it, then it prints that the function has no real roots.

The parameters that this function uses are the midpoint of the original limits input by the user, the array of coefficients expressing the function, the array of coefficients expressing the derivative of the function, the limits found inside the linked list and the pointer to the head of the list.

This function has no real outputs, but it prints the roots found, or alerts if there are no roots.

findLimits:

This function is the essence of the program. It is given the initial boundaries for the range and it recursively calls itself making the boundary as small as delimited by Macro L, set to 0.05. The function calls checkLimits to see if the boundary might contain a root, if it does and no smaller boundary has been added to the linked list, then it adds the boundary to the linked list.

The parameters of this function are the array of coefficients, the lower and upper limits of the range and the pointer to the linked list to pass on to the add\_node\_at\_end function.

The function’s only real output is an integer, which is used to tell itself after a recursive call if it saved a limit or not. The value will be 0 if no limits were saved, and 1 or 2 if it saved 1 or 2 limits. The function also uses add\_node\_at\_end to save limits into the linked list.

checkLimits:

This function checks if the limits it is given could contain a root of x. This is done using the Fundamental Theory of Algebra, if the limits have a different sign then they contain a root of f(x).

The function’s only parameters are the upper and lower limits, and the array of coefficients to pass onto the evaluateFunction function to find f(x) at the limits.

The function outputs 1 if the range is known to have a root, or -1 otherwise.

newtonsMethod:

This function is an implementation of Newtons Method to find the nearest root of a function given a point x. It finds the nearest root through linear approximations using the derivative of the function, with Newton’s formula (Xn=Xo-(f(x)/f’(x))) and recursively calls itself until it finds the root with an error of less than macro C, set to 0.0001.

The function’s parameters are the array of coefficients of the function, array of coefficients of the derivative of the function, and the value x, where it will start to look for the root.

The function’s only output is the value of x where f(x) is equal to 0 i.e. there is the root.

bisectionMethod:

This function is an implementation of the bisection method to find the root of a function of f in a given range. The function computes the midpoint between the two limits, and checks the value at the two limits and the midpoint. If any of those values is 0, then it returns the x of that value. If it isn’t, then it calls itself recursively with the limits where the root is found. These are the limits with opposite sign.

The function’s parameters are the upper and lower bounds, and the array of coefficients used to express the function inputted by the user.

The function outputs the value of x where it found f(x) = 0.

roundd:

This function rounds a number x to two decimal points using the function round(x) and simple multiplication and division by 100.

The only parameter for the function is x.

The output of this function is the rounded x.

**Test Plan**

1. Test that it validates the string correctly and inputs it into the array of coefficients, alerting the user if the string has a mistake and asking for a new input.
2. Test that it finds several roots inside a range where there exist several roots.
3. Test that it recognizes when there are no roots inside the range and successfully

computes the nearest root to the interval inputted.

1. Test that it reconizes when there are no roots at all.

**Test Results**

1. The program validates correctly and inserts the function into the coefficient array successfully.

Text

Description automatically generated

1. The program successfully found the roots. I tried this several times but will only show one test for conciseness.

Text

Description automatically generated

1. The program recognizes there are no roots inside the range and finds the nearest root.

A picture containing text, indoor

Description automatically generated

1. The program recognizes that the function has no real roots.

Text

Description automatically generated

**Observations**

1. I personally believe the parser for the function works really well, and I had a great time making it. I also think the program in general executes properly and the findLimits function is very complete and well thought of. I believe the overall planning of the program turned out to work efficiently.
2. I had a huge problem with my program shortly before the due date. My function roundd was called round, and there exists a function inside math.h that is called round. This caused an error called Segmentation fault: 11, and it took me a very long time figuring that out. I’m very glad I did.
3. I think to improve my project in the future I’d like to implement the Durand-Kerner algorithm to find complex roots as well. I intended to make this a part of my project but the complex number arithmetic got very complicated.

**I included the structure diagrams on a different document in ppt format because it made the report very long a harder to read through.**