CMPS 12A
Introduction to Programming
Programming Assignment 4

In this assignment you will write a java program that determines the real roots of a polynomial that lie within a specified range. Recall that the roots (or zeros) of an n^{th} degree polynomial

$$f(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{n-1} x^{n-1} + c_n x^n$$

are those numbers x for which $f(x) = 0$. In general an n^{th} degree polynomial has at most n roots, some of which may be real and some complex. Your program, which will be called Roots.java, will find only the real roots.

Program Operation
Your program will prompt the user to enter the degree of the polynomial, the coefficients, and the left and right endpoints of the interval in which to search. For example, to find the roots of the cubic polynomial $x^3 - 6x^2 + 11x - 6$ in the interval [-10, 10], your program would run as follows.

```
% java Roots
Enter the degree: 3
Enter 4 coefficients: -6 11 -6 1
Enter the left and right endpoints: -10 10

Root found at 1.00000
Root found at 2.00000
Root found at 3.00000
%
```

Notice that the user enters the coefficients from lowest power of x to highest power. The degree can be any non-negative integer, while the coefficients and range limits are doubles. No error checking on these inputs are required for this project. You may therefore assume that your program will be tested only on inputs that conform to these requirements. The root values will be rounded to 5 decimal places of accuracy.

Consider the fourth degree polynomial $x^4 - x^2 - 2 = (x^2 - 2)(x^2 + 1)$. Written in factored form it is easy to see that the roots are $\sqrt{2}, -\sqrt{2}, i,$ and $-i$. (See http://en.wikipedia.org/wiki/Imaginary_unit if you are unfamiliar with the number i.) As we can see the program does not the find complex roots.

```
% java Roots
Enter the degree: 4
Enter 5 coefficients: -2 0 -1 0 1
Enter the left and right endpoints: -5 5

Root found at -1.41421
Root found at 1.41421
%
```

Your program will find only those roots that are contained in the interval specified by the user. If no roots are found in that interval, a message to that effect is printed to stdout. We illustrate this with two more searches on the same polynomial: $x^4 - x^2 - 2 = (x^2 - 2)(x^2 + 1)$.

```
% java Roots
Enter the degree: 4
Enter 5 coefficients: -2 0 -1 0 1
Enter the left and right endpoints: 0 2

Root found at 1.41421
% java Roots
Enter the degree: 4
Enter 5 coefficients: -2 0 -1 0 1
Enter the left and right endpoints: 0 1

No roots were found in the specified range.
%
```

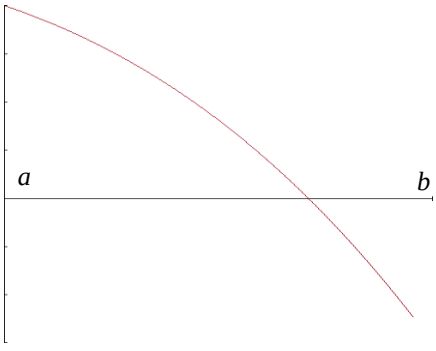
Notice that after typing the degree, your program will ask the user for the correct number of coefficients. An n^{th} degree polynomial has $n+1$ coefficients, including the zero terms, all of which must be entered by the user. When testing your program you may find it useful to store the inputs in a file and redirect stdin to read from that file rather than the keyboard. In that case the program operation will appear as follows.

```
% more infile
5
30 0 -10 -3 0 1
-10 10
% java Roots < infile
Enter the degree: Enter 6 coefficients: Enter the left and right endpoints:
Root found at -1.73205
Root found at 1.73205
Root found at 2.15443
%
```

The prompts all appear on a single line because there is no one at the keyboard hitting return to enter the inputs. All of these examples have served only to demonstrate program operation for this project. So far we've said nothing about how your program will accomplish these calculations. To start, read section 4.11 in the text, especially the example FindRoot.java at the end of that section which uses the Bisection Method to compute the square root of 2. This example is also posted on the class website.

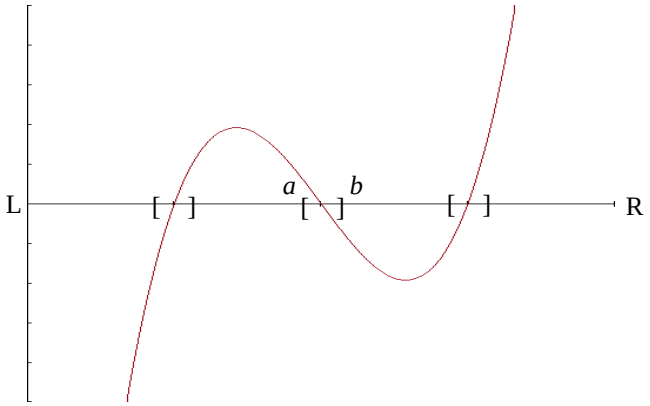
The Bisection Method

If $f(x)$ is a continuous function on an interval $[a, b]$, and if $f(a)$ and $f(b)$ have different signs, then $f(x)$ must have a zero in $[a, b]$. This fact is known as the *Intermediate Value Theorem*, and is clear from the figure below.

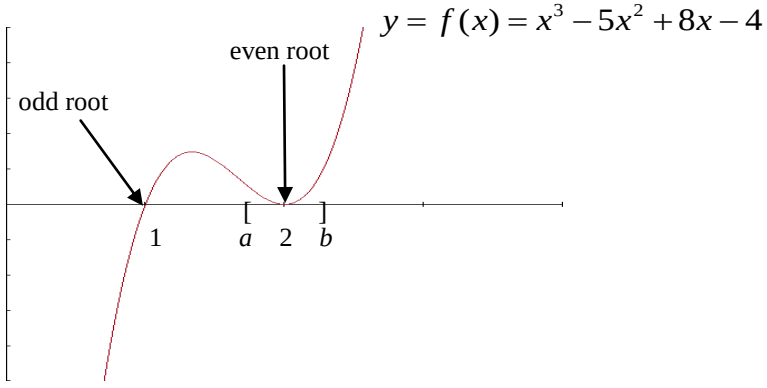


The bisection method is an iterative technique that traps the zero in smaller and smaller subintervals, until the size of the subinterval containing the zero is sufficiently small. The location of the root is then known with an error no more than the width of that final subinterval. To begin let $m = (a+b)/2$ be the midpoint of $[a, b]$. This midpoint serves as the current approximation to the root. Look at the two subintervals $[a, m]$ and $[m, b]$. If $f(a)$ and $f(m)$ have different signs, then the root is contained in $[a, m]$. If $f(m)$ and $f(b)$ have different signs, then the root is contained in $[m, b]$. If the original interval $[a, b]$ contained one root, then exactly one of these two alternatives must hold. Let us assume that it is the first subinterval $[a, m]$ that contains the root. The other subinterval is discarded and the search continues on the next iteration in $[a, m]$. Its midpoint will serve as the next approximation to the root. Notice that the "search space" has been halved on one iteration. This means that each iteration increases the accuracy of our root estimate by one binary digit, and about 4 iterations gains one decimal digit of accuracy. The algorithm continues to loop until the interval in which the root is known to exist has width less than some predefined *tolerance*.

If the initial interval contains more than one root of $f(x)$ then this method will only find one of them. If we wish to find all roots within some range *L* to *R* we must partition $[L, R]$ into a sequence of sufficiently small subintervals $[a, b]$ of equal width, then run the Bisection Method on each such subinterval for which $f(a)$ and $f(b)$ have different signs. The width $b-a$ of these smaller subintervals therefore constitutes a limit on the possible *resolution* with which roots can be detected. In other words, roots in $[L, R]$ that are closer together than the resolution $b-a$ cannot be distinguished. When run on the example pictured below, this procedure will find all three roots.

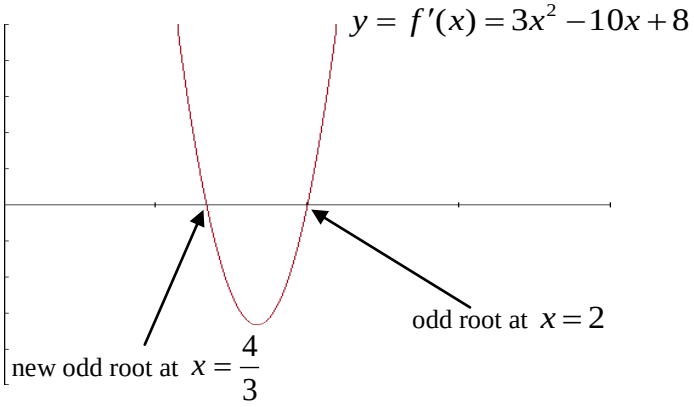


The procedure will break down however when $f(x)$ has two roots that are "infinitely close". For instance the polynomial $f(x) = (x-1)(x-2)^2 = x^3 - 5x^2 + 8x - 4$ has a so-called *double root* at the point $x = 2$ and a *single root* at $x = 1$.



No matter how fine a resolution we pick, we cannot distinguish between the "two" roots at $x=2$. In fact the bisection method cannot find this root at all since it is not the case that $f(a)$ and $f(b)$ have different signs, no matter how small $b-a$ is.

We can divide the roots of a polynomial $f(x)$ into two classes called the *odd* roots and the *even* roots, respectively. The point x_0 is an odd root if and only if the graph of $f(x)$ *crosses* the *x*-axis at x_0. We call x_0 an even root if and only if the graph *touches* the *x*-axis at x_0. Since it requires a sign change, the bisection method can only find the odd roots of a polynomial. However the even roots of $f(x)$ are among the odd roots of a related polynomial called its derivative $f'(x)$. We illustrate this with the polynomial from the previous example.



As we can see, not all the odd roots of the derivative $f'(x)$ are even roots of the original polynomial $f(x)$. It can be shown however that every even root of $f(x)$ is an odd root of $f'(x)$.

Representing Polynomials

Derivatives are studied extensively in Calculus. It is not necessary however to know how to differentiate general functions or even to know what a derivative is in order to use the above fact, so students who have not had Calculus need not be dismayed. The world of polynomials is much simpler than the world of general continuous functions in Mathematics. To specify an n^{th} degree polynomial $f(x)$ one need only specify a list of $n+1$ numbers, namely its coefficients.

$$f(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_{n-1} x^{n-1} + c_n x^n$$

In this project these numbers c_k (for $0 \le k \le n$) will be stored an array of length $n+1$. For us then, a polynomial *is* an array. The derivative $f'(x)$ is a polynomial of degree $n-1$ whose *n* coefficients are given by a simple rule.

$$f(x) = c_1 + 2c_2 x + 3c_3 x^2 \cdots + (n-1)c_{n-1} x^{n-2} + nc_n x^{n-1}$$

In other words, if the coefficient array for $f(x)$ is $(c_0, c_1, c_2, c_3, \cdots, c_{n-1}, c_n)$ then the coefficient array for $f'(x)$ is $(c_1, 2c_2, 3c_3, \cdots, (n-1)c_{n-1}, nc_n)$. To put it another way, if c_k (for $0 \le k \le n$) are the coefficients of $f(x)$ and d_k (for $0 \le k \le n-1$) are the coefficients of $f'(x)$, then $d_k = (k+1)c_{k+1}$ (for $0 \le k \le n-1$).

Once the coefficients of $f'(x)$ are calculated, one can run the Bisection Method on $f'(x)$ to find *its* odd roots. Each of the roots x_0 of $f'(x)$ are then plugged into $f(x)$ to see if $f(x_0) = 0$. If so, we have found an even root of $f(x)$. We run into another problem at this stage though. All of these calculations will be done using the type double. Whenever two quantities *u* and *v* are independently computed and stored using a floating point data type, round off and approximation errors are introduced. Even if some Mathematical theory tells us that $u = v$, the stored floating point values will not in general be equal. We can expect however that $|u - v|$ is less than some positive *threshold* value which is close to zero, but not closer than the accuracy of the calculation allows. In particular we cannot directly confirm that $f(x_0) = 0$ since x_0 is at best only an approximation to a root of $f(x)$, and for that matter our computed value for $f(x_0)$ is only an approximation to its actual value. Instead, to check whether x_0 is also a root of $f(x)$, we check that $|f(x_0)| <$ threshold.

Program Specifications

Your program is required to implement the following static functions to carry out the operations described above. Your functions must match the ones below in their name, return type, and parameter list. Your initial task is to fill in the braces with appropriate Java code and test each function thoroughly.

`static double poly(double[] C, double x){....}`

A call to `poly(C, x)` will return the value at *x* of the polynomial with coefficient array *C*. You can accomplish this by writing a loop that multiplies each coefficient by an appropriate power of *x*, and accumulates the sum of all such terms. When the loop terminates return the sum.

`static double[] diff(double[] C){....}`

The call `diff(C)` will return a reference to a newly allocated array *D* containing the coefficients of the polynomial that is the derivative of the polynomial with coefficient array *C*. In other words the function `poly(D, x)` will be the derivative of the function `poly(C, x)`.

`static double findRoot(double[] C, double a, double b, double tolerance){....}`

Assuming `poly(C, a)` and `poly(C, b)` have different signs, a call to `findRoot(C, a, b, tolerance)` will return an approximation to a root of `poly(C, x)` in the interval [*a*, *b*] whose error is no more than `tolerance`. Implement this function by using the Bisection Method illustrated at the end of section 4.11 and in the examples FindRoot1.java, FindRoot2.java and FindRoot3.java on the class webpage. This function has a *precondition* that says the polynomial takes opposite signs at the endpoints of the interval. Therefore it should only be called when that precondition is satisfied. Only after these functions are working, should you begin writing function `main()` for the project. The following steps are offered as a rough outline of program logic.

1. Declare `double` variables `resolution`, `tolerance` and `threshold`, and initialize them to 10^{-2}, 10^{-7} and 10^{-3} respectively. Declare any other variables needed by function `main()`.
2. Get the degree of the polynomial and its coefficients, along with the left and right endpoints of the search interval [*L*, *R*] from the user.
3. Calculate the coefficients of the derivative polynomial by calling your function `diff()`.
4. Enter a loop iterating over all subintervals [*a*, *b*] of width `resolution` forming a partition of [*L*, *R*].
5. For each such subinterval [*a*, *b*]
6. If the polynomial changes signs across [*a*, *b*]
7. Find an odd root of the polynomial in [*a*, *b*] accurate to within `tolerance`
8. Print the value of the root rounded to 5 decimal places

9. Else if the derivative polynomial changes signs across [*a*, *b*]
10. Find an odd root of the derivative polynomial in [*a*, *b*] accurate to within `tolerance`
11. If the absolute value of the polynomial evaluated at this root is less than `threshold`
12. Print the value of the root rounded to 5 decimal places
13. If no even or odd root was found in [*L*, *R*] print a message to that effect

You may define other functions as you see fit, but the ones described above are required for full credit. An example will be posted on the webpage illustrating how one can round and format a double value to a specific number of decimal places. You may play with the parameters `resolution`, `tolerance` and `threshold`, but the specific values mentioned above should result in your output matching the examples exactly. More such examples will be posted on the webpage for testing purposes.

What to turn in

Your final task is to write a `Makefile` for this project along the lines of the one in lab assignment 4. This Makefile should create an executable jar file called `Roots` allowing one to run the program without having to type `java` at the command line. Include a `clean` target as in lab4. Try to write another phony target called `check` that checks the files you submitted. You may also write a `submit` target as in lab4.

Submit the two files `Makefile` and `Roots.java` to the assignment name pa4. This project is considerably more involved than any of the earlier ones, so you will have more time to complete it. You should nevertheless start early, work on one thing at a time, and ask questions of myself, the TAs and on Piazza.