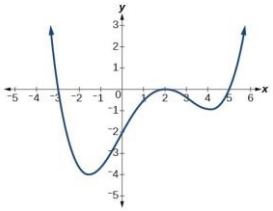

Polynomial Solver Lab

This is a much more interesting lab than it might initially seem. There are many ways in which it can be completed: both calculus-based and algebraic-based solutions. You may use any approach you are most drawn to. I am going to suggest an algebraic-approach using the Bisection method, not because it is common, but because it reveals a lot of interesting artifacts about our computing architecture and our number system.

Finding the roots of a polynomial using the bisection method.

Given any regular polynomials of degree n (where n is the highest power of x), there are n roots (input values where the solution of the polynomial is zero).

So for $x^2 - 2$, there are two roots (-1.4142, 1.4142).
For $x^3 + 2x^2 - x - 2$, there are three roots (-2, -1 and 1)

You can find roots of a polynomial y = f(x) by using the Bisection method: Send a bunch of consecutive integer values as inputs (x) into the polynomial to find their results (y). Whenever successive results change signs (go from positive to negative), we know that the graph passes through zero in between them (or perhaps on one of them), and thus there is a root of the polynomial in there. Let's call the larger of the two successive input values the upper bound and the smaller of the two the lower bound (x_u and x_l).

See if $f(x_u)$ is a root as well as $f(x_l)$. Regardless, there may be a root in between them. So, take the average of x_u and x_l. Call the average x_a. See if the average is a root (if $f(x_a)$ is zero). If it is, you found the root and can move on to the next set of consecutive inputs where the results change signs. If it is not a root, do the following:
If the average (x_a) is positive, make the average the new upper bound (x_u) and repeat.
If the average (x_a) is negative, make the average the new lower bound (x_l) and repeat.

You can model a polynomial in code with a list of descending degree coefficients.

- $x^2 - 2$ can be built with the values [1, 0, -2], i.e. $1x^2 + 0x - 2$.
- $x^3 + 2x^2 - x - 2$ can be built with [1, 2, -1, -2], i.e. $1x^3 + 2x^2 - 1x - 2$.

This all might seem easy, but you will likely come across some interesting dilemmas that speak a lot to the weaknesses of our number system and processor architecture:

Issues:

- If a solution is not a whole number, we will approach the solution as a limit. So, we will have to choose a value that is so close to zero that we can just assume that it is a solution. Consider the following identifier:

```
private static double ALMOST_ZERO = 0.0000000001;
```

Why not final? For different scientific and engineering applications, the client might want to adjust the tolerance value for how close to zero we want to consider to be zero. If the absolute value of the potential solution is less than `ALMOST_ZERO`, then we can call it a solution. It might be nice to allow the client to change the value for `ALMOST_ZERO`.

- We need to choose a range of successive input values (x) to send into the polynomial. If we go from the minimum integer size to the maximum, the program might take too long to run. Assume all solutions can be found between 100000 and 100000.

```
private static int range = 100000;
//looks for roots from -range to +range
```

It might be nice to allow the client to set the min-range and max-range values. Different applications may necessitate different spectrums of input. There is also a mathematic algorithm that can be employed to find the range of inputs for which any solutions must exist. An efficient solution would choose the range that is smaller of the two.

Design decisions are paramount. Should you create a Polynomial object? What methods will be helpful to have? Any sub-task within the process might make for a good method.

Extension:

- make a subclass of a Polynomial – the Quadratic. If a given polynomial is a Quadratic, have it find the solutions using the quadratic formula. If it is a constant or an order 1 polynomial, don't bisect. You can find the solution with a constant-efficient amount of work.

Suggested Helper Methods:

```
//post:  true if x and y are both positive, both negative or both zero
//       otherwise, returns false
//sameSign(-3, -9) => true,   sameSign(-3, 0) => false,
//sameSign(0,0) => true,      sameSign(0, 4) => false,
//sameSign(4, 9) => true,     sameSign(-3, 4) => false
public static boolean sameSign (double x, double y)
{

}

//post:  returns the average (mean) of x and y
//average(5, 6) => 5.5
public static double average(double x, double y)
{

}

//post:  coeff is an array of coefficients of decreasing powers of x
//       returns the polynomial evaluated at value x
//       if coeff is [3, 5, -1], it equates to the polynomial
//       3x^2 + 5x^1 -1x^0
//       so solve(coeff, 2) => 3(2^2) + 5(2^1) -1(2^0) returns 21
public static double solve(double[] coeff, double x)
{

}
```