Polynomials

Out: 4/4
Due: 4/22 by 11:59 PM

Learning Objectives

- Implementing a Class
- Writing Constructors, Destructors, Accessors and Mutator
- Using Objects of a Class

Polynomial

Definition 1. A **univariate polynomial** is a mathematical expression involving a sum of powers in one variable multiplied by coefficients. A polynomial in one variable with constant coefficients is given by the expression $p(x) = \sum_{i=0}^{n} c_i x^i$, where the $c_i{}'s$ are numeric values representing the coefficients. The polynomial is said to be an n^{th}-*degree* polynomial if its highest power is n. For example, $3x^4 - 2x^3 + x^2 - 1$ is a fourth-degree polynomial.

To evaluate a univariate polynomial, a numeric value is substituted for the variable and the expression is evaluated. For example, given the polynomial $p(x) = 3x^4 - 2x^3 + x^2 - 1$, $p(-2) = 67$ since $3 \times (-2)^4 - 2 \times (-2)^3 + (-2)^2 - 1 = 67$. A univariate polynomial can be represented as a *degree-coefficients* array, that is, with its first element representing the degree of the polynomial and the remaining elements its coefficients in order of descending powers. For example, the polynomials $3x^4 - 2x^3 + x^2 - 1$ and $3x^2 + 2x - 5$ are represented as $[4, 3, -2, 1, 0, -1]$ and $[2, 3, 2, -5]$, respectively.

In this project you will write an interactive program that represents univariate polynomials using arrays in which the coefficients are arranged in order of descending powers. The program will have functions that generate a string representation of a polynomial in standard form, compute the indefinite integral of a polynomial, compute a numeric approximation for the definite integral of a polynomial using the trapezoidal rule and evaluate a polynomial.

In order to effectively manage memory, the array representing the polynomial entered by the user will be dynamically allocated. To dyanically allocate an array, the following syntax may be used:

```
arrayType* arrayName = new arrayType[arraySize];
```

Example 1. double* salaries = new double[10];

Dynamically allocating an array gives a programmer the flexibility of being able to create an array of any size during run-time and deallocating the array when it is no longer needed. Unlike static arrays that are deallocated when the function in which they are declared is terminated, dynamically allocated arrays are fully within the programmer's control and may be deallocated at any point after they are created. The example above creates an array of doubles that has the size of 10. To deallocate a dynamically allocated array, use the following syntax:

```
delete[] nameOfArray;
```

This example shows how the *salaries* array can be deallocated.

Example 2. delete[] salaries;

When an array is deallocated, it is no longer accessible. It has to be re-allocated if there is a need to re-use it. Proceed with caution when working with dynamically allocated arrays. When a variable associated with a dynamically allocated array is re-used, be sure to deallocate the array before using the variable. Failure to do so leads to *memory leak*. Memory leak is the allocation of a chunk of memory that contains inaccessible data and cannot be reallocated. There is a lot more to dynamic memory allocation. For now, this is enough to get us through the project.

In this project you will define a class *Polynomial*, representing a univariate polynomial, and then write an interactive program that that perform various operations on the polynomials. This program is a re-write your last programming project, this time using a class to encapsulate the coefficients a polynomial and some functions that can be applied to polynomials. The program will be called *PolyIntegrator*. It will use objects of the *Polynomial* class.

Your program will consist of only the main function and the class. Define the class between the using directive and the main function. The Polynomial class will consist of one private instance variable (data member), poly, an array containing the degree of the polynomial as its first element followed by the coefficients of the polynomial in order of descending powers. The Polynomial class will consist of the several public member functions. As was the case with your previous project, use an *sstream* object when writing the *str* function that returns a string representation of a polynomial.

In addition to the main, your program should consists of the following functions:

```
/**
 * Creates the zero polynomial.
 */
Polynomial()

/**
 * Creates a polynomial with the deg-coefficients array
 * @param degCoeffs an array containg the deg of the polynomial
 * as its first element, followed by its coefficients in
 * order of descending powers.
 */
Polynomial(const double* degCoeffs)

/**
 * deallocates the memory associated with this polynomial
 */
~Polynomial()

/**
 * Evaluates the polynomial represented by the array at the
 * specified value.
 * @param x numeric value at which the polynomial is to be
 * evaluated.
 * @return the value of the polynomial when evaluated with x
 */
double eval(double x) const
```

```
/**
 * Gives a string representation of this polynomial in standard
 * form where zero terms, coefficients 1 or -1, and exponents 1
 * are not displayed.
 * <pre>
 * Note: Rules for Representing a Polynomial in Normalized Form:
 * 1. If the degree of the polynomial is 0, return a string
 *    representing the number.
 * 2. If the degree of the polynomial is 1, return a string
 *    representing the polynomial in the form ax + b, where when
 *    b is zero it should not be displayed and when a is -1 or 1
 *    it should not be displayed as a coefficient of x.
 * 3. If the degree of the polynomial is 2 or more, follow these
 *    steps:
 *    A. Generate the string representation of the highest order
 *       term without using 1, -1 as its coefficient.
 *    B. Generate the string representations of all other, but
 *       the last two, terms beginning from the second highest
 *       order term without the use of 1 and -1 as coefficients
 *       and without including a zero term. Then deal with the
 *       last two terms:
 *       i. If its linear term is non-zero, generate and append
 *          the linear term but without the use of 1 and -1 as
 *          its coefficient and 1 as its exponent.
 *      ii. Finally, append the constant term, the lowest order
 *          term, if it is non-zero.
 * eg: [6, 3, 0, -1, 0, 1, 1, 0] -> 3x^6 - x^4 + x^2 + x
 *     [5, -1, 0, 3, 0, -1, 1] -> -x^5 + 3x^3 - x + 1
 * </pre>
 * @return a string representation of this polynomial
 */
String str() const
```

```
/**
 * Computes a definite integral for the specified
 * polynomial using the trapezoidal rule.
 * @param x1 the lower limit
 * @param x2 the upper limit
 * @return the area under the curve in the interval [x1, x2]
 */
double nIntegrate(double x1, double x2, double delX) const

/**
 * Computes an indefinite integral for the specified
 * polynomial
 * @param c the constant term
 * @return the array representation of the integeral of the
 * specified
 * polynomial with the specified constant term.
*/
Polynomial integrate(double c) const
```

Integral of a Polynomial

The integrals of polynomials have applications in many fields including mathematics, the physical sciences and engineering. When given a univariate polynomial $f(x)$ one way to approximate its definite integral, denoted $F(x_2) - F(x_1) = \int\limits_{x_1}^{x_2} f(x)\,dx$ is to use a numerical algorithm called the trapezoidal rule. Formally, here is a pseudocode description of the trapezoidal rule:

Listing 1: A Trapezoidal Rule Algorithm

```
integral := 0
while $x_1 < x_2$ do
    integral := integral + $\frac{f(x_1)+f(x_1+\Delta x)}{2} \times \Delta x$
    $x_1 := x_1 + \Delta x$
endWhile
return integral
```

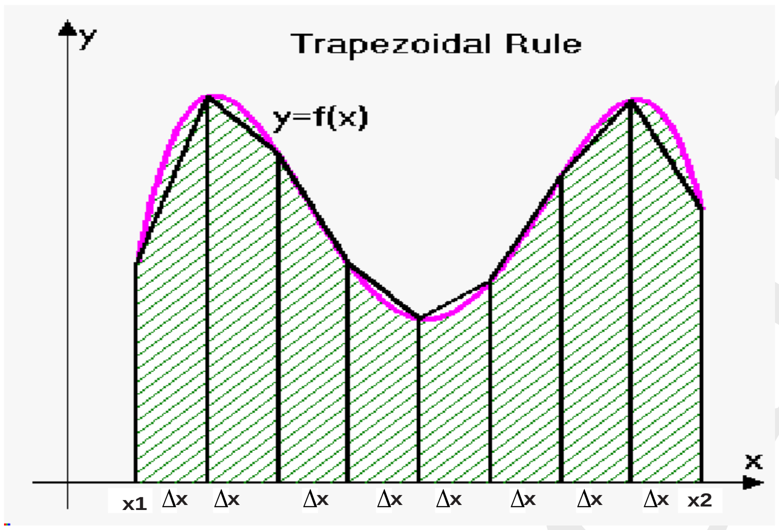

Figure 1: Approximating an Integral Using the Trapezoidal Rule

The area between the x-axis and the curve approximates the definite integral. The smaller the value of Δx, the width of the trapezoid, the better the approximation. Observe that an area between the x-axis and the curve that is below the x-axis is negative and an area above the x-axis is positive.

The indefinite integral of $f(x)$, denoted $\int f(x)dx$, is a polynomial obtained by computing the integral of each term of its terms. The integral of the term $c_i x^i$ is $\frac{c_i}{i+1}x^{i+1}$. For example, the integral of $3x^4$ is $0.6x^5$. When determining the indefinite integral, we add an arbitrary constant so the integral of $f(x)$ has one more term than $f(x)$. Suppose the definite integral of $f(x)$ is denoted $F(x)$, then we can compute the definite integral over a range of values $[x_{max}, x_{min}]$ by calculating $F(x_{max}) - F(x_{min})$ and obtain a numeric value. That is, by evaluating the indefinite integral between two values of the variable x, we obtain the definite integral.

The Main function

Write the program incrementally. First get the constructors, destructor and the *str* member fucntions to work. Then get *eval* to work. You can then write *nIntegrate* followed by *integrate* member functions.

When the functions are all working correctly, you can write the *main*. You may assume that the user always enters the correct number of coefficients with the first being non-zero. When the program begins, it does the following:

1. it prompts the user for the degree of a polynomial and its coefficients in order of descending powers as shown in the sample run. It then dynamically allocates an array and stores the degree and coefficients in an array in *degree-coefficients* format as described in this handout. The highest order coefficient of the polynomial must be non-zero. It uses this array to create a polynomial object by calling the constructor function of the Polynomial class. The program then displays the polynomial using the *str* member function.

2. It prompts the user for a value at which to evaluate the polynomial. It evaluates the polynomial using the *eval* member function and displays the result.

3. The program then prompts the user for the lower and upper limits of the definite integral of the polynomial. It computes the exact value of the definite integral by calculating the difference between the indefinite integral evaluated at the upper and lower limits. It will need to make calls to the *integrate* and *eval* functions to do this.

4. Using 1 as the integral constant and the *integrate* function, it computes the indefinite integral of the polynomial and displays it using the *polyToStr* function.

5. It also approximates the definite integral using the *nIntegrate* member function, the lower and upper limits and 1E-7 as Δx. It displays this approximation and the approximation error.

6. Again using 1 as the integral constant and the *integrate* member function, it computes the second indefinite integral of the polynomial, by integrating the first integral, and displays it using the *str* member function.

7. It computes the exact value of the second definite integral by calculating the difference between the second indefinite integral evaluated at the upper and lower limits.

8. It also approximates the second definite integral using the *nIntegrate* member function, the lower and upper limits and 1E-7 as Δx and the polynomial representing the first indefinite integral. It displays this approximation and the approximation error.

A typically run of the program would be:

Listing 2: Code Segment

```
Enter the degree of the polynomial -> 5

Enter the coefficients -> 3 0 -5 1 -5 0

f(x) = 3x^5 - 5x^3 + x^2 - 5x
Enter x at which to evaluate f(x) -> 2
f(2) = 50

Enter the lower and upper limits for the integrals -> 0 1

I[f(x),dx] = 0.5x^6 - 1.25x^4 + 0.333333x^3 - 2.5x^2 + 1

Exact:
I[3x^5 - 5x^3 + x^2 - 5x dx, 0, 1] = -2.91667
Trapezoid Rule:
I[3x^5 - 5x^3 + x^2 - 5x, deltax = 1E-7, 0, 1] = -2.91667
Error = 5.99673e-07


I[I[f(x),dx],dx] = 0.0714286x^7 - 0.25x^5 + 0.0833333x^4 - 0.833333x^3 + x + 1

Exact:
I[0.5x^6 - 1.25x^4 + 0.333333x^3 - 2.5x^2 + 1 dx, 0, 1] = 0.0714286
Trapezoid Rule:
I[0.5x^6 - 1.25x^4 + 0.333333x^3 - 2.5x^2 + 1, deltax = 1E-7, 0, 1] = 0.0714284
Error = 1.91368e-07
```

Submitting Your Work

1. The source file that you submit must have header comments that follow this template :

```
/**
 * Describe what your program does.
 * @author type your name
 * @since the date the program was written
 * Course: csc1253 Section x <br>
 * Programming Project #: 5 <br>
 * Instructor: Dr. Duncan <br>
 * @author YOUR NAME<br>
 * @since DATE THE PROGRAM WAS LAST MODIFIED
 */
```

2. Verify that your code has no syntax error and that it is ISO C++ 11 compliant. You may also verify that it calculates the integrals correctly by using the online tool at:
https://www.symbolab.com/solver/integral-applications-calculator

3. Once you are convinced that your program works correctly, using Windows explorer, navigate your way to the "My Document" folder. Navigate your way through the following path: *NetBeansProjects* | *PolyIntegrator*. You will see a file called *PolyIntegrator.cpp*, your source code. Right-click the source file. Choose *Send to* | *Compressed (zipped) folder* from the pop-up menu. [If you are not using a PC or Netbeans, the steps for locating and archiving your file will differ.] Rename the zip file *PAWSID_proj05.zip*, where *PAWSID* is the prefix of your LSU/Tiger email address - the characters left of the @ sign.

4. Submit your work (the zip file containing your source file) for grading by dragging and dropping your zip file in the digital drop box on Moodle.