

Polynomial Class using Chatgbt

and Co-pilot

**OOP-CS213-Assignment1\_Task2-2024**

Name: Hager Ahmed Mohamed

ID: 20231197

Email:hagerah5162@gmail.com

Co-pilot

Conversation

First I asked co-pilot to make a polynomial class divided into a header file and implementation file and it did it perfectly and it implemented all the functions. Then I asked him to make the menu but it made vital errors where some options need to have two polynomials to work but it only asked for one input so it isn’t working correctly. So I asked him to make the user enter three polynomials at first then choose any operation he wants, within the operation he can choose the polynomial(s) he wants to do the operation on. And Co-pilot did this very good. Then I realized that it didn’t use the == operator which checks if a polynomial is equal to another polynomial. So I made it add it.

Overall I see that the conversation was very good and it was able to fix all the problems pretty quickly once it understood the requirements. It would have been better if it understood the requirements by itself without me telling it.

Time Complexity

1. **Default constructor Polynomial()**
   * **Time complexity:** O(1)
2. **Parameterized constructor Polynomial(const vector<double>& coefficients)**
   * **Time complexity:** O(n), where n is the size of the input vector coefficients.
3. **Copy constructor Polynomial(const Polynomial& other)**
   * **Time complexity:** O(n), where n is the number of coefficients in other.
4. **Assignment operator Polynomial& operator=(const Polynomial& other)**
   * **Time complexity:** O(n), where n is the number of coefficients in other.
5. **Addition operator Polynomial Polynomial::operator+(const Polynomial& other) const**
   * **Time complexity:** O(max(n, m)), where n and m are the sizes of the two polynomials' coefficient vectors.
6. **Subtraction operator Polynomial Polynomial::operator-(const Polynomial& other) const**
   * **Time complexity:** O(max(n, m)), similar to the addition operator.
7. **Multiplication operator Polynomial Polynomial::operator\*(const Polynomial& other) const**
   * **Time complexity:** O(n \* m), where n and m are the sizes of the coefficient vectors of the two polynomials.
8. **Equality operator bool Polynomial::operator==(const Polynomial& other) const**
   * **Time complexity:** O(n), where n is the size of the coefficient vectors (assuming both polynomials are the same size).
9. **Output operator ostream& operator<<(ostream& out, const Polynomial& poly)**
   * **Time complexity:** O(n), where n is the size of the polynomial.
10. **Degree function int Polynomial::degree() const**

* **Time complexity:** O(1)

1. **Evaluate function double Polynomial::evaluate(double x) const**

* **Time complexity:** O(n), where n is the number of terms in the polynomial.

1. **Composition function Polynomial Polynomial::compose(const Polynomial& q) const**

* **Time complexity:** O(n^2), where n is the degree of the original polynomial.

1. **Derivative function Polynomial Polynomial::derivative() const**

* **Time complexity:** O(n), where n is the degree of the polynomial.

1. **Integral function Polynomial Polynomial::integral() const**

* **Time complexity:** O(n), where n is the number of terms.

1. **Definite integral function double Polynomial::integral(double x1, double x2) const**

* **Time complexity:** O(n), where n is the number of terms.

1. **Root finding function double Polynomial::getRoot(double guess, double tolerance, int maxIter)**

* **Time complexity:** O(k \* n), where k is the number of iterations, and n is the time complexity of evaluating both the polynomial and its derivative.

1. **Set coefficients function void Polynomial::setCoefficients(const vector<double>& coefficients)**

* **Time complexity:** O(n), where n is the size of the input coefficient vector.

1. **Get coefficient function double Polynomial::getCoefficient(int degree) const**

* **Time complexity:** O(1)

Overall, most functions are linear O(n), except for multiplication and composition, which involve quadratic complexity.

Suggestions and Rating of Time Complexity

**Multiplication (operator\*) - O(n \* m):**

* This is quadratic in the worst case, which can become slow for large polynomials.
* **Improvement Suggestion:**
  + Implement a **faster multiplication algorithm** like the **Karatsuba algorithm** (O(n^log3)) or use the **Fast Fourier Transform (FFT)** for polynomial multiplication (O(n log n)). These methods can significantly reduce the time complexity for large polynomials.

**Composition (compose) - O(n^2):**

* Polynomial composition is inherently expensive due to repeated multiplications and additions.
* **Improvement Suggestion:**
  + You might explore **dynamic programming** approaches to cache intermediate compositions if the composition is used frequently in certain scenarios. This won't reduce the worst-case complexity but could help in cases where you repeatedly compose with the same polynomial.

**Root Finding (Newton's Method) - O(k \* n):**

* The current complexity depends heavily on the number of iterations (k) needed for convergence. While this is typical for Newton's method, it can still be slow for polynomials with many terms.
* **Improvement Suggestion:**
  + Consider **hybrid methods** such as combining **bisection** and **Newton's method** to provide more reliable convergence. This could reduce the number of iterations in cases where Newton's method struggles near roots with flat tangents.

**Integral with Bounds (integral(x1, x2)):**

* This is O(n), which is expected. However, if integration is used frequently over large intervals, optimizations could be useful.
* **Improvement Suggestion:**
  + If the same bounds x1 and x2 are used repeatedly, **cache results** of the integral function to avoid recalculating.

Rating: 77.8/100 (for having 4 out of 18 functions not good with TC)

**Cleanliness**

* **Code Organization:** 8/10  
  Well-structured, but duplicated logic in the switch case for polynomial operations reduces cleanliness.
* **Naming Conventions:** 9/10  
  Clear and descriptive function names, but more consistency in variable naming could be helpful.
* **Error Handling:** 7/10  
  Error handling exists, but it’s scattered and can be centralized for better organization.
* **Defensive programming:** 2/10

Very little defensive programming used

**Total Cleanliness: 6.5/10**

**Clarity**

* **User Interface:** 7.5/10  
  The menu and user prompts are quite clear, making the code intuitive and easy to navigate except for the fact that the menu is pretty big.
* **Input Validation:** 8.5/10  
  Some validation exists, but the lack of range checks for selecting polynomials could confuse users.
* **Error Messages:** 9/10  
  Clear error messages, but adding more context (e.g., valid ranges for inputs) would enhance clarity.

**Total Clarity: 8/10**

**Simplicity**

* **Modularization:** 8.5/10  
  Functions are modular, but main() is too long; refactoring case logic would improve simplicity.
* **User Interaction:** 8/10  
  Simple and intuitive interaction, but some additional checks (e.g., limiting polynomial size) could simplify usage.
* **Memory Management:** 9.5/10  
  Efficient use of memory with vectors; no unnecessary complexity in managing polynomials.

**Total Simplicity: 9/10**

**Overall Rating: 83/100**

This program is well-written, but there is room for improvement in cleanliness (avoiding repetition), clarity (input validation), and simplicity (breaking down large functions).

Suggestions/Improvements

 Refactor repetitive code for selecting polynomials by creating a helper function to reduce redundancy.

 Centralize error handling to avoid scattered try-catch blocks and group related error handling logic.

 Add more inline comments to explain non-obvious code sections, improving overall readability.

 Implement input validation to handle invalid inputs like negative degrees or out-of-range polynomial indices.

 Improve error messages to be more informative and guide the user about the expected input range or type.

 Break down the main function into smaller, modular functions for handling specific operations, simplifying the structure.

 Make the polynomial vector dynamic, allowing users to add or delete polynomials during runtime instead of hardcoding a fixed size.

 Add file input/output support to allow saving and loading polynomials, making the program more practical.

 Provide better organization in the user interface by grouping related operations (e.g., arithmetic, evaluations) for easier navigation.

 Add options to dynamically add or remove polynomials during runtime for more flexibility.

 Consider creating a more structured menu system or submenus for operations, improving usability.

ChatGPT

Conversation

I told GPT that I needed to develop a Polynomial class in C++. I provided a header file and asked for the implementation to be split into three files: a header file, an implementation file, and a main file. I specifically requested a user-friendly menu that would allow me to select various operations freely without errors.

What GPT Did

Class Definition: GPT created the Polynomial class based on the provided header file. It included necessary member functions such as constructors, destructor, arithmetic operators, equality operator, and utility functions like degree, evaluate, compose, derivative, and integral.

Menu in Main: GPT implemented a main program that enabled user interaction with the polynomial functionalities through a menu that wasn’t clear and very hard to use so I asked it to make a more simpler and clear menu and it did so.

Issues We Encountered

Missing Composition Function: Initially, the compose function was not included, which caused errors when I tried to use it. I pointed this out, and GPT added the function to correctly combine polynomials.

Operator Overloading Errors: During the build process, I faced compilation errors related to operator overloading, especially when trying to multiply a polynomial with a coefficient. GPT helped correct this by adjusting the operator functions to ensure they worked with both polynomial objects and scalar values.

Incorrect Function Calls: I encountered errors when trying to use coefficients directly as polynomials without converting them first. GPT clarified the need to convert coefficients into polynomial objects before using them in operations.

Build Failures: The final build failed several times due to type mismatches and issues with function calls. I worked with GPT to revisit the logic in the compose function and refine the arithmetic operations.

Final Outcome

After resolving the errors and improving the implementation, GPT successfully built the program. The menu was functioning well, allowing me to select various polynomial operations, but the output was not very clear and understandable.

Time Complexity

 Default Constructor: O(1)

 Constructor with Coefficients: O(n)

 Copy Constructor: O(n)

 Destructor: O(1)

 Assignment Operator: O(n)

 Addition Operator: O(n)

 Subtraction Operator: O(n)

 Multiplication Operator: O(n \* m)

 Equality Operator: O(n)

 Output Operator: O(n)

 Degree Function: O(1)

 Evaluate Function: O(n)

 Compose Function: O(n^2 \* m)

 Derivative Function: O(n)

 Integral Function: O(n)

 Definite Integral: O(n)

 Root Finding (Newton's Method): O(n \* maxIter)

 Set Coefficients: O(n)

 Get Coefficient: O(1)

**Suggestions for Improvement**

**1. Multiplication Operator: O(n \* m)**

* **Suggestion**: Implement **Karatsuba multiplication** or **Fast Fourier Transform (FFT)** for polynomial multiplication. These methods can reduce the time complexity to approximately O(n log n) for large polynomials.

**2. Compose Function: O(n² \* m)**

* **Suggestion**: Instead of evaluating the composition in a naive manner, you can:
  + Use **Horner's method** to evaluate polynomials efficiently.
  + Consider caching previously computed results for polynomial compositions to avoid redundant calculations.

**3. Root Finding (Newton's Method): O(n \* maxIter)**

* **Suggestion**: Instead of evaluating the polynomial and its derivative at every iteration, consider using **memoization** to cache the results of previously computed values. This will speed up the evaluation during subsequent iterations.

**4.Precomputation**

* For functions like definite integrals or derivatives, consider precomputing values when possible, storing them, and returning the precomputed result for repeated queries

Overall Time Complexity Rating: 75 out of 100 (some complexities are too long)

**Cleanliness**

1. **Code Organization**: 7.5/10  
   The structure is mostly good, but there are some redundancies, especially in handling polynomial operations. Centralizing common logic would improve it.
2. **Naming Conventions**: 8/10  
   Function names are fairly descriptive, but variable names could benefit from more consistency and clarity, especially for readability.
3. **Error Handling**: 6/10  
   Error handling is present, but scattered. Consolidating it and adding more thorough checks would make it cleaner and easier to follow.
4. **Defensive Programming**: 3/10  
   The code lacks sufficient defensive programming to catch unexpected inputs or edge cases. Adding more robust error-checking mechanisms would greatly improve it.

**Total Cleanliness**: **6.6/10**

**Clarity**

1. **User Interface**: 7/10  
   The menu system is clear but quite extensive, which could overwhelm users. A more concise approach would help, but overall, it’s fairly intuitive.
2. **Input Validation**: 7.5/10  
   Some validation is present, but there are gaps, especially in terms of boundary checks. More validation could reduce user errors.
3. **Error Messages**: 8.5/10  
   Error messages are clear, but adding more detail about what went wrong (e.g., acceptable input ranges) would enhance user understanding.

**Total Clarity**: **7.7/10**

**Simplicity**

1. **Modularization**: 8/10  
   The code is reasonably modular, but main() is a bit too lengthy and could be broken down further to simplify the logic.
2. **User Interaction**: 7.5/10  
   The interaction is intuitive, but simplifying the menu structure and adding more user-friendly checks (like limiting options) would help reduce confusion.
3. **Memory Management**: 9/10  
   Efficient use of resources, particularly with vectors. Memory management is straightforward and doesn’t add unnecessary complexity.

**Total Simplicity**: **8.2/10**

**Overall Rating : 73/100**

The code is generally well-structured and easy to follow but could benefit from improvements in defensive programming and simplifying some of the operations to enhance both cleanliness and clarity.

Suggestions/Improvements

**Refactor Common Logic**: Identify areas with repeated code and consolidate them into reusable functions. This will improve readability and reduce redundancy.

**Enhance Defensive Programming**: Add more robust error-checking and validation to handle unexpected inputs or edge cases. This will prevent crashes and make the program more reliable.

**Simplify User Interaction**: Streamline the menu system by reducing its complexity. Consider breaking large menus into smaller, more manageable steps or grouping related options together.

**Improve Variable Naming Consistency**: Ensure variable names follow a consistent and descriptive naming convention, making the code easier to understand at a glance.

**Centralize Error Handling**: Create a more uniform approach to error handling, so that it is clear where and how errors are managed. This will make debugging easier.

**Break Down Long Functions**: For functions like main(), consider breaking them down into smaller, more focused helper functions. This will make the code more modular and maintainable.

**Provide Detailed Feedback to Users**: Enhance clarity by giving users more specific feedback, such as acceptable input ranges or hints when errors occur, making the program more user-friendly.

**Optimize Input Validation**: Strengthen input validation checks to ensure inputs fall within expected ranges, reducing potential user errors and making the system more foolproof.