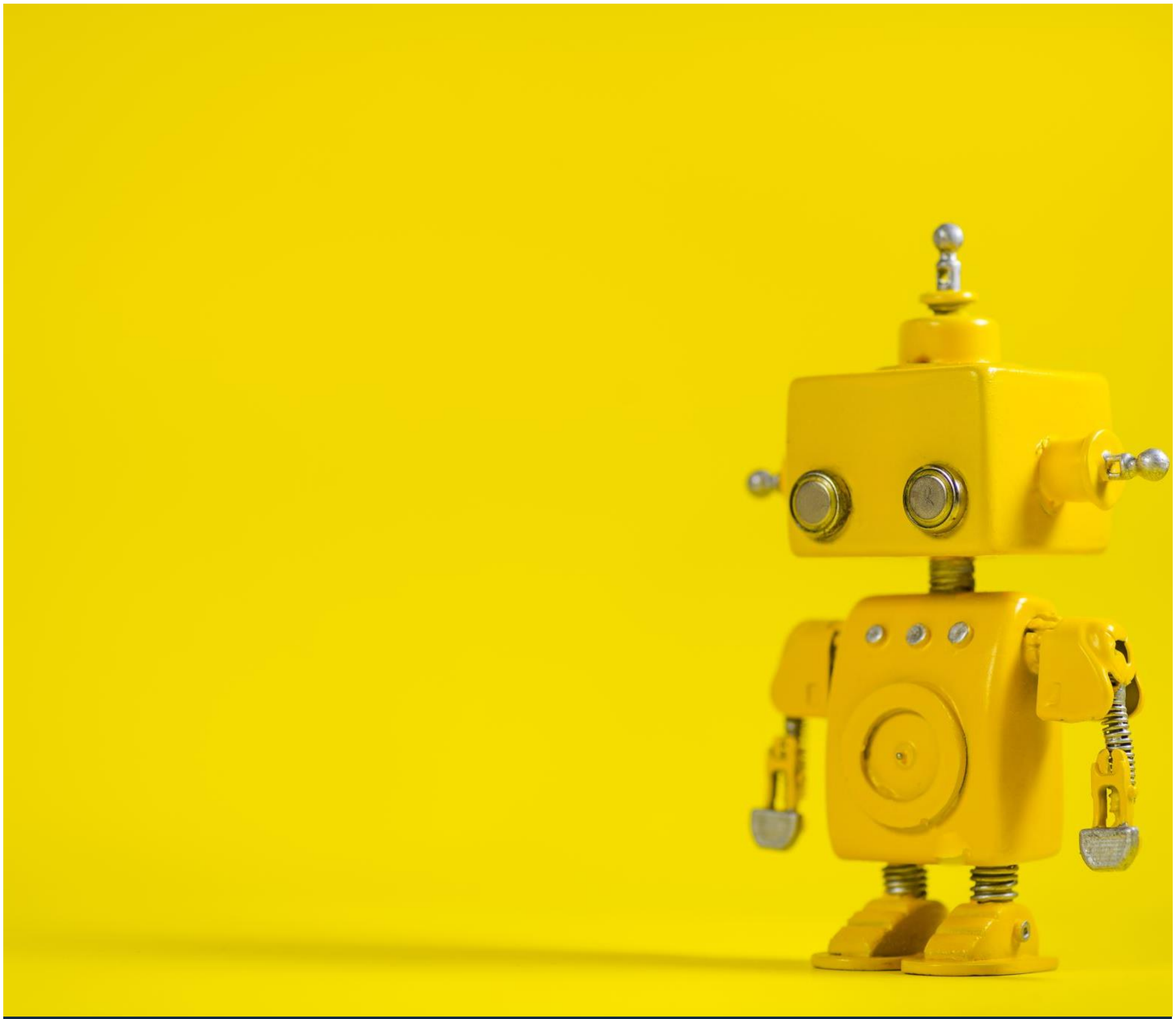


AI Written Code Assessment

A1_T2 OOP

Loai Hataba	Abdallah Mohammed	Hossam Abdelaziz
20230553	20230231	20230121

AI-Written Code Assessment

Introduction

Since humans started walking on this earth they kept innovating, discovering, and utilizing they're surroundings. Just when you think that you've reached the peak of technological and industrial evolution you're surprised with more.

In 1829 when the sewing machine was made it was heavily attacked because a lot of people worked in sewing clothes by hand which meant it would run them out of jobs. And the machine did in fact was much faster and more efficient and cost less, so businesses replaced the manual sewers with them; Yet it opened countless opportunities for people to use them. The machine took out the tedious effort that was exerted in order to make only one piece of clothing.

Similarly, the new rise of Artificial Intelligence technology is becoming more rapid, and it will surely take a lot of people out of their jobs, yet it will open a lot of other jobs. AI's number one purpose is to get rid of tedious routine work.

In spite of what has been stated we're trying to answer the question "will AI replace programmers", we have asked two different ai models to implement a polynomial class in C++ to assess their code quality, results, elegance, and their user experience.

The test was done using a 100 different test cases being as inclusive as possible including polynomials with complex roots and degrees up to a 1000!

The testing was done assessing the models' ability to write codes implementing a class of a polynomial that had 15 different methods including:

- Assignment operator
- Arithmetic operators
- Equality operator
- Output operator
- Degree
- Evaluate
- Compose
- Derivative
- Indefinite Integral
- Definite Integral
- Get Root
- Get coefficient
- Set coefficient

Parameters

- **AI Models Used**: Chat-GPT (4o mini) / Claude (3.5 Sonnet)
- **Initial Prompt**: Both Ai models were prompted with the same prompt

  > ***"You're a senior programmer with a lot of knowledge in C++, I want you to complete this class … (provide the header file)"***

  One of the most important aspects of a good prompt is the persona, where you tell the ai model who it should be so it has a clearer path of where to search and which information to provide; so by telling the models that they're programmers who are knowledgeable in C++ that would potentially help the results be better than just asking them to complete the class.

- **Modifications/Reprompting**: Both Models have been prompted more than once (with nearly the same prompts) to get somewhat of an acceptable result that matches the desired outcome.

1. Correctness

❖ Chat-GPT:

- Overall, the code was acceptable, the code works well in most normal cases in most operations, and the code seems to cover a lot of edge cases.
- Yet when met with cases such as polynomials with complex it fails.
- The get root function was a complete disaster even after a couple of tries to correct its code the model completely ruined the code making it unusable.
- The model didn't add any type of input checking or safeguarding the code, whenever an invalid input the program either runs forever and needs to be force closed or the there's some kind of segmentation error or logical that forces the program to completely quit.
- The model could sometimes slip up and forget very basic and easy stuff that would seem not that important but every detail matters, and sometimes it ignores some of your requests.
    - Example: when provided the model to make the get root value give back multiple roots if exists and set default values for the parameters it failed to comply with the requests and yielded the code nearly as it is, even a bit worse, not adding the multiple roots feature nor setting the default parameters and removing the input asking the user for the guess and tolerance values, it had to be reminded again that it didn't do what it was asked to do.
- The user experience was a bit annoying having to correct the ai continuously to get something that works.
- The code after being tested on a 100 different test cases failed at!!!

- **Rating: 70%**

❖ Claude:

- The code correctly implements polynomial operations and provides a comprehensive set of features for manipulation and evaluation
- As for edge cases, the code removes leading zeros when constructing polynomials, ensuring that the polynomial degree and operations don't get skewed by unnecessary terms. However, input validation for edge cases like empty coefficients and division by zero in Newton's method should be handled more thoroughly.
- Most functions handled the tests well, except for the getRoot function, which struggled. This is reasonable, as it uses Newton's method, an older technique that becomes less accurate as the polynomial's degree increases.

- **Error Handling**: The code lacks robust error handling in some cases (e.g., division by zero in Newton's method, input validation). Adding checks would improve reliability, especially for edge cases.
- While the model occasionally forgets to complete certain tasks, it has never ignored a prompt entirely
    - For example, when asked to create a menu for operations, it only generated one for 8 operations and left out the remaining ones. Additionally, when prompted to edit the behavior of a specific function—such as its nature or return type—the model successfully made the changes to the function itself but neglected to update the menu or the header file. As a result, I frequently had to remind it to synchronize the menu with the latest function updates.
- The code failed at !!!
- **Rating: 85%**

2. Efficiency

- **Time Complexity**: Both of the models had O(n) time in most of the methods averaging for about 4-8 microseconds (using chrono library) with the longest being O(n*m) where there's two inputs (composition)
- **Space Complexity**: Again, both of them used minimal storage where they just used variables and vectors.
- Both Models achieved very well in terms of speed and resource allocation in the best ways possible
- **Rating: 100%** (Both)

---

3. Elegance

❖ **Chat-GPT:**

- Code elegance is definitely one of gpt's strengths where using its huge training sets it figures out the best most elegant solution for a problem.
- The code overall didn't take up a lot of coding space with a total of 470 lines of code (very acceptable) with some methods only containing one line of execution.
- The model uses brilliant and efficient algorithms and techniques to solve problems and in as little coding space as it can possibly can, it achieves its goal with minimal, clear lines of code.
- **Rating: 85%**

❖ **Claude:**

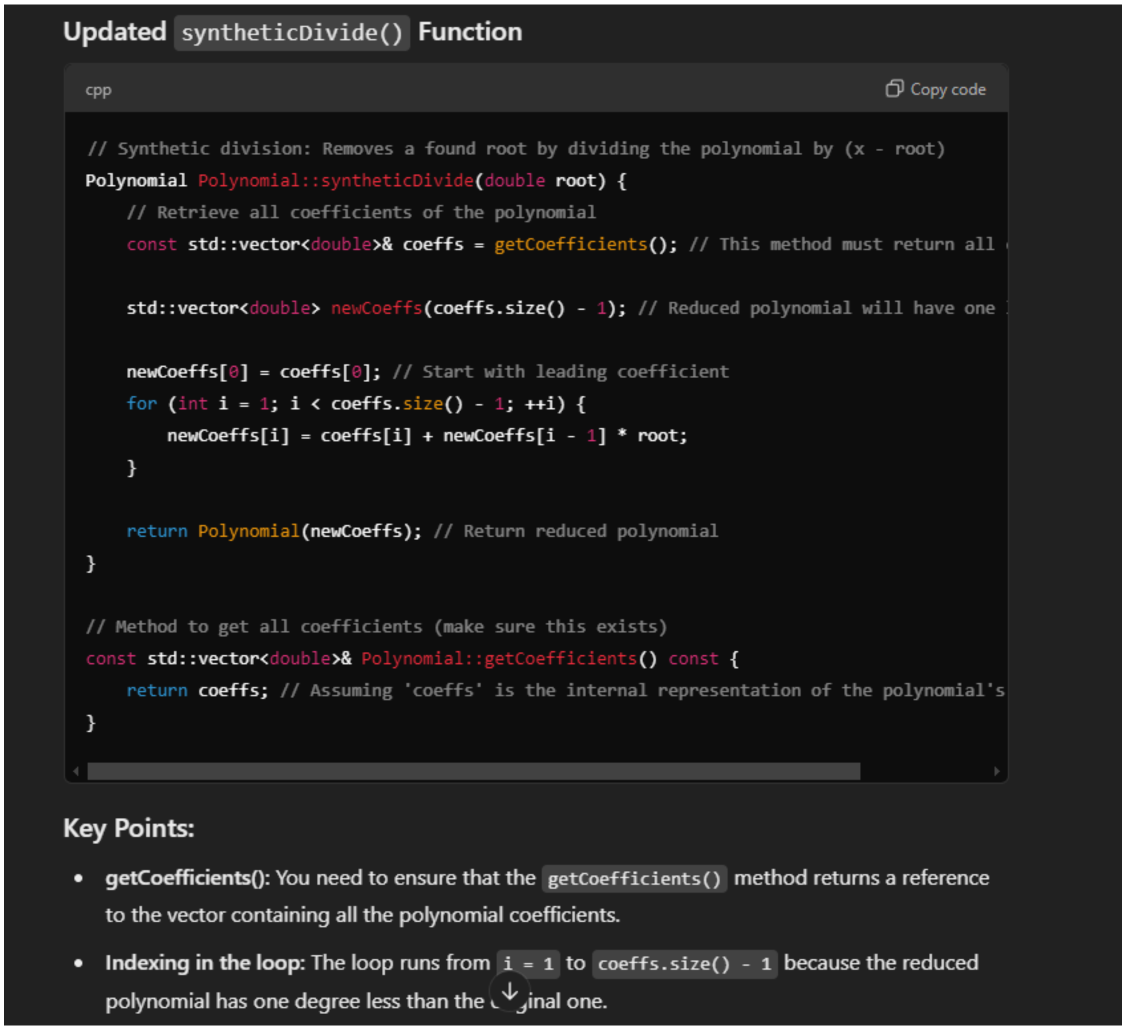
- The code balances functionality and simplicity. While operations like differentiation and integration are user-friendly, Newton's root-finding method can be somewhat overwhelming.
- Claude efficiently managed the code space, completing the task in just 356 lines of code.
- **Rating: 85%**

---

4. Cleanliness

❖ Chat-GPT:

- Despite the class being relatively not small, GPT wrote most of the code in clean and very human readable way with proper variable naming conventions and indentations.
- The model uses comments quite on point where it knows where a comment would be necessary to explain that line of code.
- A plus would be the color coding and code snapshots that the model uses to present the code, and also providing very detailed explanations under the code for what every part of the code does.
- Another thing we really liked about it also was that if something in the code was still unclear, we could easily ask the model and it would walk us step by step on what every line does exactly.



Updated `syntheticDivide()` Function

cpp

```cpp
// Synthetic division: Removes a found root by dividing the polynomial by (x - root)
Polynomial Polynomial::syntheticDivide(double root) {
    // Retrieve all coefficients of the polynomial
    const std::vector<double>& coeffs = getCoefficients(); // This method must return all

    std::vector<double> newCoeffs(coeffs.size() - 1); // Reduced polynomial will have one

    newCoeffs[0] = coeffs[0]; // Start with leading coefficient
    for (int i = 1; i < coeffs.size() - 1; ++i) {
        newCoeffs[i] = coeffs[i] + newCoeffs[i - 1] * root;
    }

    return Polynomial(newCoeffs); // Return reduced polynomial
}

// Method to get all coefficients (make sure this exists)
const std::vector<double>& Polynomial::getCoefficients() const {
    return coeffs; // Assuming 'coeffs' is the internal representation of the polynomial's
}
```

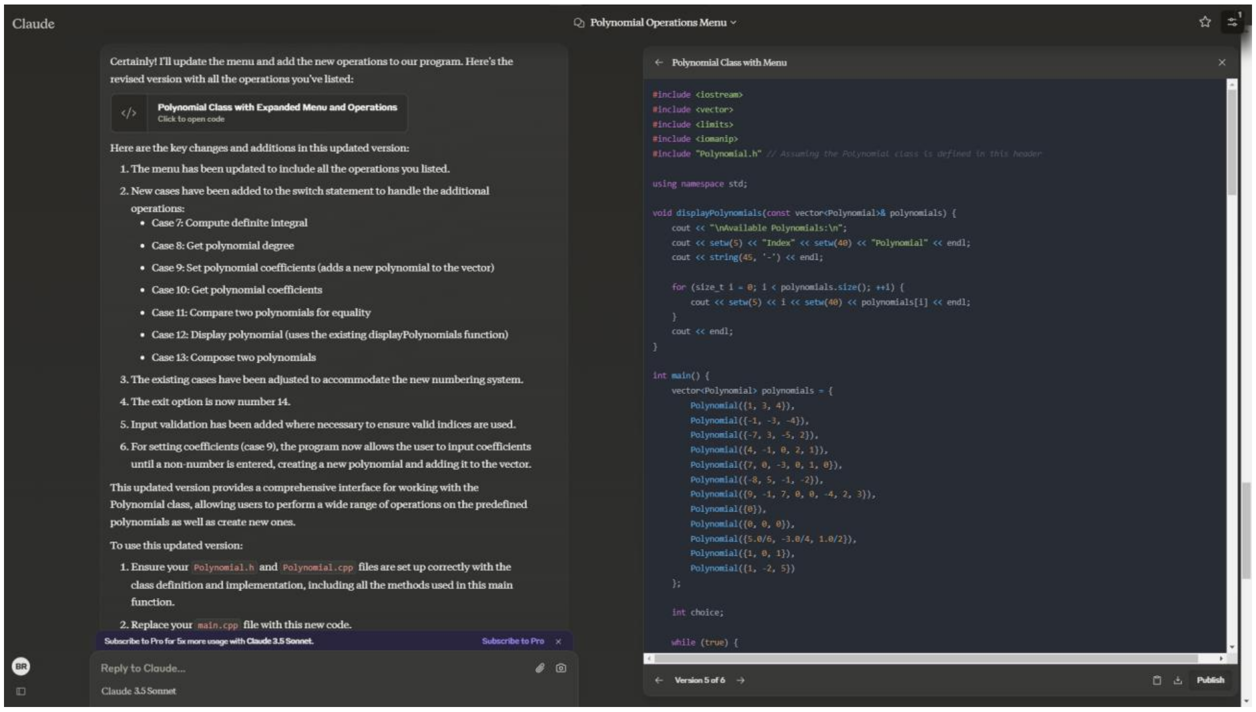
Key Points:

- **getCoefficients():** You need to ensure that the `getCoefficients()` method returns a reference to the vector containing all the polynomial coefficients.
- **Indexing in the loop:** The loop runs from `i = 1` to `coeffs.size() - 1` because the reduced polynomial has one degree less than the ginal one.

- **Rating: 90%**

❖ Claude:

- **Code structure**: The code is structured well with clear separation of header and implementation. It follows common C++ practices with proper use of encapsulation and operator overloading.
- **Naming conventions**: Most function and variable names are descriptive (e.g., degree (), evaluate(), derivative()), making the code easier to follow.
- **Code comments**: The code could benefit from more comments, especially for more complex parts like the composition, root-finding, and evaluation algorithms. This would make it easier for future readers to understand the reasoning behind certain design choices.
- Additionally, it is important to note that Claude is designed for complex tasks, such as code generation, which is reflected in its user interface by separating the code from its explanations.


Claude
Polynomial Operations Menu
Polynomial Class with Expanded Menu and Operations
Click to open code
Subscribe to Pro for 5x more usage with Claude 3.5 Sonnet.
Subscribe to Pro
Reply to Claude...
Claude 3.5 Sonnet
Polynomial Class with Menu
Version 5 of 6
Publish

Certainly! I'll update the menu and add the new operations to our program. Here's the revised version with all the operations you've listed:

Here are the key changes and additions in this updated version:

1. The menu has been updated to include all the operations you listed.
2. New cases have been added to the switch statement to handle the additional operations:
   - Case 7: Compute definite integral
   - Case 8: Get polynomial degree
   - Case 9: Set polynomial coefficients (adds a new polynomial to the vector)
   - Case 10: Get polynomial coefficients
   - Case 11: Compare two polynomials for equality
   - Case 12: Display polynomial (uses the existing displayPolynomials function)
   - Case 13: Compose two polynomials
3. The existing cases have been adjusted to accommodate the new numbering system.
4. The exit option is now number 14.
5. Input validation has been added where necessary to ensure valid indices are used.
6. For setting coefficients (case 9), the program now allows the user to input coefficients until a non-number is entered, creating a new polynomial and adding it to the vector.

This updated version provides a comprehensive interface for working with the Polynomial class, allowing users to perform a wide range of operations on the predefined polynomials as well as create new ones.

To use this updated version:

1. Ensure your `Polynomial.h` and `Polynomial.cpp` files are set up correctly with the class definition and implementation, including all the methods used in this main function.
2. Replace your `main.cpp` file with this new code.

```cpp
#include <iostream>
#include <vector>
#include <limits>
#include <iomanip>
#include "Polynomial.h" // Assuming the Polynomial class is defined in this header

using namespace std;

void displayPolynomials(const vector<Polynomial>& polynomials) {
    cout << "\nAvailable Polynomials:\n";
    cout << setw(5) << "Index" << setw(40) << "Polynomial" << endl;
    cout << string(45, '-') << endl;

    for (size_t i = 0; i < polynomials.size(); ++i) {
        cout << setw(5) << i << setw(40) << polynomials[i] << endl;
    }
    cout << endl;
}

int main() {
    vector<Polynomial> polynomials = {
        Polynomial({1, 3, 4}),
        Polynomial({-1, -3, -4}),
        Polynomial({-7, 3, -5, 2}),
        Polynomial({4, -1, 0, 2, 1}),
        Polynomial({7, 0, -3, 0, 1, 0}),
        Polynomial({-8, 5, -1, -2}),
        Polynomial({9, -1, 7, 0, 0, -4, 2, 3}),
        Polynomial({0}),
        Polynomial({0, 0, 0}),
        Polynomial({5.0/6, -3.0/4, 1.0/2}),
        Polynomial({1, 0, 1}),
        Polynomial({1, -2, 5})
    };

    int choice;

    while (true) {
```

Rating: 70%

5. Summary:

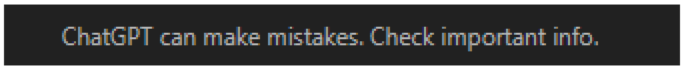
❖ Chat-GPT:

- Overall, the model did a fairly decent job in completing roughly the main idea of having a polynomial class with several methods.
- Yet aside from simple subtraction, addition, and simple integration the code is vulnerable to bugs and total program shutdowns.
- The experience using the model and having a conversation with it was definitely not an easy task, perhaps a bit frustrating when you're literally asking the model for something and it completely ignores it and responds with "of course here's your updated code" and the code isn't changed a single bit; another point is when the model gives you a fix for a problem but it completely obliterates another part of the code, where sometimes maybe the model can fix it or it may be can't and then you have no choice but to pray that you had some backup for your previous code.
- We would give the model an overall rating of **75%**

❖ Claude:

- The implementation features a well-encapsulated, object-oriented design. It effectively handles most edge cases, particularly concerning polynomial degree and coefficients.
- There is room for improvement in error handling, especially related to root-finding and user input. Additionally, increasing the number of comments and documentation would help clarify the more complex sections of the code.
- During my interaction with the model, the only Aggravating aspect was when it made edits to a function's code without synchronizing those changes in the menu or header. Aside from this issue, the model demonstrated a strong understanding of the prompts and provided efficient responses. Overall, it was a great experience.
- Would give an overall rating of: **85%**

It has been concluded that AI in abstraction won't replace humans (Programmers) that easily, for now at least. To use AI to program you would still solid knowledge of what you're asking the model to do so you can notice any bugs or errors that may arise from the code, on the models themselves its written that you should check the information that the model provides because it may not always be accurate.


ChatGPT can make mistakes. Check important info.

AI models exceed well when it comes to explaining complex concepts and breaking them into easier terminologies and using simple analogies until you get the information which is quite impressive.

Links:

-Chat-GPT conversation 1(class implementation): https://chatgpt.com/share/670fd569-b568-8011-8501-c96948df8a49

-Chat-GPT conversation 2(menu): https://chatgpt.com/share/670fd5cd-ee64-8011-98e0-78e1dd90b984

-Claude conversation 1: https://claude.ai/chat/9eb78b11-cfd2-475a-8ed1-b93ae5c3155e

-Claude conversation 2: https://claude.ai/chat/70a67b7d-6719-4ba7-b29a-60b2665016fa

-Claude conversation 3: https://claude.ai/chat/4ddd0f76-8837-4db3-b5bc-3b30d3c1d2e0

-Claude conversation 4(get root fix): https://claude.ai/chat/80922f3e-7e1c-4e0b-8654-d6357522f834

Appendix

Code 1 (ChatGPT)

```
#include "Polynomial.h"
#include <vector>
#include <iostream>
#include <chrono>
using namespace std;

Polynomial::Polynomial() : coeffs(1, 0.0) {}
// Constructor with coefficients
Polynomial::Polynomial(const vector<double>& coefficients) : coeffs(coefficients) {}
// Copy constructor
Polynomial::Polynomial(const Polynomial& other) : coeffs(other.coeffs) {}
// Destructor
Polynomial::~Polynomial() {}
// Assignment operator
Polynomial& Polynomial::operator=(const Polynomial& other) {
    if (this != &other) {
        coeffs = other.coeffs;
    }
    return *this;
}
// Addition operator
Polynomial Polynomial::operator+(const Polynomial& other) const {
    vector<double> result(max(coeffs.size(), other.coeffs.size()), 0.0);
    for (size_t i = 0; i < result.size(); ++i) {
        if (i < coeffs.size()) result[i] += coeffs[i];
        if (i < other.coeffs.size()) result[i] += other.coeffs[i];
    }
    return Polynomial(result);
}
// Subtraction operator
Polynomial Polynomial::operator-(const Polynomial& other) const {
    vector<double> result(max(coeffs.size(), other.coeffs.size()), 0.0);
    for (size_t i = 0; i < result.size(); ++i) {
        if (i < coeffs.size()) result[i] += coeffs[i];
        if (i < other.coeffs.size()) result[i] -= other.coeffs[i];
    }
```

```cpp
    return Polynomial(result);
}
// Multiplication operator
Polynomial Polynomial::operator*(const Polynomial& other) const {
    vector<double> result(coeffs.size() + other.coeffs.size() - 1, 0.0);
    for (size_t i = 0; i < coeffs.size(); ++i) {
        for (size_t j = 0; j < other.coeffs.size(); ++j) {
            result[i + j] += coeffs[i] * other.coeffs[j];
        }
    }
    return Polynomial(result);

// Equality operator
bool Polynomial::operator==(const Polynomial& other) const {
    return coeffs == other.coeffs;
}
// Output operator (friend function)
ostream& operator<<(ostream& out, const Polynomial& poly) {
    for (int i = poly.coeffs.size() - 1; i >= 0; --i) {
        if (poly.coeffs[i] != 0) {
            if (i != poly.coeffs.size() - 1 && poly.coeffs[i] > 0) out << "+";
            out << poly.coeffs[i];
            if (i > 0) out << "x";
            if (i > 1) out << "^" << i;
            out << " ";
        }
    }
    return out;
}
// Utility functions
int Polynomial::degree() const {
    return coeffs.size() - 1;
}
// Evaluate the polynomial at x
double Polynomial::evaluate(double x) const {
    double result = 0;
    for (int i = coeffs.size() - 1; i >= 0; --i) {
        result = result * x + coeffs[i];
    }
    return result;
}
// Derivative of the polynomial
Polynomial Polynomial::derivative() const {
    if (coeffs.size() <= 1) return Polynomial();
    vector<double> deriv(coeffs.size() - 1);
    for (size_t i = 1; i < coeffs.size(); ++i) {
        deriv[i - 1] = coeffs[i] * i;
    }
    return Polynomial(deriv);
}
// Integral of the polynomial (indefinite integral, constant term is 0)
Polynomial Polynomial::integral() const {
    vector<double> integ(coeffs.size() + 1, 0.0);
    for (size_t i = 0; i < coeffs.size(); ++i) {
        integ[i + 1] = coeffs[i] / (i + 1);
    }
    return Polynomial(integ);
}
// Definite integral from x1 to x2
double Polynomial::integral(double x1, double x2) const {
    Polynomial integ = integral();
    return integ.evaluate(x2) - integ.evaluate(x1);
}
// Get Root
double Polynomial::getRoot(double guess, double tolerance, int maxIter) {

    Polynomial deriv = derivative();
    for (int i = 0; i < maxIter; ++i) {
        double f_val = evaluate(guess);
        double f_deriv = deriv.evaluate(guess);
        if (abs(f_val) < tolerance) return guess;
        guess -= f_val / f_deriv;
    }
    return guess;
}
// Get coefficient of a specific degree
double Polynomial::getCoefficient(int degree) const {
    if (degree < 0 || degree >= coeffs.size()) return 0.0;
    return coeffs[degree];
}
```

```cpp
// Set coefficients
void Polynomial::setCoefficients(const std::vector<double>& coefficients) {
    coeffs = coefficients;
}
// Composition of polynomials
Polynomial Polynomial::compose(const Polynomial& q) const {
    Polynomial result({0});
    Polynomial temp({1});
    for (size_t i = 0; i < coeffs.size(); ++i) {
        result = result + temp * Polynomial({coeffs[i]});
        temp = temp * q;
    }
    return result;
}
void displayPolynomialMenu() {
    cout << "\n--- Polynomial Menu ---\n";
    cout << "1. Add two polynomials\n";
    cout << "2. Subtract two polynomials\n";
    cout << "3. Multiply two polynomials\n";
    cout << "4. Evaluate polynomial\n";
    cout << "5. Get polynomial derivative\n";
    cout << "6. Get polynomial integral\n";
    cout << "7. Compute definite integral\n";
    cout << "8. Get polynomial degree\n";
    cout << "9. Set polynomial coefficients\n";
    cout << "10. Get polynomial coefficients\n";
    cout << "11. Compare two polynomials for equality\n";
    cout << "12. Display polynomial\n";
    cout << "13. Compose two polynomials\n";
    cout << "14. Get root using Newton's method\n"; // Added menu option
    cout << "15. Exit\n"; // Updated exit option
}
vector<double> inputCoefficients() {
    int degree;
    cout << "Enter the degree of the polynomial: ";
    cin >> degree;

    vector<double> coefficients(degree + 1);
    cout << "Enter coefficients (constant term first): ";
    for (int i = 0; i <= degree; ++i) {
        cin >> coefficients[i];
    }
    return coefficients;
}

int main() {
    int choice;
    do {
        displayPolynomialMenu();
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1: {
                // Addition
                cout << "Input coefficients for the first polynomial:\n";
                Polynomial p1(inputCoefficients());
                cout << "Input coefficients for the second polynomial:\n";
                Polynomial p2(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Perform the addition
                Polynomial result = p1 + p2;
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Result of addition: " << result << endl;
                cout << "Time taken for addition: " << duration.count() << " microseconds" <<
endl;
                break;
            }
            case 2: {
                // Subtraction
                cout << "Input coefficients for the first polynomial:\n";
                Polynomial p1(inputCoefficients());
                cout << "Input coefficients for the second polynomial:\n";
                Polynomial p2(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Perform the subtraction
                Polynomial result = p1 - p2;
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Result of subtraction: " << result << endl;
                cout << "Time taken for subtraction: " << duration.count() << " microseconds" 
<< endl;
                break;
            }
            case 3: {
                // Multiplication
                cout << "Input coefficients for the first polynomial:\n";
                Polynomial p1(inputCoefficients());
                cout << "Input coefficients for the second polynomial:\n";
                Polynomial p2(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Perform the multiplication
                Polynomial result = p1 * p2;
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Result of multiplication: " << result << endl;
                cout << "Time taken for multiplication: " << duration.count() << " 
microseconds" << endl;
                break;
            }
            case 4: {
                // Evaluation
                cout << "Input coefficients for the polynomial:\n";
                Polynomial p(inputCoefficients());
                double x;
                cout << "Enter value of x: ";
                cin >> x;
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Perform the evaluation
                double result = p.evaluate(x);
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Polynomial evaluated at " << x << ": " << result << endl;
                cout << "Time taken for evaluation: " << duration.count() << " microseconds"
<< endl;
                break;
            }
            case 5: {
                // Derivative
                cout << "Input coefficients for the polynomial:\n";
                Polynomial p(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Compute the derivative
                Polynomial deriv = p.derivative();
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Derivative: " << deriv << endl;
                cout << "Time taken for derivative: " << duration.count() << " microseconds"
<< endl;
                break;
            }
            case 6: {
                // Indefinite integral
                cout << "Input coefficients for the polynomial:\n";
                Polynomial p(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Compute the integral
                Polynomial integ = p.integral();
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Integral: " << integ << endl;
                cout << "Time taken for integral: " << duration.count() << " microseconds" <<
endl;
                break;
            }
            case 7: {
                // Definite integral
                cout << "Input coefficients for the polynomial:\n";
                Polynomial p(inputCoefficients());
                double x1, x2;
                cout << "Enter lower limit of integration: ";
                cin >> x1;
                cout << "Enter upper limit of integration: ";
                cin >> x2;
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Compute the definite integral
                double result = p.integral(x1, x2);
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Definite integral from " << x1 << " to " << x2 << ": " << result << 
endl;
                cout << "Time taken for definite integral: " << duration.count() << " 
microseconds" << endl;
                break;
            }
            case 8: {
                // Degree
                cout << "Input coefficients for the polynomial:\n";
                Polynomial p(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Get the degree
                int degree = p.degree();
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Degree of polynomial: " << degree << endl;
                cout << "Time taken to get degree: " << duration.count() << " microseconds" << 
endl;
                break;
            }
            case 9: {
                // Set coefficients
                cout << "Input coefficients for the polynomial:\n";
                Polynomial p(inputCoefficients());
                cout << "Setting new coefficients for this polynomial.\n";
                p.setCoefficients(inputCoefficients());
                cout << "Coefficients set successfully.\n";
                break;
            }
            case 10: {
                // Get coefficient
                cout << "Input coefficients for the polynomial:\n";
                Polynomial p(inputCoefficients());
                int degree;
                cout << "Enter degree to get coefficient: ";
                cin >> degree;
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Get coefficient
                double coeff = p.getCoefficient(degree);
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Coefficient of degree " << degree << ": " << coeff << endl;
                cout << "Time taken to get coefficient: " << duration.count() << "
microseconds" << endl;
                break;
            }
            case 11: {
                // Compare equality
                cout << "Input coefficients for the first polynomial:\n";
                Polynomial p1(inputCoefficients());
                cout << "Input coefficients for the second polynomial:\n";
                Polynomial p2(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Compare equality
                bool equal = (p1 == p2);
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Polynomials are " << (equal ? "equal" : "not equal") << endl;
                cout << "Time taken for comparison: " << duration.count() << " microseconds"
<< endl;
                break;
            }
            case 12: {
                // Display polynomial
                cout << "Input coefficients for the polynomial:\n";
                Polynomial p(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Display the polynomial
                cout << p << endl;
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Time taken to display polynomial: " << duration.count() << "
microseconds" << endl;
                break;
            }
            case 13: {
                // Compose two polynomials
                cout << "Input coefficients for the first polynomial:\n";
                Polynomial p1(inputCoefficients())
                cout << "Input coefficients for the second polynomial:\n";
                Polynomial p2(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Compose the polynomials
                Polynomial result = p1.compose(p2);
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Composition result: " << result << endl;
                cout << "Time taken for composition: " << duration.count() << " microseconds"
<< endl;
                break;
            }
            case 14: { // Case for getting the root using Newton's method
                cout << "Input coefficients for the polynomial:\n";
                Polynomial p(inputCoefficients());
                // Start timing
                auto start = chrono::high_resolution_clock::now();
                // Find the root using Newton's method
                double root = p.getRoot();
                // End timing
                auto end = chrono::high_resolution_clock::now();
                auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
                cout << "Root found: " << root << endl;
                cout << "Time taken to find root: " << duration.count() << " microseconds" <<
endl;
                break;
            }
            case 15: { // Update exit case number
                // Exit
                cout << "Exiting...\n";
                break;
            }
            default:
                cout << "Invalid choice, please try again.\n";
        }
    } while (choice != 15); // Update loop condition to match exit case number
    return 0;
}
```

Code 2(Claude)

```
#include "Polynomial.h"
#include <numeric>
#include <stdexcept>
#include <cmath>
#include <iostream>
#include <limits>

Polynomial::Polynomial() {}
Polynomial::Polynomial(const vector<double>& coefficients) : coeffs(coefficients) {
    // Remove leading zeros
    while (!coeffs.empty() && coeffs.back() == 0) {
        coeffs.pop_back();
    }
}
Polynomial::Polynomial(const Polynomial& other) : coeffs(other.coeffs) {}
Polynomial::~Polynomial() {}
Polynomial& Polynomial::operator=(const Polynomial& other) {
    if (this != &other) {
        coeffs = other.coeffs;
    }
    return *this;
}
Polynomial Polynomial::operator+(const Polynomial& other) const {
    vector<double> result_coeffs(max(coeffs.size(), other.coeffs.size()), 0);
    for (size_t i = 0; i < coeffs.size(); ++i) {
        result_coeffs[i] += coeffs[i];
    }
    for (size_t i = 0; i < other.coeffs.size(); ++i) {
        result_coeffs[i] += other.coeffs[i];
    }
    return Polynomial(result_coeffs);

Polynomial Polynomial::operator-(const Polynomial& other) const {
    vector<double> result_coeffs(max(coeffs.size(), other.coeffs.size()), 0);
    for (size_t i = 0; i < coeffs.size(); ++i) {
        result_coeffs[i] += coeffs[i];
    }
    for (size_t i = 0; i < other.coeffs.size(); ++i) {
        result_coeffs[i] -= other.coeffs[i];
    }
    return Polynomial(result_coeffs);
}
Polynomial Polynomial::operator*(const Polynomial& other) const {
    vector<double> result_coeffs(coeffs.size() + other.coeffs.size() - 1, 0);
    for (size_t i = 0; i < coeffs.size(); ++i) {
        for (size_t j = 0; j < other.coeffs.size(); ++j) {
            result_coeffs[i + j] += coeffs[i] * other.coeffs[j];
        }
    }
    return Polynomial(result_coeffs);
}
bool Polynomial::operator==(const Polynomial& other) const {
    return coeffs == other.coeffs;
}
ostream& operator<<(ostream& out, const Polynomial& poly) {
    bool first = true;
    for (int i = poly.coeffs.size() - 1; i >= 0; --i) {
        if (poly.coeffs[i] != 0) {
            if (!first && poly.coeffs[i] > 0) {
                out << " + ";
            } else if (!first && poly.coeffs[i] < 0) {
                out << " - ";
            }
            if (abs(poly.coeffs[i]) != 1 || i == 0) {
                out << abs(poly.coeffs[i]);
            }
            if (i > 0) {
                out << "x";
                if (i > 1) {
                    out << "^" << i;
                }
            }
            first = false;
        }
    }
    if (first) {
        out << "0";
    }
    return out;
```

```
}
int Polynomial::degree() const {
    return coeffs.size() - 1;
}
double Polynomial::evaluate(double x) const {
    return accumulate(coeffs.rbegin(), coeffs.rend(), 0.0,
                      [x](double sum, double coeff) { return sum * x + coeff; });
}
Polynomial Polynomial::compose(const Polynomial& q) const {
    Polynomial result;
    for (int i = coeffs.size() - 1; i >= 0; --i) {
        result = result * q + Polynomial({coeffs[i]});
    }
    return result;
}
Polynomial Polynomial::derivative() const {
    if (coeffs.size() <= 1) {
        return Polynomial({0});
    }
    vector<double> deriv_coeffs(coeffs.size() - 1);
    for (size_t i = 1; i < coeffs.size(); ++i) {
        deriv_coeffs[i - 1] = i * coeffs[i];
    }
    return Polynomial(deriv_coeffs);
}

Polynomial Polynomial::integral() const {
    vector<double> int_coeffs(coeffs.size() + 1);
    int_coeffs[0] = 0;  // Constant of integration
    for (size_t i = 0; i < coeffs.size(); ++i) {
        int_coeffs[i + 1] = coeffs[i] / (i + 1);
    }
    return Polynomial(int_coeffs);

double Polynomial::integral(double x1, double x2) const {
    Polynomial antiderivative = integral();
    return antiderivative.evaluate(x2) - antiderivative.evaluate(x1);
}
vector<double> Polynomial::getRoot(double tolerance, int maxIter) const {
    vector<double> roots;
    Polynomial derivative = this->derivative();
    // Helper function to check if a root is unique (not already in the roots vector)
    auto isUniqueRoot = [&](double root) {
        for (double r : roots) {
            if (abs(r - root) < tolerance) {
                return false;
            }
        }
        return true;
    };
    // Helper function for Newton's method
    auto newtonMethod = [&](double guess) {
        double x = guess;
        for (int i = 0; i < maxIter; ++i) {
            double fx = this->evaluate(x);
            if (abs(fx) < tolerance) {
                return make_pair(true, x);
            }
            double dfx = derivative.evaluate(x);
            if (dfx == 0) {
                return make_pair(false, x);
            }
            x = x - fx / dfx;
        }
        return make_pair(false, x);
    };
    // Try to find roots starting from different initial guesses
    vector<double> initialGuesses = {-10, -1, 0, 1, 10};
    for (double guess : initialGuesses) {
        auto [found, root] = newtonMethod(guess);
        if (found && isUniqueRoot(root)) {
            roots.push_back(root);
        }
    }
    // If no roots found, try random guesses
    if (roots.empty()) {
        srand(time(nullptr));
        for (int i = 0; i < 10; ++i) {
            double randomGuess = (rand() / double(RAND_MAX)) * 20 - 10;  // Random number
between -10 and 10
            auto [found, root] = newtonMethod(randomGuess);
            if (found && isUniqueRoot(root)) {
                roots.push_back(root);
            }
        }
    }
    return roots;
}
void Polynomial::setCoefficients(const vector<double>& coefficients) {
    coeffs = coefficients;
    while (!coeffs.empty() && coeffs.back() == 0) {
        coeffs.pop_back();
    }
}
double Polynomial::getCoefficient(int degree) const {
    if (degree >= 0 && degree < static_cast<int>(coeffs.size())) {
        return coeffs[degree];
    }
    return 0;
}
Polynomial inputPolynomial() {
    vector<double> coeffs;
    double coeff;
    cout << "Enter coefficients (starting from constant term, enter non-number to finish):\n";
    while (cin >> coeff) {
        coeffs.push_back(coeff);
    }
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    return Polynomial(coeffs);
}

int main() {
    int choice;
    Polynomial p1, p2, result;
    while (true) {
        cout << "\nPolynomial Operations Menu:\n";
        cout << "1. Add two polynomials\n";
        cout << "2. Subtract two polynomials\n";
        cout << "3. Multiply two polynomials\n";
        cout << "4. Evaluate polynomial\n";
        cout << "5. Find derivative of a polynomial\n";
        cout << "5. Find integral of a polynomial\n";
        cout << "7. Compute definite integral\n";
        cout << "8. Get polynomial degree\n";
        cout << "9. Set polynomial coefficients\n";
        cout << "10. Get polynomial coefficients\n";
        cout << "11. Compare two polynomials for equality\n";
        cout << "12. Display polynomial\n";
        cout << "13. Compose two polynomials\n";
        cout << "14. Find a root of polynomial\n";
        cout << "15. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice
        if (choice == 15) break;
        switch(choice) {
            case 1: case 2: case 3: case 13: { // Addition, Subtraction, Multiplication,
Composition
                cout << "Enter first polynomial:\n";
                p1 = inputPolynomial();
                cout << "Enter second polynomial:\n";
                p2 = inputPolynomial();
                if (choice == 1) result = p1 + p2;
                else if (choice == 2) result = p1 - p2;
                else if (choice == 3) result = p1 * p2;
                else result = p1.compose(p2);
                cout << "Result: " << result << endl;
                break;
            }
            case 4: { // Evaluation
                cout << "Enter polynomial to evaluate:\n";
                p1 = inputPolynomial();
                double x;
                cout << "Enter x value: ";
                cin >> x;
                cout << "Result: " << p1.evaluate(x) << endl;
                break;
            }
            case 5: case 6: { // Derivative, Integral
                cout << "Enter polynomial:\n";
                p1 = inputPolynomial();
                result = (choice == 5) ? p1.derivative() : p1.integral();
                cout << ((choice == 5) ? "Derivative: " : "Integral: ") << result << endl;
                break;
            }
            case 7: { // Definite Integral
                cout << "Enter polynomial:\n";
                p1 = inputPolynomial();
                double a, b;
                cout << "Enter lower bound a: ";
                cin >> a;
                cout << "Enter upper bound b: ";
                cin >> b;
                cout << "Definite integral from " << a << " to " << b << ": " <<
p1.integral(a, b) << endl;
                break;
            }
            case 8: { // Get degree
                cout << "Enter polynomial:\n";
                p1 = inputPolynomial();
                cout << "Degree: " << p1.degree() << endl;
                break;
            }
            case 9: { // Set coefficients
                p1 = inputPolynomial();
                cout << "Polynomial set: " << p1 << endl;
                break;
            }
            case 10: { // Get coefficients
                cout << "Enter polynomial:\n";
                p1 = inputPolynomial();
                cout << "Coefficients: ";
                for (int i = 0; i <= p1.degree(); ++i) {
                    cout << p1.getCoefficient(i) << " ";
                }
                cout << endl;
                break;
            }
            case 11: { // Compare for equality
                cout << "Enter first polynomial:\n";
                p1 = inputPolynomial();
                cout << "Enter second polynomial:\n";
                p2 = inputPolynomial();
                cout << "Polynomials are " << (p1 == p2 ? "equal" : "not equal") << endl;
                break;
            }
            case 12: { // Display polynomial
                cout << "Enter polynomial:\n";
                p1 = inputPolynomial();
                cout << "Polynomial: " << p1 << endl;
                break;
            }
            case 14: { // Find roots of polynomial
                cout << "Enter polynomial:\n";
                p1 = inputPolynomial();
                vector<double> roots = p1.getRoot();
                if (roots.empty()) {
                    cout << "No roots found.\n";
                } else {
                    cout << "Roots found:\n";
                    for (double root : roots) {
                        cout << root << "\n";
                    }
                }
                break;
            }
            default:
                cout << "Invalid choice. Please try again.\n";
        }
    }
    cout << "Thank you for using the Polynomial Calculator!\n";
    return 0;
}
```

Testing Code