Комплексні числа на C та C++

Артемій Дзюменко

24 листопада 2024 р.

Зміст

1 Описи бібліотек

1.1 `<complex.h>`

`complex.h` — це стандартна бібліотека для комплексних чисел на C, додана в стандарті C99.

– Макроси для комплексних типів: `imaginary`; `complex`.

– double complex z = 1.5 + 2.5 * `I` — декларування комплексного числа.

– double imaginary i = 1.0 * `I` — декларування уявного числа.

– creal(complex double z) — отримання дійсної частини комплексного числа.

– cimag(complex double z) — отримання уявної частини комплексного числа.

– cabs(double complex z) — отримання модуля комплексного числа.

– carg(double complex z) — отримання аргументу комплексного числа.

– conj(double complex z) — отримання комплексного числа, спряженого даному.

– cproj(double complex z) — отримання проєкції комплексного чилса на сферу Рімана (представлення розширеної множини комплексних чисел $\mathbb{C} \cup \{\infty\}$ у вигляді сфери).

– cpow(double complex z1, double complex z2) — обчислення степеня комплексного числа.

– csqrt(double complex z) — обчислення кореня комплексного числа.

– cexp(double complex z) — обчислення експоненти e^z.

– Тригонометричні функції csin, ccos, ctan, casin, cacos, catan.

– Гіперболічні функції csinh, ccosh, ctanh, casinh, cacosh, catanh.

1.2 `std::complex`

`<complex>` — це стандартна бібліотека комлпексних чисел на C++, частина стандартної бібліотеки шаблонів (STL).

- std::complex<T> — шаблонний клас для комплексних чисел.
- std::complex<double> z(1.5, 2.5) — конструктор комплексного числа.
- z.real() — метод отримання дійсної частини комплексного числа.
- z.imag() — метод отримання уявної частини комплексного числа.
- std::abs(z) — отримання модуля комплексного числа.
- std::arg(z) — отримання аргументу комплексного числа.
- std::conj(z) — отримання комплексного числа, спряженого даному.

- std::norm(z) — отримання норми комплексного числа.
- std::polar(std::abs(z), std::arg(z)) — конструктор комплексного числа з модуля та аргументу.
- std::exp(z) — експонента від комплексного числа e^z.
- std::log(z) — натуральний логарифм від комплексного числа.
- std::log10(z) — логарифм за основою 10 від комлпексного числа.
- std::pow(z1, z2) — піднесення комплексного числа до степеня, який може бути як комплексним, так і дійсним.
- std::sqrt(z) — отримання квадратного кореня з комплексного числа.
- Тригонометричні функції std::sin, cos, tan, asin, acos, atan(z)
- Гіперболічні функції std::sinh, cosh, tanh, asinh, acosh, atanh.

1.3 Власна імплементація класу Complex<NumberType>

- Complex<double> z(1.0, 1.0) — конструктор комплексного числа. Є також і стандартний конструктор.
- z.get_real() — отримання дійсної частини комплексного числа.
- z.get_imag() — отримання уявної частини комплексного числа.
- z.conjugate() — отримання комплексного числа, спряженого даному.
- z.magnitude() — отримання модуля комплексного числа.
- z.phase() — отримання аргументу комплексного числа.
- z.pow(int n) — обчислення натурального степеня комплексного числа.
- z.roots[int n] — отримання вектору коренів комплексного числа.

```
Complex z1(1, 1);
auto roots = z1.roots(2);
 // У векторі тепер корені з (1 + i).
Complex z2(-1);
auto roots = z2.roots(2);
std::cout << roots[0];
// Виведеться перший корінь з -1.
```

- sqrt(z), cbrt(z) — обчислення першого кореня 2 і 3 степені з комплексного числа відповідно.
- z.exponential(), z.trigonometric() — методи, що повертають строку з гарним експоненційним та тригонометричним записами комплексного числа.

2 Вирішення рівнянь 3-го та 4-го степенів

2.1 Методи

Тут опишемо метод вирішення степеневих рівнянь на основі описаного класу PolynomialEquation. Ініціалізувати многочлен можна так:

```
PolynomialEquation polynomial(1, 2, 3, 4, 5);
```

Коефіцієнти задаютья від найстаршого до наймолодшого зліва направо. Тобто задати кубічне рівняння $x^3 - 2x^2 + 10x - 5 = 0$; можна так:

```
PolynomialEquation cubic(1, -2, 10, -5);
```

- polynomial.get_equation_type() — допоміжний метод, що повертає значення степеню найстаршого коефіцієнта, від 4 до 0.
- polynomial.solve() — метод, що автоматично визначає степінь рівняння і застосовує відповідний метод. Повертає std::vector з коренями.
- solve_linear() — метод для вирішення лінійних рівнянь. Повертає std::nan, якщо коренів або нескінченність, або жодного.
- solve_quadratic() — метод для вирішення квадратних рівнянь.
- solve_cubic() — метод для вирішення кубічних рівнянь, заснований на методі Кардано. Ключовим у цьому методі є пониження коефіцієнтів.
  Розглянемо кубічне рівняння $ax^3 + bx^2 + cx + d = 0$ Зробимо заміну:

$$x = z - \frac{b}{3a}$$

  В результаті отримаємо спрощене кубічне рівняння вигляду

$$z^3 + pz + q = 0$$

$$p = \frac{3ac - b^2}{3a^2}, \quad q = \frac{2b^3 - 9abc + 27a^2d}{27a^3}$$

  Тепер знайдемо дискримінант Δ:

$$\Delta = \frac{q^2}{4} + \frac{p^3}{27}$$

  Тепер розглянемо випадки.
  Якщо $\Delta > 0$:

$$u = \sqrt[3]{-\frac{q}{2} + \sqrt{\Delta}}, \quad v = \sqrt[3]{-\frac{q}{2} - \sqrt{\Delta}}$$

$$z_1 = u + v$$

$$z_2 = -\frac{u+v}{2} + i\frac{\sqrt{3}}{2}(u - v)$$

$$z_3 = -\frac{u+v}{2} - i\frac{\sqrt{3}}{2}(u - v)$$

Якщо $\Delta = 0$:

$$u = \sqrt[3]{-\frac{q}{2}}$$

$$z_1 = 2u, \quad z_2 = -u, \quad z_3 = -u$$

Якщо $\Delta < 0$:

$$r = 2\sqrt{-\frac{p}{3}}, \quad \theta = \arccos\left(-\frac{3q}{2p}\sqrt{-\frac{3}{p}}\right)$$

$$z_1 = r\cos\left(\frac{\theta}{3}\right)$$

$$z_2 = r\cos\left(\frac{\theta + 2\pi}{3}\right)$$

$$z_3 = r\cos\left(\frac{\theta + 4\pi}{3}\right)$$

Зрештою залишається лише прийняти до уваги заміну, зроблену на початку і отримати корені початкового рівняння:

$$x_i = z_i - \frac{b}{3a}, \quad i = 1, 2, 3$$

- solve_quartic() — метод вирішення рівнянь 4-го степеня, заснований на методі Феррарі. Розглянемо квартичне рівняння $ax^4 + bx^3 + cx^2 + dx + e = 0$ Поділимо на a і отримаємо $x^4 + ax^3 + bx^2 + cx + d = 0$, де

$$a = \frac{b}{a}, \quad b = \frac{c}{a}, \quad c = \frac{d}{a}, \quad d = \frac{e}{a}.$$

У коді також для зручності робиться заміна $a_0 = 0.25a$ and $a_0^2 = a_0 \cdot a_0$. Тепер побудуємо **кубічну резольвенту**, яка допоможе вирішити рівняння. Покладемо її коефіцєнти:

$$p = 3a_0^2 - 0.5b, \quad q = a \cdot a_0^2 - b \cdot a_0 + 0.5c, \quad r = 3a_0^4 - b \cdot a_0^2 + c \cdot a_0 - d.$$

Резольвента матиме вигляд:

$$z^3 + pz^2 + rz + pr - 0.5q^2 = 0$$

Кубічне рівняння гарантовано має хочаб один дійсний корінь. Якщо їх декілька, то треба взяти найбільший.

```
PolynomialEquation cubic_resolvent(1, p, r, p*r
    - 0.5*q*q);

auto resolvent_roots =
   cubic_resolvent.solve_cubic();

ComplexType z = resolvent_roots[0];

for (const auto& root : resolvent_roots) {
```

```
            if (root.get_imag() == 0 && root.get_real() >
                z.get_real()) {
                 z = root;
            }
        }
```

Тепер визначимо нові допоміжні коефіцієнти.

$$\alpha = \sqrt{2p + 2z}$$

$$\beta = \begin{cases} z^2 + r, & \text{якщо } \alpha = 0, \\ -\frac{q}{\alpha} \end{cases}$$

Тепер необхідно вирішити два квадратних рівняння:

$$x^2 + \alpha x + (z + \beta) = 0,$$

$$x^2 - \alpha x + (z - \beta) = 0.$$

4 корені цих рівнянь — це вже майже шукані корені початкового квартичного рівняння, але згадаємо про заміну на початку та відкоригуємо їх:

$$x_i = x_i - \frac{b}{4a}, \quad i = 1, 2, 3, 4.$$

2.2 Різниця в імплементації для стандартних бібліотек

Методи, задіяні на C, ідентичні тим, які описані на C++ зі стандартною бібліотекою та з власним класом. Але оскільки C не має класів, то многочлени імплементовані за допомогою структури з полем статичного масиву коефіцієнтів розміру 5 та вказівників на функції, що є псевдометодами. Нижче наведено приклад використання структури C_PolynomialEquation:

```
         double coefficients[5] = {1.0, 2.0, 3.0, 4.0,
            5.0};
        C_PolynomialEquation* polynom =
           PolynomialEquation_create(coefficients, 5);
        int root_count = 0;
        c_double_complex roots = polynom->solve(polynom,
           &root_count);
        for (int i = 0; i < root_count; ++i){
             printf("Root %d: %lf + %lfi \n", i+1,
                creal(roots[i]), cimag(roots[i])); // Вив
                ід на C
           std::cout << "Root " << i << ": " <<
              roots[i].real() << " + " << roots[i].imag()
              << "i/n"; // Вивід на C++
             }
        free(roots); // !!!
        PolynomialEquation_destroy(polynom); // !!!
```

У файлі-заголовку C_PolynomialEquation.h визначено аліас c_complex_double для типу double complex. Це костиль, без якого компілятор не хотів визнавати код для C. Для створення многочлена необхідна спеціальна функція-ініцілізатор PolynomialEquation_create(), який приймає масив коефіцієнтів та його розмір. Можна передати масив і розміру, меншого від 5, тоді старші коефіцієнти ініціалізатором заповняться нулями. Псевдометод solve() приймає вказівник на об'єкт C_PolynomialEquation та вказівник на змінну типу int з кількістю коренів рівняння. Цей метод **виділяє пам'ять під корені рівняння**, тому масив коренів необхідно звільнити функцією free(). Під об'єкт C_PolynomialEquation також **виділяється пам'ять**, тому потрібно викликати іншу функцію-деструктор, яка звільнить пам'ять: PolynomialEquation_destroy(), передавши в цю функцію вказівник на многочлен. Для використання ж многочленів зі стандартною бібліотекою C++, достатньо ініціалізувати його через `STL_PolynomialEquation`:

```
STL_PolynomialEquation cubic(1, -2, 10, -5);
```

3 Система лінійних рівнянь, задана матрицею-циркулянтом

3.1 Опис задачі

Розглянемо систему лінійних рівнянь, задану наступним чином:

$$\sum_{i=0}^{16383} A_j x_i = b_m, \quad i = 0, ..., 16383 \quad j = m + i \pmod{16384},$$

де $A_k, b_k \in \mathbb{R}$.

Розглянемо процес вирішення СЛР, заданої матрицею $M_{n\times n}$, $rank(M) = n$ та вектором v, $dim(v) = n$. Задамо морфізм, що діагоналізує матрицю:

$$\phi : \mathbf{Mat}_{\mathbb{C}} \to \mathbf{Mat}_{\mathbb{C}}$$

У загальному випадку використовується метод Ґаусса, але його складність $O(n^3)$, для матриці заданого розміру цей метод працюватиме заповільно. Задача полягає в діагоналізації матриці. При діагоналізації ми отримаємо матрицю, значення якої на діагоналі будуть власними числами цієї матриці. У знаходженні власних чисел допоможе **Дискретне перетворення Фур'є**, або **DFT**.

Теорема 1 (Існування базису Фур'є)**.** *Для будь-якого натурального n, матриця дискретного перетворення Фур'є F розміру $n \times n$, що визначається своїми елементами:*

$$F_{jk} = \frac{1}{\sqrt{n}} e^{-2\pi i \frac{jk}{n}}, \quad j, k = 0, 1, \ldots, n-1,$$

*є унітарною. ЇЇ рядки та стовпчики формують базу векторного простору $\mathbb{C}^n$, що називається **базисом Фур'є**.*

Власні числа матриці-циркулянта задаються дискретним перетворенням Фур'є її першого стовпчика, і матриця дискретного перетворення Фур'є діагоналізує матрицю-циркулянт.

3.2 Імплементація перетворення Фур'є

Тобто нашою задачею є діагоналізувати матрицю M, а потім для знаходження розв'язків поділити значення у векторі, до якого застосовані ті самі перетворення, що і до M, $\hat{v}$, на відповідне йому власне число. Після цього отриманий вектор значень треба перевести в початковий базис за допомогою **оберненого перетворення Фур'є**.
Оскільки на вичислення дискретного перетворення вектору v необхідно просумувати n доданків n разів: $v = [v_0, v_1, \ldots, v_{n-1}]$,

$$\hat{v_k} = \sum_{j=0}^{n-1} x_j e^{-2\pi i \frac{jk}{n}}, \quad k = 0, 1, \ldots, n-1,$$

$$\hat{v} = [\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_{n-1}]$$

то складність у загальному випадку — $O(n^2)$. В `CirculantMatrix.cpp` імплементовано **швидке перетворення Фур'є** як метод дискретного перетворення, а саме **алгоритм Кулі-Тьюкі (Cooley-Tukey)**, яке працює за $O(n \log n)$. Розглянемо на прикладі безпосередньо коду:

```
void fast_fourier_transform(std::vector<ComplexType>
    &vec) {
    unsigned long n = vec.size();
    if (n <= 1) return;

    std::vector<ComplexType> even(n / 2);
    std::vector<ComplexType> odd(n / 2);
    for (int i = 0; i < n / 2; ++i) {
        even[i] = vec[i * 2];
        odd[i] = vec[i * 2 + 1];
    }

    fast_fourier_transform(even);
    fast_fourier_transform(odd);

    std::vector<ComplexType> unity_roots(n);
    for (int k = 0; k < n / 2; ++k) {
        double angle = -2 * M_PI * k / n;
        unity_roots[k] = ComplexType(std::cos(angle),
            std::sin(angle));
    }

    for (int j = 0; j < n / 2; ++j) {
        ComplexType t = unity_roots[j] * odd[j];
        vec[j] = even[j] + t;
        vec[j + n / 2] = even[j] - t;
    }
}
```

1. Передаємо в функцію вектор.

2. База рекурсії: якщо вектор одиничний, то перетворення його не змінює.
3. Розділяємо вектор на два, один для парних індексів, інший — для непарних.
4. Рекурсивний крок — передаємо половини вектора у функцію.
5. Обчислюємо корені з одиниці.
6. Збираємо до купи пораховані перетворення:

$$X_k = E_k + W_k \times O_k$$

$$X_{k+n/2} = E_k + W_k \times O_k$$

X_k — k-ий елемент початкового вектора, E_k — k-ий елемент вектора значень з парнихми індексами, W_k — порахований корінь з одиниці, O_k — k-ий елемент вектора значень з непарними індексами.

На найвищому рівні рекурсії отримуємо перетворений початковий вектор.
Наступним кроком, що описаний у методі solve(), є обчислення значень вектору v:

```
 std::vector<ComplexType> fft_X(n);
for (int k = 0; k < n; ++k) {
    if (fft_A[k] != Complex(0.0)) {
        fft_X[k] = fft_B[k] / fft_A[k];
    } else {
        fft_X[k] = Complex(0.0);
    }
}
```

Помітимо, що якщо якийсь елемент початкового вектора стовпчика нульовий, то перетворення працює не дуже добре. Але взагалі такого не має траплятися. Також варто зауважити, що перед вирішенням безпосередньо задачі, треба на основі першого рядку циркулюючої матриці дістати перший стовпчик. Для цього можна залишити перший елемент на місці, а інші розвернути. Для цього є допоміжна функція row_to_column().
Далі, як ми вже зазначили, необхідно перевести порахований вектор значень назад в оригінальний базис. Для цього використаємо обернене перетворення Фур'є:

```
void inverse_fft(std::vector<ComplexType>& vec) {
    unsigned long n = vec.size();

    for (int i = 0; i < n; ++i) {
        vec[i] = vec[i].conjugate();
    }

    fast_fourier_transform(vec);

    for (int i = 0; i < n; ++i) {
        vec[i] = vec[i].conjugate() / n;
    }
```

Ми обертаємо вектор, змінюючи кожний його комплексний елемент на спряжений, застосовуємо перетворення, а потім застосовуємо ті самі зміни, знову спрягаючи його елементи. В результаті отримуємо розв'язки системи лінійних рівнянь.

3.3 Використання класу CirculantMatrix

- Ініціалізувати матрицю можна двома способами. По-перше, безпосередньо передавши в неї два вектори потрібного розміру. По-друге, в якості вводу користувача, є можливість задати матрицю безпосередньо з файлу, передавши в конструктор строку з назвою файлу. Приклад використання:

```
CirculantMatrix cm("inputfile.txt");
auto solutions = cm.solve();
for (auto solution: solutions){
std::cout << solution << ", ";
}
```

- В якості красивого завертання розв‘язків матриці, є метод solve_and_output(), який приймає строку з назвою файлу виводу. У файл записуються розв‘язки; час, витрачений на обрахунки; мінімальна перевірка результатів.

3.4 Різниця в імплементації для стандартних бібліотек

Як і у випадку з многочленами, для використання стандартної бібліотеки комплексних чисел на C++, достатньо використати `STL_CirculantMatrix`.
На C, аналогічно з многочленами, використано структуру C_CirculantMatrix. У ній швидке перетворення Фур‘є виділяє пам‘ять, але потім її коректно звільняє.
Для гарного завертання вирішення СЛР у файл, в окремому файлі `C_CirculantMatrixSolver.cpp` є функція C_solve_and_output(), яка приймає строку з назвою файлу вводу та строку з назвою файлу виводу. Функція сама ініціалізує C_CirculantMatrix, застосовує функцію solve() та виводить результат у файл.

4 Результати

Швидко працюють як і стандартні бібліотеки, так і власний описаний клас. Різниця в їх швидкодії така:

- Поліноми: найповільніше їх вирішує мій клас. На другому місці, з різницею в соту секунди — стандартний клас на C++; найшвидше — код, написаний на C, з відривом у ще три соті секунди.

- СЛР: Найповільніше впорався стандартний клас на C++, на тисячні секунди швидше — мій клас; Найвидше, з відривом у соті секунди, — код на C.

В якості заміру я використав вирішення 1000 різних степеневих рівнянь та СЛР з рангом $2^{14} = 16384$. Мінімальна різниця в часі виконання може бути як через погрішність таймеру, так і через відсутність мікрооптимізацій в тій чи іншій імплементації.
Як плюс своєї імплементації класу комплексних чисел, я б зазначив гарні методи виводу, які беруть до уваги значення уявної частини, що дозволяє не виводити зайві символи і взаємодіяти з об‘єктами цього класу, як зі звичайними числами, отримуючи очікувані виводу. Також косметика, як окремі методи для обертання числа в строку тригонометричної чи експоненційної форми. Як мінус, відсутність безшовної

інтеграції з стандартним шаблонним класом комплексних чисел. У плюси бібліотеки на C хочу додати кращу взаємодію з іншими числовими типами, без необхідності що-небудь кастувати, наприклад, при діленні compex double на int, на відміну від бібліотеки на C++, навіть попри відсутність шаблонів на C.
Мінус імплементації перетворення Фур'є, алгоритм підтримує лише вектори, чий розмір є степінню двійки, коли алгоритм Кулі-Тьюки працює найбільш ефективно. Загалом немає можливості з користувацької сторони задавати розмір матриці, але за бажання в заголовку C_CirculantMatrix.h можна змінити в передпроцесорі #define MATRIX_SIZE на іншу степінь двійки, а в імплементаціях на C++ змінити кілька значень в циклах. Але це точно мінус. Ще одним мінусом є деяка асиметричність імплементацій на C та C++, з одного боку обумовлена різницями в самих мовах, а з іншої — лінню

5 Додаток

5.1 Miscellaneous

- GenerateCiruclantMatrix.hpp та PolynomialGenerator.hpp — створюють випадкові набори чисел для матриці та многочленів відповідно. Приклад використання буде нижче в коді фінального файлу збірки.

- PolynomialSolver.hpp, STL_PolynomialSolver.hpp, C_PolynomialSolver.h — приймають строки з назвами файлів вводу та виводу і обертають вирішення в файлі, записуючи час виконання. Корені рівнянь виводяться як у звичайній, так і в тригонометричній та експоненційних формах. На C грамотно виділяється та очищується пам'ять, запобігаючи витокам пам'яті.

5.2 Приклад збірки

```
#include "../tools/PolynomialGenerator.hpp"
#include "../tools/GenerateCirculantMatrix.hpp"

#include "../src/C++ Proper/PolynomialSolver.hpp"
#include "../src/C++ Proper/CirculantMatrix.hpp"

#include "../src/C++ STL/STL_PolynomialSolver.hpp"
#include "../src/C++ STL/STL_CirculantMatrix.hpp"

// HEADERS FOR C
#include "../src/C/C_PolynomialSolver.hpp"
#include "../src/C/C_CirculantMatrixSolver.hpp"

int main(){

    PolynomialGenerator::initialise();
    PolynomialGenerator::generate_polynomials("../data/polynomials.txt",
       1000);
    GenerateCirculantMatrix("../data/matrix.txt");
```

```
    PolynomialSolver("../data/polynomials.txt",
       "../results/shweicomplex_polynomials_solutions.txt");
    STL_PolynomialSolver("../data/polynomials.txt",
       "../results/stdcomplex_polynomials_solutions.txt");
    C_PolynomialSolver("../data/polynomials.txt",
       "../results/complexh_polynomials_solutions.txt");

    CirculantMatrix cm("../data/matrix.txt");
    cm.solve_and_output("../results/shweicomplex_matrix_solutions.txt");

    STL_CirculantMatrix stdcm("../data/matrix.txt");
    stdcm.solve_and_output("../results/stdcomplex_matrix_solutions.txt");

    C_solve_and_output("../data/matrix.txt",
       "../results/complexh_matrix_solutions.txt");

            }
```

5.3 Makefile

```
    cmake_minimum_required(VERSION 3.26)
    project(ShweiCompleX LANGUAGES C CXX)

    set(CMAKE_CXX_STANDARD 17)

    set(SOURCES_CPP
    src/C/C_PolynomialSolver.cpp
    tools/PolynomialGenerator.cpp
    "src/C++ Proper/PolynomialEquation.cpp"
    "src/C++ Proper/PolynomialSolver.cpp"
    "src/C++ Proper/CirculantMatrix.cpp"
    tools/GenerateCirculantMatrix.cpp
    src/main.cpp
    src/C/C_PolynomialSolver.hpp
    src/C/C_CirculantMatrixSolver.hpp
    src/C/C_CirculantMatrixSolver.cpp
    "src/C++ STL/STL_PolynomialEquation.hpp"
    "src/C++ STL/STL_PolynomialEquation.cpp"
    "src/C++ STL/STL_PolynomialSolver.hpp"
    "src/C++ STL/STL_PolynomialSolver.cpp"
    "src/C++ STL/STL_CirculantMatrix.hpp"
    "src/C++ STL/STL_CirculantMatrix.cpp"
    src/C/C_CirculantMatrixSolver.cpp
    )

    set(SOURCES_C
    src/C/C_PolynomialEquation.c
    src/C/C_PolynomialEquation.h
    src/C/C_CirculantMatrix.h
    src/C/C_CirculantMatrix.c
```

```
    )

    add_executable(ShweiCompleX ${SOURCES_CPP} ${SOURCES_C})

    set_source_files_properties(${SOURCES_C} PROPERTIES LANGUAGE
       C)
```