РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ ИМЕНИ ПАТРИСА ЛУМУМБЫ

Факультет физико-математических и естественных наук
Кафедра «Математического моделирования и искусственного интеллекта»

Компьютерный практикум

Лабораторная работа №1

«Операции с комплексными числами и поиск корней уравнения»

Студент Плугин Никита

Группа НБбд-01-23

Москва

2024

Оглавление

Введение.

В данной работе будут рассмотрены алгоритмы поиска корней уравнения и операции с комплексными числами. Целью данной работы является ознакомление с данными методами.

В первой части работы приведена теория

Во второй части работы приведены алгоритмы данных методов.

В третьей части реализована сама программа.

1. Теоретическая основа.

1.1. Операции с комплексными числами

Комплексные числа представляют собой расширение вещественных чисел и записываются в виде $z = a + bi$, где a и b — вещественные числа, а i — мнимая единица, такая что $i\hat{}2 = -1$. Здесь a называется действительной частью, а b — мнимой частью комплексного числа z.

Основные операции с комплексными числами

1. Сложение:
Для двух комплексных чисел $z_1 = a + bi$ и $z_2 = c + di$ сумма определяется как:
$z_1 + z_2 = (a + c) + (b + d)i$

2. Вычитание:
Для двух комплексных чисел $z_1 = a + bi$ и $z_2 = c + di$ разность определяется как: $z_1 - z_2 = (a - c) + (b - d)i$

3. Умножение:
Для двух комплексных чисел $z_1 = a + bi$ и $z_2 = c + di$ произведение определяется как:
$z_1 \cdot z_2 = (a + bi)(c + di) = ac + adi + bci + bdi^2 = (ac - bd) + (ad + bc)i$

Поскольку $i^2 = -1$, то формула принимает вид:

$z_1 \cdot z_2 = (ac - bd) + (ad + bc)i$

4. Деление:
Для двух комплексных чисел $z_1 = a + bi$ и $z_2 = c + di$ частное определяется как:
$\frac{z_1}{z_2} = \frac{a + bi}{c + di}$

Для упрощения необходимо умножить числитель и знаменатель на сопряжённое комплексное число к знаменателю $c - di$:
$\frac{z_1}{z_2} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{(ac + bd) + (bc - ad)i}{c^2 + d^2}$

Для возведения комплексного числа $z = a + bi$ в степень n удобно использовать показательное представление комплексных чисел, основанное на формуле Эйлера: $z = re^{i\theta}$

где $r = |z| = \sqrt{a^2 + b^2}$ — модуль числа, а $\theta = arg(z)$ — аргумент числа (угол в полярных координатах).

Тогда возведение в степень n определяется как:

$$z^n = \left(re^{i\theta}\right)^n = r^n e^{in\theta}$$

Извлечение корня n-й степени из комплексного числа также удобно проводить в показательной форме. Корни из комплексного числа $z = re^{i\theta}$ определяются по формуле:

$$z^{\frac{1}{n}} = \sqrt[n]{r}\ (e^{i\frac{(\theta + 2k\pi)}{n}}, \qquad k = 0, 1, \dots, n-1$$

Это означает, что n -й корень из комплексного числа имеет n различных значений, расположенных равномерно на комплексной плоскости по углам

1.2. Методы поиска корней уравнения

Метод дихотомии, или метод бисекции, основан на теореме Больцано-Коши: если непрерывная функция $f(x)$ имеет разные знаки на концах отрезка $[a, b]$, то на этом отрезке существует хотя бы один корень уравнения $f(x) = 0$.

Алгоритм:
1. Найти середину отрезка $c = \frac{a+b}{2}$.
2. Вычислить $f(c)$.
3. Если $f(a) \cdot f(c) < 0$, то корень находится на отрезке $[a, c]$; иначе, на отрезке $[c, b]$.
4. Повторять шаги 1-3 до достижения заданной точности.

Метод простых итераций основывается на преобразовании исходного уравнения $f(x) = 0$ в эквивалентное уравнение $x = \phi(x)$.

Алгоритм:
1. Начальное приближение x_0 .
2. Итерационный процесс $x_{n+1} = \phi(x_n)$.
3. Продолжать итерации, пока не будет достигнута заданная точность $|x_{n+1} - x_n| < \epsilon$.

Метод хорд — это численный метод для нахождения корней уравнения, который можно рассматривать как обобщение метода Ньютона, где касательная заменяется хордой.

Алгоритм:
1. Начальные приближения x_0 и x_1 .
2. Итерационный процесс:

$$x_{n+1} = x_n - \frac{\left(f(x_n)(x_n - x_{n-1})\right)}{f(x_n) - f(x_{n-1})}$$

3. Продолжать итерации, пока не будет достигнута заданная точность.

Метод Ньютона, или метод касательных, использует производную функции для нахождения корней уравнения $f(x) = 0$.

Алгоритм:
1. Начальное приближение x_0 .
2. Итерационный процесс: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

3. Продолжать итерации, пока не будет достигнута заданная точность

$$|x_{n+1} - x_n| < \epsilon .$$

2. Алгоритмы.

2.1. Операции с комплексными числами

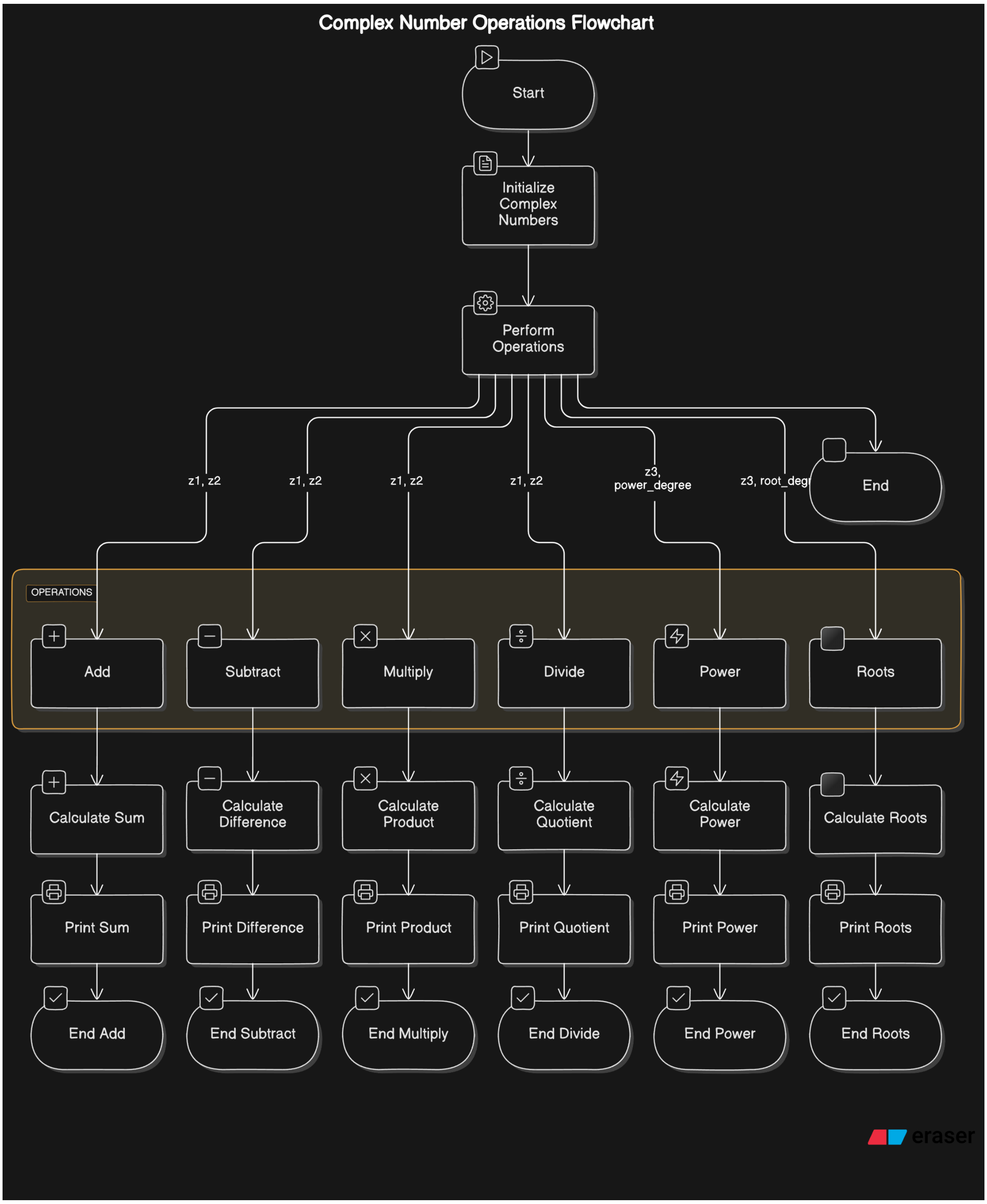

Рисунок 2-1 Блок-схема программы для операций с комплексными числами

2.2. Методы поиска корней уравнения

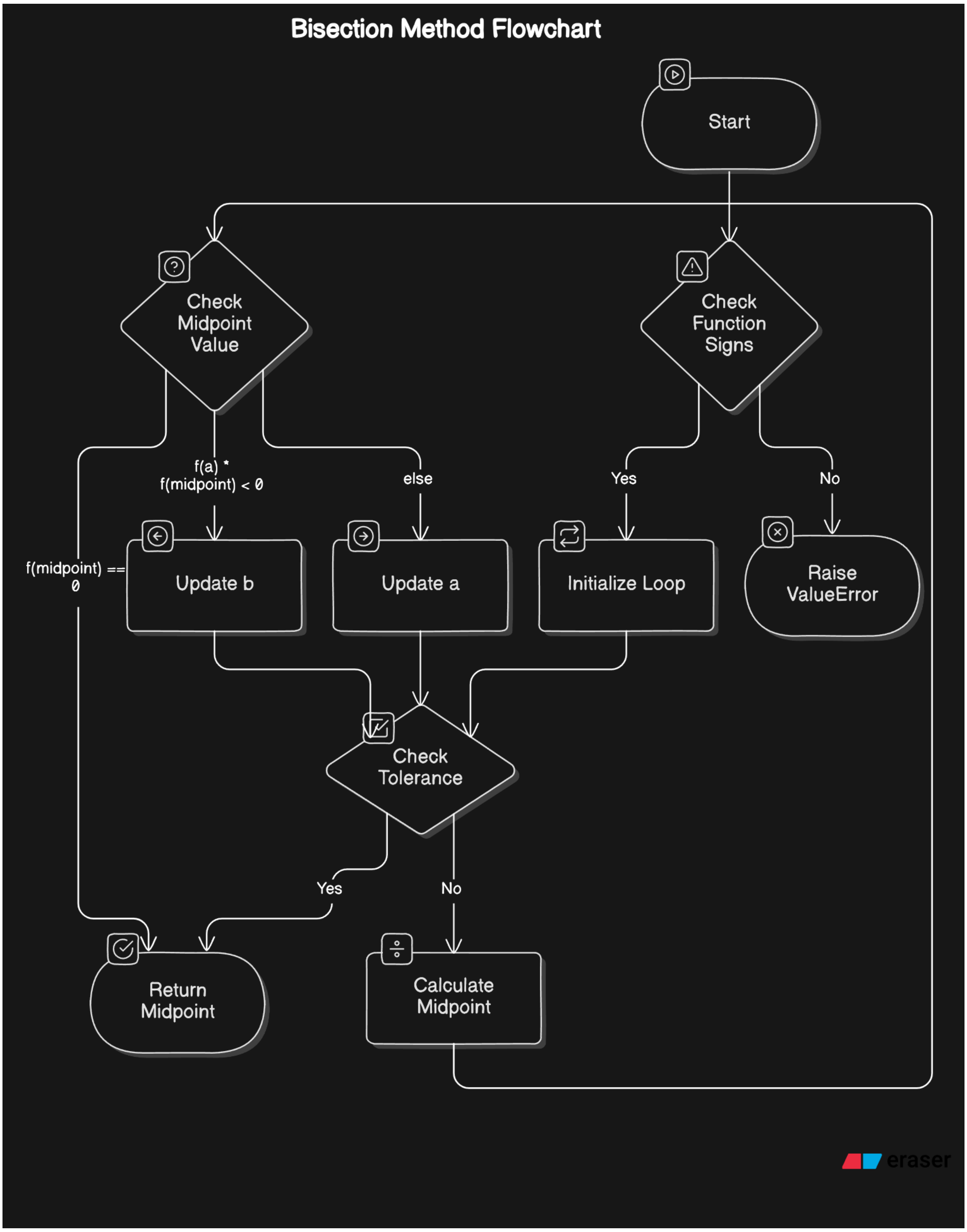

Рисунок 2-2 Блок-схема метода бисекций

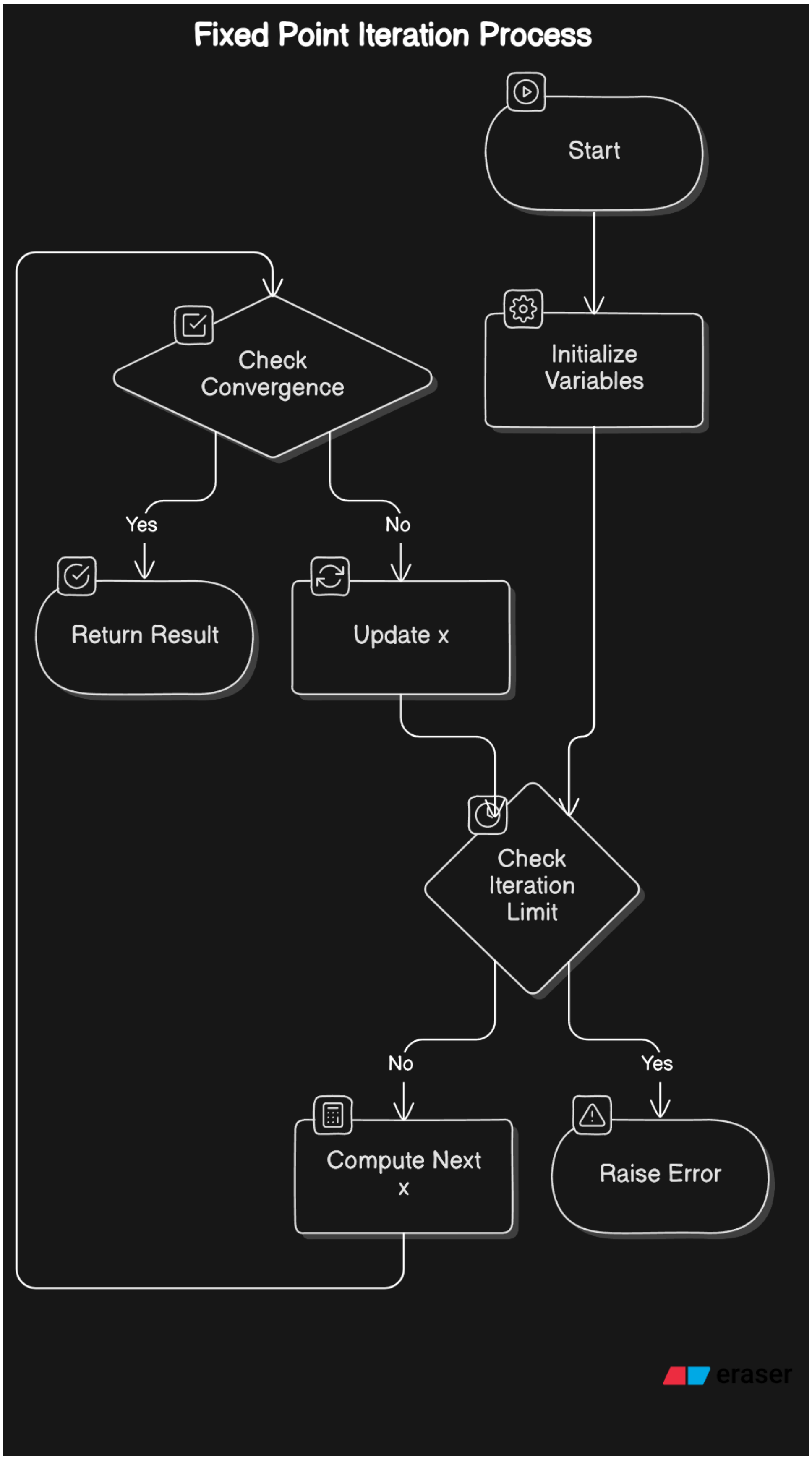

Рисунок 2-3 Блок-схема метода простых итераций

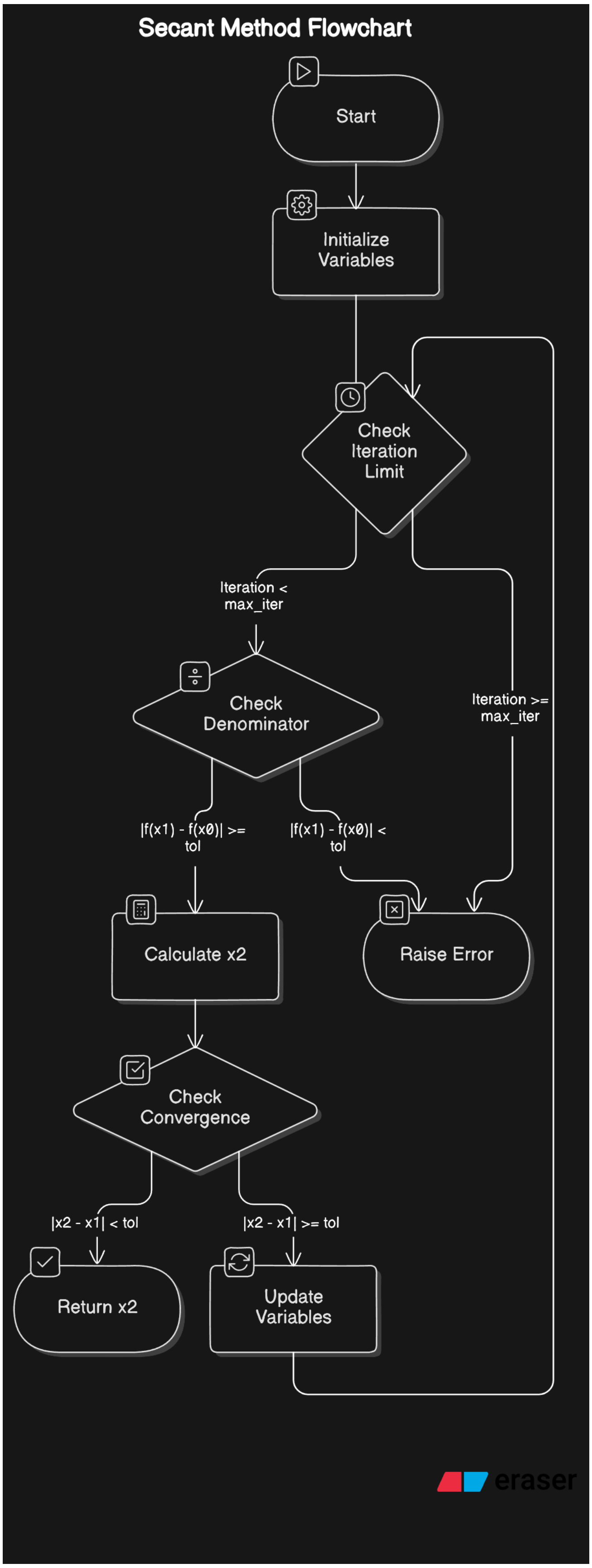

Рисунок 2-4 Блок-схема метода хорд

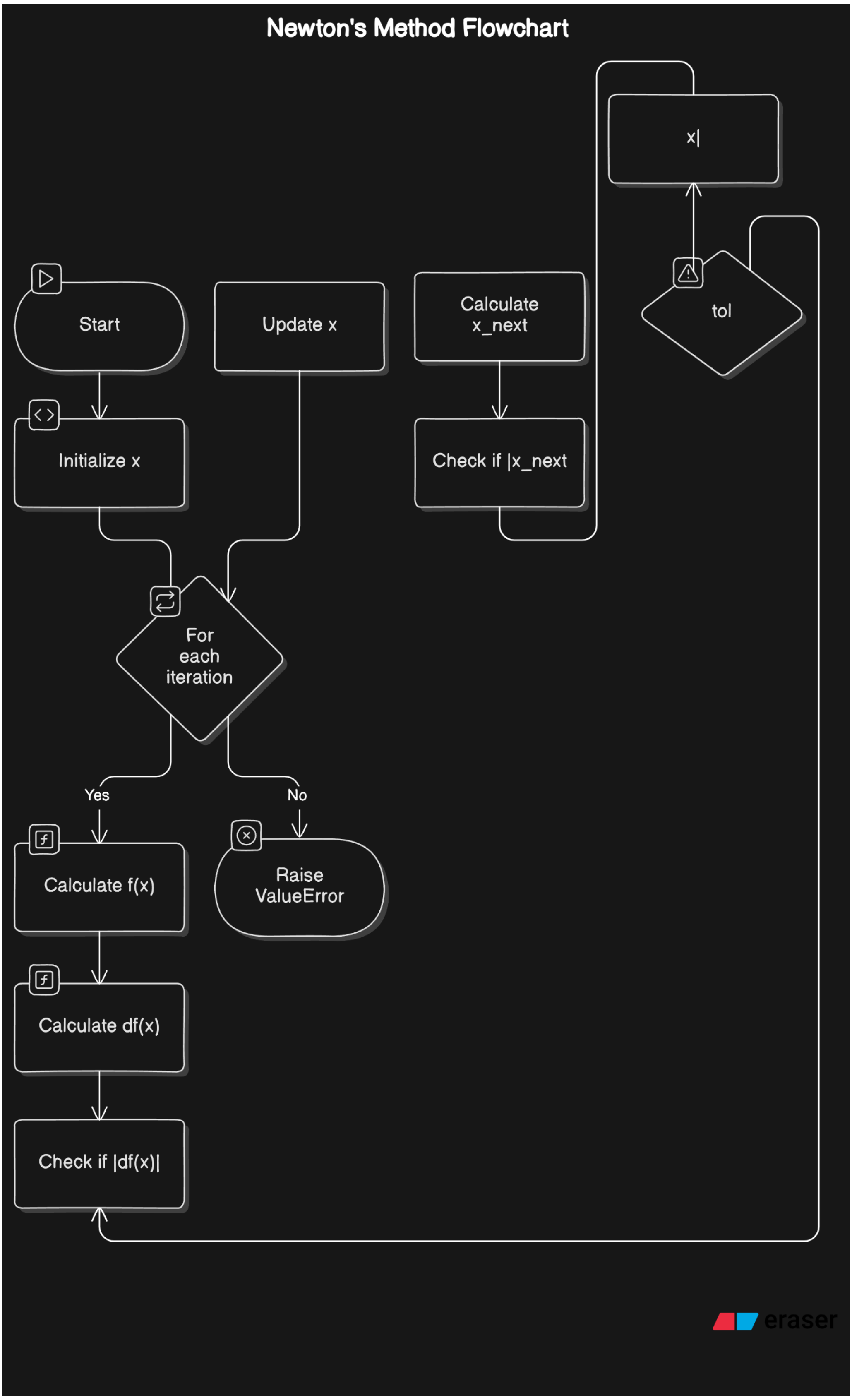

Рисунок 2-5 Блок-схема метода Ньютона

3. Программа реализации алгоритмов

3.1. Операции с комплексными числами

```
#include <iostream>
#include <cmath>

struct Complex {
    double real;
    double imag;
};

Complex add(const Complex& a, const Complex& b) {
    return {a.real + b.real, a.imag + b.imag};
}

Complex subtract(const Complex& a, const Complex& b) {
    return {a.real - b.real, a.imag - b.imag};
}

Complex multiply(const Complex& a, const Complex& b) {
    return {
        a.real * b.real - a.imag * b.imag,
        a.real * b.imag + a.imag * b.real
    };
}

Complex divide(const Complex& a, const Complex& b) {
    double denominator = b.real * b.real + b.imag * b.imag;
    return {
        (a.real * b.real + a.imag * b.imag) / denominator,
        (a.imag * b.real - a.real * b.imag) / denominator
    };
}

Complex power(const Complex& a, int n) {
    double r = std::sqrt(a.real * a.real + a.imag * a.imag);
    double theta = std::atan2(a.imag, a.real);
    double r_pow = std::pow(r, n);
    double angle = n * theta;
    return {r_pow * std::cos(angle), r_pow * std::sin(angle)};
}

void roots(const Complex& a, int n) {
    double r = std::sqrt(a.real * a.real + a.imag * a.imag);
    double theta = std::atan2(a.imag, a.real);
    double r_root = std::pow(r, 1.0 / n);
    for (int k = 0; k < n; ++k) {
        double angle = (theta + 2 * M_PI * k) / n;
        Complex root = {r_root * std::cos(angle), r_root * std::sin(angle)};
        std::cout << "Root " << k + 1 << ": " << root.real << " + " << root.imag << "i" << std::endl;
    }
}

int main() {
    Complex z1 = {-1, 1};
    Complex z2 = {-3, -1};

    Complex sum = add(z1, z2);
    Complex difference = subtract(z1, z2);
    Complex product = multiply(z1, z2);
    Complex quotient = divide(z1, z2);

    std::cout << "Sum: " << sum.real << " + " << sum.imag << "i" << std::endl;
    std::cout << "Difference: " << difference.real << " + " << difference.imag << "i" << std::endl;
    std::cout << "Product: " << product.real << " + " << product.imag << "i" << std::endl;
    std::cout << "Quotient: " << quotient.real << " + " << quotient.imag << "i" << std::endl;

    Complex z3 = {1, 2};
    int power_degree = 4;
    Complex z3_power = power(z3, power_degree);
    std::cout << z3.real << " + " << z3.imag << "i ^ " << power_degree << " = "
              << z3_power.real << " + " << z3_power.imag << "i" << std::endl;

    int root_degree = 3;
    std::cout << "Roots of " << z3.real << " + " << z3.imag << "i ^ (1/" << root_degree << "):" << std::endl;
    roots(z3, root_degree);

    return 0;
}
```

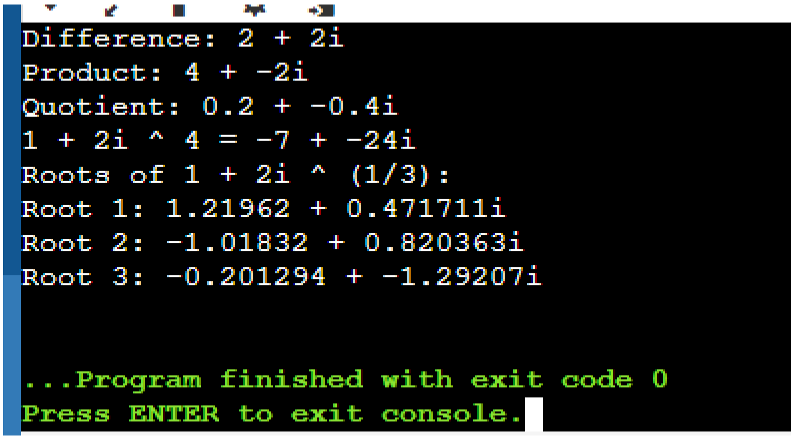

Рисунок 3-1 Результат операций с комплексными числами

3.2. Методы поиска корней уравнения

```
import math

# Функция f(x)
def f(x):
    return math.exp(-x) - math.sqrt(x - 1)

# Производная функции f(x) для метода Ньютона
def df(x):
    return -math.exp(-x) - 1/(2*math.sqrt(x - 1))

# Метод дихотомии (бисекции)
def bisection(a, b, tol=1e-6):
    if f(a) * f(b) >= 0:
        raise ValueError("Function values at the endpoints must have different signs")

    while (b - a) / 2.0 > tol:
        midpoint = (a + b) / 2.0
        if f(midpoint) == 0:
            return midpoint
        elif f(a) * f(midpoint) < 0:
            b = midpoint
        else:
            a = midpoint
    return (a + b) / 2.0

# Метод простых итераций
def fixed_point_iteration(g, x0, tol=1e-6, max_iter=1000):
    x = x0
    for _ in range(max_iter):
        x_next = g(x)
        if abs(x_next - x) < tol:
            return x_next
        x = x_next
    raise ValueError("Fixed point iteration did not converge")

# Метод хорд (секущих)
def secant(x0, x1, tol=1e-6, max_iter=1000):
    for _ in range(max_iter):
        if abs(f(x1) - f(x0)) < tol:
            raise ValueError("Denominator in secant method is too small")
        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
        if abs(x2 - x1) < tol:
            return x2
        x0, x1 = x1, x2
    raise ValueError("Secant method did not converge")

# Метод Ньютона
def newton(x0, tol=1e-6, max_iter=1000):
    x = x0
    for _ in range(max_iter):
        f_x = f(x)
        df_x = df(x)
        if abs(df_x) < tol:
            raise ValueError("Derivative is too small")
        x_next = x - f_x / df_x
        if abs(x_next - x) < tol:
```

```
            return x_next
        x = x_next
    raise ValueError("Newton's method did not converge")

# Основная программа
if __name__ == "__main__":
    # Установить начальные значения и отрезок для методов
    a, b = 1, 2.5  # Отрезок для метода дихотомии
    x0 = 1.1       # Начальное приближение для методов итераций и Ньютона
    x1 = 1.3       # Второе начальное приближение для метода хорд

    try:
        root_bisection = bisection(a, b)
        print(f"Bisection method root: {root_bisection:.6f}")
    except ValueError as e:
        print(e)

    try:

        g = lambda x: 1 + math.exp(-x)
        root_fixed_point = fixed_point_iteration(g, x0)
        print(f"Fixed point iteration root: {root_fixed_point:.6f}")
    except ValueError as e:
        print(e)

    try:
        root_secant = secant(x0, x1)
        print(f"Secant method root: {root_secant:.6f}")
    except ValueError as e:
        print(e)

    try:
        root_newton = newton(x0)
        print(f"Newton's method root: {root_newton:.6f}")
    except ValueError as e:
        print(e)
```

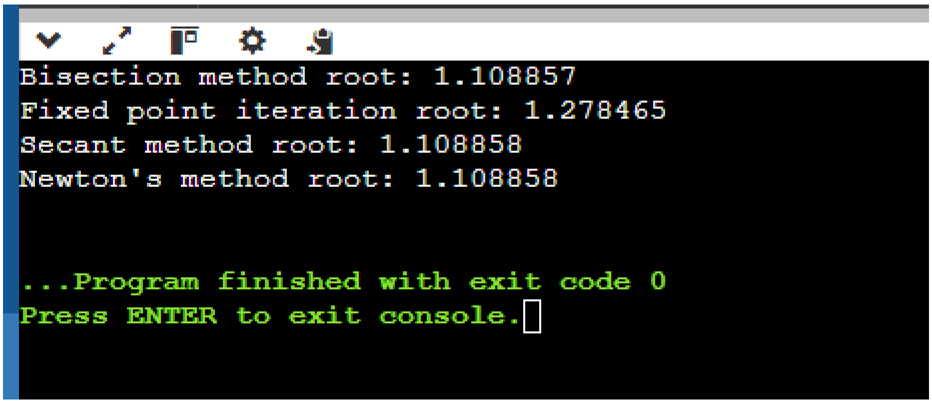

Рисунок 3-2 Поиск корней уравнения

Заключение.

В данной работе был показаны операции с комплексными числами и методы поиска корней уравнения