

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИНСТИТУТ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ СИСТЕМ  
ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК  
КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Допущена к защите  
Заведующей кафедрой ПМИ  
\_\_\_\_\_ Е.В.Разова

## ПРИЛОЖЕНИЯ КОМПЛЕКСНЫХ ЧИСЕЛ К РЕШЕНИЮ ГЕОМЕТРИЧЕСКИХ ЗАДАЧ

Курсовой проект по дисциплине «Проектная и научно-исследовательская  
деятельность»

Выполнил студент группы ПМИб-2301-52-00 \_\_\_\_\_ /Г.Е. Ступников/  
Руководитель к.ф-м.н. доцент кафедры ПМИ \_\_\_\_\_ /И.А. Пушкарев/

Работа защищена с оценкой \_\_\_\_\_ . \_\_\_\_\_ .2022

КИРОВ 2022 г.

# Оглавление

Введение	3
Теория метода комплексных чисел	5
Решение и разбор задач с применением метода	7
Заключение	12
Список литературы	13
Приложения	14
А    Листинг программы . . . . .	14

# Введение

В настоящее время в большом количестве прикладных и научных областей возникает необходимость решения геометрических задач. Основные из них - производство различных деталей и конструкций, моделирование различных объектов и явлений. В данных областях возникает потребность поиска эффективного решения поставленных задач, что подразумевает выборку оптимального метода решения или соотношения между ними. Основные методы решения задач следующие[2]:

1. Аналитический. Состоит в представлении входных и требуемых данных в виде набора переменных и констант и взаимосвязи между ними в виде алгебраических уравнений с последующим их решением.
2. Графический. Состоит в построении рисунка, полноценно отражающего набор необходимых для решения задачи входных данных и взаимосвязей между ними. Решение состоит в последовательном применении известных фактов и теорем, приводящих к получению ответа.
3. Комбинация двух предыдущих. При ручном решении применяется чаще всего.

Метод комплексных чисел является расширением аналитического метода (метод №1). Он позволяет представить геометрические объекты 2-мерной плоскости в виде набора комплексных чисел и равенств, отражающих взаимосвязи между ними.

Данный метод достаточно контринтуитивен и сложен для самостоятельного изучения (особенно непривычно выглядит спиральное подобие как геом. ипостась умножения), при этом он не рассматривается в школах на уровне основной программы[4],[1, стр.6].

Проблема состоит в том, что для данного метода отсутствуют материалы для внедрения в среду самостоятельного и школьного обучения, включающие программы, облегчающие изучение метода.

Целью данной работы является изучение метода комплексных чисел при решении

геометрических задач, реализация программной верификации решения выбранных задач.

Для достижения цели необходимо выполнить следующие задачи:

1. Изучить имеющиеся способы применения алгебры комплексных чисел при решении геометрических задач.
2. Выбрать задачи, на которых будет рассматриваться практическое применение метода.
3. Решение задач с применением метода комплексных чисел
4. Реализация программной верификации решения задач с применением метода.

# Теория метода комплексных чисел

Комплексное число  $z$  – число вида  $x + iy$ , где  $x, y \in \mathbf{R}, i = \sqrt{-1}, z \in \mathbf{C}, \mathbf{C}$  – поле комплексных чисел. У числа  $z$  можно выделить действительную  $x = \operatorname{Re}(z)$  и мнимую  $y = \operatorname{Im}(z)$  части.

На плоскости зададим прямоугольную декартову систему координат  $Oxy$  и отображение  $f : M(x; y) \leftrightarrow z = x + iy$ , где  $M \in \mathbf{P}$  – точка плоскости с координатами  $x, y \in \mathbf{R}, \mathbf{P}$  – множество точек евклидовой плоскости. Комплексное число  $z$  называют комплексной координатой соответствующей точки  $M$  и пишут  $M(z)$ . Отображение  $f$  биективно. Метод комплексных чисел основан на данном факте. Таким образом, свойства и операции комплексных чисел можно перенести на прямоугольную декартову систему координат евклидовой плоскости.

Для примера рассмотрим некоторые из свойств:

1. Модуль числа  $z = |z| = \sqrt{x_0^2 + y_0^2} = r$  – расстояние между точкой  $O$  и  $M$  (рис. 1).
2. Если  $\angle \varphi$  – ориентированный, образованный  $\overrightarrow{OM}$  с осью  $Ox$ , то  $x_0 = r \cos \varphi, y_0 = r \sin \varphi$  (из определения функций). Тогда  $z_0 = r(\cos \varphi + i \sin \varphi)$ . Такое представление комплексного числа называют тригонометрическим.
3.  $\arg z = \angle \varphi$
4. Если на плоскости комплексных чисел заданы векторы  $\overrightarrow{OA}$  и  $\overrightarrow{OB}$ , где  $O$  – начало координат,

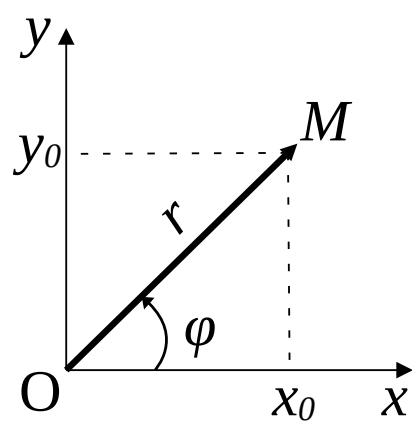


Рис. 1: Изображение числа  $z$  на плоскости

# Решение и разбор задач с применением метода

## Задача 1

**Постановка задачи:** Доказать, что если некоторая прямая пересекает прямые, содержащие стороны  $BC$ ,  $CA$ ,  $AB$  треугольника  $ABC$ , в точках  $A_1$ ,  $B_1$ ,  $C_1$  соответственно, то середины отрезков  $AA_1$ ,  $BB_1$ ,  $CC_1$  коллинеарны.

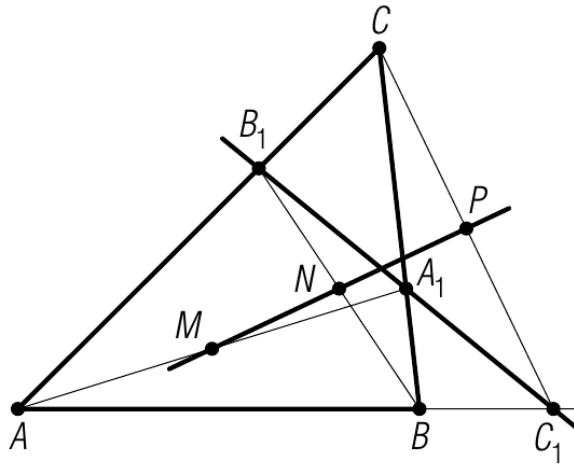


Рис. 2: Иллюстрация к задаче

**Решение задачи:** Условие коллинеарности троек точек  $A, B_1, C$ ;  $C, A_1, B$ ;  $B, C_1, A$ ;  $A_1, B_1, C_1$ :

$$\begin{cases} a(\bar{b}_1 - \bar{c}) + b_1(\bar{c} - \bar{a}) + c(\bar{a} - \bar{b}_1) = 0 \\ b(\bar{c}_1 - \bar{a}) + c_1(\bar{a} - \bar{b}) + a(\bar{b} - \bar{c}_1) = 0 \\ c(\bar{a}_1 - \bar{b}) + a_1(\bar{b} - \bar{c}) + b(\bar{c} - \bar{a}_1) = 0 \\ a_1(\bar{b}_1 - \bar{a}_1) + b_1(\bar{a}_1 - \bar{a}_1) + a_1(\bar{a}_1 - \bar{b}_1) = 0 \end{cases} \quad (1)$$

Если  $M, N, P$  – середины отрезков  $AA_1, BB_1, CC_1$ , то предстоит показать, что

$$m(\bar{n} - \bar{p}) + n(\bar{p} - \bar{m}) + p(\bar{m} - \bar{n}) = 0, \quad (2)$$

Так как  $m = \frac{1}{2}(a + a_1)$ ,  $n = \frac{1}{2}(b + b_1)$ ,  $p = \frac{1}{2}(c + c_1)$ , то доказываемое равенство (2) эквивалентно такому:

$$(a + a_1)(\bar{b} + \bar{b}_1 - \bar{c} - \bar{c}_1) + (b + b_1)(\bar{c} + \bar{c}_1 - \bar{a} - \bar{a}_1) + (c + c_1)(\bar{a} + \bar{a}_1 - \bar{b} - \bar{b}_1) = 0,$$

или, после перемножения,

$$\begin{aligned} & a(\bar{b}_1 - \bar{c}) + a(\bar{b} - \bar{c}_1) + a_1(\bar{b}_1 - \bar{c}_1) + a_1(\bar{b} - \bar{c}) + b(\bar{c}_1 - \bar{a}) + b(\bar{c} - \bar{a}_1) + \\ & + b_1(\bar{c}_1 - \bar{a}_1) + b_1(\bar{c} - \bar{a}) + c(\bar{a}_1 - \bar{b}) + c(\bar{a} - \bar{b}_1) + c_1(\bar{a}_1 - \bar{b}_1) + c_1(\bar{a} - \bar{b}) = 0. \end{aligned} \quad (3)$$

Теперь легко видеть, что (3) получается при почленном сложении равенств (1)

**Алгоритм программного решения частного случая задачи:** На вход программы передаются координаты свободных точек комплексной плоскости, в данном примере это координаты точек  $A, B, C, A_1, B_1$ . Если  $A_1, B_1$  лежат на треугольнике, то по данным входным данным строится прямая, соответствующая условиям задачи. Далее производится проверка того, что середины отрезков  $AA_1, BB_1, CC_1$  коллинеарны. Если условие выполняется, то задача считается решенной для данных входных данных и на экран выводятся координаты точек  $M, N, P$ , а также координаты всех остальных. Блок-схема алгоритма приведена на Рис. 3.



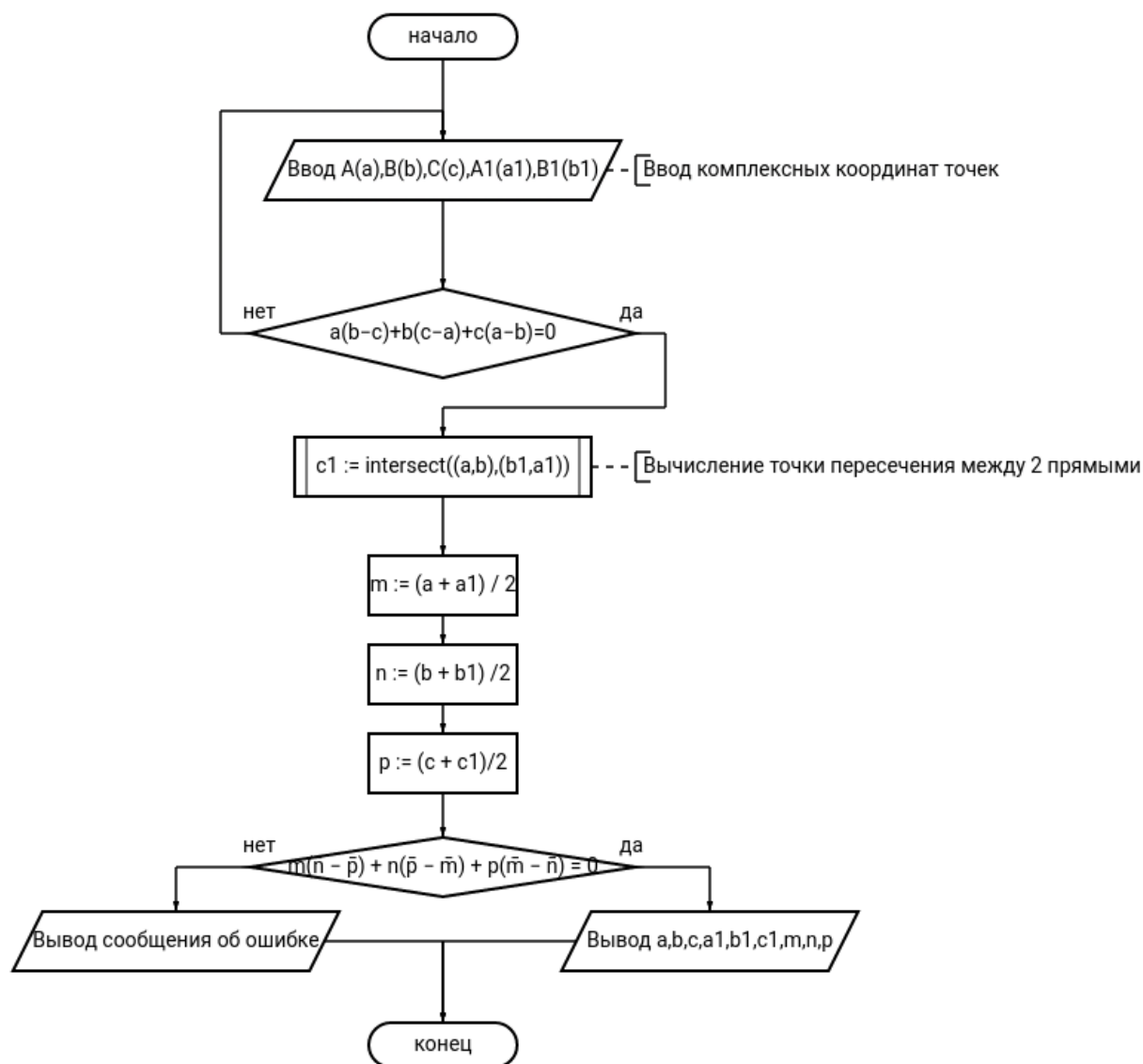


Рис. 3: Блок-схема алгоритма программы

**Программная реализация задачи:** Решение задачи написано на языке C++ в как часть программы для решения задач из данной работы. Реализация алгоритма программы предоставлена в функции task1 (файл task1.cpp):

```

inline void task1()
{
    // TODO test on cw-main data (or almost same data)
    const int numbersCount = 9;
    ComplexNumber numbers[numbersCount];
    const std::string labels[9] = { "A", "B", "C", "A1", "B1", "C1", "M", "N", "P" };
    // Reference for readability
    ComplexNumber &a = numbers[0],
  
```

```

        &b = numbers[1],
        &c = numbers[2],
        &a1 = numbers[3],
        &b1 = numbers[4],
        &c1 = numbers[5],
        &m = numbers[6],
        &n = numbers[7],
        &p = numbers[8];

Task1_ReadNumbersFromUser(numbers, labels);

std::pair<ComplexNumber, ComplexNumber> pairs[2] { { a, b }, { b1, a1 } };

c1 = intersect(pairs[0], pairs[1]);
m = ComplexNumber::middle(a, a1);
n = ComplexNumber::middle(b, b1);
p = ComplexNumber::middle(c, c1);

if (ComplexNumber::isOnSameLine(m, n, p)) {
    std::cout << "Computed coordinates:\n";
    for (size_t i = 0; i < numbersCount; i++) {
        std::cout << " " + labels[i] + ": " << numbers[i] << "\n";
    }
}
}

```

Листинг 1: Функция task1

**Демонстрация работы:** Здесь будут скриншоты работы.

## Задача 2

**Постановка задачи:** Показать, что ортогональные проекции точки, лежащей на описанной около треугольника окружности, на прямые, содержащие его стороны, коллинеарны.

**Решение задачи:** За центр окружности примем точку  $O$

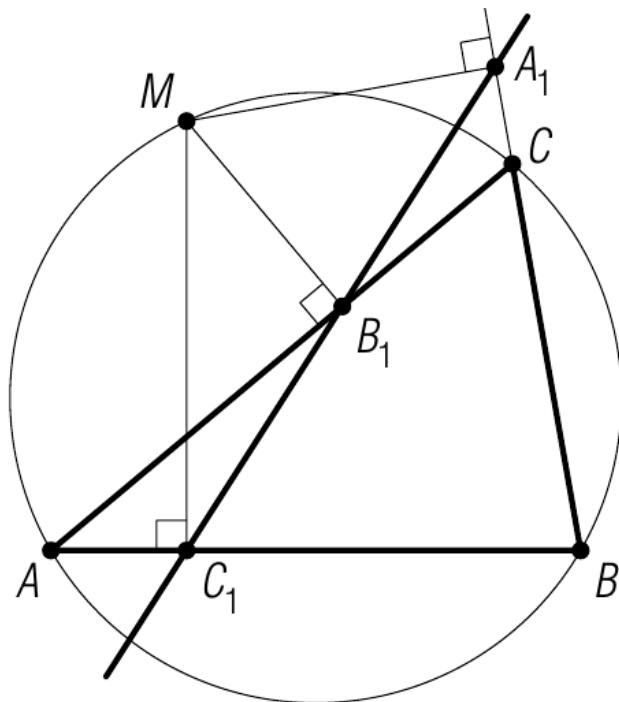


Рис. 4: Иллюстрация к задаче

### Задача 3

## Заключение

В ходе выполнения работы изложены основы метода комплексных чисел, было проиллюстрировано его применение при решении 3 задач. Каждая задача имеет решение на языке C++.

Таким образом, все поставленные задачи были успешно выполнены, цель достигнута.

## Список литературы

- [1] Алгебра комплексных чисел в геометрических задачах: Книга для учащихся математических классов школ, учителей и студентов педагогических вузов. – М.: МЦНМО, 2004. – 160 с.
- [2] Обучение методам решения геометрических задач  
<https://cyberleninka.ru/article/n/obuchenie-metodam-resheniya-geometricheskih-zadach/viewer>
- [3] Бронштейн И. Н., Семендяев К. А. Справочник по математике для инженеров и учащихся втузов. – 13-е изд., исправленное. – М.: Наука, Гл. ред. физ.-мат. лит., 1986. – 544 с.
- [4] Жмурова И. Ю. Изучение комплексных чисел в общеобразовательной школе / И. Ю. Жмурова, С. В. Баринова. // Молодой ученый. – 2020. – № 5 (295). – С. 312-314. – URL: <https://moluch.ru/archive/295/67123/> (дата обращения: 17.05.2022).

# Приложения

## Приложение А. Листинг программы

main.cpp:

```
#include "ComplexNumber.hpp"
#include "functions.hpp"

#include "task1.cpp"
#include "task2.cpp"
#include "task3.cpp"

#include <iostream>

int main(int argc, char const* argv[])
{
    bool noData = false;
    if (argc == 1) {
        std::cout << "Enter program number to launch: ";
        argc++;
        noData = true;
    }
    for (int i = 1; i < argc; i++) {
        int choice;
        if (noData) {
            std::cin >> choice;
            std::cin.ignore(32767, '\n');
        } else {
            choice = std::stoi(argv[i]);
        }

        switch (choice) {
        case 1:
            std::cout << "Task #1\n";
```

```

        task1();
        std::cout << "-----\n";
        break;
    case 2:
        std::cout << "Task #2\n";
        task2();
        std::cout << "-----\n";
        break;
    case 3:
        std::cout << "Task #3\n";
        task3();
        std::cout << "-----\n";
        break;
    default:
        std::cout << "Entered program number is incorrect, retry.\n";
        break;
    }
}
}

```

task1.cpp:

```

#include "ComplexNumber.hpp"
#include "functions.hpp"

inline void task1()
{
    // TODO test on cw-main data (or almost same data)
    const int numbersCount = 9;
    ComplexNumber numbers[numbersCount];
    const std::string labels[9] = { "A", "B", "C", "A1", "B1", "C1", "M", "N", "P" };
    // Reference for readability
    ComplexNumber &a = numbers[0],
                &b = numbers[1],
                &c = numbers[2],
                &a1 = numbers[3],
                &b1 = numbers[4],
                &c1 = numbers[5],
                &m = numbers[6],
                &n = numbers[7],
                &p = numbers[8];
}

```

```

Task1_ReadNumbersFromUser(numbers, labels);

std::pair<ComplexNumber, ComplexNumber> pairs[2] { { a, b }, { b1, a1 } };

c1 = intersect(pairs[0], pairs[1]);
m = ComplexNumber::middle(a, a1);
n = ComplexNumber::middle(b, b1);
p = ComplexNumber::middle(c, c1);

if (ComplexNumber::isOnSameLine(m, n, p)) {
    std::cout << "Computed coordinates:\n";
    for (size_t i = 0; i < numbersCount; i++) {
        std::cout << " " + labels[i] + ": " << numbers[i] << "\n";
    }
}
}

```

task2.cpp:

```

inline void task2() { }

```

task3.cpp:

```

inline void task3() { }

```

functions.hpp:

```

#ifndef FUNCTIONS_LIB
#define FUNCTIONS_LIB

#include "ComplexNumber.hpp"
#include <limits>
#include <cmath>
#include <tuple>

using lineSegment_t = std::pair<ComplexNumber, ComplexNumber>;

/**
 * @brief Intersect of 2 line segments
 * @return ComplexNumber, intersection point of lines

```



```

*/
ComplexNumber intersect(lineSegment_t first, lineSegment_t second);

bool isPointBelongsSegment(lineSegment_t segment, ComplexNumber point);

void Task1_ReadNumbersFromUser(ComplexNumber arr[5], const std::string labels[5]);

template <class T>
typename std::enable_if<!std::numeric_limits<T>::is_integer, bool>::type
    almost_equal(T x, T y, int ulp)
{
    // the machine epsilon has to be scaled to the magnitude of the values used
    // and multiplied by the desired precision in ULPs (units in the last place)
    return std::fabs(x - y) <= std::numeric_limits<T>::epsilon() * std::fabs(x + y)
        * ulp
        // unless the result is subnormal
        || std::fabs(x - y) < std::numeric_limits<T>::min();
}

#endif // FUNCTIONS_LIB

```

functions.cpp:

```

#include "functions.hpp"
#include "Line.hpp"

#include <cmath>

ComplexNumber intersect(lineSegment_t first, lineSegment_t second)
{
    Line firstL(first.first, first.second), secondL(second.first, second.second);

    double x = (secondL.B() - firstL.B()) / (firstL.K() - secondL.K());
    double tmp = firstL.y(x);
    double y = std::isinf(tmp) ? secondL.y(x) : tmp;

    return ComplexNumber(x, y);
}

bool isPointBelongsSegment(lineSegment_t segment, ComplexNumber point)
{
    ComplexNumber &a = segment.first, &b = segment.second;

```

```

Line line(a, b);

double imMax = std::max(a.Im(), b.Im()), imMin = std::min(a.Im(), b.Im());
double reMax = std::max(a.Re(), b.Re()), reMin = std::min(a.Re(), b.Re());
bool isPointInBounds
    = (imMin <= point.Im() && point.Im() <= imMax)
    && (reMin <= point.Re() && point.Re() <= reMax);

return line.isBelongs(point) && isPointInBounds;
}

void Task1_ReadNumbersFromUser(ComplexNumber arr[5], const std::string labels[5])
{
    const size_t labelsCount = 5;

    ComplexNumber &a = arr[0],
                  &b = arr[1],
                  &c = arr[2],
                  &a1 = arr[3],
                  &b1 = arr[4];

    bool isTriangle = false, isValidA1 = false, isValidB1 = false;
    while (!(isTriangle && isValidA1 && isValidB1)) {
        std::cout << "Enter coordinates of a,b,c,a1,b1 points:\n";
        if (std::cin.fail()) {
            if (std::cin.eof()) {
                std::cout << "User input was canceled. Aborting...\n";
                return;
            }
            std::cin.ignore();
            std::cin.clear();
        }
        for (size_t i = 0; i < labelsCount; i++) {
            std::cout << " " << labels[i] << ": ";
            std::cin >> arr[i];
        }
        isTriangle = a * (b - c) + b * (c - a) + c * (a - b)
            == ComplexNumber::getZero();

        isValidA1 = isPointBelongsSegment({ b, c }, a1);
        isValidB1 = isPointBelongsSegment({ a, c }, b1);
    }
}

```

```

        if (!isTriangle)
            std::cerr << "Incorrect a,b,c. Must be points of the triangle ABC\n";
        if (!isValidA1)
            std::cerr << "The a1 is incorrect. Must belong to segment of line BC\n";
        if (!isValidB1)
            std::cerr << "The b1 is incorrect. Must belong to segment of line AC\n";
    }
}

```

ComplexNumber.hpp:

```

#ifndef COMPLEXN_LIB
#define COMPLEXN_LIB

#include "Point.hpp"
#include <iostream>

// TODO write comments

class ComplexNumber {
private:
    double _imaginary;
    double _real;
    friend std::ostream& operator<<(std::ostream& out, const ComplexNumber& number);
    friend std::istream& operator>>(std::istream& in, ComplexNumber& number);

public:
    ComplexNumber(double real = 0, double imaginary = 0);
    ComplexNumber(const ComplexNumber& source);

    ComplexNumber& operator=(const ComplexNumber& b);

    ComplexNumber operator+(const ComplexNumber& b) const;
    ComplexNumber operator-(const ComplexNumber& b) const;
    ComplexNumber operator*(const ComplexNumber& b) const;
    // TODO ComplexNumber operator/(const ComplexNumber& b) const;

    operator Point() const { return Point(Re(), Im()); }

    bool operator==(const ComplexNumber& b) const;

```

```

bool operator!=(const ComplexNumber& b) const;

static bool isCollinear(const ComplexNumber& a, const ComplexNumber& b);
static bool isOnSameLine(const ComplexNumber& a, const ComplexNumber& b, const
    ComplexNumber& c);

static ComplexNumber middle(const ComplexNumber& a, const ComplexNumber& b);

const double& Re() const { return _real; }
const double& Im() const { return _imaginary; }

static ComplexNumber getZero()
{
    return ComplexNumber(0, 0);
}
};

#endif // COMPLEXN_LIB

ComplexNumber.cpp:

#include "ComplexNumber.hpp"
#include "Line.hpp"

static const ComplexNumber zero = ComplexNumber(0, 0);

ComplexNumber::ComplexNumber(double real, double imaginary) : _imaginary(imaginary),
    _real(real)
{
}

ComplexNumber::ComplexNumber(const ComplexNumber& source) : _imaginary(source.Im()),
    _real(source.Re())
{
}

std::ostream& operator<<(std::ostream& out, const ComplexNumber& number)
{
    out << number._real << " + " << number._imaginary << "i";
    return out;
}

```

```

std::istream& operator>>(std::istream& in, ComplexNumber& number)
{
    in >> number._real >> number._imaginary;
    return in;
}

ComplexNumber& ComplexNumber::operator=(const ComplexNumber& b)
{
    this->_real = b.Re();
    this->_imaginary = b.Im();
    return *this;
}

ComplexNumber ComplexNumber::operator+(const ComplexNumber& b) const
{
    return ComplexNumber(this->Re() + b.Re(), this->Im() + b.Im());
}

ComplexNumber ComplexNumber::operator-(const ComplexNumber& b) const
{
    return ComplexNumber(this->Re() - b.Re(), this->Im() - b.Im());
}

ComplexNumber ComplexNumber::operator*(const ComplexNumber& b) const
{
    return ComplexNumber(this->Re() * b.Re(), this->Im() * b.Im());
}

bool ComplexNumber::operator==(const ComplexNumber& b) const
{
    return this->Re() == b.Re() && this->Im() == b.Im();
}

bool ComplexNumber::operator!=(const ComplexNumber& b) const
{
    return !(*this == b);
}

bool ComplexNumber::isCollinear(const ComplexNumber& a, const ComplexNumber& b)
{
    return true;
}

bool ComplexNumber::isOnSameLine(const ComplexNumber& a, const ComplexNumber& b,

```

```

    const ComplexNumber& c)
{
    return Line(a, b).isBelongs(c);
}

ComplexNumber ComplexNumber::middle(const ComplexNumber& a, const ComplexNumber& b)
{
    return ComplexNumber((a.Re() + b.Re()) / 2, (a.Im() + b.Im()) / 2);
}

```

Line.hpp:

```

#ifndef LINE_LIB
#define LINE_LIB

#include "Point.hpp"
#include <tuple>

enum class LineType {
    CONST_Y, // y = const
    CONST_X, // x = const
    NORMAL // y = kx + b
};

/**
 * @brief Represents line by equation 'y = kx + b'
 */
class Line {
private:
    double _k, _b;

    /**
     * @brief Defines y or x constant value if _type is CONST_X or CONST_Y. This is
     reference for memory optimization.
     */
    double &_x = _b, &_y = _k;
    LineType _type;

    static double getKFromPoints(const Point& a, const Point& b);
    static double getBFromPoints(const Point& a, const Point& b);

```

```

public:
    Line(double k, double b) : _k(k), _b(b) { }
    Line(std::pair<Point, Point> pair);
    Line(Point first, Point second) : Line(std::make_pair(first, second)) { }

    LineType getType() const { return _type;}

    double y(double x) const;
    double x(double y) const;

    const double& K() const { return _k; }
    const double& B() const { return _b; }

    bool isInX(double x) const;
    bool isInY(double y) const;

    bool isBelongs(Point point);
};

#endif // LINE_LIB

```

Line.cpp:

```

#include "Line.hpp"
#include "functions.hpp"
#include <cmath>
#include <limits>
#include <stdexcept>

class LineEquation {
private:
    Point _pointA, _pointB;
    double _k, _b, _x, _y;
    double xDiff, yDiff;
    LineType type;

    void initByLineType();

public:
    LineEquation() {};
    LineEquation(const Point& a, const Point& b);

```

```

double K() { return _k; }
double B() { return _b; }

double xConst() { return _x; }
double yConst() { return _y; }

LineType getType() { return type; }
};

Line::Line(std::pair<Point, Point> pair)
{
    LineEquation equation = LineEquation(pair.first, pair.second);

    _k = equation.K();
    _b = equation.B();
    _type = equation.getType();
    switch (_type) {
    case LineType::CONST_X:
        _x = equation.xConst();
        break;
    case LineType::CONST_Y:
        _y = equation.yConst();
        break;
    default:
        break;
    }
}

double Line::y(double x) const
{
    switch (_type) {
    case LineType::CONST_X:
        return (_x == x)
            ? std::numeric_limits<double>::infinity()
            : 0;
    case LineType::CONST_Y:
        return _y;
    case LineType::NORMAL:
        return _k * x + _b;
    default:
        return 0;
    }
}

```



```

    }
}

double Line::x(double y) const
{
    switch (_type) {
    case LineType::CONST_X:
        return _x;
    case LineType::CONST_Y:
        return (_y == y)
            ? std::numeric_limits<double>::infinity()
            : 0;
    case LineType::NORMAL:
        return (y - _b) / _k;
    default:
        return 0;
    }
}

```

```

bool Line::isInX(double x) const
{
    switch (_type) {
    case LineType::CONST_X:
        return _x == x;
    case LineType::CONST_Y:
    case LineType::NORMAL:
        return true;
    default:
        return false;
    }
}

```

```

bool Line::isInY(double y) const
{
    switch (_type) {
    case LineType::CONST_X:
    case LineType::NORMAL:
        return true;
    case LineType::CONST_Y:
        return _y == y;
    default:
        return 0;
    }
}

```

```

    }
}

bool Line::isBelongs(Point point)
{
    switch (_type) {
    case LineType::CONST_X:
        return _x == point.X();
    case LineType::CONST_Y:
        return _y == point.Y();
    case LineType::NORMAL:
        return almost_equal(y(point.X()), point.Y(), 2);
    default:
        return false;
    }
}

void LineEquation::initByLineType()
{
    switch (type) {
    case LineType::CONST_X:
        // Y may be any, x = const
        _k = 0;
        _b = std::numeric_limits<double>::infinity();
        _x = _pointB.X();
        break;
    case LineType::CONST_Y:
        // X may be any, y = const
        _k = 0;
        _b = 0;
        _y = _pointB.Y();
        break;
    case LineType::NORMAL:
        // Normal line,  $y = kx + b$ 
        _k = yDiff / xDiff;
        _b = (-_pointA.X() * yDiff + _pointA.Y() * xDiff) / xDiff;
        break;
    default:
        break;
    }
}

```

```

LineEquation::LineEquation(const Point& a, const Point& b)
{
    if (a == b
        || std::isinf(a.X())
        || std::isinf(b.X())
        || std::isinf(a.Y())
        || std::isinf(b.Y()))
        throw std::runtime_error("Cannot create line from 2 equal points, or
            coordinates incorrect (a.e. Inf)");

    _pointA = a, _pointB = b;

    yDiff = _pointB.Y() - _pointA.Y();
    xDiff = _pointB.X() - _pointA.X();
    type = (xDiff == 0)
        ? LineType::CONST_X
        : ((yDiff == 0) ? LineType::CONST_Y : LineType::NORMAL);

    initByLineType();
    return;
}

```

Point.hpp:

```

#ifndef POINT_LIB
#define POINT_LIB

class Point {
private:
    double _x, _y;

public:
    Point(double x = 0, double y = 0) : _x(x), _y(y) {};

    bool operator==(const Point& a) const { return X() == a.X() && Y() == a.Y(); }
    bool operator!=(const Point& a) const { return !(*this == a); }

    const double& X() const { return _x; }
    const double& Y() const { return _y; }
};

```

```
#endif // POINT_LIB
```

Point.cpp: