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

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 $x = Re(z)$ и мнимую $y = Im(z)$ 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,

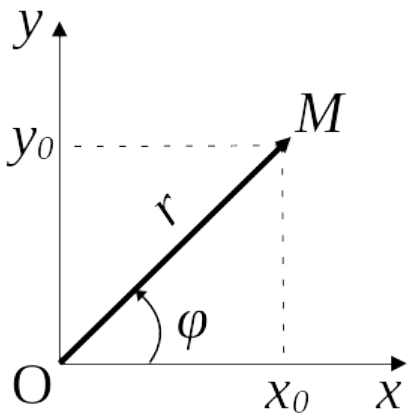

Рис. 1: Изображение числа z на плоскости

Решение и разбор задач с применением метода

Задача 1

Постановка задачи: Точка D симметрична центру описанной около треугольника ABC окружности, относительно прямой AB. Доказать, что расстояние CD выражается формулой

$$CD^2 = R^2 + AC^2 + BC^2 - AB^2 \tag{1}$$

где R - радиус описанной окружности.

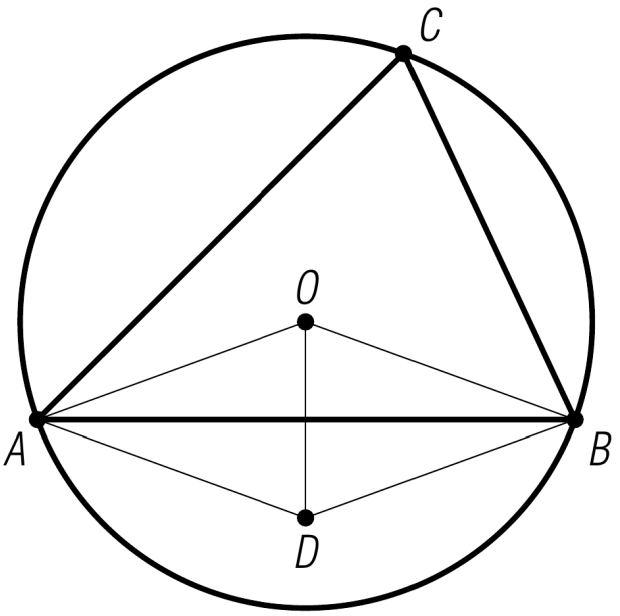

Рис. 2: Иллюстрация к задаче

Решение задачи: За начало координат плоскости будем считать точку O – центр описанной около треугольника ABC окружности. Уравнение данной окружности имеет вид $z\bar{z} = |z|^2 = x^2+y^2 = R^2$, где $z \in \mathbf{C}$. Рассмотрим четырехугольник $OADB$: $OA = AD = OB = BD$ по условию задачи и отрезки AB, OD пересекаются под прямым углом, следовательно, $OADB$ – ромб. На основании данного факта верно

следующее утверждение – $d = a + b$. Тогда

$$CD^2 = (d-c)(\bar{d}-\bar{c}) = (a+b-c)(\bar{a}+\bar{b}-\bar{c}) = 3R^2+(a\bar{b}+\bar{a}b)-(a\bar{c}+\bar{a}c)-(b\bar{c}+\bar{b}c). \quad (2)$$

Этому же выражению равна правая часть доказываемого равенства:

$$\begin{aligned} R^2 + AC^2 + BC^2 - AB^2 = R^2 + (a-c)(\bar{a}-\bar{c}) + (b-c)(\bar{b}-\bar{c}) - (a-b)(\bar{a}-\bar{b}) = \\ = 3R^2 - (a\bar{c}+\bar{a}c) - (b\bar{c}+\bar{b}c) + (a\bar{b}+\bar{a}b) \end{aligned} \quad (3)$$

Таким образом утверждение 1 верно, что и требовалось доказать.

Алгоритм программного решения частного случая задачи: На вход программы передаются координаты свободных точек комплексной плоскости, в данном примере это координаты точек A, B, C. Если A, B, C не лежат на одной прямой, то по данным входным данным строится описанная окружность и точки O, D. Если условие 1 выполняется, то задача считается решенной для данных входных данных и на экран выводятся координаты точек O, D, а также координаты всех остальных. Блок-схема алгоритма приведена на Рис. 3.

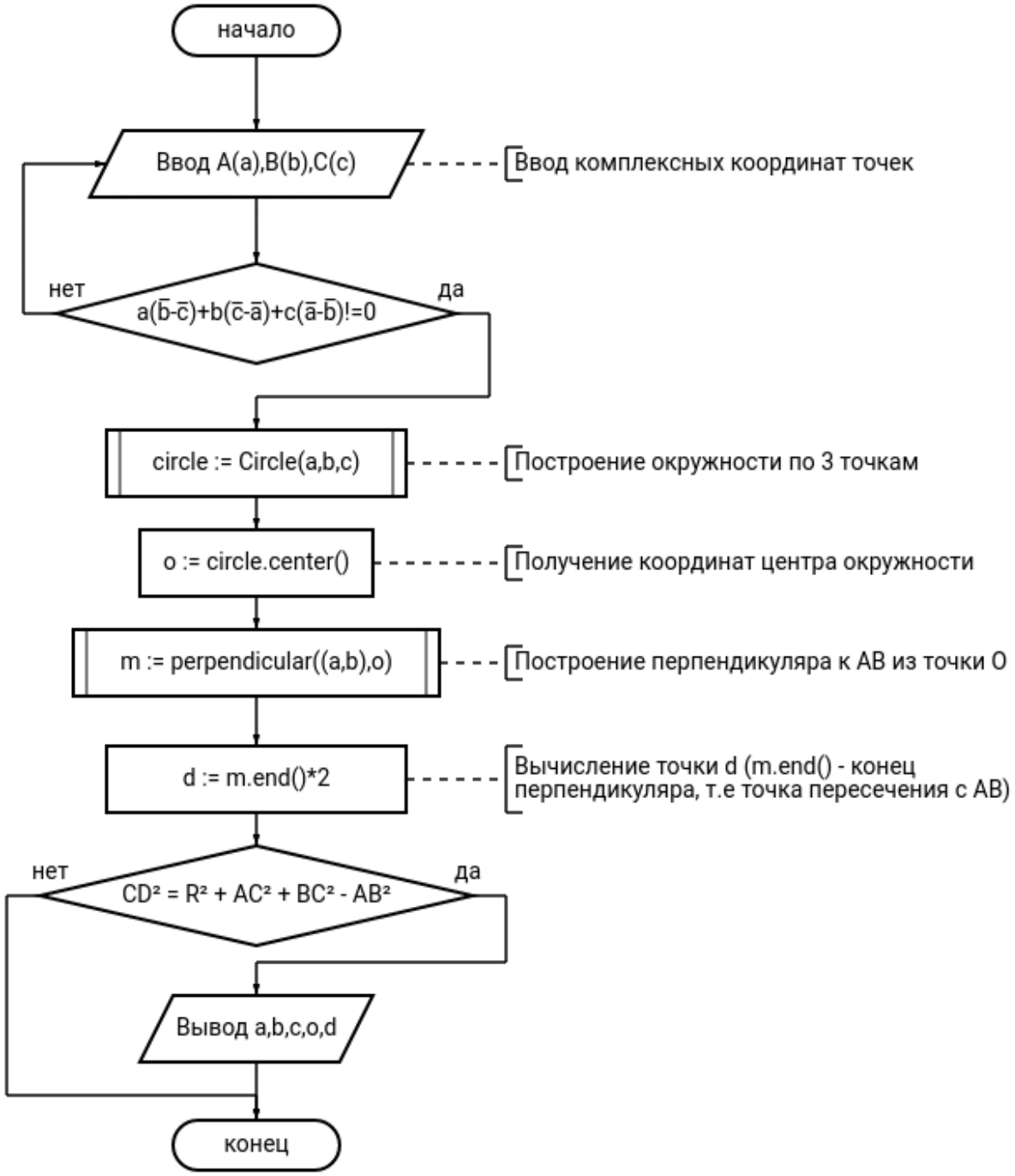

Рис. 3: Блок-схема алгоритма программы

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

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

```
inline void task1::solve(int& returnCode, const ProgramOptions& options)
{
   returnCode = 0;
   const int numbersCount = 5;
   /**
    * @brief Random constant values. They need for get x() or y() values of
    * lines kind of 'y = const' or 'x = const'
    */
   const int &randX = numbersCount, &randY = numbersCount;
   ComplexNumber numbers[numbersCount];
   const std::string labels[numbersCount] { "A", "B", "C", "D", "O" };
   // clang-format off

   // References for readability
   ComplexNumber &a = numbers[0],
                 &b = numbers[1],
                 &c = numbers[2],
                 &d = numbers[3],
                 &o = numbers[4];
   // clang-format on

   readTriangleFromUser(numbers, labels, options, returnCode);
   if (returnCode != EXIT_SUCCESS)
      return;

   Circle circle { static_cast< Point >(a),
                   static_cast< Point >(b),
                   static_cast< Point >(c) };

   o = ComplexNumber::floor({ circle.center() }, 2);

   const Line AB { a, b };
   const Line p = Line::makePerpendicular(AB, static_cast< Point >(o));
   const Point i = Line::intersect(AB, p);

   switch (p.getType()) {
      case LineType::CONST_X:
         d = Point { p.x(randY), -static_cast< Point >(o).Y() + 2 * i.Y() };
         break;
      case LineType::CONST_Y:
```

```
        d = Point { -static_cast< Point >(o).X() + 2 * i.X(), p.y(randX) };
        break;
      case LineType::NORMAL:
        d = Point { -static_cast< Point >(o).X() + 2 * i.X(),
                    -static_cast< Point >(o).Y() + 2 * i.Y() };
        break;
    }
    d = ComplexNumber::floor(d, 2);

    const LineSegment sCD { static_cast< Point >(c), static_cast< Point >(d) },
      sAC { static_cast< Point >(a), static_cast< Point >(c) },
      sBC { static_cast< Point >(b), static_cast< Point >(c) },
      sAB { static_cast< Point >(a), static_cast< Point >(b) };

    // Check 'CD^2 = R^2 + AC^2 + BC^2 - AB^2'
    if (almost_equal(power(sCD.length(), 2),
                     power(circle.radius(), 2) + power(sAC.length(), 2) +
                       power(sBC.length(), 2) - power(sAB.length(), 2),
                     2)) {
      printMessage(options, "Computed coordinates:\n");
      printNumbers(options, numbers, labels, numbersCount);
    } else
      std::cerr << "CD^2 != R^2 + AC^2 + BC^2 - AB^2.\n";
}
```

Листинг 1: Функция task1

Задача 2

Демонстрация работы:

Постановка задачи:

Решение задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

Задача 3

Демонстрация работы:

Постановка задачи:

Решение задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

Задача 4

Демонстрация работы:

Постановка задачи:

Решение задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

Задача 5

Демонстрация работы:

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.

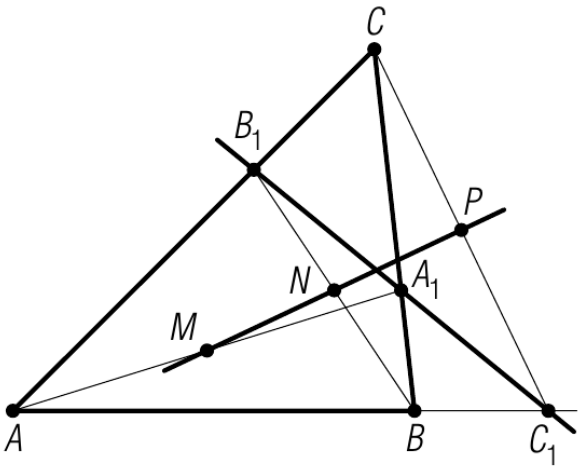

Рис. 4: Иллюстрация к задаче

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

$$\begin{cases} a(\bar{b_1} - \bar{c}) + b_1(\bar{c} - \bar{a}) + c(\bar{a} - \bar{b_1}) = 0 \\ b(\bar{c_1} - \bar{a}) + c_1(\bar{a} - \bar{b}) + a(\bar{b} - \bar{c_1}) = 0 \\ c(\bar{a_1} - \bar{b}) + a_1(\bar{b} - \bar{c}) + b(\bar{c} - \bar{a_1}) = 0 \\ a_1(\bar{b_1} - \bar{a_1}) + b_1(\bar{a_1} - \bar{a_1}) + a_1(\bar{a_1} - \bar{b_1}) = 0 \end{cases} \tag{4}$$

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

$$m(\bar{n} - \bar{p}) + n(\bar{p} - \bar{m}) + p(\bar{m} - \bar{n}) = 0, \tag{5}$$

Так как $m = \dfrac{1}{2}(a + a_1),\ n = \dfrac{1}{2}(b + b_1),\ p = \dfrac{1}{2}(c + c_1)$, то доказываемое равенство (5) 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,

$$\begin{aligned} &a(\bar{b_1} - \bar{c}) + a(\bar{b} - \bar{c_1}) + a_1(\bar{b_1} - \bar{c_1}) + a_1(\bar{b} - \bar{c}) + b(\bar{c_1} - \bar{a}) + b(\bar{c} - \bar{a_1}) + \\ &+ b_1(\bar{c_1} - \bar{a_1}) + b_1(\bar{c} - \bar{a}) + c(\bar{a_1} - \bar{b}) + c(\bar{a} - \bar{b_1}) + c_1(\bar{a_1} - \bar{b_1}) + c_1(\bar{a} - \bar{b}) = 0. \end{aligned} \tag{6}$$

Теперь легко видеть, что (6) получается при почленном сложении равенств (4)

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. 5.

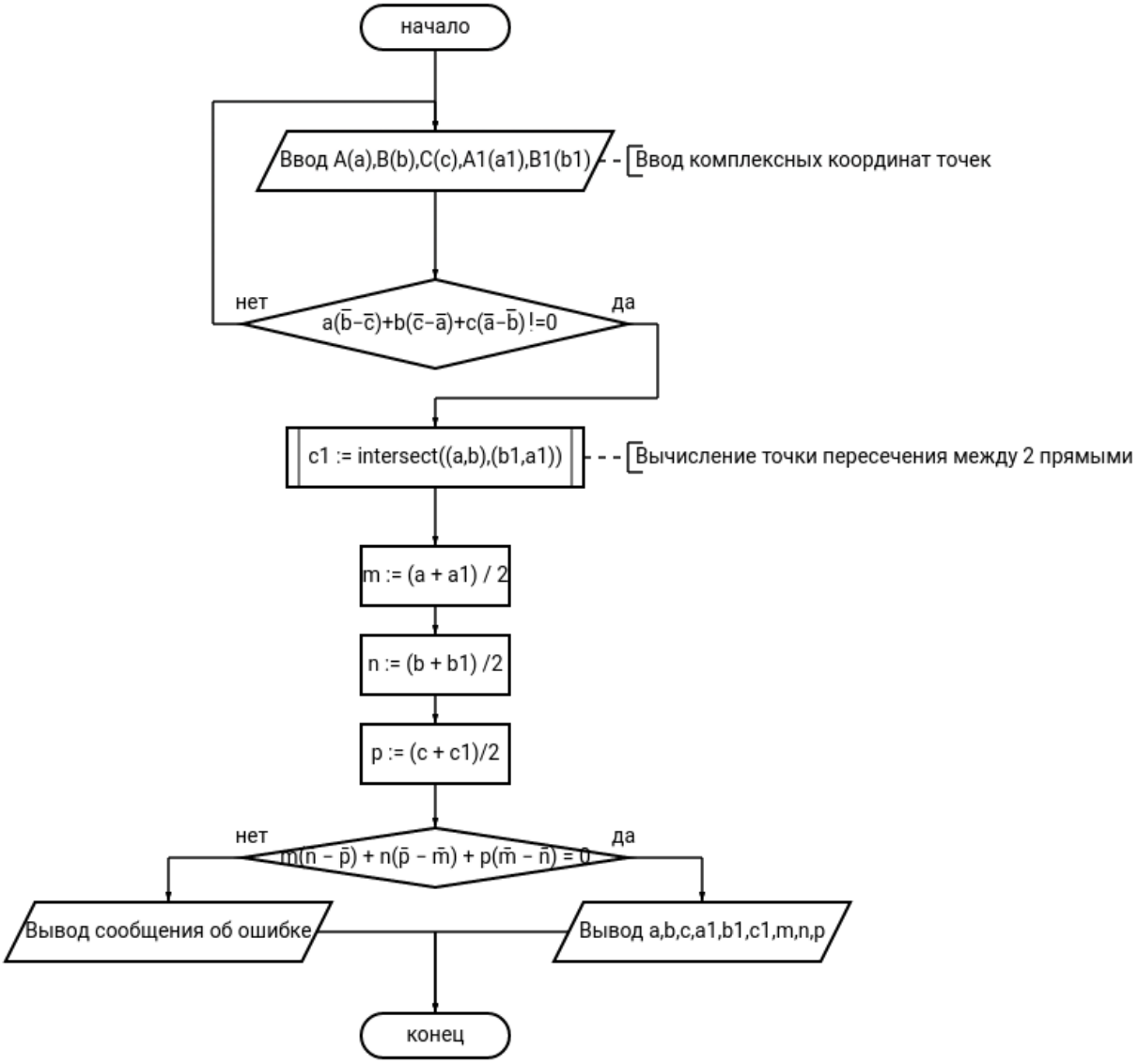

Рис. 5: Блок-схема алгоритма программы

Программная реализация задачи: Решение задачи написано на языке C++ в виде части программы для решения задач из данной работы. Реализация алгоритма программы предоставлена в функции task5::solve (файл task5.cpp):

```
#include "tfunctions.hpp"

void task5::solve(int& returnCode, const ProgramOptions& options)
{
   returnCode = 0;
   const int numbersAmount = 9;
   ComplexNumber numbers[numbersAmount];
   const std::string labels[numbersAmount] { "A", "B", "C", "A1", "B1",
                                             "C1", "M", "N", "P" };
   // clang-format off

   // References 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];
   // clang-format on
   readNumbersFromUser(numbers, labels, options, returnCode);
   if (returnCode != EXIT_SUCCESS)
      return;

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

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

   if (Line::isOnSameLine(m, n, p)) {
      printMessage(options, "Computed coordinates:");
      printNumbers(options, numbers, labels, numbersAmount);
   } else
```

```
    std::cerr
      << "The computed points M,N,P is not belong to the same line.\n";
}
```

Листинг 2: Функция task5

Демонстрация работы: Здесь будут скриншоты работы.

Задача 6

Постановка задачи:

Решение задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

Задача 7

Демонстрация работы:

Постановка задачи:

Решение задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

Задача 8

Демонстрация работы:

Постановка задачи:

Решение задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

Задача 9

Демонстрация работы:

Постановка задачи:

Решение задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

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] Кибирев В. В. Обучение методам решения геометрических задач // Вестник БГУ. 2014. №15. URL: https://cyberleninka.ru/article/n/obuchenie-metodam-resheniya-geometricheskih-zadach (дата обращения: 19.07.2022).

[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

Файл меню main.cpp:

```
#include "ComplexNumber.hpp"
#include "tfunctions.hpp"

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

#include <cstring>
#include <iostream>

const char* taskMessageBegin = "Task #";
const char* taskMessageEnd = "-----\n";

void printTaskBegin(const ProgramOptions& options, int choice)
{
   const std::string message = taskMessageBegin + std::to_string(choice) + '\n';
   printMessage(options, message.c_str());
}

void printTaskEnd(const ProgramOptions& options)
{
   printMessage(options, taskMessageEnd);
}

int main(int argc, char const* argv[])
{
   int parameters = argc;
   ProgramOptions options;
   bool useStdinToInit = false;
```

```
    if (parameters == 1) {
        printMessage(options, "Enter program number to launch: ");
        parameters = 2;
        useStdinToInit = true;
    }
    for (int i = 1; i < parameters; i++) {
        int choice, returnCode = 0;
        if (useStdinToInit) {
            std::cin >> choice;
            std::cin.ignore();
        } else {
            if (strcmp(argv[i], "-d") == 0) {
                options.outputStyle = ProgramOptions::UNIX;
                continue;
            }
            choice = std::stoi(argv[i]);
        }

        switch (choice) {
            case 1:
                printTaskBegin(options, choice);
                task1::solve(returnCode, options);
                printTaskEnd(options);
                break;
            case 5:
                printTaskBegin(options, choice);
                task5::solve(returnCode, options);
                printTaskEnd(options);
                break;
            default:
                std::cerr << "Entered program number is incorrect, retry.\n";
                break;
        }

        if (returnCode != EXIT_SUCCESS)
            return returnCode;
    }
}
```

task1.cpp:

```
#include "tfunctions.hpp"
```

```cpp
#include "Circle.hpp"
#include "ComplexNumber.hpp"
#include "LineSegment.hpp"
#include "functions.hpp"

inline void task1::solve(int& returnCode, const ProgramOptions& options)
{
   returnCode = 0;
   const int numbersCount = 5;
   /**
    * @brief Random constant values. They need for get x() or y() values of
    * lines kind of 'y = const' or 'x = const'
    */
   const int &randX = numbersCount, &randY = numbersCount;
   ComplexNumber numbers[numbersCount];
   const std::string labels[numbersCount] { "A", "B", "C", "D", "O" };
   // clang-format off

   // References for readability
   ComplexNumber &a = numbers[0],
                 &b = numbers[1],
                 &c = numbers[2],
                 &d = numbers[3],
                 &o = numbers[4];
   // clang-format on

   readTriangleFromUser(numbers, labels, options, returnCode);
   if (returnCode != EXIT_SUCCESS)
      return;

   Circle circle { static_cast< Point >(a),
                   static_cast< Point >(b),
                   static_cast< Point >(c) };

   o = ComplexNumber::floor({ circle.center() }, 2);

   const Line AB { a, b };
   const Line p = Line::makePerpendicular(AB, static_cast< Point >(o));
   const Point i = Line::intersect(AB, p);
```

```
    switch (p.getType()) {
      case LineType::CONST_X:
        d = Point { p.x(randY), -static_cast< Point >(o).Y() + 2 * i.Y() };
        break;
      case LineType::CONST_Y:
        d = Point { -static_cast< Point >(o).X() + 2 * i.X(), p.y(randX) };
        break;
      case LineType::NORMAL:
        d = Point { -static_cast< Point >(o).X() + 2 * i.X(),
                    -static_cast< Point >(o).Y() + 2 * i.Y() };
        break;
    }
    d = ComplexNumber::floor(d, 2);

    const LineSegment sCD { static_cast< Point >(c), static_cast< Point >(d) },
      sAC { static_cast< Point >(a), static_cast< Point >(c) },
      sBC { static_cast< Point >(b), static_cast< Point >(c) },
      sAB { static_cast< Point >(a), static_cast< Point >(b) };

    // Check 'CD^2 = R^2 + AC^2 + BC^2 - AB^2'
    if (almost_equal(power(sCD.length(), 2),
                     power(circle.radius(), 2) + power(sAC.length(), 2) +
                       power(sBC.length(), 2) - power(sAB.length(), 2),
                     2)) {
      printMessage(options, "Computed coordinates:\n");
      printNumbers(options, numbers, labels, numbersCount);
    } else
      std::cerr << "CD^2 != R^2 + AC^2 + BC^2 - AB^2.\n";
}
```

task2.cpp:

```
inline void task2() {}
```

task3.cpp:

```
inline void task3() {}
```

tfunctions.hpp:

```
/**
 * @file tfunctions.hpp
 * @author Grigory Stupnikov (gs.obr@ya.ru)
 * @brief Namespaces and functions with task implementation code
 * @version 0.1
 * @date 2022-07-14
 *
 * @copyright Copyright © 2022 Grigory Stupnikov. All rights reserved. Licensed
 * under GNU GPLv3. See https://opensource.org/licenses/GPL-3.0.
 */
#ifndef COURSEWORK_4_1_TFUNCTIONS_HPP
#define COURSEWORK_4_1_TFUNCTIONS_HPP

#include "ComplexNumber.hpp"

#include <tuple>

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

struct ProgramOptions
{
   enum OutputStyle
   {
      UNIX, // Plain, no output except result, useful for debug
      RICH  // Plain, print messages to user
   } outputStyle = RICH;
};

/**
 * @brief Intersect of 2 line segments
 * @return ComplexNumber, intersection point of lines
 */
ComplexNumber intersect(const clineSegment_t& first, clineSegment_t second);

bool isPointBelongsSegment(const clineSegment_t& segment, ComplexNumber point);

/**
 * @brief Print message. It prints only numbers if options.outputStyle == UNIX.
 * Passing non null-terminated string is UB, don't do this!
 * Supported format specifiers:
 * %N - complex number (ComplexNumber*)
```

```
 * %s - string (const char*)
 * @param ... data to print
 */
void printMessage(const ProgramOptions& options, const char* format, ...);

void printNumbers(const ProgramOptions& options, const ComplexNumber numbers[],
                  const std::string labels[], const size_t amount);

namespace task1 {
   void readTriangleFromUser(ComplexNumber arr[3], const std::string labels[3],
                             const ProgramOptions& options, int& returnCode);
   void solve(int& returnCode, const ProgramOptions& options);
}

namespace task5 {
   void readNumbersFromUser(ComplexNumber arr[5], const std::string labels[5],
                            const ProgramOptions& options, int& returnCode);

   void solve(int& returnCode, const ProgramOptions& options);
}

#endif // COURSEWORK_4_1_TFUNCTIONS_HPP
```

tfunctions.cpp:

```
#include "tfunctions.hpp"
#include "Line.hpp"

#include <cmath>
#include <cstdarg>
#include <cstring>
#include <functions.hpp>
#include <unordered_map>

void printElementRichStyle(const char* specifier, const void* data);
void printElementUnixStyle(const char* specifier, const void* data);
void printElement(const ProgramOptions& options, const char* specifier,
                  const void* data);
bool isValidSpecifier(const std::string& specifier);
const void* extractElipsisElement(va_list* elipsis,
                                  const std::string& specifier);
```

```
const size_t specifiersAmount = 2;

enum class ElementType
{
   ComplexNumber,
   String
};

const std::unordered_map< std::string, ElementType > specifiers = {
   { "%N", ElementType::ComplexNumber },
   { "%s", ElementType::String }
};

ComplexNumber intersect(const clineSegment_t& first, clineSegment_t second)
{
   return ComplexNumber { Line::intersect(Line(first.first, first.second),
                                          Line(second.first, second.second)) };
}

bool isPointBelongsSegment(const clineSegment_t& segment, ComplexNumber cpoint)
{
   const 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 <= cpoint.Im() && cpoint.Im() <= imMax) &&
                          (reMin <= cpoint.Re() && cpoint.Re() <= reMax);

   return line.isBelongs(static_cast< Point >(cpoint)) && isPointInBounds;
}

void task1::readTriangleFromUser(ComplexNumber arr[3],
                                 const std::string labels[3],
                                 const ProgramOptions& options, int& returnCode)
{
   const size_t labelsCount = 3;

   // clang-format off

   ComplexNumber &a = arr[0],
```

```
                &b = arr[1],
                &c = arr[2];
    // clang-format on

    bool isTriangle = false;
    while (!isTriangle) {
        printMessage(options, "Enter coordinates of triangle's points:\n");
        if (std::cin.fail()) {
            if (std::cin.eof()) {
                std::cerr << "User input was canceled. Aborting...\n";
                return;
            }
            std::cin.ignore();
            std::cin.clear();
        }

        for (size_t i = 0; i < labelsCount; i++) {
            printMessage(options, (' ' + labels[i] + ": ").c_str());
            std::cin >> arr[i];
            arr[i] = ComplexNumber::floor(arr[i], 2);
        }

        isTriangle = !Line::isOnSameLine(a, b, c);
        if (!isTriangle)
            std::cerr << "Incorrect a,b,c. Must be points of the triangle ABC\n";
    }
}

void task5::readNumbersFromUser(ComplexNumber arr[5],
                                const std::string labels[5],
                                const ProgramOptions& options, int& returnCode)
{
    returnCode = 0;
    const size_t labelsCount = 5;

    // clang-format off
    ComplexNumber &a = arr[0],
                  &b = arr[1],
                  &c = arr[2],
                  &a1 = arr[3],
                  &b1 = arr[4];
```

```
    // clang-format on

    bool isTriangle = false, isValidA1 = false, isValidB1 = false;
    while (!(isTriangle && isValidA1 && isValidB1)) {
        printMessage(options, "Enter coordinates of a,b,c,a1,b1 points:\n");
        if (std::cin.fail()) {
            if (std::cin.eof()) {
                std::cerr << "User input was canceled. Aborting...\n";
                returnCode = 1;
                return;
            }
            std::cin.ignore();
            std::cin.clear();
        }
        for (size_t i = 0; i < labelsCount; i++) {
            printMessage(options, (' ' + labels[i] + ": ").c_str());
            std::cin >> arr[i];
            arr[i] = ComplexNumber::floor(arr[i], 2);
        }

        isTriangle = !Line::isOnSameLine(a, b, c);
        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";
    }
}

void printElementUnixStyle(const ElementType& type, const void* data)
{
    const ComplexNumber* number;

    switch (type) {
        case ElementType::ComplexNumber:
```

```
         number = static_cast< const ComplexNumber* >(data);
         if (number)
            std::cout << number->Re() << ' ' << number->Im();
         break;

      case ElementType::String:
      default:
         break;
   }
}

void printElementRichStyle(const ElementType& type, const void* data)
{
   const ComplexNumber* number;
   const char* string;
   switch (type) {
      case ElementType::ComplexNumber:
         number = static_cast< const ComplexNumber* >(data);
         if (number)
            std::cout << *number;
         break;
      case ElementType::String:
         string = static_cast< const char* >(data);
         if (string)
            std::cout << string;
      default:
         break;
   }
}

void printElement(const ProgramOptions& options, const ElementType& type,
                  const void* data)
{
   switch (options.outputStyle) {
      case ProgramOptions::UNIX:
         printElementUnixStyle(type, data);
         break;
      case ProgramOptions::RICH:
         printElementRichStyle(type, data);
         break;
      default:
```

```
         break;
   }
}

void printChar(const ProgramOptions& options, const char& c)
{
   switch (options.outputStyle) {
      case ProgramOptions::RICH:
         std::cout << c;
         break;
      default:
         break;
   }
}

bool isValidSpecifier(const std::string& specifier)
{
   if (specifiers.find(specifier) != specifiers.cend())
      return true;
   else
      return false;
}

void printMessage(const ProgramOptions& options, const char* format, ...)
{
   va_list data;
   size_t formatLen = strlen(format);
   const void* element;

   va_start(data, format);
   for (size_t i = 0; i < formatLen; ++i) {
      switch (format[i]) {
         case '%':
            if (i < formatLen - 1) {
               const std::string specifier(&format[i], 2);
               if (isValidSpecifier(specifier)) {
                  element = extractElipsisElement(&data, specifier);
                  printElement(options, specifiers.at(specifier), element);
               }
               ++i;
            }
```

```
            break;
         default:
            printChar(options, format[i]);
            break;
      }
   }
   va_end(data);
}

void printNumbers(const ProgramOptions& options, const ComplexNumber numbers[],
                  const std::string labels[], const size_t amount)
{
   switch (options.outputStyle) {
      case ProgramOptions::UNIX:
         for (size_t i = 0; i < amount; i++) {
            printElementUnixStyle(ElementType::ComplexNumber, &numbers[i]);
            if (i < amount - 1)
               std::cout << ' ';
         }
         break;
      case ProgramOptions::RICH:
         for (size_t i = 0; i < amount; i++) {
            printElementRichStyle(ElementType::ComplexNumber, &numbers[i]);
            if (i < amount - 1)
               std::cout << ' ';
         }
         break;
      default:
         break;
   }
}

const void* extractElipsisElement(va_list* elipsis,
                                  const std::string& specifier)
{
   switch (specifiers.at(specifier)) {
      case ElementType::ComplexNumber:
         return va_arg(*elipsis, ComplexNumber*);
      case ElementType::String:
         return va_arg(*elipsis, const char*);
      default:
```

```
        return nullptr;
    }
}
```

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(const Point& point);

    void 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;

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

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

    static ComplexNumber conjugate(const ComplexNumber& number);
```

```
    /**
     * @brief Round ComplexNumber to specified digits after decimal separator
     *
     * @param number source number
     * @param ulp amount digits after decimal separator
     * @return ComplexNumber - Rounded number
     */
    static ComplexNumber floor(const ComplexNumber& number, int8_t ulp);

    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"

#include "functions.hpp"

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

ComplexNumber::ComplexNumber(const Point& point)
{
    _real = point.X();
    _imaginary = point.Y();
}

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;
}

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

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);
}
```

```
ComplexNumber ComplexNumber::conjugate(const ComplexNumber& number)
{
   return ComplexNumber(number.Re(), -number.Im());
}

ComplexNumber ComplexNumber::floor(const ComplexNumber& number, int8_t ulp)
{
   return ComplexNumber(::floor(number._real, ulp),
                        ::floor(number._imaginary, 2));
}
```

Line.hpp:

```
#ifndef LINE_LIB
#define LINE_LIB

#include "ComplexNumber.hpp"
#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);

 class LineEquation;
 void finishInit(const LineEquation& initedEquation);

public:
 Line(double k, double b);
 Line(const std::pair< Point, Point >& pair);
 Line(const Point& first, const Point& second) :
   Line(std::make_pair(first, second))
 {
 }
 /**
  * @brief Construct a new Line object (algorithm is same as for the two
  * Points)
  */
 Line(const ComplexNumber& first, const ComplexNumber& second);
 Line(const Line& source);

 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) const;
 bool isCollinear(const Line& other) const;

 /**
  * @brief Swap Line @b this with @b other
  */
 void swap(Line& other) { Line::swap(*this, other); };

 void operator=(const Line& other);

 static Line makePerpendicular(const Line& to, const Point& from);
 /**
```

```
     * @brief Intersect of 2 line segments
     * @return Point, intersection point of lines, or (Inf;Inf) if lines is
     * collinear
     */
    static Point intersect(const Line& first, const Line& second);
    static bool isOnSameLine(const Point& a, const Point& b, const Point& c);
    static bool isOnSameLine(const ComplexNumber& a, const ComplexNumber& b,
                             const ComplexNumber& c);

    static void swap(Line& left, Line& right);
};

#endif // LINE_LIB
```

Line.cpp:

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

class Line::LineEquation
{
  private:
    Point _pointA, _pointB;
    double _k, _b, _x, _y;
    double xDiff, yDiff;
    bool _inited = false;
    LineType type;

    void initByLineType();

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

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

    double xConst() const { return _x; }
```

```
   double yConst() const { return _y; }

   bool isInited() const { return _inited; }

   LineType getType() const { return type; }
};

Point intersectEqualType(const Line& first, const Line& second);
Point intersectSubNormalType(const Line& first, const Line& second);

#pragma region Constructors
void Line::finishInit(const LineEquation& initedEquation)
{
   if (!initedEquation.isInited())
      throw std::runtime_error(
        "Cannot finish line initialization with not inited equation!");

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

Line::Line(double k, double b) : _k(k), _b(b)
{
   if (std::isinf(k)) {
      throw std::runtime_error(
        "Cannot construct line x = ? from equation y = kx + b");
   } else if (k == 0) {
      _type = LineType::CONST_Y;
      _y = b;
   }
```

```
}

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

Line::Line(const ComplexNumber& first, const ComplexNumber& second)
{
   LineEquation equation = LineEquation(first, second);
   finishInit(equation);
}

Line::Line(const Line& source)
{
   *this = source;
}
#pragma endregion
#pragma region Getters and methods
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::CONST_Y:
         return _y == y;
      case LineType::NORMAL:
         return true;
      default:
         return false;
   }
}

bool Line::isBelongs(Point point) const
{
   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;
    }
}
#pragma endregion

void Line::operator=(const Line& other)
{
    _k = other._k;
    _b = other._b;
    _type = other._type;
}

Line Line::makePerpendicular(const Line& to, const Point& from)
{
    /**
     * @brief Random constant
     */
    const unsigned int yDiff = 5;

    switch (to.getType()) {
        case LineType::CONST_X:
            // Perpendicular is CONST_Y, y = from.Y
            return Line { 0, from.Y() };
        case LineType::CONST_Y:
            // Perpendicular is CONST_X, x = from.X
            // We should use constructor by 2 points because y = kx + b not
            // represent equation x = const
            return Line { from, Point { from.X(), from.Y() + yDiff } };
        case LineType::NORMAL:
            // Perpendicular is NORMAL, k = -1/to.k and 'to' on perpendicular
            {
                const double k = -1 / to.K();
                const double b = from.Y() + from.X() / to.K();
                return Line(k, b);
            }
        default:
            return Line(0, 0);
    }
}
```

```
Point Line::intersect(const Line& first, const Line& second)
{

   if (first._type == second._type) {
      return intersectEqualType(first, second);
   } else if (first._type == LineType::NORMAL ||
            second._type == LineType::NORMAL) {
      return intersectSubNormalType(first, second);
   } else {
      // first is CONST_Y, second is CONST_X or vice versa
      // Make first is CONST_X
      Line f { first }, s { second };
      if (f._type != LineType::CONST_X)
         Line::swap(f, s);

      return Point { f._x, s._y };
   }
}

bool Line::isCollinear(const Line& other) const
{
   // TODO
   return true;
}

bool Line::isOnSameLine(const Point& a, const Point& b, const Point& c)
{
   return Line(a, b).isBelongs(c);
}
bool Line::isOnSameLine(const ComplexNumber& a, const ComplexNumber& b,
                        const ComplexNumber& c)
{
   return isOnSameLine(
     static_cast< Point >(a), static_cast< Point >(b), static_cast< Point >(c));
}

void Line::swap(Line& left, Line& right)
{
   Line tmp { left };
   left = right;
```

```
   right = tmp;
}

#pragma region Implementation
void Line::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;
   }
}

Line::LineEquation::LineEquation(const ComplexNumber& a,
                                 const ComplexNumber& b) :
  LineEquation(static_cast< Point >(a), static_cast< Point >(b))
{
}

Line::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();
    _inited = true;
    return;
}

Point intersectEqualType(const Line& first, const Line& second)
{
    switch (first.getType()) {
        case LineType::CONST_X:
        case LineType::CONST_Y:
            return Point { std::numeric_limits< double >::infinity(),
                           std::numeric_limits< double >::infinity() };
        case LineType::NORMAL: {
            const double x = (second.B() - first.B()) / (first.K() - second.K());
            const double y = second.y(x);
            return Point { x, y };
        }
        default:
            return Point();
    }
}

Point intersectSubNormalType(const Line& first, const Line& second)
{
    Line f { first }, s { second };
    /**
     * @brief Random constant value
     */
    const unsigned int cv = 0;

    // Make first is NORMAL
    if (f.getType() != LineType::NORMAL)
```

```
      Line::swap(f, s);

   switch (s.getType()) {
      case LineType::CONST_X:
         return Point { s.x(cv), f.y(s.x(cv)) };
      case LineType::CONST_Y:
         return Point { f.x(s.y(cv)), s.y(cv) };
      default:
         return Point();
   }
}
#pragma endregion
```

Определение класса Point (header-only):

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); }

   static Point middle(const Point& a, const Point& b){
      return Point((a._x + b._x) / 2, (a._y + b._y) / 2);
   }

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

```
#endif // POINT_LIB
```