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 существует вектор $\overrightarrow{OM}$ (точка O – начало координат), то отображение $M(z) \to OM$ биективно.

5. Из свойств операций над векторами следует, что сумме и разности векторов однозначно соответствуют сумма и разность соответствующих им комплексных чисел.

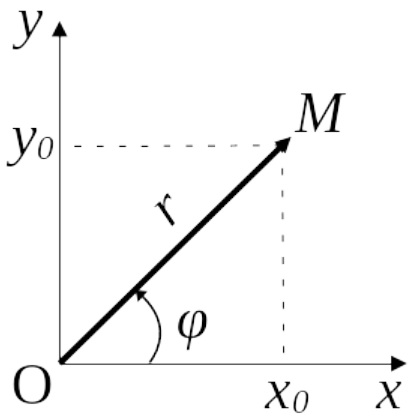

Рис. 1: Изображение числа z на плоскости

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.

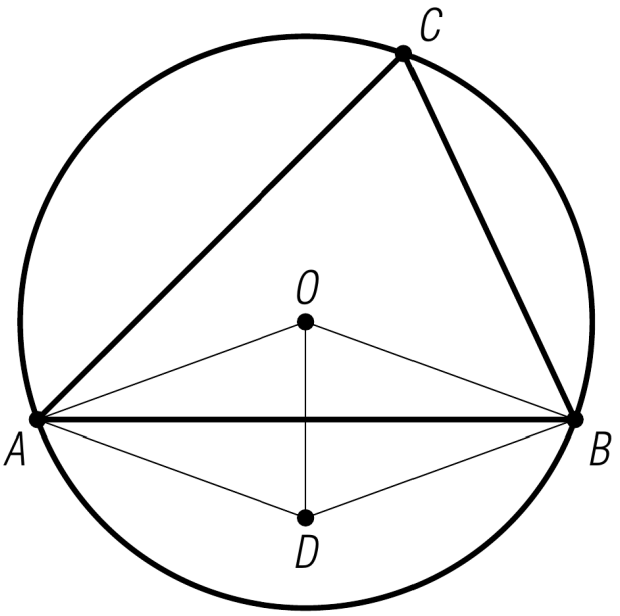

Рис. 2: Иллюстрация к задаче

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.

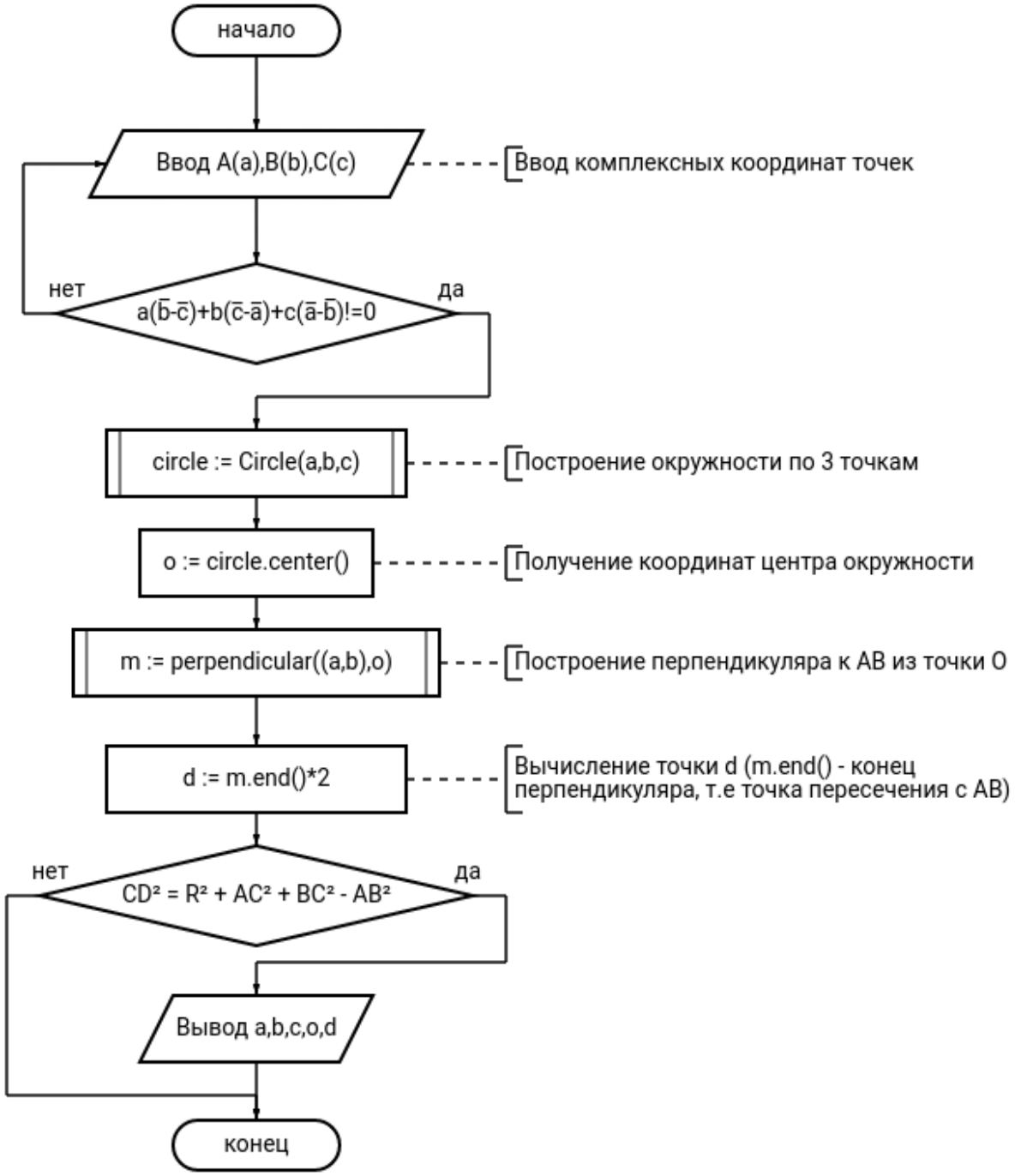

Рис. 3: Блок-схема алгоритма программы

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):

```
/**
 * @brief Amount digits after decimal separator
 */
const int8_t roundPrecision = 2;
```

```
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::round({ circle.center() }, roundPrecision);
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::round(d, roundPrecision);
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'
  const double lhs = power(sCD.length(), 2);
  const double rhs = power(circle.radius(), 2) + power(sAC.length(), 2) +
                     power(sBC.length(), 2) - power(sAB.length(), 2);
  if (areEqual(lhs, rhs, comparePrecision)) {
     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::solve

Демонстрация работы: TODO

Задача 2

Постановка задачи: Точка M – середина дуги AB окружности. Доказать, что для произвольной точки N этой окружности имеет место равенство

$$\left|AM^2 - MN^2\right| = AN \cdot BN. \tag{4}$$

Решение задачи: Пусть точкам A, B, M, N соответствуют комплексные числа a, b, m, n. Для упрощения доказательства возьмем дугу AB такую, что $a = \bar{b}, b = \bar{a}$ и $m = 1 + i0 = 1$ соответственно, и окружность радиуса $r = 1$. За начало координат примем центр окружности (см рис. 4). Тогда уравнение окружности имеет вид $z\bar{z} = 1$, и поэтому $a = \bar{b}, b = \bar{a}$.

Находим: $AN \cdot BN = |a - n| \cdot |b - n| = |(a - n)(\bar{a} - n)| = \left|a\bar{a} - na - n\bar{a} + n^2\right| = \left|1 + n^2 - n(a + \bar{a})\right|$.

Так как $AM^2 = (a - 1)(\bar{a} - 1)$ и $MN^2 = (n - 1)(\bar{n} - 1)$, то $\left|AM^2 - MN^2\right| = |n + \bar{n} - (a + \bar{a})|$. Умножив это равенство на $|n| = 1$, получим: $\left|AM^2 - MN^2\right| = \left|n^2 + 1 - n(a + \bar{a})\right| = AN \cdot BN$.

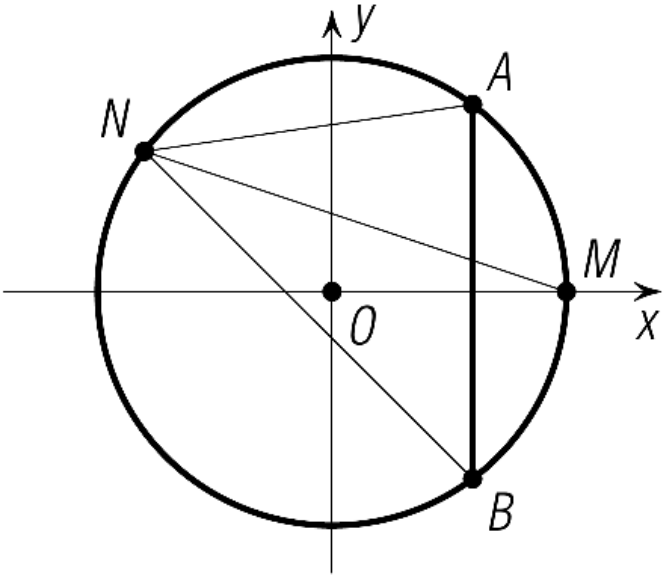

Рис. 4: Иллюстрация к доказательству

Алгоритм программного решения частного случая задачи: Строится окружность радиуса $r = 1$. На вход программы передаются координаты свободных точек комплексной плоскости, в данном примере это координаты точек A, B, N, а также вспомогательная точка T ($T \in AB$), которая нужна для однозначного построения дуги. Если A, B, N принадлежат построенной окружности, то по данным входным данным вычисляются координаты точки M. Если условие задачи (4) выполняется, то задача считается решенной для данных входных данных и на экран выводятся координаты точки M, а также координаты всех остальных. Блок-схема алгоритма приведена на Рис. 5.

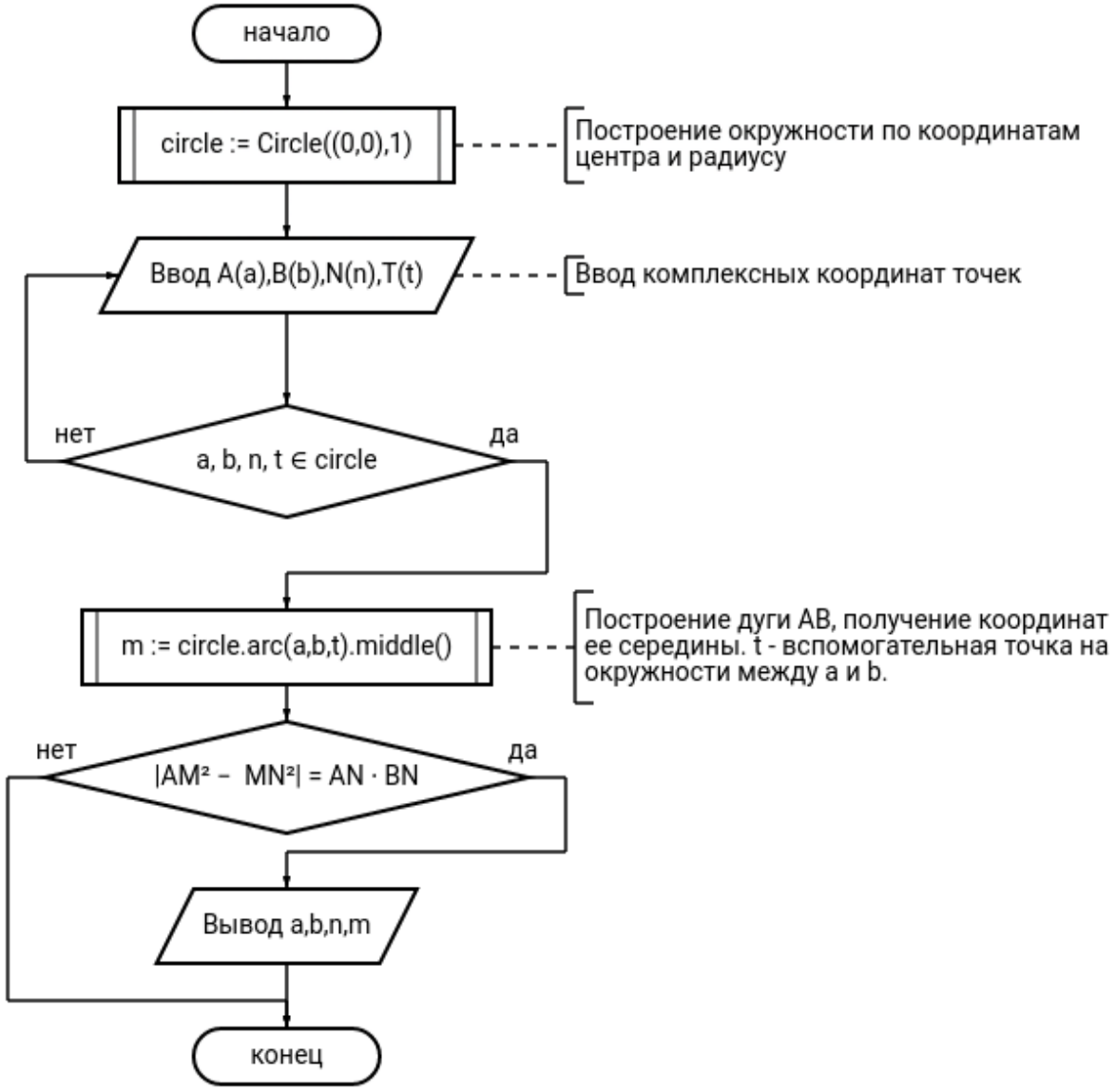

Рис. 5: Блок-схема алгоритма программы

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 task2::solve (файл task2.cpp):

```
inline void task2::solve(int& returnCode, const ProgramOptions& options)
{
   /**
    * @brief Compare with precision up to comparePrecision digits after decimal
    * separator.
    */
   const uint8_t comparePrecision = 1;
   const int numbersCount = 5;
```

```
    const uint32_t circleRadius = 1;
    const bool useApproximation = true;
    ComplexNumber numbers[numbersCount];
    const std::string labels[numbersCount] { "A", "B", "N", "T", "M" };
    // clang-format off
    // References for readability
    ComplexNumber &a = numbers[0],
                  &b = numbers[1],
                  &n = numbers[2],
                  &t = numbers[3],
                  &m = numbers[4];
    // clang-format on
    const Circle circle { Point::zero(), circleRadius };
    returnCode = 0;
    do {
       readNumbersFromUser(numbers, labels, options, returnCode);
       if (returnCode != EXIT_SUCCESS)
          return;
    } while (!circle.isBelongs(static_cast< Point >(a), comparePrecision) ||
             !circle.isBelongs(static_cast< Point >(b), comparePrecision) ||
             !circle.isBelongs(static_cast< Point >(n), comparePrecision));
    m = CircleArc(circle,
                  static_cast< Point >(a),
                  static_cast< Point >(b),
                  static_cast< Point >(t),
                  useApproximation)
          .middle();
    const LineSegment AM { static_cast< Point >(a), static_cast< Point >(m) },
      MN { static_cast< Point >(m), static_cast< Point >(n) },
      AN { static_cast< Point >(a), static_cast< Point >(n) },
      BN { static_cast< Point >(b), static_cast< Point >(n) };
    // Check '|AM^2 - MN^2| = AN * BN'
    if (areEqual(std::abs(power(AM.length(), 2) - power(MN.length(), 2)),
                 AN.length() * BN.length(),
                 static_cast< int8_t >(comparePrecision))) {
       printMessage(options, "Computed coordinates:\n");
       printNumbers(options, numbers, labels, numbersCount);
    } else
       std::cerr << "|AM^2 - MN^2| != AN * BN.\n";
}
```

Листинг 2: Функция task2::solve

Демонстрация работы: TODO

Задача 3

Постановка задачи: Докажите, что сумма квадратов диагоналей параллелограмма равна сумме квадратов всех его сторон (Рис. 6). Таким образом, требуется доказать, что

$$AD^2 + BC^2 = AB^2 + CD^2 + BD^2 + AC^2 \tag{5}$$

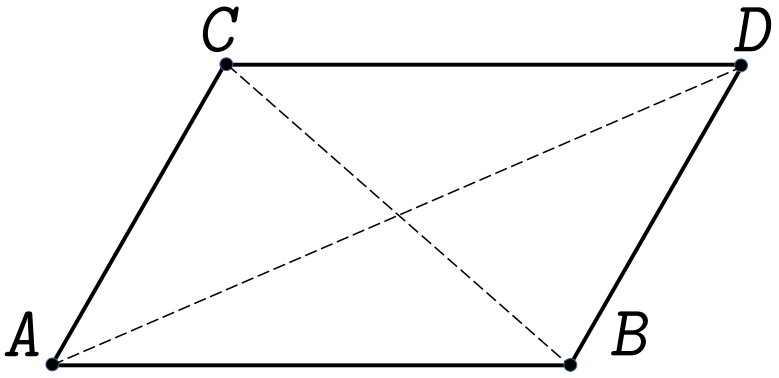

Рис. 6: Иллюстрация к задаче

Решение задачи: Зададим точку $A(a) = 0 + i \cdot 0$. Тогда верны следующие утверждения: $d = c + b$, $\overrightarrow{CB} = b - c$.

$$AB^2 + BD^2 + CD^2 + AC^2 = |b|^2 + |c|^2 + |b|^2 + |c|^2 = 2b\bar{b} + 2c\bar{c} \tag{6}$$

$$AD^2 + BC^2 = |d|^2 + |b - c|^2 = d\bar{d} + (b - c)(\bar{b} - \bar{c}) = (c + b)(\bar{c} + \bar{c}) + b\bar{b} - b\bar{c} - \tag{7}$$
$$- c\bar{b} + c\bar{c} = c\bar{c} + c\bar{b} + b\bar{c} + 2b\bar{b} + c\bar{c} - b\bar{c} - c\bar{b} = 2b\bar{b} + 2c\bar{c}$$

Таким образом, выражения (6) и (7) равны друг другу, 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 образуют треугольник, то по данным входным строится параллелограмм. Если условие задачи (5) выполняется, то задача считается решенной для данных входных данных и на экран выводятся координаты точки D, а также координаты всех остальных. Блок-схема алгоритма приведена на Рис. 7.

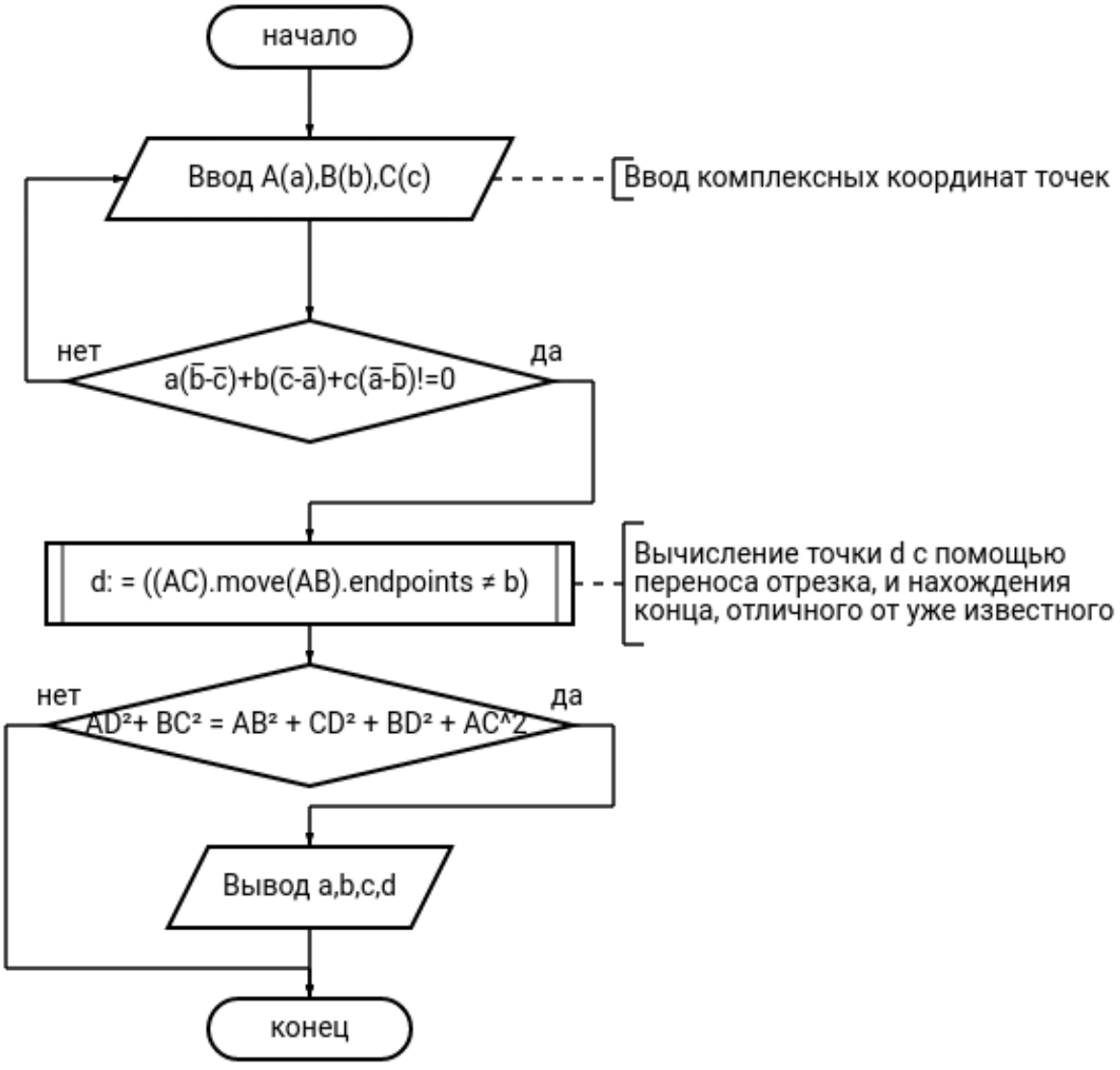

Рис. 7: Блок-схема алгоритма программы

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 task3::solve (файл task3.cpp):

```
inline void task3::solve(int& returnCode, const ProgramOptions& options)
{
   /**
    * @brief Compare with precision up to comparePrecision digits after decimal
    * separator.
    */
   const uint8_t comparePrecision = 1;
   const int numbersCount = 4;
   ComplexNumber numbers[numbersCount];
   const std::string labels[numbersCount] { "A", "B", "C", "D" };
   // clang-format off
   // References for readability
   ComplexNumber &a = numbers[0],
                 &b = numbers[1],
                 &c = numbers[2],
                 &d = numbers[3];
   // clang-format on
   returnCode = 0;
   task1::readTriangleFromUser(numbers, labels, options, returnCode);
   if (returnCode != EXIT_SUCCESS)
      return;
   const LineSegment AB { static_cast< Point >(a), static_cast< Point >(b) },
     AC { static_cast< Point >(a), static_cast< Point >(c) },
     BC { static_cast< Point >(b), static_cast< Point >(c) };
   const std::pair< Point, Point >& pair = AC.move(AB).getEndpoints();
   d = (pair.first == static_cast< Point >(b)) ? pair.second : pair.first;
   const LineSegment BD { static_cast< Point >(b), static_cast< Point >(d) },
     CD { static_cast< Point >(c), static_cast< Point >(d) },
     AD { static_cast< Point >(a), static_cast< Point >(d) };
   // Check 'AD^2 + BC^2 = AB^2 + CD^2 + BD^2 + AC^2'
   if (areEqual(power(AD.length(), 2) + power(BC.length(), 2),
                power(AB.length(), 2) + power(CD.length(), 2) +
                  power(BD.length(), 2) + power(AC.length(), 2),
                static_cast< int8_t >(comparePrecision))) {
      printMessage(options, "Computed coordinates:\n");
      printNumbers(options, numbers, labels, numbersCount);
   } else
      std::cerr << "AD^2 + BC^2 != AB^2 + CD^2 + BD^2 + AC^2.\n";
}
```

Листинг 3: Функция task3::solve

Демонстрация работы: TODO

Задача 4

Постановка задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

Демонстрация работы: TODO

Задача 5

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.

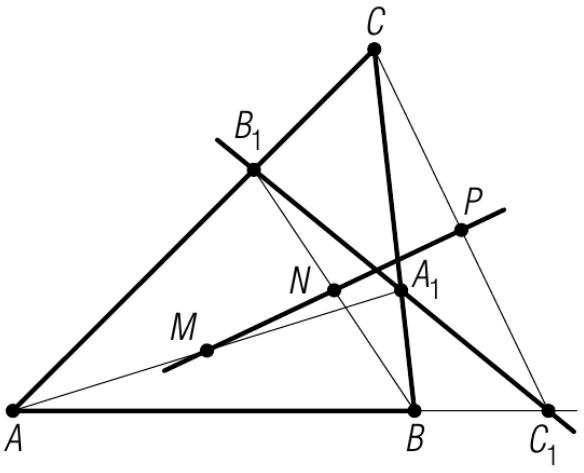

Рис. 8: Иллюстрация к задаче

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

$$\begin{cases} a(\bar{b_1} - \bar{c}) + b_1(\bar{c} - \bar{a}) + c(\bar{a} - \bar{b_1}) = 0 \\ b(\bar{c_1} - \bar{a}) + c_1(\bar{a} - \bar{b}) + a(\bar{b} - \bar{c_1}) = 0 \\ c(\bar{a_1} - \bar{b}) + a_1(\bar{b} - \bar{c}) + b(\bar{c} - \bar{a_1}) = 0 \\ a_1(\bar{b_1} - \bar{a_1}) + b_1(\bar{a_1} - \bar{a_1}) + a_1(\bar{a_1} - \bar{b_1}) = 0 \end{cases} \tag{8}$$

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

$$m(\bar{n} - \bar{p}) + n(\bar{p} - \bar{m}) + p(\bar{m} - \bar{n}) = 0, \tag{9}$$

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 (9) 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,

$$\begin{aligned} &a(\bar{b_1} - \bar{c}) + a(\bar{b} - \bar{c_1}) + a_1(\bar{b_1} - \bar{c_1}) + a_1(\bar{b} - \bar{c}) + b(\bar{c_1} - \bar{a}) + b(\bar{c} - \bar{a_1}) + \\ &+ b_1(\bar{c_1} - \bar{a_1}) + b_1(\bar{c} - \bar{a}) + c(\bar{a_1} - \bar{b}) + c(\bar{a} - \bar{b_1}) + c_1(\bar{a_1} - \bar{b_1}) + c_1(\bar{a} - \bar{b}) = 0. \end{aligned} \tag{10}$$

Теперь легко видеть, что (10) получается при почленном сложении равенств (8)

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. 9.

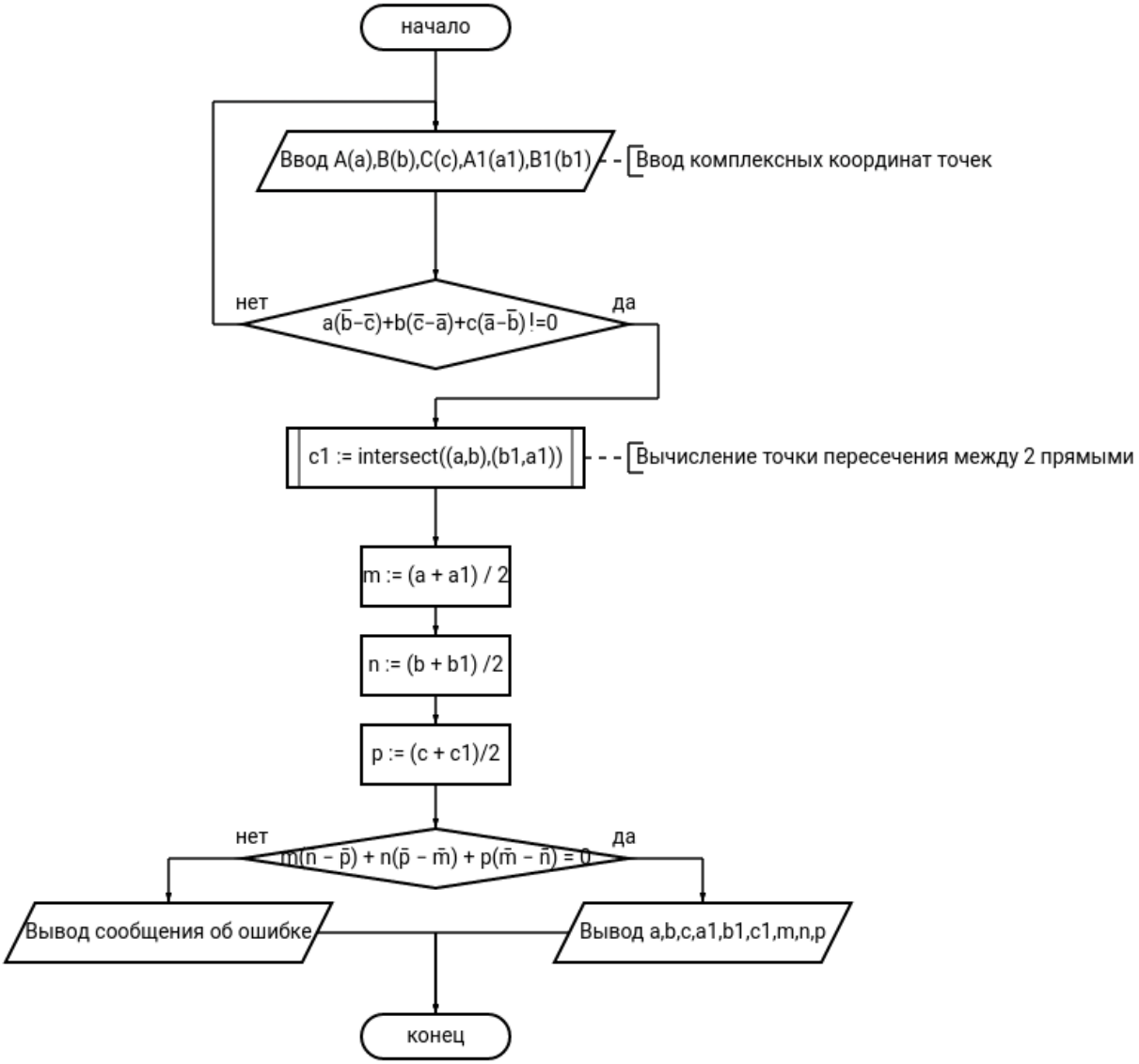

Рис. 9: Блок-схема алгоритма программы

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):

```
inline 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 4: Функция task5::solve

Демонстрация работы: Здесь будут скриншоты работы. TODO

Задача 6

Постановка задачи: Докажите, что диагонали вписанного в окружность четырёхугольника перпендикулярны, тогда и только тогда, когда сумма квадратов его противоположных сторон равна квадрату диаметра описанной окружности.

Решение задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

Демонстрация работы: TODO

Задача 7

Постановка задачи: Докажите, что если средние линии четырёхугольника равны, то его диагонали перпендикулярны, и обратно.

Решение задачи:

Алгоритм программного решения частного случая задачи:

Программная реализация задачи:

Демонстрация работы: TODO

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 "task4.cpp"
#include "task5.cpp"
#include "task6.cpp"
#include "task7.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 2:
            printTaskBegin(options, choice);
            task2::solve(returnCode, options);
            printTaskEnd(options);
            break;
        case 3:
            printTaskBegin(options, choice);
            task3::solve(returnCode, options);
            printTaskEnd(options);
            break;
        case 4:
            printTaskBegin(options, choice);
            task4::solve(returnCode, options);
            printTaskEnd(options);
            break;
        case 5:
            printTaskBegin(options, choice);
            task5::solve(returnCode, options);
            printTaskEnd(options);
            break;
```

```
            case 6:
               printTaskBegin(options, choice);
               task6::solve(returnCode, options);
               printTaskEnd(options);
               break;
            case 7:
               printTaskBegin(options, choice);
               task7::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"
#include "Circle.hpp"
#include "ComplexNumber.hpp"
#include "LineSegment.hpp"
#include "functions.hpp"
inline void task1::solve(int& returnCode, const ProgramOptions& options)
{
   /**
    * @brief Numbers should differs lesser than comparePrecision
    */
   const double comparePrecision = 0.1;
   /**
    * @brief Amount digits after decimal separator
    */
   const int8_t roundPrecision = 2;
   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::round({ circle.center() }, roundPrecision);
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::round(d, roundPrecision);
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'
const double lhs = power(sCD.length(), 2);
const double rhs = power(circle.radius(), 2) + power(sAC.length(), 2) +
```

```
                    power(sBC.length(), 2) - power(sAB.length(), 2);
    if (areEqual(lhs, rhs, comparePrecision)) {
        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:

```
#include "Circle.hpp"
#include "CircleArc.hpp"
#include "ComplexNumber.hpp"
#include "LineSegment.hpp"
#include "functions.hpp"
#include "tfunctions.hpp"
inline void task2::solve(int& returnCode, const ProgramOptions& options)
{
    /**
     * @brief Compare with precision up to comparePrecision digits after decimal
     * separator.
     */
    const uint8_t comparePrecision = 1;
    const int numbersCount = 5;
    const uint32_t circleRadius = 1;
    const bool useApproximation = true;
    ComplexNumber numbers[numbersCount];
    const std::string labels[numbersCount] { "A", "B", "N", "T", "M" };
    // clang-format off
    // References for readability
    ComplexNumber &a = numbers[0],
                  &b = numbers[1],
                  &n = numbers[2],
                  &t = numbers[3],
                  &m = numbers[4];
    // clang-format on
    const Circle circle { Point::zero(), circleRadius };
    returnCode = 0;
    do {
        readNumbersFromUser(numbers, labels, options, returnCode);
        if (returnCode != EXIT_SUCCESS)
            return;
```

```
    } while (!circle.isBelongs(static_cast< Point >(a), comparePrecision) ||
             !circle.isBelongs(static_cast< Point >(b), comparePrecision) ||
             !circle.isBelongs(static_cast< Point >(n), comparePrecision));
    m = CircleArc(circle,
                  static_cast< Point >(a),
                  static_cast< Point >(b),
                  static_cast< Point >(t),
                  useApproximation)
          .middle();
    const LineSegment AM { static_cast< Point >(a), static_cast< Point >(m) },
      MN { static_cast< Point >(m), static_cast< Point >(n) },
      AN { static_cast< Point >(a), static_cast< Point >(n) },
      BN { static_cast< Point >(b), static_cast< Point >(n) };
    // Check '|AM^2 - MN^2| = AN * BN'
    if (areEqual(std::abs(power(AM.length(), 2) - power(MN.length(), 2)),
                 AN.length() * BN.length(),
                 static_cast< int8_t >(comparePrecision))) {
       printMessage(options, "Computed coordinates:\n");
       printNumbers(options, numbers, labels, numbersCount);
    } else
       std::cerr << "|AM^2 - MN^2| != AN * BN.\n";
}
```

task3.cpp:

```
#include "LineSegment.hpp"
#include "functions.hpp"
#include "tfunctions.hpp"
inline void task3::solve(int& returnCode, const ProgramOptions& options)
{
    /**
     * @brief Compare with precision up to comparePrecision digits after decimal
     * separator.
     */
    const uint8_t comparePrecision = 1;
    const int numbersCount = 4;
    ComplexNumber numbers[numbersCount];
    const std::string labels[numbersCount] { "A", "B", "C", "D" };
    // clang-format off
    // References for readability
    ComplexNumber &a = numbers[0],
                  &b = numbers[1],
```

```
                &c = numbers[2],
                &d = numbers[3];
    // clang-format on
    returnCode = 0;
    task1::readTriangleFromUser(numbers, labels, options, returnCode);
    if (returnCode != EXIT_SUCCESS)
        return;
    const LineSegment AB { static_cast< Point >(a), static_cast< Point >(b) },
      AC { static_cast< Point >(a), static_cast< Point >(c) },
      BC { static_cast< Point >(b), static_cast< Point >(c) };
    const std::pair< Point, Point >& pair = AC.move(AB).getEndpoints();
    d = (pair.first == static_cast< Point >(b)) ? pair.second : pair.first;
    const LineSegment BD { static_cast< Point >(b), static_cast< Point >(d) },
      CD { static_cast< Point >(c), static_cast< Point >(d) },
      AD { static_cast< Point >(a), static_cast< Point >(d) };
    // Check 'AD^2 + BC^2 = AB^2 + CD^2 + BD^2 + AC^2'
    if (areEqual(power(AD.length(), 2) + power(BC.length(), 2),
                 power(AB.length(), 2) + power(CD.length(), 2) +
                   power(BD.length(), 2) + power(AC.length(), 2),
                 static_cast< int8_t >(comparePrecision))) {
        printMessage(options, "Computed coordinates:\n");
        printNumbers(options, numbers, labels, numbersCount);
    } else
        std::cerr << "AD^2 + BC^2 != AB^2 + CD^2 + BD^2 + AC^2.\n";
}
```

task4.cpp:

```
#include "tfunctions.hpp"
inline void task4::solve(int& returnCode, const ProgramOptions& options) {}
```

task5.cpp:

```
#include "ComplexNumber.hpp"
#include "Line.hpp"
#include "tfunctions.hpp"
inline 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";
}
```

task6.cpp:

```
#include "tfunctions.hpp"
inline void task6::solve(int& returnCode, const ProgramOptions& options) {}
```

task7.cpp:

```
#include "tfunctions.hpp"
inline void task7::solve(int& returnCode, const ProgramOptions& options) {}
```

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 task2 {
    void readNumbersFromUser(ComplexNumber arr[4], const std::string labels[4],
                             const ProgramOptions& options, int& returnCode);
    void solve(int& returnCode, const ProgramOptions& options);
}
namespace task3 {
    void solve(int& returnCode, const ProgramOptions& options);
}
namespace task4 {
    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);
}
namespace task6 {
    void solve(int& returnCode, const ProgramOptions& options);
}
namespace task7 {
    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::round(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 task2::readNumbersFromUser(ComplexNumber arr[4],
                                const std::string labels[4],
                                const ProgramOptions& options, int& returnCode)
{
   returnCode = 0;
   const size_t labelsCount = 4;
   // clang-format off
   ComplexNumber &a = arr[0],
                 &b = arr[1];
   // clang-format on
   bool isEqual = true;
   while (isEqual) {
      printMessage(
        options,
        "Enter coordinates of a,b,n,t points (must conform x^2+y^2 = 1, t between a
           and b):\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::round(arr[i], 2);
        }
        isEqual = a == b;
        if (isEqual)
            std::cerr << "Incorrect a,b. Must be not equal\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::round(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 COURSEWORK_4_1_COMPLEXNUMBER_HPP
#define COURSEWORK_4_1_COMPLEXNUMBER_HPP
#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 round(const ComplexNumber& number, int8_t ulp);
    const double& Re() const { return _real; }
    const double& Im() const { return _imaginary; }
    static ComplexNumber zero() { return ComplexNumber(0, 0); }
};
#endif // COURSEWORK_4_1_COMPLEXNUMBER_HPP
```

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

Line.hpp:

```
#ifndef COURSEWORK_4_1_LINE_HPP
#define COURSEWORK_4_1_LINE_HPP
#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 // COURSEWORK_4_1_LINE_HPP
```

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, should be > 0
     */
    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
```

Point.hpp:

```
#ifndef COURSEWORK_4_1_POINT_HPP
#define COURSEWORK_4_1_POINT_HPP
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); }
   Point operator-(const Point& other) const
   {
      return Point(_x - other._x, _y - other._y);
   }
   Point operator+(const Point& other) const
   {
      return Point(_x + other._x, _y + other._y);
   }
     const double& X() const
   {
      return _x;
   }
   const double& Y() const { return _y; }
   static Point middle(const Point& a, const Point& b)
   {
      return Point((a._x + b._x) / 2, (a._y + b._y) / 2);
   }
   static Point zero() { return Point { 0, 0 }; }
   /**
    * @brief Does check is point has one or both coordinates at infinity
    *
    * @param point A point that need to check
    */
   static bool isAtInfinity(const Point& point);
};
#endif // COURSEWORK_4_1_POINT_HPP
```

Point.cpp:

```
#include "Point.hpp"
#include <cmath>
bool Point::isAtInfinity(const Point& point)
{
   return std::isinf(point._x) || std::isinf(point._y);
}
```

Circle.hpp:

```
#ifndef COURSEWORK_4_1_CIRCLE_HPP
#define COURSEWORK_4_1_CIRCLE_HPP
#include "Angle.hpp"
```

```
#include "Point.hpp"
#include <cstdint>
#include <utility>
class Circle
{
  private:
   Point _center;
   double _radius;
  public:
   enum ApproximationMethod
   {
      BY_X_AXIS, // Point's X coordinate should be remain untouched
      BY_Y_AXIS // Point's Y coordinate should be remain untouched
   };
   Circle(const Point& center, double radius);
   Circle(const Point& a, const Point& b, const Point& c);
   Point center() const { return _center; }
   const double& radius() const { return _radius; }
   std::pair< double, double > y(double x) const;
   std::pair< double, double > x(double y) const;
   bool isBelongs(const Point& a) const;
   /**
    * @brief Does check is Point belongs to this circle with precision up to dds
    * digits after decimal separator
    *
    * @param a A point that need to check
    * @param dds Compare precision - digits after decimal separator
    */
   bool isBelongs(const Point& a, int8_t dds) const;
   /**
    * @brief Get the angle of the point.
    *
    * @param a Point. Should belongs to this circle, otherwise angle will be
    * calculated approximately
    * @return Angle of the point
    */
   Angle getAngle(const Point& a) const;
   /**
    * @brief Get the point by its angle.
    *
    * @param a Angle
```

```
     * @return Point defined by angle
     */
    Point getPoint(const Angle& a) const;
    /**
     * @brief Get the exact point, that belongs to this circle
     *
     * @param a A point that need to clarify
     * @param m approximation method
     * @return Point - resulted point, that belongs to this circle
     */
    Point getExactPoint(const Point& a, ApproximationMethod m = BY_Y_AXIS) const;
    ~Circle();
};
#endif // COURSEWORK_4_1_CIRCLE_HPP
```

Circle.cpp:

```
#include "Circle.hpp"
#include "Line.hpp"
#include "functions.hpp"
#include <cmath>
#include <exception>
Circle::Circle(const Point& center, double radius)
{
    bool isValidInput = !std::isinf(radius) && !Point::isAtInfinity(center);
    if (!isValidInput)
        throw std::runtime_error(
          "Cannot construct the circle for infinity radius or center");
    _center = center;
    _radius = radius;
}
Circle::Circle(const Point& a, const Point& b, const Point& c)
{
    if (Line(a, b).isBelongs(c))
        throw std::runtime_error(
          "Cannot construct the circle by 3 points. They belongs same line");
    /**
     * @brief Calculted coordinates of circle _center
     * @see https://en.wikipedia.org/wiki/Ellipse#Circles
     */
    double X, Y;
    const double &aX = a.X(), &aY = a.Y(), &bX = b.X(), &bY = b.Y(), &cX = c.X(),
```

```
                &cY = c.Y();
   const double denominator =
     (aX * (bY - cY) + bX * (cY - aY) + cX * (aY - bY));
   X = (-1.0 / 2.0) *
       (aY * ((power(bX, 2) + power(bY, 2)) - (power(cX, 2) + power(cY, 2))) +
        bY * ((power(cX, 2) + power(cY, 2)) - (power(aX, 2) + power(aY, 2))) +
        cY * ((power(aX, 2) + power(aY, 2)) - (power(bX, 2) + power(bY, 2)))) /
       denominator;
   Y = (1.0 / 2.0) *
       (aX * ((power(bX, 2) + power(bY, 2)) - (power(cX, 2) + power(cY, 2))) +
        bX * ((power(cX, 2) + power(cY, 2)) - (power(aX, 2) + power(aY, 2))) +
        cX * ((power(aX, 2) + power(aY, 2)) - (power(bX, 2) + power(bY, 2)))) /
       denominator;
   _center = Point(X, Y);
   _radius = sqrt(power(aX - _center.X(), 2) + power(aY - _center.Y(), 2));
}
bool Circle::isBelongs(const Point& a) const
{
   return almost_equal(power(a.X() - _center.X(), 2) +
                       power(a.Y() - _center.Y(), 2),
                     power(_radius, 2));
}
bool Circle::isBelongs(const Point& a, int8_t dds) const
{
   if (dds < 0)
      throw std::invalid_argument(
        "Circle::isBelongs: cannot set negative precision");
   return round(power(a.X() - _center.X(), 2) + power(a.Y() - _center.Y(), 2),
                dds) == round(power(_radius, 2), dds);
}
std::pair< double, double > Circle::y(double x) const
{
   const double value =
     std::sqrt(power(_radius, 2) - power(x - _center.X(), 2));
   return { _center.Y() + value, _center.Y() - value };
}
std::pair< double, double > Circle::x(double y) const
{
   const double value =
     std::sqrt(power(_radius, 2) - power(y - _center.Y(), 2));
   return { _center.X() + value, _center.X() - value };
```

```
}
Angle Circle::getAngle(const Point& a) const
{
   Point exact { a };
   if (!this->isBelongs(exact))
      exact = getExactPoint(exact);
   Line l(_center, exact);
   switch (l.getType()) {
      case LineType::CONST_X:
         return (a.Y() > _center.Y()) ? Angle(90.0) : Angle(270.0);
      case LineType::CONST_Y:
         return (a.X() > _center.X()) ? Angle(0.0) : Angle(180.0);
      case LineType::NORMAL: {
         double degree = std::atan(l.K()) * 180 / M_PI;
         if (a.X() < _center.X())
            degree = 360.0 - degree;
         if (degree < 0)
            degree = 360.0 + degree;
         return Angle(degree);
      }
      default:
         break;
   }
   throw std::runtime_error("Cannot get angle of the point on circle");
}
Point Circle::getPoint(const Angle& a) const
{
   /**
    * @brief Convert degrees in radians and getting not scaled X
    */
   const double xRaw = std::cos(a.degrees() * M_PI / 180.0);
   const double x = xRaw * _radius + _center.X();
   auto yValues = y(x);
   const double y = (a.degrees() > 180.0)
                    ? std::min(yValues.first, yValues.second)
                    : std::max(yValues.first, yValues.second);
   return Point(x, y);
}
Point Circle::getExactPoint(const Point& a, ApproximationMethod m) const
{
   switch (m) {
```

```
      case BY_X_AXIS: {
         auto yValues = y(a.X());
         const double y = (std::fabs(yValues.first - a.X()) <
                           std::fabs(yValues.second - a.X()))
                            ? yValues.first
                            : yValues.second;
         return Point(a.X(), y);
      }
      case BY_Y_AXIS: {
         auto xValues = x(a.Y());
         const double x = (std::fabs(xValues.first - a.X()) <
                           std::fabs(xValues.second - a.X()))
                            ? xValues.first
                            : xValues.second;
         return Point(x, a.Y());
      }
      default:
         break;
   }
   return Point::zero();
}
Circle::~Circle()
{
   _center.~Point();
}
```

CircleArc.hpp:

```
#ifndef COURSEWORK_4_1_CIRCLEARC_HPP
#define COURSEWORK_4_1_CIRCLEARC_HPP
#include "Angle.hpp"
#include "Circle.hpp"
class CircleArc
{
  private:
   const Circle& _circle;
   /**
    * @brief Describes boundaries info:
    * 0 - left angle
    * 1 - right angle
    */
```

```
    Angle _boundaries[2];
    /**
     * @brief Does finish initialization by 3 points. The _circle must be defined
     * before call.
     */
    void initByExactValues(const Point& a, const Point& b,
                           const Point& betweenAB);
  public:
    /**
     * @brief Construct a new circle arc by source circle and 3 points. All
     * points should to be at the circle and not equal each other.
     *
     * @param circle Source circle
     * @param a Begin point of the arc
     * @param b End point of the arc
     * @param betweenAB A point between a,b. Needed for unambiguous
     * constructing resulted arc.
     */
    CircleArc(const Circle& circle, const Point& a, const Point& b,
              const Point& betweenAB);
    /**
     * @param approximate Construct circle arc by approximate points. Exact
     * values will be given from the circle.
     */
    CircleArc(const Circle& circle, const Point& a, const Point& b,
              const Point& betweenAB, bool approximate);
    /**
     * @brief Does check is Point belongs to this circle arc
     *
     * @param point A point that need to check
     */
    bool isBelongs(const Point& point) const;
    Point middle() const;
    ~CircleArc();
};
#endif // COURSEWORK_4_1_CIRCLEARC_HPP
```

CircleArc.cpp:

```
#include "CircleArc.hpp"
#include "functions.hpp"
#include <cmath>
```

```
#include <stdexcept>
void CircleArc::initByExactValues(const Point& a, const Point& b,
                                  const Point& betweenAB)
{
   bool isValidInput = _circle.isBelongs(a) && _circle.isBelongs(b) &&
                       _circle.isBelongs(betweenAB) && a != b && b != betweenAB;
   if (!isValidInput)
      throw std::invalid_argument(
        "CircleArc: Cannot construct. One or more points not at the circle or points
           equal");
   Angle angles[3] { _circle.getAngle(a),
                     _circle.getAngle(b),
                     _circle.getAngle(betweenAB) };
   const Angle& min = std::min(angles[0], angles[1]);
   const Angle& max = (min == angles[0]) ? angles[1] : angles[0];
   const Angle& between = angles[2];
   if (between > max) {
      _boundaries[0] = min;
      _boundaries[1] = max;
   } else {
      _boundaries[0] = max;
      _boundaries[1] = min;
   }
}
CircleArc::CircleArc(const Circle& circle, const Point& a, const Point& b,
                     const Point& betweenAB) :
  _circle(circle)
{
   initByExactValues(a, b, betweenAB);
}
CircleArc::CircleArc(const Circle& circle, const Point& a, const Point& b,
                     const Point& betweenAB, bool approximate) :
  _circle(circle)
{
   if (approximate) {
      initByExactValues(_circle.getExactPoint(a),
                        _circle.getExactPoint(b),
                        _circle.getExactPoint(betweenAB));
   } else
      initByExactValues(a, b, betweenAB);
}
```

```
bool CircleArc::isBelongs(const Point& point) const
{
   Point tmp = point;
   if (!_circle.isBelongs(tmp))
      tmp = _circle.getExactPoint(tmp);
   const Angle &left = _boundaries[0], &right = _boundaries[1];
   const Angle& input = _circle.getAngle(point);
   if (left <= input && input <= right)
      return true;
   else
      return false;
   throw std::runtime_error(
     "CircleArc::isBelongs: passed invalid point, we tried to fix it but no
        success.");
}
Point CircleArc::middle() const
{
   const Angle &left = _boundaries[0], &right = _boundaries[1];
   double degrees;
   if (left < right) {
      degrees = (left + right).degrees() / 2;
      return _circle.getPoint(Angle(degrees));
   } else if (left > right) {
      degrees = (left + (Angle::fullAngle() - left + right) / 2).degrees();
      return _circle.getPoint(degrees);
   } else {
      return _circle.getPoint(left);
   }
   throw std::runtime_error(
     "CircleArc::middle: cannot construct middle. This is bug.");
}
CircleArc::~CircleArc()
{
   _boundaries[0].~Angle();
   _boundaries[1].~Angle();
}
```

LineSegment.hpp:

```
#ifndef COURSEWORK_4_1_LINESEGMENT_HPP
#define COURSEWORK_4_1_LINESEGMENT_HPP
```

```
#include "Line.hpp"
class LineSegment
{
  private:
   Line _line;
   Point _endpoints[2];
  public:
   LineSegment(const Point& a, const Point& b);
   LineSegment(const Line& l, const Point endpoints[2]);
   std::pair< Point, Point > getEndpoints() const;
   double length() const;
   /**
    * @brief Move this line segment along the specified segment
    *
    * @param other The line segment along which the segment will move. One of it
    * endpoints must be enpoint of this segment
    */
   LineSegment move(const LineSegment& other) const;
   ~LineSegment();
};
#endif // COURSEWORK_4_1_LINESEGMENT_HPP
```

LineSegment.cpp:

```
#include "LineSegment.hpp"
#include "functions.hpp"
#include "tfunctions.hpp"
#include <cmath>
LineSegment::LineSegment(const Point& a, const Point& b) :
  _line(a, b), _endpoints { a, b }
{
}
LineSegment::LineSegment(const Line& l, const Point endpoints[2]) :
  _line(l), _endpoints { endpoints[0], endpoints[1] }
{
   bool isPointsCorrect =
     l.isBelongs(endpoints[0]) && l.isBelongs(endpoints[1]);
   if (!isPointsCorrect)
      throw std::runtime_error(
        "Cannot construct line segment: endpoints not on soource line");
}
std::pair< Point, Point > LineSegment::getEndpoints() const
```

```
{
   return std::pair< Point, Point > { _endpoints[0], _endpoints[1] };
}
double LineSegment::length() const
{
   // ((a_x-b_x)^2+(a_y-b_y)^2)^(1/2)
   return sqrt(power(_endpoints[1].X() - _endpoints[0].X(), 2) +
               power(_endpoints[1].Y() - _endpoints[0].Y(), 2));
}
LineSegment LineSegment::move(const LineSegment& other) const
{
   const std::pair< Point, Point >& endpoints = other.getEndpoints();
   const Point otherPoints[2] { endpoints.first, endpoints.second };
   uint8_t thisIdx = 0, otherIdx = 0;
   if (_endpoints[0] == endpoints.first) {
      otherIdx = 1;
   } else if (_endpoints[1] == endpoints.first) {
      otherIdx = 1;
      thisIdx = 1;
   } else if (_endpoints[1] == endpoints.second) {
      otherIdx = 0;
      thisIdx = 1;
   } else
      throw std::invalid_argument(
        "LineSegment::move: one of argument endpoints must be enpoint of this
            segment");
   double dx, dy;
   dx = otherPoints[otherIdx].X() - _endpoints[thisIdx].X();
   dy = otherPoints[otherIdx].Y() - _endpoints[thisIdx].Y();
   Point movePoint(dx, dy);
   return LineSegment(_endpoints[0] + movePoint, _endpoints[1] + movePoint);
}
LineSegment::~LineSegment()
{
   /**
    * @brief Cleanup of points and line
    */
   _line.~Line();
   _endpoints[0].~Point();
   _endpoints[1].~Point();
}
```

Angle.hpp:

```
#ifndef COURSEWORK_4_1_ANGLE_HPP
#define COURSEWORK_4_1_ANGLE_HPP
class Angle
{
  private:
   /**
    * @brief Value in degrees. Must be between 0 and 360
    */
   double _degrees;
  public:
   /**
    * @brief Construct a new Angle object by degrees
    *
    * @param value an angle in degrees
    */
   Angle(double value = 0.0);
   bool operator==(const Angle& other) const;
   bool operator!=(const Angle& other) const;
   bool operator>(const Angle& other) const;
   bool operator<(const Angle& other) const;
   bool operator<=(const Angle& other) const;
   bool operator>=(const Angle& other) const;
   Angle operator+(const Angle& other) const;
   Angle operator-(const Angle& other) const;
   /**
    * @brief Divide angle value to number
    *
    * @param value Number that will divides this angle
    */
   Angle operator/(const double& number) const;
   /**
    * @brief Return value of this angle in degrees
    */
   double degrees() const { return _degrees; }
   static Angle fullAngle();
};
#endif // COURSEWORK_4_1_ANGLE _HPP
```

Angle.cpp:

```
#include "Angle.hpp"
#include <cmath>
#include <stdexcept>
Angle::Angle(double value)
{
   if (0.0 <= value && value <= 360.0 && !std::isinf(value))
      _degrees = value;
   else
      throw std::runtime_error(
        "Cannot construct Angle by incorrect degrees value (" +
        std::to_string(value) + ")");
}
bool Angle::operator==(const Angle& other) const
{
   return _degrees == other._degrees;
}
bool Angle::operator!=(const Angle& other) const
{
   return !(*this == other);
}
bool Angle::operator>(const Angle& other) const
{
   return _degrees > other._degrees;
}
bool Angle::operator<(const Angle& other) const
{
   return _degrees < other._degrees;
}
bool Angle::operator<=(const Angle& other) const
{
   return *this == other || *this < other;
}
bool Angle::operator>=(const Angle& other) const
{
   return *this == other || *this > other;
}
Angle Angle::operator+(const Angle& other) const
{
   double result = _degrees + other._degrees;
   if (result > 360.0) {
      result = std::fmod(result, fullAngle()._degrees);
```

```
    }
    return Angle(result);
}
Angle Angle::operator-(const Angle& other) const
{
    return Angle(_degrees - other._degrees);
}
Angle Angle::operator/(const double& number) const
{
    return Angle(_degrees / number);
}
Angle Angle::fullAngle()
{
    return Angle(360.0);
}
```

functions.hpp:

```
#ifndef CPPLIB_FUNCTIONS_HPP
#define CPPLIB_FUNCTIONS_HPP
#include <cmath>
#include <iostream>
#include <limits>
int getRandomNumber(int from, int to);
/**
 * @brief Round double to specified digits after decimal separator
 *
 * @param number source number
 * @param dds amount digits after decimal separator. Should be positive
 * @return double - Rounded number
 */
double round(double number, int8_t dds = 0);
/**
 * @brief Compare 2 numbers with specified precision
 *
 * @param a first number
 * @param b second number
 * @param dds Compare precision - digits after decimal separator. Should be
 * positive
 * @return true if the numbers are equal, false otherwise
 */
```

```
bool areEqual(double a, double b, int8_t dds = 0);
/**
 * @brief Compare 2 numbers with specified precision
 *
 * @param a first number
 * @param b second number
 * @param precision Compare precision. Should be positive
 * @return true if the numbers are differs lesser than specified precision
 * value, false otherwise
 */
bool areEqual(double a, double b, double precision = 0.01);
template< class T >
T power(T a, uint power)
{
   if (power == 0)
      return 1;
   T result = a;
   while (power != 1) {
      result *= a;
      --power;
   }
   return result;
}
template< class T >
typename std::enable_if< !std::numeric_limits< T >::is_integer, bool >::type
    almost_equal(
  T x, T y, int ulp = 2)
{
   // 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();
}
int getNumberDigits(int number);
#endif // CPPLIB_FUNCTIONS_HPP
```

functions.cpp:

```
#include <chrono>
```

```
#include <cstring>
#include <random>
#include "functions.hpp"
int getRandomNumber(int from, int to)
{
   try {
      if (from > to)
         throw std::runtime_error(
           "Incorrect couple 'from - to' for generating random numbers");
   } catch (const std::runtime_error& err) {
      std::cerr << err.what() << '\n';
      exit(1);
   }
   unsigned int now = static_cast< unsigned >(
     std::chrono::high_resolution_clock::now().time_since_epoch().count() %
     10000);
   std::mt19937 engine(now);
   std::uniform_int_distribution< int > random(from, to);
   return random(engine);
}
double round(double number, int8_t dds)
{
   if (dds < 0)
      throw std::invalid_argument("round: dds should be positive.");
   uint mult = 1;
   while (dds != 0) {
      mult *= 10;
      --dds;
   }
   return std::round(number * mult) / static_cast< double >(mult);
}
bool areEqual(double a, double b, int8_t dds)
{
   if (dds < 0)
      throw std::invalid_argument("round: dds should be positive.");
   return round(a, dds) == round(b, dds);
}
bool areEqual(double a, double b, double precision)
{
   if (precision < 0.0)
      throw std::invalid_argument("round: dds should be positive.");
```

```
   return std::fabs(a - b) <= precision;
}
int getNumberDigits(int number)
{
   int numberDigits = 0;
   while (number != 0) {
      ++numberDigits;
      number /= 10;
   }
   return numberDigits;
}
```