

МИНОБРНАУКИ РОССИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»  
(ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механики

Кафедра математического обеспечения ЭВМ

**Реализация палитры команд с возможностью нечеткого  
поиска в приложениях, использующих Qt**

Магистерская диссертация

Направление 01.04.02 Прикладная математика и информатика

Профиль Название профиля

Зав. кафедрой \_\_\_\_\_ д.т.н., проф. Г.В. Абрамов \_\_\_\_\_. 2020 г.

Обучающийся \_\_\_\_\_ Д.В. Польшаков

Руководитель \_\_\_\_\_ к.ф.-м.н., доц. М.К. Чернышов

Воронеж 2020

# Список сокращения

ПО – программное обеспечение

ГИП – графический интерфейс пользователя

# Содержание

<b>Содержание</b> . . . . .	3
<b>Введение</b> . . . . .	4
<b>1 Обзор существующих реализаций палитр команд</b> . . . . .	5
<b>2 Анализ задачи</b> . . . . .	7
2.1 Общий анализ . . . . .	7
2.2 Постановка задачи . . . . .	8
<b>3 Реализация</b> . . . . .	10
3.1 Средства реализации . . . . .	10
3.2 Требования к программному и аппаратному обеспечению . . .	10
<b>Заключение</b> . . . . .	12
<b>Список использованных источников</b> . . . . .	13
<b>Список использованных источников</b> . . . . .	13
<b>Приложения</b>	
<b>Приложение А Вспомогательные классы</b> . . . . .	15
<b>Приложение В Интерполирование кривой на плоскости</b> . . . . .	21
<b>Приложение С Интерполирование кривой на двумерной сфере</b> . .	30
<b>Приложение D Интерполирование кривой на ориентационной сфере</b> . . . . .	40

# Введение

С ростом возможностей персональных компьютеров растут и возможность программного обеспечения. Такие группы программ как графические редакторы, текстовые процессоры среды разработки и многое другое обрывают огромным числом функций. Для доступа к этим функциям используются элементы ГИП.

Для быстрого доступа к функциям приложения, используются горячие клавиши. Но обычно горячие клавиши создаются только для самых часто используемых команд, остальные же приходится выбирать вручную в интерфейсе.

Но как бы хорошо ни был разработан интерфейс, число функций может оказаться настолько большим, что появляется проблема с поиском нужного элемента управления. Кроме того, некоторые системы поддерживают возможность добавления сторонних модулей. В таком случае место расположения элементов регулируется сторонними разработчиками, а не авторами оригинального приложения.

Для упрощения поиска элементов была придумана технология «палитра команд». Это специальное окно в интерфейсе приложения, где отображаются все доступные функции. Иногда рядом с функцией отображается сочетание горячих клавиш, с помощью которого эта функция может быть вызвана. Это помогает пользователю легко запоминать новые сочетания, который он забыл или не знал раньше. В этом же окне есть поле для ввода поискового запроса.

Целью данной работы было реализовать механизм, который позволял бы добавлять палитру команд в сторонние приложения без их пересборки.

# Глава 1

## Обзор существующих реализаций палитр команд

Впервые палитра команд появилась 1 июля 2011 году в редакторе Sublime Text 2 [1]. Вслед за этим подобный функционал был реализован и в таких программах, как: Atom[2], VSCode[3], JupyterLab[4].

Но это были лишь единичные случаи. В апреле 2017 года появилась альфа версия приложения Plotinus[5], которое позволяет добавлять палитру команд в любое приложение, которое написано с использованием графической библиотеки GTK.

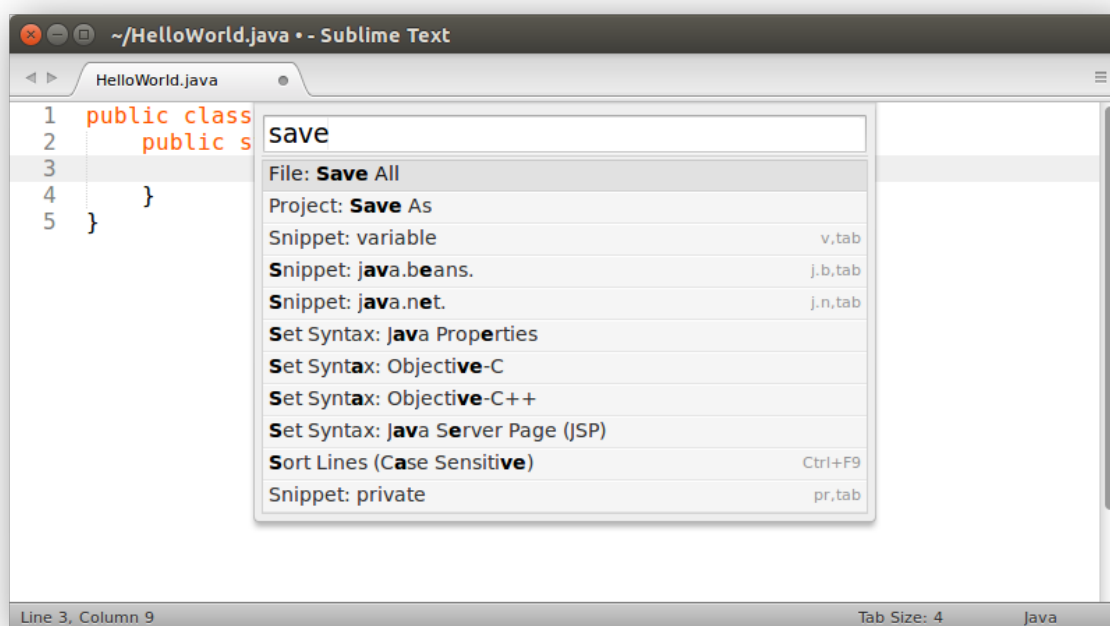


Рис. 1.1. Sublime Text

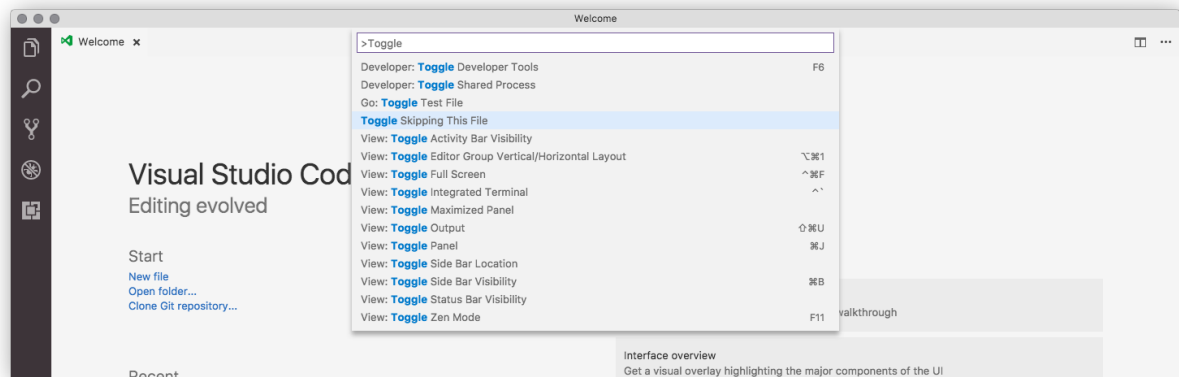


Рис. 1.2. VSCode

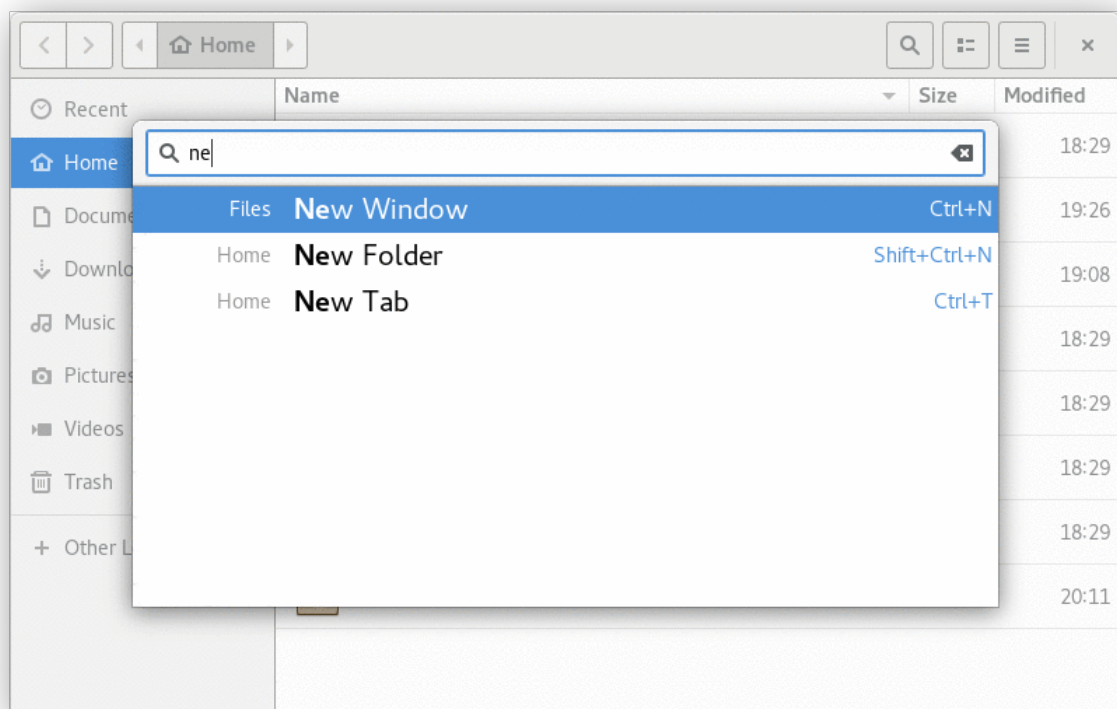


Рис. 1.3. Plotinus

## Глава 2

# Анализ задачи

### 2.1 Общий анализ

Система управления должна позволять контролировать множество приложений. Для её реализации лучше всего подходит клиент-серверная архитектура. Для каждого целевого приложения будет запускаться отдельный клиент, который будет заниматься сбором информации об элементах управления и передавать её на сервер.

В качестве сервера будет выступать приложение, которое запускает целевые приложения вместе с клиентами. После этого сервер принимает входящее соединение от клиента и отображающее окно поиска элемента для текущего активного окна.

Для удобной работы окно поиска должно отображаться окно поверх работающего приложения. И в нем должен производиться нечеткий поиск.

В операционной системе Linux существует две основных графических библиотеки: GTK и Qt. Библиотека GTK, начиная с третьей версии предоставляет возможность запускать приложения с дополнительными модулями, которые могут расширять функциональность приложения. Этим воспользовались разработчики библиотеки Plotinus, реализовав возможность добавления палитры команд в любое приложение, использующее GTK3.

### Реализация библиотеки

Для внедрения библиотеки требуется реализовать заглушки для функций Qt, которые будут вызывать специальный обработчик, а затем продолжать нормальное выполнение функции. Дополнительную сложность создает то, что библиотека Qt написана на языке C++, который из-за поддержки классов и перегрузок функций использует т.н. «искажение имен» (name mangling). Таким образом чтобы создать такой обработчик нужно специальным образом сконструировать имя функции исходя из имени класса, метода, набора параметров и возвращаемого значения.

Создание таких обработчиков является рутинной работой в которой человек легко может допустить ошибку. Поэтому вместо ручного написания каждого обработчика нужно написать генератор, который может добавить нужные обработчики имя минимальный и необходимый набор данных (имя класса, метода и т.д.).

## **Способ получения информации об элементах**

В общем случае Qt не предоставляет возможности добавить сторонний модуль в существующую программу, поэтому для этого придется использовать средство системы. А именно — инъекцию динамической библиотеки.

В операционной системе Linux есть специальная переменная окружения, которая говорит загрузчику, чтобы определенная библиотека была загружена раньше других. Таким образом операционная система в первую очередь загружает указанную библиотеку, что передает вызывающей программе указатели на функции загруженной библиотеки. Такой механизм позволяет регистрировать вызовы различных функций.

Для получения информации об элементах интерфейса можно загрузить специальную библиотеку, которая будет регистрировать создания, изменения и удаления элементов интерфейса. Затем собранная информация будет передавать на сервер для последующей работы.

Пользовательский интерфейс через специальное API получает от сервера информацию о доступных элементах в текущем приложении. После того, как пользователь произвел выбор, вызывается специальная функция на стороне сервера, которая приводит к отправке команды клиенту.

Клиент на своей стороне реализует возможность активации элемента интерфейса (нажатие кнопки, выбор меню и т.п.).

## **2.2 Постановка задачи**

Разработать набор программ, которые в комплексе будут решать следующие задачи:

- запускать целевые приложения в специальном окружении;
- собирать информацию о существующих элементах графического приложения;
- сохранять информацию о всех запущенных приложениях;
- отображать пользователю окно для поиска и выбора элемента;
- активировать выбранный пользователем элемент.



Этот набор программ должен быть реализован в виде следующих элементов:

1. Библиотека для инъекции и сбора информации в конкретном приложении.
2. Приложение для сохранения информации, полученной из нескольких приложений с библиотекой из п.1.
3. Графический интерфейс для запуска приложений.

## Глава 3

# Реализация

### 3.1 Средства реализации

- Язык программирования C для написания внедряемой библиотеки;
- Язык программирования Python для написания графического интерфейса и генератора кода;
- Система сборки CMake.

В данной работе используются два языка программирования по следующим причинам:

- Язык C предоставляет низкоуровневый интерфейс, который позволяет переопределить нужные функции во внедряемой библиотеке и в то же время без особой сложности в нем можно реализовать передачу данных в сокет.
- Язык Python, напротив, позволяет писать высокоуровневый код, что упрощает разработку прикладных приложений (в частности приложений с графическим интерфейсом).

### 3.2 Требования к программному и аппаратному обеспечению

Приложение предназначено для использования на IBM PC-совместимых компьютерах с операционной системой Linux.

Работы приложения требуются:

- аппаратное обеспечение согласно требованиям ОС;
- наличие интерпретатора Python;

Требования к целевому приложению:

- приложение должно использовать графическую библиотеку Qt;
- должна быть произведена динамическая линковка с данной библиотекой.

# Заключение

В данной работе была поставлена задача разработать комплекс программ, который позволил бы пользоваться палитрой команд в произвольных приложениях, разработанных с использованием графической библиотеке Qt.

В ходе работы были подробно рассмотрены способы решения проблемы с получением информации из существующего приложения

Был реализован комплекс приложений, который позволяет отображать палитру команд в произвольных Qt приложениях, опираясь на информацию, которую разработчик оригинального приложения связал с элементом интерфейса.

Как говорилось ранее, палитра команд является удобной частью интерфейса и начинает появляться все в большем числе приложений. Полученные в ходе данной работы результаты могут найти практическое применение в системах основанных на Linux.

# Список использованных источников

- [1] <https://www.sublimetext.com/2>
- [2] <https://atom.io/packages/command-palette>
- [3] [https://code.visualstudio.com/docs/getstarted/userinterface#\\_command-palette](https://code.visualstudio.com/docs/getstarted/userinterface#_command-palette)
- [4] <https://jupyterlab.readthedocs.io/en/stable/user/commands.html>
- [5] <https://github.com/p-e-w/plotinus/releases>
- [6] Побегайло А. П. Применение кватернионов в компьютерной геометрии и графике / А. П. Побегайло. — Минск : БГУ, 2010. — 216 с.
- [7] Роджерс Д. Алгоритмические основы машинной графики / Д. Роджерс. — М. : Мир, 1989. — 512 с.
- [8] Роджерс Д. Математические основы машинной графики / Д. Роджерс. — М. : Мир, 2001. — 604 с.
- [9] Перемитина Т. О. Компьютерная графика : учеб. пособие / Т. О. Перемитина. — Томск : Эль Контент, 2012. — 144 с.
- [10] Никулин Е. А. Компьютерная геометрия и алгоритмы машинной графики / Е. А. Никулин. — СПб. : БХВ-Санкт-Петербург, 2003. — 550 с.
- [11] Петров М. Н. Компьютерная графика : учебник для вузов / М. Н. Петров. — СПб. : Питер, 2011. — 544 с.
- [12] Порев В. Компьютерная графика : учеб. пособие / В. Порев. — СПб. : БХВ-Петербург, 2002. — 428 с.
- [13] Шикин Е. В. Компьютерная графика. Полигональные модели : учеб. пособие / Е. В. Шикин, А. В. Боресков. — М. : ДИАЛОГ-МИФИ, 2005. — 461 с.

- [14] Сиденко Л. А. Компьютерная графика и геометрическое моделирование : учеб. пособие / Л. А. Сиденко. — СПб. : Питер, 2010. — 224 с.
- [15] Джамбруно М. Трёхмерная графика и анимация / М. Джамбруно. — М. : Вильямс, 2002. — 638 с.
- [16] Ласло М. Вычислительная геометрия и компьютерная графика на C++ / М. Ласло. — М. : БИНОМ, 1997. — 301 с.
- [17] Хилл Ф. OpenGL : Программирование компьютерной графики / Ф. Хилл. — СПб. : Питер, 2002. — 1081 с.
- [18] OpenGL : руководство по программированию / М. Ву [и др.]. — СПб. : Питер, 2006. — 623 с.
- [19] Боресков А. В. Графика трёхмерной компьютерной игры на основе OpenGL / А. В. Боресков. — М. : Диалог-МИФИ, 2004. — 383 с.
- [20] Иванов А. О. Компьютерная геометрия / А. О. Иванов. — Москва : Интернет-Университет Информационных Технологий, 2010. — 388 с.
- [21] Куликов А. И. Алгоритмические основы современной компьютерной графики / А. И. Куликов, Т. Э. Овчинникова. — Москва : Интернет-Университет Информационных Технологий, 2007. — 195 с.
- [22] Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]. — Санкт-Петербург: Питер, 2001. — 368 с.
- [23] Java Platform Standard Edition 8 Documentation. — URL: <http://docs.oracle.com/javase/8/docs/> (дата обращения: 13.05.2018).
- [24] LWJGL — Lightweight Java Game Library Documentation. — URL: <https://javadoc.lwjgl.org/> (дата обращения: 13.05.2018).

# Приложение А

## Вспомогательные классы

### PolynomsCreator.java

```
package com.ran.engine.factories.interpolation.tools;

//
// ...

public class PolynomsCreator {

    private static final PolynomsCreator INSTANCE = new PolynomsCreator();

    public static PolynomsCreator getInstance() {
        return INSTANCE;
    }

    public DoubleFunction<SingleDouble> createBernsteinPolynom(int n, int m) {
        if (n < 0 || m < 0 || n < m) {
            throw new CreationException("Incorrect parameters while creating Bernstein polynom");
        }
        long combinations = countCombinations(n, m);
        return new DoubleFunction<>(point -> new SingleDouble(
            combinations * power(1 - point, n - m) * power(point, m)), 0.0, 1.0);
    }

    public DoubleFunction<SingleDouble> createSmoothingPolynom(int k) {
        if (k < 0) {
            throw new CreationException("Incorrect parameters while creating smoothing polynom");
        }
        List<DoubleFunction<SingleDouble>> bernsteinPolynoms = new ArrayList<>(k + 1);
        for (int i = k + 1; i <= 2 * k + 1; i++) {
            bernsteinPolynoms.add(createBernsteinPolynom(2 * k + 1, i));
        }
        return new DoubleFunction<>(point -> new SingleDouble(bernsteinPolynoms.stream()
            .mapToDouble(polynom -> polynom.apply(point).getValue()).sum()), 0.0, 1.0);
    }

    private long countCombinations(int n, int m) {
        if (n < 0 || m < 0 || n < m) {
            throw new CreationException("Incorrect parameters while combinations counting");
        }
        if (m * 2 < n) {
            m = n - m;
        }
    }
}
```

```

        long result = 1;
        for (int i = m + 1; i <= n; i++) {
            result *= i;
        }
        for (int i = 1; i <= n - m; i++) {
            result /= i;
        }
        return result;
    }

    private double power(double x, int power) {
        double result = 1.0;
        for (int i = 0; i < power; i++) {
            result *= x;
        }
        return result;
    }
}

```

## RotationCreator.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class RotationCreator {

    private static final RotationCreator INSTANCE = new RotationCreator();

    public static RotationCreator getInstance() {
        return INSTANCE;
    }

    public DoubleMatrix createRotation(ThreeDoubleVector axis, double angle) {
        double n1 = axis.getX();
        double n2 = axis.getY();
        double n3 = axis.getZ();

        double sin = Math.sin(angle);
        double cos = Math.cos(angle);
        double vers = 1.0 - cos;

        double[][] matrix = new double[][] {
            {n1 * n1 + (1.0 - n1 * n1) * cos,
             n1 * n2 * vers - n3 * sin,
             n1 * n3 * vers + n2 * sin      },
            {n2 * n1 * vers + n3 * sin,
             n2 * n2 + (1.0 - n2 * n2) * cos,
             n2 * n3 * vers - n1 * sin      },
            {n3 * n1 * vers - n2 * sin,
             n3 * n2 * vers + n1 * sin,
             n3 * n3 + (1.0 - n3 * n3) * cos}
        };
        return new DoubleMatrix(matrix);
    }

    public Pair<ThreeDoubleVector, Double> getAxisAndAngleForRotation(DoubleMatrix rotation) {

```



```

double r21Diff = rotation.get(1, 0) - rotation.get(0, 1);
double r13Diff = rotation.get(0, 2) - rotation.get(2, 0);
double r32Diff = rotation.get(2, 1) - rotation.get(1, 2);
double sin = 0.5 * Math.sqrt(r21Diff * r21Diff + r13Diff * r13Diff + r32Diff * r32Diff);
double cos = 0.5 * (rotation.get(0, 0) + rotation.get(1, 1) +
    rotation.get(2, 2) - 1.0);
double phi = Math.atan2(sin, cos);
ThreeDoubleVector axis;
if (ArithmeticOperations.doubleEquals(phi, 0.0) ||
    ArithmeticOperations.doubleEquals(phi, Math.PI)) {
    axis = new ThreeDoubleVector(
        Math.sqrt((rotation.get(0, 0) + 1.0) / 2.0),
        Math.sqrt((rotation.get(1, 1) + 1.0) / 2.0),
        Math.sqrt((rotation.get(2, 2) + 1.0) / 2.0)
    );
} else {
    axis = new ThreeDoubleVector(
        r32Diff / (2.0 * sin),
        r13Diff / (2.0 * sin),
        r21Diff / (2.0 * sin)
    );
}
return new Pair<>(axis, phi);
}

public DoubleMatrix createReversedRotationByRotation(DoubleMatrix rotation) {
    Pair<ThreeDoubleVector, Double> axisAndAngle = getAxisAndAngleForRotation(rotation);
    ThreeDoubleVector axis = axisAndAngle.getLeft();
    double angle = axisAndAngle.getRight();
    return createRotation(axis, -angle);
}
}

```

## CurvesDeformationCreator.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class CurvesDeformationCreator {

    private static final CurvesDeformationCreator INSTANCE = new CurvesDeformationCreator();

    public static CurvesDeformationCreator getInstance() {
        return INSTANCE;
    }

    public <T extends AlgebraicObject<T>> DoubleFunction<T> deformCurves(
        DoubleFunction<T> firstCurve, DoubleFunction<T> secondCurve, int degree) {
        return deformCurves(firstCurve, secondCurve, degree,
            GroupMultiplicationOperationFactory.getMultiplicationOperation());
    }

    public <T extends AlgebraicObject<T>> DoubleFunction<T> deformCurves(
        DoubleFunction<T> firstCurve, DoubleFunction<T> secondCurve, int degree,
        BiFunction<DoubleFunction<T>, DoubleFunction<T>, DoubleFunction<T>>
            groupMultiplicationOperation) {
    }
}

```

```

        if (!firstCurve.apply(0.0).equals(secondCurve.apply(0.0))) {
            throw new AlgebraicException("Start points of curves must coincide for curves " +
                "deformation");
        }
        DoubleFunction<SingleDouble> smoothingPolynom = PolynomsCreator.getInstance()
            .createSmoothingPolynom(degree);
        DoubleFunction<SingleDouble> tauMinus = new DoubleFunction<>() {
            point -> new SingleDouble((1.0 - smoothingPolynom.apply(point).getValue()) *
                point), 0.0, 1.0);
        };
        DoubleFunction<SingleDouble> tauPlus = new DoubleFunction<>() {
            point -> new SingleDouble(smoothingPolynom.apply(point).getValue() * point),
                0.0, 1.0);
        };
        return groupMultiplicationOperation.apply(secondCurve.superposition(tauPlus),
            firstCurve.superposition(tauMinus));
    }

    public <T extends AlgebraicObject<T>> DoubleFunction<T> deformCurvesWithCommonEnd(
        DoubleFunction<T> firstCurve, DoubleFunction<T> secondCurve, int degree) {
        return deformCurvesWithCommonEnd(firstCurve, secondCurve, degree,
            GroupMultiplicationOperationFactory.getMultiplicationOperation());
    }

    public <T extends AlgebraicObject<T>> DoubleFunction<T> deformCurvesWithCommonEnd(
        DoubleFunction<T> firstCurve, DoubleFunction<T> secondCurve, int degree,
        BiFunction<DoubleFunction<T>, DoubleFunction<T>, DoubleFunction<T>>
            groupMultiplicationOperation) {
        if (!firstCurve.apply(0.0).equals(secondCurve.apply(0.0))) {
            throw new AlgebraicException("Start points of curves must coincide for curves " +
                "deformation");
        }
        DoubleFunction<SingleDouble> smoothingPolynom = PolynomsCreator.getInstance()
            .createSmoothingPolynom(degree);
        DoubleFunction<SingleDouble> tauMinus = new DoubleFunction<>() {
            point -> new SingleDouble((1.0 - smoothingPolynom.apply(point).getValue()) *
                point), 0.0, 1.0);
        };
        DoubleFunction<SingleDouble> tauMinusReversed = tauMinus.reversed();
        DoubleFunction<SingleDouble> tauPlus = new DoubleFunction<>() {
            point -> new SingleDouble(smoothingPolynom.apply(point).getValue() * point),
                0.0, 1.0);
        };
        DoubleFunction<SingleDouble> tauPlusReversed = tauPlus.reversed();
        DoubleFunction<SingleDouble> tauPlusFixed = new DoubleFunction<SingleDouble>() {
            point -> new SingleDouble(1.0 - tauPlusReversed.apply(point).getValue()),
                0.0, 1.0);
        };
        return groupMultiplicationOperation.apply(secondCurve.superposition(tauMinusReversed),
            firstCurve.superposition(tauPlusFixed));
    }
}

```

## CurvesSmoothingCreator.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class CurvesSmoothingCreator {

    private static final CurvesSmoothingCreator INSTANCE = new CurvesSmoothingCreator();
}

```

```

public static CurvesSmoothingCreator getInstance() {
    return INSTANCE;
}

public <T extends AlgebraicObject<T>> DoubleFunction<T> smoothCurves(
    DoubleFunction<T> firstCurve, DoubleFunction<T> secondCurve, int degree) {
    return smoothCurves(firstCurve, secondCurve, degree,
        GroupMultiplicationOperationFactory.getMultiplicationOperation());
}

public <T extends AlgebraicObject<T>> DoubleFunction<T> smoothCurves(
    DoubleFunction<T> firstCurve, DoubleFunction<T> secondCurve, int degree,
    BiFunction<DoubleFunction<T>, DoubleFunction<T>, DoubleFunction<T>>
        groupMultiplicationOperation) {
    if (!firstCurve.apply(0.0).equals(secondCurve.apply(0.0))) {
        throw new AlgebraicException("Start points of curves must coincide for curves " +
            "deformation");
    }
    DoubleFunction<SingleDouble> smoothingPolynom = PolynomsCreator.getInstance()
        .createSmoothingPolynom(degree);
    DoubleFunction<SingleDouble> sigmaMinus = new DoubleFunction<>() {
        point -> new SingleDouble((1.0 - smoothingPolynom.apply(point).getValue()) *
            (1.0 - point)), 0.0, 1.0);
    DoubleFunction<SingleDouble> sigmaPlus = new DoubleFunction<>() {
        point -> new SingleDouble(smoothingPolynom.apply(point).getValue() * point,
            0.0, 1.0);
    return groupMultiplicationOperation.apply(secondCurve.superposition(sigmaPlus),
        firstCurve.superposition(sigmaMinus));
}
}

```

## AbstractInterpolatedCurveCreator.java

```

package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public abstract class AbstractInterpolatedCurveCreator<I, O extends AlgebraicObject<O>,
    P extends InputParameters> implements InterpolatedCurveCreator<I, O, P> {

    protected void validateVerticesList(List<I> verticesList) {
        if (verticesList.size() < 3) {
            throw new InterpolationException("Interpolation requires at least 3 vertices");
        }
    }

    protected DoubleFunction<TwoDoubleVector> buildFinalCurve(
        List<Double> timeMoments,
        List<DoubleFunction<TwoDoubleVector>> segments,
        int segmentsQuantity) {
        TimeMomentsUtil timeMomentsUtil = TimeMomentsUtil.getInstance();
        List<DoubleFunction<TwoDoubleVector>> curveSegments = new ArrayList<>(segmentsQuantity);
        for (int i = 0; i < segmentsQuantity; i++) {
            double startTime = timeMoments.get(i);
            double endTime = timeMoments.get(i + 1);
            DoubleFunction<TwoDoubleVector> currentSegment = segments.get(i);

```

```

        DoubleFunction<TwoDoubleVector> alignedCurveSegment =
            currentSegment.superposition(timeMomentsUtil.buildAligningFunction(
                startTime, endTime));
        curveSegments.add(alignedCurveSegment);
    }
    return DoubleMultifunction.makeMultifunction(curveSegments);
}

```

## Приложение В

# Интерполирование кривой на плоскости

### AbstractPlainCurveCreator.java

```
package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public abstract class AbstractPlainCurveCreator extends AbstractInterpolatedCurveCreator<
    TwoDoubleVector, TwoDoubleVector, SimpleInputParameters> {
}
```

### CircleArcsBuilder.java

```
package com.ran.engine.factories.interpolation.tools;

//
// ...

public class CircleArcsBuilder {

    private static CircleArcsBuilder INSTANCE = new CircleArcsBuilder();

    public static CircleArcsBuilder getInstance() {
        return INSTANCE;
    }

    public Result buildCircle(TwoDoubleVector firstPoint, TwoDoubleVector secondPoint,
        TwoDoubleVector thirdPoint) {
        if (LineEvaluator.arePointsOnOneLine(firstPoint, secondPoint, thirdPoint)) {
            SegmentsBuilder segmentsBuilder = SegmentsBuilder.getInstance();
            SegmentsBuilder.Result firstSegmentsResult = segmentsBuilder.buildSegment(
                firstPoint, secondPoint);
            SegmentsBuilder.Result secondSegmentsResult = segmentsBuilder.buildSegment(
                secondPoint, thirdPoint);
            return new Result(
                firstSegmentsResult.getSegment(), secondSegmentsResult.getSegment(),
                0.0, 0.0,

```

```

        firstSegmentsResult.getLength(), secondSegmentsResult.getLength()
    );
}

Line firstPerpendicular = LineEvaluator.evaluateMiddlePerpendicularLine(
    firstPoint, secondPoint);
Line secondPerpendicular = LineEvaluator.evaluateMiddlePerpendicularLine(
    secondPoint, thirdPoint);
TwoDoubleVector circleCenter = LineEvaluator.evaluateLinesIntersection(
    firstPerpendicular, secondPerpendicular);

TwoDoubleVector firstVector = firstPoint.subtract(circleCenter);
TwoDoubleVector secondVector = secondPoint.subtract(circleCenter);
TwoDoubleVector thirdVector = thirdPoint.subtract(circleCenter);

double firstAngle = VectorManipulator.countAngleBetweenVectors(
    firstVector, secondVector);
double secondAngle = VectorManipulator.countAngleBetweenVectors(
    secondVector, thirdVector);
if (firstAngle + secondAngle > Math.PI * 2.0) {
    firstAngle -= Math.PI * 2.0;
    secondAngle -= Math.PI * 2.0;
}

double radius = firstVector.getNorm();
double firstLength = Math.abs(firstAngle) * radius;
double secondLength = Math.abs(secondAngle) * radius;

DoubleFunction<TwoDoubleVector> firstArc =
    buildArc(circleCenter, firstVector, radius, firstAngle);
DoubleFunction<TwoDoubleVector> secondArc =
    buildArc(circleCenter, secondVector, radius, secondAngle);

return new Result(
    firstArc, secondArc,
    firstAngle, secondAngle,
    firstLength, secondLength
);
}

private DoubleFunction<TwoDoubleVector> buildArc(TwoDoubleVector center,
    TwoDoubleVector startVector,
    double radius, double angle) {
    double startAngle = VectorManipulator.countVectorAngle(startVector);
    return new DoubleFunction<>(point -> {
        double currentAngle = startAngle + angle * point;
        double x = Math.cos(currentAngle) * radius;
        double y = Math.sin(currentAngle) * radius;
        return new TwoDoubleVector(center.getX() + x, center.getY() + y);
    }, 0.0, 1.0);
}

public static class Result {
    private final DoubleFunction<TwoDoubleVector> firstArc;
    private final DoubleFunction<TwoDoubleVector> secondArc;
    private final double firstAngle;
    private final double secondAngle;
    private final double firstArcLength;
    private final double secondArcLength;
}

```

```

    //      , get-    set-
}
}

```

## PlaneByPointsCurveCreator.java

```

package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public class PlaneByPointsCurveCreator extends AbstractPlainCurveCreator {

    private static final PlaneByPointsCurveCreator INSTANCE =
        new PlaneByPointsCurveCreator();

    public static PlaneByPointsCurveCreator getInstance() {
        return INSTANCE;
    }

    @Override
    public DoubleFunction<TwoDoubleVector> interpolateCurve(
        List<TwoDoubleVector> vertices, SimpleInputParameters parameters, int degree) {
        validateVerticesList(vertices);

        double t0 = parameters.getT0();
        double t1 = parameters.getT1();
        int k = vertices.size();

        CurvesDeformationCreator deformationCreator = CurvesDeformationCreator.getInstance();
        CircleArcsBuilder circleArcsBuilder = CircleArcsBuilder.getInstance();
        TimeMomentsUtil timeMomentsUtil = TimeMomentsUtil.getInstance();

        List<DoubleFunction<TwoDoubleVector>> constantFunctions = new ArrayList<>(k);
        for (int i = 0; i < k; i++) {
            constantFunctions.add(DoubleFunction.createConstantFunction(vertices.get(i)));
        }

        List<DoubleFunction<TwoDoubleVector>> segments = new ArrayList<>(k - 1);
        List<Pair<Double, Double>> arcsLengths = new ArrayList<>(k - 2);
        CircleArcsBuilder.Result currentArcsBuildingResult =
            circleArcsBuilder.buildCircle(vertices.get(0), vertices.get(1), vertices.get(2));
        segments.add(currentArcsBuildingResult.getFirstArc());
        arcsLengths.add(currentArcsBuildingResult.getArcsLengths());

        for (int i = 1; i < k - 2; i++) {
            CircleArcsBuilder.Result nextArcsBuildingResult =
                circleArcsBuilder.buildCircle(vertices.get(i), vertices.get(i + 1),
                    vertices.get(i + 2));
            DoubleFunction<TwoDoubleVector> deformedFunction = deformationCreator.deformCurves(
                currentArcsBuildingResult.getSecondArc().substract(constantFunctions.get(i)),
                nextArcsBuildingResult.getFirstArc().substract(constantFunctions.get(i)),
                degree, GroupMultiplicationOperationFactory.getSummationOperation());
            segments.add(deformedFunction.add(constantFunctions.get(i)));
            arcsLengths.add(nextArcsBuildingResult.getArcsLengths());
            currentArcsBuildingResult = nextArcsBuildingResult;
        }
        segments.add(currentArcsBuildingResult.getSecondArc());
    }
}

```

```

        List<Double> timeMoments = timeMomentsUtil.countTimeMoments(arcsLengths, t0, t1, k);
        return buildFinalCurve(timeMoments, segments, k - 1);
    }
}

```

## SegmentsBuilder.java

```

package com.ran.engine.factories.interpolation.tools;

import com.ran.engine.algebra.function.DoubleFunction;
import com.ran.engine.algebra.vector.TwoDoubleVector;

public class SegmentsBuilder {

    private static SegmentsBuilder INSTANCE = new SegmentsBuilder();

    public static SegmentsBuilder getInstance() {
        return INSTANCE;
    }

    public Result buildSegment(TwoDoubleVector firstPoint, TwoDoubleVector secondPoint) {
        DoubleFunction<TwoDoubleVector> segment = countSegment(firstPoint, secondPoint);
        double length = countLength(firstPoint, secondPoint);
        return new Result(segment, length);
    }

    private DoubleFunction<TwoDoubleVector> countSegment(TwoDoubleVector firstPoint,
        TwoDoubleVector secondPoint) {
        return new DoubleFunction<>() {
            point -> new TwoDoubleVector(firstPoint.getX() + (secondPoint.getX() -
                firstPoint.getX()) * point,
                firstPoint.getY() + (secondPoint.getY() - firstPoint.getY()) * point),
                0.0, 1.0);
        }
    }

    private double countLength(TwoDoubleVector firstPoint, TwoDoubleVector secondPoint) {
        double xDiff = firstPoint.getX() - secondPoint.getX();
        double yDiff = firstPoint.getY() - secondPoint.getY();
        return Math.sqrt(xDiff * xDiff + yDiff * yDiff);
    }

    public static class Result {
        private final DoubleFunction<TwoDoubleVector> segment;
        private final double length;

        //      , get-   set-
    }
}

```

## PlaneBezierCurveCreator.java

```

package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

```



```

public class PlaneBezierCurveCreator extends AbstractPlainCurveCreator {

    private static final PlaneBezierCurveCreator INSTANCE = new PlaneBezierCurveCreator();

    public static PlaneBezierCurveCreator getInstance() {
        return INSTANCE;
    }

    @Override
    public DoubleFunction<TwoDoubleVector> interpolateCurve(
        List<TwoDoubleVector> vertices, SimpleInputParameters parameters, int degree) {
        validateVerticesList(vertices);

        double t0 = parameters.getT0();
        double t1 = parameters.getT1();
        int k = vertices.size();

        CurvesSmoothingCreator curvesSmoothingCreator = CurvesSmoothingCreator.getInstance();
        SegmentsBuilder segmentsBuilder = SegmentsBuilder.getInstance();

        List<DoubleFunction<TwoDoubleVector>> constantFunctions = new ArrayList<>(k);
        for (int i = 0; i < k; i++) {
            constantFunctions.add(DoubleFunction.createConstantFunction(vertices.get(i)));
        }

        List<TwoDoubleVector> segmentsCenters = new ArrayList<>(k - 1);
        for (int i = 0; i < k - 1; i++) {
            TwoDoubleVector firstVector = vertices.get(i);
            TwoDoubleVector secondVector = vertices.get(i + 1);
            segmentsCenters.add(new TwoDoubleVector(
                (firstVector.getX() + secondVector.getX()) / 2.0,
                (firstVector.getY() + secondVector.getY()) / 2.0
            ));
        }

        List<DoubleFunction<TwoDoubleVector>> halfSegmentsForward = new ArrayList<>(k - 1);
        List<DoubleFunction<TwoDoubleVector>> halfSegmentsBack = new ArrayList<>(k - 1);
        for (int i = 0; i < k - 1; i++) {
            SegmentsBuilder.Result firstHalfSegmentsBuilderResult = segmentsBuilder
                .buildSegment(vertices.get(i), segmentsCenters.get(i));
            halfSegmentsForward.add(firstHalfSegmentsBuilderResult.getSegment());
            SegmentsBuilder.Result secondHalfSegmentsBuilderResult = segmentsBuilder
                .buildSegment(vertices.get(i + 1), segmentsCenters.get(i));
            halfSegmentsBack.add(secondHalfSegmentsBuilderResult.getSegment());
        }

        List<DoubleFunction<TwoDoubleVector>> smoothedSegments = new ArrayList<>(k);
        smoothedSegments.add(halfSegmentsForward.get(0));
        for (int i = 1; i < k - 1; i++) {
            smoothedSegments.add(curvesSmoothingCreator.smoothCurves(
                halfSegmentsBack.get(i - 1).subtract(constantFunctions.get(i)),
                halfSegmentsForward.get(i).subtract(constantFunctions.get(i)),
                degree, GroupMultiplicationOperationFactory.getSummationOperation()
            ).add(constantFunctions.get(i)));
        }
        smoothedSegments.add(new DoubleFunction<>() {
            point -> halfSegmentsBack.get(k - 2).apply(1.0 - point));
        });

        List<Double> timeMoments = new ArrayList<>(k + 1);
    }
}

```

```

        double timeDelta = t1 - t0;
        for (int i = 0; i < k + 1; i++) {
            timeMoments.add(t0 + i * timeDelta);
        }
        return buildFinalCurve(timeMoments, smoothedSegments, k);
    }
}

```

## TangentSegmentBuilder.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class TangentSegmentBuilder {

    private static final TangentSegmentBuilder INSTANCE = new TangentSegmentBuilder();

    public static TangentSegmentBuilder getInstance() {
        return INSTANCE;
    }

    public Result buildTangent(TwoDoubleVector point,
                               double tangentAngle,
                               Double forwardLength,
                               Double backLength) {
        DoubleFunction<TwoDoubleVector> forwardSegment = createTangentSegment(
            point, tangentAngle, forwardLength);
        DoubleFunction<TwoDoubleVector> backSegment = createTangentSegment(
            point, tangentAngle + Math.PI, backLength);
        return new Result(forwardSegment, backSegment, forwardLength, backLength);
    }

    private DoubleFunction<TwoDoubleVector> createTangentSegment(TwoDoubleVector point,
                                                                    double tangentAngle,
                                                                    Double segmentLength) {

        if (segmentLength == null) {
            return null;
        }
        TwoDoubleVector shiftVector = new TwoDoubleVector(
            segmentLength * Math.cos(tangentAngle),
            segmentLength * Math.sin(tangentAngle));
        TwoDoubleVector farPoint = point.add(shiftVector);
        return SegmentsBuilder.getInstance().buildSegment(point, farPoint).getSegment();
    }

    public static class Result {
        private DoubleFunction<TwoDoubleVector> forwardSegment;
        private DoubleFunction<TwoDoubleVector> backSegment;
        private Double forwardLength;
        private Double backLength;

        //      , get-   set-
    }
}

```

# PlaneByTangentAnglesCurveCreator.java

```
package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public class PlaneByTangentAnglesCurveCreator extends AbstractInterpolatedCurveCreator<
    Pair<TwoDoubleVector, Double>, TwoDoubleVector, SimpleInputParameters> {

    private static final PlaneByTangentAnglesCurveCreator INSTANCE =
        new PlaneByTangentAnglesCurveCreator();

    public static PlaneByTangentAnglesCurveCreator getInstance() {
        return INSTANCE;
    }

    @Override
    public DoubleFunction<TwoDoubleVector> interpolateCurve(
        List<Pair<TwoDoubleVector, Double>> verticesWithTangentAnglesList,
        SimpleInputParameters parameters, int degree) {
        validateVerticesList(verticesWithTangentAnglesList);

        double t0 = parameters.getT0();
        double t1 = parameters.getT1();
        int k = verticesWithTangentAnglesList.size();

        CurvesDeformationCreator deformationCreator = CurvesDeformationCreator.getInstance();
        CircleArcsBuilder circleArcsBuilder = CircleArcsBuilder.getInstance();
        SegmentsBuilder segmentsBuilder = SegmentsBuilder.getInstance();
        TangentSegmentBuilder tangentSegmentBuilder = TangentSegmentBuilder.getInstance();
        TimeMomentsUtil timeMomentsUtil = TimeMomentsUtil.getInstance();

        List<DoubleFunction<TwoDoubleVector>> constantFunctions = new ArrayList<>(k);
        for (int i = 0; i < k; i++) {
            constantFunctions.add(DoubleFunction.createConstantFunction(
                verticesWithTangentAnglesList.get(i).getLeft()));
        }

        List<CircleArcsBuilder.Result> circleArcsResults = new ArrayList<>(k - 2);
        for (int i = 1; i < k - 1; i++) {
            circleArcsResults.add(circleArcsBuilder.buildCircle(
                verticesWithTangentAnglesList.get(i - 1).getLeft(),
                verticesWithTangentAnglesList.get(i).getLeft(),
                verticesWithTangentAnglesList.get(i + 1).getLeft()
            ));
        }

        List<SegmentsBuilder.Result> directSegmentsResults = new ArrayList<>(k - 1);
        for (int i = 0; i < k - 1; i++) {
            directSegmentsResults.add(segmentsBuilder.buildSegment(
                verticesWithTangentAnglesList.get(i).getLeft(),
                verticesWithTangentAnglesList.get(i + 1).getLeft()
            ));
        }

        List<TangentSegmentBuilder.Result> tangentSegmentBuilderResults = new ArrayList<>(k);
        for (int i = 0; i < k; i++) {
            if (verticesWithTangentAnglesList.get(i).getRight() == null) {
                tangentSegmentBuilderResults.add(null);
            }
        }
    }
}
```

```

    } else {
        Double forwardLength = null, backLength = null;
        if (i + 1 < k) {
            if (verticesWithTangentAnglesList.get(i + 1).getRight() != null ||
                i + 1 == k - 1) {
                forwardLength = directSegmentsResults.get(i).getLength();
            } else {
                forwardLength = circleArcsResults.get(i).getFirstArcLength();
            }
        }
        if (i - 1 >= 0) {
            if (verticesWithTangentAnglesList.get(i - 1).getRight() != null ||
                i - 1 == 0) {
                backLength = directSegmentsResults.get(i - 1).getLength();
            } else {
                backLength = circleArcsResults.get(i - 2).getSecondArcLength();
            }
        }
        tangentSegmentBuilderResults.add(tangentSegmentBuilder.buildTangent(
            verticesWithTangentAnglesList.get(i).getLeft(),
            verticesWithTangentAnglesList.get(i).getRight(),
            forwardLength, backLength));
    }
}

List<Pair<Double, Double>> arcsLengths = new ArrayList<>(k - 2);
for (int i = 0; i < k - 2; i++) {
    if (verticesWithTangentAnglesList.get(i + 1).getRight() == null) {
        arcsLengths.add(circleArcsResults.get(i).getArcsLengths());
    } else {
        arcsLengths.add(tangentSegmentBuilderResults.get(i + 1).getLengths());
    }
}

List<DoubleFunction<TwoDoubleVector>> deformedSegments = new ArrayList<>(k - 1);
for (int i = 0; i < k - 1; i++) {
    if (verticesWithTangentAnglesList.get(i).getRight() != null &&
        verticesWithTangentAnglesList.get(i + 1).getRight() != null) {
        DoubleFunction<TwoDoubleVector> firstDeformedCurve = deformationCreator
            .deformCurves(
                tangentSegmentBuilderResults.get(i).getForwardSegment()
                    .subtract(constantFunctions.get(i)),
                directSegmentsResults.get(i).getSegment().subtract(
                    constantFunctions.get(i)), degree,
                GroupMultiplicationOperationFactory.getSummationOperation());
        DoubleFunction<TwoDoubleVector> secondDeformedCurve = deformationCreator
            .deformCurvesWithCommonEnd(
                directSegmentsResults.get(i).getSegment().subtract(
                    constantFunctions.get(i)),
                tangentSegmentBuilderResults.get(i + 1).getBackSegment()
                    .subtract(constantFunctions.get(i + 1)), degree,
                GroupMultiplicationOperationFactory.getSummationOperation());
        deformedSegments.add(deformationCreator.deformCurves(
            firstDeformedCurve, secondDeformedCurve, degree,
            GroupMultiplicationOperationFactory.getSummationOperation()
                .add(constantFunctions.get(i))));
    } else if (verticesWithTangentAnglesList.get(i).getRight() != null) {
        if (i == k - 2) {
            deformedSegments.add(deformationCreator.deformCurves(
                tangentSegmentBuilderResults.get(i).getForwardSegment()

```

```

        .subtract(constantFunctions.get(i)),
        directSegmentsResults.get(i).getSegment().subtract(
            constantFunctions.get(i)), degree,
        GroupMultiplicationOperationFactory.getSummationOperation()
        .add(constantFunctions.get(i)));
    } else {
        deformedSegments.add(deformationCreator.deformCurves(
            tangentSegmentBuilderResults.get(i).getForwardSegment()
            .subtract(constantFunctions.get(i)),
            circleArcsResults.get(i).getFirstArc().subtract(
                constantFunctions.get(i)), degree,
            GroupMultiplicationOperationFactory.getSummationOperation()
            .add(constantFunctions.get(i)));
    }
} else if (verticesWithTangentAnglesList.get(i + 1).getRight() != null) {
    if (i == 0) {
        deformedSegments.add(deformationCreator.deformCurvesWithCommonEnd(
            directSegmentsResults.get(i).getSegment().subtract(
                constantFunctions.get(i)),
            tangentSegmentBuilderResults.get(i + 1).getBackSegment()
            .subtract(constantFunctions.get(i + 1)), degree,
            GroupMultiplicationOperationFactory.getSummationOperation()
            .add(constantFunctions.get(i)));
    } else {
        deformedSegments.add(deformationCreator.deformCurvesWithCommonEnd(
            circleArcsResults.get(i - 1).getSecondArc().subtract(
                constantFunctions.get(i)),
            tangentSegmentBuilderResults.get(i + 1).getBackSegment()
            .subtract(constantFunctions.get(i + 1)), degree,
            GroupMultiplicationOperationFactory.getSummationOperation()
            .add(constantFunctions.get(i)));
    }
} else {
    if (i == 0) {
        deformedSegments.add(circleArcsResults.get(i).getFirstArc());
    } else if (i == k - 2) {
        deformedSegments.add(circleArcsResults.get(i - 1).getSecondArc());
    } else {
        deformedSegments.add(deformationCreator.deformCurves(
            circleArcsResults.get(i - 1).getSecondArc().subtract(
                constantFunctions.get(i)),
            circleArcsResults.get(i).getFirstArc().subtract(
                constantFunctions.get(i)), degree,
            GroupMultiplicationOperationFactory.getSummationOperation()
            .add(constantFunctions.get(i)));
    }
}
}

List<Double> timeMoments = timeMomentsUtil.countTimeMoments(arcsLengths, t0, t1, k);
return buildFinalCurve(timeMoments, deformedSegments, k - 1);
}
}

```

# Приложение С

## Интерполирование кривой на двумерной сфере

### AbstractSphereCurveCreator.java

```
package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public abstract class AbstractSphereCurveCreator extends AbstractInterpolatedCurveCreator<
    ThreeDoubleVector, ThreeDoubleVector, SimpleInputParameters> {

    @Override
    protected void validateVerticesList(List<ThreeDoubleVector> verticesList) {
        super.validateVerticesList(verticesList);
        double radius = verticesList.get(0).getNorm();
        if (verticesList.stream().anyMatch(vertice -> ArithmeticOperations.doubleNotEquals(
            vertice.getNorm(), radius))) {
            throw new InterpolationException("All vertices must belong to the same sphere");
        }
    }

    protected DoubleFunction<ThreeDoubleVector> buildFinalCurve(
        List<Double> timeMoments,
        List<ThreeDoubleVector> vertices,
        List<DoubleFunction<DoubleMatrix>> rotationsOnSegments,
        int segmentsQuantity) {
        TimeMomentsUtil timeMomentsUtil = TimeMomentsUtil.getInstance();
        List<DoubleFunction<ThreeDoubleVector>> curveSegments =
            new ArrayList<>(segmentsQuantity);
        for (int i = 0; i < segmentsQuantity; i++) {
            double startTime = timeMoments.get(i);
            double endTime = timeMoments.get(i + 1);
            DoubleFunction<DoubleMatrix> currentRotation = rotationsOnSegments.get(i);
            DoubleVector currentVertice = vertices.get(i).getDoubleVector();
            DoubleFunction<ThreeDoubleVector> curveSegmentWithoutAligning = new DoubleFunction<>() {
                @Override
                public ThreeDoubleVector apply(ThreeDoubleVector point) {
                    return currentRotation.apply(point).multiply(currentVertice);
                }
            };
            DoubleFunction<ThreeDoubleVector> alignedCurveSegment =
                curveSegmentWithoutAligning.superposition(
```

```

        timeMomentsUtil.buildAligningFunction(startTime, endTime));
        curveSegments.add(alignedCurveSegment);
    }
    return DoubleMultifunction.makeMultifunction(curveSegments);
}
}

```

## ArcsBuilder.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class ArcsBuilder {

    private static final ArcsBuilder INSTANCE = new ArcsBuilder();

    public static ArcsBuilder getInstance() {
        return INSTANCE;
    }

    public Result buildArcsBetweenVerticesOnSphere(
        ThreeDoubleVector p1, ThreeDoubleVector p2, ThreeDoubleVector p3) {
        ThreeDoubleVector a = (p3.subtract(p2)).multiply(p1.subtract(p2));
        double aNorm = a.getNorm();
        if (ArithmeticOperations.doubleEquals(aNorm, 0.0)) {
            throw new AlgebraicException("Every three sequential vertices must not coincide");
        }

        double mixedProduction = p1.mixedMultiply(p2, p3);
        ThreeDoubleVector n = a.multiply(1.0 / aNorm);
        ThreeDoubleVector c = n.multiply(mixedProduction / aNorm);

        ThreeDoubleVector r1 = p1.subtract(c);
        ThreeDoubleVector r2 = p2.subtract(c);
        ThreeDoubleVector r3 = p3.subtract(c);

        ThreeDoubleVector n1 = r1.multiply(r2);
        ThreeDoubleVector n2 = r2.multiply(r3);

        double n1Norm = n1.getNorm();
        double n2Norm = n2.getNorm();
        double s1 = r1.scalarMultiply(r2);
        double s2 = r2.scalarMultiply(r3);

        double firstAtan2 = Math.atan2(n1Norm, s1);
        double phi = -(n1.scalarMultiply(n) > 0 ? firstAtan2 : 2 * Math.PI - firstAtan2);

        double secondAtan2 = Math.atan2(n2Norm, s2);
        double psi = -(n2.scalarMultiply(n) > 0 ? secondAtan2 : 2 * Math.PI - secondAtan2);

        return new Result(
            new DoubleFunction<>(point -> RotationCreator.getInstance()
                .createRotation(n, point * phi), 0.0, 1.0),
            new DoubleFunction<>(point -> RotationCreator.getInstance()
                .createRotation(n, point * psi), 0.0, 1.0),
            phi, psi);
    }
}

```

```

    }

    public static class Result {
        private final DoubleFunction<DoubleMatrix> firstRotation;
        private final DoubleFunction<DoubleMatrix> secondRotation;
        private final double firstAngle;
        private final double secondAngle;

        //      , get-   set-
    }
}

```

## SphereByPointsCurveCreator.java

```

package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public class SphereByPointsCurveCreator extends AbstractSphereCurveCreator {

    private static final SphereByPointsCurveCreator INSTANCE = new SphereByPointsCurveCreator();

    public static SphereByPointsCurveCreator getInstance() {
        return INSTANCE;
    }

    @Override
    public DoubleFunction<ThreeDoubleVector> interpolateCurve(List<ThreeDoubleVector> vertices,
                                                                SimpleInputParameters parameters,
                                                                int degree) {

        validateVerticesList(vertices);

        double t0 = parameters.getT0();
        double t1 = parameters.getT1();
        int k = vertices.size();

        CurvesDeformationCreator deformationCreator = CurvesDeformationCreator.getInstance();
        ArcsBuilder arcsBuilder = ArcsBuilder.getInstance();
        TimeMomentsUtil timeMomentsUtil = TimeMomentsUtil.getInstance();

        List<DoubleFunction<DoubleMatrix>> rotationsOnSegments = new ArrayList<>(k - 1);
        List<Pair<Double, Double>> rotationAngles = new ArrayList<>(k - 2);
        ArcsBuilder.Result currentArcsBuildingResult =
            arcsBuilder.buildArcsBetweenVerticesOnSphere(
                vertices.get(0), vertices.get(1), vertices.get(2));
        rotationsOnSegments.add(currentArcsBuildingResult.getFirstRotation());
        rotationAngles.add(currentArcsBuildingResult.getAngles());

        for (int i = 1; i < k - 2; i++) {
            ArcsBuilder.Result nextArcsBuildingResult =
                arcsBuilder.buildArcsBetweenVerticesOnSphere(
                    vertices.get(i), vertices.get(i + 1), vertices.get(i + 2));
            DoubleFunction<DoubleMatrix> deformedFunction = deformationCreator.deformCurves(
                currentArcsBuildingResult.getSecondRotation(),
                nextArcsBuildingResult.getFirstRotation(), degree);
            rotationsOnSegments.add(deformedFunction);
            rotationAngles.add(nextArcsBuildingResult.getAngles());
        }
    }
}

```



```

        currentArcsBuildingResult = nextArcsBuildingResult;
    }
    rotationsOnSegments.add(currentArcsBuildingResult.getSecondRotation());

    List<Double> timeMoments = timeMomentsUtil.countTimeMoments(rotationAngles, t0, t1, k);
    return buildFinalCurve(timeMoments, vertices, rotationsOnSegments, k - 1);
}
}

```

## BigArcsBuilder.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class BigArcsBuilder {

    private static final BigArcsBuilder INSTANCE = new BigArcsBuilder();

    public static BigArcsBuilder getInstance() {
        return INSTANCE;
    }

    public Result buildBigArcBetweenVerticesOnSphere(
        ThreeDoubleVector p1, ThreeDoubleVector p2) {
        ThreeDoubleVector a = p1.multiply(p2);
        double aNorm = a.getNorm();
        if (ArithmeticOperations.doubleEquals(aNorm, 0.0)) {
            throw new AlgebraicException("Every two sequential vertices must not coincide");
        }

        ThreeDoubleVector n = a.multiply(1.0 / aNorm);
        double phi = -Math.atan(aNorm / p1.scalarMultiply(p2));

        return new BigArcsBuilder.Result(
            new DoubleFunction<>(point -> RotationCreator.getInstance()
                .createRotation(n, point * phi), 0.0, 1.0), phi);
    }

    public static class Result {
        private final DoubleFunction<DoubleMatrix> rotation;
        private final double angle;

        //      , get-   set-
    }
}

```

## SphereBezierCurveCreator.java

```

package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public class SphereBezierCurveCreator extends AbstractSphereCurveCreator {

```

```

private static final SphereBezierCurveCreator INSTANCE =
    new SphereBezierCurveCreator();

public static SphereBezierCurveCreator getInstance() {
    return INSTANCE;
}

@Override
public DoubleFunction<ThreeDoubleVector> interpolateCurve(
    List<ThreeDoubleVector> verticesList,
    SimpleInputParameters parameters, int degree) {
    validateVerticesList(verticesList);

    double t0 = parameters.getT0();
    double t1 = parameters.getT1();
    int k = verticesList.size();

    CurvesSmoothingCreator curvesSmoothingCreator = CurvesSmoothingCreator.getInstance();
    BigArcsBuilder bigArcsBuilder = BigArcsBuilder.getInstance();

    List<ThreeDoubleVector> arcsCenters = new ArrayList<>(k - 1);
    for (int i = 0; i < k - 1; i++) {
        BigArcsBuilder.Result bigArcsBuilderResult =
            bigArcsBuilder.buildBigArcBetweenVerticesOnSphere(
                verticesList.get(i), verticesList.get(i + 1));
        arcsCenters.add(new ThreeDoubleVector(bigArcsBuilderResult.getRotation().apply(0.5)
            .multiply(verticesList.get(i).getDoubleVector())));
    }

    List<DoubleFunction<DoubleMatrix>> halfRotationsForward = new ArrayList<>(k - 1);
    List<DoubleFunction<DoubleMatrix>> halfRotationsBack = new ArrayList<>(k - 1);
    for (int i = 0; i < k - 1; i++) {
        BigArcsBuilder.Result firstHalfBigArcsBuilderResult = bigArcsBuilder
            .buildBigArcBetweenVerticesOnSphere(
                verticesList.get(i), arcsCenters.get(i));
        halfRotationsForward.add(firstHalfBigArcsBuilderResult.getRotation());
        BigArcsBuilder.Result secondHalfBigArcsBuilderResult = bigArcsBuilder
            .buildBigArcBetweenVerticesOnSphere(
                verticesList.get(i + 1), arcsCenters.get(i));
        halfRotationsBack.add(secondHalfBigArcsBuilderResult.getRotation());
    }

    List<DoubleFunction<DoubleMatrix>> smoothedRotations = new ArrayList<>(k);
    smoothedRotations.add(halfRotationsForward.get(0));
    for (int i = 1; i < k - 1; i++) {
        smoothedRotations.add(curvesSmoothingCreator.smoothCurves(
            halfRotationsBack.get(i - 1), halfRotationsForward.get(i), degree));
    }
    smoothedRotations.add(new DoubleFunction<>() {
        point -> halfRotationsBack.get(k - 2).apply(1.0 - point),
        0.0, 1.0
    });

    List<Double> timeMoments = new ArrayList<>(k + 1);
    double timeDelta = t1 - t0;
    for (int i = 0; i < k + 1; i++) {
        timeMoments.add(t0 + i * timeDelta);
    }
    return buildFinalCurve(timeMoments, verticesList, smoothedRotations, k);
}

```

```

    }
}

```

## TangentBuilder.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class TangentBuilder {

    private static final Logger LOG = LoggerFactory.getLogger(TangentBuilder.class);

    private static final TangentBuilder INSTANCE = new TangentBuilder();
    private static final DoubleMatrix Z_HALF_PI_ROTATION = RotationCreator.getInstance()
        .createRotation(ThreeDoubleVector.Z_ONE_THREE_DOUBLE_VECTOR, -Math.PI / 2.0);

    public static TangentBuilder getInstance() {
        return INSTANCE;
    }

    public Result buildTangent(ThreeDoubleVector point,
                              double tangentAngle,
                              Double forwardRotationAngle,
                              Double backRotationAngle) {
        LOG.trace("point = {}, tangentAngle = {}, forwardRotationAngle = {}, " +
            "backRotationAngle = {}", point, tangentAngle, forwardRotationAngle,
            backRotationAngle);
        ThreeDoubleVector a, b;
        if (ArithmeticOperations.doubleEquals(point.getX(), 0.0) &&
            ArithmeticOperations.doubleEquals(point.getY(), 0.0)) {
            a = ThreeDoubleVector.X_ONE_THREE_DOUBLE_VECTOR;
            if (point.getZ() > 0.0) {
                b = ThreeDoubleVector.MINUS_Y_ONE_THREE_DOUBLE_VECTOR;
            } else {
                b = ThreeDoubleVector.Y_ONE_THREE_DOUBLE_VECTOR;
            }
        } else {
            a = new ThreeDoubleVector(Z_HALF_PI_ROTATION.multiply(
                new ThreeDoubleVector(point.getX(), point.getY(), 0.0)
                    .getDoubleVector()).normalized());
            b = point.multiply(a).normalized();
        }
        ThreeDoubleVector n = a.multiply(Math.sin(tangentAngle))
            .add(b.multiply(-Math.cos(tangentAngle)));
        LOG.trace("a = {}, b = {}, n = {}", a, b, n);

        DoubleFunction<DoubleMatrix> forwardRotation = null;
        DoubleFunction<DoubleMatrix> backRotation = null;

        if (forwardRotationAngle != null) {
            forwardRotation = new DoubleFunction<>() {
                u -> RotationCreator.getInstance().createRotation(
                    n, u * forwardRotationAngle,
                    0.0, 1.0);
            }
        }
        if (backRotationAngle != null) {

```

```

        backRotation = new DoubleFunction<>() {
            u -> RotationCreator.getInstance().createRotation(n, -u * backRotationAngle),
            0.0, 1.0);
    }
    return new Result(forwardRotation, backRotation, forwardRotationAngle,
        backRotationAngle);
}

public static class Result {
    private final DoubleFunction<DoubleMatrix> forwardRotation;
    private final DoubleFunction<DoubleMatrix> backRotation;
    private final Double forwardAngle, backAngle;

    //      , get-   set-
}
}

```

## SphereByTangentAnglesCurveCreator.java

```

package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public class SphereByTangentAnglesCurveCreator extends AbstractInterpolatedCurveCreator<
    Pair<ThreeDoubleVector, Double>, ThreeDoubleVector, SimpleInputParameters> {

    private static final Logger LOG = LoggerFactory.getLogger(
        SphereByTangentAnglesCurveCreator.class);

    private static final SphereByTangentAnglesCurveCreator INSTANCE =
        new SphereByTangentAnglesCurveCreator();

    public static SphereByTangentAnglesCurveCreator getInstance() {
        return INSTANCE;
    }

    @Override
    public DoubleFunction<ThreeDoubleVector> interpolateCurve(
        List<Pair<ThreeDoubleVector, Double>> verticesWithTangentAnglesList,
        SimpleInputParameters parameters, int degree) {
        LOG.trace("verticesWithTangentAnglesList = {}, parameters = {}, degree = {}",
            verticesWithTangentAnglesList, parameters, degree);
        validateVerticesList(verticesWithTangentAnglesList);

        double t0 = parameters.getT0();
        double t1 = parameters.getT1();
        int k = verticesWithTangentAnglesList.size();
        LOG.trace("t0 = {}, t1 = {}, k = {}", t0, t1, k);

        CurvesDeformationCreator deformationCreator = CurvesDeformationCreator.getInstance();
        ArcsBuilder arcsBuilder = ArcsBuilder.getInstance();
        BigArcsBuilder bigArcsBuilder = BigArcsBuilder.getInstance();
        TangentBuilder tangentBuilder = TangentBuilder.getInstance();
        TimeMomentsUtil timeMomentsUtil = TimeMomentsUtil.getInstance();

        LOG.trace("Before calling ArcsBuilder");
        List<ArcsBuilder.Result> smallArcsResults = new ArrayList<>(k - 2);
    }
}

```

```

for (int i = 1; i < k - 1; i++) {
    smallArcsResults.add(arcsBuilder.buildArcsBetweenVerticesOnSphere(
        verticesWithTangentAnglesList.get(i - 1).getLeft(),
        verticesWithTangentAnglesList.get(i).getLeft(),
        verticesWithTangentAnglesList.get(i + 1).getLeft()
    ));
}
LOG.trace("smallArcsResults = {}", smallArcsResults);

LOG.trace("Before calling BigArcsBuilder");
List<BigArcsBuilder.Result> bigArcsResults = new ArrayList<>(k - 1);
for (int i = 0; i < k - 1; i++) {
    bigArcsResults.add(bigArcsBuilder.buildBigArcBetweenVerticesOnSphere(
        verticesWithTangentAnglesList.get(i).getLeft(),
        verticesWithTangentAnglesList.get(i + 1).getLeft()
    ));
}
LOG.trace("bigArcsResults = {}", bigArcsResults);

LOG.trace("Before calling TangentBuilder");
List<TangentBuilder.Result> tangentBuilderResults = new ArrayList<>(k);
for (int i = 0; i < k; i++) {
    LOG.trace("Point #{} = {}", i, verticesWithTangentAnglesList.get(i));
    if (verticesWithTangentAnglesList.get(i).getRight() == null) {
        LOG.trace("Tangle angle is null");
        tangentBuilderResults.add(null);
    } else {
        LOG.trace("Processing angles");
        Double forwardAngle = null, backAngle = null;
        if (i + 1 < k) {
            if (verticesWithTangentAnglesList.get(i + 1).getRight() != null ||
                i + 1 == k - 1) {
                forwardAngle = Math.abs(bigArcsResults.get(i).getAngle());
            } else {
                forwardAngle = Math.abs(smallArcsResults.get(i).getFirstAngle());
            }
        }
        if (i - 1 >= 0) {
            if (verticesWithTangentAnglesList.get(i - 1).getRight() != null ||
                i - 1 == 0) {
                backAngle = Math.abs(bigArcsResults.get(i - 1).getAngle());
            } else {
                backAngle = Math.abs(smallArcsResults.get(i - 2).getSecondAngle());
            }
        }
        LOG.trace("forwardAngle = {}, backAngle = {}", forwardAngle, backAngle);
        tangentBuilderResults.add(tangentBuilder.buildTangent(
            verticesWithTangentAnglesList.get(i).getLeft(),
            verticesWithTangentAnglesList.get(i).getRight(),
            forwardAngle,
            backAngle
        ));
    }
}

List<Pair<Double, Double>> rotationAngles = new ArrayList<>(k - 2);
for (int i = 0; i < k - 2; i++) {
    if (verticesWithTangentAnglesList.get(i + 1).getRight() == null) {
        rotationAngles.add(smallArcsResults.get(i).getAngles());
    } else {

```

```

        rotationAngles.add(tangentBuilderResults.get(i + 1).getAngles());
    }
}
LOG.trace("rotationAngles = {}", rotationAngles);

LOG.trace("Before building rotationsOnSegments");
List<DoubleFunction<DoubleMatrix>> rotationsOnSegments = new ArrayList<>(k - 1);
for (int i = 0; i < k - 1; i++) {
    LOG.trace("Building rotation between points {} and {}: {} and {}",
        i, i + 1, verticesWithTangentAnglesList.get(i),
        verticesWithTangentAnglesList.get(i + 1));
    if (verticesWithTangentAnglesList.get(i).getRight() != null &&
        verticesWithTangentAnglesList.get(i + 1).getRight() != null) {
        LOG.trace("Angles are set on both points");
        DoubleFunction<DoubleMatrix> firstDeformedCurve = deformationCreator
            .deformCurves(
                tangentBuilderResults.get(i).getForwardRotation(),
                bigArcsResults.get(i).getRotation(), degree);
        DoubleFunction<DoubleMatrix> secondDeformedCurve = deformationCreator
            .deformCurvesWithCommonEnd(
                bigArcsResults.get(i).getRotation(),
                tangentBuilderResults.get(i + 1).getBackRotation(), degree);
        rotationsOnSegments.add(deformationCreator.deformCurves(
            firstDeformedCurve, secondDeformedCurve, degree));
    } else if (verticesWithTangentAnglesList.get(i).getRight() != null) {
        LOG.trace("Angle is set only on the first point");
        if (i == k - 2) {
            rotationsOnSegments.add(deformationCreator.deformCurves(
                tangentBuilderResults.get(i).getForwardRotation(),
                bigArcsResults.get(i).getRotation(), degree));
        } else {
            rotationsOnSegments.add(deformationCreator.deformCurves(
                tangentBuilderResults.get(i).getForwardRotation(),
                smallArcsResults.get(i).getFirstRotation(), degree));
        }
    } else if (verticesWithTangentAnglesList.get(i + 1).getRight() != null) {
        LOG.trace("Angle is set only on the second point");
        if (i == 0) {
            rotationsOnSegments.add(deformationCreator.deformCurvesWithCommonEnd(
                bigArcsResults.get(i).getRotation(),
                tangentBuilderResults.get(i + 1).getBackRotation(), degree));
        } else {
            rotationsOnSegments.add(deformationCreator.deformCurvesWithCommonEnd(
                smallArcsResults.get(i - 1).getSecondRotation(),
                tangentBuilderResults.get(i + 1).getBackRotation(), degree));
        }
    } else {
        LOG.trace("Angles are not set on both points");
        if (i == 0) {
            rotationsOnSegments.add(smallArcsResults.get(i).getFirstRotation());
        } else if (i == k - 2) {
            rotationsOnSegments.add(smallArcsResults.get(i - 1).getSecondRotation());
        } else {
            rotationsOnSegments.add(deformationCreator.deformCurves(
                smallArcsResults.get(i - 1).getSecondRotation(),
                smallArcsResults.get(i).getFirstRotation(), degree));
        }
    }
}
}

```

```

List<Double> timeMoments = timeMomentsUtil.countTimeMoments(rotationAngles, t0, t1, k);
LOG.trace("timeMoments = {}", timeMoments);

LOG.trace("Before building curveSegments");
List<DoubleFunction<ThreeDoubleVector>> curveSegments = new ArrayList<>(k - 1);
for (int i = 0; i < k - 1; i++) {
    LOG.trace("Building curve between {} and {} points");
    double startTime = timeMoments.get(i);
    double endTime = timeMoments.get(i + 1);
    DoubleFunction<DoubleMatrix> currentRotation = rotationsOnSegments.get(i);
    DoubleVector currentVertex = verticesWithTangentAnglesList.get(i).getLeft()
        .getDoubleVector();
    LOG.trace("startTime = {}, endTime = {}, currentVertex = {}", startTime, endTime,
        currentVertex);
    DoubleFunction<ThreeDoubleVector> curveSegmentWithoutAligning = new DoubleFunction<>(
        point -> new ThreeDoubleVector(currentRotation.apply(point)
            .multiply(currentVertex)), 0.0, 1.0);
    DoubleFunction<ThreeDoubleVector> alignedCurveSegment =
        curveSegmentWithoutAligning.superposition(timeMomentsUtil
            .buildAligningFunction(startTime, endTime));
    curveSegments.add(alignedCurveSegment);
}
return DoubleMultifunction.makeMultifunction(curveSegments);
}

@Override
protected void validateVerticesList(
    List<Pair<ThreeDoubleVector, Double>> verticesWithTangentAnglesList) {
    super.validateVerticesList(verticesWithTangentAnglesList);
    double radius = verticesWithTangentAnglesList.get(0).getLeft().getNorm();
    if (verticesWithTangentAnglesList.stream().anyMatch(
        verticeWithTangentAngle -> ArithmeticOperations.doubleNotEquals(
            verticeWithTangentAngle.getLeft().getNorm(), radius))) {
        String message = "All vertices must belong to the same sphere";
        LOG.error(message);
        throw new InterpolationException(message);
    }
}
}

```

## Приложение D

# Интерполирование кривой на ориентационной сфере

## AbstractOrientationCurveCreator.java

```
package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public abstract class AbstractOrientationCurveCreator extends AbstractInterpolatedCurveCreator<
    Quaternion, Quaternion, SimpleInputParameters> {

    @Override
    protected void validateVerticesList(List<Quaternion> verticesList) {
        super.validateVerticesList(verticesList);
        if (verticesList.stream().anyMatch(quaternion -> !quaternion.isIdentity())) {
            throw new InterpolationException("All quaternions must be identity");
        }
    }

    protected DoubleFunction<Quaternion> buildFinalCurve(
        List<Double> timeMoments,
        List<Quaternion> quaternions,
        List<DoubleFunction<Quaternion>> rotationsOnSegments,
        int segmentsQuantity) {
        TimeMomentsUtil timeMomentsUtil = TimeMomentsUtil.getInstance();
        List<DoubleFunction<Quaternion>> orientationCurveSegments =
            new ArrayList<>(segmentsQuantity);
        for (int i = 0; i < segmentsQuantity; i++) {
            double startTime = timeMoments.get(i);
            double endTime = timeMoments.get(i + 1);
            DoubleFunction<Quaternion> currentRotation = rotationsOnSegments.get(i);
            Quaternion currentQuaternion = quaternions.get(i);
            DoubleFunction<Quaternion> curveSegmentWithoutAligning = new DoubleFunction<>() {
                @Override
                public Quaternion apply(Quaternion point) {
                    return currentRotation.apply(point).multiply(currentQuaternion, 0.0, 1.0);
                }
            };
            DoubleFunction<Quaternion> alignedCurveSegment =
                curveSegmentWithoutAligning.superposition(timeMomentsUtil
                    .buildAligningFunction(startTime, endTime));
            orientationCurveSegments.add(alignedCurveSegment);
        }
    }
}
```



```

        return DoubleMultifunction.makeMultifunction(orientationCurveSegments);
    }
}

```

## OrientationArcsBuilder.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class OrientationArcsBuilder {

    private static final OrientationArcsBuilder INSTANCE = new OrientationArcsBuilder();

    public static OrientationArcsBuilder getInstance() {
        return INSTANCE;
    }

    public Result buildArcsBetweenQuaternionsOnThreeDimensionalSphere(
        Quaternion p1, Quaternion p2, Quaternion p3) {
        Quaternion hNotNormalized = p1.quaternionVectorMultiply(p2, p3);
        if (ArithmeticOperations.doubleEquals(hNotNormalized.getNorm(), 0.0)) {
            OrientationBigArcsBuilder bigArcsBuilder = OrientationBigArcsBuilder.getInstance();
            OrientationBigArcsBuilder.Result firstArc = bigArcsBuilder
                .buildOrientationBigArcsBetweenQuaternions(p1, p2);
            OrientationBigArcsBuilder.Result secondArc = bigArcsBuilder
                .buildOrientationBigArcsBetweenQuaternions(p2, p3);
            return new Result(firstArc.getRotation(), secondArc.getRotation(),
                firstArc.getAngle(), secondArc.getAngle());
        } else {
            Quaternion h = hNotNormalized.normalized();
            Quaternion hConjugate = h.getConjugate();

            Quaternion r1 = p1.multiply(hConjugate);
            Quaternion r2 = p2.multiply(hConjugate);
            Quaternion r3 = p3.multiply(hConjugate);

            ThreeDoubleVector m1 = r1.getVector();
            ThreeDoubleVector m2 = r2.getVector();
            ThreeDoubleVector m3 = r3.getVector();

            ArcsBuilder.Result arcsBuilderResult = ArcsBuilder.getInstance()
                .buildArcsBetweenVerticesOnSphere(m1, m2, m3);

            DoubleFunction<Quaternion> firstRotation = buildRotationFunction(
                arcsBuilderResult.getFirstRotation(), m1, r1);
            DoubleFunction<Quaternion> secondRotation = buildRotationFunction(
                arcsBuilderResult.getSecondRotation(), m2, r2);

            return new Result(firstRotation, secondRotation,
                arcsBuilderResult.getFirstAngle(), arcsBuilderResult.getSecondAngle());
        }
    }

    private DoubleFunction<Quaternion> buildRotationFunction(
        DoubleFunction<DoubleMatrix> matrixRotation,
        ThreeDoubleVector m, Quaternion r) {

```

```

        return new DoubleFunction<>() {
            point -> {
                DoubleMatrix rotation = matrixRotation.apply(point);
                Quaternion leftFactor = Quaternion.createFromVector(
                    new ThreeDoubleVector(rotation.multiply(m.getDoubleVector())));
                return leftFactor.multiply(r.getConjugate());
            },
            0.0, 1.0
        );
    }

    public static class Result {
        private final DoubleFunction<Quaternion> firstRotation;
        private final DoubleFunction<Quaternion> secondRotation;
        private final double firstAngle;
        private final double secondAngle;

        //      , get-   set-
    }
}

```

## OrientationByPointsCurveCreator.java

```

package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public class OrientationByPointsCurveCreator extends AbstractOrientationCurveCreator {

    private static final OrientationByPointsCurveCreator INSTANCE =
        new OrientationByPointsCurveCreator();

    public static OrientationByPointsCurveCreator getInstance() {
        return INSTANCE;
    }

    @Override
    public DoubleFunction<Quaternion> interpolateCurve(List<Quaternion> quaternions,
                                                         SimpleInputParameters parameters,
                                                         int degree) {

        validateVerticesList(quaternions);

        double t0 = parameters.getT0();
        double t1 = parameters.getT1();
        int k = quaternions.size();

        CurvesDeformationCreator deformationCreator = CurvesDeformationCreator.getInstance();
        OrientationArcsBuilder orientationArcsBuilder = OrientationArcsBuilder.getInstance();
        TimeMomentsUtil timeMomentsUtil = TimeMomentsUtil.getInstance();

        List<DoubleFunction<Quaternion>> rotationsOnSegments = new ArrayList<>(k - 1);
        List<Pair<Double, Double>> rotationAngles = new ArrayList<>(k - 2);
        OrientationArcsBuilder.Result currentOrientationArcsBuildingResult =
            orientationArcsBuilder.buildArcsBetweenQuaternionsOnThreeDimensionalSphere(
                quaternions.get(0), quaternions.get(1), quaternions.get(2));
        rotationsOnSegments.add(currentOrientationArcsBuildingResult.getFirstRotation());
        rotationAngles.add(currentOrientationArcsBuildingResult.getAngles());
    }
}

```

```

    for (int i = 1; i < k - 2; i++) {
        OrientationArcsBuilder.Result nextOrientationArcsBuildingResult =
            orientationArcsBuilder.buildArcsBetweenQuaternionsOnThreeDimensionalSphere(
                quaternions.get(i), quaternions.get(i + 1), quaternions.get(i + 2));
        DoubleFunction<Quaternion> deformedFunction = deformationCreator.deformCurves(
            currentOrientationArcsBuildingResult.getSecondRotation(),
            nextOrientationArcsBuildingResult.getFirstRotation(), degree);
        rotationsOnSegments.add(deformedFunction);
        rotationAngles.add(nextOrientationArcsBuildingResult.getAngles());
        currentOrientationArcsBuildingResult = nextOrientationArcsBuildingResult;
    }
    rotationsOnSegments.add(currentOrientationArcsBuildingResult.getSecondRotation());

    List<Double> timeMoments = timeMomentsUtil.countTimeMoments(rotationAngles, t0, t1, k);
    return buildFinalCurve(timeMoments, quaternions, rotationsOnSegments, k - 1);
}
}

```

## OrientationBigArcsBuilder.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class OrientationBigArcsBuilder {

    private static OrientationBigArcsBuilder INSTANCE = new OrientationBigArcsBuilder();

    public static OrientationBigArcsBuilder getInstance() {
        return INSTANCE;
    }

    public Result buildOrientationBigArcsBetweenQuaternions(Quaternion p1, Quaternion p2) {
        Quaternion r = p2.multiply(p1.getConjugate());
        ThreeDoubleVector axis = r.getVector().normalized();
        double cos = r.getScalar();
        double angle = (Math.acos(cos) * 2.0 + Math.PI) % (2.0 * Math.PI) - Math.PI;
        DoubleFunction<Quaternion> rotation = new DoubleFunction<>() {
            Quaternion createForRotation(axis, angle * point),
            0.0, 1.0
        };
        return new Result(rotation, angle);
    }

    public static class Result {
        private DoubleFunction<Quaternion> rotation;
        private double angle;

        //      , get-   set-
    }
}

```

# OrientationBezierCurveCreator.java

```

package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public class OrientationBezierCurveCreator extends AbstractOrientationCurveCreator {

    @Override
    public DoubleFunction<Quaternion> interpolateCurve(
        List<Quaternion> quaternions, SimpleInputParameters parameters, int degree) {
        validateVerticesList(quaternions);

        double t0 = parameters.getT0();
        double t1 = parameters.getT1();
        int k = quaternions.size();

        CurvesSmoothingCreator curvesSmoothingCreator = CurvesSmoothingCreator.getInstance();
        OrientationBigArcsBuilder bigArcsBuilder = OrientationBigArcsBuilder.getInstance();

        List<Quaternion> arcsCenters = new ArrayList<>(k - 1);
        for (int i = 0; i < k - 1; i++) {
            OrientationBigArcsBuilder.Result bigArcsBuilderResult = bigArcsBuilder
                .buildOrientationBigArcsBetweenQuaternions(quaternions.get(i),
                    quaternions.get(i + 1));
            arcsCenters.add(bigArcsBuilderResult.getRotation().apply(0.5)
                .multiply(quaternions.get(i)));
        }

        List<DoubleFunction<Quaternion>> halfRotationsForward = new ArrayList<>(k - 1);
        List<DoubleFunction<Quaternion>> halfRotationsBack = new ArrayList<>(k - 1);
        for (int i = 0; i < k - 1; i++) {
            OrientationBigArcsBuilder.Result firstHalfBigArcsBuilderResult = bigArcsBuilder
                .buildOrientationBigArcsBetweenQuaternions(quaternions.get(i),
                    arcsCenters.get(i));
            halfRotationsForward.add(firstHalfBigArcsBuilderResult.getRotation());
            OrientationBigArcsBuilder.Result secondHalfBigArcsBuilderResult = bigArcsBuilder
                .buildOrientationBigArcsBetweenQuaternions(quaternions.get(i + 1),
                    arcsCenters.get(i));
            halfRotationsBack.add(secondHalfBigArcsBuilderResult.getRotation());
        }

        List<DoubleFunction<Quaternion>> smoothedRotations = new ArrayList<>(k);
        smoothedRotations.add(halfRotationsForward.get(0));
        for (int i = 1; i < k - 1; i++) {
            smoothedRotations.add(curvesSmoothingCreator.smoothCurves(
                halfRotationsBack.get(i - 1), halfRotationsForward.get(i), degree));
        }
        smoothedRotations.add(new DoubleFunction<>() {
            point -> halfRotationsBack.get(k - 2).apply(1.0 - point),
            0.0, 1.0
        });

        List<Double> timeMoments = new ArrayList<>(k + 1);
        double timeDelta = t1 - t0;
        for (int i = 0; i < k + 1; i++) {
            timeMoments.add(t0 + i * timeDelta);
        }
        return buildFinalCurve(timeMoments, quaternions, smoothedRotations, k);
    }
}

```

```

    }
}

```

## TangentOrientationBuilder.java

```

package com.ran.engine.factories.interpolation.tools;

//
// ...

public class TangentOrientationBuilder {

    private static final Logger LOG = LoggerFactory.getLogger(TangentOrientationBuilder.class);

    private static TangentOrientationBuilder INSTANCE = new TangentOrientationBuilder();
    private static final DoubleMatrix Z_HALF_PI_ROTATION =
        RotationCreator.getInstance().createRotation(
            ThreeDoubleVector.Z_ONE_THREE_DOUBLE_VECTOR, -Math.PI / 2.0);

    public static TangentOrientationBuilder getInstance() {
        return INSTANCE;
    }

    public Result buildTangentOrientation(Quaternion orientation,
                                         double tangentAngle,
                                         Double forwardRotationAngle,
                                         Double backRotationAngle) {
        LOG.trace("orientation = {}, tangentAngle = {}, forwardRotationAngle = {}, " +
            "backRotationAngle = {}",
            orientation, tangentAngle, forwardRotationAngle, backRotationAngle);
        ThreeDoubleVector a, b;
        ThreeDoubleVector point = orientation.multiply(Quaternion.createFromVector(
            ThreeDoubleVector.Z_ONE_THREE_DOUBLE_VECTOR))
            .multiply(orientation.getConjugate()).getVector();
        if (ArithmeticOperations.doubleEquals(point.getX(), 0.0) &&
            ArithmeticOperations.doubleEquals(point.getY(), 0.0)) {
            a = ThreeDoubleVector.X_ONE_THREE_DOUBLE_VECTOR;
            if (point.getZ() > 0.0) {
                b = ThreeDoubleVector.MINUS_Y_ONE_THREE_DOUBLE_VECTOR;
            } else {
                b = ThreeDoubleVector.Y_ONE_THREE_DOUBLE_VECTOR;
            }
        } else {
            a = new ThreeDoubleVector(Z_HALF_PI_ROTATION.multiply(
                new ThreeDoubleVector(point.getX(), point.getY(), 0.0)
                    .getDoubleVector()).normalized());
            b = point.multiply(a).normalized();
        }
        ThreeDoubleVector n = a.multiply(Math.sin(tangentAngle))
            .add(b.multiply(-Math.cos(tangentAngle)));
        LOG.trace("a = {}, b = {}, n = {}", a, b, n);

        DoubleFunction<Quaternion> forwardRotation = null;
        DoubleFunction<Quaternion> backRotation = null;

        if (forwardRotationAngle != null) {
            forwardRotation = new DoubleFunction<>() {
                u -> Quaternion.createForRotation(n, -u * forwardRotationAngle),
            };
        }
    }
}

```

```

        0.0, 1.0);
    }
    if (backRotationAngle != null) {
        backRotation = new DoubleFunction<>() {
            u -> Quaternion.createForRotation(n, u * backRotationAngle),
            0.0, 1.0);
        }
    }
    return new Result(forwardRotation, backRotation,
        forwardRotationAngle, backRotationAngle);
}

public static class Result {
    private final DoubleFunction<Quaternion> forwardRotation;
    private final DoubleFunction<Quaternion> backRotation;
    private final Double forwardAngle, backAngle;

    //      , get-    set-
}
}

```

## OrientationByTangentAnglesCurveCreator.java

```

package com.ran.engine.factories.interpolation.curvecreators;

//
// ...

public class OrientationByTangentAnglesCurveCreator extends AbstractInterpolatedCurveCreator<
    Pair<Quaternion, Double>, Quaternion, SimpleInputParameters> {

    private static final Logger LOG = LoggerFactory.getLogger(
        OrientationByTangentAnglesCurveCreator.class);

    private static final OrientationByTangentAnglesCurveCreator INSTANCE =
        new OrientationByTangentAnglesCurveCreator();

    public static OrientationByTangentAnglesCurveCreator getInstance() {
        return INSTANCE;
    }

    @Override
    public DoubleFunction<Quaternion> interpolateCurve(
        List<Pair<Quaternion, Double>> quaternionsWithTangentAnglesList,
        SimpleInputParameters parameters, int degree) {
        LOG.trace("quaternionsWithTangentAnglesList = {}, parameters = {}, degree = {}",
            quaternionsWithTangentAnglesList, parameters, degree);
        validateVerticesList(quaternionsWithTangentAnglesList);

        double t0 = parameters.getT0();
        double t1 = parameters.getT1();
        int k = quaternionsWithTangentAnglesList.size();
        LOG.trace("t0 = {}, t1 = {}, k = {}", t0, t1, k);

        CurvesDeformationCreator deformationCreator = CurvesDeformationCreator.getInstance();
        OrientationArcsBuilder arcsBuilder = OrientationArcsBuilder.getInstance();
        OrientationBigArcsBuilder bigArcsBuilder = OrientationBigArcsBuilder.getInstance();
        TangentOrientationBuilder tangentBuilder = TangentOrientationBuilder.getInstance();
        TimeMomentsUtil timeMomentsUtil = TimeMomentsUtil.getInstance();
    }
}

```

```

LOG.trace("Before calling OrientationArcsBuilder");
List<OrientationArcsBuilder.Result> smallArcsResults = new ArrayList<>(k - 2);
for (int i = 1; i < k - 1; i++) {
    smallArcsResults.add(arcsBuilder.buildArcsBetweenQuaternionsOnThreeDimensionalSphere(
        quaternionsWithTangentAnglesList.get(i - 1).getLeft(),
        quaternionsWithTangentAnglesList.get(i).getLeft(),
        quaternionsWithTangentAnglesList.get(i + 1).getLeft()
    ));
}
LOG.trace("smallArcsResults = {}", smallArcsResults);

LOG.trace("Before calling OrientationBigArcsBuilder");
List<OrientationBigArcsBuilder.Result> bigArcsResults = new ArrayList<>(k - 1);
for (int i = 0; i < k - 1; i++) {
    bigArcsResults.add(bigArcsBuilder.buildOrientationBigArcsBetweenQuaternions(
        quaternionsWithTangentAnglesList.get(i).getLeft(),
        quaternionsWithTangentAnglesList.get(i + 1).getLeft()
    ));
}
LOG.trace("bigArcsResults = {}", bigArcsResults);

LOG.trace("Before calling TangentOrientationBuilder");
List<TangentOrientationBuilder.Result> tangentBuilderResults = new ArrayList<>(k);
for (int i = 0; i < k; i++) {
    LOG.trace("Point #{} = {}", i, quaternionsWithTangentAnglesList.get(i));
    if (quaternionsWithTangentAnglesList.get(i).getRight() == null) {
        LOG.trace("Tangle angle is null");
        tangentBuilderResults.add(null);
    } else {
        LOG.trace("Processing angles");
        Double forwardAngle = null, backAngle = null;
        if (i + 1 < k) {
            if (quaternionsWithTangentAnglesList.get(i + 1).getRight() != null ||
                i + 1 == k - 1) {
                forwardAngle = Math.abs(bigArcsResults.get(i).getAngle());
            } else {
                forwardAngle = Math.abs(smallArcsResults.get(i).getFirstAngle());
            }
        }
        if (i - 1 >= 0) {
            if (quaternionsWithTangentAnglesList.get(i - 1).getRight() != null ||
                i - 1 == 0) {
                backAngle = Math.abs(bigArcsResults.get(i - 1).getAngle());
            } else {
                backAngle = Math.abs(smallArcsResults.get(i - 2).getSecondAngle());
            }
        }
        LOG.trace("forwardAngle = {}, backAngle = {}", forwardAngle, backAngle);
        tangentBuilderResults.add(tangentBuilder.buildTangentOrientation(
            quaternionsWithTangentAnglesList.get(i).getLeft(),
            quaternionsWithTangentAnglesList.get(i).getRight(),
            forwardAngle,
            backAngle
        ));
    }
}

List<Pair<Double, Double>> rotationAngles = new ArrayList<>(k - 2);
for (int i = 0; i < k - 2; i++) {

```

```

        if (quaternionsWithTangentAnglesList.get(i + 1).getRight() == null) {
            rotationAngles.add(smallArcsResults.get(i).getAngles());
        } else {
            rotationAngles.add(tangentBuilderResults.get(i + 1).getAngles());
        }
    }
    LOG.trace("rotationAngles = {}", rotationAngles);

    LOG.trace("Before building rotationsOnSegments");
    List<DoubleFunction<Quaternion>> rotationsOnSegments = new ArrayList<>(k - 1);
    for (int i = 0; i < k - 1; i++) {
        LOG.trace("Building rotation between points {} and {}: {} and {}",
            i, i + 1, quaternionsWithTangentAnglesList.get(i),
            quaternionsWithTangentAnglesList.get(i + 1));
        if (quaternionsWithTangentAnglesList.get(i).getRight() != null &&
            quaternionsWithTangentAnglesList.get(i + 1).getRight() != null) {
            LOG.trace("Angles are set on both points");
            DoubleFunction<Quaternion> firstDeformedCurve = deformationCreator.deformCurves(
                tangentBuilderResults.get(i).getForwardRotation(),
                bigArcsResults.get(i).getRotation(), degree);
            DoubleFunction<Quaternion> secondDeformedCurve = deformationCreator
                .deformCurvesWithCommonEnd(
                    bigArcsResults.get(i).getRotation(),
                    tangentBuilderResults.get(i + 1).getBackRotation(), degree);
            rotationsOnSegments.add(deformationCreator.deformCurves(
                firstDeformedCurve, secondDeformedCurve, degree));
        } else if (quaternionsWithTangentAnglesList.get(i).getRight() != null) {
            LOG.trace("Angle is set only on the first point");
            if (i == k - 2) {
                rotationsOnSegments.add(deformationCreator.deformCurves(
                    tangentBuilderResults.get(i).getForwardRotation(),
                    bigArcsResults.get(i).getRotation(), degree));
            } else {
                rotationsOnSegments.add(deformationCreator.deformCurves(
                    tangentBuilderResults.get(i).getForwardRotation(),
                    smallArcsResults.get(i).getFirstRotation(), degree));
            }
        } else if (quaternionsWithTangentAnglesList.get(i + 1).getRight() != null) {
            LOG.trace("Angle is set only on the second point");
            if (i == 0) {
                rotationsOnSegments.add(deformationCreator.deformCurvesWithCommonEnd(
                    bigArcsResults.get(i).getRotation(),
                    tangentBuilderResults.get(i + 1).getBackRotation(), degree));
            } else {
                rotationsOnSegments.add(deformationCreator.deformCurvesWithCommonEnd(
                    smallArcsResults.get(i - 1).getSecondRotation(),
                    tangentBuilderResults.get(i + 1).getBackRotation(), degree));
            }
        } else {
            LOG.trace("Angles are not set on both points");
            if (i == 0) {
                rotationsOnSegments.add(smallArcsResults.get(i).getFirstRotation());
            } else if (i == k - 2) {
                rotationsOnSegments.add(smallArcsResults.get(i - 1).getSecondRotation());
            } else {
                rotationsOnSegments.add(deformationCreator.deformCurves(
                    smallArcsResults.get(i - 1).getSecondRotation(),
                    smallArcsResults.get(i).getFirstRotation(), degree));
            }
        }
    }
}

```



```

    }

    List<Double> timeMoments = timeMomentsUtil.countTimeMoments(rotationAngles, t0, t1, k);
    LOG.trace("timeMoments = {}", timeMoments);

    LOG.trace("Before building curveSegments");
    List<DoubleFunction<Quaternion>> curveSegments = new ArrayList<>(k - 1);
    for (int i = 0; i < k - 1; i++) {
        LOG.trace("Building curve between {} and {} points");
        double startTime = timeMoments.get(i);
        double endTime = timeMoments.get(i + 1);
        DoubleFunction<Quaternion> currentRotation = rotationsOnSegments.get(i);
        Quaternion currentOrientation = quaternionsWithTangentAnglesList.get(i).getLeft();
        LOG.trace("startTime = {}, endTime = {}, currentOrientation = {}",
            startTime, endTime, currentOrientation);
        DoubleFunction<Quaternion> curveSegmentWithoutAligning = new DoubleFunction<>() {
            @Override
            public Quaternion apply(Quaternion point) {
                return currentRotation.apply(point).multiply(currentOrientation);
            }
        };
        DoubleFunction<Quaternion> alignedCurveSegment =
            curveSegmentWithoutAligning.superposition(timeMomentsUtil
                .buildAligningFunction(startTime, endTime));
        curveSegments.add(alignedCurveSegment);
    }
    return DoubleMultifunction.makeMultifunction(curveSegments);
}

@Override
protected void validateVerticesList(
    List<Pair<Quaternion, Double>> quaternionsWithTangentAnglesList) {
    super.validateVerticesList(quaternionsWithTangentAnglesList);
    if (quaternionsWithTangentAnglesList.stream().anyMatch(
        quaternionWithTangentAngle -> !quaternionWithTangentAngle
            .getLeft().isIdentity())) {
        throw new InterpolationException("All quaternions must be identity");
    }
}
}

```