МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ  
ВЫСШЕГО ОБРАЗОВАНИЯ  
ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И  
ТЕЛЕКОММУНИКАЦИЙ

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Допускаю к защите

Заведующий кафедрой

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Иномистов В. Ю. /

(подпись) (Ф. И. О.)

**Разработка Android-приложения**

Пояснительная записка дипломной работы

ТПЖА.010551.012 ПЗ

Разработал студент гр. ПМ-51 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Буров С. Д. /

(подпись)

Руководитель преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Белиц А. Б /

(подпись)

Киров 2015

Содержание

[Введение 3](#_Toc422409550)

[1 Обзор аналогов 4](#_Toc422409551)

[1.1 Gradiant Signature Recognition 4](#_Toc422409552)

[1.2 Цели 6](#_Toc422409553)

[2 Математический аппарат 7](#_Toc422409554)

[2.1 Изучение методов получения динамических кривых 7](#_Toc422409555)

[2.2 Замена параметра 12](#_Toc422409556)

[2.3 Задача биометрической верификации подписи 15](#_Toc422409557)

[2.4 Выделение экстремальных точек 16](#_Toc422409558)

[2.5 Вариативность почерка 19](#_Toc422409559)

[3 Структура приложения 20](#_Toc422409560)

[3.1 Параметрически заданные кривые 20](#_Toc422409561)

[3.2 Сравнение подписей 23](#_Toc422409562)

[4 Разработка пользовательского интерфейса 26](#_Toc422409563)

[Заключение 27](#_Toc422409564)

[Приложение А (обязательное) Библиографический список 28](#_Toc422409565)

[Приложение Б (Обязательное) Часть листинга программы 29](#_Toc422409566)

# Введение

Данная работа посвящена разработке Андроид-приложения для распознавания рукописной подписи. Смартфоны с сенсорным экраном получили широкое распространение по всему миру. Высокие технологии всё глубже проникают в нашу жизнь, однако с нами остаётся и наследие прошлых лет. Заимствованный у персональных компьютеров и перенесённый на смартфоны механихм авторизации с использованием паролей является надёжным и проверенным решением. Проблемой этого решения является необходимость использования клавиатурного интерфейса вместо предполагаемного управления жестами.

Одним из альтернативных подходов к авторизации пользователя является биометрическая авторизация. Использование уникальных для каждого человека признаков позволяет вывести средства авторизации на новый уровень эргономичности.

Биометрия подписи выделяется среди остальных биометрических признаков тем, что основана на моторике работы групп мышц. Иными словами это процесс, а не статический признак. Процессом не получится наследить, как отпечатками пальцев, не удастся оставить на фотографии, как снимок роговицы глаза. И в тоже время он является неотъемлемой частью нас, как и любой другой биометрический признак. Биометиря подписи является одной из наиболее защищённых биометрических признаков.

Сенсорные интерфейсы хорошо подходят для ввода подобного рода информации, что может способствовать широкому распростанению технологии авторизации на основе биометрии подписи.

# Обзор аналогов

## Gradiant Signature Recognition

Андроид-приложение Gradiant Signature Recognition основано на Signature Recognition SDK от компании Gradiant и показывает возможности биометрической верификации, основанной на биометрии подписи.

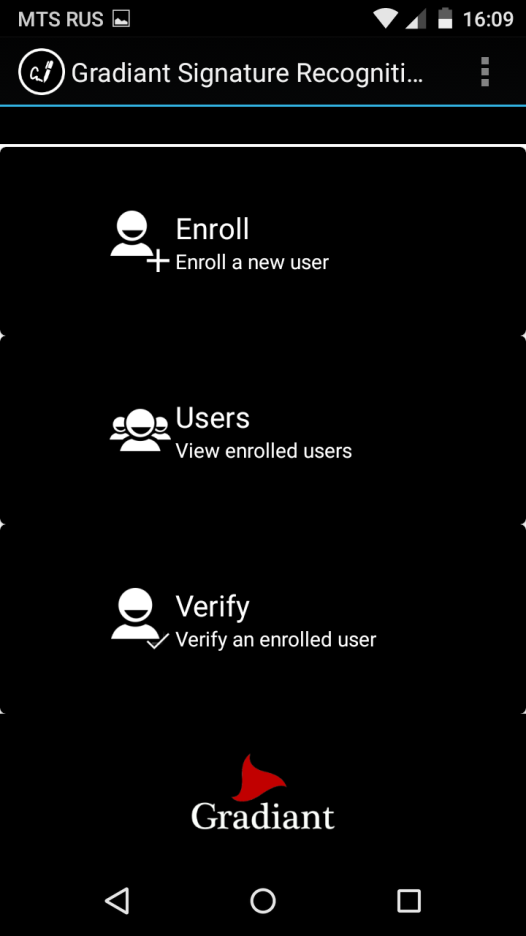


Рисунок 1 - Главное меню программы Graidiant Signature Recognition

Приложение позволяет зарегистрировать человека, введя его имя и три раза повторив подпись на экране смартфона. В случае, если подписи сильно отличаются друг от друга, приложение попросит повторить ввод подписи ещё раз. Подписи зарегистрированных людей хранятся на серверах компании Gradiant. По условиям private policy компания обязуется непредоставлять эти данные кому либо. Данные хранятся один год, но их можно удалить досрочно, удалив зарегистрированного пользователя из приложения.

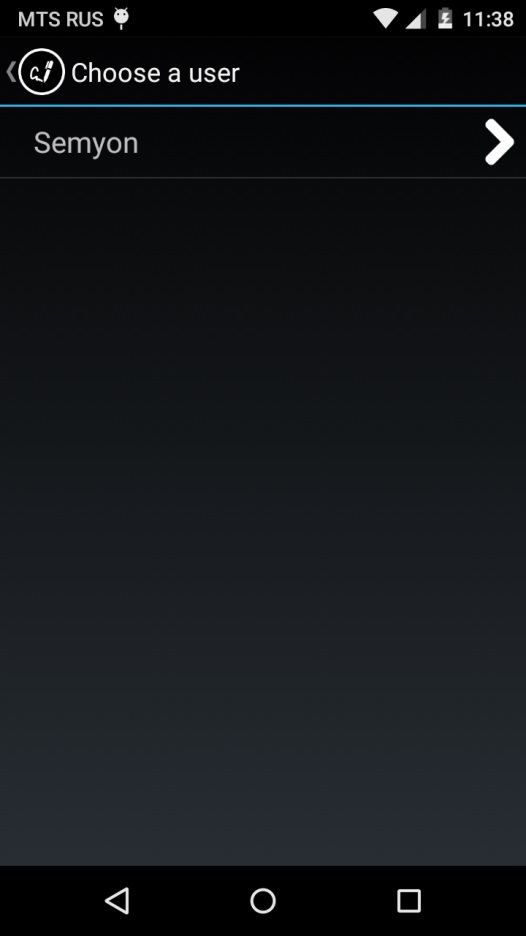


Рисунок 2 - Выбор пользователя для верификации

Для верификации необходимо выбрать соответствующий пункт меню, выбрать человека и ввести подпись в предложенное окно.

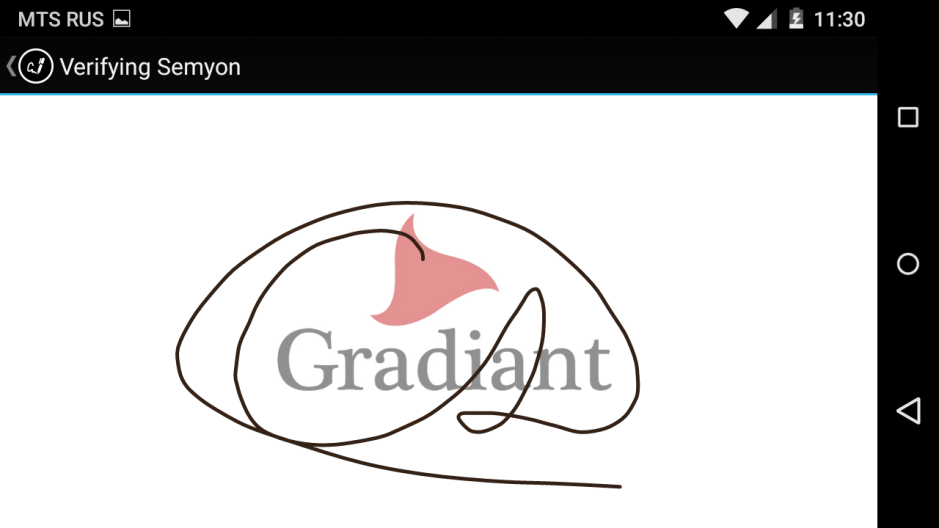


Рисунок 3 - Поле ввода подписи

Возможных исходов три: расписавшийся человек не является владельцем аккаунта; расписавшийся человек является владельцем аккаунта, но подпись непохожа на привязанную к аккаунту; подписи совпадают, расписавшийся человек является владельцем аккаунта.

## Цели

В связи с особенностями дистрибьюции другие аналоги рассмотреть не удалось. Однако, приложение Graidiant Signature Recognition довольно чётко даёт понять, какой должна быть система авторизации, основанная на биометрии подписи. Однако приложение обладает закрытым исходным кодом, и технология сопоставления является тайной.

Приложение, разработанное на открытой технологии способно быстрее развиваться и совершенствовать технологии, непрерывно повышая качество сопоставления и безопасность работы.

# Математический аппарат

## Изучение методов получения динамических кривых

Имеется последовательность точек , принадлежащих некоторой параметрически заданной плоской кривой . – значение параметра кривой соответствующего . Известно, что кривая гладкая. Необходимо интерполировать кривой такой, что имеет непрерывные производные до второго порядка включительно.

Поскольку функции и заданы параметрически, их можно интерполировать независимо друг от друга. Используем для интерполяции кубические сплайны. Данный тип интерполяционных функций обладает непрерывной производной до второго порядка включительно, а также большим количеством способов интерполяции.

Кубический сплайн можно записать, как

, (1)

где , .

Интерполяционный сплайн позволяет хорошо аппроксимировать гладкие функции только если узловые точки приналежат функции или близки к ней. Если исходные данные содержат в себе случайную компоненту, то интерполяционные полиномы будут следовать тем же случайным флуктуациям и дадут неверное представление о природе интерполируемой функции. Кроме того, в случае повышенных требований к гладкости интерполяционных кривых, можно позволить сплайну отклоняться от узловых точек.

Предположим, что ординаты точек получены из следующего выражения:

(2)

где образуют последователность независимых, случайно распределённых величин с .

В этом случае мы можем восстановить построив сплайн , минимизирующий значение следующего выражения:

(3)

где .

Параметр отражает относительное влияние конфликтующих между собой условий: оставаться вблизи узловых точек и получения гладкой аппроксимирующей функции. Заметим, что линейная функция удовлетворяет условию:

, (4)

что свидетельствует о том, что в предельном случае, когда и значение имеет лишь гладкость функции сплайн становится прямой. Сдругой стороны при значение имеет лишь близость к узловым точкам и полученный сплайн будет проходить точно через заданные точки.

Принимая во внимание кусочную природу сплайна можно записать как:

, (5)

А учитывая, что сплайн построен из кубических сегментов вторая производная на любом интервале является линейной функцией:

, (6)

где .

Тогда штрафная функция может быть переписана следующим образом:

, (7)

где .

Рассмотрим случай естественного сплайна, проходящего через точки и удовлетворяющего граничным условиям и . Дополнительной особенностью построения сглаживающего сплайна по сравнению с интерполяционным сплайном является необходимость определения ординат , которые более не являются ординатами .

Для построения кубического сплайна необходимо найти коэффициентов: . Можно сконцентрироваться на задаче определения параметров и , если элиминировать оставшиеся параметры и . Рассмотрим поэтому *i*-тый сегмент, стягивающий разрыв между и и поэтому удовлетворяющий следующим условиям:

1. ; (8)
2. ; (9)
3. ; (10)
4. . (11)

Первое условие можно принять как тождество. По второму условию получаем

, (12)

что позволяет выразить

. (13)

Третье условие вновь примем за тождество. Четвёртое условие позволяет установить

, (14)

из чего получаем:

. (15)

Подставив это в предыдущее уравнение получим окончательную зависимость

. (16)

Теперь и выражены через и . Чтобы определить последние, используем условие непрерывности первой производной для соединения сегментов:

(17)

Заменив и на полученные выше выражения получим:

(18)

Варируя *i* от *1* до *n-1* и учитывая граничные условия получим следующую матричную систему:

(19)

где

(20)

(21)

(22)

Матричное уравнение может быть кратко записано как:

(23)

Сходным образом можно переписать и штрафную функцию:

(24)

где – диагональная матрица .

Подставив

(25)

можно получить штравную функцию, зависящую только от вектора *d*, содержащего ординаты узловых точек:

(26)

Оптимальными значениями орлинат будут те, что минимизируют функцию . Продифференцировав это выражение и приравняв к 0 получим:

(27)

Что является условием минимизации. Из этого можно получить:

(28)

Затем умножим это слева на

, (29)

вновь преобразуем при помощи

(30)

и сгруппируем:

(31)

Где .

Как только эта система будет решена относительно *b*, *d* может быть получено по формуле:

(32)

После этого могут быть получены и оставшиеся коэффициенты [3].

## Замена параметра

### Кривая. Параметрически заданная кривая. Длина кривой. Естественная параметризация

Пусть – евклидово *n*-мерное пространство. – открытый интервал прямой (*I* может совпадать с ).

Элементарной кривой в называется множество точек, гомеоморфное некоторому интервалу *I* числовой прямой .

Множество точек в назовём кривой, если для каждой её точки существует часть кривой, содержащая эту точку,гомеоморфная некоторому интервалу числовой прямой , т. е. В некоторой окрестности каждой своей точки кривая является элементарной.

Пусть – прямоугольная декартова система координат и – координаты точки *p* кривой , – её радиус вектор. Тогда ***r*** как и *p* есть функции параметра , т. е.

(33)

Данное выражение называют векторным уравнением кривой. Разложив по координатным векторам базиса:

(34)

Получим параметрическое уравнение кривой :

(35)

Кривая называется гладкой, если является гладкой функцией, т.е. имеет все производные:

(36)

Которые являются непрерывными функциями, и касательный вектор (вектор скорости) в каждой точке отличен от нулевого

(37)

Вектор является направляющим вектором касательной к кривой.

Пусть кривая задана уравнениями (33) и . Тогда длина дуги кривой , соединяющей точки и определяется интегралом:

(38)

Пусть – фиксированная точка кривой , а – переменная. Обозначим через *s* длину дуги кривой , соединяющей точки и . Тогда имеем:

(39)

т.е. *s* есть функция параметра *t*: . Можно показать, что к ней существует обратная . Подставляя это в векторное уравнение кривой (33) осуществляем замену параметра:

(40)

Где в качестве параметра точки *p* выступает *s* – длина дуги . Такая параметризация называется естественной [4].

### Плоские кривые

Пусть – некоторая кривая на евклидовой плоскости, заданная векторным уравнением (33) или параметрическими уравнениями:

(41)

Длина касательного вектора вычисляется по формуле:

(42)

а длина дуги кривой, соединяющей и примет вид:

. (43)

### Численная репараметризация

Имеется плоская кривая , имеющая непрерывные производные до второго порядка включительно. Функции и являются кубическими интерполяционными сплайнами. Необходимо выполнить замену параметра , где – длина дуги кривой .

Для репараметризации кривой численными методами, построим сеточную функцию . В качестве ординат используем объединение узловых точек сплайнов и . Найдём значение абсцисс численным интегрированием:

(44)

Поскольку интеграл аддитивен по промежутку интегрирования, достаточно найти интеграл независимо для всех отрезков . Теперь, имея сетчатую функцию , можно получить последовательности точек и и интерполировать их. Полученная параметрическая кривая проходит через узловые точки при значении параметра и может считаться естественно параметризованной.

## Задача биометрической верификации подписи

Под динамической кривой подразумевается параметрически заданная кривая . Информация, содержащаяся в данной временной последовательности, отражает динамику мускульных движений руки и поэтому может быть использована как биометрическая характеристика человека.

Динамические кривые могут быть параметризованы различными способами. Наиболее распространённой является временная параметризация. Однако помимо временной параметризации возможна параметризация по длине кривой:

,

что означает постоянство скорости перемещения вдоль траектории.

Также допустива аффинная параметризация по длине кривой, удовлетворяющая условию:

,

что означает постоянство площади паралеллограмма, построенного на векторах и , где параметр *s* находится из условия:

.

Динамические кривые содержат информацию как о динамике движения пера, так и о временной последовательности точек, составляющих траекторию. В данном методе рассматривается только форма кривой, поэтому далее под динамической кривой будет пониматься параметризованная по длине кривая

,

где параметр *s* находится из условия

.

Скорость написания рукописных кривых тесно связана с формой траектории: при выполнении закруглённых движений скорость ниже, чем на прямолинейных участках. Также темп выполнения подписи сильно варьируется от одного исполнения к другому, поэтому отказ от параметризации по времени выглядит оправданным [1].

## Выделение экстремальных точек

Основными методами распознавания рукописной подписи являются: статистический подход, основанный на аппарате скрытых марковских моделей, подход с использованием нейронных сетей, а также сопоставление динамических кривых методом динамической трансформации временной шкалы (dynamic time warping). В зависимости от принципов сопоставления элементов траекторий в литературе встречаются различные вариации последнего метода:

* Symmetric Dynamic Time Warping;
* Continuous Dynamic Time Warping;
* Piecewise Aggregate Approximation Dynamic Time Warping;
* Iterative Deeping Dynamic Time Warping.

В большинстве методов сопоставлению подвергаются все точки траектории. Однако это ведёт к высокой вычислительной стоимости, а также приводит поддельную подпись в соответствие с оригиналом.

Для устранения этих недостатков был предложен метод сравнения подписей на основе экстремальных точек (extreme points warping, EPW). При этом исходный сигнал представляется как череда пиков и впадин, а процесс сравнения существенно ускоряется за счёт необходимости находить соответствие только между точками соответствующего типа [1].

### Вертикальные и горизонтальные экстремумы

Дана плоская, естественно параметризованная кривая . Функции и – субические интерполяционные сплайны:

(45)

где

Необходимо найти вертикальные и горизонтальные экстремумы кривой .

Рассмотрим сегменты, из которых состоят сплайн-функции. Каждый сегмент представляет собой кубический многочлен и используется на определённом интервале. Заметим, что если сплайн имеет экстремум в определённой точке, то его имеет и соответствующий этой точке сегмент сплайна. Верно и обратное: если многочлен, используемый в сегменте сплайна имеет экстреммум на промежутке, на котором этот сегмент используется в сплайне, то и сам сплайн будет иметь экстремум в этой точке.

Таким образом для нахождения всех экстремальных точек сплайна достаточно найти экстремумы всех кубических многочленов, его составляющих и определить, попадают ли они на интервал соответствующего сегмента.

Найдём первую производную и приравняем к 0:

(46)

Решив полученное квадратное уравнение получим точки возможного экстремума сплайна:

(47)

Полученные корни в количестве от нуля до двух проверяются на принадлежность к интервалу, на котором сплайн действителен и в случае попадания в интервал являются экстремумами всего сплайна. Данный подход может быть использован как для поиска горизонтальных, так и вертикальных экстремумов.

### Экстмемум по кривизне

Похожим образом получим и экстремумы по кривизне.

Пусть кривая задана естественной параметризацией:

. (48)

Вектор ускорения – это вектор кривизны кривой, а его длина:

(49)

Кривизна кривой в точке *p*. Радиус кривизны кривой:

(50)

Рассмотрим и на объединённой сетке узлов каждого сплайна. На каждом из промежутков между узлами сегменты сплайнов не меняются, что позволяет говорить о единой функции кривизны для данного промежутка:

(51)

Найдём производную и приравняем к 0:

(52)

Данное уравнение распадается на 2 условия:

(53)

Причём второе условие можно упростить:

(54)

Тогда вся система преобразуется к виду:

(55)

Теперь, зная вид функции, подставим её в уравнения

(56)

(57)

Найдём экстремальные точки:

(58)

(59)

(60)

Теперь вернёмся в корням знаменателя:

(61)

Данное условие выполняется только при равенстве соответствующих коэффициентов сплайнов, что на практике встречается редко. В случае, если оно всё же выполнилось, получаем точку разрыва, в которую не должен попадать корень числителя. Затем необходимо проверить, что корень попадает в интервал, на котором действительна построенная нами функция кривизны.

## Вариативность почерка

Одной из основных проблем распознавания почерка является учёт вариативных изменений кривой. При этом возможны такие локальные трансформации, при которых появляются или исчезают отдельные элементы подписи.

Вариативным изменением 1-го рода будем называть такую вариацию траектории, при которой происходит локальное изменение её формы.

Вариативным изменением 2-го рода будем называть такую вариацию траектории, при которой происходит добавление и удаление её элементов.

Основу метода верификации составляет принцип сопоставления двух кривых на основе нахождения соответствующих пар вертикальных экстремумов, принадлежащих двум образам. При этом, естественно, имеет смысл искать соответствия для вертикальных экстремумов только вида «минимум-минимум» и «максимум-максимум». При рассмотрении вариативных изменений вводится ограничение, что два соответствующих сегмента кривой могут отличаться не более чем на два вертикальных экстремума. На практике это ограничение выполняется практически всегда [2].

### Сопоставление траекторий

Для описания базовых сопоставимых единиц траекторий вводится понятие хорды – это отрезок, соединяющий вертикальные экстремальные точки противоположного типа. На вход алгоритма сопоставления двух динамических кривых поступает набор упорядоченный набор хорд, являющихся допустимыми сочетаниями пар экстремумов. Задачей алгоритма является нахождение оптимального соответствия между экстремальными точками, представляющими концы хорд.

Данная задача решалась методом динамического программирования. Первым этапом алгоритма является заполнение таблицы штрафов, в которой на пересечении строк и столбцов находятся величины, характеризующие степень отличия соответствующих элементов рукописных кривых. При этом применялись две метрики: «манхеттенская» и «по Журавлёву». Первая метрика учитывает положение пары хорд относительно друг друга и относительно места в кривой, а также относительные размеры хорд:

Сопоставление по указанной мере производилось изсоображений, чтобыразличие по одной из координат не приводилок неверному решению, что не исключено при сравнении по евклидовой мере. Однако данные меры не учитывают поведение кривой на участке между двумя вертикальными экстремумами. Поэтому дополнительнобыла введена мера структурной схожести («по Журавлёву») участков траектории, принимающая значение 0 или 1.

Сам алгоритм динамического программирования состоит из двух последовательных действий: прямого прохода, когда ищется локально-оптимальное продолжение траектории и обратного прохода, когда выбирается путь, доставляющий минимальный финальный штраф за несоответствие двух траекторий.

Поскольку при прямом проходе алгоритма имеет смысл находить соответствие только между вертикальными экстремумами одного типа, то из ячейки возможно продолжение в одну из трёх ячеек: , , . В каждой ячейке может находиться до трёх «подтаблиц» содержащих штрафы за сопоставление хорд с учётом вариативной трансформации 1-го рода [2].

### Анализ формы траектории

Алгоритм сопоставления позволяет получить набор гипотез для отбора кандидата на роль эталонной подписи, отсеяв грубые структурные несоответствия. Для тчательного анализа желательно проводить более детальное исследование кривых. Для этого предлагается сравнивать углы между касательными в парах соответствующих друг другу экстремальных точках. Поскольку всё, что требуется получить от касательных – это угол между ними, предлагается вместо них использовать касательный вектор.

# Структура приложения

Разработка приложения проводилась на языке Java, ограничиваясь возможностями шестой версии. Целевым Андроид API было выбрано API версии 19 (Android KITKAT). Архитектура приложения следует принципам объектно ориентированного программирования. Основные классы приложения имеют сложную структуру наследования.

## Параметрически заданные кривые

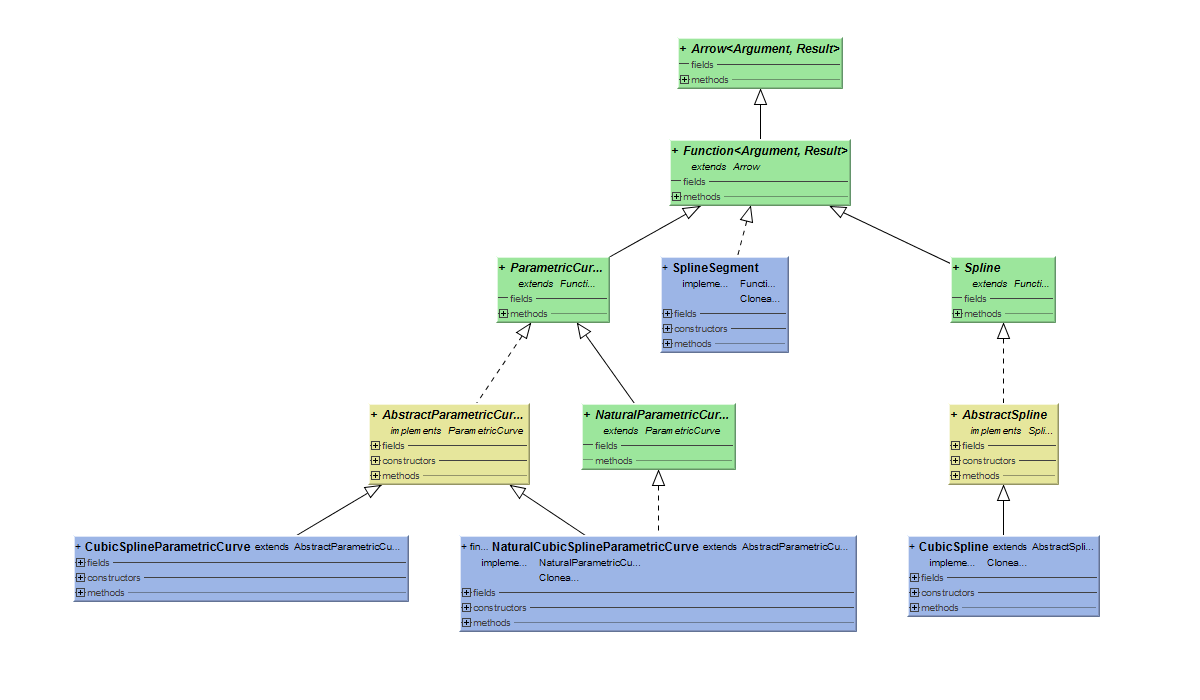


Рисунок 4 - UML диаграма классов параметрически заданных кривых

### Arrow<Argument, Result>

Поскольку в языке Java 6 нет встроенной поддержки лямбда-выражений, приложение получило свою реализацию интерфейсов функций. Во главе иерархии находится параметризированный интерфейс **Arrow**, служащий для представления объекта, умеющего по параметру функции типа **Argument** возвращать результат типа **Result**. Следует отметить, что не правильно считать **Arrow** функцией одной переменной, либо функцией, возвращающей одно значение, поскольку не смотря на то, что функция имеет один параметр и возвращает один результат, **Argument** и **Result** могут представлять собой обёртки, содержащие несколько значений.

Данный интерфейс в силу своей гибкости и простоты используется во многих методах классов в качестве типа входного параметра.

### Function<Argument, Result>

Данный интерфейс является в приложении представлением математической функции. Объект класса, реализующего данный интерфейс, в дополнение к значению имеет и производные.

### SplineSegment implements Function<Double, Double>

Класс **SplineSegment** служит для представления звена сплайновой кривой. Полями этого класса являются степень полинома, массив коэффициентов, и границы, на которых данный сегмент следует применять. Реализация интерфейса **Function** требует реализации методов вычисления значения и производной.

### Иерархия Spline <- AbstractSpline <- CubicSpline

Данная иерархия классов реализует принципы построения сплайновых кривых. Основной интерфейс сплайна объявлен в интерфейсе **Spline**. В **AbstractSpline** реализован механизм проксирования вызовов функций получения значений и производных в соответствующие сегменты сплайна. Класс **CubicSpline** отвечает за контроль порядка сплайна и недопускает добавление сегментов степени выше третьей.

### ParametricCurve extends Function<Double, Point2D>

Класс **ParametricCurve** является базовым интерфейсом в иерархии классов параметрически заданных кривых. Он наследуется от интерфнйса **Function<Double, Point2D>**, что означает, что классы, реализующие данный интерфейс, должны иметь значение и производные в вещественных точках. Класс **Point2D** является представлением точки на двумерной плоскости и хранит две координаты.

### NaturalParametricCurve extents ParametricCurve

Данный интерфейс не вносит дополнительной функциональности. Его наличие обусловлено необходимостью отделения естественно параметризованных кривых в отдельный класс объектов, так как они обладают некоторыми отличительными особенностями. Данный тип гарантирует, что в качестве агрумента будет передаваться естественно параметризованная кривая.

### AbstractParametricCurve implements ParametricCurve

На данном уровне иерархии наследования реализуются методы, унаследованные **ParametricCurve** от **Function**. При этом предполагается, что за значение координаты в точке будет отвечать функция одной переменной (объект, реализующий интерфейс **Function<Double, Double>**).

### CubicSplineParametricCurve и NaturalSplineParametricCurve

Данные классы конкретизируют функции, использующиеся для получения значений координат, поскольку многие из использованных подходов при получении объекта подписи опираются именно на сплайновую природу параметрически заданной функции.

## Сравнение подписей

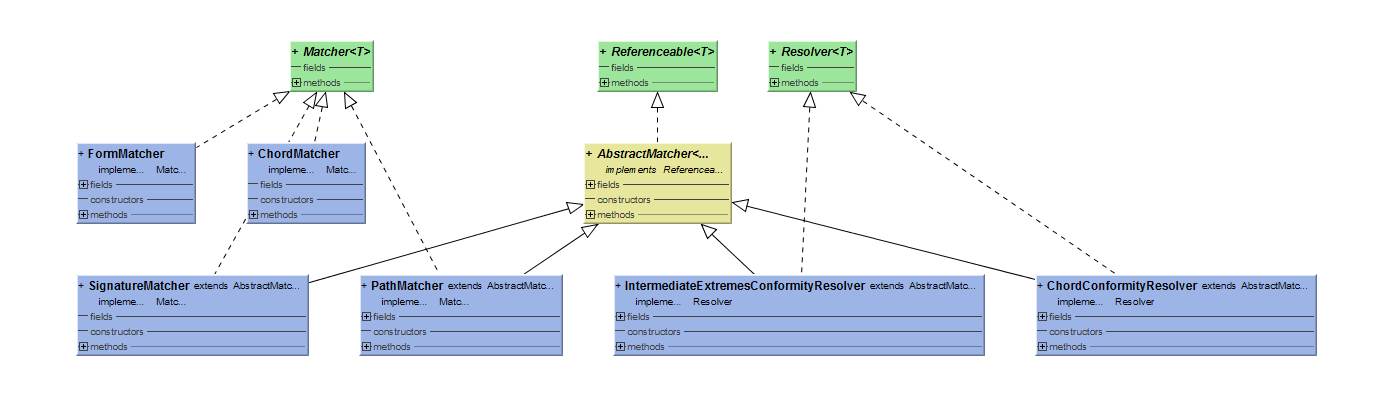


Рисунок 5 - Диаграма классов иерархии Matcher и Resolver

Алгоритм сравнения подписей разбит на несколько шагов. За каждый шаг отвечает отдельный класс, имеющий свои параметры сопоставления. Результатом сопоставления является класс, содержащий в себе результаты сопоставления, а также параметры, при которых данный результат был получен.

### SignatureMatcher

Данный класс является обёрткой для всех остальных сопоставлений. В нём происходит анализ результатов и построение финального ответа.

Первый шаг сопоставления – сопоставление траекторий. Класс **SignatureMatcher** инициализирует экземпляр класса **PathMatcher** и передаёт ему объект подписи для анализа структуры траектории. Результатом сопоставления **PathMatcher** являются оптимальная метрика и карта сопоставления экстремальных точек подписей друг другу. В случае, когда структура сравниваемых подписей совпадает, **SignatureMatcher** инициализирует экземпляр класса **FormMatcher** - следующий шаг сопоставления, осуществляющий анализ формы траектории. Класс **FormMatcher** возвращает информацию о среднем и максимальном расхождении углов касательных векторов, а также индекс хорды, на которой расхождение максимально. Все эти данные и возвращаются **SignatureMatcher** в качестве ответа.

### PathMatcher

На класс **PathMatcher** возложена задача сопоставления структуры траектории и построение карты соответствия экстремальных точек сопоставляемых подписей. Для первоначального сопоставления структуры происходит построение допустимого набора хорд. Задача сопоставления хорд делегируется классу **ChordConformityResolver.** Из полученной в результате сопоставления карты соответствия хорд формируется карта соответствия вертикальных экстремумов. После этого для соответствующих пар экстремумов, уже находящихся в карте, происходит отбор горизонтальных экстремумов, заключённых между ними. Для наборов этих экстремумов также запускается сопоставление, делегируемое классу **IntermediateConformityResolver.** Полученные соответствия добавляются в карту к вертикальным экстремумам, а процесс сопоствления повторяется для экстремумов по кривизне.

В результате формируется карта соответствия экстремумов сопоставляемых подписей, а также метрика, с которой было получено сопоставление вертикальных экстремумов.

### ChordConformityResolver и IntermediateConformityResolver

Данный класс отвечает за нахождение оптимального соответствия между двумя наборами хорд. Для начала процесса оптимизации необходимо построить таблицу штрафов. Этим занимается параметризированный Builder-класс **MetricsTableBuilder**. Сгенерированная с его помощью таблица штрафов (объект класса **MetricsTable**) позволяет произвести оптимизацию и получить список ячеек, через которые проходит оптимальный путь. Затем по списку индексов строится карта соответствий хорд.

Аналогично работает **IntermediateConformityResolver**, но с другими метриками.

### FormMatcher

Все рассмотренные выше классы отвечали за сопоставление структуры траектории. Класс **FormMatcher** отвечает за следуюзий шаг сопоставления - анализ формы траектории.

Последовательно получаем соответствующиепары соседних точек и рассчитываем угол между их касательными векторами. Считаем максимальное расхождение значений углов и среднее. Эти величины и возвращаются в **SignatureMatcher**.

# Разработка пользовательского интерфейса

Пользовательский интерфейс программы минималистичен и содержит лишь элементы, необходимые для демонстрации возможностей распознавания подписи.

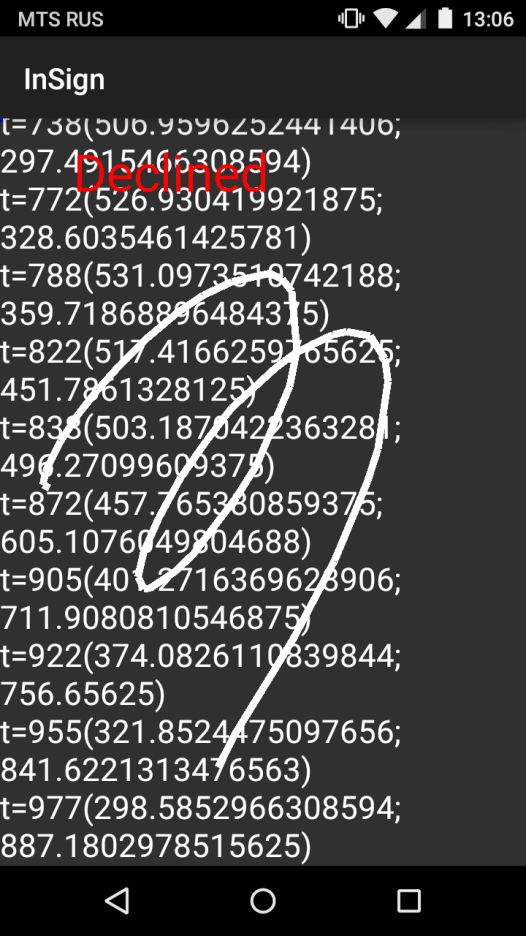


Рисунок 6 - Пользовательский интерфейс

Интерфейс состоит из поля ввода подписи, на котором отображается введённая пользователем подпись. После завершения вычиснений, на экран также выводится результат сопоставления: Accepted или Declined.

# Заключение

Было разработано андроид-приложение для распознавания подписи. В ходе разработки были созданы:

* модуль для матричных операций и решения линейных алгебраических уравнений;
* модуль для интерполяции сглаживающими сплайнами;
* модуль для численного интегрирования;
* модуль для представления параметрически заданных кривых и перехода к естественной параметризации;
* модуль для представления подписи и сопоставления подписей друг другу.

# Приложение А (обязательное) Библиографический список

1. Д.В. Колядин, И.Б. Петров. Алгоритм выделения экстремальных точек применительно к задаче биометрической верификации рукописно подписи// Электронный журнал «Исследовано в России», 47, стр. 532-540, 2005 г. <http://zhurnal.ape.relarn.ru/articles/2005/047.pdf>
2. Д.В. Колядин, И.Б. Петров. Быстрое сопоставление рукописных динамических подписей в биометрической системе контроля доступа// Электронный журнал «Исследовано в России», 83, стр. 870-878, 2005 г. <http://zhurnal.ape.relarn.ru/articles/2005/083.pdf>
3. D.S.G. Pollock. SMOOTHING WITH CUBIC SPLINES// Department of Economics Queen Mary College University of London, 31 стр,
4. Сорокина, М. В. Основы топологии и дифференциальной геометрии в упражнениях и задачах/ М. В. Сорокина/ Учеб-ное пособие для студентов педагогических вузов. – Пенза: Пензенский гос. пед. ун-т им. В. Г. Белинского, 2008 – 80 с.

# Приложение Б (Обязательное) Часть листинга программы

Листинг 1 - Код метода решения СЛАУ методом Гаусса с выбором главного элемента.

**public static** Vector Gauss(Matrix matrix, Vector freeVector) {  
 Objects.*requireNonNull*(matrix, **"Matrix can not be null!"**);  
 Objects.*requireNonNull*(freeVector, **"Vector can not be null"**);  
  
 **if** (matrix.getRowsCount() != matrix.getColumnsCount())  
 **throw new** IllegalArgumentException(**"Numeric solution require square matrix"**);  
 **if** (matrix.getRowsCount() != freeVector.getSize())  
 **throw new** IllegalArgumentException(**"Vector size must match the number of rows in matrix"**);  
  
 Matrix m = matrix.getFactory().newInstance(matrix);  
 Vector v = freeVector.getFactory().newInstance(freeVector);  
  
  
 **int**[] trR = **new int**[m.getRowsCount()],  
 trC = **new int**[m.getColumnsCount()];  
 **for** (**int** i = 0; i < trR.**length**; i++)  
 trR[i] = i;  
 **for** (**int** j = 0; j < trC.**length**; j++)  
 trC[j] = j;  
  
 *//-- Checked: square --* **int** matrixSize = m.getRowsCount();  
 **double** maxElement = 0;  
 **int** iIdx = 0, jIdx = 0;  
 **for** (**int** k = 0; k < matrixSize - 1; k++) {  
 *//-- Abs maximal element finding --* maxElement = -1;  
 iIdx = -1;  
 jIdx = -1;  
 **for** (**int** i = k; i < matrixSize; i++) {  
 **for** (**int** j = k; j < matrixSize; j++) {  
 **if** (maxElement < Math.*abs*(m.get(trR[i], trC[j]))) {  
 maxElement = Math.*abs*(m.get(trR[i], trC[j]));  
 iIdx = i;  
 jIdx = j;  
 }  
 }  
 }  
 **int** temp = trR[k];  
 trR[k] = trR[iIdx];  
 trR[iIdx] = temp;  
 temp = trC[k];  
 trC[k] = trC[jIdx];  
 trC[jIdx] = temp;  
  
 *//-- Subtractions --* **for** (**int** i = k + 1; i < matrixSize; i++) {  
 **double** mi = -m.get(trR[i], trC[k]) / m.get(trR[k], trC[k]);  
 *//-- Matrix --* m.set(trR[i], trC[k], 0);  
 **for** (**int** j = k + 1; j < matrixSize; j++) {  
 **double** Mkj = m.get(trR[k], trC[j]),  
 Mij = m.get(trR[i], trC[j]);  
 m.set(trR[i], trC[j], Mij + mi \* Mkj);  
 }  
 *//-- Vector --* **double** Vk = v.get(trR[k]),  
 Vi = v.get(trR[i]);  
 v.set(trR[i], Vi + mi \* Vk);  
 }  
 }  
  
 Vector x = v.getFactory().newInstance(v.getSize());  
 **for** (**int** i = matrixSize - 1; i >= 0; i--) {  
 **double** summ = v.get(trR[i]);  
 **for** (**int** j = matrixSize - 1; j > i; j--) {  
 **double** Mij = m.get(trR[i], trC[j]);  
 summ -= Mij \* x.get(trC[j]);  
 }  
 x.set(trC[i], summ / m.get(trR[i], trC[i]));  
 }  
 **return** x;  
}

Листинг 2 – Код метода интерполяции сглаживающими сплайнами

**public static** CubicSpline Smoothing(**final** Point2D[] points, **final double** lambda) {  
 *//-- x indexes changes from 0 to n --* **int** n = points.**length** - 1;  
 *//-- h indexes changes from 0 to n-1 --* **int** nh = n - 1;  
 **final double**[] h = **new double**[nh + 1];  
 **for** (**int** k = 0; k <= nh; k++)  
 h[k] = points[k + 1].getX() - points[k].getX();  
 *//-- R indexes changes from 0 to n-2 and from 0 to n-2 --* **int** nR = n - 2;  
 Matrix R = MatrixImpl.***FACTORY***.newInstance(nR + 1, nR + 1);  
 R.set(0, 0, 2.0 \* (h[0] + h[1]));  
 R.set(0, 1, h[1]);  
 **for** (**int** i = 1; i <= nR - 1; i++) {  
 R.set(i, i - 1, h[i]);  
 R.set(i, i, 2.0 \* (h[i] + h[i + 1]));  
 R.set(i, i + 1, h[i + 1]);  
 }  
 R.set(nR, nR - 1, h[nR]);  
 R.set(nR, nR, 2.0 \* (h[nR] + h[nR + 1]));  
  
 *//-- Qt indexes changes from 0 to n-2 and from 0 to n --* **int** nQtRow = n - 2;  
 **int** nQtCol = n;  
 Matrix Qt = MatrixImpl.***FACTORY***.newInstance(nQtRow + 1, nQtCol + 1);  
 **double** rRight = 0;  
 **double** rLeft = 3.0 / h[0];  
 *//-- last index = n-2 --* **for** (**int** i = 0; i <= nQtRow; i++) {  
 rRight = 3.0 / h[i + 1];  
 Qt.set(i, i, rLeft);  
 Qt.set(i, i + 1, -(rLeft + rRight));  
 Qt.set(i, i + 2, rRight);  
 rLeft = rRight;  
 }  
  
 Matrix Q = MatrixImpl.***FACTORY***.newInstance(Qt);  
 Q.transpose();  
  
 **int** nSigma = n;  
 Matrix Sigma = MatrixImpl.***FACTORY***.E(nSigma + 1);  
 **double** sigma = Math.*sqrt*(1.0 / 12.0);  
  
 Vector y = VectorImpl.***FACTORY***.newInstance(n + 1);  
 **for** (**int** k = 0; k < y.getSize(); k++)  
 y.set(k, points[k].getY());  
  
 **double** mu = (2.0 / 3.0) \* (1.0 - lambda) / lambda;  
  
 **double** coeff = mu \* sigma;  
  
 Matrix A = Qt.multiply(Sigma).multiply(Q).multiply(coeff).add(R);  
  
 Vector B = Qt.multiply(y);  
  
 Vector b = LAES.*Gauss*(A, B);  
  
 **final** Vector bExt = VectorImpl.***FACTORY***.newInstance(b.getSize() + 2);  
 bExt.set(0, 0);  
 bExt.set(bExt.getSize() - 1, 0);  
 **for** (**int** k = 0; k < b.getSize(); k++)  
 bExt.set(k + 1, b.get(k));  
  
 **final** Vector d = y.add(Sigma.multiply(Q).multiply(b).multiply(coeff).negate());  
  
 Coefficient a = **new** Coefficient() {  
 @Override  
 **public** Double valueIn(Integer k) {  
 **return** (bExt.get(k + 1) - bExt.get(k)) / (3.0 \* h[k]);  
 }  
 };  
 Coefficient c = **new** Coefficient() {  
 @Override  
 **public** Double valueIn(Integer k) {  
 **return** (d.get(k + 1) - d.get(k)) / h[k] - (1.0 / 3.0) \* (bExt.get(k + 1) + 2.0 \* bExt.get(k)) \* h[k];  
 }  
 };  
  
 **double**[] coefficients = **new double**[] {  
 d.get(0),  
 (d.get(1) - d.get(0)) / h[0] - (1.0 / 3.0) \* (bExt.get(1) + 2.0 \* bExt.get(0)) \* h[0],  
 bExt.get(0),  
 (bExt.get(1) - bExt.get(0)) / (3.0 \* h[0])};  
 CubicSpline spline = **new** CubicSpline(coefficients, points[0].getX(), points[1].getX());  
  
 **for** (**int** k = 1; k <= n - 1; k++) {  
 coefficients[0] = d.get(k);  
 coefficients[1] = (d.get(k + 1) - d.get(k)) / h[k] - (1.0 / 3.0) \* (bExt.get(k + 1) + 2.0 \* bExt.get(k)) \* h[k];  
 coefficients[2] = bExt.get(k);  
 coefficients[3] = (bExt.get(k + 1) - bExt.get(k)) / (3.0 \* h[k]);  
 spline.addRight(coefficients, points[k + 1].getX());  
 }  
  
 **return** spline;  
 }  
}

Листинг 3- Код метода интергирования методом Гаусса-Лежандра

**private static class** LinearVariableReplacement **implements** Arrow<Double, Double> {  
 **private double lowerLimit**, **upperLimit**;  
 **public** LinearVariableReplacement(**double** lowerLimit, **double** upperLimit) {  
 **this**.**lowerLimit** = lowerLimit;  
 **this**.**upperLimit** = upperLimit;  
 }  
 @Override  
 **public** Double valueIn(Double x) {  
 **return** (**upperLimit** - **lowerLimit**) / 2.0 \* x + (**upperLimit** + **lowerLimit**) / 2.0;  
 }  
  
 **public** Double integralCorrectionCoefficient() {  
 **return** (**upperLimit** - **lowerLimit**) / 2.0;  
 }  
}  
  
**private static double** integrate(**final** Knot[] knots, **final** Arrow<Double, Double> arrow) {  
 **double** integral = 0;  
 **for** (**int** k = 0; k < knots.**length**; k++) {  
 integral += knots[k].**weight** \* arrow.valueIn(knots[k].**point**);  
 }  
 **return** integral;  
}  
  
**public static double** integrate(**final int** pointCountRule, **final** Integral integral, **final double** precision) {  
 **if** (pointCountRule < 1 || pointCountRule > ***KNOTS***.**length** - 1)  
 **throw new** IllegalArgumentException(**"pointCountRule should be more or equals 1 and less or equals "** + (***KNOTS***.**length** - 1));  
 **double** integralValue = -precision \* 2, integralPrevValue = 0;  
 **int** intervalCount = ***DEFAULT\_INTERVAL\_COUNT***;  
 **final double** integrationInterval = integral.getUpperLimit() - integral.getLowerLimit();  
 **while** (Math.*abs*(integralValue - integralPrevValue) > precision / 2.0) {  
 integralPrevValue = integralValue;  
 integralValue = 0;  
 **double** step = integrationInterval / intervalCount;  
 **double** xStart, xEnd = integral.getLowerLimit();  
 **for** (**int** k = 0; k < intervalCount; k++) {  
 xStart = xEnd;  
 xEnd += step;  
 **final** LinearVariableReplacement variableReplacement = **new** LinearVariableReplacement(xStart, xEnd);  
 **final** Arrow<Double, Double> integratedArrow = **new** Arrow<Double, Double>() {  
 @Override  
 **public** Double valueIn(Double x) {  
 **return** integral.getArrow().valueIn(variableReplacement.valueIn(x));  
 }  
 };  
 integralValue += variableReplacement.integralCorrectionCoefficient() \* *integrate*(***KNOTS***[pointCountRule], integratedArrow);  
 }  
 intervalCount \*= ***INTERGAL\_STEP\_MULTIPLIER***;  
 }  
 **return** (integralValue + integralPrevValue) / 2.0;  
}

Листинг 4 – Код метода численной репараметризации

**public static** NaturalCubicSplineParametricCurve fromCurve(**final** ParametricCurve curve, \_\_<AbstractSpline> naturalParametrization) {  
 Arrow<Double, Double> arcLengthFunction = **new** Arrow<Double, Double>() {  
 @Override  
 **public** Double valueIn(Double x) {  
 Point2D point = curve.derivative(1, x);  
 **return** Math.*sqrt*(point.getX() \* point.getX() + point.getY() \* point.getY());  
 }  
 };  
 Point2D[] ts = **new** Point2D[***NATURAL\_PARAMETRIZATION\_KNOTS\_COUNT***];  
 ts[0] = **new** Point2D(0, 0);  
 **double** actualStep = (curve.getParameterMax() - curve.getParameterMin()) / (***NATURAL\_PARAMETRIZATION\_KNOTS\_COUNT*** - 1);  
  
 **for** (**int** k = 1; k < ***NATURAL\_PARAMETRIZATION\_KNOTS\_COUNT***; k++) {  
 **double** t = ts[k - 1].getY() + actualStep;  
 Integral arcSegment = **new** Integral(arcLengthFunction, ts[k - 1].getY(), t);  
 **double** value = Intergrate.GaussLegendre.*fivePointRule*(arcSegment, ***NATURAL\_PARAMETRIZATION\_INTEGRAL\_PRECISION***);  
 ts[k] = **new** Point2D(value + ts[k - 1].getX(), t);  
 }  
  
 **if** (naturalParametrization != **null**) {  
 AbstractSpline tsSpline = Interpolation.Splines.*Smoothing*(ts, 1);  
 naturalParametrization.setRef(tsSpline);  
 }  
  
 Point2D[] xs = **new** Point2D[***NATURAL\_PARAMETRIZATION\_KNOTS\_COUNT***],  
 ys = **new** Point2D[***NATURAL\_PARAMETRIZATION\_KNOTS\_COUNT***];  
  
 **for** (**int** k = 0; k < ts.**length**; k++) {  
 Point2D point = curve.valueIn(ts[k].getY());  
 xs[k] = **new** Point2D(ts[k].getX(), point.getX());  
 ys[k] = **new** Point2D(ts[k].getX(), point.getY());  
 }  
  
 CubicSpline xsSpline = Interpolation.Splines.*Smoothing*(xs, 1);  
 CubicSpline ysSpline = Interpolation.Splines.*Smoothing*(ys, 1);  
  
 NaturalCubicSplineParametricCurve naturalParametricCurve = **new** NaturalCubicSplineParametricCurve(xsSpline, ysSpline, xsSpline.getLeftBound(), xsSpline.getRightBound());  
  
 **return** naturalParametricCurve;  
}