Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Факультет «Специальное машиностроение»
Кафедра СМ-7 «Робототехнические системы и мехатроника»

РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ НА ТЕМУ:

Расчёт траекторий движения промышленного манипуляционного робота на основе нейронных сетей

Студент _____ _____ _____
(Группа) (Подпись, дата) (И.О. Фамилия)

Научный руководитель _____ _____
(Подпись, дата) (И.О. Фамилия)

Москва, 2024 г.

РЕФЕРАТ

Пояснительная записка к научно-исследовательской работе содержит 23 страницы, 3 рисунка, 2 таблицы, 14 использованных источников, 10 приложений.

Ключевые слова: промышленная робототехника, нейронные сети математическое моделирование, компьютерное моделирование.

Работа рассматривает применение нейронных сетей в алгоритме управления промышленным манипуляционным роботом для расчёта оптимальной траектории движения. Поставлены цели и задачи исследования, проведён анализ предметной области и научной литературы, проведён обзор средств компьютерного моделирования, сформулированы требования к разрабатываемому программному обеспечению для работы с моделью робота-манипулятора.

Содержание

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ПМР – промышленные манипуляционные роботы

НС – нейронные сети

РТК – робототехнический комплекс

ПЗК – прямая задача кинематики

ОЗК – обратная задача кинематики

ОЗД – обратная задача динамики

TCP – Tool center point

BFGS – Broyden-Fletcher-Goldfarb-Shanno

ВВЕДЕНИЕ

Согласно словарю Вебстера, *робот – автоматическое устройство, выполняющее функции, обычно приписываемые человеку*. Более точно промышленный робот можно охарактеризовать как: перепрограммируемый многофункциональный манипулятор, предназначенный для осуществления различных заранее заданных перемещений материалов, деталей, инструментов или специальных приспособлений с целью выполнения различных работ. Итак, робот представляет собой перепрограммируемый универсальный манипулятор, снабженный внешними датчиками и способный выполнять различные производственные задачи. Это определение предполагает наличие у робота интеллекта, обусловленного заложенными в компьютер алгоритмами систем управления и очувствления. Следовательно, промышленный робот представляет собой универсальный, оснащенный компьютером манипулятор, состоящий из нескольких твердых звеньев, последовательно соединенных вращательными или поступательными сочленениями. [1]

Промышленные манипуляционные роботы (ПМР) способны выполнять различные операции по перемещению, сборке, сварке, покраске и другим видам обработки объектов. ПМР широко используются в современном производстве, так как они повышают производительность, качество и безопасность труда. Однако, для эффективного использования ПМР необходимо решать ряд сложных задач, связанных с планированием и управлением их движения.

Одной из таких задач является расчёт траекторий движения ПМР, то есть определение последовательности положений и ориентаций звеньев робота в пространстве, которые обеспечивают выполнение заданной операции. Эта задача имеет большое практическое значение, так как от качества траектории зависят скорость, точность и безопасность движения робота. Традиционные методы расчёта траекторий основаны на аналитических или численных алгоритмах, которые требуют знания геометрии, кинематики и динамики робота, а также учёта механических и программных ограничений. Эти методы имеют ряд недостатков, таких как высокая вычислительная сложность, низкая универсальность и адаптивность, чувствительность к ошибкам измерения и настройки параметров.

В связи с этим, в последнее время активно развиваются альтернативные подходы к расчёту траекторий движения ПМР, основанные на использовании нейронных сетей. Нейронные сети (НС) – искусственные, многослойные высокопараллельные логические структуры, составленные из формальных нейронов.[2] НС способны обучаться на данных и аппроксимировать сложные нелинейные зависимости. Преимущества НС заключаются в том, что они не требуют явного знания модели робота и ограничений, а также могут адаптироваться к изменяющимся условиям и обобщать результаты на новые ситуации. Существует множество видов НС, различающихся по архитектуре, функциям активации, алгоритмам обучения и применению. В зависимости от поставленной задачи, могут использоваться разные типы НС, такие как многослойные, рекуррентные, свёрточные, глубокие нейронные сети.[3]

Целью данной работы является **разработка методов применения нейронных сетей для расчёта траекторий движения промышленного манипуляционного робота**.

Для достижения этой цели были поставлены следующие задачи:

1. Изучить основные принципы и методы компьютерного моделирования кинематики и динамики ПМР, а также существующие программные средства для этой цели.
2. Разработать программное обеспечение для компьютерного моделирования работы ПМР по следующему алгоритму:
    а) спроектировать архитектуру и функциональность системы, учитывая требования к надёжности, производительности, удобству и гибкости;
    б) выбрать подходящие технологии и инструменты для реализации системы, такие как языки программирования, библиотеки, фреймворки и среды разработки;
    в) разработать модули системы, отвечающие за ввод исходных данных, расчёт кинематических и динамических параметров, визуализацию результатов и взаимодействие с пользователем;
    г) интегрировать модули системы в единое приложение и провести его отладку и тестирование на различных примерах движения ПМР;
3. Провести обзор существующих методов расчёта траекторий движения ПМР, основанных на НС, и выделить их достоинства и недостатки.
4. Выбрать подходящий тип НС и разработать алгоритм её обучения на основе синтетических и реальных данных о движении ПМР.
5. Реализовать программную модель НС и провести её тестирование на различных сценариях движения ПМР.
6. Сравнить полученные результаты с традиционными методами расчёта траекторий и оценить эффективность применения НС.

ОСНОВНАЯ ЧАСТЬ

1 ИССЛЕДОВАТЕЛЬСКАЯ ЧАСТЬ

1.1 **Определение области применения разрабатываемого алгоритма**

Нейронные сети в робототехнике используются на разных уровнях управления. Их можно применять для контроля отдельного привода в качестве нечёткого регулятора следящего устройства [4], для создания человеко-машинного интерфейса при управлении роботом при помощи жестов [5], применять в качестве основы для машинного зрения и управлять робототехническим комплексом, состоящим из нескольких ПМР и их оснастки [6].

В рамках данной работы исследуются возможности использования нейронных сетей для поиска оптимальных траекторий движения манипуляционных роботов различных конфигураций в промышленных условиях.

1.2 **Обзор существующих примеров использования НС для управления ПМР**

Опыт применения ПМР в условиях производства показывает следующие значимые проблемы:

1) **сложность учёта препятствий для планирования траекторий движения,** что приводит к сталкиванию робота с его оснасткой, станками и прочим оборудованием;
2) **изменение динамических параметров манипулятора во время работы,** приводящее к ошибкам при точном позиционировании и сказывающееся на быстродействии робота;
3) **попадание робота в сингулярную конфигурацию осей,** приводящее к резкому движению робота в местах сочленений и, следовательно, к ошибкам при учёте скорости движения программой управления;
4) **невозможность прохождения по траектории без остановки и изменения конфигурации** из-за механических ограничениях в осях.

Третья и четвертая проблемы связаны с механическим исполнением робота, количеством степеней подвижности робота и расположением осей манипулятора друг относительно друга.

Попытки обойти механические ограничения степеней подвижности манипулятора путём добавления избыточной подвижности манипулятору для достижения необходимой непрерывности траектории технологического процесса приводит к появлению сингулярностей, когда избыточность подвижности приводит к множеству способов достижения желаемой точки, что делает траекторию робота менее предсказуемой.

Изменение расположения степеней подвижности манипулятора или добавление дополнительных ограничений на них приводит к уменьшению коэффициента сервиса в точках области достижимости и, следовательно, увеличивает количество случаев невозможных непрерывных траекторий.

Для решения данных проблем возможно использование нейронных сетей в алгоритмах управления роботами. При исследовании литературы найдены следующие примеры решения подобных задач с применением алгоритмов на основе НС.

В работе М. М. Кожевникова, А. П. Пашкевича, О. А. Чумакова ”Планирование траекторий промышленных роботов-манипуляторов на основе нейронных сетей” [7] предложен новый метод планирования траекторий роботов-манипуляторов в рабочей среде с препятствиями, основанный на использовании топологически упорядоченной нейронной сети. Метод позволяет эффективно учесть сложную форму препятствий в промышленных роботизированных комплексах и обеспечивает приемлемое для практики количество тестов на столкновение.

В статье ”Robust Adaptive Sliding Mode Neural Networks Control for Industrial Robot Manipulators” [8] предлагается адаптивный контроллер с использованием нейронных сетей с радиально-базисной функцией активации для промышленных роботов-манипуляторов в условиях неизвестных динамических характеристик манипулятора. Эта предлагаемая структура управления сочетает в себе метод скользящего режима, аппроксимацию функции активации и адаптивный метод для повышения высокой точности следящего управления. Предлагаемый алгоритм на базе НС может успешно решать небольшие задачи благодаря своей простой структуре, более быстрым законам обновления обучения и лучшей аппроксимации для неизвестных динамических параметров манипулятора. Все параметры предлагаемой системы управления определяются теоремой устойчивости Ляпунова и настраиваются в режиме онлайн с помощью алгоритма адаптивного обучения. Таким образом, стабильность, надежность и желаемые характеристики отслеживания НС для ПМР гарантированы. Моделирование и эксперименты, выполненные на трехзвенном ПМР, предлагаются в сравнении с пропорциональным интегрально-дифференциальным контроллером и управлением на базе адаптивной нечетко-логической системы управления.

Для обхода проблем 3 и 4 предлагается использование манипуляторов с избыточной подвижностью, рассчитывающих свои траектории движения на основе сложных самообучающихся алгоритмов учитывающих конфигурацию манипулятора на несколько опорных точек вперёд текущей и подстраивающих своё движение для наиболее эффективного прохождения траектории.

Как показывает следующее исследование [9] современные алгоритмы кинематического управления ПМР на основе нейронных сетей достигают большей производительности при контролируемой точности, чем традиционные алгоритмы, а также не имеет значительной зависимости от сложности конструкции и количества степеней подвижности манипулятора.

В рамках данной работы разрабатывается алгоритм управления ПМР для решения перечисленных проблем. Основой этого алгоритма является самообучающийся контроллер на базе НС, используемый для расчёта оптимальной траектории движения. Данный контроллер можно разделить на основные три уровня:

- **определение формы траектории движения, опорных точек на ней, направление и величину скорости инструмента в них –** уровень, на котором по заданному пользователем референсу траектории или по заданным основным точкам обрабатываемой детали строится математическое представление траектории движения;
- **поиск всех возможных решений ОЗК для опорных точек и связывание их в единое перемещение –** уровень работы нейросетевого алгоритма, оценивающего различные конфигурации ПМР в опорных точках по параметрам весовой функции;
- **контроль актуаторов степеней подвижности для обеспечения необходимых кинематических параметров движения –** уровень управления приводами на основе стандартных регуляторов или нечётких алгоитмов.

Для обеспечения точности работы нейросетевого алгоритма необходима тренировка НС и подбор параметров учёта входных данных. Тренировка заключается в обработке массива данных о перемещении манипулятора под действием управляющего воздействия, эти данные можно получить анализируя движения реального робота или проводя компьютерное моделирование в современных программных пакетах симуляции физических процессов и работы механизмов.

1.3 Обзор существующих программных комплексов для моделирования кинематики и динамики работы ПМР

Для использования программных комплексов моделирования работы ПМР в качестве среды обучения нейронных сетей данные пакеты должны отвечать ряду условий:

1) **точно воспроизводить динамику ПМР в процессе работы** для обеспечения корректности работы обученного алгоритма в реальных условиях;
2) **иметь возможность добавления кинематических схем или настройки существующих** для обучения алгоритма на широкой номенклатуре ПМР;
3) **иметь графическую и численную визуализацию изменяющихся физических параметров ПМР** для возможности отслеживания прогресса обучения и отладки работы алгоритма;
4) **иметь поддержку современных стандартов свободно используемых языков программирования;**
5) **предоставлять инструменты для построения и настройки нейронных сетей.**

На данный момент существует множество программ и программных пакетов для симуляции работы робототехнических комплексов. В рамках работы рассматриваются следующие образцы ПО:

1. **Siemens Tecnomatix** [10] – программное обеспечение,позволяющее спроектировать роботизированные производственные ячейки, оптимизировать их производительность, автономно запрограммировать ПМР и промоделировать их работу.

   Плюсы:

   - точное моделирование кинематики и динамики ПМР из библиотеки программы;
   - возможность полной симуляции работы с учётом тонкостей техпроцесса, что позволяет легко проектировать РТК.

   Минусы:

   - поддерживает только номенклатуру роботов от партнёров компаний-партнёров Siemens;
   - использует проприетарный язык SIRL (Siemens Industrial Robot Language);
   - нет возможности программирования в сторонних средах и, следовательно, нет инструментов работы с НС.
2. **Visual Components** [11] – это расширенный набор для проектирования и моделирования производственных линий. Можно моделировать и анализировать целые производственные процессы, включая робототехническое оборудование, материальный поток, действия человека-оператора и многое другое.

   Плюсы:

   - хорошие графические возможности визуализации;
   - обширная библиотека ПМР и компонентов РТК;
   - наличие пакетов расширения для расчётов различных параметров движения.

   Минусы:

   - устаревший стандарт языка Python, что ограничивает номенклатуру возможных библиотек для использования;
   - необходимость в дополнительном пакете для моделирования динамики, который требует дополнительного изучения документации для корректного использования.
3. **RoboDK** [12] – инструмент оффлайн программирования для промышленных роботов. Он позволяет создавать программы для робота с использованием Python или задавать движение визуально благодаря интегрированной среде 3D-моделирования.

   Плюсы:

   - возможность визуального программирования ПМР;
   - поддержка скриптовых языков программирования и языков программирования производителей роботов;
   - возможность использования моделей ПМР, созданных в программах моделирования.

   Минусы:

   - отсутствие поддержки быстрых языков программирования, что уменьшает быстродействие системы.

4. **SprutCAM Robot** [13] – программное обеспечение для оффлайн программирования российского производства, позволяющее проектировать робототехнические ячейки и комплексы, рассчитывать траектории движения при различных технологических процессах для роботов от разных производителей

   Плюсы:

   - для программирования ПМР доступен специальный функционал: контроль столкновений, обход зоны сингулярности, контроль пределов рабочей зоны;
   - используя стандартные шаблоны, можно быстро создавать собственные кинематические модели роботизированных ячеек.

   Минусы:

   - нет возможности работы со стандартными языками программирования
   - сложность подключения НС в качестве системы управления.

По результатам обзора получаем следующую сравнительную таблицу:

Таблица 1.1 Сравнительная таблица пакетов компьютерного моделирования

	Siemens Tecnomatix	**Visual Components**	**RoboDK**	**SprutCAM Robot**
Точное моделирование динамики ПМР	+	+ При использовании дополнений	+ При использовании дополнений	+
Возможность добавления настраиваемых кинематических схем	-	+	+	+
Визуализация результатов моделирования в виде графиков	-	-	+ При использовании дополнений	+
Возможность работы с открытыми языками программирования современных стандартов	-	-	+	-
Наличие инструментов работы с нейронными сетями	-	+ При использовании дополнений	-	-

1.4 Определение требований для разрабатываемого программного пакета для моделирования

Ни в одном из перечисленных выше пакетов компьютерного моделирования не представлен в полной мере функционал для разработки и тестирования исследуемого алгоритма управления ПМР. В связи с этим принято решение о разработке программного обеспечения с использованием методов симуляции из рассмотренных приложений и научной литературы.

Структура приложения показана в таблице:

Таблица 1.2 Структура разрабатываемого программного комплекса

<table>
<tr><td colspan="6">Приложение: «CAR – Computer Aided Robotics»</td></tr>
<tr><td rowspan="3">Основные математические функции</td><td colspan="3">Функции работы с роботом</td><td colspan="2">Интерфейс пользователя</td></tr>
<tr><td>Задание траектории</td><td>Определение конфигурации робота</td><td>Управление приводами</td><td rowspan="2">Основной интерфейс</td><td rowspan="2">Средства отображения 3Д моделей</td></tr>
<tr><td>Базовые движения: PTP, LIN, CIRC; Сплайновые функции; Разбиение траекторий</td><td>Решение ОЗК и ОЗД; Использование НС для нахождения траектории.</td><td>Регуляторы, корректирующие звенья.</td></tr>
</table>

В качестве основного языка программирования выбран объектно-ориентируемый язык C++ с системой сборки CMake и компилятором CLang 16.0.5.

В качестве языка взаимодействия оператора с роботом и для настроек параметров нейронной сети выбран высокоуровневый язык Python 3.

Ограничения возможностей ПО

Область использования данного программного обеспечения ограничена следующими параметрами:

1) использование только разомкнутых кинематических цепей;
2) оси степеней подвижности в начальной конфигурации параллельны мировым осям XYZ;
3) алгоритм избегания столкновения лишь со статическими препятствиями и с подвижными устройствами, находящимися в одной сети обмена информацией.

2 ПРАКТИЧЕСКАЯ ЧАСТЬ

В основе пакета компьютерного моделирования лежат следующие программные блоки:

- **базовые математические функции,** используемые для работы с матрицами и векторами, для расчёта положения объектов в трёхмерном пространстве, для решения функций минимизации;
- **программное представление кинематической структуры ПМР;**
- **решение ОЗК** для нахождения вариантов пространственной конфигурации звеньев ПМР при достижении точки в пространстве;
- **средства отображения трёхмерной графики;**
- **пользовательский интерфейс на основе оконного приложения.**

2.1 **Реализация базовых математических функций**

Для реализации кинематической схемы моделируемого промышленного робота необходимо использование методов линейной алгебры, векторной геометрии и матричных уравнений при расчёте положений звеньев манипулятора. Для использования перечисленных математических функций требуется использование сторонних математических библиотек, а также написания собственных расчётных функций.

2.1.1 **Использованные внешние библиотеки**

В качестве библиотеки для линейной алгебры, векторных и матричных вычислений используется C++ библиотека Eigen[14].

Eigen – библиотека линейной алгебры для языка программирования C++ с открытым исходным кодом. Написана на шаблонах и предназначена для векторно-матричных вычислений и связанных с ними операций.

Eigen не имеет никаких зависимостей, кроме стандартной библиотеки C++ и использует систему сборки CMake, поддерживает множество компиляторов C++.

Eigen – библиотека с открытым исходным кодом, бесплатно распространяющаяся для ознакомительных, учебных и коммерческих целей.

Из библиотеки функций Eigen используются:

- поддержку комплексных чисел в вычислениях;
- структура ”вектор”, поддерживающая векторы динамического размера и математические операции над ними;
- шаблон универсальной матрицы, подходящий для многомерных вычислений.

2.1.2 **Реализованные математические функции**

При решении поставленной задачи были реализованы следующие математические функции:

- Функции **DegToRad** и **RadToDeg** для перевода значений угла из градусов в радианы и обратно. В качестве аргумента и вывода функции используется тип числа с плавающей точкой.
- Функция вычисления проекции одного вектора на другой **projVector**.

$$l = \frac{\bar{a} \cdot \bar{b}}{|\bar{b}|} \tag{2.1}$$

В качестве аргументов $\bar{a}$ и $\bar{b}$ уравнения (2.1) принимаются два трёхмерных вектора, а выводится длина l проекции в миллиметрах в виде числа с плавающей точкой. Функция может обрабатывать исключение и выдавать ошибку при подстановки в $\bar{b}$ нулевого вектора.

- Функция вычисления угла между векторами **getAngle**.

$$\alpha = \arccos \frac{\bar{a} \cdot \bar{b}}{|\bar{a}| \cdot |\bar{b}|} \tag{2.2}$$

В качестве аргументов $\bar{a}$ и $\bar{b}$ уравнения (2.2) принимаются два трёхмерных вектора, а выводится угол α между векторами в радианах в виде числа с плавающей точкой. Функция может обрабатывать исключение и выдавать ошибку при подстановки в $\bar{a}$ или $\bar{b}$ нулевого вектора.

- Функция вычисления угла между векторами с учётом направления вращения вокруг заданной оси **getAngleAroundAxis**.
  На первом шаге вычисляются новые вектора $\bar{a_1}$ и $\bar{b_1}$, являющиеся проекциями векторов на плоскость перпендикулярную оси $\bar{Axis}$
  Дальше вычисляется угол между двумя проекциями:

$$\alpha = \arccos \frac{\bar{a_1} \cdot \bar{b_1}}{|\bar{a_1}| \cdot |\bar{b_1}|} \tag{2.3}$$

Последним шагом вычисляется векторное произведение $\bar{c} = \bar{a_1} \times \bar{b_1}$, по направлению которого относительно $\bar{Axis}$ определяется знак ответа.
В качестве аргументов $\bar{a}$, $\bar{b}$ и $\bar{Axis}$ уравнения (2.3) принимаются три трёхмерных вектора – два вектора для сравнения и ось поворота, а выводится угол α с учётом знака в радианах в виде числа с плавающей точкой. Функция может обрабатывать исключение и выдавать ошибку при подстановки в $\bar{a}$, $\bar{b}$ или $\bar{Axis}$ нулевого вектора, а также при подстановки векторов, параллельных оси $\bar{Axis}$.

- Функции **Rx**, **Ry** и **Rz** для получения матриц поворота относительно осей **X**, **Y** и **Z** соответственно. В качестве аргумента функции используется α – угол поворота вокруг оси используя тип числа с плавающей точкой.

$$Rx = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix} \tag{2.4}$$

$$Ry = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix} \tag{2.5}$$

$$Rz = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.6}$$

- Функция **calcJacobian** для вычисления по данной дифференцируемой многомерной функции $Y = F(X)$ и известной функцией подсчёта ошибки, в инициирующей точке X_0, с заданной точностью e.

$$J = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \cdots & \frac{\partial y_n}{\partial x_n} \end{pmatrix} \tag{2.7}$$

Для вычисления Якобиана функции $(y_1, y_2,...,y_n) = f(x_1, x_2,...,x_n)$ необходимо найти частные производные всех y_i системы по всем переменным x_i. Для этого от инициирующей точки отступается шаг по увеличению и по уменьшению одной из координат на величину заданной точности. Вычисляются и сравниваются значения функции в полученных точках, получая величину ошибки. При делении полученной разности ошибок на удвоенную точность получается частичная производная по переменной x_i.

$$J_i = \begin{pmatrix} \frac{\partial y_1}{\partial x_i} \\ \frac{\partial y_2}{\partial x_i} \\ \vdots \\ \frac{\partial y_n}{\partial x_i} \end{pmatrix} \tag{2.8}$$

После прохождения i по всему отрезку от 1 до n из полученных столбцов J_i (2.8) строится матрица J (2.7).

- Функция **BFGS** реализация алгоритма Бройдена – Флетчера – Гольдфарба – Шанно. На вход функции подаётся точка поиска, количество независимых аргументов системы, дифференцируемая функция, функция оценки спуска и её производная, начальное приближение, дополнительно можно задать точность, величину шага изменения, максимальное количество итераций для поиска.
  Порядок работы:
  1. **Инициализация**: Задаётся начальное приближение матрицы Гессиана H_0 как единичная матрица.
  2. **Получение направления поиска**: Решается система $H_k \cdot p = -g_k$ для получения направления поиска p, где H_k - текущая аппроксимация матрицы Гессиана, а g_k - градиент функции в точке x_k.
  3. **Одномерная оптимизация (поиск по линии)**: Находится приемлемый размер шага α_k в направлении p, полученном на предыдущем шаге.
  4. **Обновление**: Вычисляется новая точка $x_{k+1} = x_k + \alpha_k p$.
  5. **Обновление Гессиана**: Обновляется аппроксимация матрицы Гессиана H_k на основе градиента в новой точке и разности градиентов.
  6. **Проверка условий остановки**: Если выполнены условия остановки (например, $||g_k|| < \varepsilon$ для некоторого малого $\varepsilon > 0$), алгоритм останавливается. В противном случае начинаем с шага 2.

Реализация данных программных методов приведена в приложении 1 и 2.

2.2 **Реализация математических зависимостей манипулятора**

В качестве способа хранения кинематических и инерционных параметров ПМР используется C++ класс RobotArm, реализованный в файлах RobotArm.h и RobotArm.cpp (Приложение 9 и 10).

Внутри класса реализованы динамические массивы звеньев робота и его сочленений, созданы метода добавления звеньев и степеней подвижности между ними с последующей реинициализацией параметров рассматриваемой модели ПМР, а также функции решения ПЗК и ОЗК.

Для хранения характеристик звеньев и сочленений создаются классы RobotLink (Приложение 7 и 8 – RobotLink.h, RobotLink.cpp) RobotJoint (Приложение 5 и 6 – RobotJoint.h, RobotJoint.cpp).

Для согласования взаимодействия классов и добавления общих функций подсчета создаётся библиотека функций RobotAdditions (Приложение 3 и 4 – RobotAdditions.h, RobotAdditions.cpp).

2.2.1 **Определение параметров Денавита–Хартенберга**

Для решения ПЗК манипулятора требуется получить матрицу перехода для TCP относительно основания робота. Данное преобразование легко выполнить, используя параметры Денавита — Хартенберга.

Вычисление параметров Денавита — Хартенберга каждого звена ПМР происходит по следующему алгоритму:

1) определение точек отсчета параметров из конфигурации звеньев ПМР;

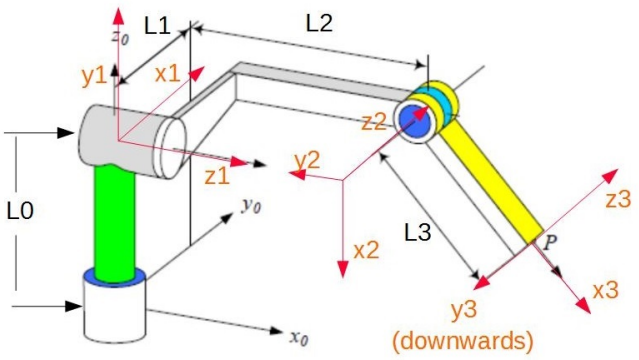

Рис. 2.1 Пример определения точек отсчёта параметров Денавита — Хартенберга

2) расчет параметров, используя соотношение:
   - a_i - расстояние вдоль оси x_i от z_i до z_i
   - α_i - угол вокруг оси x_i от z_i до z_i
   - d_i - расстояние вдоль оси z_i от x_i до x_i
   - θ_i - угол вокруг оси z_i от x_i до x_i

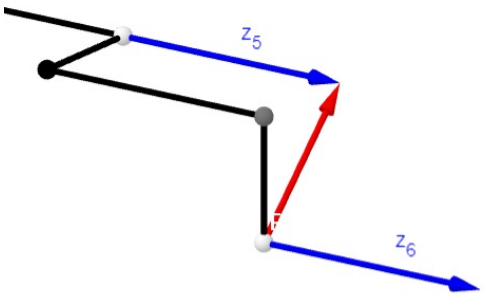

Рис. 2.2 Определение точки отсчёта и направления осей в параллельном случае

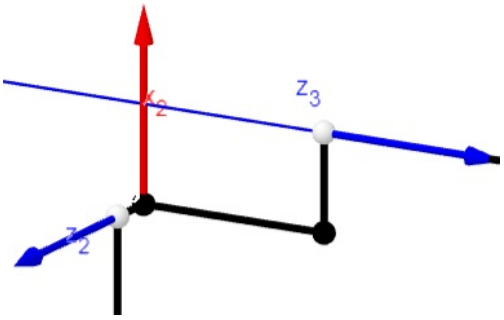

Рис. 2.3 Определение точки отсчёта и направления осей в непараллельном случае

Так как в рамках решаемой задачи рассматриваются ПМР, направления степеней подвижности которых параллельны осям мировой системы координат, то для определения точек отсчёта все конфигурации звеньев можно разделить на две группы: с параллельными входной и выходной осями звена и с не параллельными. Конфигурацию осей x и z звена относительно предыдущего можно увидеть на рисунках 2.2 и 2.3. Из данных о координатах полученной точки и направления осей x и z вычисляются параметры Денавита — Хартенберга.

2.2.2 **Решение обратной задачи кинематики**

Для решения ОЗК внутри класса RobotArm реализован метод **solveIK-POS**, использующий алгоритм BFGS из написанной библиотеки математики. Для работы алгоритма вовнутрь функции **MathAdditions::BFGS** передаются следующие параметры:

1) желаемая точка положения TCP робота в пространстве в виде матрицы 4 на 4, где R – матрица поворота:

$$POINT = \begin{pmatrix} & & & x_{point} \\ & R & & y_{point} \\ & & & z_{point} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.9}$$

2) количество степеней свободы;
3) ссылку на метод решения ПЗК из класса RobotArm;
4) ссылки на функции оценки спуска из библиотеки RobotAdditions;
5) исходное положение ПМР как начальное приближение для поиска.

В результате получаем вектор размера количества степеней свободы, элементы которого являются обобщёнными координатами каждой степени подвижности ПМР.

2.3 **Графическое ядро приложения**

Для создания пользовательского интерфейса разрабатываемого программного обеспечения моделирования работы ПМР создается графическая оболочка приложения.

Её структура такова:

- **уровень отрисовки трёхмерных моделей** – основная часть пользовательского интерфейса;
- **уровень оконного оформления приложения** – модуль, отвечающий за вызов средства отображения трёхмерной графики в среде ОС;
- **уровень отрисовки инфографики отслеживаемых показателей ПМР.**

Для каждого из этих уровней выбирается сторонняя библиотека для оптимизации процесса разработки.

2.3.1 **Выбор средств отображения трёхмерных моделей**

В качестве библиотеки для работы с трёхмерной графикой была выбрана библиотека BGFX.

BGFX – кроссплатформенная, не зависящая от графического API, библиотека рендеринга с открытым исходным кодом.

2.3.2 **Реализация пользовательского оконного интерфейса**

В качестве библиотек для взаимодействия с оконным интерфейсом ОС были выбраны библиотеки SDL3 и ImGUI.

SDL – Simple DirectMedia Layer – это кроссплатформенная библиотека разработки, предназначенная для обеспечения низкоуровневого доступа к аудио, клавиатуре, мыши, джойстику и графическому оборудованию компьютера.

Immediate Mode Graphical User Interface (ImGui) – библиотека графического интерфейса, используемая для быстрой отрисовки пользовательского интерфейса.

2.3.3 **Вывод результатов моделирования**

Для работы с графиками, диаграммами и другими способами визуализации данных используется библиотека Matplot++.

Matplot++ – это графическая библиотека для визуализации данных, которая обеспечивает интерактивное построение графиков, средства для экспорта графиков в высококачественные форматы для научных публикаций, компактный синтаксис, совместимый с аналогичными библиотеками, десятки категорий графиков со специализированными алгоритмами, несколько стилей кодирования и хранения инженерных данных.

ЗАКЛЮЧЕНИЕ

По итогам научной исследовательской работы получены следующие выводы:

1. Проведен анализ предметной области и определена область использования нейронных сетей в алгоритме управления промышленным манипуляционным роботом.
2. Описана структура управляющего алгоритма.
3. Рассмотрены существующие программные пакеты для компьютерного моделирования работы робота-манипулятора.
4. Определена структура разрабатываемого программного обеспечения.
5. Реализованы базовые математические функции и методы представления структуры промышленного манипуляционного робота в памяти компьютера на языке C++.
6. Выбраны графические библиотеки для создания пользовательского интерфейса в дальнейшей разработке приложения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Фу К.*, *Гонсалес Р.*, *Ли К.* Робототехника. — М.: Мир, 1989.
2. Большая российская энциклопедия под ред. Осипова Ю. С. — М.: Большая российская энциклопедия, 2004-2017. — URL: https://bigenc.ru/.
3. *Сущеня Р. В.*, *Кокаев А. Э.* НЕЙРОННЫЕ СЕТИ И ИХ КЛАССИФИКАЦИЯ. ОСНОВНЫЕ ВИДЫ НЕЙРОННЫХ СЕТЕЙ // Вестник науки. — 2023. — № 8. — URL: https://cyberleninka.ru/article/n/neyronnye-seti-i-ih-klassifikatsiya-osnovnye-vidy-neyronnyh-setey.
4. *Ahmed A. A.* Modeling and Control of a Single link Robot Arm Using PID Controller and a Neural Network-based Controller // INTERNATIONAL JOURNAL FOR INNOVATIVE RESEARCH IN MULTIDISCIPLINARY FIELD. — 2020.
5. *Neto P.*, *Pires J. N.*, *Moreira A. P.* Accelerometer-based control of an industrial robotic arm. — 2009. — Сент. — DOI: 10.1109/roman.2009.5326285.
6. A Brief Review of Neural Networks Based Learning and Control and Their Applications for Robots / Y. Jiang [и др.]. — 2017. — Янв. — DOI: 10.1155/2017/1895897.
7. *Кожевников М. М.*, *Пашкевич А. П.*, *Чумаков О. А.* Планирование траекторий промышленных роботов-манипуляторов на основе нейронных сетей // Доклады Белорусского государственного университета информатики и радиоэлектроники. — 2010. — URL: https://cyberleninka.ru/article/n/planirovanie-traektoriy-promyshlennyh-robotov-manipulyatorov-na-osnove-neyronnyh-setey.
8. *Yen V. T.*, *Wang N.*, *Cuong P. V.* Robust Adaptive Sliding Mode Neural Networks Control for Industrial Robot Manipulators. — 2019. — Февр. — DOI: 10.1007/s12555-018-0210-y.
9. *Ганин П.*, *А.И. К.* АЛГОРИТМЫ КИНЕМАТИЧЕСКОГО УПРАВЛЕНИЯ МНОГОЗВЕННЫМИ МАНИПУЛЯЦИОННЫМИ РОБОТАМИ НА ОСНОВЕ НЕЧЁТКОЙ НЕЙРОННОЙ СЕТИ // Вестник БГТУ имени В. Г. Шухова. — 2021. — URL: https://cyberleninka.ru/article/n/algoritmy-kinematicheskogo-upravleniya-mnogozvennymi-manipulyatsionnymi-robotami-na-osnove-nechyotkoy-neyronnoy-seti.
10. Siemens Tecnomatix. — URL: https://plm.sw.siemens.com/en-US/tecnomatix/robotics-programming-simulation/.
11. Visual components. — URL: https://www.visualcomponents.com/.
12. RoboDK. — URL: https://robodk.com/index.
13. SprutCAM Robot. — URL: https://sprut.ru/sprutcam-robot/.
14. Eigen library. — URL: https://eigen.tuxfamily.org/.

ПРИЛОЖЕНИЯ

Приложение 1

MathAdditions.h

```
#pragma once

#include <Eigen/Dense>

#define PI 3.141592653589793

namespace MathAdditions
{
    template <typename X>
    using forwardFunc = X (*)(Eigen::VectorXd x);

    template <typename X>
    using errorFunc = Eigen::VectorXd (*)(X a, X b);

    double DegToRad(double deg);

    double RadToDeg(double rad);

    double projVector(Eigen::Vector3d a, Eigen::Vector3d b);

    double getAngleAroundAxis(Eigen::Vector3d a, Eigen::Vector3d b, Eigen::
    Vector3d Axis);

    double getAngle(Eigen::Vector3d a, Eigen::Vector3d b);

    Eigen::Matrix3d Rx(double angle);

    Eigen::Matrix3d Ry(double angle);

    Eigen::Matrix3d Rz(double angle);

    template <typename T>
    Eigen::MatrixXd calcJacobian(forwardFunc<T> forwFunc, errorFunc<T>
    errFunc, Eigen::VectorXd x_init, int num_DOF, double eps = 1e-6);

    template <typename T>
    Eigen::VectorXd BFGS(T target, int num_DOF,
                         forwardFunc<T> forwFunc,
                         double (*f)(forwardFunc<T>, Eigen::VectorXd q, T
    target),
                         Eigen::VectorXd (*df)(forwardFunc<T>, Eigen::
    VectorXd q, T target),
                         Eigen::VectorXd x_init,
                         double eps = 1e-6, double alpha = 0.01, int
```

```
    max_iterations = 100);
}
```

Приложение 2

MathAdditions.cpp

```cpp
#include <MathAdditions.h>
#include <iostream>

/**
 * Converts degrees to radians.
 *
 * @param deg the value in degrees to be converted
 *
 * @return the value in radians
 */
double MathAdditions::DegToRad(double deg)
{
    return deg * PI / 180.0;
}

/**
 * Converts radians to degrees.
 *
 * @param rad the value in radians to be converted
 *
 * @return the value in degrees
 */
double MathAdditions::RadToDeg(double rad)
{
    return rad * 180.0 / PI;
}

/**
 * Calculates the projection of vector `a` onto vector `b`.
 *
 * @param a The first vector.
 * @param b The second vector.
 *
 * @return The projection of vector `a` onto vector `b`.
 *
 * @throws std::invalid_argument if the second vector is null.
 */
double MathAdditions::projVector(Eigen::Vector3d a, Eigen::Vector3d b)
{
    if (b.norm() == 0)
    {
        throw std::invalid_argument("Vector b must not be null");
    }
    return a.dot(b) / b.norm();
```

```
}

/**
 * Calculates the angle between two vectors around a given axis.
 *
 * @param a The first vector.
 * @param b The second vector.
 * @param Axis The axis around which the angle is calculated.
 *
 * @return The angle between the vectors in radians with direction of
   rotation.
 *
 * @throws std::invalid_argument if either of the vectors is null or one of
   the vectors is parallel to the axis.
 */
double MathAdditions::getAngleAroundAxis(Eigen::Vector3d a, Eigen::Vector3d b
   , Eigen::Vector3d Axis)
{
    if (Axis.norm() == 0)
        throw std::invalid_argument("Axis must not be null");

    Axis.normalize();
    a = a - MathAdditions::projVector(a, Axis) * Axis;
    b = b - MathAdditions::projVector(b, Axis) * Axis;
    Eigen::Vector3d c = a.cross(b);
    if (a.norm() == 0 || b.norm() == 0)
        throw std::invalid_argument("Vectors must not be parallel to axis");
    else if (a.norm()*b.norm() == a.dot(b))
        return 0;
    else if (a.norm()*b.norm() == -a.dot(b))
        return PI;
    else if (MathAdditions::projVector(a, b) == 0)
        if (MathAdditions::projVector(c, Axis) > 0)
            return PI/2;
        else
            return -PI/2;
    else
        if (MathAdditions::projVector(c, Axis) > 0)
            return acos(a.dot(b) / (a.norm() * b.norm()));
        else
            return -acos(a.dot(b) / (a.norm() * b.norm()));
}

/**
 * Calculates the angle between two vectors around a given axis.
 *
 * @param a The first vector.
 * @param b The second vector.
```

```
 *
 * @return The angle between the vectors in radians.
 *
 * @throws std::invalid_argument if either of the vectors is null.
 */
double MathAdditions::getAngle(Eigen::Vector3d a, Eigen::Vector3d b)
{
    if (a.norm() == 0 || b.norm() == 0)
        throw std::invalid_argument("Vectors must not be null");
    else if (a.norm()*b.norm() == a.dot(b))
        return 0;
    else if (a.norm()*b.norm() == -a.dot(b))
        return PI;
    else if (MathAdditions::projVector(a, b) == 0)
        return PI/2;
    else
        return acos(a.dot(b) / (a.norm() * b.norm()));
}

/**
 * Generates a 3x3 rotation matrix around the x-axis
 *
 * @param angle The angle of rotation in radians
 *
 * @return The rotation matrix
 */
Eigen::Matrix3d MathAdditions::Rx(double angle)
{
    Eigen::Matrix3d Rx;
    Rx << 1,          0,           0,
          0, cos(angle), -sin(angle),
          0, sin(angle),  cos(angle);
    return Rx;
}

/**
 * Generates a 3x3 rotation matrix around the y-axis
 *
 * @param angle The angle of rotation in radians
 *
 * @return The rotation matrix.
 */
Eigen::Matrix3d MathAdditions::Ry(double angle)
{
    Eigen::Matrix3d Ry;
    Ry << cos(angle), 0, sin(angle),
                  0, 1,          0,
         -sin(angle), 0, cos(angle);
```

```
    return Ry;
}

/**
 * Generates a 3x3 rotation matrix around the z-axis
 *
 * @param angle The angle of rotation in radians
 *
 * @return The rotation matrix
 */
Eigen::Matrix3d MathAdditions::Rz(double angle)
{
    Eigen::Matrix3d Rz;
    Rz << cos(angle), -sin(angle), 0,
          sin(angle),  cos(angle), 0,
                   0,           0, 1;
    return Rz;
}

/**
 * Calculates the Jacobian matrix for a given forward function and error function.
 *
 * @tparam T the type of the output of the forward function and the input of the error function
 *
 * @param forwFunc a pointer to the forward function
 * @param errFunc a pointer to the error function
 * @param x_init the initial vector
 * @param num_DOF the number of degrees of freedom
 * @param eps the epsilon value (default: 1e-6)
 *
 * @return the Jacobian matrix
 *
 */
template <typename T>
Eigen::MatrixXd MathAdditions::calcJacobian(forwardFunc<T> forwFunc, // Forward function
                                            errorFunc<T> errFunc, // Error function
                                            Eigen::VectorXd x_init, int num_DOF, double eps)
// Eigen::MatrixXd calcJacobian(forwardFunc<T> forwFunc, Eigen::VectorXd (*errFunc)(T a, T b), Eigen::VectorXd x_init, int num_DOF, double eps = 1e-6)
{
  int vectorSize = x_init.size();

```

```
    Eigen::MatrixXd J(num_DOF, vectorSize);

    for (int i = 0; i < vectorSize; i++)
    {
        // Calculate forward kinematics for q + delta_q
            Eigen::VectorXd x_plus = x_init;
            x_plus(i) += eps;
            T T_plus = forwFunc(x_plus);

        // Calculate forward kinematics for q - delta_q
        Eigen::VectorXd x_minus = x_init;
            x_minus(i) -= eps;
            T T_minus = forwFunc(x_minus);

        // Calculate partial derivative
        Eigen::VectorXd derivative = errFunc(T_plus, T_minus) / (2 * eps);

            // Add to Jacobian matrix
        // J.block<num_DOF,1>(0,i) = derivative;
            J.col(i) = derivative;
    }

    return J;
}

/**
 * This function implements the Broyden-Fletcher-Goldfarb-Shanno (BFGS)
    optimization algorithm
 * to find the minimum of a given cost function. It iteratively updates the
    Hessian approximation
 * to approximate the inverse of the true Hessian matrix, and calculates the
    search direction
 * using the updated Hessian approximation and the gradient of the cost
    function.
 *
 * @tparam T The type of the target value and forward function.
 * @param target The target value.
 * @param num_DOF The number of degrees of freedom.
 * @param forwFunc The forward function.
 * @param f The cost function.
 * @param df The gradient of the cost function.
 * @param x_init The initial guess. Default is a zero vector of size num_DOF.
 * @param eps The tolerance. Default is 1e-6.
 * @param alpha The step size. Default is 0.01.
 * @param max_iterations The maximum number of iterations. Default is 100.
 * @return The optimized vector x that minimizes the cost function.
 *
 * @throws std::invalid_argument if the initial guess is not of size num_DOF.
```

```
223  */
template <typename T>
Eigen::VectorXd MathAdditions::BFGS(T target, int num_DOF, // Target value,
    number of degrees of freedom
226                                     forwardFunc<T> forwFunc, // Forward
    function
                                        double (*f)(forwardFunc<T>, Eigen::
    VectorXd q, T target), // Cost function
                                        Eigen::VectorXd (*df)(forwardFunc<T>,
    Eigen::VectorXd q, T target), // Gradient of cost function
229                                     Eigen::VectorXd x_init, // Initial guess
                                        double eps, double alpha, int
    max_iterations) // Tolerance, step size, max iterations
{
232
    if (x_init.size() != num_DOF) {
        throw std::invalid_argument("x_init must be of size num_DOF");
235     }

    // Initial guess
238     Eigen::VectorXd x = x_init;

    // Initial Hessian approximation
241     Eigen::MatrixXd H = Eigen::MatrixXd::Identity(x.size(), x.size());


244     // BFGS iterations
    for (int i = 0; i < max_iterations; ++i) {
        // Calculate search direction
247         Eigen::VectorXd p = -H * df(forwFunc, x, target);

        // Update x for comparison
250         Eigen::VectorXd x_new = x + alpha * p;

        // Check for convergence
253         if ((x_new - x).norm() < eps) {
            break;
        }
256
        // Update Hessian approximation
        Eigen::VectorXd s = x_new - x;
259         Eigen::VectorXd y = df(forwFunc, x_new, target) - df(forwFunc, x,
    target);
        double rho = 1 / y.dot(s);
        H = (Eigen::MatrixXd::Identity(x.size(), x.size()) - rho * s * y.
    transpose()) * H
262             * (Eigen::MatrixXd::Identity(x.size(), x.size()) - rho * y * s.
    transpose())
```

```
            + rho * s * s.transpose();

        // Update x
        x = x_new;
    }

    return x;
}
```

RobotAdditions.h

```
#pragma once

#include <Eigen/Dense>
#include <MathAdditions.h>

#define PI 3.141592653589793

using DirectPoint = Eigen::Matrix4d; //definition of point as matrix 4x4

template <typename X>
using forwardFunc = X (*)(Eigen::VectorXd x);

enum Axes //Enumeration for Axis direction (Z - UP)
{
  NoneAxis,
  X_Axis,
  Y_Axis,
  Z_Axis
};

enum MoveType //Enumeration for joint movement
{
  NoneJoint,
  RotJoint,
  DispJoint
};

struct Offset //Structure for point without orientation
{
  double x, y, z;
};

struct DHParams //D-H params structure for robot arm calculation
{
  double alpha;
  double a;
  double theta;
  double d;
};

namespace RobotAdditions
{
  Eigen::Matrix4d CalcTransposeMatrix(DHParams params, double jointValue,
    MoveType jointType);
```

```
45    DHParams CalcDHParams(DirectPoint point_o, DirectPoint point_e);

      Eigen::VectorXd errorRobotPoses(DirectPoint needPoint, DirectPoint
       currentPoint);
48
      Eigen::MatrixXd calcRobotJacobian(DirectPoint (*forwFunc)(Eigen::VectorXd q
       ), Eigen::VectorXd q_init, double eps = 1e-6);

51    double costRobotFunction(DirectPoint (*forwFunc)(Eigen::VectorXd q), Eigen
       ::VectorXd q, DirectPoint target);

      Eigen::VectorXd gradientCostRobotFunction(DirectPoint (*forwFunc)(Eigen::
       VectorXd q), Eigen::VectorXd q_init, DirectPoint target);
54 }
```

Приложение 4

RobotAdditions.cpp

```
#include <iostream>
#include "RobotAdditions.h"

/**
 * Calculates the transpose matrix of a given set of DH parameters, joint value, and joint type
 *
 * @param params the DH parameters (Denavit-Hartenberg parameters)
 * @param jointValue the value of the joint (either the rotation angle or the displacement)
 * @param jointType the type of the joint (rotation or displacement)
 *
 * @return the transpose matrix of the DH parameters
 */
Eigen::Matrix4d RobotAdditions::CalcTransposeMatrix(DHParams params, double jointValue, MoveType jointType)
{
    Eigen::Matrix4d T;

    double theta = params.theta;
    double d = params.d;
    double a = params.a;
    double alpha = params.alpha;

    if (jointValue == RotJoint)
    {
        theta = params.theta + jointValue;
    }
    else
    {
        d = params.d + jointValue;
    }
    T << cos(theta), -sin(theta) * cos(alpha), sin(theta) * sin(alpha), a * cos(theta),
        sin(theta), cos(theta) * cos(alpha), -cos(theta) * sin(alpha), a * sin(theta),
        0, sin(alpha), cos(alpha), d,
        0, 0, 0, 1;

    return T;
}

/**
 * Calculates the DH parameters between i-1 and i XYZ axes
```

```
 *
 * @param point_o DirectPoint that represents the i-1 XYZ axes.
 * @param point_e DirectPoint that represents the i XYZ axes
 *
 * @return The calculated DHParams object.
 *
 * @throws std::invalid_argument if one of the vectors that represent the
   coordinate axis is null.
 */
DHParams RobotAdditions::CalcDHParams(DirectPoint point_o, DirectPoint
   point_e)
{
    Eigen::Vector3d offset;
    offset << point_e(0, 3) - point_o(0, 3), point_e(1, 3) - point_o(1, 3),
   point_e(2, 3) - point_o(2, 3);

    Eigen::Vector3d axisX1, axisZ1, axisX2, axisZ2;
    axisX1 << point_o(0, 0), point_o(1, 0), point_o(2, 0);
    axisZ1 << point_o(0, 2), point_o(1, 2), point_o(2, 2);
    axisX2 << point_e(0, 0), point_e(1, 0), point_e(2, 0);
    axisZ2 << point_e(0, 2), point_e(1, 2), point_e(2, 2);

    DHParams params;
    try
    {
        params.a = MathAdditions::projVector(offset, axisX2);
    }
    catch (const std::invalid_argument &e)
    {
        throw(e);
    }
    try
    {
        params.alpha = MathAdditions::getAngleAroundAxis(axisZ1, axisZ2,
   axisX2);
    }
    catch (const std::invalid_argument &e)
    {
        throw(e);
    }
    try
    {
        params.d = MathAdditions::projVector(offset, axisZ1);
    }
    catch (const std::invalid_argument &e)
    {
        throw(e);
    }
```

```
84      try
        {
            params.theta = MathAdditions::getAngleAroundAxis(axisX1, axisX2,
       axisZ1);
87      }
        catch (const std::invalid_argument &e)
        {
90          throw(e);
        }
        return params;
93  }

    /**
96   * Calculate the error between the desired point and the current point of a
       robot.
     *
     * @param needPoint The desired point represented as a DirectPoint object.
99   * @param currentPoint The current point represented as a DirectPoint object.
     *
     * @return An Eigen::VectorXd object representing the error between the
       desired point and the current point.
102  * The error vector has 6 elements, where the first 3 elements represent the
       position error
     * and the last 3 elements represent the rotation error.
     */
105 Eigen::VectorXd RobotAdditions::errorRobotPoses(DirectPoint needPoint,
       DirectPoint currentPoint)
    {
        // Extract position and rotation of current point
108     Eigen::Vector3d current_position;
        current_position << currentPoint(0, 3), currentPoint(1, 3), currentPoint
       (2, 3);
        Eigen::Matrix3d current_rotation = currentPoint.block<3, 3>(0, 0);
111
        // Extract position and rotation of needed point
        Eigen::Vector3d need_position;
114     need_position << needPoint(0, 3), needPoint(1, 3), needPoint(2, 3);
        Eigen::Matrix3d need_rotation = needPoint.block<3, 3>(0, 0);

117     // Calculate position error
        Eigen::Vector3d position_error = need_position - current_position;

120     // Calculate rotation error
        Eigen::Matrix3d rotation_error = need_rotation.inverse() *
       current_rotation;
        Eigen::Vector3d angle_error = rotation_error.eulerAngles(0, 1, 2);
123
        // Create error vector
```

```
    Eigen::VectorXd error(6);
126     error << position_error, angle_error;
    return error;
}
129 
/**
 * Calculates the robot Jacobian matrix.
132  *
 * @param forwFunc the forward function object for the direct point
 * @param q_init the initial vector value
135  * @param eps the epsilon value (default: 1e-6)
 *
 * @return the calculated Jacobian matrix
138  */
Eigen::MatrixXd RobotAdditions::calcRobotJacobian(forwardFunc<DirectPoint>
    forwFunc, Eigen::VectorXd q_init, double eps)
{
141     return MathAdditions::calcJacobian<DirectPoint>(forwFunc, RobotAdditions
    ::errorRobotPoses, q_init, 6, eps);
}

144 /**
 * Calculates the cost of a robot function.
 *
147  * @param forwFunc a pointer to a function that calculates the forward
    transformation of the robot
 * @param q the joint configuration of the robot
 * @param target the desired end-effector transformation
150  *
 * @return the cost of the robot function
 */
153 double RobotAdditions::costRobotFunction(DirectPoint (*forwFunc)(Eigen::
    VectorXd q), Eigen::VectorXd q, DirectPoint target)
{
    // Calculate end-effector transformation matrix
156     DirectPoint T = forwFunc(q);

    // Calculate position error
159     Eigen::Vector3d position_error = T.block<3, 1>(0, 3) - target.block<3,
    1>(0, 3);

    // Calculate rotation error
162     Eigen::Matrix3d rotation_error = T.block<3, 3>(0, 0) - target.block<3,
    3>(0, 0);

    // Calculate cost
165     return position_error.squaredNorm() + rotation_error.norm();
}
```

```
168 /**
     * Calculates the gradient of the cost function for a robot.
     *
171  * @param forwFunc a pointer to a function that calculates the forward
       kinematics of the robot
     * @param q_init the initial configuration of the robot
     * @param target the target position and orientation for the end-effector
174  *
     * @return the gradient of the cost function
     */
177 Eigen::VectorXd RobotAdditions::gradientCostRobotFunction(DirectPoint (*
       forwFunc)(Eigen::VectorXd q), Eigen::VectorXd q_init, DirectPoint target)
    {
        // Calculate Jacobian
180     Eigen::MatrixXd J = calcRobotJacobian(forwFunc, q_init);

        // Calculate end-effector transformation matrix
183     DirectPoint T = forwFunc(q_init);

        // Calculate position error
186     Eigen::Vector3d position_error = T.block<3, 1>(0, 3) - target.block<3,
       1>(0, 3);

        // Calculate rotation error
189     Eigen::Matrix3d rotation_error = T.block<3, 3>(0, 0) - target.block<3,
       3>(0, 0);

        // Calculate gradient
192     return 2 * J.transpose() * position_error + 2 * rotation_error.norm() * J
       .transpose();
    }
```

Приложение 5

RobotJoint.h

```
#pragma once

#include <Eigen/Dense>
#include ”RobotAdditions.h”

class RobotJoint //Class of joints and motors of Robot
{
private:

public:

  double curCoord;
  MoveType jointType;

  RobotJoint() = default;
  RobotJoint(MoveType j);
};
```

Приложение 6

RobotJoint.cpp

```
#include ”RobotJoint.h”

RobotJoint::RobotJoint(MoveType j)
{
  this->curCoord = 0;
  this->jointType = j;
}
```

RobotLink.h

```
#pragma once

#include <Eigen/Dense>
#include ”RobotAdditions.h”

class RobotLink //Class of links of Robot include math and visualization
{
private: //Private members

public:
    //Dynamic parameters
  double mass;
  Offset massPoint;
  double inertia;

    //Kinematic parameters
  Axes PreviousJointOrientation;
  Axes ExitJointOrientation;
  Offset ExitPoint;

  //Class constructor
  RobotLink() = default;
  RobotLink(const RobotLink &t);
  RobotLink(Offset endLink, Axes AxisLink, Axes AxisLinkPrev);
};
```

Приложение 8

RobotLink.cpp

```
#include ”RobotLink.h”

RobotLink::RobotLink(Offset endLink, Axes AxisLink, Axes AxisLinkPrev)
{
  this->ExitPoint = endLink;
  this->ExitJointOrientation = AxisLink;
  this->PreviousJointOrientation = AxisLinkPrev;
  this->mass = 0;
  Offset temp; //remake with dynamics adding
  temp.x = 0;
  temp.y = 0;
  temp.z = 0;
  this->massPoint = temp;
  this->inertia = 0;
}

RobotLink::RobotLink(const RobotLink &t)
{
  this->ExitPoint = t.ExitPoint;
  this->ExitJointOrientation = t.ExitJointOrientation;
  this->PreviousJointOrientation = t.PreviousJointOrientation;
  this->mass = t.mass;
  this->massPoint = t.massPoint;
  this->inertia = t.inertia;
}
```

RobotArm.h

```
#pragma once
#include <vector>
#include <Eigen/Dense>
#include <MathAdditions.h>
#include ”RobotLink.h”
#include ”RobotJoint.h”
#include ”RobotAdditions.h”

class RobotArm
{
private:
  bool isInitialized = false;

  RobotLink LinkZero;
  std::vector<RobotLink> links;
  std::vector<RobotJoint> joints;
  std::vector<DirectPoint> DHPoints;
  std::vector<DHParams> LinkJointParams;

  void CalcDHPoints();
  void PointsToParams();
  Offset CalcLinkFullOffset(int index);

public:
  Offset originPosition;

  RobotArm() = default;
  RobotArm(Offset originPoint, Offset endLinkZero, Axes AxisLinkZero);

  void AddLink(Offset endLink, Axes AxisLink, MoveType jointType);

  void initialize();

  Eigen::VectorXd getJointAngles();

  void setJointAngles(Eigen::VectorXd q);

  DirectPoint ForwardKinematics(Eigen::VectorXd q);

  static DirectPoint ForwardKinematics_static(RobotArm* arm, Eigen::VectorXd
    q);

  Eigen::VectorXd solveIK_POS(DirectPoint needPoint, std::string method);
```

```
44    Eigen::VectorXd solveIK_VEL(Eigen::VectorXd needVelocity);
};
```

RobotArm.cpp

```
#include <iostream>
#include "RobotArm.h"

RobotArm::RobotArm(Offset originPoint, Offset endLinkZero, Axes AxisLinkZero)
{
  this->originPosition = originPoint;
  RobotLink NewLink(endLinkZero, AxisLinkZero, NoneAxis);
  this->LinkZero = NewLink;
}

/**
 * Adds a new link to the RobotArm.
 *
 * @param endLink the offset of the end of the new link
 * @param AxisLink the axes of the new link
 * @param jointType the type of joint for the new link
 *
 * @throws std::invalid_argument if the offset is zero
 */
void RobotArm::AddLink(Offset endLink, Axes AxisLink, MoveType jointType)
{
  if (!((endLink.x)||(endLink.y)||(endLink.z)))
  {
    throw std::invalid_argument("Offset must not be zero");
  }

  // Add new joint
  RobotJoint NewJoint(jointType);
  this->joints.push_back(NewJoint);

  // Add new link
  Axes prevAxis;
  if (this->links.size() > 0)
    prevAxis = this->links.back().ExitJointOrientation;
  else
    prevAxis = this->LinkZero.ExitJointOrientation;
  RobotLink NewLink(endLink, AxisLink, prevAxis);
  this->links.push_back(NewLink);
  this->isInitialized = false;
}

/**
 * Calculates the full offset of a robot arm link based on the given index
 *
```

```
 * @param index The index of the link
 *
 * @return The calculated offset of the link
 */
Offset RobotArm::CalcLinkFullOffset(int index)
{
  Offset offset;
  if (index == 0)
  {
    offset.x = this->LinkZero.ExitPoint.x;
    offset.y = this->LinkZero.ExitPoint.y;
    offset.z = this->LinkZero.ExitPoint.z;
    return offset;
  }
  else
  {
    offset.x = this->links[index].ExitPoint.x + this->CalcLinkFullOffset(
   index - 1).x;
    offset.y = this->links[index].ExitPoint.y + this->CalcLinkFullOffset(
   index - 1).y;
    offset.z = this->links[index].ExitPoint.z + this->CalcLinkFullOffset(
   index - 1).z;
    return offset;
  }
}

/**
 * Calculate the DH points for the robot arm using links info
 */
void RobotArm::CalcDHPoints()
{
  DirectPoint NewPoint;
  switch (this->LinkZero.ExitJointOrientation)
  {
  case X_Axis:
    NewPoint << 0, 0, 1, 0,
          1, 0, 0, 0,
          0, 1, 0, 0,
          0, 0, 0, 1;
    break;
  case Y_Axis:
    NewPoint << 0, 1, 0, 0,
          0, 0, 1, 0,
          1, 0, 0, 0,
          0, 0, 0, 1;
    break;
  case Z_Axis:
    NewPoint << 1, 0, 0, 0,
```

```
90          0, 1, 0, 0,
            0, 0, 1, 0,
            0, 0, 0, 1;
93    break;
  default:
    break;
96  }
  this->DHPoints.push_back(NewPoint);

99  int count = this->links.size();
  for (int i = 0; i < count; i++)
  {
102   switch (this->links[i].ExitJointOrientation)
    {
    case X_Axis:
105   NewPoint << 0, 0, 1, 0,
              1, 0, 0, 0,
              0, 1, 0, 0,
108           0, 0, 0, 1;
      break;
    case Y_Axis:
111   NewPoint << 0, 1, 0, 0,
              0, 0, 1, 0,
              1, 0, 0, 0,
114           0, 0, 0, 1;
      break;
    case Z_Axis:
117   NewPoint << 1, 0, 0, 0,
              0, 1, 0, 0,
              0, 0, 1, 0,
120           0, 0, 0, 1;
      break;
    default:
123   break;
    }
    this->DHPoints.push_back(NewPoint);
126 }

  for(int i = count; i > 0; i--)
129 {
    RobotLink link = this->links[i];
    RobotLink prevLink = this->links[i - 1];
132 double x = link.ExitPoint.x;
    double y = link.ExitPoint.y;
    double z = link.ExitPoint.z;
135 Eigen::Vector2d v;
    DirectPoint NewPoint;
    if (link.ExitJointOrientation == prevLink.ExitJointOrientation) // case
```

```
of parallel axis
 {
   switch (link.ExitJointOrientation)
   {
   case X_Axis:
     v << y, z;
     v.normalize();
     NewPoint <<    0,     0, 1, this->CalcLinkFullOffset(i).x,
             v[0], -v[1], 0, this->CalcLinkFullOffset(i).y,
             v[1],  v[0], 0, this->CalcLinkFullOffset(i).z,
                0,     0, 0, 1;
     this->DHPoints[i] = NewPoint;
     break;
   case Y_Axis:
     v << x, z;
     v.normalize();
     NewPoint << v[0], -v[1], 0, this->CalcLinkFullOffset(i).x,
                 0,     0, 1, this->CalcLinkFullOffset(i).y,
             v[1],  v[0], 0, this->CalcLinkFullOffset(i).z,
                 0,     0, 0, 1;
     this->DHPoints[i] = NewPoint;
     break;
   case Z_Axis:
     v << x, y;
     v.normalize();
     NewPoint << v[0], -v[1], 0, this->CalcLinkFullOffset(i).x,
             v[1],  v[0], 0, this->CalcLinkFullOffset(i).y,
                 0,     0, 1, this->CalcLinkFullOffset(i).z,
                 0,     0, 0, 1;
     this->DHPoints[i] = NewPoint;
     break;
   default:
     break;
   }
 }
 // cases of nonparallel axis
 else if ((link.ExitJointOrientation == X_Axis) && (prevLink.
ExitJointOrientation == Y_Axis))
 {
   DirectPoint NewPoint;
   NewPoint <<  0, 0, 1, this->CalcLinkFullOffset(i-1).x,
            0, 1, 0, this->CalcLinkFullOffset(i).y,
           -1, 0, 0, this->CalcLinkFullOffset(i).z,
            0, 0, 0, 1;
   this->DHPoints[i] = NewPoint;
 }
 else if ((link.ExitJointOrientation == Y_Axis) && (prevLink.
ExitJointOrientation == X_Axis))
```

```
183     {
          DirectPoint NewPoint;
          NewPoint << 0, 1, 0, this->CalcLinkFullOffset(i).x,
186                   0, 0, 1, this->CalcLinkFullOffset(i-1).y,
                      1, 0, 0, this->CalcLinkFullOffset(i).z,
                      0, 0, 0, 1;
189       this->DHPoints[i] = NewPoint;
        }
        else if ((link.ExitJointOrientation == Y_Axis) && (prevLink.
      ExitJointOrientation == Z_Axis))
192     {
          DirectPoint NewPoint;
          NewPoint << -1, 0, 0, this->CalcLinkFullOffset(i).x,
195                  0, 0, 1, this->CalcLinkFullOffset(i-1).y,
                     0, 1, 1, this->CalcLinkFullOffset(i).z,
                     0, 0, 0, 1;
198       this->DHPoints[i] = NewPoint;
        }
        else if ((link.ExitJointOrientation == Z_Axis) && (prevLink.
      ExitJointOrientation == Y_Axis))
201     {
          DirectPoint NewPoint;
          NewPoint << 1, 0, 0, this->CalcLinkFullOffset(i).x,
204                 0, 1, 0, this->CalcLinkFullOffset(i).y,
                    0, 0, 1, this->CalcLinkFullOffset(i-1).z,
                    0, 0, 0, 1;
207       this->DHPoints[i] = NewPoint;
        }
        else if ((link.ExitJointOrientation == Z_Axis) && (prevLink.
      ExitJointOrientation == X_Axis))
210     {
          DirectPoint NewPoint;
          NewPoint <<  0, 1, 0, this->CalcLinkFullOffset(i).x,
213                 -1, 0, 0, this->CalcLinkFullOffset(i).y,
                     0, 0, 1, this->CalcLinkFullOffset(i-1).z,
                     0, 0, 0, 1;
216       this->DHPoints[i] = NewPoint;

        }
219     else if ((link.ExitJointOrientation == X_Axis) && (prevLink.
      ExitJointOrientation == Z_Axis))
        {
          DirectPoint NewPoint;
222       NewPoint << 0, 0, 1, this->CalcLinkFullOffset(i-1).x,
                      1, 0, 0, this->CalcLinkFullOffset(i).y,
                      0, 1, 0, this->CalcLinkFullOffset(i).z,
225                   0, 0, 0, 1;
          this->DHPoints[i] = NewPoint;
```

```
    }
  }
}

/**
 * Converts a set of DH points to DH parameters.
 *
 * @throws std::invalid_argument if any axis of the DH points are invalid.
 */
void RobotArm::PointsToParams()
{
  int count = this->DHPoints.size();
  for (int i = 1; i < count; i++)
    try {
      DHParams params = RobotAdditions::CalcDHParams(this->DHPoints[i-1],
    this->DHPoints[i]);
      this->LinkJointParams.push_back(params);
    }
    catch (const std::invalid_argument& e) {
      throw(e);
    }
}

/**
 * Calculates the forward kinematics of the robot arm
 *
 * @return The transformation matrix representing the end-effector position
 */
DirectPoint RobotArm::ForwardKinematics(Eigen::VectorXd q)
{
  if (!this->isInitialized)
  {
    Eigen::Matrix4d T = Eigen::Matrix4d::Identity();
    int count = this->LinkJointParams.size();
    for (int i = 0; i < count; i++)
    {
      DHParams params = this->LinkJointParams[i];
      T = T * RobotAdditions::CalcTransposeMatrix(params, q[i], this->joints[
    i].jointType);
      // T = T * CalcTransposeMatrix(params, this->joints[i].curCoord, this->
    joints[i].jointType);
    }
    return T;
  }
  else
  {
    std::cout << "RobotArm is not initialized" << std::endl;
    std::cout << "Do you want to initialize it now? (y/n)" << std::endl;
```

```
    char answer;
    std::cin >> answer;
    if (answer == 'y')
    {
      this->initialize();
      return this->ForwardKinematics(q);
    }
    else
      return DirectPoint::Identity();
  }
}

/**
 * Initializes the RobotArm.
 *
 * @throws std::exception if one of the DH points is invalid
 */
void RobotArm::initialize()
{
  if (!this->isInitialized)
    try {
      this->CalcDHPoints();
      this->PointsToParams();
      this->isInitialized = true;
    }
    catch(const std::exception& e)
    {
      std::cerr << "One of the DH points is invalid: "<< e.what() << '\n';
      throw(e);
    }
  else
    std::cout << "RobotArm is already initialized" << std::endl;
}

/**
 * Retrieves the joint angles of the robot arm.
 *
 * @return An Eigen::VectorXd object representing the joint angles
 */
Eigen::VectorXd RobotArm::getJointAngles()
{
  Eigen::VectorXd q(this->joints.size());
  q = Eigen::VectorXd::Zero(6);
  for (int i = 0; i < this->joints.size(); i++)
    q[i] = this->joints[i].curCoord;
  return q;
}
```

```
/**
 * Sets the joint angles of the robot arm.
 *
 * @param q An Eigen::VectorXd representing the joint angles.
 */
void RobotArm::setJointAngles(Eigen::VectorXd q)
{
  for (int i = 0; i < this->joints.size(); i++)
    this->joints[i].curCoord = q[i];
}

Eigen::VectorXd RobotArm::solveIK_POS(DirectPoint needPoint, std::string
    method = "1 - BFGS")
{
  Eigen::VectorXd q = this->getJointAngles();

  std::function<DirectPoint(Eigen::VectorXd)> forwFunct = std::bind(&RobotArm
    ::ForwardKinematics, this, std::placeholders::_1);
    auto forwFunc = forwFunct.target<DirectPoint(*)(Eigen::VectorXd)>();

    // Call the optimization functions
    if (forwFunc)
    {
        switch (method.c_str()[0])
        {
        case '1':
            return MathAdditions::BFGS<DirectPoint>(needPoint, q.size(), *
    forwFunc,
                            RobotAdditions::costRobotFunction,
                            RobotAdditions::gradientCostRobotFunction,
                            q);
            break;
        default:
            return Eigen::VectorXd::Zero(6);
            break;
        }
    }
    else return Eigen::VectorXd::Zero(6);
}

Eigen::VectorXd RobotArm::solveIK_VEL(Eigen::VectorXd needVelocity)
{
  Eigen::VectorXd q = this->getJointAngles();

  std::function<DirectPoint(Eigen::VectorXd)> forwFunct = std::bind(&RobotArm
    ::ForwardKinematics, this, std::placeholders::_1);
    auto forwFunc = forwFunct.target<DirectPoint(*)(Eigen::VectorXd)>();

```

```
    if (forwFunc)
    {
366 Eigen::MatrixXd J = RobotAdditions::calcRobotJacobian(*forwFunc, q);
    return J.inverse() * needVelocity;
  }
369 else return Eigen::VectorXd::Zero(q.size());
}
```