Федеральное государственное автономное образовательное учреждение высшего образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Факультет систем управления и робототехники

Отчет о научно исследовательской работе по теме: «РАЗРАБОТКА АЛГОРИТМОВ ДЛЯ ВЗАИМОДЕЙСТВИЯ С РОБОТОМ-МАНИПУЛЯТОРОМ С КОМПЬЮТЕРА (С ИСПОЛЬЗОВАНИЕМ TCP)»

Выполнил: студент гр. R3341
А. А. Румянцев

Проверил: преподаватель
доцент, старший научный сотрудник, инженер В. С. Громов

Санкт-Петербург
2025 г.

Содержание

Введение

В настоящее время в промышленной и других сферах все чаще используются роботы-манипуляторы, управляемые со специального пульта или автоматически через загрузку программы на робота. Более предпочтительным является вариант управления без человека – это безопаснее и выгоднее. Однако роботы используют достаточно устаревший язык программирования, например MELFA-BASIC. Написание кода для подобных роботов может быть неудобным, а программы получаться громоздкими. Разработка нового языка программирования для роботов потребует больших вложений, что также не выгодно. Управление с пульта, в свою очередь, требует от оператора высокой квалификации – необходимы знания техники безопасности и принципы работы оборудования. Обучение специалиста для управления роботом-манипулятором с пульта является ресурсоемким процессом, требующим значительных временных и финансовых затрат.

Для повышения безопасности и эффективности взаимодействия с роботом, необходимо максимально отдалить человека от робота, при этом реализовать все основные функции для работы с ним так, чтобы их можно было использовать из некой виртуальной централизованной системы по нажатию кнопок. Реализовать данную идею можно в виде программного интерфейса – аналога физического пульта управления роботом в виде программы на компьютере. Такой подход также позволит упростить обучение специалистов для управления роботом-манипулятором. Кроме того, программу можно купить один раз и установить на множество компьютеров, а разработка и покупка нескольких физических пультов управления будет ресурсозатратным процессом. Однако сейчас программных интерфейсов, позволяющих взаимодействовать с роботом с компьютера сравнительно немного, а те, что уже есть, постепенно устаревают. Возникает необходимость написания нового программного интерфейса для взаимодействия с роботом. Как и любая другая программа, структурно она делится на две части – одна отвечает за внешний вид и удобство управления (интерфейс), другая же обеспечивает взаимодействие с роботом на уровне, не видном пользователю. В рамках данной работы разрабатывалась внутренняя логика программы для взаимодействия компьютера с роботом-манипулятором по протоколу TCP. Пользовательская часть интерфейса при этом рассматривалась как вспомогательная.

1. Подготовка к написанию программы

1.1. Ознакомление с объектом работы

Перед выполнением задания был проведен инструктаж по технике безопасности обращения с роботом-манипулятором.

Под наблюдением преподавателя были изучены ручной режим управления роботом со специального пульта и автоматический с помощью простейших программ на языке MELFA-BASIC, загружаемых на робота.

Для написания программ для робота был изучен язык программирования MELFA-BASIC, некоторые его основные команды и описание представлены далее:

- **SERVO ON** – включение двигателей,
- **SERVO OFF** – выключение двигателей,
- **END** – завершение программы, обязательно размещается в конце файла,
- **JOVRD 100** – скорость движения в процентах от максимальной,
- **SPD 100** – скорость движения при интерполяционных командах,
- **MOV P1** – движение в заданную точку P1,
- **WHILE, FOR** – циклы с условиями,
- **OPEN "COM3:"AS #1** – открытие TCP/IP порта 10003 для подключения интерфейса #1,
- **CLOSE #1** – закрытие TCP/IP порта 10003 для подключения интерфейса #1,
- **DEF INTE DCOMM** – объявление переменной DCOMM целочисленного типа.

1.2. Выбор подходящего языка программирования

Существует достаточно много различных языков программирования, подходящих под реализацию задачи взаимодействия с роботом с компьютера. В рамках данной работы был выбран язык программирования Python, так как он достаточно часто используется в сфере робототехники, имеет достаточно простой и легкочитаемый синтаксис, имеет большое количество готовых библиотек и является кроссплатформенным (программу можно запустить на разных операционных системах).

В ходе выполнения работы использовались следующие библиотеки:

- **socket** – для работы с сетевыми соединениями,
- **typing** – средства для статической типизации переменных и функций,
- **re** – модуль для работы с регулярными выражениями,
- **yaml** – для чтения и записи файлов в формате YAML,
- **enum** – позволяет создавать перечисления с именованными значениями.

Библиотека socket понадобилась для установки TCP-соединения между компьютером и роботом и передачи/получения пакетов.

Для общего улучшения и упрощения кода использовалась библиотека typing.

Модуль re понадобился для обработки ответов с робота.

Библиотека yaml позволила реализовать чтение и сохранение введенных настроек IP и порта, чтобы пользователю не пришлось каждый раз вводить эти данные заново при запуске программы.

Для перечисления команд, статусов сетевого взаимодействия с роботом, декартовых и сочлененных координат понадобилась библиотека enum.

2. Реализация программы для взаимодействия компьютера с роботом-манипулятором

2.1. Определение формата сообщения

Для начала необходимо определиться с форматом передаваемого с компьютера на робот-манипулятор сообщения и обратно.

Сообщение должно быть простое, соответствующее шаблону которое понимает робот.

Сначала будет отправляться номер команды, после чего шесть координат и пара чисел для решения обратной задачи кинематики, если хотим передвижение по декартовым координатам. Если движение сочлененное, то достаточно передать шесть углов вращения.

Было решено, что робот будет отправлять компьютеру 12 координат и пару для решения обратной задачи кинематики как одно сообщение (изменение декартовых координат влияет на сочлененные и наоборот).

В ходе выполнения работы экспериментальным путем было выяснено, что робот отправляет свои декартовы и сочлененные координаты в следующем

формате:

$$[\prime(J_1, J_2, J_3, J_4, J_5, J_6)(X, Y, Z, A, B, C)(K_1, K_2)\prime],$$

то есть как список, содержащий одну строку.

Это означает, что роботу нужно отправлять координаты в виде строк шаблона:

- $(J_1, J_2, J_3, J_4, J_5, J_6)$ – если движение сочлененное,
- $(X, Y, Z, A, B, C)(K_1, K_2)$ – если движение в декартовых координатах,

при этом все значения с плавающей точкой, кроме K_i – они целочисленные.

2.2. Определение основных команд

Проще всего отправлять роботу не строковые команды вида 'EXIT', а численные в виде строк. Например, команда '0' – завершение работы робота.

Создадим для удобства перечисление enum, где каждой команде с названием будет присвоено собственное число.

```
class RobotCommand(Enum):
    EXIT = 0
    GET_POSITION = 1
    MOVE_LINEAR = 2
    MOVE_JOINTS = 3
```

Листинг 1: Определение перечисления с командами для робота.

Краткое описание команд:

- **'0'** – завершение работы робота. На клиентской стороне отключение связи,
- **'1'** – робот вышлет 12 своих координат и кинематическую пару,
- **'2', '3'** – движение робота по декартовым координатам или сочлененное соответственно. После робот снова высылает свою позицию.

В программе для робота будем проверять пришедшее сообщение на совпадение с одним из этих чисел. Далее будет выполняться соответствующий алгоритм.

Например, при получении '2' робот будет двигаться по декартовым координатам в соответствии с присланной после команды в заданном шаблоне следующей позицией робота, после чего вышлет обратно 12 своих координат и пару чисел для решения обратной задачи кинематики.

2.3. Клиентская программа для общения с роботом

Напишем клиентский скрипт, который будет основой сетевеого взаимодействия компьютера с роботом.

Для начала определим статусы сетевого взаимодействия как перечисление. Они помогут при отладке ошибок.

```
class ConnectionStatus(Enum):
    SUCCESS = auto()
    NOT_CONNECTED = auto()
    CONNECTION_ERROR = auto()
    SEND_ERROR = auto()
    RECEIVE_ERROR = auto()
    NONE = auto()
```

Листинг 2: Определение перечисления статусов соединения компьютера с роботом.

Нумерация статусов автоматически с 1 и по порядку. Смысл статусов полностью соответствует их названиям.

Теперь реализуем основные функции для взаимодействия компьютера с роботом. Полностью скрипт представлен в приложении А на листинге 8.

Краткое описание функций скрипта:

- **__init__(self)** – при создании объекта класса вызывается данная функция. Она присвоит объекту поля: sock – сетевой сокет с параметрами AF_INET (семейство адресов IPv4), SOCK_STREAM (протокол TCP); булевое connected – значение истина или ложь в зависимости от того, подключен ли сокет к указанному IP и порту;
- **connect(self, ip: str, port: int) -> ConnectionStatus** – подключение сокета к указанному IP адресу и порту. Возвращает статус соединения. Также присваивает значение "истина" переменной connected, если подключение удалось;
- **disconnect(self) -> ConnectionStatus** – закрывает соединение, если оно есть. Возвращает статус соединения.
- **send(self, data: str) -> ConnectionStatus** – кодирует переданную строку сообщение data и отправляет по протоколу TCP. Возвращает статус соединения;
- **receive(self, out_data: list, buffer_size: int = 1024) -> ConnectionStatus** –

принимает данные по протоколу TCP. Записывает в декодированном виде полученную информацию в список out_data. Возвращает статус соединения.

2.4. Программа на роботе для общения с клиентом

На языке MELFA-BASIC напишем простую программу для теста связи между компьютером и роботом:

```
JOVRD 100
SPD 100
DEF INTE DCOMM
DCOMM = 1
PHELP = P_CURR
SERVO ON
OPEN "COM3:" as #1

WHILE DCOMM > 0
    INPUT #1, DCOMM
    IF DCOMM = 2 THEN
        INPUT #1, PHELP
        PRINT #1, PHELP
    ENDIF
WEND

CLOSE #1
SERVO OFF
END
```

Листинг 3: Простая программа на роботе для проверки связи с клиентом.

Краткое описание программы: в переменную DCOMM ожидается поступление номера команды от компьютера. Когда команда ’2’ пришла, робот записывает приходящие следующим же сообщением от компьютера декартовы координаты и кинематическую пару, после чего высылает обратно то же самое, что и получил.

Если прислали команду ’0’, то робот выйдет из цикла и завершит работу.

Таким образом проверяется, что робот корректно обрабатывает полученные координаты, компьютер верно получает те же координаты обратно. После чего робот успешно завершает работу.

2.5. Проверка связи между компьютером и роботом

Сообщение роботу нужно отправить в определенном формате. Также ответ от робота нужно обработать. Для этого напишем скрипт, который будем использовать каждый раз перед отправкой и получением координат. Полностью скрипт представлен на листинге 9 в приложении Б.

Краткое описание скрипта:

- **build_position_request(pos: List[float]) -> str** – на вход получает координаты как список чисел с плавающей точкой. Преобразует список в строку в соответствии с шаблоном $(v_1, v_2, v_3, v_4, v_5, v_6)$. Возвращает результат преобразования;
- **parse_position_response(response: list)** – получает ответ от робота в виде списка из строки с 12 координатами и кинематической пары и с помощью регулярного выражения делит строку внутри списка на три отдельных списка из сочлененных и декартовых координат и кинематической пары. Возвращает кортеж из трех соответствующих списков.

Также нам нужен скрипт для записи и хранения IP и порта робота. Создадим yaml файл, в котором запишем:

```
connection:
  ip: "192.168.0.4"
  port: 10003
```

Листинг 4: Файл, где хранятся данные для подключения к роботу-манипулятору.

Напишем скрипт, в котором во время работы программы будут сохранены эти данные. Полностью скрипт представлен в приложении В на листинге 10.

Краткое описание скрипта:

- **load(cls, path: str = "config.yaml")** – загружает IP адрес и порт из файла по указанном пути. Сохраняет в свои поля ip, port. По умолчанию файл находится в том же каталоге, где и скрипт;
- **save(cls, path: str = "config.yaml")** – сохраняет конфигурацию IP и порта в файл по указанному пути;
- **set_config(cls, ip: str, port: int)** – меняет конфигурацию IP и порта во время работы программы. Может пригодиться, если необходимо будет отключиться от одного робота и подключиться к другому, не закрывая программу.

Напишем простую программу для компьютера для проверки соединения с роботом:

```
import socket
from config_manager import Config
from command_manager import CommandMessageManager

if __name__ == "__main__":
    Config.load()

    pos = [100.0, -100.0, 100.0, 100.0, 100.0, 100.0]
    test = CommandMessageManager.build_position_request(pos)
    test += "(7,0)"
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((Config.ip, Config.port))
    s.sendall("2".encode('utf-8'))
    s.sendall(test.encode('utf-8'))
    ans = s.recv(1024).decode('utf-8')
    print(ans)
    s.sendall("0".encode('utf-8'))
    s.close()
```

Листинг 5: Простая программа на клиенте для проверки соединения с роботом.

Краткое описание программы: загружается конфигурация IP и порта из файла. Задается некоторый набор из шести координат, из которого создается строка на отправку. Так как отправляются декартовы координаты, необходимо еще к строке на отправку добавить в строковом виде в таком же формате кинематическую пару. После чего создается сокет с полученными из файла IP адресом и портом. Отправляется закодированная команда '2', за которой следуют закодированные обработанные декартовы координаты. После чего принимается, декодируется и выводится в консоль ответ от робота. В конце высылается закодированная команда '0' для завершения работы робота. На клиенте же закрывается сокет.

В присутствии преподавателя на робота была загружена программа, представленная на листинге 3. На компьютере была запущена программа, представленная на листинге 5.

Было выслано с компьютера:

$$(100.0, -100.0, 100.0, 100.0, 100.0, 100.0)\ (7, 0)$$

Было получено от робота:

$$(+100.0, -100.0, +100.0, +100.0, +100.0, +100.0)\ (7, 0)$$

Робот и компьютер взаимодействуют корректно – на компьютер пришло то же, что было с него выслано. Появились дополнительные плюсы, которые при обработке ответа от робота пропадут, то есть они незначащие. Ответ робота был выведен в консоль в сыром виде.

2.6. Клиентская программа для управления роботом

Доработаем нашу программу скриптами с функциями, позволяющими полноценно взаимодействовать с роботом с компьютера.

Для начала введем перечисления для декартовых и сочлененных координат.

После обработки ответа робота мы получаем, например, список из шести декартовых координат. Чтобы подвинуться по какой-то из координат, необходимо добавить шаг к этой координате. Координаты в списке располагаются по порядку. Значит, при обращении к нулевому элементу получим первую координату X, при обращении к пятому получим шестую координату C. Аналогично с сочлененными координатами.

```
class CartesianAxis(Enum):
    X = 0
    Y = 1
    Z = 2
    A = 3
    B = 4
    C = 5

class JointAxis(Enum):
    J1 = 0
    J2 = 1
    J3 = 2
    J4 = 3
    J5 = 4
    J6 = 5
```

Листинг 6: Определение перечислений, где каждой координате соответствует свой индекс в списке.

Также зададим тип координатам. Есть декартовы линейные, декартовы вращательные и сочлененные координаты.

- Линейные декартовы координаты X, Y, Z – mode: cartesian_linear,
- Вращательные декартовы координаты A, B, C – mode: cartesian_rotation,
- Сочлененные координаты $J_1 ... J_6$ – mode: joint_rotation.

Для определения команды из перечисления по ее типу напишем такой словарь:

```
MODE_TO_COMMAND = {
    "cartesian_linear": RobotCommand.MOVE_LINEAR,
    "cartesian_rotation": RobotCommand.MOVE_LINEAR,
    "joint_rotation": RobotCommand.MOVE_JOINTS,
}
```

Листинг 7: Словарь для приведения типа координаты к команде из перечисления.

Нам необходимо в программе на компьютере где-то хранить координаты робота и кинематическую пару. Создадим виртуального робота на клиенте. Скрипт расположен в приложении Г на листинге 11.

Краткое описание функций скрипта:

- **__init__(self, cartesian: ..., joint: ..., kin_sol: ...)** – при создании объекта класса в качестве его полей можно указать декартовы и сочлененные координаты и кинематическую пару. Иначе они будут заданы нулевыми;
- **update_cartesian(self, new_values: ..., kinematic_sol: ...)** – обновляет декартовы координаты и кинематическую пару виртуального робота;
- **update_joint(self, new_values: ...)** – обновляет сочлененные координаты виртуального робота;
- **calculate_next_move(self, mode: ..., axis_index: int, step: float) -> ...** – проверяет тип координаты и считает соответствующие следующие координаты робота. Функция принимает тип координаты, ее индекс в списке координат и шаг, на который нужно сдвинуть робота по этой координате (миллиметры для линейных координат, градусы для вращательных). Возвращает список соответствующих координат, где одно из значений смещено на некоторый шаг.

Так как обратная задача кинематики на клиенте не решается, то кинематическая пара нужна только для того, чтобы выслать ее обратно вместе с

декартовыми координатами роботу.

В интерфейсе подразумевается наличие кнопок каждой координаты в двух типах: со знаком минус и со знаком плюс. Для упрощения логики программы введем кортеж, который будем называть move_info. В нем будут содержаться тип координаты, ее порядковый номер в списке и направление, задаваемое знаком плюс или минус. Примеры кортежей приведены далее.

- **("cartesian_linear" , CartesianAxis.X.value, '+')** – декартова линейная координата X, порядковый номер определяется перечислением декартовых координат (в этом случае будет 0) и положительное направление;
- **("cartesian_rotation" , CartesianAxis.A.value, '-')** – декартова вращательная координата A, порядковый номер в списке определяется аналогично (в данном случае 4) и отрицательное направление;
- **("joint_rotation" , JointAxis.J1.value, '+')** – сочлененная вращательная координата J_1, порядковый номер 0 и положительное направление.

Теперь для всего, что мы уже написали на клиенте, нужна простая надслойка – скрипт, функции которого будут объединять несколько действий, необходимых для полноценного взаимодействия с роботом. Эту надслойку можно будет использовать, например, в обработчиках нажатия кнопок в графическом интерфейсе. Скрипт представлен в приложении Д на листинге 12.

Краткое описание функций данного скрипта:

- **__init__(self)** – создает объект класса и присваивает поля: robot – виртуальный робот, client – клиентская обертка над сокетом и last_..._status – статус сетевого взаимодействия для каждой последней операции компьютера с роботом (присоединение, отправка, получение и отключение; может пригодиться для удобной отладки сетевых проблем). Далее во всех функциях присваивание сетевых статусов аналогичное;
- **connect(self, ip: str, port: int)** – обертка подключения над клиентской оберткой подключения над сокетом. Возвращает булевую переменную connected объекта класса Client. Дополнительная обертка позволяет при необходимости добавить некоторую логику до прямого соединения с роботом;
- **disconnect(self)** – обертка отключения над клиентской оберткой отключения над сокетом. Возвращает отрицание булевой переменной connected объекта класса Client. В функцию до прямого отключения от робота допол-

нительно добавлена логика оповещения робота, что клиент отключается – высылается команда ’0’ для завершения работы робота;

- **is_connected(self)** – возвращает булевую переменную connected объекта класса Client. Непрямая проверка соединения. Есть возможность добавить дополнительную логику;
- **get_pos(self)** – высылает команду ’1’ роботу, чтобы получить его текущие координаты и кинематическую пару. Возвращает ответ робота без обработки. Подразумевается, что обработка будет сделана извне этой функции, ее методы могут быть любыми в соответствии с нуждами;
- **move_axis(self, move_info: ..., step: float)** – принимает кортеж из трех элементов, описанных ранее, и шаг. По направлению определяет знак шага. Вызывает метод calculate_next_move виртуального робота, куда передает тип координаты, ее порядковый номер и шаг (подразумевается, что в виртуальном роботе уже сохранены координаты настоящего). Далее отправляет на робот номер команды в соответствии с конвертацией из словаря MODE_TO_COMMAND. С помощью build_position_request создает сообщение роботу с новыми координатами, полученными от виртуального робота. Если тип координат декартовый, то добавляет в конец сообщения кинематическую пару в соответствии с форматом сообщения. После этого отправляет номер команды и координаты. Получает ответ от робота и возвращает его без обработки.

Алгоритмы на клиенте готовы. Осталось написать алгоритм на роботе и добавить интерфейс с кнопками.

2.7. Программа управления робота при взаимодействии с клиентом

Напишем программу для робота на языке MELFA BASIC для реализации выполнения роботом команд, присылаемыех с клиента. Полностью программа представлена в приложении Е на листинге 13.

Краткое описание программы: робот ждет номер команды в переменную DCOMM. Есть вспомогательные переменные, куда будут записываться приходящие с клиента координаты – PHELP для декартовых, JHELP для сочлененных. Пока на робота не отправили команду ’0’, он в цикле ждет следующую команду. При получении команды ’1’, робот вышлет обратно свои текущие 12 координат и кинематическую пару. Если придет команда ’2’, то робот примет декартовы

координаты и кинематическую пару сразу после команды, с помощью MOV переместится в новые декартовы координаты, далее отправит ответ о своих текущих координатах (они должны соответствовать тем, что были отправлены). Аналогично при получении команды ’3’, однако координаты сочлененные и не нужна кинематическая пара. Если приходит команда ’0’, то цикл завершается, робот выключается, предварительно закрыв соединение.

A. Приложение

```
import socket
import connection_status as cs

class Client:
    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.
           SOCK_STREAM)
        self.connected = False

    def connect(self, ip: str, port: int) -> cs.
       ConnectionStatus:
        try:
            self.sock.connect((ip, port))
            self.connected = True

            return cs.ConnectionStatus.SUCCESS
        except socket.error:
            self.connected = False

            return cs.ConnectionStatus.CONNECTION_ERROR

    def disconnect(self) -> cs.ConnectionStatus:
        if self.connected:
            try:
                self.sock.close()
                self.connected = False

                return cs.ConnectionStatus.SUCCESS
            except socket.error:
                return cs.ConnectionStatus.CONNECTION_ERROR

```

```
30          return cs.ConnectionStatus.NOT_CONNECTED
31
32      def send(self, data: str) -> cs.ConnectionStatus:
33          if not self.connected:
34              return cs.ConnectionStatus.NOT_CONNECTED
35          try:
36              self.sock.sendall(data.encode('utf-8'))
37
38              return cs.ConnectionStatus.SUCCESS
39          except socket.error:
40               return cs.ConnectionStatus.SEND_ERROR
41
42      def receive(self, out_data: list, buffer_size: int = 1024)
            -> cs.ConnectionStatus:
43          if not self.connected:
44              return cs.ConnectionStatus.NOT_CONNECTED
45          try:
46              received = self.sock.recv(buffer_size)
47              out_data.clear()
48              out_data.append(received.decode('utf-8'))
49
50              return cs.ConnectionStatus.SUCCESS
51          except socket.error:
52              return cs.ConnectionStatus.RECEIVE_ERROR
```

Листинг 8: Основа для общения компьютера с роботом.

Б. Приложение

```
1  from typing import List
2  import re
3
4  class CommandMessageManager:
5      @staticmethod
6      def build_position_request(pos: List[float]) -> str:
7          if len(pos) != 6:
8              raise ValueError('List of pos must contain 6
                   values')
9
10         return f"({','.join(map(str, pos))})"
11
```

```
    @staticmethod
    def parse_position_response(response: list):
        try:
            matches = re.findall(r'\((.*?)\)', response[0])
            lists = [list(map(float, group.split(','))) for
               group in matches[:2]]
            lists.append(list(map(int, matches[2].split(',')))
               )

            return tuple(lists)
        except ValueError:
            return None
```

Листинг 9: Скрипт с функциями для обработки координат на отправку или получение.

B. Приложение

```
import yaml

class Config:
    ip: str = ""
    port: int = 0

    @classmethod
    def load(cls, path: str = "config.yaml"):
        with open(path, "r") as file:
            config = yaml.safe_load(file)
            cls.ip = config["connection"]["ip"]
            cls.port = config["connection"]["port"]

    @classmethod
    def save(cls, path: str = "config.yaml"):
        with open(path, "w") as file:
            yaml.safe_dump({
                "connection": {
                    "ip": cls.ip,
                    "port": cls.port
                }
            }, file)

```

```
    @classmethod
    def set_config(cls, ip: str, port: int):
        cls.ip = ip
        cls.port = port
```

Листинг 10: Скрипт для работы с чтением и сохранением конфигурации IP и порта робота.

Г. Приложение

```
from typing import List, Literal, Tuple

class VirtualRobot:
    def __init__(self,
                 cartesian: List[float] = None,
                 joint: List[float] = None,
                 kin_sol: Tuple[int, int] = None):
        self.cartesian = cartesian if cartesian is not None
           else [0.0] * 6
        self.joint = joint if joint is not None else [0.0] * 6
        self.kinematic_sol = kin_sol if kin_sol is not None
           else (0, 0)

    def update_cartesian(self, new_values: List[float],
       kinematic_sol: Tuple[int, int]):
        if len(new_values) != 6:
            raise ValueError("Cartesian must have 6 values")
        self.cartesian = new_values.copy()
        self.kinematic_sol = kinematic_sol

    def update_joint(self, new_values: List[float]):
        if len(new_values) != 6:
            raise ValueError("Joint must have 6 values")
        self.joint = new_values.copy()

    def calculate_next_move(
        self,
        mode: Literal["cartesian_linear", "cartesian_rotation"
           , "joint_rotation"],
        axis_index: int,
        step: float
```

```
    ) -> List[float]:
        if not (0 <= axis_index < 6):
            raise ValueError("Value of axis_index must be in
               range from 0 to 5")

        if "cartesian" in mode:
            new_pos = self.cartesian.copy()
        elif "joint" in mode:
            new_pos = self.joint.copy()
        else:
            raise ValueError("Expected 'cartesian' or 'joint'
               mode")

        new_pos[axis_index] += step
        return new_pos
```

Листинг 11: Скрипт виртуального робота: сохранение координат робота и кинематической пары. Вычисление следующих координат по индексу и шагу.

Д. Приложение

```
from typing import Tuple, Literal

from virtual_robot import VirtualRobot
from command_manager import CommandMessageManager
from robot_command import RobotCommand, MODE_TO_COMMAND
from client import Client
from connection_status import ConnectionStatus

class BackendController:
    def __init__(self):
        self.robot = VirtualRobot()
        self.client = Client()
        self.last_connection_status = ConnectionStatus.NONE
        self.last_cmd_send_status = ConnectionStatus.NONE
        self.last_params_send_status = ConnectionStatus.NONE
        self.last_receive_status = ConnectionStatus.NONE

    def connect(self, ip: str, port: int):
        self.last_connection_status = self.client.connect(ip,
           port)
```

```
        return self.client.connected

    def disconnect(self):
        cmd = str(RobotCommand.EXIT.value)
        self.last_cmd_send_status = self.client.send(cmd)

        self.last_connection_status = self.client.disconnect()

        return not self.client.connected

    def is_connected(self):
        return self.client.connected

    def get_pos(self):
        cmd = str(RobotCommand.GET_POSITION.value)
        self.last_cmd_send_status = self.client.send(cmd)

        ans = []
        self.last_receive_status = self.client.receive(ans)

        return ans

    def move_axis(self,
                  move_info: Tuple[Literal["cartesian_linear",
                      "cartesian_rotation", "joint_rotation"],
                      int, Literal['+', '-']],
                  step: float):
        mode, axis_index, direction = move_info

        signed_step = step if direction == '+' else -step
        pos = self.robot.calculate_next_move(mode, axis_index,
            signed_step)

        cmd = str(MODE_TO_COMMAND[mode].value)

        params = CommandMessageManager.build_position_request(
           pos)
        if "cartesian" in mode:
            params = f"{params}({','.join(map(str, self.robot.
               kinematic_sol))})"

```

```
56          self.last_cmd_send_status = self.client.send(cmd)
57          self.last_params_send_status = self.client.send(params
               )
58
59          ans = []
60          self.last_receive_status = self.client.receive(ans)
61
62          return ans
```

Листинг 12: Скрипт, объединяющий различные функции для полноценного взаимодействия с роботом с компьютера.

E. Приложение

```
1  JOVRD 100
2  SPD 100
3  DEF INTE DCOMM
4  DCOMM = 1
5  PHELP = P_CURR
6  JHELP = J_CURR
7  SERVO ON
8  OPEN "COM3:" AS #1
9
10 WHILE DCOMM > 0
11      INPUT #1, DCOMM
12      IF DCOMM = 1 THEN
13          PRINT #1, J_CURR, P_CURR
14      ENDIF
15      IF DCOMM = 2 THEN
16          INPUT #1, PHELP
17          MOV PHELP
18          PRINT #1, J_CURR, P_CURR
19      ENDIF
20      IF DCOMM = 3 THEN
21          INPUT #1, JHELP
22          MOV JHELP
23          PRINT #1, J_CURR, P_CURR
24      ENDIF
25 WEND
26
27 CLOSE #1
```

```
28 SERVO OFF
29 END
```

Листинг 13: Программа на роботе-манипуляторе для выполнения команд, присылаемых клиентом.