

**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 ТСР)»**

Выполнил: студент гр. R3341

А. А. Румянцев

Проверил: преподаватель

доцент, старший научный сотрудник, инженер В. С. Громов

Санкт-Петербург

2025 г.

## Содержание

<b>Введение</b>	<b>3</b>
<b>1 Подготовка к написанию программы</b>	<b>4</b>
1.1 Ознакомление с объектом работы . . . . .	4
1.2 Выбор подходящего языка программирования . . . . .	4
<b>2 Реализация программы для взаимодействия компьютера с роботом-манипулятором</b>	<b>5</b>
2.1 Определение формата сообщения . . . . .	5
2.2 Определение основных команд . . . . .	6
2.3 Клиентская программа для общения с роботом . . . . .	6
2.4 Программа на роботе для общения с клиентом . . . . .	8
2.5 Проверка связи между компьютером и роботом . . . . .	8
<b>А Приложение</b>	<b>11</b>
<b>Б Приложение</b>	<b>12</b>
<b>В Приложение</b>	<b>13</b>

## 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 (сам интерфейс), другая же обеспечивает взаимодействие с роботом на уровне, не видимом 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 понадобилась 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.

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

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

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 по типу '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.

```
1 class RobotCommand(Enum):  
2     EXIT = 0  
3     GET_POSITION = 1  
4     MOVE_LINEAR = 2  
5     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 сетевого взаимодействия компьютера с роботом.

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

```

1 class ConnectionStatus(Enum):
2     SUCCESS = auto()
3     NOT_CONNECTED = auto()
4     CONNECTION_ERROR = auto()
5     SEND_ERROR = auto()
6     RECEIVE_ERROR = auto()
7     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 6.

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); булево 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:

```
1 JOVRD 100
2 SPD 100
3 DEF INTE DCOMM
4 DCOMM = 1
5 PHELP = P_CURR
6 SERVO ON
7 OPEN "COM3:" as #1
8
9 WHILE DCOMM > 0
10     INPUT #1, DCOMM
11     IF DCOMM = 2 THEN
12         INPUT #1, PHELP
13         PRINT #1, PHELP
14     ENDIF
15 WEND
16
17 CLOSE #1
18 SERVO OFF
19 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 7 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:

```
1 connection:
2   ip: "192.168.0.4"
3   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 8.

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:

```
1 import socket
2 from config_manager import Config
```

```

3 from command_manager import CommandMessageManager
4
5 if __name__ == "__main__":
6     Config.load()
7
8     pos = [100.0, -100.0, 100.0, 100.0, 100.0, 100.0]
9     test = CommandMessageManager.build_position_request(pos)
10    test += "(7,0)"
11    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12    s.connect((Config.ip, Config.port))
13    s.sendall("2".encode('utf-8'))
14    s.sendall(test.encode('utf-8'))
15    ans = s.recv(1024).decode('utf-8')
16    print(ans)
17    s.sendall("0".encode('utf-8'))
18    s.close()

```

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

Краткое описание программы: загружается конфигурация IP и порта из файла. Задается некоторая позиция, из которой 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 и сокет закрывается.

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

## А. Приложение

```
1 import socket
2 import connection_status as cs
3
4 class Client:
5     def __init__(self):
6         self.sock = socket.socket(socket.AF_INET, socket.
7             SOCK_STREAM)
8         self.connected = False
9
10    def connect(self, ip: str, port: int) -> cs.
11        ConnectionStatus:
12        try:
13            self.sock.connect((ip, port))
14            self.connected = True
15
16            return cs.ConnectionStatus.SUCCESS
17        except socket.error:
18            self.connected = False
19
20            return cs.ConnectionStatus.CONNECTION_ERROR
21
22    def disconnect(self) -> cs.ConnectionStatus:
23    if self.connected:
24        try:
25            self.sock.close()
26            self.connected = False
27
28            return cs.ConnectionStatus.SUCCESS
29        except socket.error:
30            return cs.ConnectionStatus.CONNECTION_ERROR
31
32    return cs.ConnectionStatus.NOT_CONNECTED
```

```

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)
43         -> cs.ConnectionStatus:
44         if not self.connected:
45             return cs.ConnectionStatus.NOT_CONNECTED
46         try:
47             received = self.sock.recv(buffer_size)
48             out_data.clear()
49             out_data.append(received.decode('utf-8'))
50
51             return cs.ConnectionStatus.SUCCESS
52         except socket.error:
53             return cs.ConnectionStatus.RECEIVE_ERROR

```

Листинг 6: 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
9                 values')
10
11         return f"({'','.join(map(str, pos))})"
12
13     @staticmethod
14     def parse_position_response(response: list):

```

```

14         try:
15             matches = re.findall(r'\((.*?)\)', response[0])
16             lists = [list(map(float, group.split(','))) for
17                       group in matches[:2]]
18             lists.append(list(map(int, matches[2].split(','))))
19
20             return tuple(lists)
21         except ValueError:
22             return None

```

Листинг 7: Скрипт с функциями для обработки координат на отправку или получение.

## В. Приложение

```

1 import yaml
2
3 class Config:
4     ip: str = ""
5     port: int = 0
6
7     @classmethod
8     def load(cls, path: str = "config.yaml"):
9         with open(path, "r") as file:
10             config = yaml.safe_load(file)
11             cls.ip = config["connection"]["ip"]
12             cls.port = config["connection"]["port"]
13
14     @classmethod
15     def save(cls, path: str = "config.yaml"):
16         with open(path, "w") as file:
17             yaml.safe_dump({
18                 "connection": {
19                     "ip": cls.ip,
20                     "port": cls.port
21                 }
22             }, file)
23
24     @classmethod
25     def set_config(cls, ip: str, port: int):

```

```
26         cls.ip = ip
27         cls.port = port
```

Листинг 8: Скрипт для работы с чтением и сохранением конфигурации IP и порта робота.