МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ ИМЕНИ ПАТРИСА ЛУМУМБЫ»

Факультет физико-математических и естественных наук

Кафедра информационных технологий

«Допустить к защите»

Заведующий кафедрой
информационных технологий
д.ф.-м.н.
_____ Ю.Н. Орлов

«___»_____ 20_Г.

Выпускная квалификационная работа бакалавра

Направление 02.03.02 «Фундаментальная информатика и информационные технологии»

ТЕМА «Разработка системы управления робототехническим манипулятором на подвижной платформе TurtleBot3»

Выполнил студент **Ломакина София Васильевна**

(Фамилия, имя, отчество)

Группа НФИбд-02-19

Студ. билет № 1032179140

Руководитель выпускной квалификационной работы

Киселёв Г. А., к.т.н., ст. преподаватель
(Ф.И.О., степень, звание, должность)

(Подпись)

Автор

(Подпись)

г. Москва

2023 г.

Федеральное государственное автономное образовательное учреждение

высшего образования

«Российский университет дружбы народов имени Патриса Лумумбы»

АННОТАЦИЯ

выпускной квалификационной работы

Ломакиной Софии Васильевны

на тему: Разработка системы управления робототехническим манипулятором на подвижной платформе TurtleBot3

Объем дипломного проекта — 48 страниц. Он содержит 25 рисунков, 7 источников литературы. Работа содержит введение, три главы (теоретический раздел, проектирование системы, построение прототипа системы), заключение,список литературных источников и приложение.

Во введении определяется тема работы, её актуальность, а также определяются цели и задачи работы, предоставляются сведения об апробации и публикациях по теме.

Первая глава содержит изучение современных робототехнических манипуляторов и методов управления ими, а также методы машинного обучения. Представлены классические алгоритмы машинного обучения.

Вторая глава посвящена системе ROS и ее методам. Представлена установка этой системы и объяснена ее внутренняя структура, файловая система.

В третьей главе описывается реализация системы, а именно: физическое проектирование, включающее в себя моделирование и управление робототехническим манипулятором и платформой TurtleBot3

В заключении представлены результаты и выводы, полученные в процессе работы.

Автор ВКР _____ _____

(Подпись) (ФИО)

Оглавление

АННОТАЦИЯ 2
Введение 4
1. МЕТОДЫ РЕШЕНИЯ ЗАДАЧИ УПРАВЛЕНИЯ РОБОТОТЕХНИЧЕСКИМ МАНИПУЛЯТОРОМ 5
1.1. Обзор робототехнических реализаций 5
1.2. Обзор методов машинного обучения 8
1.3. Обзор алгоритмов обучения с подкреплением 11
1.3.1. Q-Learning 11
1.3.2. Deep Q-Network (DQN) 15
1.3.3. Deep Deterministic Policy Gradient (DDPG) 20
2. РАЗРАБОТКА СИСТЕМЫ УПРАВЛЕНИЯ 20
2.1. Что такое ROS 21
2.2. Уровень файловой системы 21
2.3. Уровень вычислений 23
2.4. Требования к моделированию 27
2.5. Особенности работы манипулятора на подвижной платформе 28
2.6. OpenMANIPULATOR на платформе TurtleBot3 29
3. МОДЕЛИРОВАНИЕ В СРЕДЕ GAZEBO 30
3.1. Turtlebot3 30
3.2. Open Manipulator 36
3.3. Машинное обучение 38
Заключение 43
Список использованных источников и литературы 44
Приложение А 45
Приложение Б 45

Введение

Современная робототехника становится всё более значимой сферой науки и техники, в которой активно идёт работа над созданием различных роботизированных систем, автоматических устройств и механизмов. Разнообразные робототехнические манипуляторы наиболее востребованы на производственных предприятиях, в логистике, транспортировке, складском хозяйстве и в многих других отраслях промышленности.

В текущем исследовании рассмотрена проблема моделирования движений робототехнического манипулятора на подвижной платформе turtlebot3 с использованием алгоритмов машинного обучения с подкреплением. Одной из проблем, которые возникают при проектировании и моделировании робототехнических систем, является разработка эффективных алгоритмов управления движением манипулятора на подвижной платформе. В связи с этим имеет большое значение разработка новых методов управления робототехническими манипуляторами, основанных на алгоритмах машинного обучения.

Цель данного исследования - разработка и анализ алгоритмов машинного обучения с подкреплением для управления робототехническим манипулятором на подвижной платформе turtlebot3. Результаты данного исследования могут быть использованы для более эффективного моделирования и управления робототехническими системами, а также для повышения их производительности и безопасности работы.

1. МЕТОДЫ РЕШЕНИЯ ЗАДАЧИ УПРАВЛЕНИЯ РОБОТОТЕХНИЧЕСКИМ МАНИПУЛЯТОРОМ

1.1. Обзор робототехнических реализаций

Современные методы управления роботами-манипуляторами представляют собой сложную и динамическую область исследований и разработок. Они охватывают широкий спектр технологий, от традиционных контроллеров и программного обеспечения до искусственного интеллекта и машинного обучения.

Одним из наиболее распространенных методов управления роботами-манипуляторами является программируемое управление, которое позволяет программировать робота для выполнения определенных задач. Этот метод в основном используется в производственном окружении, где роботы могут выполнять множество однотипных задач. Однако в современных условиях, когда требования к гибкости и адаптивности увеличиваются, приходится использовать новые методы управления роботами-манипуляторами. Такие методы включают в себя управление на основе обратной связи, где информация о положении и скорости робота используется для определения наилучшей траектории движения.

Другой метод - это управление на основе показателей, где роботы могут использовать информацию о силе и давлении для регулирования своих движений. Этот метод особенно полезен в области монтажа, где роботы могут использовать силу и давление для установки и фиксации деталей.

Современные методы управления роботами-манипуляторами также включают в себя искусственный интеллект и машинное обучение. Эти методы позволяют роботам учиться и адаптироваться, в результате чего они могут самостоятельно находить решения для выполнения сложных задач.

Роботы-манипуляторы обычно имеют от трех до шести степеней свободы, обеспечиваемых серией суставов, которые соединяют различные звенья устройства.

Соединения манипулятора обычно представляют собой вращательные или призматические соединения.

- Шарнирное соединение (поворотное соединение): вращение вокруг одной оси, т. е. крутящее движение.
- Призматический шарнир (линейный шарнир): поступательное движение вокруг одной оси, т. е. разгибательное движение.
- Цилиндрический шарнир: вращение и перемещение вокруг одной оси
- Шарнирное соединение (шарнирное соединение): три градуса вращения

Тип руки определяет количество ее степеней свободы, а также ее рабочее пространство (насколько далеко она может дотянуться).

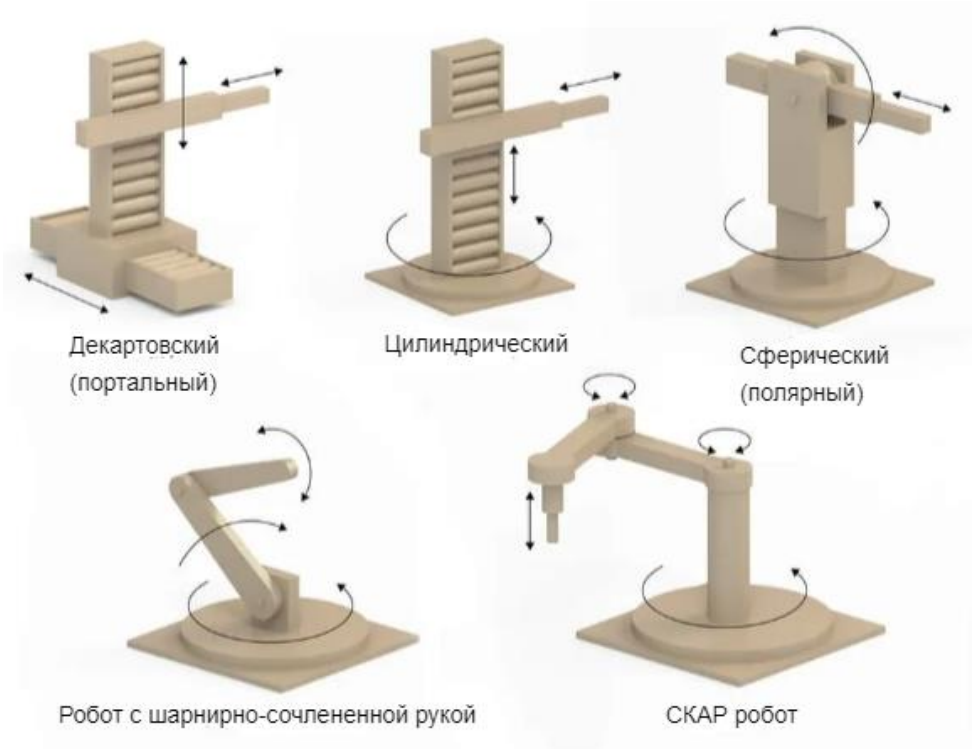

Рис. 1.1. Различные конфигурации манипулятора робота

Декартовы или портальные манипуляторы могут перемещаться по трем осям через систему рельсов, как печатающая головка 3D-принтера для моделирования наплавленным методом (FDM). Их очень легко программировать, и они могут поднимать тяжелые грузы. Обычное использование включает в себя самовывоз и нанесение герметика.

Цилиндрические роботы-манипуляторы имеют вращающееся соединение в основании и два призматических соединения: одно для перемещения по оси Z и одно для перемещения по горизонтальной плоскости. Эта роботизированная система жесткая и точная, но встречается реже, чем другие типы.

Сферические или полярные роботы-манипуляторы имеют два вращающихся шарнира, в том числе один в основании, и третий призматический шарнир, позволяющий разгибать руку. Эти манипуляторы требуют менее сложных алгоритмов и полезны для операций сварки и захвата.

Роботы с шарнирными руками предлагают максимально возможные степени свободы и обычно имеют от трех до шести вращающихся шарниров. Самым большим преимуществом робота с шарнирной рукой является его широкий диапазон движений; однако он может быть менее точным, чем другие типы. Общие области применения включают дуговую сварку и окраску распылением.

Манипуляторы Selective Compliance Assembly Robot Arm (SCARA) имеют один призматический шарнир для перемещения по оси Z и два параллельных вращающихся шарнира. Роботы SCARA широко используются при сборке. Они часто могут поднимать более тяжелые грузы, чем другие типы роботов.

Роботы-манипуляторы широко используются в промышленных условиях благодаря их точности, воспроизводимости, возможностям автоматизации, способности поднимать тяжелые грузы и универсальности с точки зрения различных концевых эффекторов, которые можно использовать.

Промышленное использование роботов-манипуляторов включает:

- Сборка: когда манипулятор используется для добавления различных компонентов в сложную деталь путем прикрепления каждого компонента в правильном месте.

- Pick and place: когда манипулятор используется для перемещения объекта из одной области в другую, например, с рабочего стола на поддон.
- Упаковка: когда манипулятор используется для помещения объекта в коробку, поддон или другую форму упаковки перед, в некоторых случаях, запечатыванием упаковки.
- Контроль качества: манипуляторы можно использовать для стресс-тестирования определенных деталей, например, вытягивая или скручивая их.

1.2. Обзор методов машинного обучения

Машинное обучение - это раздел искусственного интеллекта, который изучает методы, алгоритмы и технологии, позволяющие компьютерной системе извлекать закономерности из набора данных, обучаться на основе этих данных и делать предсказания или принимать решения на основе полученных знаний. Машинное обучение применяется во многих областях, таких как обработка естественного языка, компьютерное зрение, рекомендательные системы и др.

Центральной идеей машинного обучения является существующая математическая связь между любой комбинацией входных и выходных данных. Модель машинного обучения не имеет сведений об этой взаимосвязи заранее, но может сгенерировать их, если будет предоставлено достаточное количество наборов данных. Это означает, что каждый алгоритм машинного обучения строится вокруг модифицируемой математической функции.

По способам обучения машинное обучение включает в себя:

1. **Supervised learning** - контролируемое обучение (обучение с учителем).

Задача машинного обучения с учителем состоит в том, чтобы найти зависимость между входными данными (так называемыми признаками) и выходными данными (так называемыми метками классов). Эта зависимость представляется моделью, которая обучается на обучающих данных, предоставленных в качестве входных

данных. Используя эту модель, мы можем предсказать значение выходных данных классов для новых входных данных, которые не использовались во время обучения.

Математически задача машинного обучения с учителем может быть представлена как задача минимизации функции ошибки (также называемой функции потерь), которая измеряет насколько хорошо модель предсказывает выходные данные для входных данных. Чтобы найти модель, которая минимизирует функцию ошибки, мы используем алгоритмы оптимизации, такие как метод стохастического градиентного спуска или методы определения градиента, чтобы скорректировать параметры модели на каждом шаге обучения.

Таким образом, математическая постановка задачи машинного обучения с учителем сводится к определению функции, которая описывает зависимость между входными и выходными данными, и нахождению параметров этой функции, которые минимизируют функцию ошибки во время обучения.

2. **Unsupervised learning** - неконтролируемое обучение (обучение без учителя)

В задаче машинного обучения без учителя мы не имеем явных выходных данных, которые мы хотим предсказать. Вместо этого, мы стремимся кластеризовать или понизить размерность данных, выделить скрытые структуры или выделять аномалии в данных.

Математическая постановка задачи машинного обучения без учителя может быть представлена как минимизация функции потерь, которая измеряет ошибку кластеризации или понижения размерности данных. Например, в задаче кластеризации, мы должны найти способ разбить данные на несколько кластеров, где объекты в каждом кластере сильно коррелируют друг с другом, но имеют мало связей между кластерами. Используя различные методы кластеризации, мы можем оптимизировать функцию потерь, такую как инерция кластеризации, которая измеряет, насколько сильно кластеры различаются друг от друга.

В задачах понижения размерности, мы стараемся определить наиболее информативные признаки, которые отображают изначальные данные в пространство меньшей размерности. Мы можем оптимизировать функцию потерь,

такую как среднеквадратическая ошибка, которая измеряет разницу между изначальными данными и их проекцией в пространство меньшей размерности.

Таким образом, математическая постановка задачи машинного обучения без учителя состоит в определении функции, которая описывает скрытые закономерности в данных, и в оптимизации параметров этой функции с помощью методов оптимизации, таких как градиентный спуск или эволюционные алгоритмы.

3. **Semi-supervised learning** - полуавтоматическое обучение или частичное обучение (обучение с частичным привлечением учителя)

Задача машинного обучения с частичным привлечением учителя заключается в том, чтобы обучить модель на данных, где только часть данных размечена (есть метки классов), а остальные данные не размечены. Машина должна использовать как размеченную, так и неразмеченную информацию для наилучшего обучения.

Основная суть этой модели заключается в том, что неразмеченные данные могут содержать ценную информацию об обучающих данных. Эти данные могут помочь определить особенности, которые могут быть упущены при обучении только с размеченными данными.

Типичный пример задачи, где может использоваться semi-supervised learning, это задача классификации изображений, где требуется определить наличие объекта на фотографии. Если у нас есть много неразмеченных фотографий с объектами, которые мы хотим обнаружить, то можно использовать эти неразмеченные данные для улучшения качества модели и повышения точности классификации.

4. **Reinforcement learning** - обучение с подкреплением

Задача машинного обучения с подкреплением состоит в том, чтобы обучить агента, принимающего последовательность решений в динамической среде, получая за каждое принятое решение награду или штраф в зависимости от его действий. Основная цель агента состоит в максимизации награды, полученной за взаимодействие со средой.

Эта задача может быть сформулирована с помощью марковского процесса принятия решений (MDP), который описывает динамику взаимодействия агента со средой. MDP имеет четыре компонента: множество состояний, множество действий, функцию перехода и функцию награды. Функция перехода определяет вероятность перехода из одного состояния в другое, если агент совершает определенное действие. Функция награды определяет количество награды, полученной агентом за совершенное действие на определенном состоянии.

Цель алгоритма машинного обучения состоит в том, чтобы получить стратегию выбора действий, которая максимизирует сумму награды, получаемой агентом за взаимодействие со средой. Для этого используются различные направления: методы без модели, которые основываются на обучении из опыта, и методы с моделью, которые используют базовые знания о среде для оптимизации стратегии.

1.3. Обзор алгоритмов обучения с подкреплением

1.3.1. Q-Learning

Q-Learning - это алгоритм обучения с подкреплением, который может быть использован для обучения агентов принимать оптимальные решения в динамических окружениях. Суть алгоритма заключается в том, что агент анализирует текущую ситуацию, выбирает наилучшее действие и осуществляет его. Затем агент наблюдает за состоянием окружающей среды, которое изменяется в результате совершенного действия, и получает награду. Данные полученные из такого взаимодействия между окружающим миром и агентом, используются для обновления внутренней информации агента о том, какие действия приводят к каким наградам в каждой из ситуаций.

В Q-Learning каждый раз агент выбирает действие, которое максимизирует значение функции качества действий (Q-функции), которая хранит информацию о том, какой же выигрыш ожидается от каждого действия в каждом состоянии. Значение Q-функции может быть вычислено известными формулами обновления, которые

используют количество полученной награды, ожидаемый будущий выигрыш и текущее значение Q-функции в состоянии и совершенном действии.

Первоначально Q-функция заполняется случайными значениями, однако с каждым новым опытом она настраивается для получения наилучших значений. Этот процесс может продолжаться на протяжении многих итераций, в ходе которых агент пытается найти оптимальные решения в различных ситуациях.

С математической точки зрения Q-learning это без модельный алгоритм ОП, основанный уравнении Беллмана:

$$Q(s,a) = r + \gamma max_{a'} Q(s',a')$$

Уравнение утверждает, что значение Q для определенной пары состояние-действие должно быть наградой, полученной при переходе в новое состояние (путем выполнения этого действия), добавленной к значению наилучшего действия в следующем состоянии.

Реализацию алгоритма Q-Learning можно выразить следующим образом:

```
import numpy as np

# определяем граф смежности, хранящий значения вознаграждения за
перемещение между вершинами
# -1 означает, что перемещение между вершинами невозможно
R = np.array([
        [-1, -1, -1, -1, 0, -1],
        [-1, -1, -1, 0, -1, 100],
        [-1, -1, -1, 0, -1, -1],
        [-1, 0, 0, -1, 0, -1],
        [0, -1, -1, 0, -1, 100],
        [-1, 0, -1, -1, 0, 100]
        ])

# определяем Q-таблицу, заполняем ее нулями
Q = np.zeros([6, 6])

# устанавливаем коэффициент дисконтирования
```

```
gamma = 0.8

# определяем число эпизодов обучения
num_episodes = 1000

# обучение
for episode in range(num_episodes):
    # выбираем случайное начальное состояние
    state = np.random.randint(0, 6)
    while state != 5:
        # находим все возможные действия, и выбираем одно из них
случайным образом
        possible_actions = []
        for action in range(6):
            if R[state, action] != -1:
                possible_actions.append(action)
        next_state = possible_actions[np.random.randint(0,
len(possible_actions))]

        # обновляем значение Q-таблицы
        Q[state, next_state] = R[state, next_state] + gamma *
np.max(Q[next_state])

        # перемещаемся в следующее состояние
        state = next_state

# выполняем тестирование обученной модели
current_state = 2
steps = [current_state]

while current_state != 5:
    next_step_index = np.where(Q[current_state] ==
np.max(Q[current_state]))[0]

    if len(next_step_index) > 1:
        next_step_index = np.random.choice(next_step_index, size=1)
    else:
        next_step_index = next_step_index[0]

    steps.append(next_step_index)
    current_state = next_step_index

print(steps)
```

В данной программе решается задача выбора наиболее выгодного маршрута по графу, где вершины являются состояниями, а их значения вознаграждения - выгодностью перехода между вершинами.

Сначала определяется граф смежности, хранящий значения вознаграждения за перемещение между вершинами. Далее, создается Q-таблица, заполненная нулями. Далее, в цикле проходим итерации по заданному числу эпизодов обучения, при каждом из которых выбираем случайное начальное состояние и перемещаемся по графу, выбирая случайное действие на каждом шаге. После каждого перемещения происходит обновление Q-таблицы в соответствии с выражением:

```
Q[state, next_state] = R[state, next_state] + gamma * np.max(Q[next_state])
```

,где state - текущее состояние, next_state - следующее состояние, R - граф смежности с выгодностью перемещения между вершинами, Q - Q-таблица, gamma - коэффициент дисконтирования.

В конце обучения выполняется тестирование обученной модели - выбирается начальное состояние и перемещение по графу до тех пор, пока не будет достигнута конечная вершина.

1.3.2. Deep Q-Network (DQN)

Deep Q-Network (DQN) - это алгоритм глубокого обучения обратной связи, который был разработан для решения проблемы продвинутых игр на Atari. Он использует нейронную сеть для оценки ценности действий в окружении и обновляет ее, чтобы улучшить качество принимаемых решений. DQN является одним из наиболее известных алгоритмов обучения с подкреплением и позволяет создавать агентов, способных самостоятельно обучаться и принимать лучшие решения в сложных средах.

Алгоритм DQN построен на основе технологии Q-обучения (Q-learning), применяемой в обучении с подкреплением. Он использует нейронную сеть, которая принимает состояние среды, выполняет преобразование и возвращает значения для каждого возможного действия. Нейронная сеть обучается обновлять Q-функцию, которая выражает значение каждого действия в окружении на основе текущего состояния. Эта обученная Q-функция используется для выбора действия, которое должен выполнить агент в следующем шаге.

DQN включает ряд технических усовершенствований, которые решают проблемы, возникающие при обучении с подкреплением с использованием нейронных сетей. Например, он использует методы реплея (replay) и фиксированной Q-цели (fixed Q-target), чтобы уменьшить нестабильность обучения и предотвратить очень большие изменения ценности действий, которые могут привести к выбору неправильного действия.

Метод реплея предполагает сохранение копий всех состояний, которые агент проходил на протяжении игры. Эти состояния воспроизводятся случайным образом во время процесса обучения, что позволяет нейронной сети рассматривать каждое состояние регулярно и избежать зацикливания на определенных шаблонах.

Метод фиксированной Q-цели заключается в использовании отдельной целевой Q-сети для вычисления ценности действий в окружении. Это предотвращает схлопывание (collapse) сети, которое может произойти при обучении с подкреплением, когда она использует свои же значения Q-функции во время обучения.

Классической реализацией алгоритма Deep Q-Network (DQN) является игра Atari. Поэтому, приведу пример реализации алгоритма DQN на примере игры Atari - Pong.

Первым шагом будет установка необходимых библиотек.

```
import random
```

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import gym
import time
```

Далее, создается объект для взаимодействия с игрой Pong.

```
env = gym.make("Pong-v0")
```

Следующим шагом является создание класса DQN, который будет использоваться для обучения модели.

```
class DQN:
    def __init__(self, state_space, action_space):
        self.state_space = state_space
        self.action_space = action_space
        self.memory = []
        self.gamma = 0.99
        self.eps = 1.0
        self.eps_decay_rate = 0.995
        self.eps_min = 0.01
        self.batch_size = 64
        self.model = self.create_model()
        self.target_model = self.create_model()
        self.update_target_model()

    def create_model(self):
        model = tf.keras.models.Sequential([
            tf.keras.layers.Dense(64, activation="relu",
input_dim=self.state_space),
            tf.keras.layers.Dense(32, activation="relu"),
            tf.keras.layers.Dense(self.action_space,
activation="linear")
        ])

        model.compile(loss="mse",
optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001))
        return model

    def act(self, state):
        if np.random.rand() < self.eps:
```

```
            return np.random.choice(self.action_space)
        return np.argmax(self.model.predict(state)[0])

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))
        if len(self.memory) > 10000:
            self.memory.pop(0)

    def replay(self):
        if len(self.memory) < self.batch_size:
            return

        X = np.zeros((self.batch_size, self.state_space))
        Y = np.zeros((self.batch_size, self.action_space))

        mini_batch = random.sample(self.memory, self.batch_size)

        for index, (state, action, reward, next_state, done) in
enumerate(mini_batch):
            target = self.target_model.predict(state)[0]

            if done:
              target[action] = reward
            else:
              target[action] = reward + self.gamma *
np.max(self.target_model.predict(next_state)[0])

            X[index] = state
            Y[index] = target

        self.model.fit(X, Y, epochs=1, batch_size=self.batch_size,
verbose=0)

        if self.eps > self.eps_min:
            self.eps *= self.eps_decay_rate

    def update_target_model(self):
        self.target_model.set_weights(self.model.get_weights())
```

В данном классе используются следующие параметры:

- state_space - количество состояний игры;
- action_space - количество возможных действий игрока;
- gamma - дисконтный фактор;

- eps - начальное значение параметра ε-жадности;
- eps_decay_rate - скорость затухания параметра ε-жадности;
- eps_min - минимальное значение параметра ε-жадности;
- batch_size - размер батча;
- memory - списко для хранения предыдущих состояний игры;
- model - основная модель нейронной сети для обучения;
- target_model - целевая модель нейронной сети;
- create_model() - функция для создания нейронной сети;
- act() - функция для совершения действия в игре;
- remember() - функция, для запоминания предыдущих состояний игры и действий игрока;
- replay() - функция для обучения модели на батчах;
- update_target_model() - функция для обновления целевой модели нейронной сети.

Далее, производится обучение модели.

```
num_episodes = 1000
MAX_STEP = 20000
num_steps = 0

agent = DQN(6400, 3)

for episode in range(num_episodes):
    state = env.reset()
    state = np.reshape(state, [1, 6400])
    done = False
    score = 0

    while not done and num_steps < MAX_STEP:

        #env.render()

        action = agent.act(state)

        next_state, reward, d
       one, _ = env.step(action)
        next_state = np.reshape(next_state, [1, 6400])
```

```
        reward = reward if not done else -10

        score += reward
        num_steps += 1

        agent.remember(state, action, reward, next_state, done)
        agent.replay()

        if num_steps % 100 == 0:
            agent.update_target_model()

        state = next_state

    print("Episode:", episode, "Score:", score, "Steps:", num_steps,
"Epsilon:", agent.eps)
```

Для обучения модели задается количество эпизодов и максимальное количество шагов в каждом эпизоде.
Далее инициализируется агент и запускается игра.

В цикле обучения происходит следующее:

1. Считывание текущего состояния игры;
2. На основе текущего состояния выбор действия с помощью функции act();
3. Выполнение действия в игре и получение следующего состояния, награды и признака окончания эпизода;
4. Обучение модели на батче, полученном из сохраненных состояний и действий;
5. Обновление модели на каждых 100 шагах.

В конце игры выводится на экран количество очков, количество шагов и значение параметра ε-жадности.

1.3.3. Deep Deterministic Policy Gradient (DDPG)

Алгоритм Deep Deterministic Policy Gradient (DDPG) является алгоритмом глубокого обучения с подкреплением, который использует Deep Q Network (DQN) для обучения оценки ценности действий (Q-значений) и определения лучшего действия для каждого состояния окружающей среды.

Однако, в отличие от DQN, который может использовать только дискретные действия, DDPG позволяет работать с непрерывными пространствами действий. Кроме того, DDPG использует две нейронные сети - Actor и Critic, для определения лучшей стратегии поведения агента и оценки ценности действий.

В Actor-сети используется функция, которая на вход принимает состояние окружающей среды и выходит с выбранным действием. Critic-сеть используется для оценки ценности действий в данном состоянии.

DDPG использует off-policy метод, который позволяет обучаться на данных, которые были собраны другим агентом или использовать ранее собранные данные.

2. РАЗРАБОТКА СИСТЕМЫ УПРАВЛЕНИЯ

2.1. Что такое ROS

ROS (Robot Operating System) - это платформа для разработки программного обеспечения на основе открытого исходного кода, предназначенная для создания и управления роботизированными системами. ROS разработан для упрощения процесса создания и тестирования сложных роботизированных систем и обеспечивает набор инструментов и библиотек, которые могут быть использованы для реализации различных функций при создании роботов.

Основными особенностями ROS являются модульность, возможность многократного использования кода, высокая скорость передачи данных, использование стандартов связи и спецификаций для обмена данными, поддержка различных языков программирования, таких как C++, Python, Java и др. Платформа также предоставляет широкий набор инструментов для быстрой разработки роботов, таких как библиотеки для управления роботами, симуляторы, инструменты визуализации и т.д.

ROS успешно применяется в различных областях, таких как промышленность, медицина, образование, исследования и т.д. В настоящее время ROS является одной из наиболее популярных платформ для разработки роботов и представляет собой важный инструмент для инженеров и исследователей, которые занимаются созданием роботизированных систем и их компонентов.

2.2. Уровень файловой системы

Программы ROS организуются с помощью пакетов, каждый из которых содержит все необходимые файлы - cpp, py, параметры, конфигурации и т.д. Несколько пакетов, имеющих общую цель, могут быть объединены в метапакет (стек), а иерархия файловой системы ROS показывает связь между пакетами и метапакетами. Система ROS устанавливается в корневой каталог Linux и использует иерархию каталогов, определенную стандартом иерархии файловых систем (FHS) в Linux.

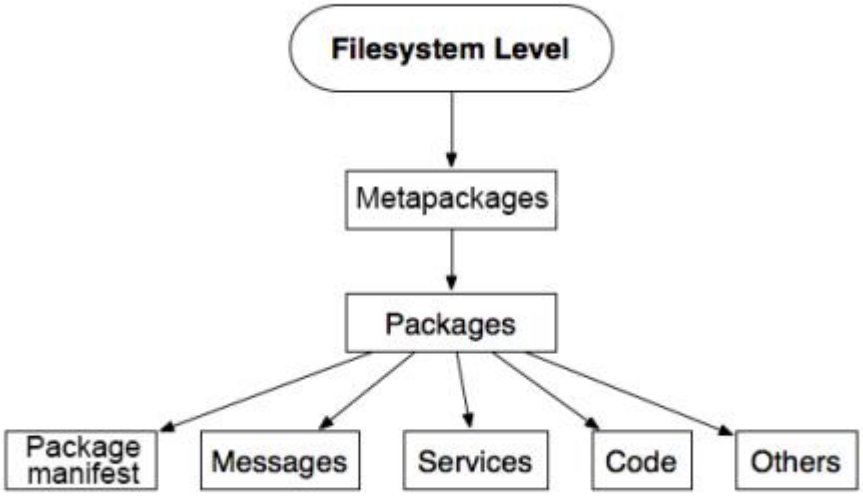

Рис.2.1 Файловая система ROS

Мастерская (рабочее пространство - workspace) - это каталог, содержащий пакеты с исходными файлами, которые можно скомпилировать, используя мастерскую. Это удобно, когда нужно скомпилировать несколько пакетов одновременно и централизовать все разработки. Вы можете создать свою мастерскую и назвать ее как угодно, где угодно. В ROS по умолчанию мастерской является директория catkin_ws, где вы можете вносить изменения, компилировать и устанавливать пакеты catkin. Путь: /home/user/catkin_ws. Для перехода в директорию рабочего пространства в терминале Linux нужно ввести команду: «cd ~/catkin_ws».

Каталог, содержащий несколько папок, предоставляет различные пространства с разными функциями. Например, исходное пространство (src) предназначено для хранения своих пакетов, проектов и клонированных пакетов с одним из важных файлов - CMakeLists.txt, который используется для сборки пакета с помощью

инструмента CMake. Пространство сборки (build) содержит промежуточные файлы для пакетов и проектов, а пространство разработки (devel) используется для хранения скомпилированных программ и их тестирования без установки или для экспортирования пакета. Для перемещения между пакетами и их файлами ROS обеспечивает нам полезную утилиту rosbash, которая является подмножеством bash Linux и имеет подобные команды. Интерпретатор оболочки bash Linux используется для выполнения навигации по файловой системе, запуска программ и общения с устройствами.

2.3. Уровень вычислений

Данный уровень представляет собой граф вычислений - сеть процессов ROS, которые совместно обрабатывают данные.

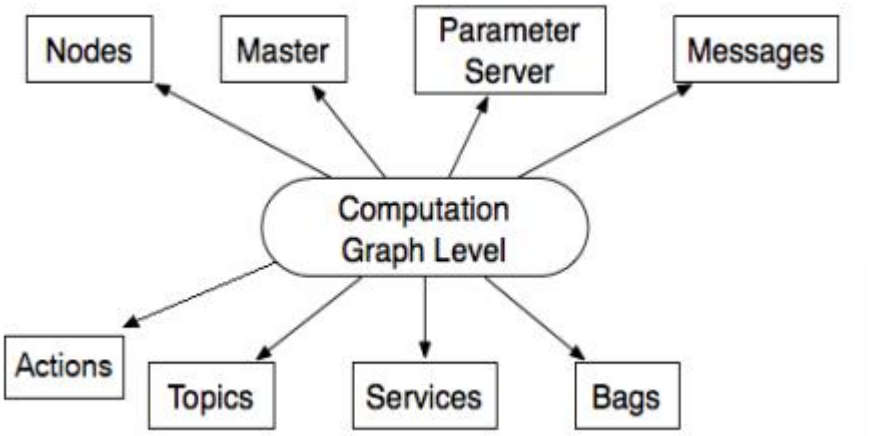

Рис.2.2. Составляющие графа вычислений

Master (мастер) в системе ROS присваивает названия узлам и сохраняет их в системе; отслеживает издателей и подписчиков, связанных с определенными темами. Он играет ключевую роль для соединения узлов ROS между собой, похоже на сервер DNS. Запуск мастера осуществляется через команду "roscore", которая также запускает сервер параметров ROS и систему логирования rosout.

Nodes (узлы) в ROS - это исполняемые файлы, позволяющие программам взаимодействовать с другими процессами. Использование узлов в системе ROS имеет много преимуществ: увеличивает отказоустойчивость, изолирует код и

функциональные возможности, делает систему более удобной. Узлы также могут обмениваться сообщениями, состоящими из данных, команд или другой информации, необходимой для решения конкретных задач.

На рисунках 2.3 и 2.4 показана схема установления соединения между процессами (Узел – мастер; мастер – узел; узел – узел).

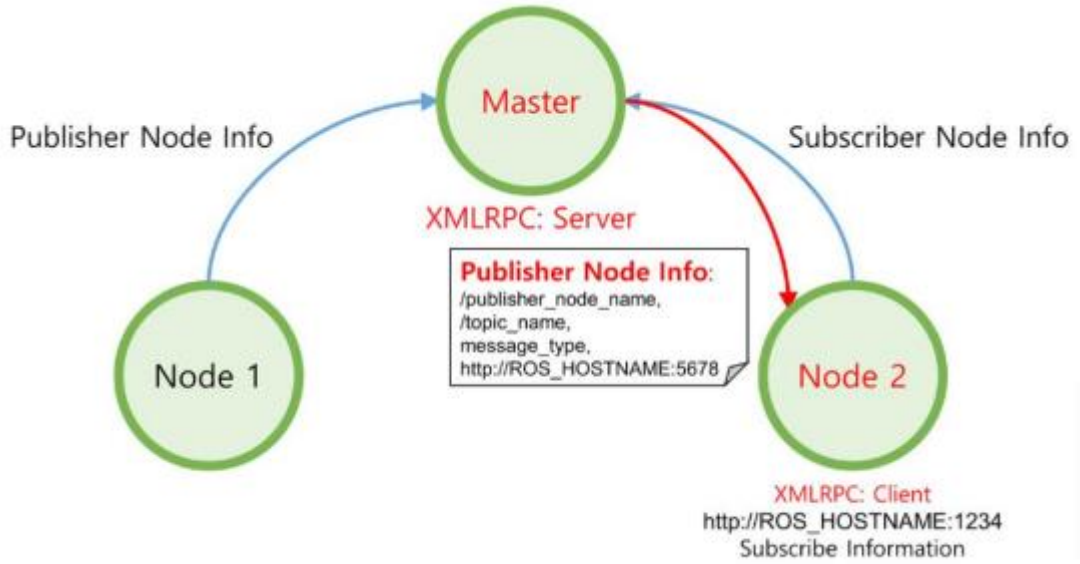

Рис.2.3. Мастер-узлы и XMLRPC

Когда узел запускается, он регистрирует информацию о себе в Мастере, такую как его имя, название темы, тип сообщения, URL-адрес и номер порта. Для общения с Мастером узлы используют протокол XMLRPC (XML Remote Procedure Call). Мастер выступает в качестве сервера XMLRPC, а узлы являются клиентами. XMLRPC - это протокол RPC (Remote Procedure Call), использующий XML для кодирования вызовов и протокол HTTP для передачи данных.

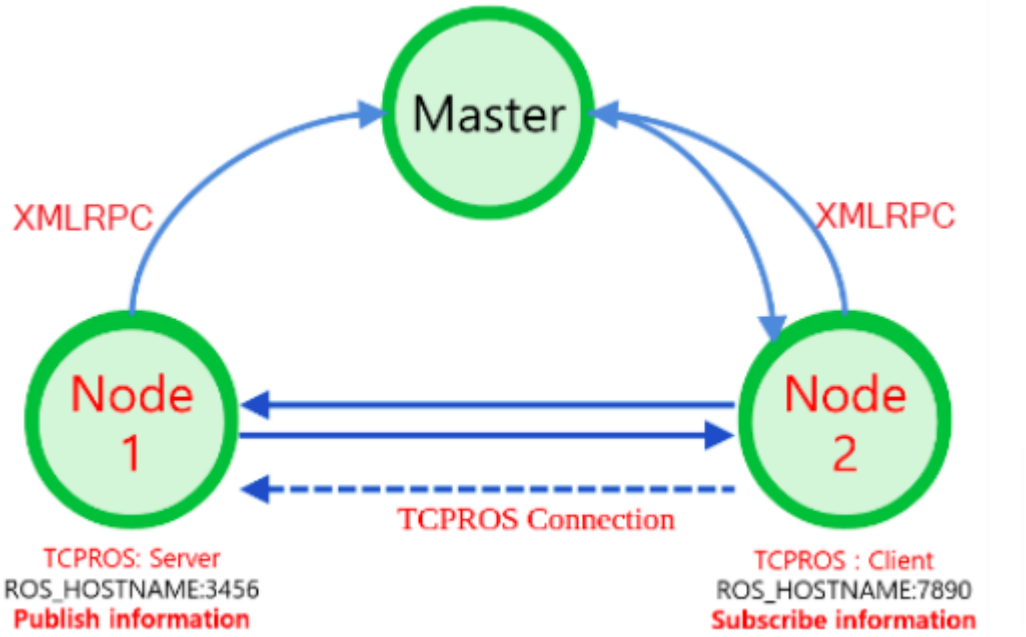

Рис.2.4. TCPROS

Для передачи сообщений между узлами ROS используется протокол TCPROS, который обеспечивает прямую связь между узлами, независимо от Мастера. Этот протокол является транспортным уровнем для сообщений ROS и использует стандартные сокеты TCP/IP для передачи данных. На компьютере, где выполняется узел, переменная ROS_HOSTNAME хранит URL-адрес узла, а порт имеет уникальное значение, которое задается произвольно.

Parameter server – это сервер, который хранит набор значений, используемых узлами во время выполнения программы, например, радиус колеса робота. С помощью команды "rosparam" можно изменять настройки узлов, хранящихся на сервере, например, настраивать максимальную и минимальную скорость робота.

Messages – это сообщения различных типов, которые могут содержать разнообразные данные (текст, изображения, положение робота и другие), а также иметь структуры, такие как массив сообщений. Общее количество типов сообщений составляет более 200.

Существует 3 метода обмена сообщениями (рисунок 6):

1. Тема (topic) обеспечивает однонаправленную связь (передачу / прием сообщений)
2. Сервис (service) обеспечивает двунаправленную связь (запрос (request) / ответ (response) на сообщение)
3. Действие (action) обеспечивает двунаправленную связь (цель сообщения (goal) / результат (result) / обратную связь (feedback)).

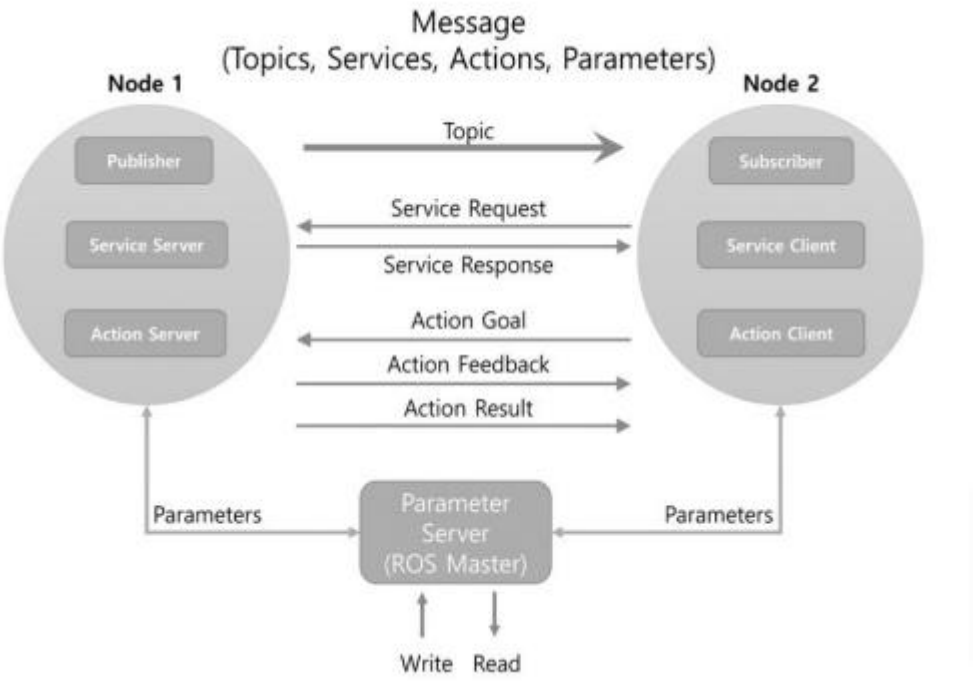

Рис.2.5. Методы обмена сообщениями между узлами

Рассмотрим эти методы подробнее.

1. Topic (тема, топик).

В системе ROS используется шина, называемая темой, для передачи данных между узлами, которые делятся на издателей и подписчиков. Первым шагом узел-издатель регистрирует свою тему в мастере, а затем начинает отправлять сообщения через нее. Узлы-подписчики получают информацию о теме узла-издателя от мастера и затем напрямую соединяются с ним, чтобы получать сообщения. На рисунках 2.4 и 2.5 показано, как узлы соединяются в системе ROS, используя мастер. Тема может использоваться для передачи потока сообщений от датчиков, таких как данные о местоположении робота или расстояние до препятствий. Узлы могут не только

публиковать сообщения в теме, но и подписываться на нее, чтобы получать информацию.

2. Service (Сервис).

ROS предназначен для обеспечения коммуникации между узлами через модель сервис. Эта модель включает в себя сервер, который отвечает на запросы, и клиента, который отправляет запросы, и работает путем обмена двумя сообщениями: запросом и ответом. В отличие от темы, сервис представляет собой однократный обмен сообщениями, и после завершения обмена связь между узлами прерывается.

3. Action (Действие).

Для задач, требующих продолжительного выполнения и обратной связи с клиентом, обычно используется модель коммуникации в режиме Действие (Action). В отличие от Сервиса, этот подход в основном используется для управления сложными задачами робота, такими как перемещение к указанной точке или запуск лазерного сканирования.

4. Rosbag и bag-файлы.

Важно отметить rosbag как функциональный инструмент, позволяющий сохранять данные, передаваемые через топики ROS, на специальные bag-файлы. Rosbag также способен воспроизводить bag-файлы с данными, которые будут публиковаться в те же топики, что и изначально. Это полезно для имитации данных от сенсоров, доступ к которым отсутствует.

2.4. Требования к моделированию

В работе использовался дистрибутив ОС Linux Ubuntu 20.04, установленный на виртуальной машине Oracle VM VirtualBox и дистрибутив ROS: Noetic, который установлен в соответствии с инструкцией на официальном сайте.

Для программирования был выбран язык Python, по причине его простоты, развитой системы API и библиотеки для ROS.

Для написания кода был выбран редактор VS Code, созданный Microsoft, так как он обладает большими возможностями, включая отладку программ, подсветку синтаксиса, интеллектуальное завершение кода и встроенный Git, что значительно упрощает процесс разработки.

2.5. Особенности работы манипулятора на подвижной платформе

Манипуляторы на подвижной платформе используются для выполнения различных задач, которые требуют мобильности и гибкости. Такие манипуляторы могут быть установлены на автомобили, роботы и другие подвижные платформы, что позволяет им обеспечивать доставку различных грузов и выполнение различных операций.

Одной из особенностей работы манипуляторов на подвижных платформах является необходимость обеспечения их устойчивости при перемещении по неровной поверхности. Это достигается за счет различных технических решений, таких как амортизаторы и системы балансировки.

Кроме того, манипуляторы на подвижных платформах должны быть достаточно гибкими, чтобы выполнять разнообразные задачи. Они должны сочетать в себе высокую точность и мощность, чтобы обеспечивать эффективную работу в условиях, где скорость и точность играют ключевую роль.

Для того чтобы манипулятор мог работать на подвижной платформе, его управление должно быть специально настроено. Различные моменты управления, такие как скорость перемещения, угол поворота и расстояние от груза, должны быть учтены по отношению к состоянию подвижной платформы. Это позволяет обеспечить эффективное взаимодействие манипулятора и платформы при выполнении задач.

Кроме того, манипулятор на подвижной платформе может быть оборудован дополнительными системами, такими как системы видеонаблюдения и управления приводами. Это позволяет обеспечить более точную и эффективную работу в различных условиях.

В целом, манипуляторы на подвижной платформе являются достаточно эффективным средством выполнения различных задач, которые требуют мобильности. Они способны обеспечивать высокую точность и мощность, а также гибкость и устойчивость при выполнении задач в различных условиях.

2.6. OpenMANIPULATOR на платформе TurtleBot3

Манипулятор OpenMANIPULATOR - это многозвенный манипулятор, разработанный компанией ROBOTIS для использования в роботах на платформе TurtleBot3. Он имеет длину около 40 см и состоит из 4 звеньев, каждое из которых может поворачиваться на 360 градусов. Это позволяет манипулятору перемещать объекты в трехмерном пространстве, изменять положение схвата и выполнять различные действия.

Рис.2.6. OpenManipulator на платформе TurtleBot3

Особенности работы манипулятора OpenMANIPULATOR на платформе TurtleBot3 включают гибкость, простоту управления и широкий спектр возможностей.

1. Гибкость.

Манипулятор OpenMANIPULATOR может быть использован для широкого спектра задач: от простых операций взятия и перемещения объектов до сложных манипуляций, таких как сборка и сварка. Это за счет гибкой конструкции и возможности изменять конфигурацию звеньев.

2. Простота управления.

Управление манипулятором OpenMANIPULATOR на платформе TurtleBot3 осуществляется через ROS (Robot Operating System). ROS - это фреймворк для разработки роботов, предоставляющий различные инструменты для управления и мониторинга роботов. ROS позволяет программировать манипулятор с помощью языка Python или C++.

3. Расширенные возможности.

Манипулятор OpenMANIPULATOR на платформе TurtleBot3 имеет расширенные возможности, такие как распознавание объектов и автоматическая навигация. С помощью камеры и датчиков на платформе TurtleBot3 манипулятор может распознавать объекты, определять их положение и выполнять соответствующие манипуляции.

Исходя из вышеперечисленного, было решено, что данная модель роботизированного манипулятора наиболее других подходит для выполнения данного проекта.

3. МОДЕЛИРОВАНИЕ В СРЕДЕ GAZEBO

3.1. Turtlebot3

После установки выбранной версии ROS, необходимо установить создать рабочую директорию, куда следует клонировать репозитории для работы с Turtlebot3. На рисунке №3.1 запущена стандартная сцена мира с Turtlebot3 модели burger в среде Gazebo.

Запускается данная сцена с помощью команд:

```
export TURTLEBOT3_MODEL=burger
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

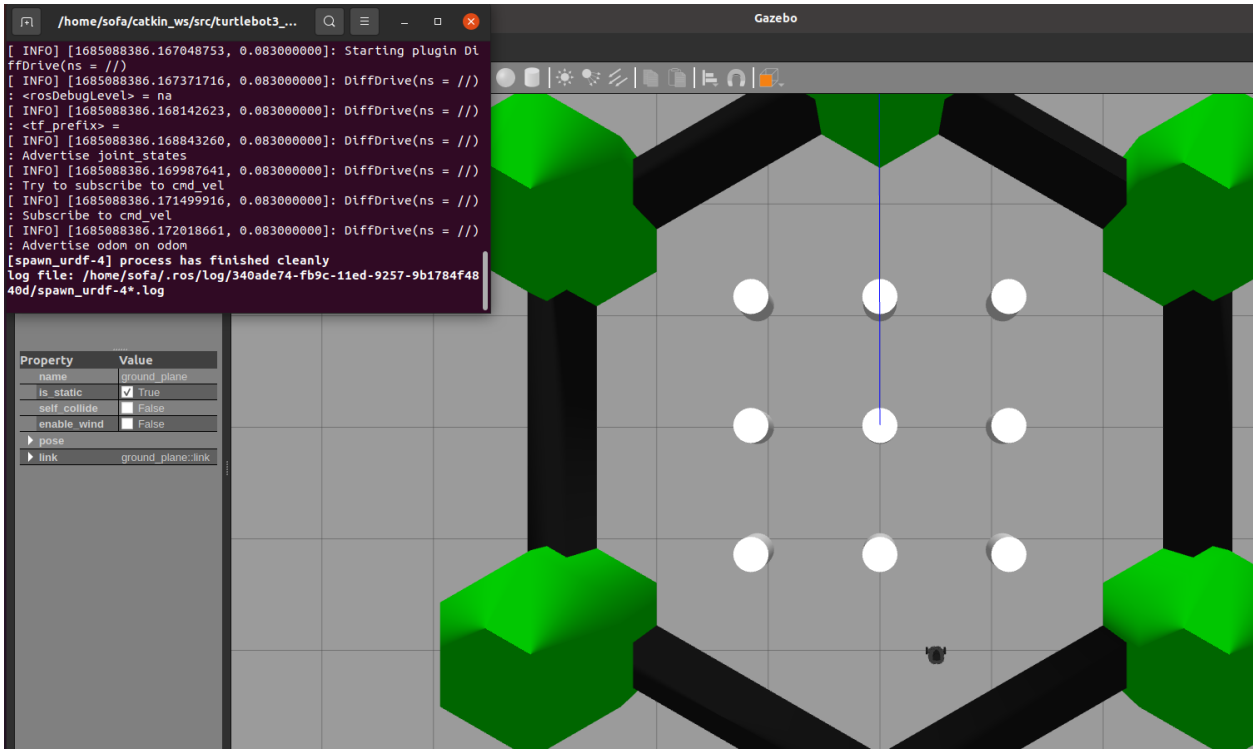

Рис.3.1. turtlebot3_world.launch

Далее проверяем файл для управления роботом с клавиатуры с помощью команд:

```
export TURTLEBOT3_MODEL=burger
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

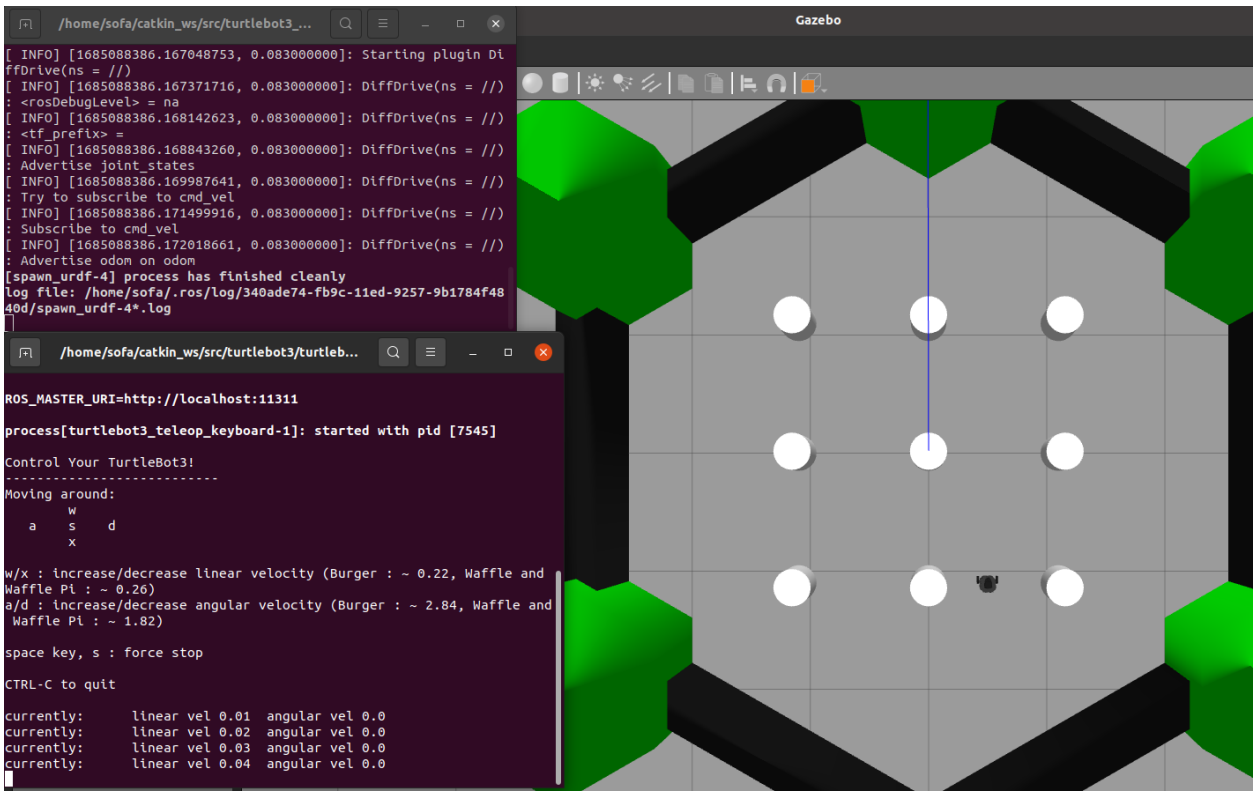

Рис.3.2. Управление с клавиатуры

С помощью пакета Slam Toolbox получаем информацию с лазерных сканеров для считывания информации об окружении и на ее основе создаем 2D-карту пространства. Используем команды:

```
export TURTLEBOT3_MODEL=burger
roslaunch turtlebot3_slam turtlebot3_slam.launch
slam_methods:=gmapping
```

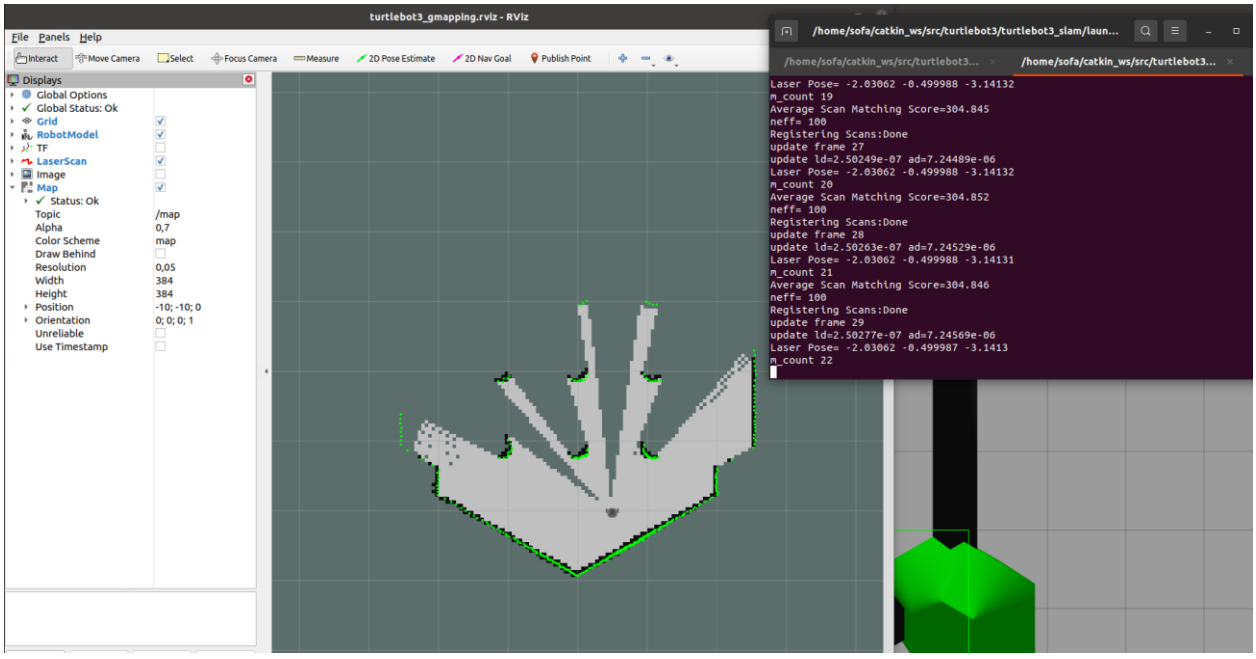

Рис.3.3. turtlebot3_slam.launch

Далее, управляя роботом с клавиатуры, исследуем окружение для составление карты.

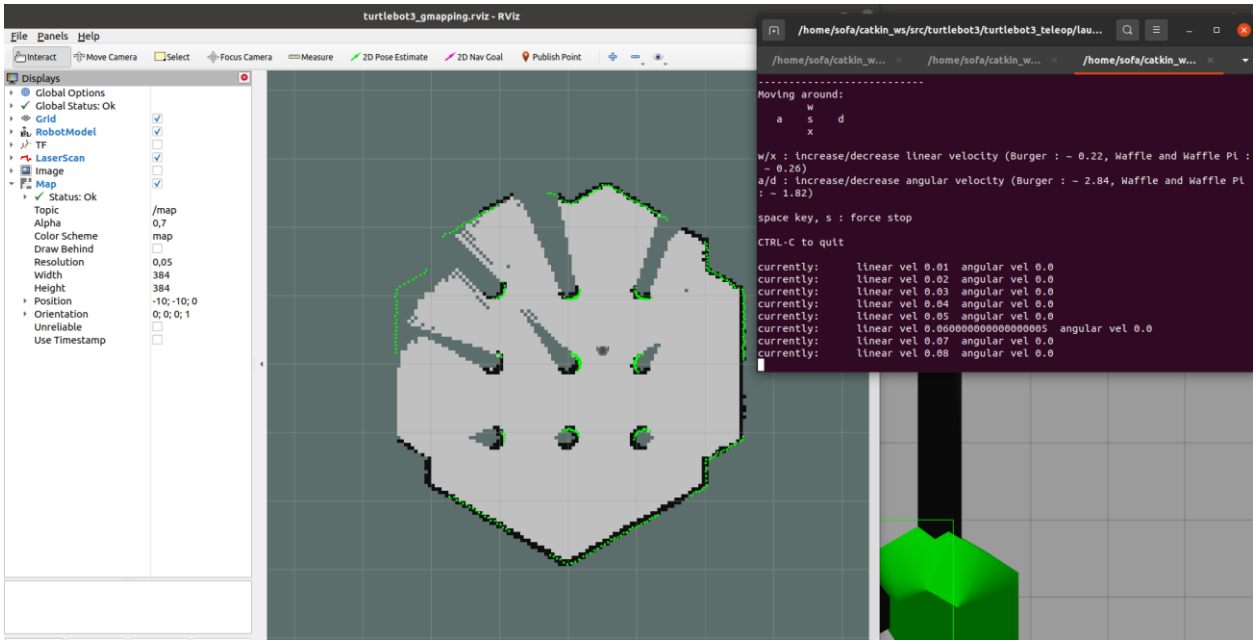

Рис.3.4. Сканирование окружения

После полного сканирования создаем 2D-карту пространства:

```
rosrun map_server map_saver -f ~/map
```

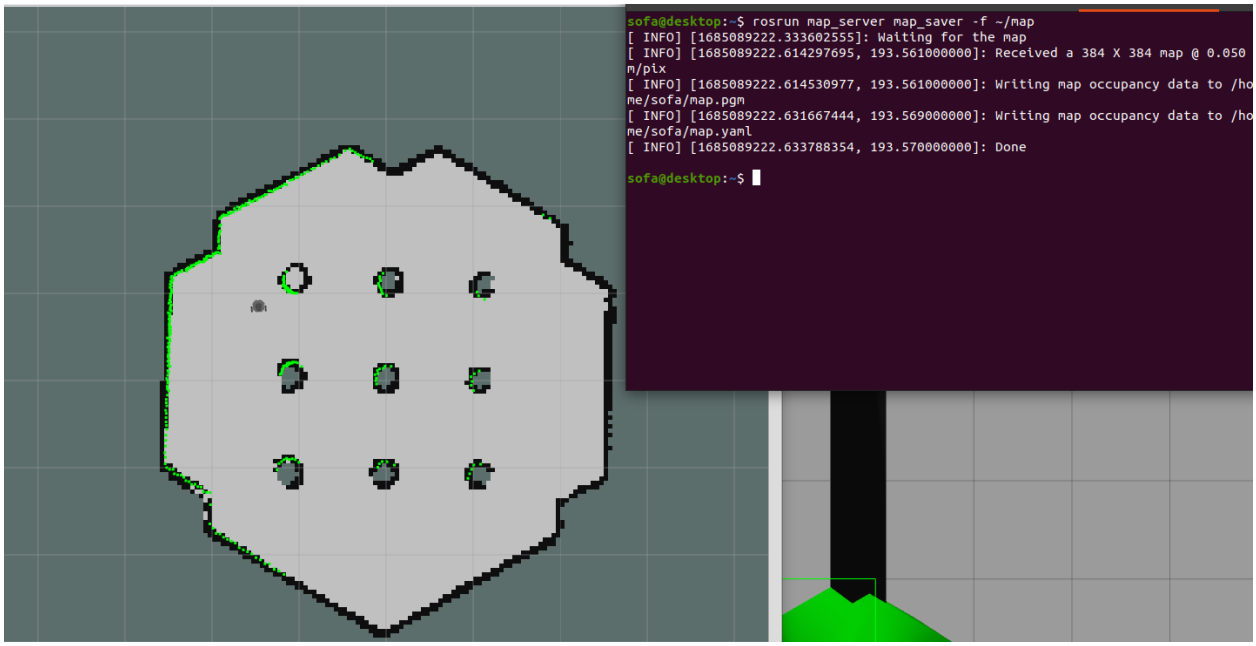

Рис.3.5. Сохранение карты

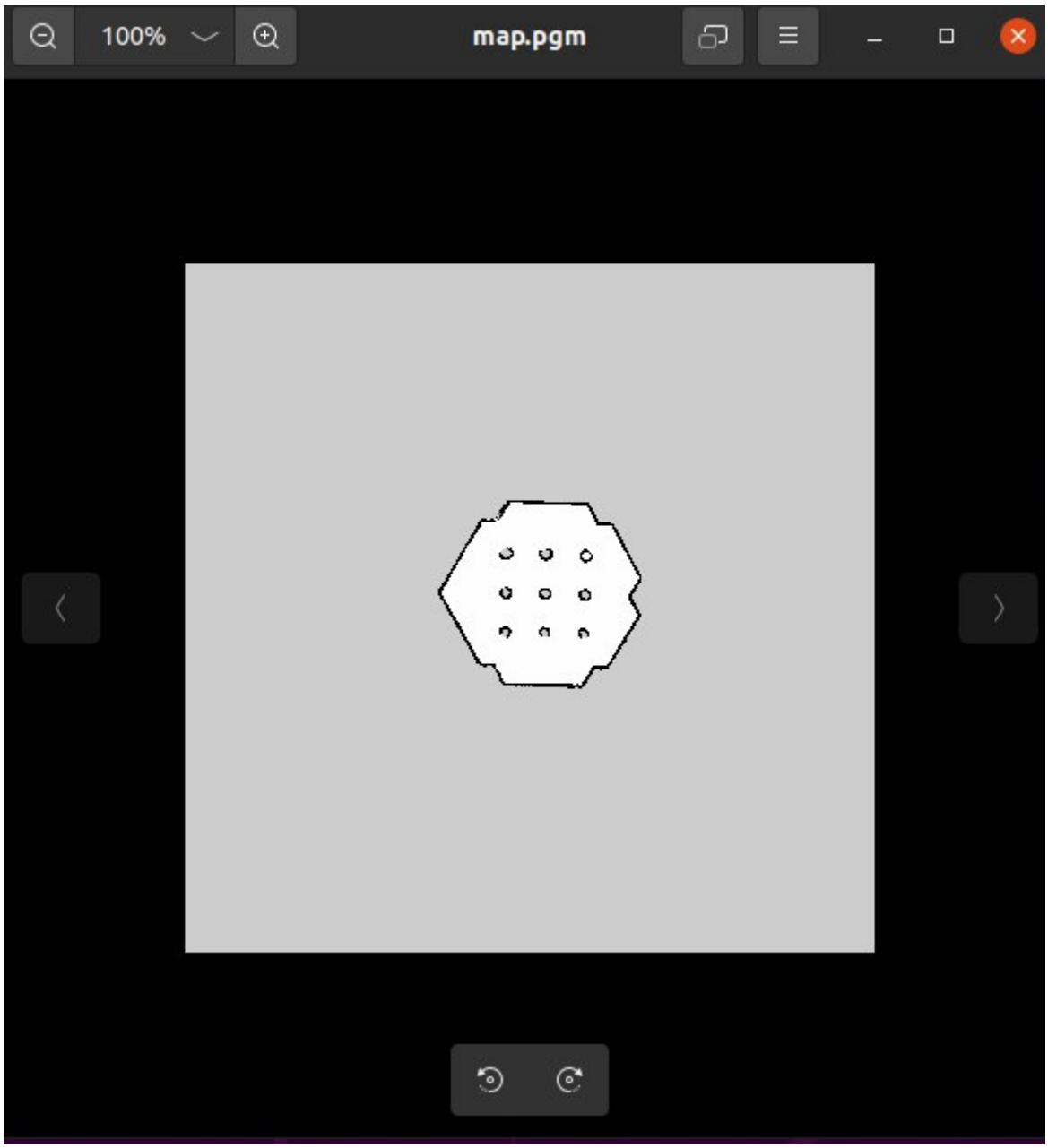

Рис.3.6. 2D-карта пространства

С помощью созданной карты можно осуществить навигацию роботом в пространстве. Запускаем сцену с turtlebot3 в gazebo и запускаем navigation с подключением ранее созданной карты.

```
export TURTLEBOT3_MODEL=burger
roslaunch turtlebot3_navigation turtlebot3_navigation.launch
map_file:=$HOME/map.yaml
```

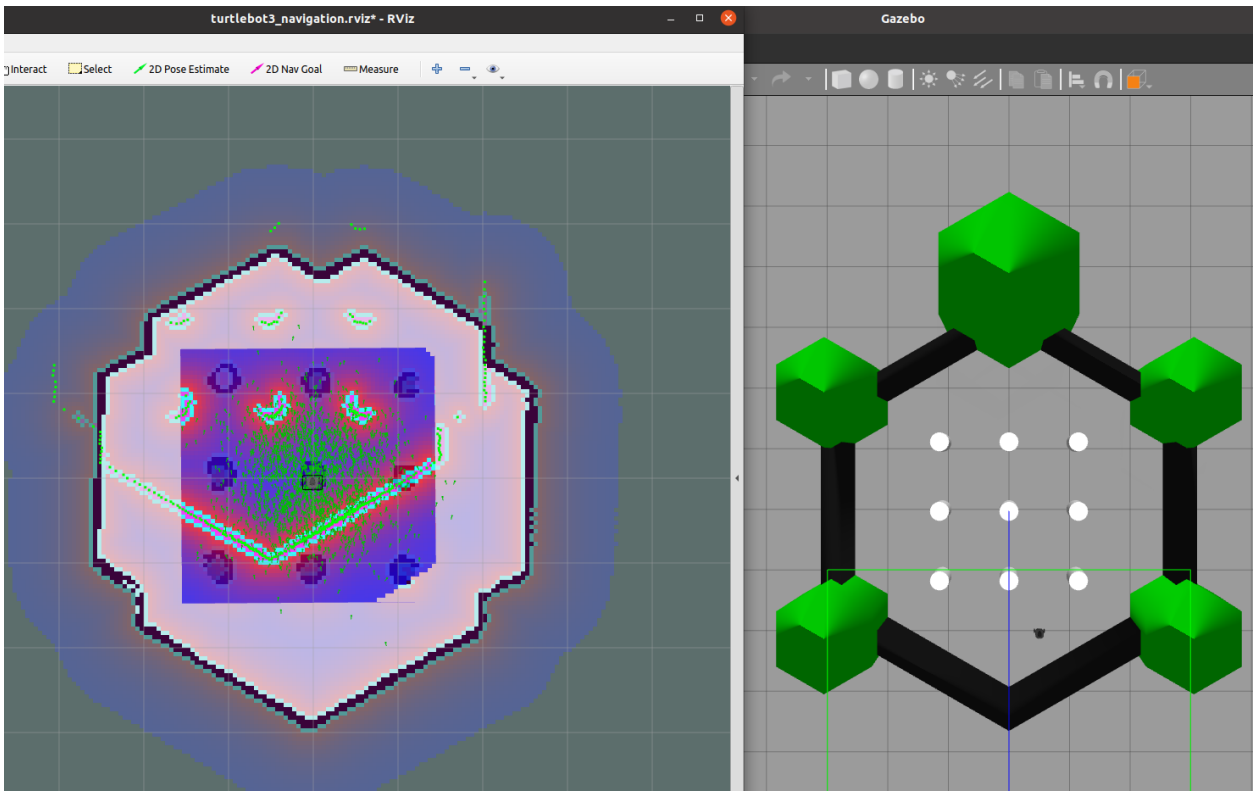

Рис.3.7. navigation

Далее задаем направление движения робота:

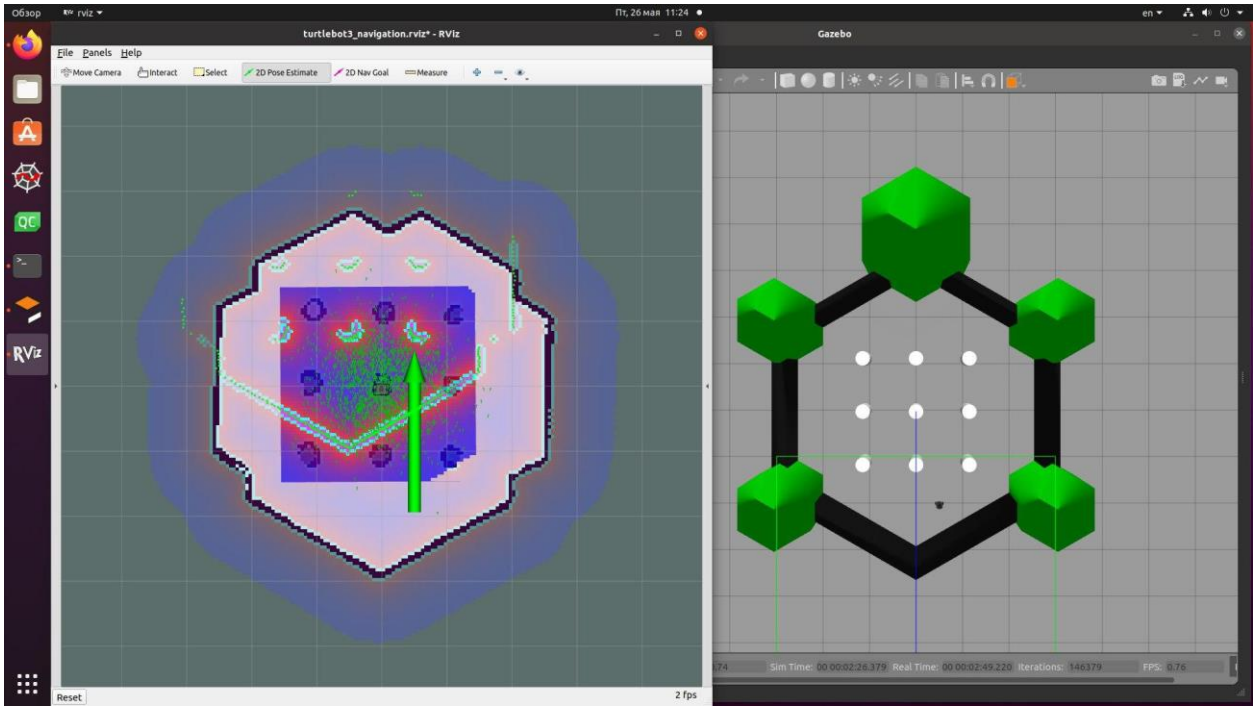

Рис.3.8. Определение направления движения

После чего указываем координаты перемещения робота, по которым он и будет следовать.

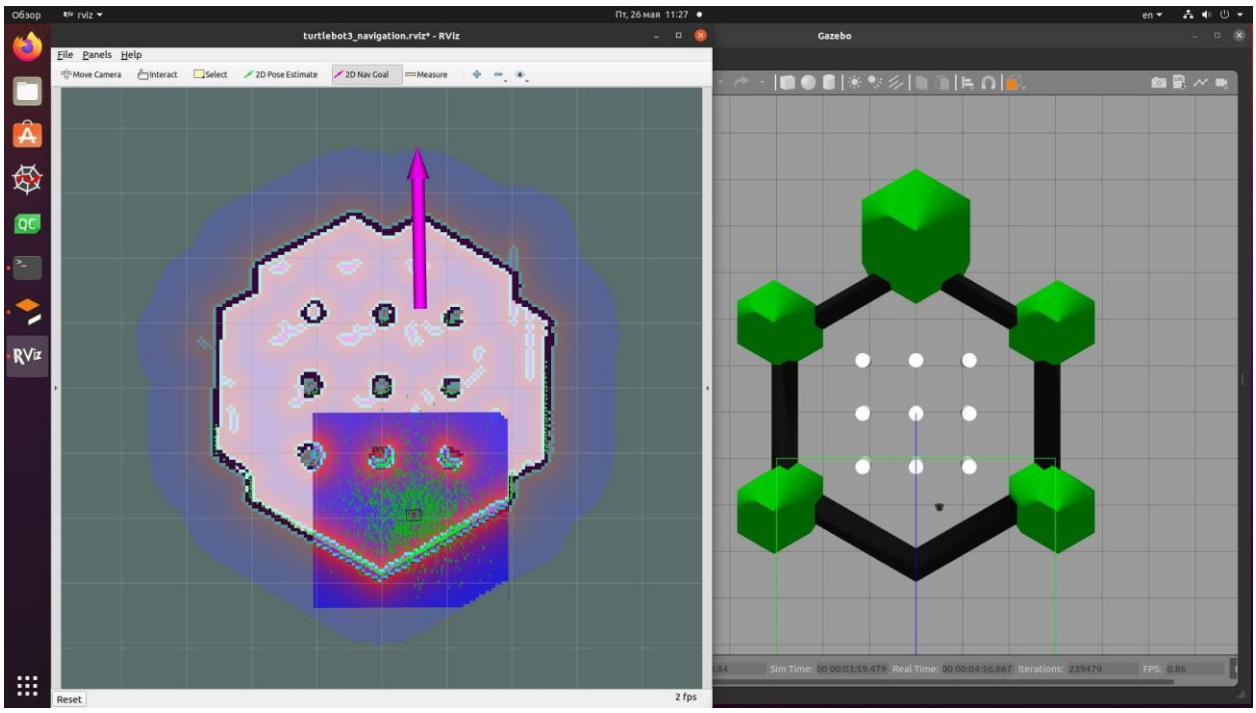

Рис.3.9. Указание координат

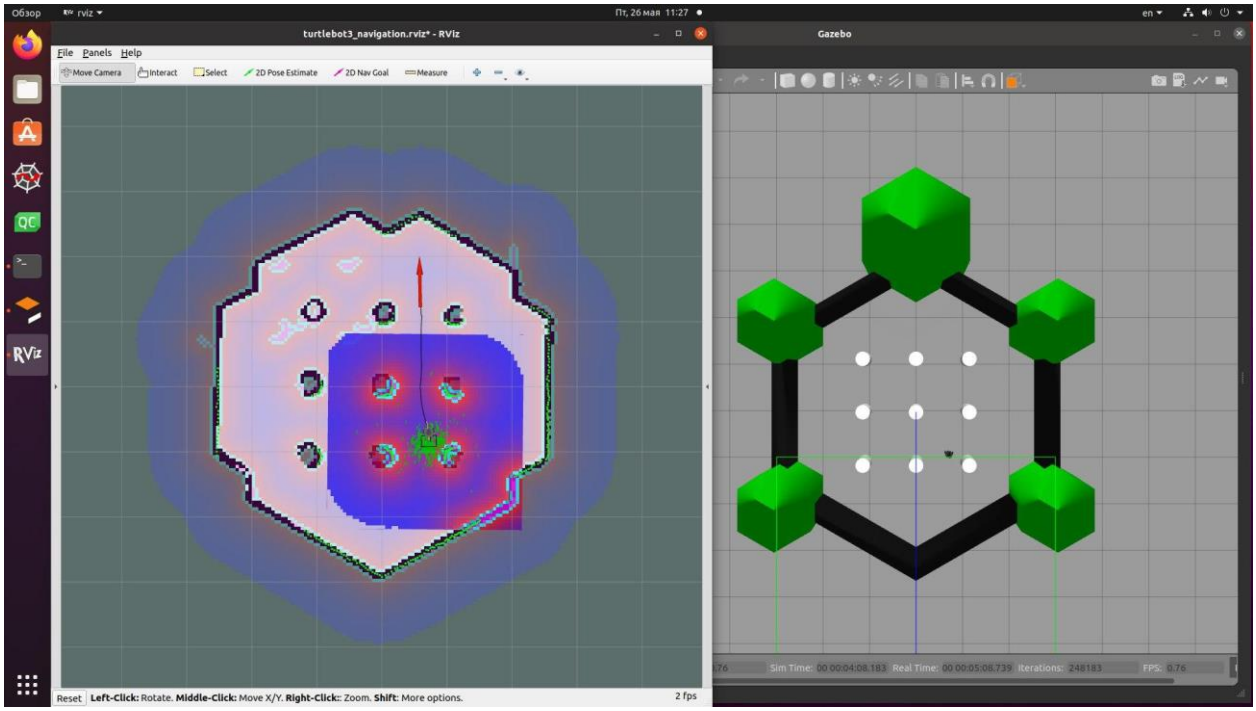

Рис.3.10. Процесс перемещения робота по координатам

3.2. Open Manipulator

Для запуска и работы с Open Manipulator необходимо установить соответствующие пакеты:

git clone -b noetic-devel https://github.com/ROBOTIS-GIT/open_manipulator.git
git clone -b noetic-devel https://github.com/ROBOTIS-GIT/open_manipulator_msgs.git
git clone -b noetic-devel https://github.com/ROBOTIS-GIT/open_manipulator_simulations.git

Запускаем сцену с манипулятором командой:

roslaunch open_manipulator_gazebo open_manipulator_gazebo.launch

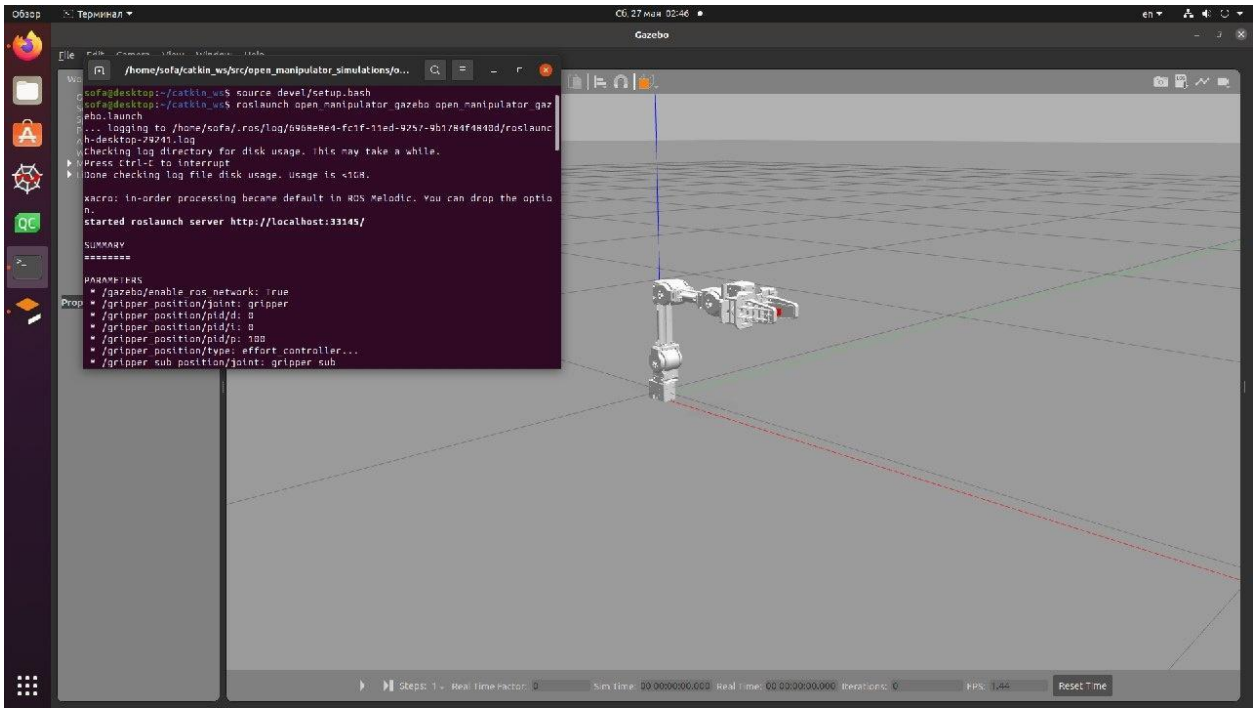

Рис.3.11. Запуск сцены с манипулятором

Одним из способов управления манипулятором является ручное управление, которое запускается командами:

roslaunch open_manipulator_controller open_manipulator_controller.launch use_platform:=false – controller for Gazebo

roslaunch open_manipulator_control_gui open_manipulator_control_gui.launch – operation in Gazebo

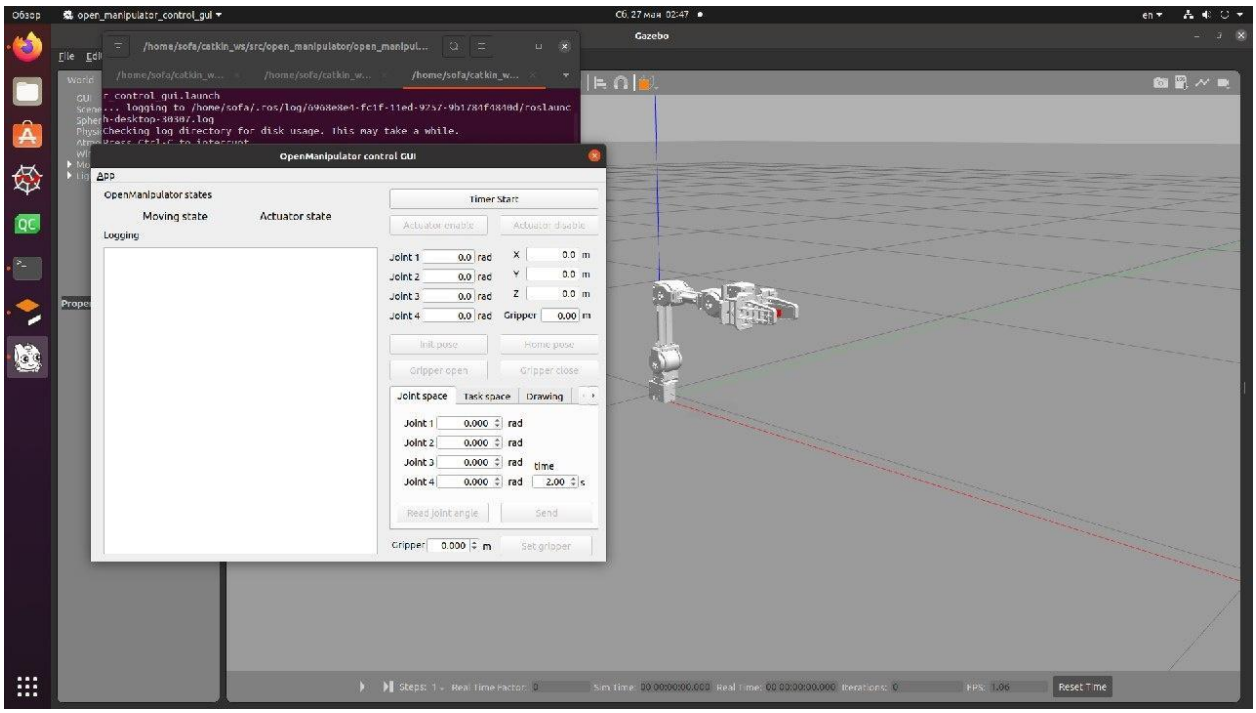

Рис.3.12. Запуск системы управления манипулятором

Далее запускаем сцену в Gazebo и запускаем таймер в управлении. После чего можно управлять манипулятором.

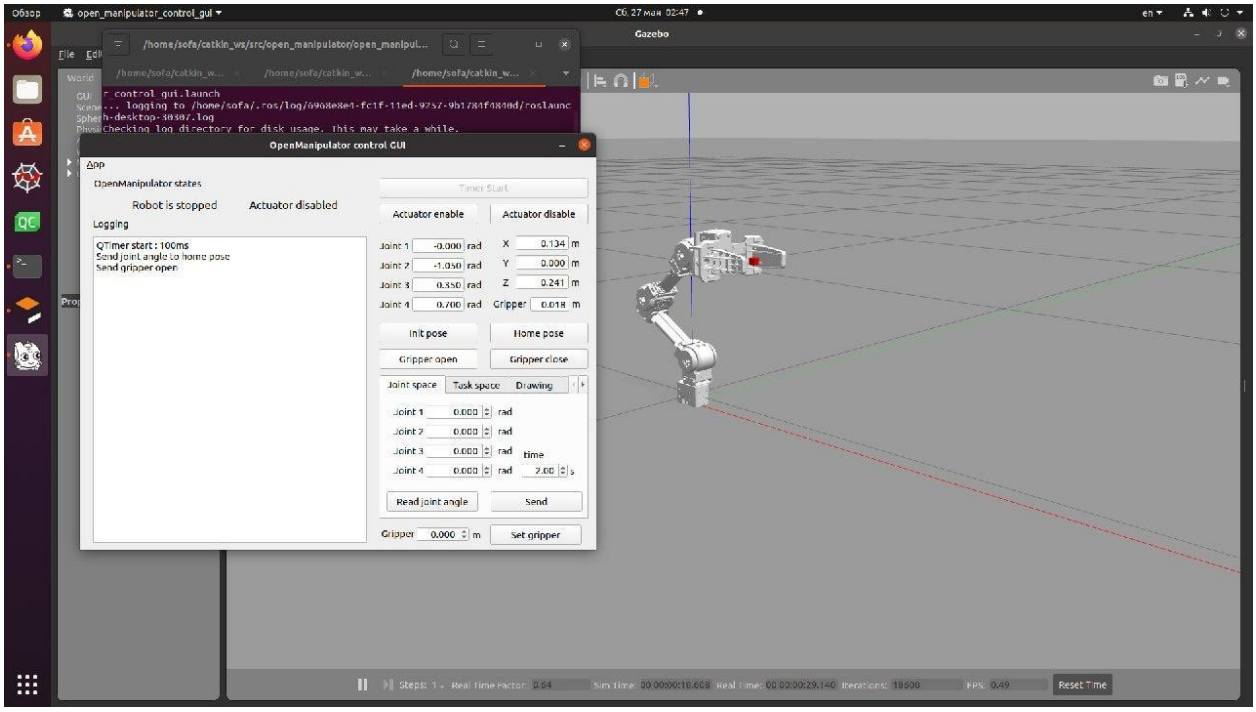

Рис.3.13. Управление манипулятором

3.3. Машинное обучение

Когда turtlebot3 совершает действие в определенном состоянии, он получает вознаграждение. Вознаграждение может быть положительным или отрицательным. Когда turtlebot3 добирается до цели, он получает большое положительное вознаграждение. Когда turtlebot3 сталкивается с препятствием, он получает большое отрицательное вознаграждение.

Запускаем сцену:

roslaunch turtlebot3_gazebo turtlebot3_stage_1.launch

И запускаем алгоритм и считывание состояний наград:

roslaunch turtlebot3_dqn result_graph.launch

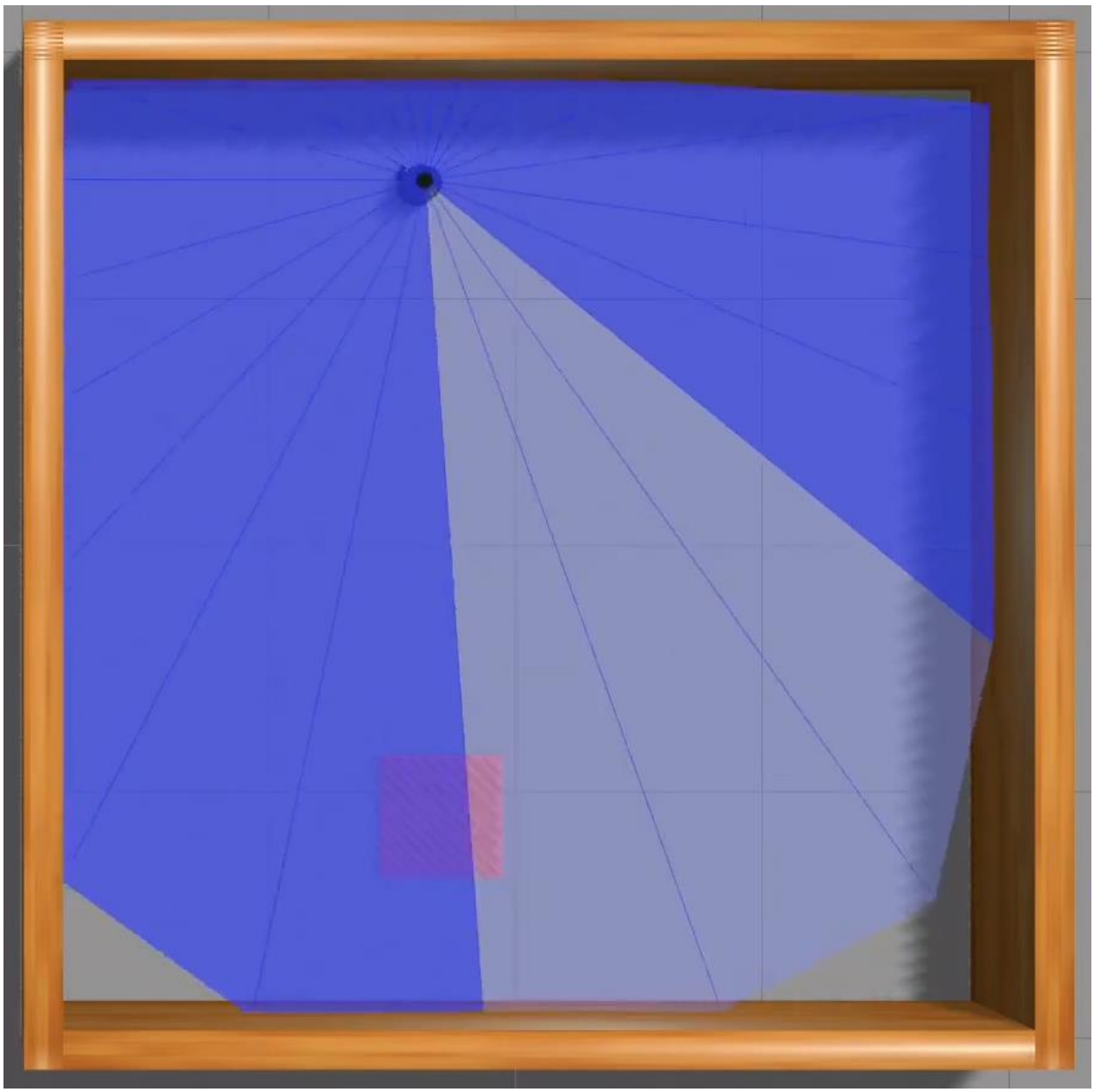

Рис.3.14. Сцена

Считывание награды в первых этапах обучения:

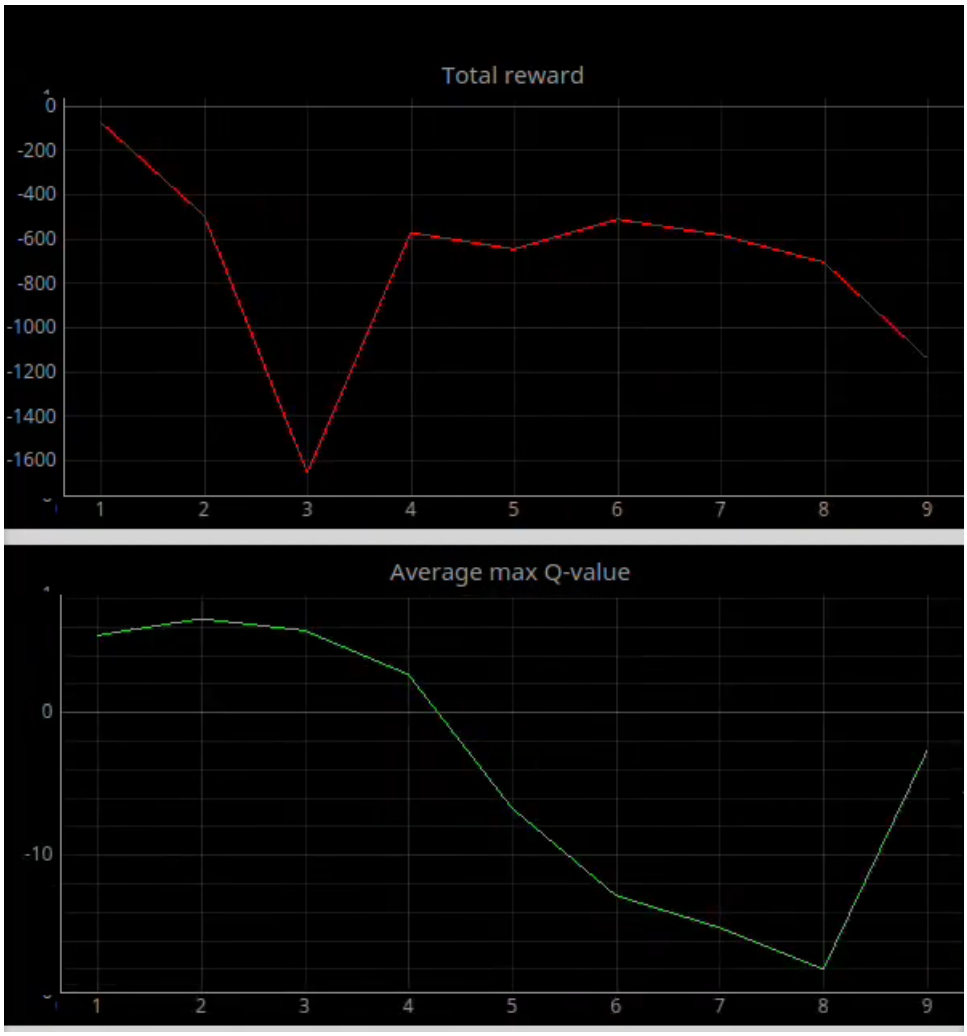

Рис.3.15. Графики вознаграждения

```
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use AVX instruct
ions, but these are available on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use AVX2 instruc
tions, but these are available on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use FMA instruct
ions, but these are available on your machine and could speed up CPU computations.
[INFO] [1532515157.980269, 0.678000]: Goal position : 0.6, 0.0
[INFO] [1532515162.313211, 4.831000]: Goal!!
[INFO] [1532515163.218013, 5.705000]: Goal position : -1.1, 0.3
[INFO] [1532515172.198353, 14.510000]: Collision!!
[INFO] [1532515172.290801, 14.598000]: Ep: 1 score: -70.89 memory: 136 epsilon: 1.00 time: 0:00:15
[INFO] [1532515187.297705, 14.704000]: Collision!!
[INFO] [1532515187.400697, 14.804000]: Ep: 2 score: -492.20 memory: 227 epsilon: 0.99 time: 0:00:30
[INFO] [1532515205.978065, 18.167000]: Collision!!
[INFO] [1532515206.152766, 18.328000]: Ep: 3 score: -1652.21 memory: 340 epsilon: 0.98 time: 0:00:48
[INFO] [1532515222.513555, 15.994000]: Collision!!
[INFO] [1532515222.650921, 16.125000]: Ep: 4 score: -569.25 memory: 443 epsilon: 0.97 time: 0:01:05
[INFO] [1532515237.916849, 14.895000]: Collision!!
[INFO] [1532515238.072937, 15.046000]: Ep: 5 score: -643.31 memory: 540 epsilon: 0.96 time: 0:01:20
```

Рис.3.16. Вывод информации о перемещении

Считывание награды спустя ~200 попыток:

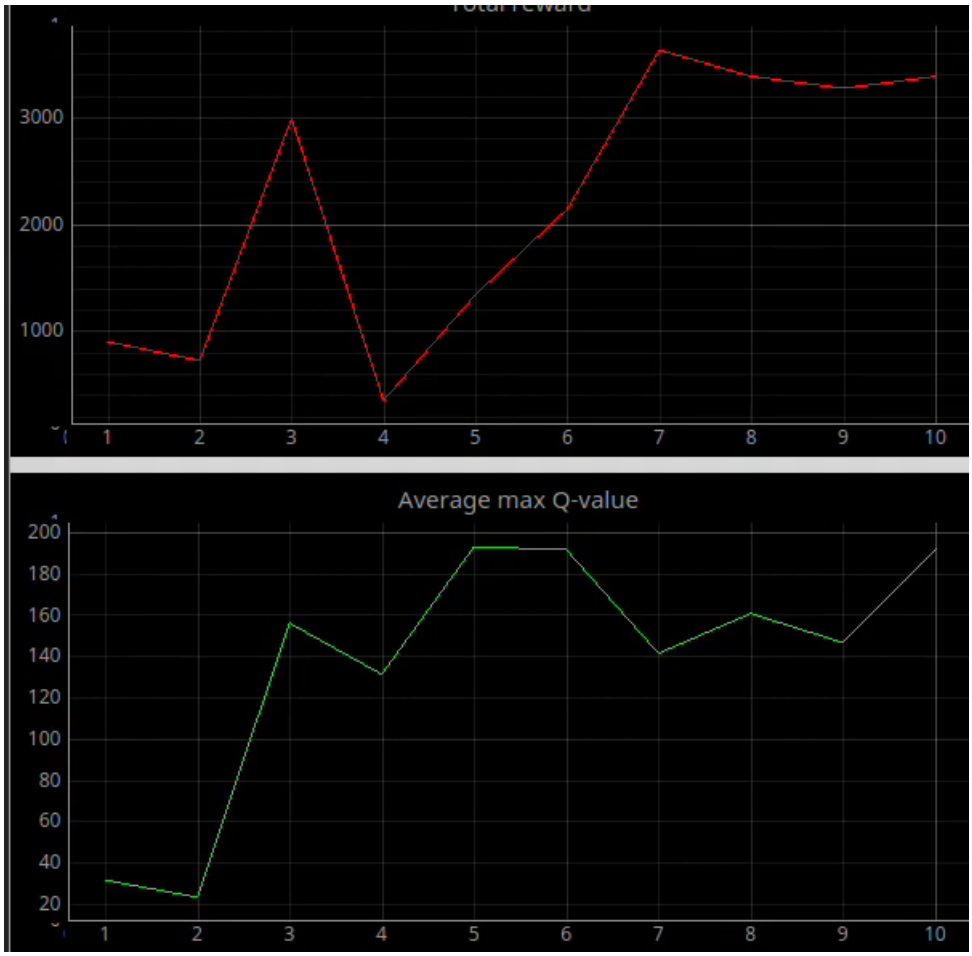

Рис.3.17. Графики вознаграждения

```
[INFO] [1532429018.979696, 30.562000]: Goal!!
[INFO] [1532429019.937945, 31.462000]: Goal position : 0.0, 1.2
[INFO] [1532429042.243996, 52.461000]: Goal!!
[INFO] [1532429043.352533, 53.421000]: Goal position : -0.4, -0.4
[INFO] [1532429064.560084, 73.503000]: Goal!!
[INFO] [1532429065.466281, 74.343000]: Goal position : 0.6, 0.0
[INFO] [1532429080.640756, 88.788000]: Goal!!
[INFO] [1532429081.563289, 89.645000]: Goal position : 0.5, -0.4
[INFO] [1532429084.538134, 92.488000]: Goal!!
[INFO] [1532429085.496203, 93.311000]: Goal position : -0.5, -1.0
[INFO] [1532429093.905474, 101.126000]: Goal!!
[INFO] [1532429094.755998, 101.934000]: Goal position : -0.8, 1.1
Time out!!
[INFO] [1532429104.603220, 111.591000]: Ep: 190 score: 3385.42 memory: 4440 epsilon: 0.15 time: 0:15:22
[INFO] [1532429125.903548, 20.661000]: Goal!!
[INFO] [1532429126.892125, 21.601000]: Goal position : 0.3, -1.0
[INFO] [1532429148.593497, 42.683000]: Goal!!
[INFO] [1532429149.604660, 43.644000]: Goal position : 0.3, 0.3
[INFO] [1532429162.145003, 55.629000]: Goal!!
[INFO] [1532429163.075809, 56.522000]: Goal position : -1.2, 0.3
[INFO] [1532429179.825859, 72.923000]: Goal!!
[INFO] [1532429180.758103, 73.801000]: Goal position : -1.1, -1.2
```

Рис.3.18. Вывод информации о перемещении

Заключение

В процессе выполнения выпускной квалификационной работы был произведен анализ и обзор фреймворка робототехники ROS. Были изучены такие элементы, как файловая система Linux, язык программирования Python, принципы взаимодействия узлов в системе ROS, способы отправки команд через терминал, а также инструменты визуализации RViz и 3D-симуляции Gazebo.

Были изучены основные принципы взаимодействия узлов в системе ROS с помощью команд rosbash. Описаны этапы создания программ в ROS и ключевые моменты использования инструментов и возможностей платформы. Результатом является программа для перемещения виртуального мобильного робота Open Manipulator в заданную точку на плоскости и предложенные решения для использования мобильного робота TurtleBot3.

В результате изучения и анализа данного робототехнического фреймворка были сделаны следующие выводы:

Применение алгоритмов машинного обучения с подкреплением позволяет достигнуть значительного улучшения точности и скорости движения манипулятора на платформе в сравнении с ручным управлением.
Таким образом, данное исследование подтверждает потенциал применения алгоритмов машинного обучения с подкреплением в робототехнических системах для повышения эффективности и точности их работы.

Список использованных источников и литературы

1. Marwan Qaid Mohammed, L.C. Kwek, Shing Chyi Chua. Review of Deep Reinforcement Learning-Based Object Grasping: Techniques, Open Challenges, and Recommendations //
2. Саттон Р. С., Барто Э. Дж. Обучение с подкреплением: Введение. 2-е изд. / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2020. – 552 с. //
3. Лаура Г., Лун К.В. Глубокое обучение с подкреплением: теория и практика на языке Python. — СПб.: Питер, 2022. — 416 с.: ил. — (Серия «Библиотека программиста»). //
4. «ROS Robot Programming» YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, TaeHoon Lim: ROBOTIS Co., 2017 - 487 c
5. ROS википедия [электронный ресурс] — http://wiki.ros.org/ //
6. Томас Бройнль. Встраиваемые робототехнические системы: проектирование и применение мобильных роботов со встроенными системами управления / Томас Бройнль: Институт компьютерных исследований, 2012. - 520 с. //
7. O'Kane, Jason. A Gentle Introduction to ROS / O'Kane, Jason: CreateSpace, 2013 – 166 с. //

Приложение А

Установка пакета turtlebot3

Перейдём в директорию, в которую будет загружен данный пакет:

cd ~/catkin_ws/src

Загрузим пакет turtlebot3 с официального репозитория с помощью команды git:

git clone -b kinetic-devel https://github.com/ros/urdf.git

git clone https://github.com/ROBOTIS-GIT/turtlebot3

Введём следующую команды для сборки пакета в рабочем пространстве catkin:

cd ~/catkin_ws$ catkin_make

Перед тем как запустить симулятор gazebo, нужно будет добавить специальную строку в .bashrc.

.bashrc - это сценарий оболочки bash, который находится в домашнем каталоге пользователя. Он используется для сохранения и загрузки настроек терминала и переменных среды.

Для этого введем следующую команду в терминале:

gedit ~/.bashrc и добавим следующую строку в конец этого файла:

export TURTLEBOT3_MODEL=burger

Сохраним конфигурацию файла bashrc в текущем терминале:

source ~/.bashrc

Приложение Б

Код узла result_graph

```
import rospy
import pyqtgraph as pg
import sys
import pickle
from std_msgs.msg import Float32MultiArray, Float32
from PyQt5.QtGui import *
from PyQt5.QtCore import *

class Window(QMainWindow):
    def __init__(self):
        super(Window, self).__init__()
        self.setWindowTitle("Result")
        self.setGeometry(50, 50, 600, 650)
        self.graph_sub = rospy.Subscriber('result',
Float32MultiArray, self.data)
        self.ep = []
        self.data = []
        self.rewards = []
        self.x = []
        self.count = 1
        self.size_ep = 0
        load_data = False

        if load_data:
            self.ep, self.data = self.load_data()
            self.size_ep = len(self.ep)
        self.plot()

    def data(self, data):
        self.data.append(data.data[0])
        self.ep.append(self.size_ep + self.count)
        self.count += 1
        self.rewards.append(data.data[1])


    def plot(self):
```

```
        self.qValuePlt = pg.PlotWidget(self, title="Average max
Q-value")
        self.qValuePlt.move(0, 320)
        self.qValuePlt.resize(600, 300)
        self.timer1 = pg.QtCore.QTimer()
        self.timer1.timeout.connect(self.update)
        self.timer1.start(200)

        self.rewardsPlt = pg.PlotWidget(self, title="Total
reward")
        self.rewardsPlt.move(0, 10)
        self.rewardsPlt.resize(600, 300)

        self.timer2 = pg.QtCore.QTimer()
        self.timer2.timeout.connect(self.update)
        self.timer2.start(100)

        self.show()

    def update(self):
        self.rewardsPlt.showGrid(x=True, y=True)
        self.qValuePlt.showGrid(x=True, y=True)
        self.rewardsPlt.plot(self.ep, self.data, pen=(255, 0, 0))
        self.save_data([self.ep, self.data])
        self.qValuePlt.plot(self.ep, self.rewards, pen=(0, 255,
0))

    def load_data(self):
        try:
            with open("graph.txt") as f:
                x, y = pickle.load(f)
        except:
            x, y = [], []
        return x, y

    def save_data(self, data):
        with open("graph.txt", "wb") as f:
            pickle.dump(data, f)


def run():
```

```
        rospy.init_node('graph')
        app = QApplication(sys.argv)
        GUI = Window()
        sys.exit(app.exec_())

run()
```