Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Рудя Вячеслава Васильовича*
(ПІБ)

академічної групи *121М-19-1*
(шифр)

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

на тему: *Методи, алгоритми та програмне забезпечення для біонічного хапання з уникненням перешкод за допомогою дерев октантів та глибокого навчання*

_____ *В.В. Рудь*

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
розділів кваліфікаційної роботи				
спеціальний	Доц. Сироткіна О.І.			
економічний	Доц. Касьяненко Л.В.			
Рецензент				
Нормоконтролер	Доц. Сироткіна О.І.			

Дніпро
2020

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_____ І.М. Удовик
(підпис) (прізвище, ініціали)

« » _____ 20 20 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121М-19-1* *Рудю Вячеславу Васильовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Методи, алгоритми та програмне забезпечення для біонічного хапання з уникненням перешкод за допомогою дерев октантів та глибокого навчання*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 22.10.2020 р. № 888

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес оптимізації та покращення точності руху та уникнення перешкод для маніпулятора з пристроєм хапання.

Предмет досліджень – системи управління роботами-маніпуляторами.

Мета роботи – підвищення ефективності системи розпізнавання перешкод та планування шляху до точки хапання для робота-маніпулятора на основі дерева октантів використовуючи систему ROS.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна полягає у тому, що удосконалено методи хапання та планування руху для робота маніпулятора на основі дерева октантів та глибокого навчання.

Практична цінність результатів полягає у тому, що запропонована система розпізнавання об'єкта, планування руху та хапання дозволяє ефективно розпізнавати об'єкти, знаходити позу для хапання, планувати шлях з уникненням розпізнаних перешкод та забезпечувати надійне хапання.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз теми та постановка задачі	12.09.2020-30.09.2020
Побудова системи планування шляху для навігації мобільних роботів	01.10.2020-31.10.2020
Створення автоматизованої системи для вирішення задачі ідентифікаційної експертизи бензинів	01.11.2020-07.12.2020

Завдання видав _____ *Сироткіна О.І.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Рудь В.В.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *12.09.2020 р.*

Термін подання кваліфікаційної роботи до ЕК *10.12.2019*

РЕФЕРАТ

Пояснювальна записка: 100 стор., 36 рис., 4 додатка, 46 джерел.

Метою дослідження є підвищення ефективності процесу розпізнавання об’єктів, розпізнавання позицій для їх захоплення, розпізнання перешкод, руху до пози хапання з уникненням перешкод та надійного хапання об’єкту.

Об’єктом дослідження є процес захоплення роботизованої руки в складних умовах із використанням дерева октантів для планування шляху з уникненням перешкод.

Предметом дослідження є моделі та методи виявлення положення захоплення, сприйняття перешкод та планування шляху захоплення та успішного захоплення об’єкта.

Методи дослідження. Для вирішення цієї проблеми використовуються такі методи, як пошук шляхів, встановлення порогів, вейвлет-перетворення, вилучення особливостей, штучні нейронні мережі та машинне навчання.

Наукова новизна. полягає у тому, що удосконалено процесс розпізнавання об’єктів, розпізнавання позицій для їх захоплення, розпізнання перешкод, руху до пози хапання з уникненням перешкод та надійного хапання об’єкту.

Практичне значення результатів полягає у тому, що запропонована система розпізнавання об’єктів, розпізнавання позицій для їх захоплення, розпізнання перешкод, руху до пози хапання з уникненням перешкод та надійного хапання об’єкту ефективно виконує свої задачі.

У розділі «Економіка» проведено розрахунки трудомісткості розробки програмного забезпечення, витрат на створення ПО і тривалості його розробки.

Список ключових слів: робот, маніпулятор, ROS, MoveIt, YOLO, Fin-Ray ефект, хапання, розпізнання об’єктів, уникнення перешкод, дерево октантів, хмара точок, 3D-друк.

ABSTRACT

Explanatory note: 100 pages, 36 Figures, 4 applications, 46 sources.

Purpose of the master's thesis. Increase the efficiency of the process of recognizing objects, recognizing positions to capture them, recognizing obstacles, moving to the grasping position with the obstacle avoidance, and secure grasping of the object.

The research object is a process of robotic arm grasping in a complex environment using octrees for path planning to avoid obstacles.

The research subject is the models and methods of detecting grasping position, obstacle perception, and path planning to grasp and successfully grasp the object.

Research methods. There are methods used to solve this problem, such as pathfinding, thresholding, wavelet transform, feature extraction, artificial neural networks, and machine learning.

Originality of research is determined by improved approach of object recognition, grasp pose detection, obstacle recognition, a movement to the grasping position with the avoidance of obstacles, and secure grasping.

Practical value is that the proposed system of object recognition, grasp pose detection, obstacle recognition, a movement to the grasping position with obstacle avoidance, and secure grasping of the object effectively performs its tasks.

In the Economics section, the complexity of software development, software development costs, and the actual development duration are calculated.

Keywords: robot, manipulator, ROS, MoveIt, YOLO, Fin-Ray effect, grasping, object recognition, obstacle avoidance, octant tree, point cloud, 3D printing.

CONTENTS

LIST OF ABBREVIATIONS

FK – Forward Kinematics

IK – Inversive Kinematics

ROS – Robot Operating System

CAD – Computer-Aided Design

API – Application Programming Interface

YOLO – You Only Look Once

R-CNN – Region-Based Convolutional Neural Network

TOF – Time of Flight

LSTM – Long Short-Term Memory

DOF – Degree of Freedom

INTRODUCTION

The relevance of research. Controlling a robot arm is one of the essential parts of robotic software development. It consists of next problems: object recognition, grasp pose selection, obstacle detection, motion planning. This study considers problems of integration of independent arm control systems. The areas of manipulator arm control are detecting objects and obstacles, detecting objects to grasp, determining objects to grasp, finding good poses of the gripper to grasp the object, motion planning to the grasp pose using detected obstacles, and slip detection or detection of secure grasping.

This issue is a modern problem primarily in the fields of industry, general-purpose robots, and experimental robots.

There are existing researches and developments for the mentioned problems. But combining all existing solutions and approaches might be a more complex task. The final solution should combine the most appropriate solution in each area and bring them together as a complete system.

The purpose of the research is an improvement of current approaches for robotic arm grasping. There should be a complete method for controlling the robot's arm. The method should include objects to grasp detection, obstacle detection, poses of the gripper for grasping, motion planning to chosen pose, slip detection of the chosen object in the gripper.

The object of research is a process of robotic arm grasping in a complex environment using octrees for path planning to avoid obstacles.

The subject of research is the models and methods of detecting grasping position, obstacle perception, and path planning to grasp and successfully grasp the object.

Methods of research. There are methods used to solve this problem, such as pathfinding, thresholding, wavelet transform, feature extraction, artificial neural networks, and machine learning.

The originality of the research is determined by improved object recognition approach, grasp pose detection, obstacle recognition, a movement to the grasping position with the avoidance of obstacles, and secure grasping.

This study considers current publications that address these issues. Existing algorithms and approaches have been found in managing both parts of the robot manipulator and solutions that combine several areas or integrate several existing approaches. There is a brief review of current literature and publications on the above algorithms and approaches. The advantages and disadvantages of the considered methods and approaches are determined. There are solutions that cover either some areas or only one of them, which does not meet the problem's requirements.

The researched problem is relevant in robotic arm development. It was found that some studies and literature mention the stated problem. But almost all of them are concentrated on independent solutions, while the problem is the integration of small target solutions in a complete system.

The practical value is that the proposed object recognition system, grasp pose detection, obstacle recognition, movement to the grasping position with obstacle avoidance, and secure grasping of the object effectively performs its tasks.

The personal contribution of the author:

1. Scientific results of the work are obtained by the author independently.
2. Choice of research methods and implementation technologies.
3. Development of the theoretical part of the work, which explores and systematizes knowledge of existing approaches of object detection, grasp pose detection, motion planning, and slip detection.
4. Development of the new system for object detection, grasp pose detection, motion planning, and slip detection.
5. Testing and evaluation of the results.

Structure and scope of the work. The work consists of an introduction, four sections, and conclusions. It contains 100 pages, including 71 pages of text of the main part with 36 figures, a list of used sources with 46 items on 4 pages, 4 appendices on 11 pages.

SECTION 1
ANALYSIS OF ROBOTIC ARM CONTROL METHODS

1.1. Robotic grasping

One of the most significant areas of interaction between robot and environment is the manipulation of objects. Manipulation can be a part of many highly useful processes: assembling, sorting, moving (Fig. 1.1.), ordering, cargo loading, etc. These processes are used in industrial and casual robots.

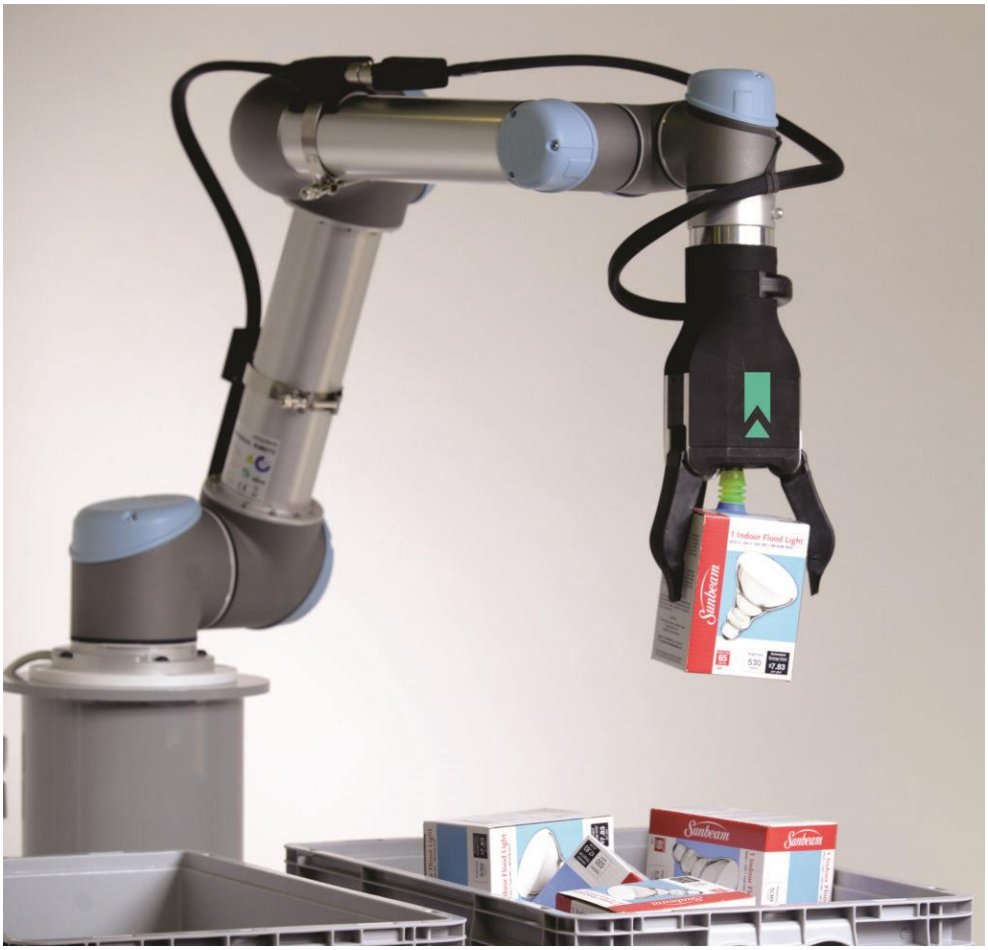


Fig. 1.1 End-effector with a three-finger gripper

Robotic manipulators should have an end effector, also known as End of Arm Tooling. Robot end-effectors are subdivided into next types [1]:

– Impactive: physically grasp by applying direct force to the object.

- Ingressive: penetrate a surface of the object to pick it up.
- Astrictive: attractive forces are applied, like vacuum or magnetism.
- Contigutive: requires direct contact for adhesion.

The considered problem is related to the impactive type of gripper.

This type of gripper can be implemented as gripping fingers. In most common cases, there are two fingers with one movable join on each. With this configuration, you can clamp and securely hold most objects [2].

The force required to hold the object can be calculated with the following formula [3]:

$$F = \frac{ma}{\mu n} \tag{1.1}$$

where:

- F – is the force required to grip the object,
- m – is the mass of the object,
- a – is the acceleration of the object,
- μ – is the coefficient friction,
- n – is the number of fingers in the gripper.

The shape of the fingers can be chosen based on the shape of objects to grasp. But multi-purpose robot usually interacts with different shapes. In this case, the good point is to have an adjustable finger shape. It can be done with additional joints or with flexible fingers for the gripper. During the research, the research group has been using flexible fingers for the gripper as the most straightforward implementation of an adjustable gripping surface.

In the real world, we may have additional forces. One force we usually have is gravity. So more complete question should consider gravitational force. Hence, another term is introduced, and the formula becomes [4]:

$$F = \frac{m(a+g)}{\mu n} \tag{1.1}$$

where:

- F – is the force required to grip the object,
- m – is the mass of the object,
- a – is the acceleration of the object,
- μ – is the coefficient friction
- n – is the number of fingers in the gripper,
- g – gravity acceleration.

1.1.1. Fin-Ray effect

The Fin-Ray effect is an effect that describes a flexible construction that bends around an object when force toward the flexible structure is applied. Such construction gives an additional contact area that provides additional friction. Rays of fish fins inspired the structure. Fish fins have a structure with two bones that are attached to each other with elastic tissue. The tail fin is the main point to apply force for the movement. The tin consists of several basic structures stacked one above the other. The structure must be light but strong enough since excess weight would produce unnecessary energy loss. The design, which copies a fish fin, consists of two attached longitudinal fibers. There are cross fibers among the longitudinal fibers that keep the whole structure after assembly. The cross and longitudinal fibers are connected flexibly, which allows the required movement between them [5].

Fin-Ray Effect structure can be used in two ways. One way is to use it when the structure forms a manipulator. The second way is when the structure forms a finger gripper, which adaptively adjusts the shapes of objects transmitted.

To create a gripper, at least two fingers must act against each other and thereby maintain the objects carrying.

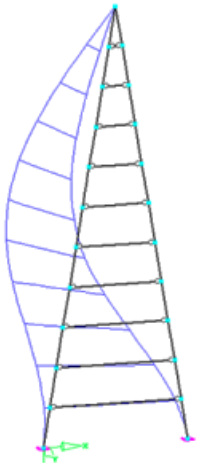


Fig. 1.2. The structure bends over an object

To move the manipulator endpoint is necessary to change the relative position of the beginning of the longitudinal fibers. It is a structure with two longitudinal fibers; the manipulator only allows two-dimensional motion. To achieve special three-dimensional motion must be a structure supplemented by other longitudinal fibers that is perpendicular to the two original fibers.

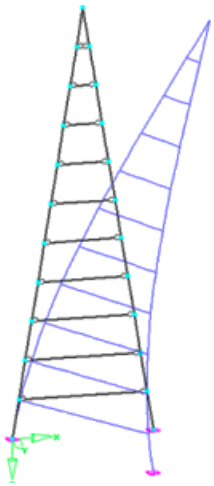


Fig. 1.3. The structure leans with applied force

The Fin-Ray effect was invented in 1997 by Leif Kniese and patented by EvoLogics GmbH Berlin [6].

Such finger construction provides a bigger contact spot that provides more traction with a gripped object. It provides a more secure grasp or decreases the required force that should be applied to the grasp.

1.1.2. Finger material

The research team was faced with a choice of how to produce fingers for the gripper. 3D printing was chosen as the most straightforward way which meets all requirements. 3D printing is a part of additive technologies; it is one of the easiest ways to make a piece of product or a prototype. The idea of the technology is to make a volumetric object by adding a substance layer by layer [7]. The process consists of next steps: modeling in CAD software, export from the CAD application, checking for model errors, processing with slicer – software to convert a model to series of thin layers described in G-code for a 3D printer, printing, finishing – adding more fine details with subtractive technologies.

Most used types of filaments:

- ABS – durable, impact-resistant, quite flexible, lightweight. Melts at 210-250 °C.
- PLA – easy to print, low flexibility. Melts at 180-230 °C.
- Nylon – strong, flexible, and high durable. Melts at 220-260 °C.
- TPU – thermoplastic polyurethane. Durable, very flexible. Melts at 190-220 °C [8].

Authors [9] propose a way to print 3d models with multiple materials. It can be useful to print fingers, where transverse lintels could be done with less flexible plastic while the body is done using the most flexible plastic.

1.2. Object detection and segmentation

Object detection is a task for computer vision and image processing. The task is to detect instances of objects on an image. It is an essential task in computer vision.

In the perspective of this study, object detection is necessary to detect an object to grasp. We need to separate all objects camera sees and the target object to grasp. The chosen method should label all objects and provide their masks. Then mask can be used to filter out all data except object-related data.

Image classification assigns a class label to an image, and object localization provides a bounding box around one or more objects in an image. Object detection is more challenging and combines these two tasks, draws a bounding box around each object of interest within the image, and assigns them a class label. Together, all these problems are referred to as object recognition. The next step is masking the object of interest, what called object segmentation [10].

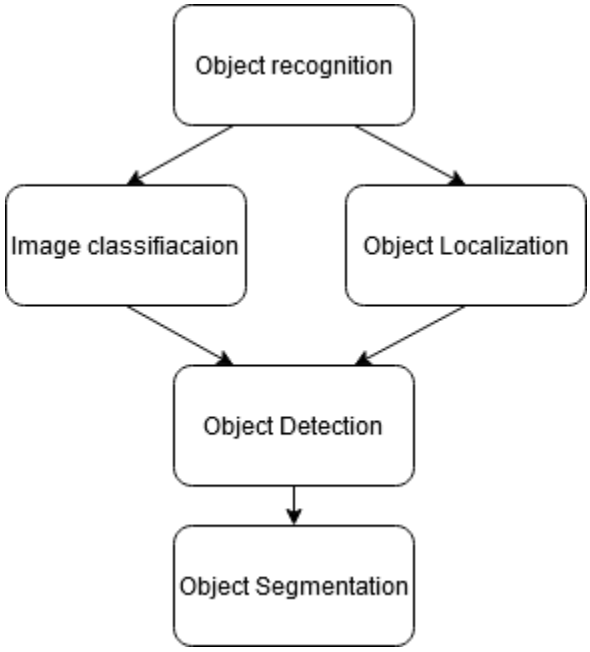


Fig. 1.4. Image processing tasks

So that we have the next computer vision tasks, which are also represented in Fig. 1.4.:

- Image Classification – recognize the type (class) of the object on the image. As an output, we retrieve labels of recognized objects.
- Object Localization – locate the object on the image. The output is bounding boxes on images around objects.
- Object Detection – locate and label the object. The output is bounding boxes with labels.
- Object segmentation – locate, label, and mask objects. As an output, we have a bitmap where one color means there is an object, and another color is for another space.

1.3. Grasp pose selection

Grasping an object is an everyday task, which humans and some animals perform subconsciously with both ease and reliability. By watching adults and gathering their own experiences, children rapidly learn how to grasp objects without the need to develop complex calculation models, solve complicated equations, or remember every object encountered so far by heart. With robots becoming progressively more intelligent in interacting with their environment, the need for a robust solution for grasping everyday objects is of utmost importance. Nevertheless, robotic grasping still provides many challenges for researchers and is still among the most demanding modern robotics problems. Finding a general solution would open up many new possibilities for robots to autonomously explore their environment and would enable them to perform better at assisting humans [10, 11].

While for humans and some animals, grasping is a routine task and is not difficult to perform, for robots, it is a complicated problem that requires in-depth research. People can know objects and know how to grip them, or when an object is unknown, people can decide how to take the item basing o similarity to previously known objects.

The same solution can be used for robots. There are two ways to detect poses for the gripper to grasp the object [12]:

- Finding previously known objects.
- Finding similarity of detected objects to geometrical primitives [3].

1.4. Obstacle detection and avoidance

The most crucial goal for operating the arm is to be safe – the robot should not touch any object except the target. There two possible collisions [13]:

- The robot itself – many arms have enough freedom to touch itself or other parts of the robot.
- Environment objects – any objects in the working environment except the robot.

We should take into account both parts of the obstacles.

1.4.1. Robot description

Usually, a robot is described as a hierarchical 3d object. This description can be used to avoid collisions with the environment and self-collisions. Modern robot software is using 3d description if it moves any part of the robot. In ROS (described in section 2.4), there is a special format for such a description – URDF and xacro files [14].

Robot elements are described as links and joints. The structure of them is presented in Fig. 1.5.

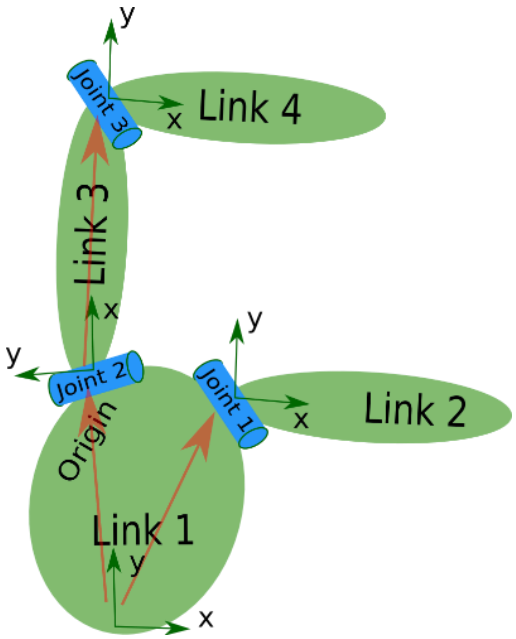


Fig. 1.5. Example hierarchy

URDF is acronym for Unified Robot Description Format, which is an XML format for representing a robot model. The robot in the image is a tree structure described with URDF below:

```
<robot name="test_robot">
 <link name="link1" />
 <link name="link2" />
 <link name="link3" />
 <link name="link4" />

 <joint name="joint1" type="continuous">
 <parent link="link1"/>
```

```
 <child link="link2"/>
 </joint>

 <joint name="joint2" type="continuous">
 <parent link="link1"/>
 <child link="link3"/>
 </joint>

 <joint name="joint3" type="continuous">
 <parent link="link3"/>
 <child link="link4"/>
 </joint>
</robot>
```

Xacro is an XML macro language. It provides a way to construct more readable XML files by using macros that expand to larger XML expressions.

After parsing xacro files and XML descriptions, it is one constructed tree. urdf_to_graphiz can visualize it graphically, as presented in Fig. 1.5.

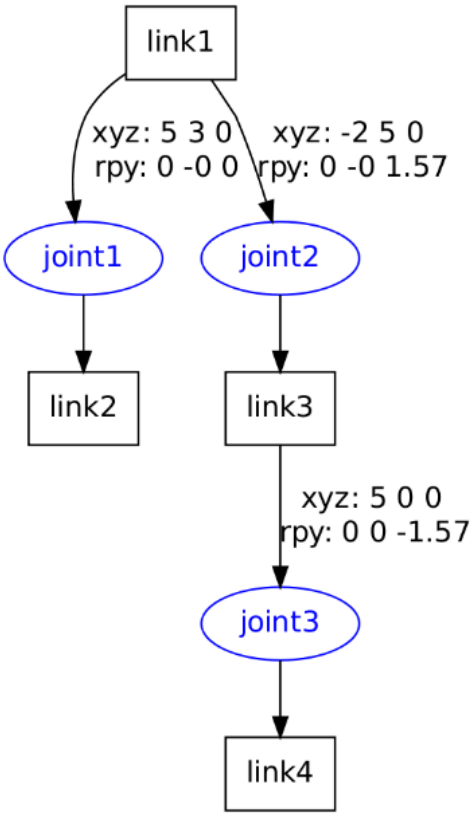


Fig. 1.6. Example hierarchy from the description

Having the information about arm parts and their positions in the space makes it possible to avoid self-collisions. There are many already developed approaches for self-collision avoidance [16, 4], or they can be a part of a complete obstacle avoidance algorithm [17].

1.4.2. Environment perception

The next problem is to detect and store information about the environment. The system should scan the environment with perception input devices, represent it as a data structure and process to avoid obstacles during path planning.

Since the robotic arm can move in 3D space, it should be full of volumetric obstacle detection. Usually is done by binocular or monocular 3D scanning using SLAM.

SLAM – simultaneous localization and mapping, is an approach to process multiple frames and sensor positions to build a volumetric model of the environment [15].

Visual slam can be performed even with a single camera (monocular) setup. It is cheap and straightforward. Depth is not fully available from one image; instead, we need multiple images to match them and compute a disparity [18].

With two cameras, it is even easier – we have a fixed distance between cameras

Having the disparity, we can set a distance from the camera for each point and, based on it, build a 3D representation of an image for the current frame.

The whole SLAM approach implies building a full map. SLAM can use multiple different types of sensors, and the powers and limits of various types of sensors have been a significant driver of new algorithms. Statistical independence is required to cope with metric bias and noise in measurements. Data from different types of sensors can be processed by different SLAM algorithms whose approaches are more compatible with the sensors. Laser scanners or visual features provide details of many points within an area, sometimes rendering SLAM inference is unnecessary because shapes in these point clouds can be quickly and unambiguously aligned at each step via image registration. At the opposite extreme, tactile sensors are too sparse as they contain only information about points very close to the agent. Hence, they require strong models to compensate in pure tactile SLAM. Most applicable SLAM tasks are somewhere between these visual and tactile areas [19].

We can use odometry, GPS, and localization based on joint angles for localization, based on camera images.

Loop closure is the problem of localization itself when recognizing a previously visited location. It can be complicated because of discrepancies and errors in input localization data. Typical loop closure methods consist of applying a second algorithm to compute some type of sensor measure similarity and re-set the location priors when a match is detected [20].

1.4.3. Point cloud

After the robot received any depth data from the image, it should store this data. One way is a point cloud – a set of points in space. When a camera (or it is generated with multiple frames) captured a depth image, each pixel of the depth image can be converted to a point in space relative to the camera position. Then this data could be added to previously generated point clouds from previous frames [21].

But if the robot only captures and adds the data, data will become bigger and bigger. Even when the camera sees the same object, it will add more and more points. It is an unnecessary significant amount of data.

1.4.4. Voxels and Octree

To represent flat images, we often use raster graphics – a two-dimensional regular rectangle grid, where each cell is called a pixel.

With high resolution, this data can be significant; there are several compression algorithms to decrease its size. One of them is quadtree.

Quadtree is a recursive algorithm of subdivision of one big quad exactly to four smaller equal quads. Subdivision occurs only when there are different colors in the area of the quad, and the maximum depth is not achieved [22].

Step by step:

1. Make one big quad with the size of the image.

2. Check if the depth is not maxed, and it has different colors. If no – finished.

3. Subdivide into four quads.

4. Check each quad if the depth is not maxed, and it has different colors inside.

– Has different colors – Go to 3.

– Has the same colors or depth is maxed – finished.

In the end, we have a tree structure, where the root node is the biggest quad, which was subdivided into smaller ones (Fig. 1.6). In case if it's an optimization of an image, it stops when the quad size achieves pixel size. When the input is not a raster image, it should have a specific depth limit.

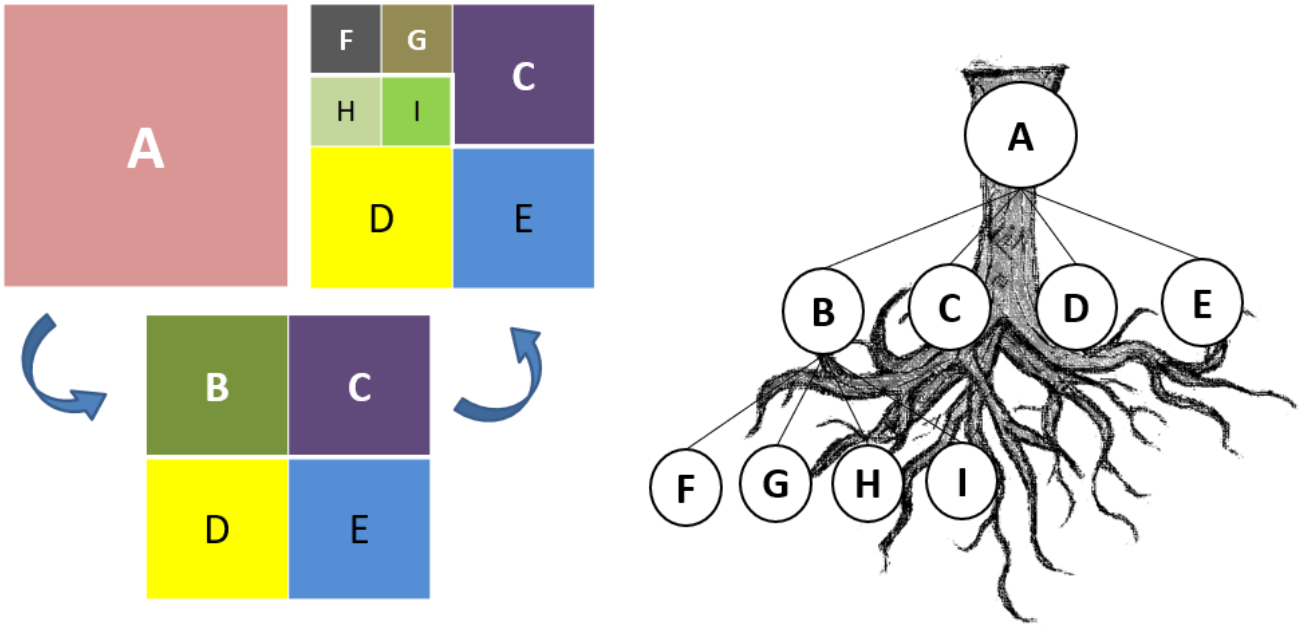


Fig. 1.7. Quadtree example

One of the ways to represent a 3D environment discretely is voxels. A voxel is a volumetric pixel. Basically, a voxel is a value in the regular grid. Usually, the value is a flag if it is occupied; additionally, it can contain any data, like color. In the typical case, if we have a voxel grid, the voxel does not represent a real position in 3d space; it contains only its position instead. Such an approach can be more efficient than more

common polygonal graphics, but sometimes it can be more computationally expensive [23].

One way of optimization of the voxels is also tree data structure – octree. It is like a quadtree approach, but for octants: each internal node has exactly eight equal children. Octant here can be understood as variable size voxel. In this case, in the beginning, we have one big scene-size root octant and recursively subdivide it until the desired depth is achieved. In the image below (Fig. 1.7), you can see an example octant with a different depth for octree octants. Octants are subdivided only when they contain some object inside, but not the whole octant is occupied by that object [24, 25].

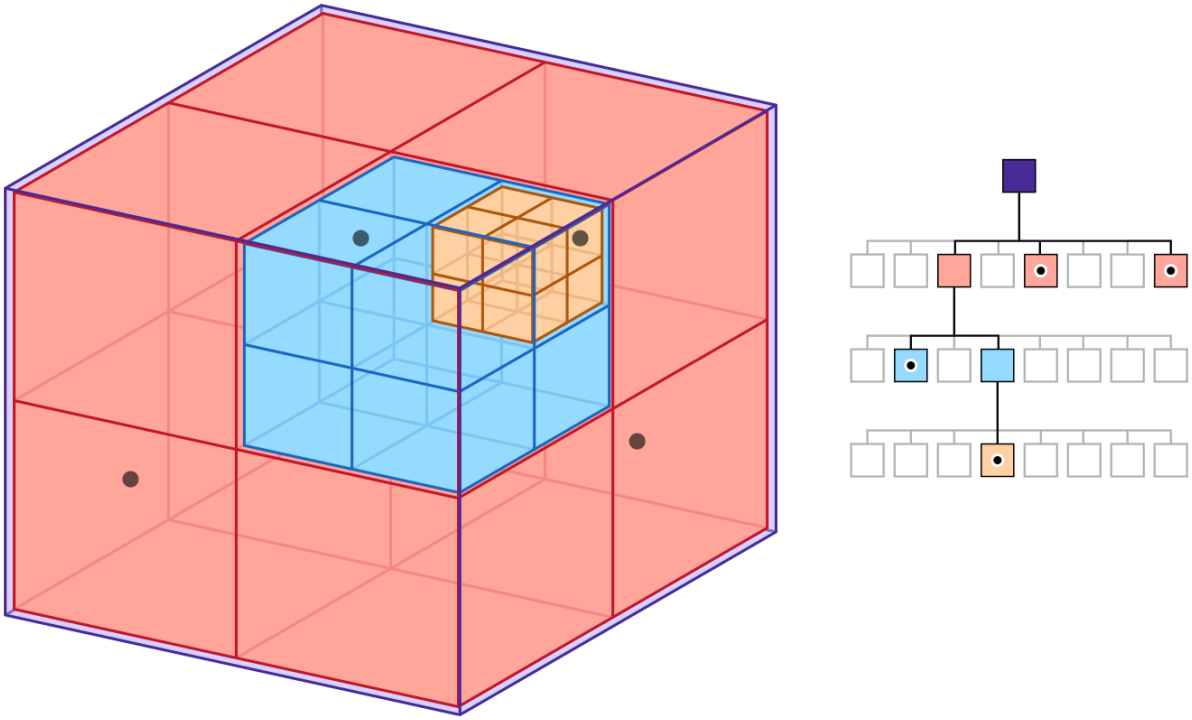


Fig. 1.8. Octree on sample data

In the illustration below (Fig. 1.8), you can see how each level of subdivision ads more fine details. But at the same time, when each subdivided octant would be occupied, there is no need to subdivide it at all; this octant will be kept as one piece.

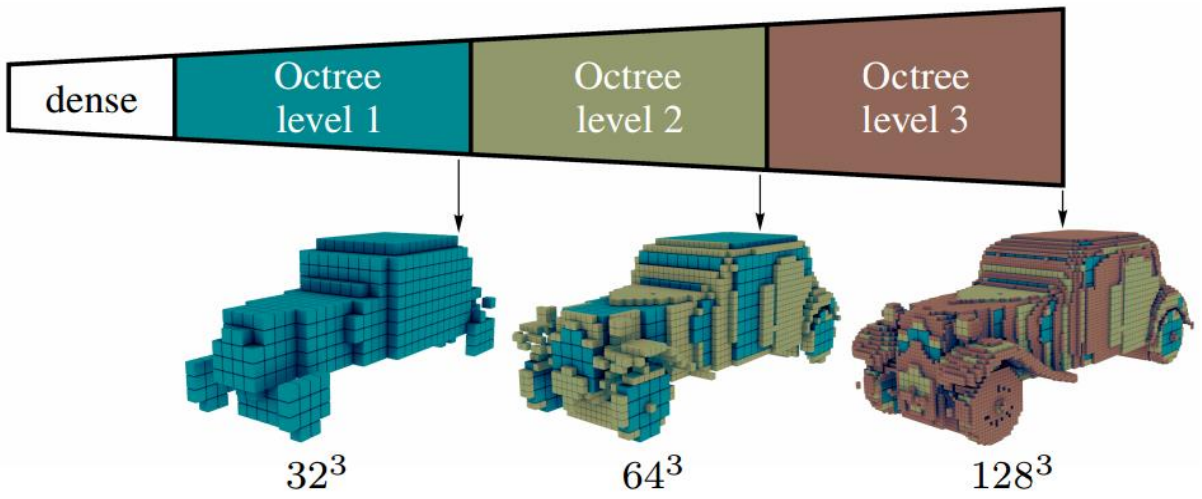


Fig. 1.9. The variable density of octree

1.5. Arm motion

The critical part of controlling the robotic arm is the motion planning of the arm. It includes problems as forward and inverse kinematics and motion planning itself. Forward and inverse kinematics is the way of representation of the arm's state with joint angles or with a cartesian pose of the end joint or end effector. Motion planning is building a plan of arm's movement to the required position set by joint angles or with the end joint or end effector's cartesian pose.

1.5.1. Forward and inverse kinematics

Forward kinematics is a process of defining an end-effector position from angles on each joint on the arm [26].

Inverse kinematics is a process of defining the rotation of the required angles from the desired end-effector position (Fig. 1.9.) [26].

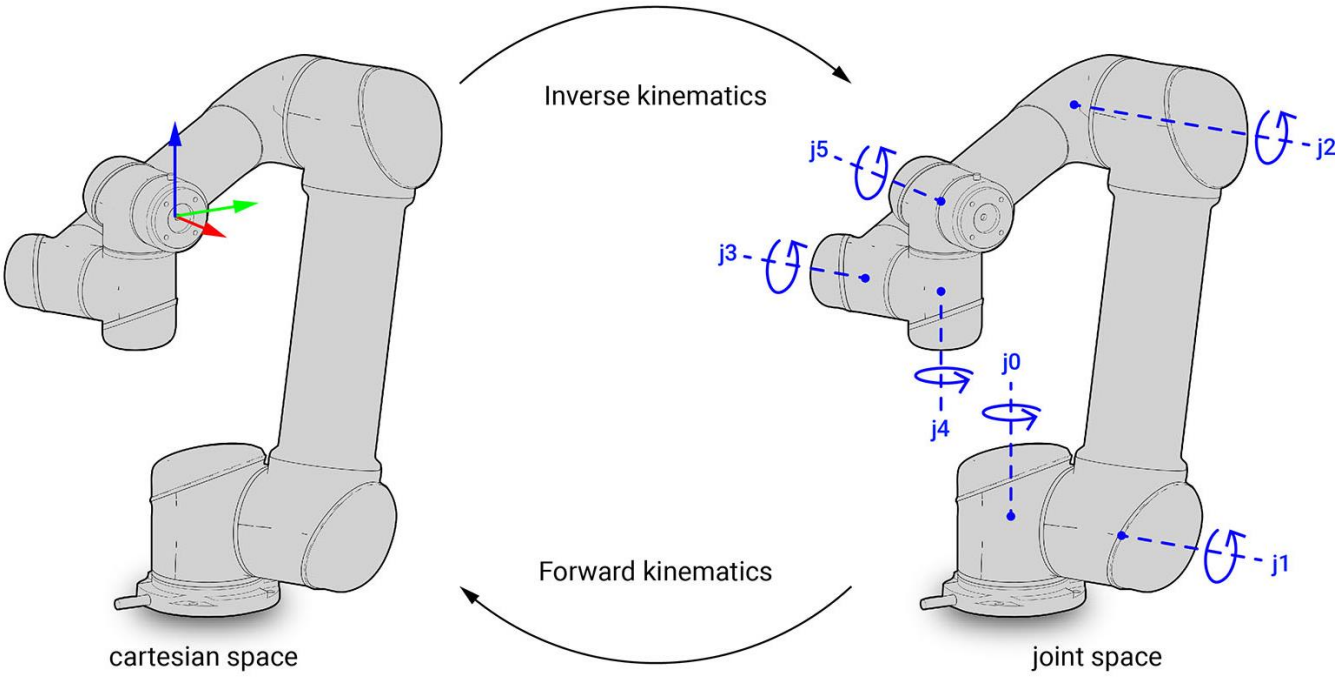


Fig. 1.10. Directions of transformation in FK and IK

Accordingly, these processes are the opposite. FK defines the position in cartesian space by joint space, and IK defines joint-space positions from cartesian space [26]. On the lowest level, there is FK because the only way to move the arm is to set angles for each joint. But usually, there is a target end effector should achieve, and it exists in cartesian space; in this case, IK can provide angles to achieve the desired position.

1.5.2. Motion planning

Motion planning or path planning is a computational problem to find a sequence of available actions that moves from a source position to a destination.

For mobile robots, this problem contains the only movement of solid shape in an environment while motion planning for robotics arm works with arms with multiple degrees of freedom, avoiding collision with itself and the environment [27].

A robust motion planning approach is sampling-based planning. It means that the robot makes random points (samples) it can achieve in a workspace, stores positions to achieve them, and builds a path, including these samples [28].

1.6. Grasp and Slip detection

The robotic gripper can perform activities that are not available for humans, but some activities are not available or difficult to perform. One such problem is a perception of a secure grasp; it means that the grasped object is robustly fixed in the gripper and is not moving.

The robotic hand is considered a mechatronic instrument that can do some activities that are impossible for humans. The robotic hand is widely used in manufacturing and dangerous nuclear industries, as well as in precise applications such as military or medical implementations. In addition, repetitive and maintenance tasks could be achieved with high-performance accuracy. Consequently, evolving robotic hand is required to cover a wide range of tasks and provide a robotic hand with special types of sensors to measure the grasping force for a particular object. Grasping objects could be achieved using the dexterous robotic hand presented with the ability to grasp both soft and hard objects. In gripping operation has been implemented by robotic hands that use particular types of tactile sensors, which employ physical properties and events through contact with objects. Many tactile sensors have been developed, and the sensor hardware has evolved to achieve specific gripping tasks.

Moreover, to accomplish the gripping mechanism using a robotic hand, some efforts have been expended in developing tactile pressure sensor structures, such as in. In most recent studies, advanced robotic manipulations have used tactile pressure sensors implemented in different applications. The main exciting issue in advanced robotic manipulation tasks is that the robotic hand is required to be equipped with distributed tactile pressure sensors that can continuously provide information about the magnitude and direction of forces at all contact points between the sensing area and a subjected object. Numerous studies have reported the proposed method that uses tactile sensor information through physical contact between the sensor and an object to detect both pressure force and hardness of the object. In addition, several studies have documented that tactile pressure sensors have been utilized successfully in different design concepts and action principles. These tactile sensors have presented the process

of determining physical features with the environment, measuring applied forces exerted over an object, and the art in tactile sensing and investigating trends [29] [30].

Grasp is successful when the object is in the gripping area between fingers, they are closed, and there is no sliding of the gripped object. There are two types of sensors considered to use for it:

- ToF sensor measures the distance to an object using a light trip duration from a source to a receiver.
- A tactile sensor measure is a device that measures information arising from physical interaction with its environment.

An optical ToF sensor can be used to detect the object's presence between the gripper's fingers. There are various types of ToF sensors out there, but most are Time of Flight cameras and laser scanners that use a technology called Lidar (Light detection and ranging) to measure the depth of various points in an image by illuminating with infrared light [30].

Data is generated and captured with ToF sensors are beneficial. Such data can provide pedestrian detection, authenticate users based on facial features, perform environment mapping using SLAM algorithms, and many more.

ToF sensors use a tiny laser to fire out infrared light where the light produced out will bounce off any object and return to the sensor. Based on the time difference between the light emission and its return to the sensor after being reflected by an object, the sensor can measure the distance between the object and the sensor.

Tactile sensors are generally modeled after the biological sense of cutaneous touch, which is capable of detecting stimuli resulting from mechanical stimulation, temperature, and pain (although pain-sensing is not common in artificial tactile sensors). Tactile sensors are used in robotics, computer hardware, and security systems. A typical application of tactile sensors is in touchscreen devices on mobile phones and computing.

The flexible tactile sensor has been extensively investigated as a critical component for emerging electronics applications such as robotics, computer hardware, wearable devices, and security systems [31].

The active sensors require external power for their operation, which is called an excitation signal. The sensor modifies that signal to produce the output signal. The active sensors sometimes are called parametric because their own properties change in response to an external effect, and these properties can be subsequently converted into electric signals. It can be stated that a sensor's parameter modulates the excitation signal, and that modulation carries information of the measured value. For example, a thermistor is a temperature-sensitive resistor. It does not generate an electric signal, but by passing an electric current through it (excitation signal), its resistance can be measured by detecting current and/or voltage variations across the thermistor. These variations (presented in ohms) directly relate to temperature through a known transfer function. Another example of an active sensor is a resistive strain gauge in which electrical resistance relates to a strain. An electric current must be applied to it from an external power source to measure a sensor's resistance [32].

1.7. Conclusions to the first section

This section is devoted to algorithms and approaches that can be used for researched problems. Attention is paid to gripper's fingers structure and material, object segmentation methods, grasp pose selection, obstacle detection and avoidance approaches, arm motion algorithms, and slip detection solutions.

The shape of the fingers can be chosen based on the shape of objects to grasp. But multi-purpose robot usually interacts with different shapes. In this case, the good point is to have an adjustable finger shape. It can be done with additional joints or with flexible fingers for the gripper. During the research, the research group has been using flexible fingers for the gripper as the most straightforward implementation of an adjustable gripping surface. The Fin-Ray effect is an effect that describes a flexible construction that bends around an object when force toward the flexible structure is applied. Such construction gives an additional contact area that provides additional friction.

The research team was faced with a choice of how to produce fingers for the gripper. 3D printing was chosen as the most straightforward way which meets all requirements. 3D printing is a part of additive technologies; it is one of the easiest ways to make a piece of product or a prototype.

In the perspective of this study, object detection is necessary to detect an object to grasp. We need to separate all objects camera sees and the target object to grasp. The chosen method should label all objects and provide their masks. Then mask can be used to filter out all data except object-related data.

While for humans and some animals, grasping is a routine task and is not difficult to perform, for robots, it is a complicated problem that requires in-depth research. People can know objects and know how to grip them, or when an object is unknown, people can decide how to take the item basing o similarity to previously known objects. The same solution can be used for robots. There are two ways to detect poses for the gripper to grasp the object [12]:

- Finding previously known objects.
- Finding similarity of detected objects to geometrical primitives [3].

The most crucial goal for operating the arm is to be safe – the robot should not touch any object except the target. There two possible collisions [13]:

- The robot itself – many arms have enough freedom to touch itself or other parts of the robot.
- Environment objects – any objects in the working environment except the robot.

Usually, a robot is described as a hierarchical 3d object. This description can be used to avoid collisions with the environment and self-collisions. Modern robot software is using 3d description if it moves any part of the robot.

The next problem is to detect and store information about the environment. The system should scan the environment with perception input devices, represent it as a data structure and process to avoid obstacles during path planning.Since the robotic arm can move in 3D space, it should be full of volumetric obstacle detection. Usually is done by binocular or monocular 3D scanning using SLAM.

SECTION 2
DEVELOPMENT OF THE ARM CONTROL ALGORITHM

2.1. Robotic platform

During the research, we have been working on the robot of the RT-Lions team of Reutlingen University. The robot was built for participation in the RoboCup championship.

The basement of the robot is an omnidirectional moving platform Neobotix MPO-700 (Fig. 2.1.). Its four Omni-Drive-Modules enable it to move exceptionally smoothly in any direction. This robot is even capable of rotating freely while driving to its destination. The Omni-Drive-Modules of the MPO-700 feature essential benefits compared to other omnidirectional drive kinematics.

Fig. 2.1. Mobile platform MPO-700

Depending on the intended application, the MPO-700 can be used on its own, in combination with other robot vehicles, and in combination with stationary systems.

Furthermore, application-specific extensions can be integrated into the basic platform. These might be a customized cargo area, a robot arm, or special sensors.

The MPO-700 may only be used in laboratories, test halls, or similar environments. It is not recommended to use the MPO-700 in any other surroundings, especially not outdoors.

MPO-700 has a computer with preinstalled ROS packages to move the robot. We can access this computer with a remote desktop connection and run preinstalled software.

On top of it, we placed a general-purpose robot platform that contains the main software module. This platform is a central part of the robot, which orchestrates the whole system. The platform uses Linux 16.04 with ROS Kinetic.

Mechanically it is done with an aluminum body. On the body, we have ventilation holes, ports, emergency stop buttons. There is a router with a Wi-Fi antenna inside. So, we can connect remotely to the robot without cables.

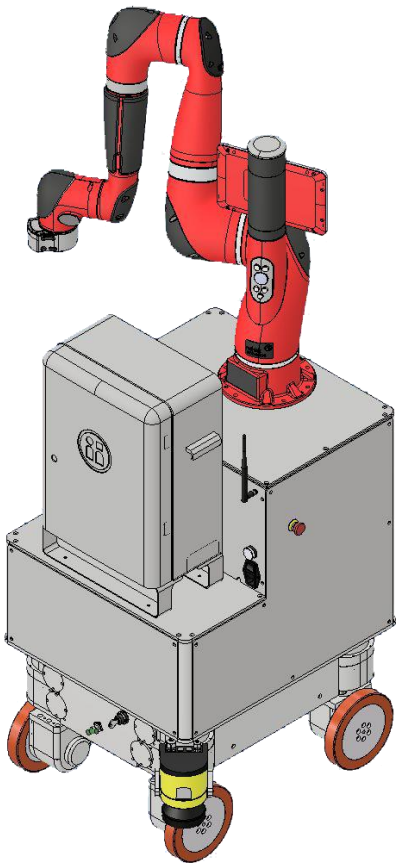


Fig. 2.2. 3D model of the used robot

The next level is the robotic manipulator, which represented Sawyer arm. It also has its own computer with required ROS nodes. Usually, we do not have and need access to the robot. Everything can be done via API. This part is not a subject of this work, so it does not require a comprehensive description.

2.2. Sawyer arm

During the research, the team was able to use Sawyer Arm made by Rethink Robotics. It is 7 degrees of freedom robotics arm. It is provided with built-in software Intera.

In most common cases, there are 6 degrees of freedom for the robotics arm; it is enough minimum to achieve any desired position in the workspace. An additional degree of freedom can be considered additional flexibility, more positions to avoid obstacles, and properly grasping an object. But existing software for motion planning works better with arms that have 6 degrees of freedom. More degrees of freedom also requires more computations for planning, so it can be slower to find the best way to approach the object.

The Intera 5 software platform provides an easy-to-deploy approach to industrial automation. The software innovation drives manufacturing productivity worldwide and enables a high acceptance of its new robot co-workers. Rethink Robotics GmbH provides increased functionality and capabilities through frequent software releases. To this day, the Intera 5 software platform developed to control Sawyer remains the most innovative operating system in collaborative robotics [33].

2.2.1. Motion interface

The Sawyer arm is supplied with a built-in motion interface. The Motion Interface enables the user to easily generate smooth motions for Sawyer by specifying a sequence of waypoints: the motion controller on Sawyer will automatically generate and run a trajectory that passes through the waypoints. The user can specify a variety

of options, such as a maximum speed or requiring that the endpoint move on a linear path between the waypoints.

With this API, user can set joint angles or cartesian pose of the end of the arm.

2.3. Gripper

The gripper is a custom solution based on fingers with the Fin-Ray effect (Fig. 2.3.). The fingers can adapt to these spherical objects in two dimensions when gripping them. This is achieved through semi-circular recesses in the struts of the fingers. In addition, the fingers have a pocket-shaped recess on the inside, which makes it easier to grip spherical objects. The two joints at the top have no semi-circular recesses, but they are provided with constrictions to prevent the finger from twisting when gripping small objects.

The research group has been using flexible gripper fingers as the most straightforward implementation of an adjustable gripping surface during the research. The fingers are done with FDM 3D printing with TPU and ABS material. Other parts of the case are made with aluminum and ABS plastic.

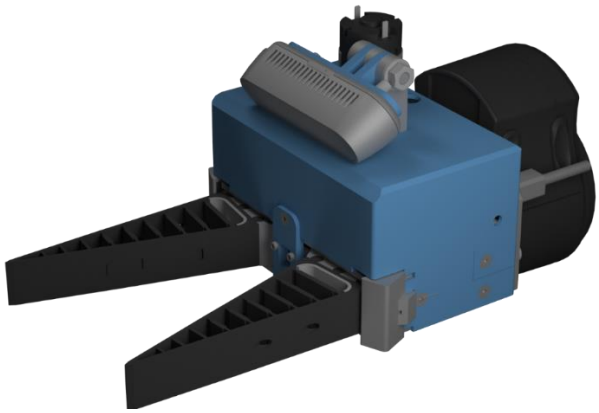


Fig. 2.3. The gripper

On the top of the gripper, we can mount a Realsense D435 camera (described in section 2.5) on a rotatable connection. This camera can be used for building the octree or object recognition for grasp pose detection.

Sensors and motor are connected to an Arduino board (Fig. 2.4.), which retrieves data from sensors, converts it to the needed format, and sends it to the main computer. The central computer can send a signal to open or close fingers to the Arduino board,

The main body covered with a big box contains the mechanism to close the fingers. It is driven by one motor and several gears (Fig. 2.5.).

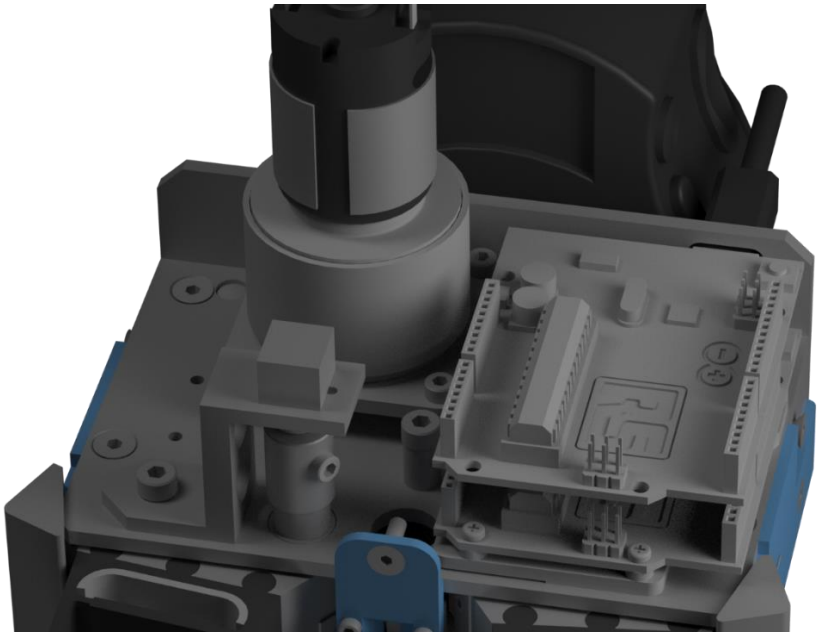


Fig. 2.4. Gripper without the body

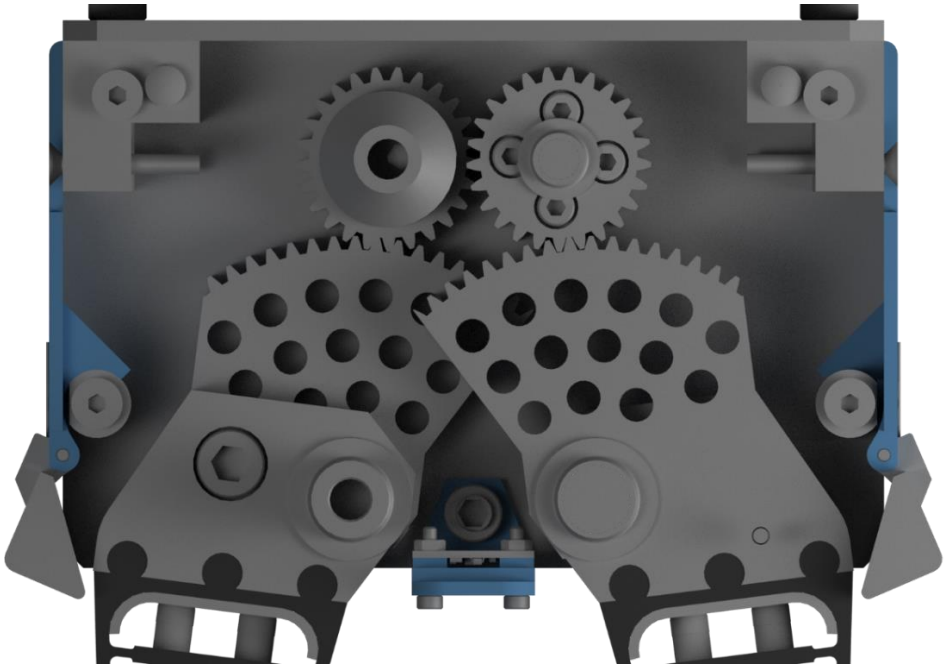


Fig. 2.5. The mechanical structure of gripper

2.4. Robot Operating System

ROS is a platform for robot development. It provides a framework for robot software communication and a variety of built-in software. ROS, as a platform, can be subdivided into three parts [2]:

- Tooling for package distribution.
- ROS client library for compatible languages.
- Packages for several purposes, such as visualization, navigation, message publishing, etc.

Despite the fact that it is called an operating system, it is not an operating system in the usual sense. But it provides more abilities than just a framework; it provides OS-like functions such as hardware abstraction, package management, developer toolchain, strict file organization, which is even called file system (when it is rather a file organization structure).

ROS framework is organized as many nodes that work in separate processes. A node can send and receive messages through topics; provide services; call services provided by another node; set and receive data in parameter server. The main process called ROS Master orchestrates all mentioned above; it should be run first. ROS Master register nodes; set up inter-node communication with topics and services and control parameter server updates. Inter-node communication is peer-to-peer between nodes after they were registered.

A ROS node is a single process running the ROS graph. Every node must have a name; it is the very first step during registration. Multiple nodes with different names can exist only under different namespaces. If the node is defined as anonymous, it will randomly generate an additional identifier to add to its given name. Nodes are the main part of ROS programming; each ROS node takes actions based on information received from other nodes, sends information to other nodes, or sends and receives requests for actions to and from other nodes.

The message system in ROS consists of topics – buses through which nodes can send and receive messages. The topic must have a unique name within its namespace

as well as nodes. To send messages to the topic node should register the publisher. To receive messages it must subscribe to a topic. The topic has its data format, it can be built-in message types or user-defined with msg files. The content of the mentioned messages can be next: data from sensors, control commands for a motor, state information, actuator commands, or anything else.

The other way to communicate between nodes is through services. A service is a function or procedure registered with a unique name and can be called by its name. The service can require input data via parameter and return data after its call. Services are used for rare actions but not for constant processing of continuous data stream.

Besides the framework, ROS also provides a variety of tools that allow developers to visualize and record data, easily navigate the ROS package structures, and create scripts automating complex configuration and setup processes. These tools provide solutions to many common robotics development problems that significantly increases the capabilities of systems using ROS. Examples of built-in packages:

- RViz – is a three-dimensional visualizer used to visualize almost any data. Out of the box, it can visualize robots, the environments they work in, and sensor data. It is a highly configurable and extendable tool; it can be configured for many different purposes and supports extensions with plugins.
- rosbag is a command-line tool used to record and playback ROS message data. rosbag uses a file format called bags, which log ROS messages by listening to topics and recording messages as they come in. After data is recorded, you can play it again, i.e., publish the same messages in the same order as it was during the record. Besides the command line tool rosbag, rqt_bag provides a GUI interface to rosbag.
- catkin is a build system for ROS packages. It is based on environments where you can store

2.4.1. ROS Filesystem

A package is a unit of ROS software. A package contains one or more ROS programs (nodes), libraries, configuration files, etc., organized together as a single unit

(Fig. 2.7.). Packages are an atomic build and release element in ROS software (Fig. 2.6.). The package is defined with a package manifest that contains information about the package, author, license, dependencies, compilation flags, and so on. The package.xml file inside a ROS package is the manifest file for that package [34].

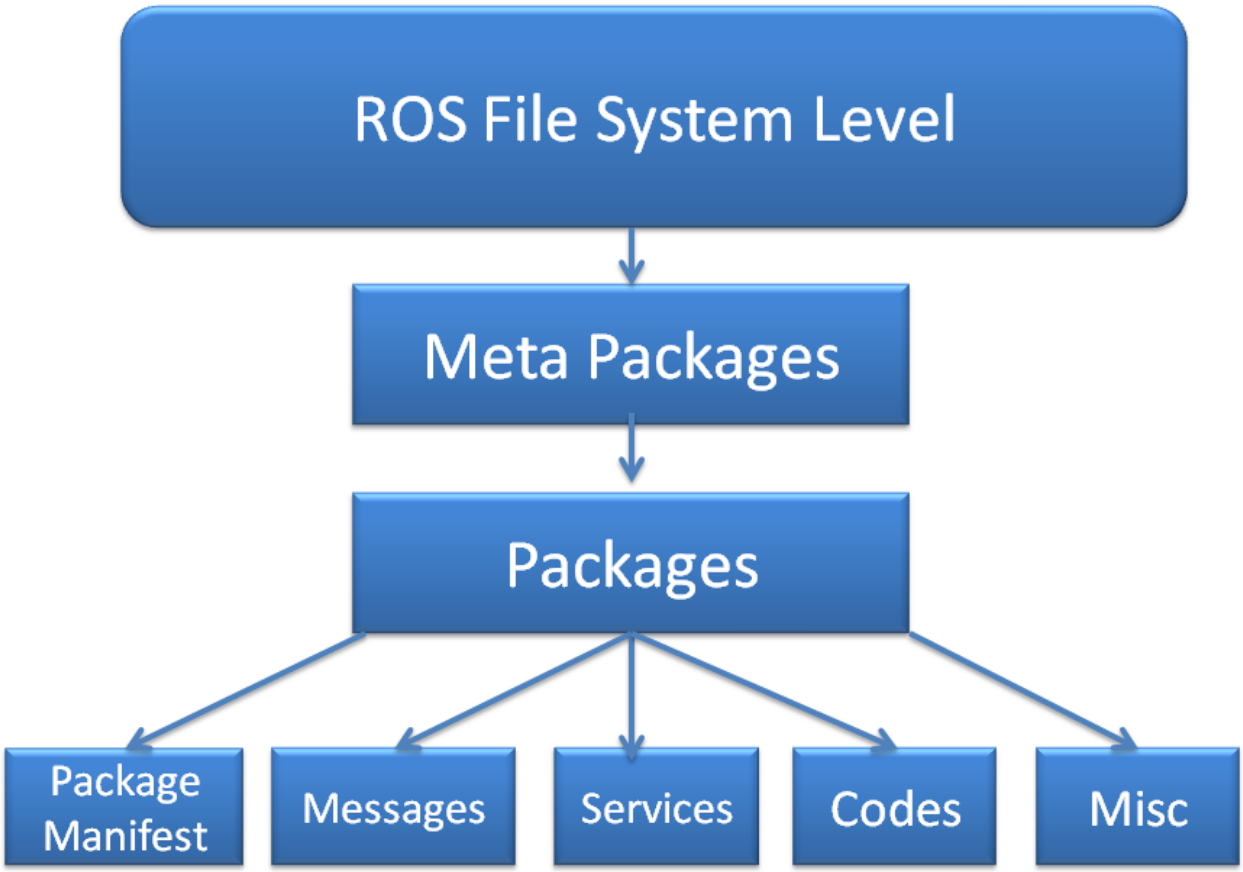


Fig. 2.6. Ros Filesystem [34]

The term metapackage refers to one or more related packages that can be loosely grouped. In principle, metapackages are virtual packages that do not contain any source code or typical files usually found in packages. The metapackage manifest is very close to the package manifest, but the difference is that it can include packages inside it as runtime dependencies and declare an export tag [2, 34].

ROS message is a structured piece of information that can be sent between nodes. It can be defined with a ".msg" file inside a "msg" folder in a package folder.

The ROS service is a kind of request-response interaction between processes. Request and response data can be defined in "srv" folder inside the package folder with ".srv" extension.

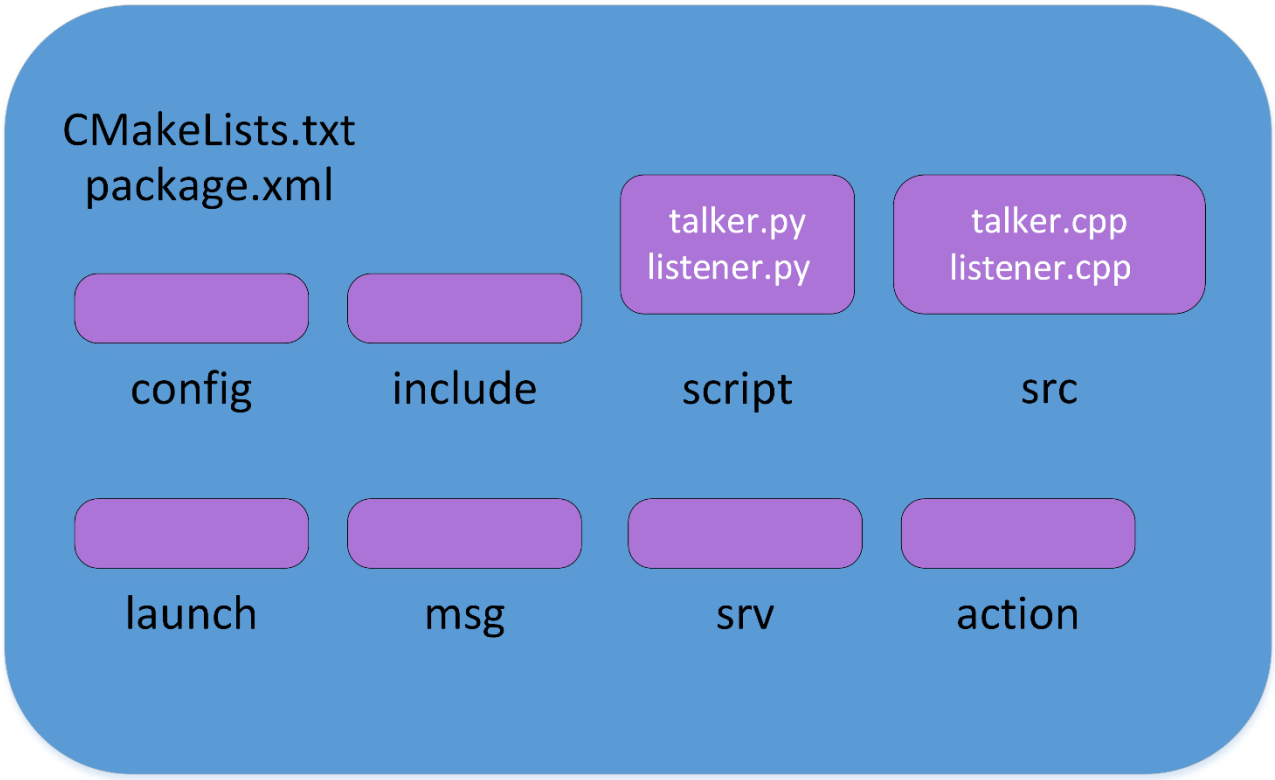


Fig. 2.7. Structure of typical C++ package

2.4.2. Transformation Tree

Any robot will have a lot of subsystems, such as a mobile base; perception devices like laser scanners, lidars, cameras attached to the base to provide the ability for it to navigate through the environment, and a manipulator arm with an end-effector that will do the actual grabbing of items. A really good item-fetching robot might have many more features than mentioned before; items are enough to make coordinate frames a critical concern.

In our 3D world, a position is a vector of three numbers (x, y, z) that describe how far we have translated along each axis, concerning some origin. Similarly, orientation is a vector of three numbers (roll, pitch, yaw) that describe how far we have

rotated about each axis, again concerning some origin. Taken together, a (position, orientation) pair is called a pose. For clarity, this kind of pose, which varies in six dimensions (three for translation plus three for rotation), is sometimes called a 6D pose. Given the pose of one thing relative to another, we can transform data between their frames of reference, a process that usually involves some matrix multiplications.

We need to know: what is the pose of the origin of the laser with respect to the pose of the base? That's not all, of course. And if we're going to use the base-mounted camera to find items in the environment, then we likely need to know the camera's pose with respect to the base. If we're going to use the locations of items found by the camera to send goals to the hand, then we further need to know the pose of the camera with respect to the hand. This case is especially interesting because the camera-to-hand relationship might be changing all the time as the arm moves the hand with respect to the camera. Then you have the mobile base moving around in the world (e.g., defined by a map), so there's a base-to-world relationship that is also continually changing.

We also need a message format to use when publishing information about transforms. In tf, we use tf/tfMessage, sent over the /tf topic. You don't need to know the details of this message because you're unlikely ever to manipulate one manually. It's enough to know that each tf/tfMessage message contains a list of transforms, specifying for each one the names of the frames involved (referred to as parent and child), their relative position and orientation, and the time at which that transform was measured or computed. Time turns out to be extremely important when we talk about sensor data and coordinate frames. If you want to combine a laser scan from one second ago with a scan from five seconds ago, then you had better keep track of where that laser was overtime and be able to convert the scan data between its one-second-ago pose and its five-seconds-ago pose. We don't want every node that works with transform data to reinvent the publishing, subscribing, remembering, or computing transforms. So, tf also provides a set of libraries that can be used in any node to perform those common tasks. For example, if you create a of listener in your node, then, behind the scenes, your node will subscribe to the /tf topic and maintain a buffer of all the tf/tfMessage data published by other nodes in the system. Then you can ask questions

of tf, like: Where is the laser with respect to the base? Or, where was the hand with respect to the map two seconds ago? How does this point cloud be taken from the depth camera look in the laser frame? In each case, the tf libraries handle all the matrix manipulations for you, chaining together transforms and going back in time through its buffer as needed. As is often the case for a robust system, tf is relatively complex, and there are a variety of ways in which things can go wrong. Consequently, there many tf-specific introspections and debugging tools to help you understand what's happening, from printing a single transform on the console to rendering a graphical view of the entire transform hierarchy [35].

2.5. Environment perception using Realsense

The Intel RealSense depth camera D435 is a stereo solution, offering quality depth for various applications. It is a wide field of view that is perfect for applications such as robotics or augmented and virtual reality, where the most important part is seeing as much as possible. With a range of up to 10m, this small form factor camera can easily be integrated into any solution and comes complete with Intel RealSense SDK 2.0 and cross-platform support [36].

The Intel RealSense D435 depth camera has a wider FOV at approximately 85° field of view.

D435 depth camera has a global shutter. Cameras with a rolling shutter record all the pixels in a scene by rapidly scanning either left and right or vertically. This will usually happen for a couple of frames, but the data will be saved as a single frame. Global shutter cameras operate differently in that they snapshot the whole scene in a single frame, so every pixel is captured simultaneously. In practice, because rolling shutter cameras capture an image in sections slightly divided by time, it can lead to odd image artifacts when something in the scene is moving rapidly.

The output of the Realsense camera is a colored point cloud.

2.5.1. Post-processing of point cloud with PCL

The density of the produced point cloud could be too big to process it further. Also, the camera can capture objects which are out of the available workspace of the arm. Both problems lead to additional unnecessary computation.

To solve these problems, we use PCL – Point Cloud Library. It is a set of algorithms for point cloud data, including filtering, feature estimation, surface reconstruction, registration, model fitting, and segmentation [37].

The large density of the objects can be solved by downsampling the raw point cloud with the VoxelGrid filter. The VoxelGrid class creates a 3D voxel over the input point cloud data. Then, in each voxel, all the points present will be approximated (i.e., downsampled) with their centroid. This approach is slightly slower than approximating them with the voxel center, but it represents the underlying surface more accurately.

Points that are outside of the workspace are removed with the PassThrough filter. This filter simply deletes all points where coordinates out of set limits.

2.6. Object segmentation with YOLO

One family of object detection approaches is YOLO what stands for You Only Look Once. This approach's advantage is speed; it is usually faster than another popular R-CNN approach, although it is less accurate.

Joseph Redmon first described the YOLO model in 2015 with a paper titled "You Only Look Once: Unified, Real-Time Object Detection." [38]. During the work on this paper, they achieved the 45 FPS performance.

The detection system uses classifiers to perform detection. This system takes a classifier for that object and evaluates it at various locations, and scales in a test image to detect an object. At the same time, systems such as Deformable Part Models (DPM) use a sliding window approach where the classifier runs at evenly spaced locations throughout the image.

Later approaches like R-CNN use region techniques to create potential bounding boxes in the image and then run a classifier on those suggested rectangles. After classification, post-processing is used to refine the bounding boxes, eliminate duplicate detections, and re-evaluate the boxes based on other objects in the scene. Those complex pipelines are slow and difficult to optimize because each component must be trained separately. YOLO learns from full images and directly optimizes detection performance. This unified model has several advantages over traditional object detection methods. First, YOLO is very fast. Since it treats detection as a regression problem, it doesn't need a complex pipeline. It merely runs a neural network on a new image during testing to predict the detection. Second, YOLO considers globally about the image when making predictions. In contradistinction to sliding window and region proposal-based techniques, instead, YOLO sees the entire image during the training and test time, so it implicitly encodes contextual information about classes as well as their appearance. Fast R-CNN is a top detection method, mistakes background patches in an image for objects because it cannot see the larger context. YOLO makes less than half the number of background errors compared to Fast R-CNN. Third, YOLO learns generalizable representations of objects. When it is trained on natural images and tested on the artwork, YOLO successfully outperforms top detection methods like R-CNN and DPM by a wide margin. Since YOLO is highly generalizable, it does not tend to break down when applied to unexpected inputs or new domains.

YOLO still performs behind state-of-the-art detection systems in terms of accuracy. But it can quickly identify objects in images, it struggles to localize some objects, tiny ones, precisely.

2.7. Grasp pose detection

Before grasping any object, the object should be recognized and chosen. It is a problem in robotics arm control to solve. Approaches to grasp perception can be subdivided into the following ways [12]:

- With known models. Based on the recognition of predefined loaded models. The method tries to find similar objects among predefined CAD models. It can be more accurate with known objects but usually does not work if the object is not recognized.
- Without known models. When the system allows finding a grasping position for the object even if it is an unknown object, but at the same time, it can also support the detection of known objects.

There is an implementation of the second approach – the Grasp Pose Detection (GPD) package. It provides 6-DOF grasp poses for objects. It means that an object can be grasped from any side with any orientation if the arm can provide such a pose. At the moment, it supports only two-finger parallel grippers, which is appropriate for the gripper we use. Besides, the gripper is not a parallel jaw gripper, rather a scissors-type [3].

As an input, GPD consumes point cloud data and outputs available grasp poses. The main strengths of GPD are:

- works for novel objects (no CAD models required for detection),
- works in dense clutter, and
- outputs 6-DOF grasp poses (enabling more than just top-down grasps).

If we use just raw point cloud in GPD, it will cause excessive grasp poses, like grasping the table or other objects. We need to filter it out. It can be done by setting the workspace, which includes only objects on the table. A different and more useful approach is filtering out all points from the point cloud that are not related to the target object. The filtering approach is explained in part 2.6

As the output, we have several scored grasp poses. The score shows how likely grasp candidate can be grasped. We choose a candidate with the best score. With further work, we can introduce an additional value – the cost of movement, which is a sum of angle differences for each joint. And Select the grasp pose, which also is closer to the current arm position.

2.8. Environment description with Octomap

The OctoMap library implements a 3D occupancy grid mapping approach by providing data structures and mapping algorithms in C ++, especially suitable for robotics. The implementation of the map is based on an octree and is designed to satisfy the next requirements [39]:

- Full 3D model. Octomap can model arbitrary environments without any prior assumptions about it. It represents occupied areas as well as free space. New areas of the environment are implicitly encoded in the map to build. While the distinction between free and occupied space is crucial for safe robot navigation, information about unknown areas is essential, e.g., for autonomous exploration of an environment.
- Updatable. It can add new information or sensor readings at any time. Modeling and updating are done in a probabilistic manner. This took into account a sensor noise or measurements which result from dynamic changes in the environment, e.g., because of dynamic objects. Moreover, multiple robots can contribute to the same map, and a previously recorded map will be extended when new areas are explored.
- Flexible. The size of the map is not necessary to be known in advance. Instead, the map is dynamically expanded upon need. The map is variable-resolution so that, for instance, a high-level planner can use a rough map, but a local planner may operate using a sufficient resolution. This also allows for efficient visualizations that scale from rough overviews to detailed close-up views.
- Compactness. The map is stored efficiently, as well as in RAM and persistent storage. It is possible to generate a well-compressed file for usage later or for convenient exchange between robots even under bandwidth constraints.

In the research, we are not using Octomap directly, but we use it inside the MoveIt framework.

2.9. Motion planning with obstacle avoidance using MoveIt!

MoveIt is an easy-to-use open-source robot manipulation platform for commercial application development, prototyping, and testing algorithms [40].

The picture below (Fig. 2.8) shows a high-level system architecture for the primary node move_group. This node serves for integration purposes: it connects all components.

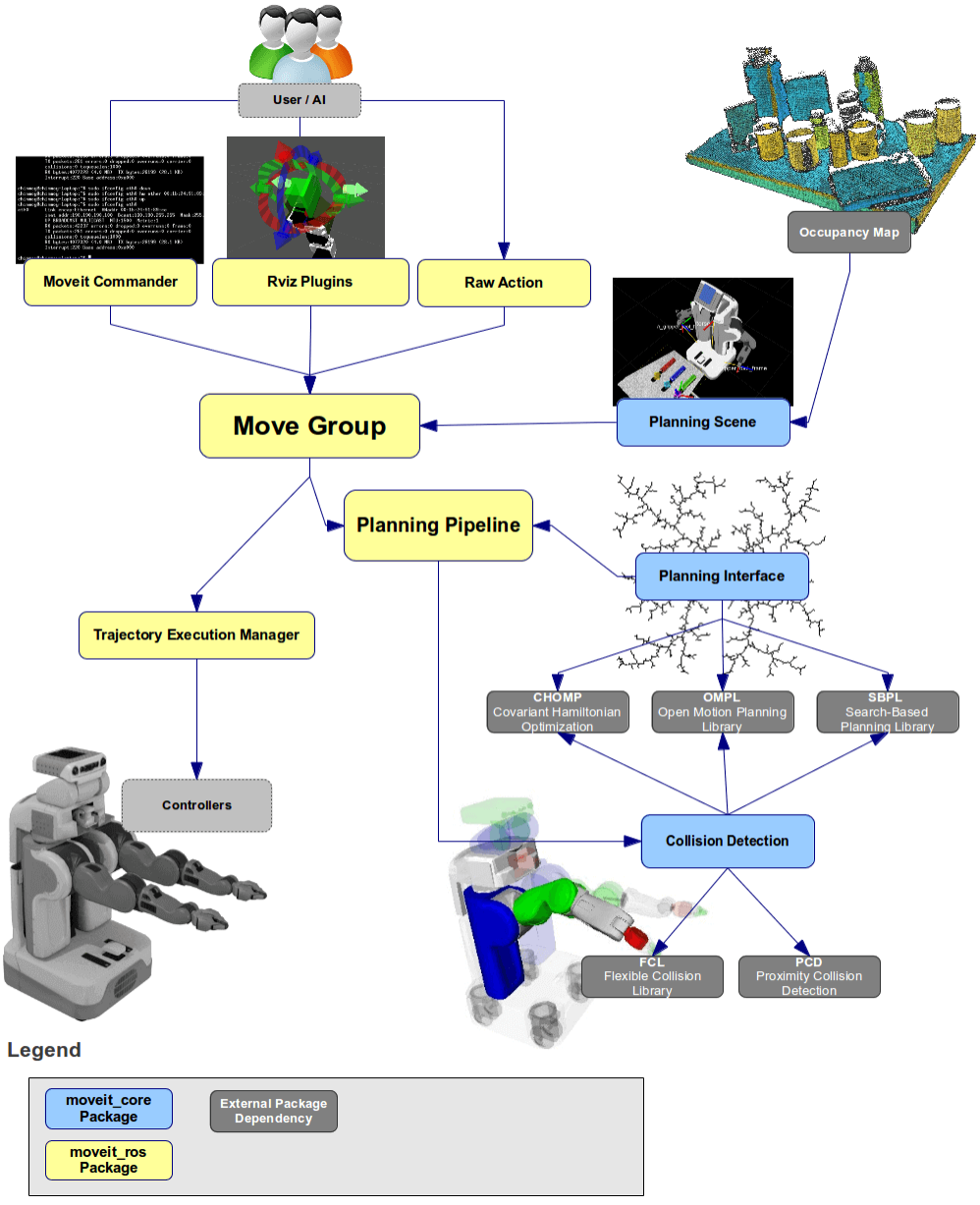


Fig. 2.8. – MoveIt structure

The pipeline is:

1. receiving the action from outside (operator, software call, etc.);
2. call chosen motion planner to build appropriate motion;
3. execute the motion on the arm via robot controllers.

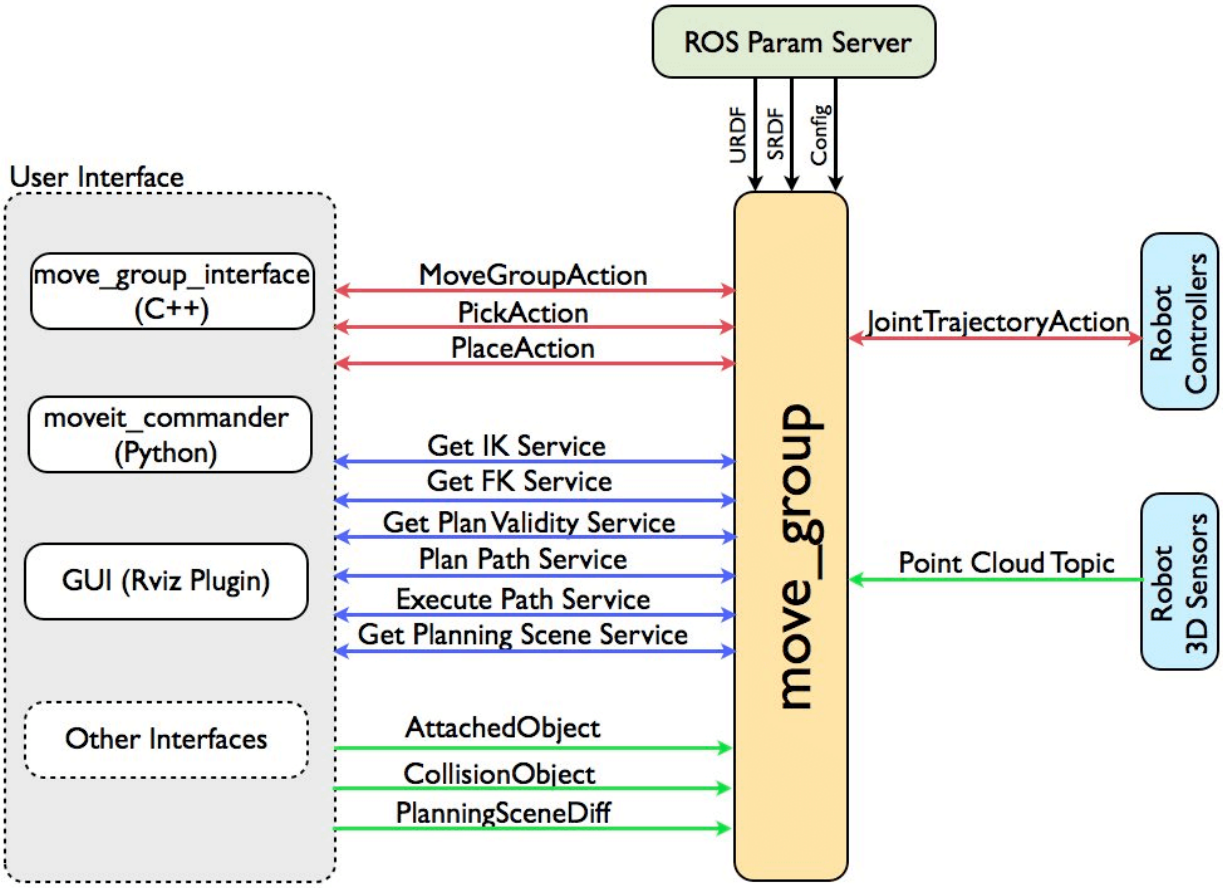


Fig. 2.9. move_group interaction

2.9.1. Configuration

To use MoveIt! With any robotics arm, a configuration is needed. A robot arm developer may provide it, or you can do it with Setup Assistant.

URDF configuration for MoveIt! Should specify meshed for collision checking. It is possible to provide meshes for visualization and or collision checking separately in URDF. You can provide detailed visualization meshes, but collision meshes should not be very detailed. The performance of checking collisions for robot link depends on

the number of triangles of collision meshes. The number of triangles in the robot description should be less than a few thousand [41].

The SRDF or Semantic Robot Description Format complements the URDF. It describes default robot configurations, joint groups, additional collision checking information, and additional transforms that may be needed to specify the robot's pose completely. The recommended way to generate an SRDF is by using the MoveIt! Setup Assistant.

Usually, URDF describes information only about the physical joints of the robot. To define the pose of the root link on the robot accordingly to a world coordinate system, you often may need additional joints. In this case, to specify this connection. A virtual joint is used. For example, a mobile robot like the PR2, which can move around in the plane, is specified using a planar virtual joint that attaches the world coordinate frame to the robot's frame. If a robot is fixed (like an industrial manipulator), it should be attached to the world using a fixed joint.

A group (sometimes called JointGroup or Planning Group) is a central MoveIt! Concept. MoveIt! always applies an action on a particular group. The only joints MoveIt! will use are the joints in the group for planning. Other joints will be left stationary. (To build a motion plan where all joints in the robot may move, creating a group of all joints is necessary.) A group is simply a collection of links and joints. Each group can be defined in one of the following different ways:

- Collection of Joints. The group automatically includes all the child links of each joint.
- Collection of Links. The group automatically includes all the parent links. All the parent joints of the links are also included in the group.
- Serial Chain. A serial chain is specified with the tip link and the base link the tip link. The last chain of the chain is called the tip link. The parent link for the first joint in a chain is called the base link.
- Collection of Sub-Groups. A collection of groups can be assigned to another group. E.g., you can define as two groups, right_arm and left_arm, and then you can define a new group named both_arms that includes these two groups.

Certain groups in a robot can be given a special designation as an end-effector. An end-effector is typically connected to another group (like an arm) through a fixed joint. There should be no common links between the end-effector and the parent group it is connected to. It is necessary to check it when specifying groups.

In Setup Assistant, the Default Self-Collision Matrix Generator looks for pairs of links on the robot definition that can be safely disabled from collision checking to decrease motion planning processing time. Such pairs of links will be disabled when they are:

- always in the collision;
- in collision in the robot's default position;
- when they are adjacent to each other on the given kinematic chain.

The sampling density determines the number of random positions of the robot to check for self-collision. With higher densities, more computation time is required, but on the other hand, lower densities give a higher possibility of disabling pairs that should not be disabled. The default value is 10,000 collision checks. Collision checking is done in parallel to reduce processing time.

2.9.2. Move Group interactions

move_group talks to the robot through ROS topics and actions. It communicates with the robot to get current state information (positions of the joints, etc.), get the point clouds and other sensor data from the robot sensors, and to talk to the controllers on the robot.

move_group listens on the /joint_states topic for determining the current state information - i.e., determining where each joint of the robot is. move_group is capable of listening to multiple publishers on this topic even if they are publishing only partial information about the robot state (e.g., separate publishers may be used for the arm and mobile base of a robot). Note that move_group will not set up its own joint state publisher - this is something that has to be implemented on each robot.

move_group monitors transform information using the ROS TF library. This allows the node to get global information about the robot's pose (among other things). E.g., the ROS navigation stack will publish the transform between the map frame and base frame of the robot to TF. move_group can use TF to figure out this transform for internal use. Note that move_group only listens to TF. To publish TF information from your robot, you will need to have a robot_state_publisher node running on your robot.

move_group talks to the controllers on the robot using the FollowJointTrajectoryAction interface. This is a ROS action interface. A server on the robot needs to service this action - this server is not provided by move_group itself. move_group will only instantiate a client to talk to this controller action server on your robot.

move_group uses the Planning Scene Monitor to maintain a planning scene, which is a representation of the world and the current state of the robot. The robot state can include any objects carried by the robot, which are considered to be rigidly attached to the robot. More details on the architecture for maintaining and updating the planning scene are outlined in the Planning Scene section below.

move_group is structured to be easily extensible - individual capabilities like pick and place, kinematics, motion planning are actually implemented as separate plugins with a common base class. The plugins are configured using ROS through a set of ROS YAML parameters and through the use of the ROS pluginlib library. Most users will not have to configure move_group plugins since they come automatically configured in the launch files generated by the MoveIt Setup Assistant [41, 42].

2.9.3. Motion planning

MoveIt works with motion planners through a plugin interface. This allows MoveIt to communicate with and use different motion planners from multiple libraries, making MoveIt easily extensible. The motion planners' interface is through a ROS Action or service (offered by the move_group node). The default motion planners for

move_group are configured using OMPL and the MoveIt interface to OMPL by the MoveIt Setup Assistant.

The motion plan request clearly specifies what you would like the motion planner to do. Typically, you will be asking the motion planner to move an arm to a different location (in joint space) or the end-effector to a new pose. Collisions are checked for by default (including self-collisions). You can attach an object to the end-effector (or any part of the robot), e.g. if the robot picks up an object. This allows the motion planner to account for the motion of the object while planning paths. You can also specify constraints for the motion planner to check - the inbuilt constraints provided by MoveIt are kinematic constraints:

- Position constraints - restrict the position of a link to lie within a region of space
- Orientation constraints - restrict the orientation of a link to lie within the specified roll, pitch, or yaw limits
- Visibility constraints - restrict a point on a link to lie within the visibility cone for a particular sensor
- Joint constraints - restrict a joint to lie between two values
- User-specified constraints - you can also specify your own constraints with a user-defined callback.

The move_group node will generate the desired trajectory in response to your motion plan request. This trajectory will move the arm (or any group of joints) to the desired location. Note that the result coming out of move_group is a trajectory and not just a path - _move_group will use the desired maximum velocities and accelerations (if specified) to generate a trajectory that obeys velocity and acceleration constraints at the joint level.

The complete motion planning pipeline chains together a motion planner with other components called planning request adapters (Fig. 2.10). Planning request adapters allow for pre-processing motion plan requests and post-processing motion plan responses. Pre-processing is useful in several situations, e.g., when a start state for the robot is slightly outside the robot's specified joint limits. Post-processing is needed

for several other operations, e.g., converting paths generated for a robot into time-parameterized trajectories. MoveIt provides a set of default motion planning adapters that each perform a particular function [42].

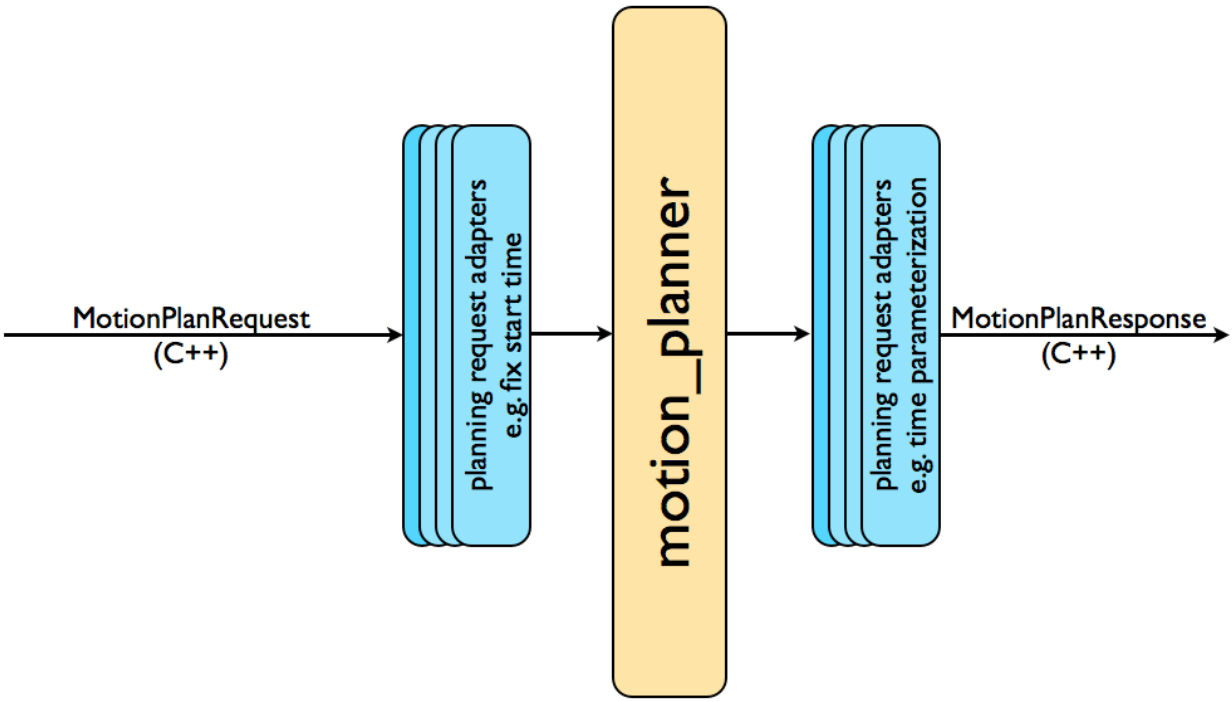


Fig. 2.10. The motion planning pipeline

The fixed start state bounds adapter fixes the start state within the joint limits specified in the URDF. This adapter's need arises when the joint limits for the physical robot are not correctly configured. The robot may then end up in a configuration where one or more of its joints is slightly outside its joint limits. In this case, the motion planner cannot plan since it will think that the starting state is outside joint limits. The "FixStartStateBounds" planning request adapter will "fix" the start state by moving it to the joint limit. However, this is obviously not the right solution every time - e.g., where the joint is really outside its joint limits by a large amount. The adapter parameter specifies how much the joint can be outside its limits to be "fixable".

The fix workspace bounds adapter will specify a default workspace for planning: a cube of size 10 m x 10 m x 10 m. This workspace will only be specified if the planning request to the planner does not have these fields filled in.

The fix starts state collision adapter will attempt to sample a new collision-free configuration near a specified configuration (in a collision) by perturbing the joint values by a small amount. The amount that it will perturb the values by is specified by a "jiggle_factor" parameter that controls the perturbation as a percentage of the joint's entire range of motion. The other parameter for this adapter specifies how many random perturbations the adapter will sample before giving up [43].

This adapter is applied when the start state for a motion plan does not obey the specified path constraints. It will attempt to plan a path between the robot's current configuration to a new location where the path constraint is obeyed. The new location will serve as the start state for planning.

The motion planners will typically generate "kinematic paths", i.e., paths that do not obey any velocity or acceleration constraints and are not time parameterized. This adapter will "time parameterize" the motion plans by applying velocity and acceleration constraints.

There are many planners included in MoveIt. All of them are part of OMPL (Fig. 2.11., 2.12.). We need to choose the best planners to use. A comparison of them is made by Kajane Thinakaran in her thesis. RRTConnect, SBL, FMT, and BiTRRT were shortlisted as the primary planners for the Sawyer arm. RRTConnect and SBL are the fastest planners. FMT gives the best path qualities, while BiTRRT has a very high solution probability in high-obstacle-density environments. The TRAC-IK kinematics plugin is recommended as it generally shows a better performance than KDL or LMA [44].

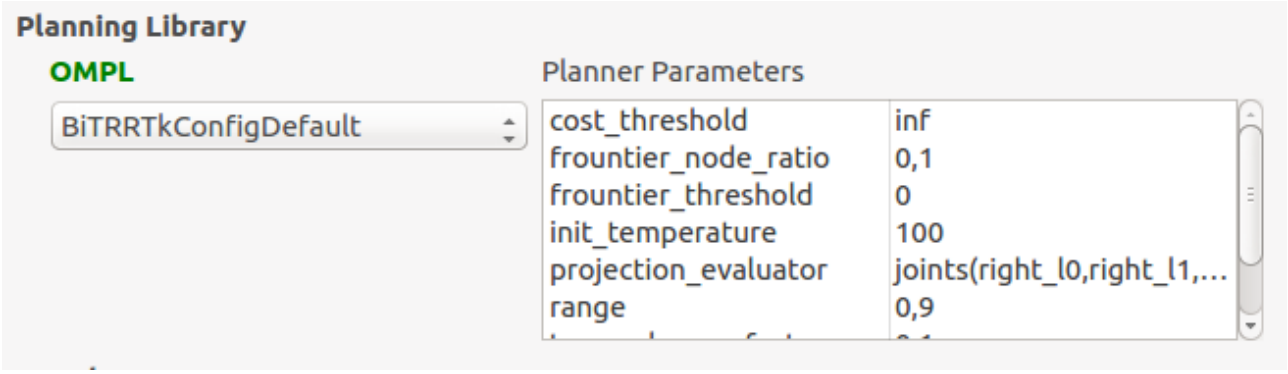


Fig. 2.11. Choosing of planner

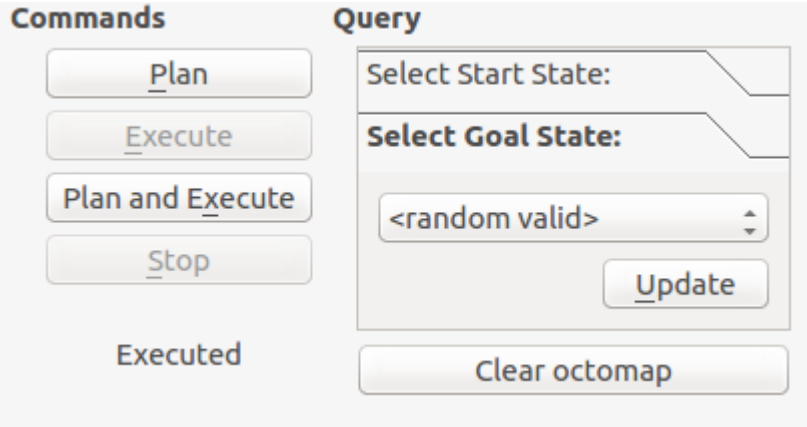


Fig. 2.12. Rviz commands

2.9.4. Environment perception

The planning scene is used to represent the world around the robot and stores the robot's state. It is maintained by the planning scene monitor inside the move group node. The planning scene monitor listens to:

- State Information: on the joint_states topic
- Sensor Information: using the world geometry monitor described below
- World geometry information: from user input on the planning_scene topic (as a planning scene diff).

The world geometry monitor builds world geometry using information from the sensors on the robot and from user input. It uses the occupancy map monitor described below to build a 3D representation of the environment around the robot and augments that with information on the planning_scene topic for adding object information.

The Occupancy map monitor uses an Octomap to maintain the occupancy map of the environment. The Octomap can actually encode probabilistic information about individual cells, although this information is not currently used in MoveIt. The Octomap can directly be passed into FCL, the collision checking library that MoveIt uses.

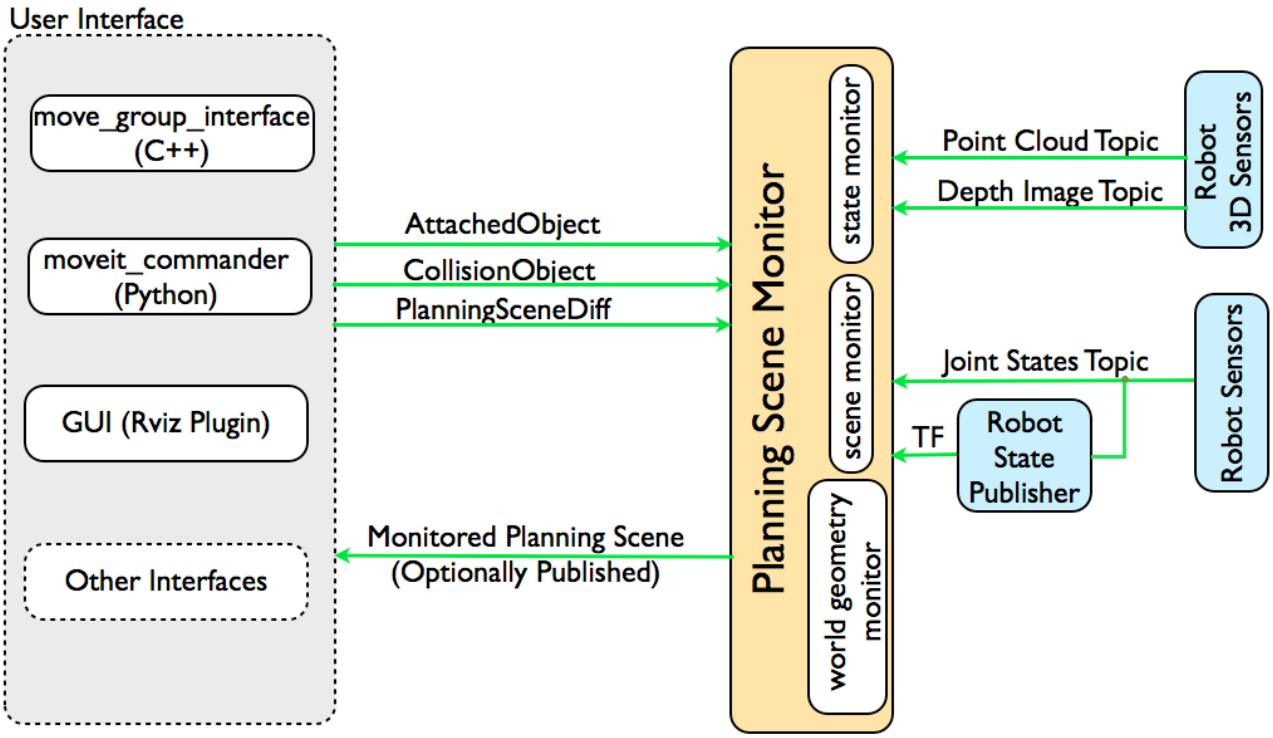


Fig. 2.13. Planning Scene Monitor

2.9.5. Collision checking

Collision checking in MoveIt is configured inside a Planning Scene using the CollisionWorld object. Fortunately, MoveIt is set up so that users never really have to worry about how collision checking is happening. Collision checking in MoveIt is mainly carried out using the FCL package - MoveIt's primary CC library [45].

MoveIt supports collision checking for different types of objects, including:

- Meshes
- Primitive Shapes - e.g., boxes, cylinders, cones, spheres, and planes
- Octomap - the Octomap object can be directly used for collision checking

Collision checking is a very expensive operation, often accounting for close to 90% of the computational expense during motion planning. The Allowed Collision Matrix or ACM encodes a binary value corresponding to the need to check for collision between pairs of bodies (which could be on the robot or in the world). If the value corresponding to two bodies is set to 1 in the ACM, this specifies that a collision check

between the two bodies is not needed. This would happen if, e.g., the two bodies are always so far away that they would never collide with each other.

2.10. Visualization in Rviz

Rviz is a highly extensible 3D visualization tool widely used in ROS development. Although the main task for the tool is visualization, it also can be an input tool.

The MoveIt! Rviz plugin allows us to set up virtual environments (scenes), create start and goal states for the robot interactively, test various motion planners, and visualize the output.

2.10.1. MoveIt! Rviz plugin

The easiest way to get started using MoveIt! is using its RViz plugin. This tool is the primary visualizer in ROS and a handy tool for debugging robotics. The MoveIt! Rviz plugin allows you to set up virtual environments (also called scenes), create start and goal states for the robot interactively, test all motion planners, and visualize the built path. The most useful parts of the plugin are listed below. With interface on Fig. 2.14. you can choose a planner and set its parameters. On interface in Fig. 2.15. there are motion planning, we can set the goal to achieve.

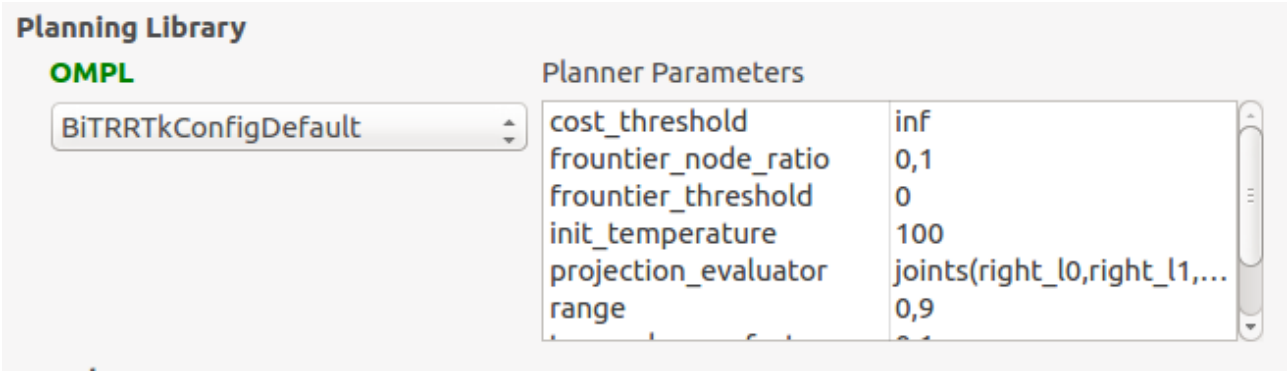


Fig. 2.14. Choosing of planner

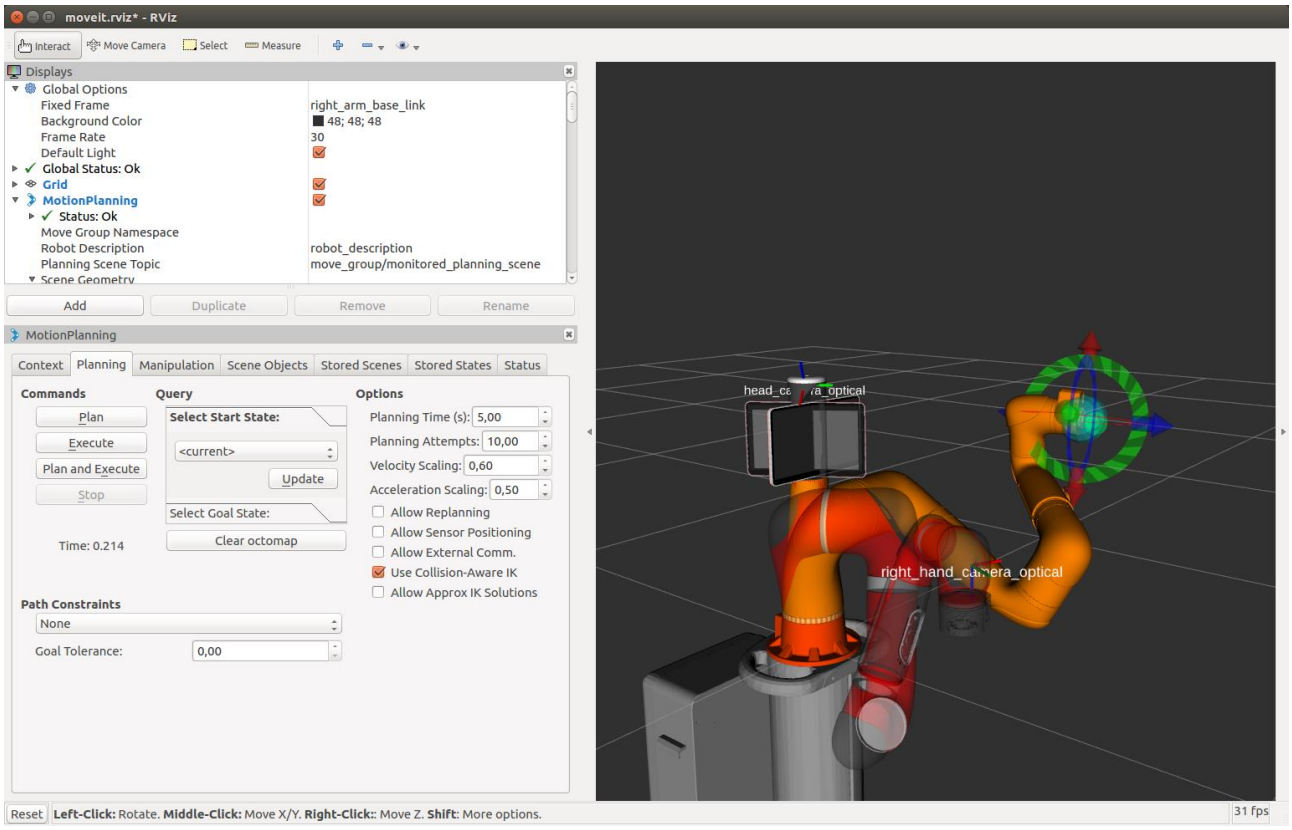


Fig. 2.15. Rviz commands

There are four different overlapping visualization approaches:

– The robot's configuration in the planning scene planning environment (is active by default).

– The planned path for the robot (default setup).

– Green ghost: The start state for motion planning (disabled by default).

– Orange ghost: The goal state for motion planning (is active by default).

Each of these visualizations can be toggled on and off using checkboxes:

– The planning scene robot using the Show Robot Visual checkbox in the Scene Robot tab.

– The planned path using the Show Robot Visual checkbox in the Planned Path tab.

– The start state using the Query Start State checkbox in the Planning Request tab.

– The goal state using the Query Goal State checkbox in the Planning Request tab.

2.11. Simulation in Gazebo

Gazebo (Fig. 2.16.) is a 3D dynamic simulation software with the ability to simulate robots' populations accurately and efficiently in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs.

Typical uses of Gazebo include:

- testing robotics algorithms,
- designing robots,
- performing regression testing with realistic scenarios

A few critical features of Gazebo include:

- a rich library of robot models and environments,
- several physics engines,
- a wide variety of sensors emulation,
- convenient programmatic and graphical interfaces.

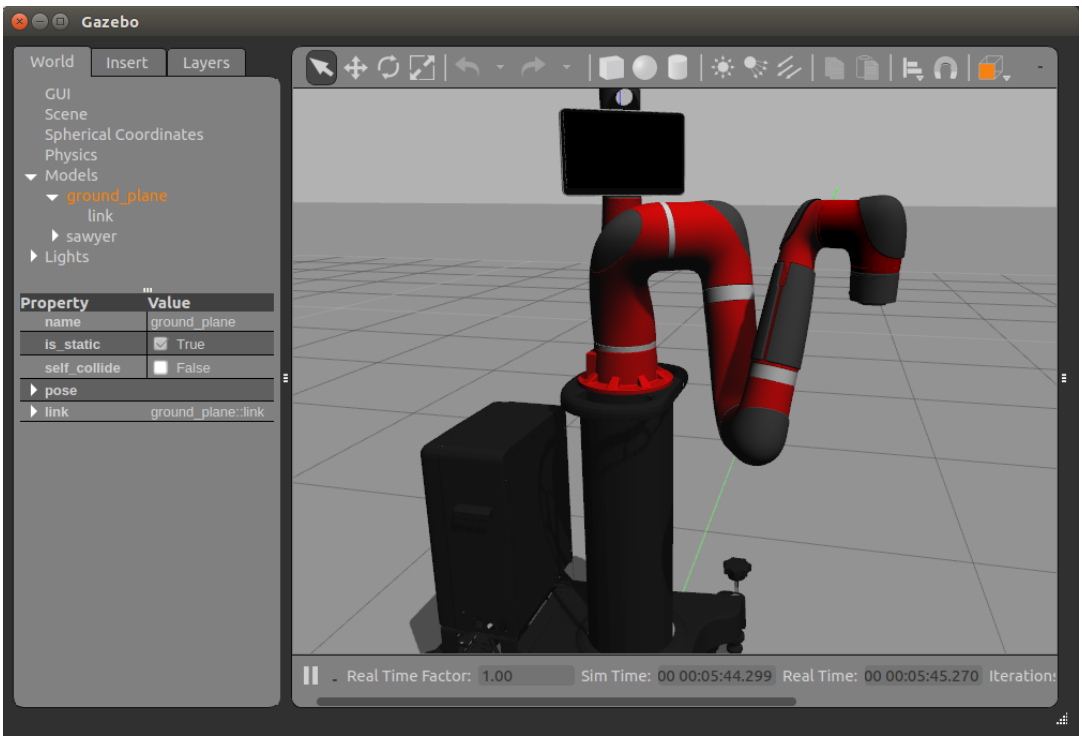


Fig. 2.16. Gazebo interface

2.12. Slip detection with ToF and Tactile sensor using Deep Learning

To implement a robust method of grasping the object, we have chosen two types of sensors – ToF and tactile sensor.

ToF sensor is used to recognize object presence between the gripper. Although GPD may provide accurate enough poses to grasp, it can add additional precision. This approach also decreases the requirements for calibration accuracy. ToF sensor could also be used for slip detection, but it was considered excessive during the research. Data from the tactile sensor is enough to detect a slip securely. We use VL6180X Time-of-Flight Distance Sensor from Pololu.

The easiest way to achieve haptic perception is to use a tactile sensor. This can be used to record the change of capacity or resistance of the gripper's fingers. We use Force-Sensing Linear Potentiometer from Interlink Electronics.

The approach is described in Atmaraaj Gopal's thesis [46]. The approach is to use machine learning to detect a slip.

To recognize the slip, there two general approaches: strict algorithm and machine learning. We used the second one. Values from the tactile sensors are directly processed with LTSM neural network. Without any additional filtering. The advantages of such an approach:

- It is easy. We don't need to develop an algorithm. We don't even really need filtering before. So it's decreases complexity.
- It does not need adjustments for other grippers. We can even use the same model on different grippers.

We trained a model using a stable grasp and slip state. Slip data is obtained by sliding the gripper along with the cylinder (Fig.2.17.). The average sliding velocity values are 25mm/s, 100mm/s, and 500mm/s, which is done on cardboard, aluminum, and plastic. When the gripper slides along the object, it produces fluctuations of signal, which should be way bigger than values when the grasp is stable.

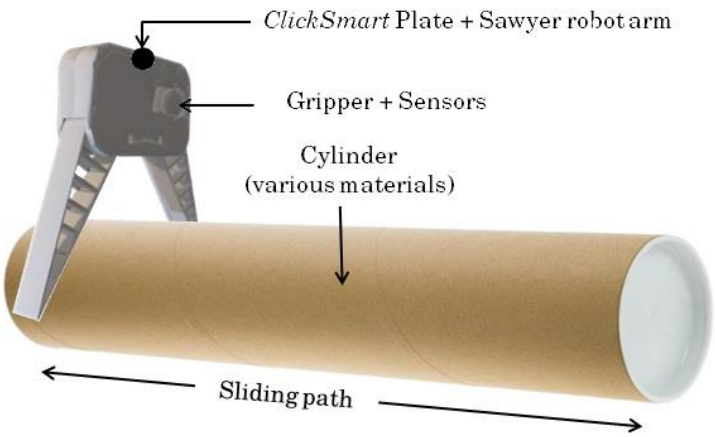


Fig.2.17. Data Acquisition

Stable (non-slip) data is done by stable gripping of the object when the object is not moved relative to the gripper.

So that we have a dataset with values of the movement and when there is no movement. It can be directly used to train NN.

2.13. Conclusions to the second section

The purpose is the development of the arm controlling algorithm. This section explains the chosen approaches and implementations used to develop the system. All areas of arm control are covered.

This study describes a used robot called LeonaRT, developed by the RT-Lions team. The robot was built for participation in the RoboCup championship. The basement of the robot is an omnidirectional moving platform Neobotix MPO-700 (Fig. 2.1.). Its four Omni-Drive-Modules enable it to move exceptionally smoothly in any direction. This robot is even capable of rotating freely while driving to its destination. The Omni-Drive-Modules of the MPO-700 feature essential benefits compared to other omnidirectional drive kinematics.

During the research, the team was able to use Sawyer Arm made by Rethink Robotics. It is 7 degrees of freedom robotics arm. It is provided with built-in software Intera.

ROS is a platform for robot development. It provides a framework for robot software communication and a variety of built-in software. ROS framework is organized as many nodes that work in separate processes. A node can send and receive messages through topics; provide services; call services provided by another node; set and receive data in parameter server.

The Intel RealSense depth camera D435 is a stereo solution, offering quality depth for various applications. It is a wide field of view that is perfect for applications such as robotics or augmented and virtual reality, where the most important part is seeing as much as possible. With a range of up to 10m, this small form factor camera can easily be integrated into any solution and comes complete with Intel RealSense SDK 2.0 and cross-platform support.

The density of the produced point cloud could be too big to process it further. Also, the camera can capture objects which are out of the available workspace of the arm. Both problems lead to additional unnecessary computation. To solve these problems, we use PCL – Point Cloud Library. It is a set of algorithms for point cloud data, including filtering, feature estimation, surface reconstruction, registration, model fitting, and segmentation.

One family of object detection approaches is YOLO what stands for You Only Look Once. This approach's advantage is speed; it is usually faster than another popular R-CNN approach, although it is less accurate.

Before grasping any object, the object should be recognized and chosen. It is a problem in robotics arm control to solve. There is an implementation of the second approach – the Grasp Pose Detection (GPD) package. It provides 6-DOF grasp poses for objects. It means that an object can be grasped from any side with any orientation if the arm can provide such a pose. At the moment, it supports only two-finger parallel grippers, which is appropriate for the gripper we use. Besides, the gripper is not a parallel jaw gripper, rather a scissors-type.

The OctoMap library implements a 3D occupancy grid mapping approach by providing data structures and mapping algorithms in C++, especially suitable for robotics. The implementation of the map is based on an octree.

MoveIt is an easy-to-use open-source robot manipulation platform for commercial application development, prototyping, and testing algorithms.

Rviz is a highly extensible 3D visualization tool widely used in ROS development. Although the main task for the tool is visualization, it also can be an input tool. The MoveIt! Rviz plugin allows us to set up virtual environments (scenes), create start and goal states for the robot interactively, test various motion planners, and visualize the output.

Gazebo (Fig. 2.16.) is a 3D dynamic simulation software with the ability to simulate robots' populations accurately and efficiently in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs.

SECTION 3
RESULTS OF THE DEVELOPED SOLUTION

3.1. System setup

The developed system is tested on ROS Kinetic, which requires Ubuntu 16.04.

3.1.1. Simulation

The first step is adding the ROS repository.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

If the command above does not work, you can use the next command.

Alternatively, you can use curl instead of the apt-key command, which can be helpful if you are behind a proxy server.

```
curl -
sSL 'http://keyserver.ubuntu.com/pks/lookup?op=get&search=0xC1CF6E31E6BADE8868B172B4F42ED6FBAB17C654' | sudo apt-key add -
```

Make sure that the system is up to date

```
sudo apt-get update
```

Then on the development machine, we use a full installation

```
sudo apt-get install ros-kinetic-desktop-full
```

The ROS environment should set all variables automatically to each bash session.

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Now we have basic ROS functionality, but we need additional tools to manage workspaces and packages. There are many tools and requirements that are distributed separately. For example, rosinstall is a frequently used command-line tool that enables you to easily download many source trees for ROS packages with one command.

```
sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python-wstool build-essential
```

To be able to use many ROS tools, we will need to initialize rosdep. It enables us to install system dependencies if we want to compile packages from the source, and it is required to run some core components in ROS.

```
sudo apt install python-rosdep
```

The following step is rosdep initialization

```
sudo rosdep init
rosdep update
```

Then next packages should be installed to work with the developed solution.

```
sudo apt-get update
sudo apt-get install git-core python-argparse python-wstool python-vcstools
python-rosdep ros-kinetic-control-msgs ros-kinetic-joystick-drivers ros-kinetic-
xacro ros-kinetic-tf2-ros ros-kinetic-rviz ros-kinetic-cv-bridge ros-kinetic-
actionlib ros-kinetic-actionlib-msgs ros-kinetic-dynamic-reconfigure ros-
kinetic-trajectory-msgs ros-kinetic-rospy-message-converter
sudo apt-get install gazebo7 ros-kinetic-qt-build ros-kinetic-gazebo-ros-control
ros-kinetic-gazebo-ros-pkgs ros-kinetic-ros-control ros-kinetic-control-toolbox
ros-kinetic-realtime-tools ros-kinetic-ros-controllers ros-kinetic-xacro python-
wstool ros-kinetic-tf-conversions ros-kinetic-kdl-parser ros-kinetic-sns-ik-lib
sudo apt-get install ros-kinetic-moveit
sudo apt-get install ros-kinetic-gazebo-ros-pkgs ros-kinetic-gazebo-ros-control
```

Then we need to install Realsense plugin for Gazebo. We compile it from sources. To compile it we need to create a build folder and make using CMAKE as follows:

```
mkdir build
cd build
cmake
make
```

The plugin binaries will be installed so that Gazebo finds them. Also, the needed models will be copied to the default gazebo models folder.

```
make install
```

Then we need configured packages in the workspace. Where {workspace} is a path to the workspace.

```
cd {workspace}
git clone https://github.com/RethinkRobotics/intera_sdk.git
git clone https://github.com/RethinkRobotics/intera_common.git
git clone -b obstacle_avoidance
https://github.com/ViacheslavRud/sawyer_robot.git
git clone -b obstacle_avoidance
https://github.com/ViacheslavRud/sawyer_simulator.git
git clone -b obstacle_avoidance
https://github.com/ViacheslavRud/sawyer_moveit.git
git clone -b obstacle_avoidance https://github.com/ViacheslavRud/realsense-
ros.git
cd
catkin_make
```

In case if any errors with the Realsense library version, this line

```
find_package(realsense2 2.35.2)
```

to this

```
find_package(realsense2 2.34)
```

3.2. Running

3.2.1. Simulation

To run the simulation of the motion planning pipeline, you need to run the simulated world in Gazebo. The following command runs it.

```
roslaunch sawyer_gazebo sawyer_world.launch
```

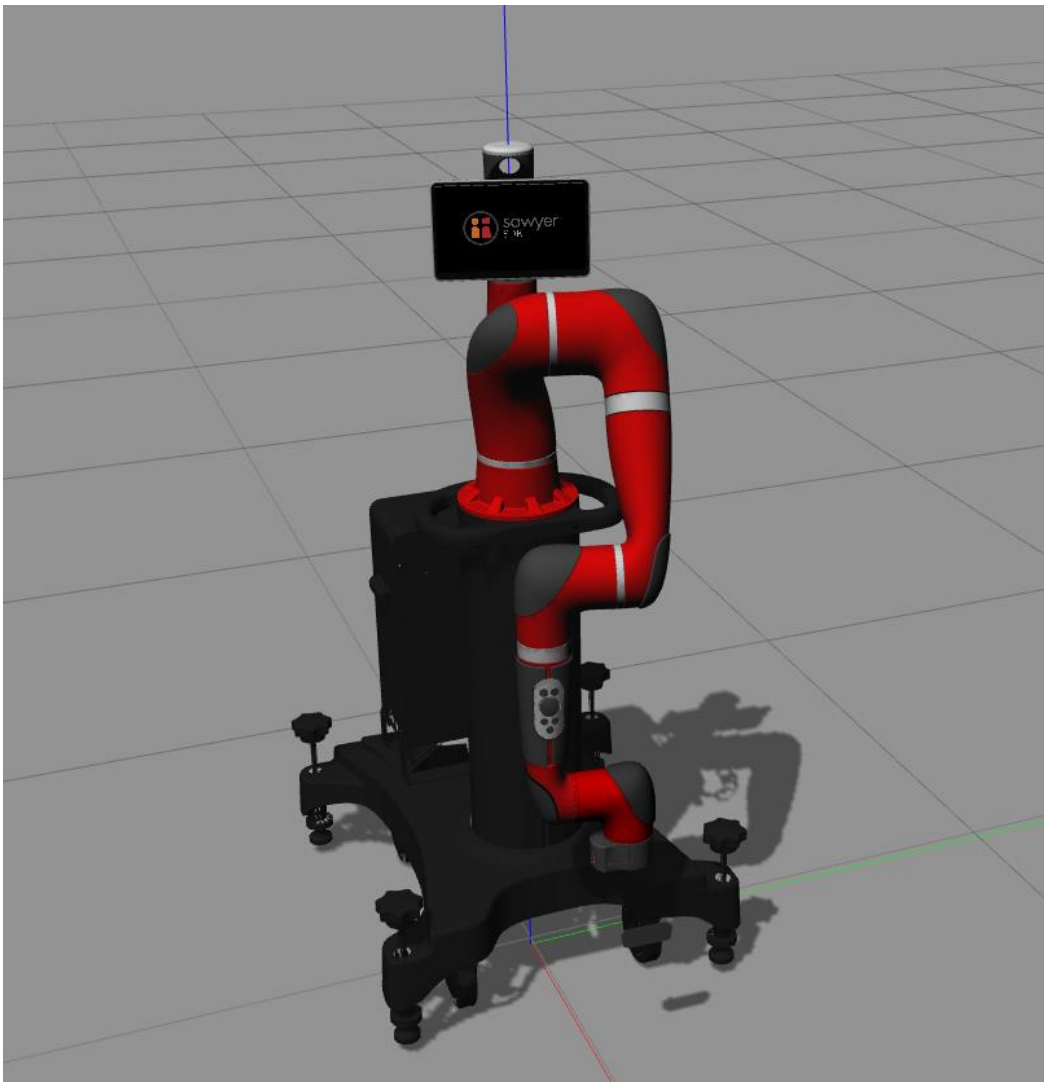


Fig. 3.1. Simulated robot in the initial pose

Then to move the manipulator to the appropriate starting position, the next command is useful. It should be run in the second terminal.

```
rosrun intera_examples joint_torque_springs.py
```

It should be disabled after the robot took the appropriate position.

```
rosrun intera_interface joint_trajectory_action_server.py
```

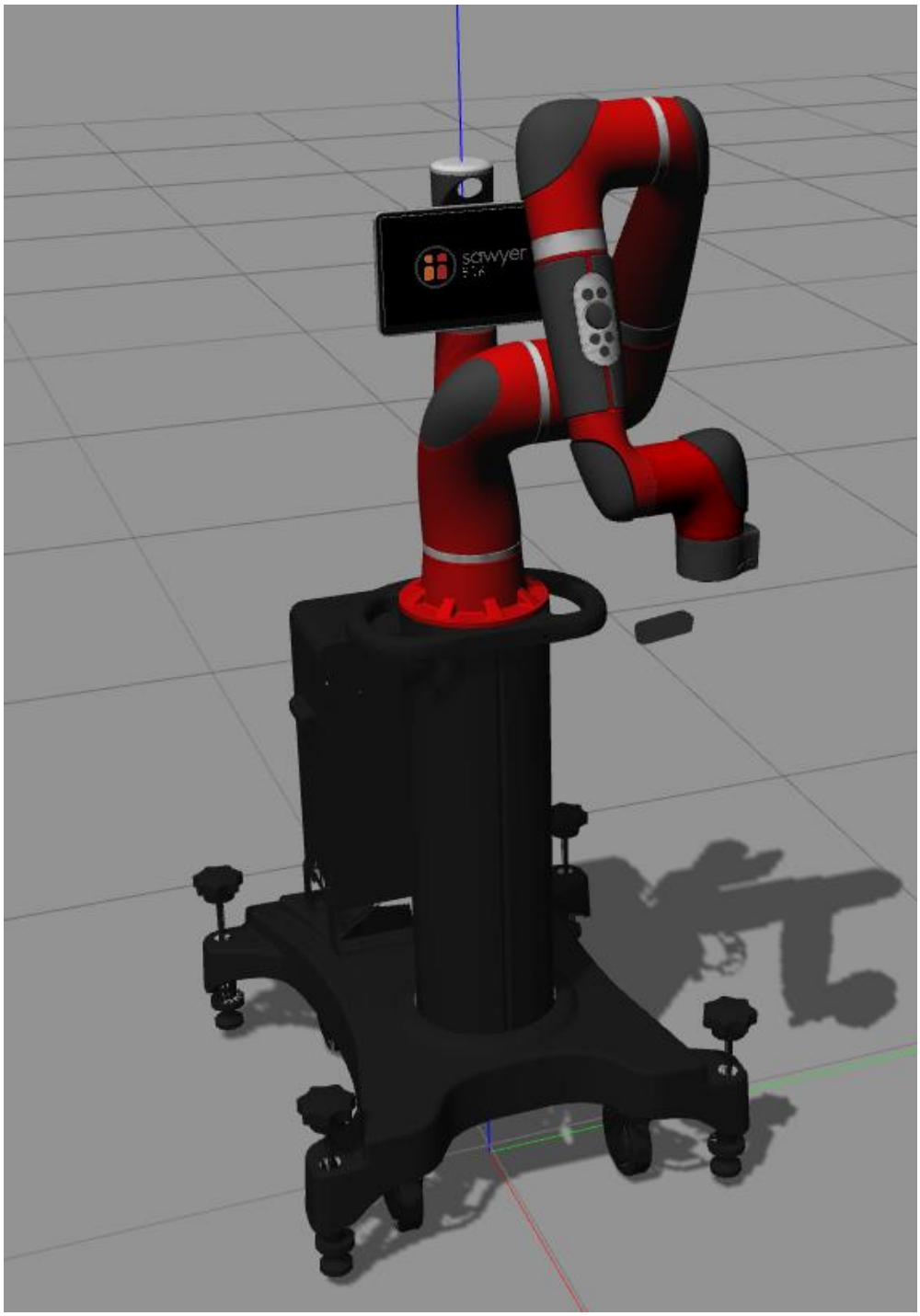


Fig. 3.2. Robot in a good starting position

At the next step, a new terminal should run the following command. This command runs the MoveIt interface in Rviz we can use to control the robot manually. As a result, Rviz should be opened and show the same as in the image below (Fig. 3.3.).

```
roslaunch sawyer_moveit_config sawyer_moveit.launch
```

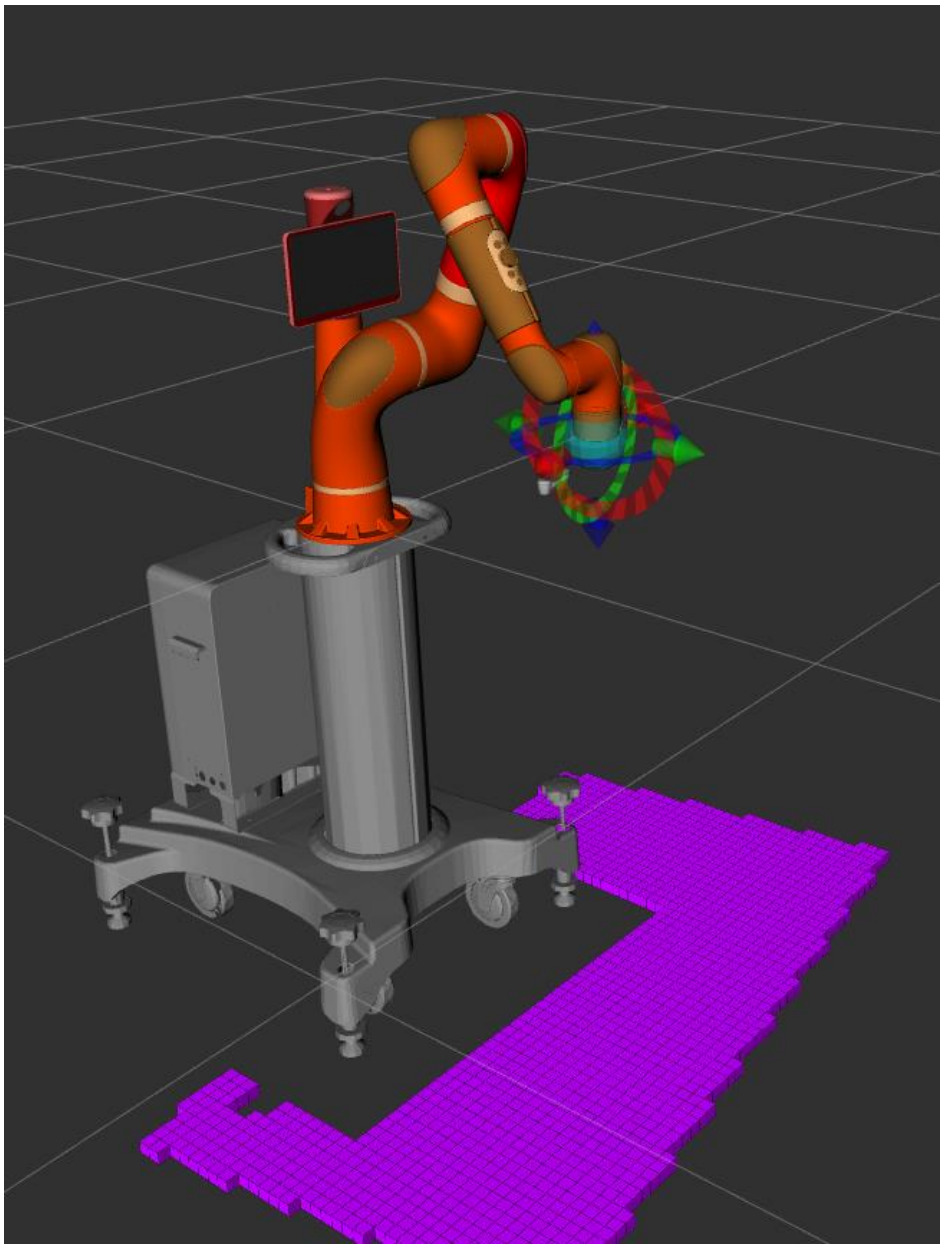


Fig. 3.3. Simulation with octree

Then RViz will show the planning interface provided with the Moveit plugin. Here we can set the parameters we need.

Alternatively, it can be done with a configuration file located at catkin_ws/sawyer_moveit_config/config and then loaded in the NAME_planning_pipeline.launch file. When the system is set-up on the robot, it should use the configuration file.

The gripper is not represented in the simulation. We need to test it on the real hardware.

3.2.2. Real robot

First, the operator's pc should be connected to the robot network, and the virtual environment should be set to leonart. Then setup is similar to the simulation. The user should run the next commands in individual terminals.

```
rosrun intera_examples joint_torque_springs.py
rosrun intera_interface joint_trajectory_action_server.py
rosrun intera_interface enable_robot.py -e
```

Then we may need to setup the gripper.

1. Connect gripper (Hand 2)
2. Check port with Arduino IDE
3. Connect hand camera
4. Start Remmina Remote Desktop Client
5. Connect to dutchman.local (LeonART)
6. Start camera node on LeonartPC

```
rosenv sawyer && roslaunch realsense2_camera rs_camera.launch
enable_pointcloud:=true serial_no:=844212070164'
```

7. Check the topic published on the DesktopPC with 'rostopic list | grep camera', topics with /camera* must be visible

Grasping Pipeline (/home/deep/RTL/moveordie_ws/src/itchy_hands/scripts/pick_and_place.py)

1. Navigate to grasping package, `'ws && cd itchy_hands'`
2. Open folder in VS Code, 'code'
3. View itchy_hands/scripts/pick_and_place.py
4. Get current end/effector pose with `'rosrun tf tf_echo right_arm_base_link gripper_tcp'`
5. Get current joint states with `'rostopic echo /joint_states'`
6. Replace values in Python file accordingly (set workspace points, init and detection joint_states)
7. Run file or (optional) set breakpoints in Python code and hit F5 to run.

3.3. Conclusions to the third section

As a result, we have a complete system that can find and grasp an object. This has been accomplished with modern solutions in each area. The full structure is presented in appendix B, and the brief structure is in Fig. 3.5. below.

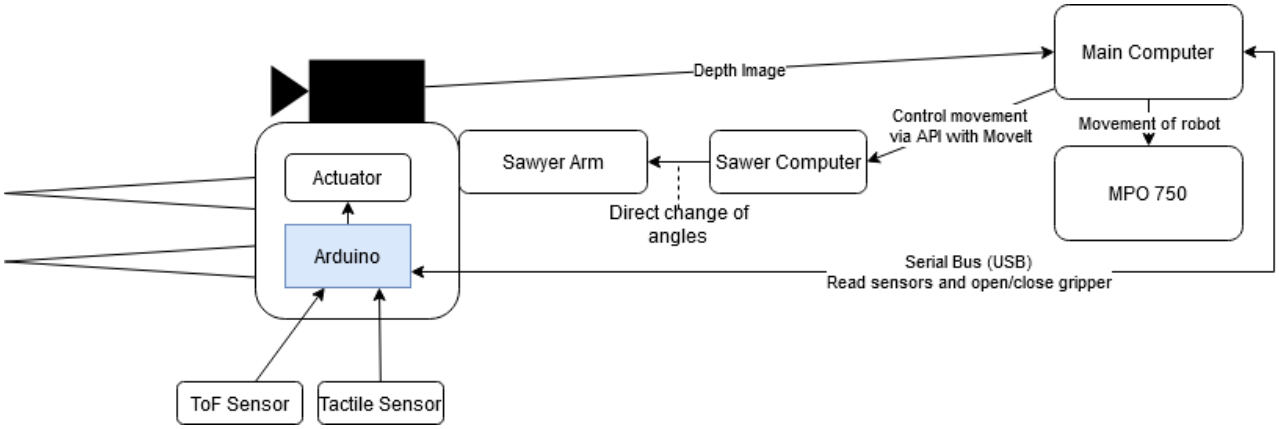


Fig. 3.5. The basic structure of the developed solution

The central component of the system is the main computer, which contains most of the developed software. The main computer is located in the body placed on the movable platform MPO 750. This computer has installed ubuntu 16.04. with ROS Kinetic.

The critical part is an end-effector represented by the gripper with fingers based on the Fin-Ray effect. They are done with TPU polymer. The movement of fingers is driven by a single actuator that gives the symmetrical motion of two fingers. There are two sensors on the gripper, ToF and tactile. They are used to recognize a secure grasp. Sensors and motors are connected to the Arduino board, which communicates with the main computer. The gripper is attached to Sawyer arm manipulator.

On top of the gripper, we have a Realsense camera, which provides pointcloud data that used to detect objects and grasp poses.

We need to define a workspace and filter the object from the environment since we need to grasp poses only for the target object. Filtering is done by YOLO and setting

the workspace in the GPD package. Then GPD package provides grasp poses for grasping.

When we have the poses, it is time to move the arm to achieve the required pose. Rethink Robotics provides Intera SDK can move the arm. It has built-in movement to cartesian pose, but it does not meet all expectations. Instead, we use the MoveIt framework, which is the industry standard for robotic arm control. It can avoid obstacles using Octomap generated from the point cloud we receive from the camera.

The developed algorithm (Fig. 3.6.):

1. Detect an object we need. It's done with YOLO. We need it to separate an object from the environment.
2. Mask detected object. In this step, we should remove excessive points from the point cloud.
3. Run GPD on masked point cloud to retrieve poses to grasp. At this step, we have several grasp poses, with the score of each pose. We select the pose with the best score.
4. We need pre and post grasp poses [43].
5. We move the gripper to pre grasp pose.
6. Move the arm to the object until it grasps.
7. When it approached the object, close the gripper.
8. Now the object can be taken to the post grasp pose, which is a bit farther from the grasp pose.
9. We continuously check if there is a slip as we move the arm to the post grasp pose. If an object has fallen – repeat grasping from the pre grasp pose (step 5). If the object is still in the gripper – move it further.
10. Check if there is a slip. If yes – increase the gripping force.

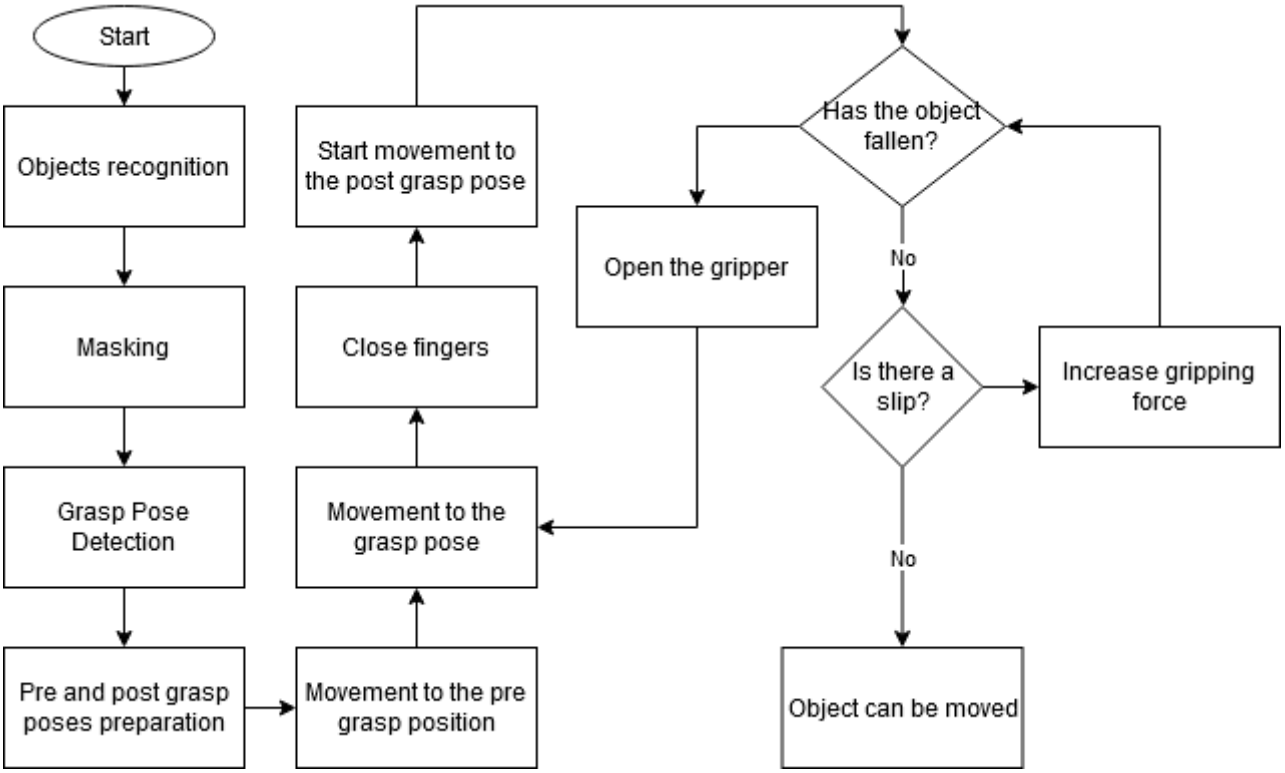


Fig. 3.6. The algorithm of grasping

The described approach is tested using simulation and real hardware. We have satisfactory results of grasping different objects. The developed method described in this study works as expected.

РОЗДІЛ 4
ЕКОНОМІКА

При розробці програмного забезпечення (ПЗ) важливими етапами є визначення трудомісткості розробки і розрахунок витрат на створення програмного продукту.

4.1. Визначення трудомісткості розробки програмного забезпечення

Задані дані:

1. передбачуване число операторів – 2300;
2. коефіцієнт складності програми – 1,7;
3. коефіцієнт корекції програми в ході її розробки – 0,09;
4. годинна заробітна плата програміста, грн/год – 165;
5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,05;
6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 0,85;
7. вартість машино-години ЕОМ, грн/год – 43.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д, \text{ людино-годин}, \quad (3.1)$$

де t_o - витрати праці на підготовку й опис поставленої задачі (приймається 50);

t_u - витрати праці на дослідження алгоритму рішення задачі;

t_a - витрати праці на розробку блок-схеми алгоритму;

$t_п$ - витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ - витрати праці на налагодження програми на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q * C * (1 + p), \tag{3.2}$$

де q - передбачуване число операторів;

C - коефіцієнт складності програми;

p - коефіцієнт корекції програми в ході її розробки.

$$Q = 2300 * 1,7 * (1 + 0.99) = 4262, \text{ людино-годин.} \tag{3.3}$$

Витрати праці на вивчення опису задачі t_u визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q * B}{(75..85) * k}, \text{ людино-годин,} \tag{3.4}$$

де B - коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі;

k - коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності.

$$t_u = \frac{4262 * 1,05}{45 * 0,85} = 71, \text{ людино-годин.} \quad (3.5)$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20..25) * k}, \text{ людино-годин,} \quad (3.6)$$

$$t_a = \frac{4262}{20 * 0,85} = 251, \text{ людино-година.} \quad (3.7)$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20..25 * k)}, \text{ людино-годин,} \quad (3.8)$$

$$t_n = \frac{4262}{25 * 0,85} = 201, \text{ людино-година.} \quad (3.9)$$

Витрати праці на налагодження програми:

– за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4..5) * k}, \text{ людино-годин,} \quad (3.10)$$

$$t_{отл} = \frac{4262}{5 * 1} = 853, \text{ людино-годин;} \quad (3.11)$$

– за умови комплексного налагодження завдання:

$$t_{отл}^{k} = 1,5 * t_{отл}, \text{ людино-годин,} \quad (3.12)$$

$$t_{отл}^{k} = 1,5 * 853 = 1280, \text{ людино-годин.} \quad (3.13)$$

Витрати праці на підготовку документації:

$$t_{д} = t_{др} + t_{до}, \text{ людино-годин,} \quad (3.14)$$

де $t_{др}$ - трудомісткість підготовки матеріалів і рукопису.

$$t_{др} = \frac{Q}{15..20 * k}, \text{ людино-годин,} \quad (3.15)$$

$$t_{др} = \frac{4262}{20 * 0,85} = 85, \text{ людино-година,} \quad (3.16)$$

$t_{до}$ - трудомісткість редагування, печатки й оформлення документації:

$$t_{до} = 0,75 * t_{др}, \text{ людино-годин,} \quad (3.17)$$

$$t_{до} = 0,75 * 251 = 189, \text{ людино-годин,} \quad (3.18)$$

$$t_д = 251 + 189 = 440, \text{ людино-годин.} \quad (3.19)$$

Тепер розрахуємо трудомісткість ПЗ:

$$t = 50 + 71 + 251 + 201 + 853 + 440 = 1866, \text{ людино-годин.} \quad (3.20)$$

4.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ *Кпо* включають витрати на заробітну плату виконавця програми *Зз/п* і витрат машинного часу, необхідного на налагодження програми на ЕОМ:

$$К_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн.} \quad (3.21)$$

Заробітна плата виконавців визначається за формулою:

$$З_{ЗП} = t * С_{пр}, \text{ грн,} \quad (3.22)$$

де *t* - загальна трудомісткість, людино-годин;

$C_{пр}$ - середня годинна заробітна плата програміста, грн/година.

$$З_{ЗП} = 1886 * 165 = 3078890, \text{ грн.} \quad (3.23)$$

Вартість машинного часу, необхідного для налагодження програми:

$$З_{мв} = t_{отл} * С_{мч}, \text{ грн,} \quad (3.24)$$

де $t_{отл}$ - трудомісткість налагодження програми на ЕОМ, год,

$C_{мч}$ - вартість машино-години ЕОМ, грн/год,

$C_{мч}$ = 35, грн/год.

$$З_{мв} = 853 * 43 = 36679, \text{ грн.} \quad (3.25)$$

Визначені в такий спосіб витрати на створення програмного забезпечення є частиною одноразових капітальних витрат на створення АСКП:

$$К_{по} = 307890 + 36679 = 344569, \text{ грн.} \quad (3.26)$$

Очікуваний період створення ПЗ:

$$Т = \frac{t}{B_k * F_p}, \text{ міс,} \quad (3.27)$$

де B_k - число виконавців (приймається 1),

F_p - місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

$$Т = \frac{1866}{1*176} = 10,3, \text{ міс.} \quad (3.28)$$

4.3. Маркетингові дослідження ринку збуту розробленого програмного продукту

У цьому підрозділі буде проаналізовано та досліджено ринок збуту та потенційних покупців розробленої системи, що здатна аналізувати навколишнє середовище, знаходити перешкоди, об’єкти для взаємодії, пози для хапання, а також планує рух маніпулятора до визначеної пози та визначає надійність хапання об’єкта.

Метою такого дослідження є реалізація створеного програмного продукту зацікавленим промисловим підприємствам з різних галузей промисловості де необхідно виконувати здачі які можуть бути автоматизовані за допомогою розробленого рішення.

В ході роботи була розроблена система, яка аналізує навколишнє середовище, знаходить перешкоди, об'єкти для взаємодії, позує для захоплення, планує рух маніпулятора до певного положення та забезпечує надійне схоплення об'єкта. Було виконано тестування системи, перевірка продуктивності та налаштування найкращих параметрів результатів. Отримана система була розроблена дослідницькою групою RT-Lions, факультет Technik, Університет Ройтлінгену. Дослідницький апаратний апарат включає камеру Intel Realsense, маніпулятор Sawyer Arm від Rethink Robotics та захоплюючий пристрій внутрішньої розробки.

Зараз використання маніпуляторів є важливим напрямом у робототехніці. Робототехніка широко використовується у різних областях. Вважається усталеним поділ всієї робототехніки на промислову (роботи-маніпулятори) і сервісну (персональні - наприклад, роботи-пилососи; професійні - дрони) галузі згідно прикладного призначенням.

Обидві галузі переживають зростання, проте причини цього глибоко різні. Промислова робототехніка зростає (в середньому на 15% в рік) за рахунок стрімкої роботизації китайської економіки і, в той час як зростання сервісної робототехніки має більш глибокі причини: велика частина світової економіки є

сервісною економікою. Саме тому сервісна робототехніка показує більш значне зростання вже зараз (на рівні 25% в рік) при щодо менших в абсолютному значенні цифрах у порівнянні з про-мислення.

Промисловий робот - це автоматично керований, багатоцільовий маніпулятор, запрограмований за трьома і більше осями. Він може бути або зафіксованим в заданому місці, або може мати можливість пересуватися для виконання промислових задач з автоматизації. Якщо трохи спростити термінологію, то промислова робототехніка - це все, що знаходиться в виробничому цеху; головним чином це різні маніпулятори. На сьогоднішній день це найпоширеніший вид роботів - всього в світі встановлено майже два мільйони промислових роботів.

91% всіх промислових роботів в 2016 році був встановлений в секторі обробної промисловості. В обробній промисловості галузі - лідери по покупкам роботів не змінюються вже більше п'яти років: це автомобільна промисловість і електроніка. Середньорічний темп зростання продажів по всіх галузях обробної промисловості в 2011-2016 роках склав 13%; для автомобілебудування - 12%, а для електронної промисловості - 19%.

Наведемо кілька прикладів операцій, які можуть виконуватися роботами:

- зварювання і пайка (дугова, точкова, лазерна, інше);
- розлив, напилення, дозування (забарвлення, емалювання, інше);
- обробка (різка, фрезерування, шліфування, інше);
- складання та розбирання (запресовування, монтаж, інше);
- переміщення і упаковка.

Одними з найбільш помітних виробників промислових роботів на міжнародному ринку є FANUC, Yaskawa, ABB, Festo, Sawer, Kuka, Rozum Robotics, Universal Robots, Denso тощо. На мою думку, саме компанія Rozum Robotics може стати потенційним покупцем нової системи, так як основними продуктами компанії є колаборативні роботи PULSE. Головні переваги перед конкурентами, насамперед Universal Robots (Данія), - це більш високе співвідношення ваги (8 кг) і корисного навантаження (3 кг) і більш низька ціна,

що досягається за рахунок використання електроприводів власної розробки. Колаборативні роботи мають багатоцільове призначення і можуть використовуватися при різних роботах: зварювання, складання, нанесенні покриттів, дозуванні, обробці і різанню, упаковці, пакетування і ін. Додатково компанія продає свої електроприводи як самостійний продукт.

Розробку нової системи для робота-маніпулятора можна віднести до складної програмної продукції, яка потребує спеціального налагодження. В цьому випадку найчастіше програмне забезпечення розробляється за замовленням споживача.

4.4. Оцінка економічної ефективності впровадження програмного забезпечення

У цьому розділі буде проведено аналіз ефективності впровадження створеної нової системи для робота-маніпулятора.

Через наявність лише удосконаленої системи та оскільки дана робота передбачає впровадження розробленого рішення на дослідницькому роботі, та не включає в себе промисловий комплекс, який можна впровадити, неможливо розрахувати економічний ефект, в якому обсязі необхідні інвестиції, який термін окупності і очікуваний прибуток.

Тому розглядається тільки соціальний ефект.

За допомогою провадження удосконаленої системи до планування шляху мобільним роботом для навігації може:

- запобігти виникненню нещасного випадку на підприємстві;
- збільшити функціонал та можливості мобільного робота;
- підвищити продуктивність праці;
- скоротити кількість працівників.

Можна зробити висновок, що впровадження нової системи повинно мати позитивний економічний ефект тому, що дана розробка є актуальною та

необхідною для широкого сектору робототехніки як напряму, з актуальним соціальним ефектом.

Висновок:

У результаті виконання кваліфікаційної роботи було створено нову систему для робота-маніпулятора. У даному економічному розділі було визначено трудомісткість на розробку додатку, що складає 1866 люд-год, проведено підрахунок вартості роботи по створенню описаної системи, які склали 423850 грн. та розраховано час на його створення – 10,3 міс. Було проаналізовано та досліджено ринок збуту та потенційних покупців нової системи навігації мобільного робота. Визначено, що саме компанія Rozum Robotics може стати потенційним покупцем нової системи, так як основними продуктами компанії є роботи маніпулятори. Створена система є найбільш актуальною для промислових завдань. Її впровадження повинно також мати позитивний соціальний ефект.

CONCLUSIONS

The study consisted of considering and integrating existing methods and algorithms to solve problems of robotic grasping, planning of movements with obstacle avoidance, and recognition of slipping. The most appropriate algorithms and implementations were identified.

The main approaches in each area are considered. There are practical implementations of these algorithms that successfully solve the problem. However, the main problem is the integration of these implementations into a single system. Some integrations are already available in the systems under study, such as Octomap building an octant tree that describes obstacles to motion planning in MoveIt. However, a complete system that would meet all the requirements and be suitable for this equipment has not been identified.

The first step is to recognize the object. The main approaches to object recognition are considered, their advantages and disadvantages are described. As a result, YOLO was chosen as a fast and straightforward way to recognize objects in an image.

Next, we need to find the positions to grab the object. The most advanced solution to this problem is the GPD package. At the entrance, we get a point-cloud where we distinguish objects and finds positions to grab. We can define a workspace to remove unnecessary objects from view. We can also filter objects by removing points from the point cloud on a mask taken from a specific object.

A special device has been developed for direct grasping. The fingers are created using 3D printing in the form of a Fin-Ray structure that allows you to cover the object to increase the contact area. The larger the contact area, the less slippage is possible, and the less effort is required for a secure grip. The fingers are driven by a single electric motor, which transmits force to them by gears. This configuration provides a symmetrical closure of the grasping device.

To identify the object immediately in front of the grab, a ToF sensor is used, which measures the distance to the object using the duration of the return of the emitted light. This allows you to determine that the object is close enough to capture it.

When the object is in the process of grasping, you need to adjust the grasping force to fix the object in the device securely. This is done by slip detection. Tactile sensors are used for this purpose. If the object is moving, the sensors give higher values than when the object is kept stable. These signals are processed by a deep neural network. The signal is transmitted to the neural network almost raw, only normalized. That is, no noise and error filtering. This approach has proven itself well, the system works reliably and does not require fine-tuning. It is even possible to transfer the trained model between different grasping devices.

The grasping device is mounted on the Sawyer Arm manipulator mounted on a moving robot on the MPO-750 platform. This is a manipulator with 7 degrees of freedom. It comes with your computer. The hand is controlled via this computer using the Intera SDK, which is adapted for use with MoveIt.

The central part of traffic planning is the MoveIt framework. This is a surprisingly flexible and advanced framework. It contains many built-in traffic planners from the OMPL library. Among which the most suitable for the robot’s tasks were selected.

The runtime of all software is ROS. This system creates a flexible environment for running any program. Programs work as nodes that exchange messages through topics or call each other’s services.

The described system was developed and tested on the basis of Leonart’s robot developed by the RT-Lions team.

REFERENCES

1. Monkman G. J., Hesse S., Steinmann R., Schunk H. Robot Grippers / G.J.Monkman. S. Hesse, R. Steinmann – Wiley-VCH, 2007 – 62 p.

2. Koubaa A. Robot Operating System (ROS) / A. Koubaa – Springer International Publishing, 2016. – 728 p.

3. ten Pas A., Gualtieri M., Saenko K., Platt R. Grasp Pose Detection in Point Clouds / A. ten Pas, M. Gualtieri, K. Saenko, R. Platt – The International Journal of Robotics Research, 2017. – 17 p.

4. De Santis A., Albu-Schaffer A., Ott C., SicilianoB., Hirzinger G. The skeleton algorithm for self-collision avoidance of a humanoid manipulator / A. De Santis, A. Albu-Schaffer, C. Ott, B. Siciliano, G. Hirzinger – IEEE/ASME international conference on advanced intelligent mechatronics, Zurich, 2007. – 6 p.

5. Pfaff O., Simeonov S., Cirovic I., Stano P., Application of finray effect approach for production process automation / O. Pfaff, S. Simeonov, I. Cirovic, P. Stano – Annals & Proceedings of DAAAM International, 2011 – 2 p.

6. Crooks W., Vukasin G., O'Sullivan M., Messner W., Rogers C., Fin ray® effect inspired soft robotic gripper: From the robosoft grand challenge toward Optimization / W. Crooks, G. Vukasin, M. O'Sullivan, W. Messner and C. Rogers – Frontiers in Robotics and AI, 2016 – 9 p.

7. Decuir1 F., Phelan K., Hollins B. Mechanical Strength of 3-D Printed Filaments / F. Decuir1, K. Phelan, B. Hollins – 32nd Southern Biomedical Engineering Conference, 2016 – 2 p.

8. Covestro Deutschland AG Processing of TPU by Injection Molding / Covestro Deutschland AG – Dormagen, 2016 – 32 p.

9. Takahashi H., Punpongsanon P., Kim J. Programmable Filament: Printed Filaments for Multi-material 3D Printing / H. Takahashi, P. Punpongsanon, J. Kim – UIST '20: Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology, 2020 – 13 p.

10. Voigtlaender P., Chai Y., Schroff F., Adam H., Leibe B., Liang-Chieh Chen FEELVOS: Fast End-to-End Embedding Learning for Video Object Segmentation / P. Voigtlaender, Y. Chai, F. Schroff, H. Adam, B. Leibe, Liang-Chieh Chen – CVPR, 2019 – 10 p.

11. Du G., Wang K., Lian S., Zhao K. Vision-based Robotic Grasping From Object Localization, Object Pose Estimation to Grasp Estimation for Parallel Grippers: A Review / G. Du, K. Wang, S. Lian, K. Zhao – Artificial Intelligence Review, 2020 – 39 p.

12. Schmidt P., Vahrenkamp N., W'achter M., Asfour T. Grasping of Unknown Objects using Deep Convolutional NeuralNetworks based on Depth Images / P. Schmidt, N. Vahrenkamp, M. W'achter, T. Asfour – IEEE International Conference on Robotics and Automation (ICRA), 2018 – 8 p.

13. Gautam R., Gedam A., Zade A., Mahawadiwar A. Review on Development of Industrial Robotic Arm / R. Gautam, A. Gedam, A. Zade, A. Mahawadiwar – International Research Journal of Engineering and Technology (IRJET), 2017 – 4 p.

14. Lalancette C., Loretz S. ROS URDF C. Lalancette, S. Loretz – http://wiki.ros.org/urdf

15. Davison A. J. Real-Time Simultaneous Localisation and Mapping with a Single Camera / A. J. Davison – IEEE International Conference on Computer Vision (ICCV), 2003 – 8 p.

16. Yovchev K., Chikurtev D., Chivarov N., Shivarov N. Precise Positioning of a Robotic Arm Manipulator Using Stereo Computer Vision and Iterative Learning Control / K. Yovchev, D. Chikurtev, N. Chivarov, N. Shivarov – International Conference on Robotics in Alpe-Adria Danube Region RAAD, Springer, 2017 – 8 p.

17. Safeea M., Neto P., Bearee R. On-line collision avoidance for collaborative robot manipulators by adjusting off-line generated paths: An industrial use case / M. Safeea, P. Neto, R. Bearee – Robotics and Autonomous Systems, Elsevier, 2019, 119, pp.278-288.

18. R. Mur-Artal J. D. Tardos Open-Source SLAM System forMonocular, Stereo and RGB-D Cameras / R. Mur-Artal J. D. Tardos – IEEE Transactions on Robotics, 2016 – 9 p.

19. Mur-Artal R., Montiel J. M. M., Tardos J. D., ORB-SLAM: aversatile and accurate monocular SLAM system / R. Mur-Artal, J. M. M. Montiel, J. D. Tardos – IEEE Trans, 2015 Robot.,vol. 31, no. 5, pp. 1147–1163.

20. Newcombe R. A., Davison A. J., Izadi S., Kohli P., Hilliges O., Shotton J., Molyneaux D., Hodges S., Kim D., Fitzgibbon A. KinectFusion: Real-time dense surface mapping and tracking / A. Newcombe, A. J. Davison, S. Izadi, P. Kohli, O. Hilliges,J. Shotton, D. Molyneaux, S. Hodges, D. Kim, A. Fitzgibbon – in IEEEInt. Symp. on Mixed and Augmented Reality (ISMAR), 2011.

21. Xu T., Tian B., Zhu Y., Tigris: Architecture and Algorithms for 3D Perceptionin Point Clouds / T. Xu, B. Tian, Y. Zhu – the 52nd Annual IEEE/ACM International Symposium, 2019 – 14p.

22. Dupuy J., Iehl J.-C., Poulin P. Quadtrees on the GPU / J. Dupuy, J.-C. Iehl, P. Poulin – GPU Pro 360 (pp.211-222), 2018 10.1201/9781351052108-12.

23. Liu S., Ororbia A., Giles C. Learning a Hierarchical Latent-Variable Model of Voxelized 3D Shapes / S. Liu, A. Ororbia, C. Giles – International Conference on 3D Vision (3DV), 2017 – 13 p.

24. Koh N., Jayaraman, P. Zheng, J Parallel Point Cloud Compression Using Truncated Octree / N. Koh, Jayaraman, P. Zheng – 2020 International Conference on Cyberworlds (CW), 2020 – 1-8 p.

25. Dricot A., Ascenso J. Hybrid Octree-Plane Point Cloud Geometry Coding / A. Dricot, J. Ascenso – 27th European Signal Processing Conference (EUSIPCO), 2019 – 1-5 p.

26. Kucuk S., Bingul Z. Robot Kinematics: Forward and Inverse Kinematics / S. Kucuk, Z. Bingul – Industrial Robotics: Theory, Modelling and Control 2006, 10.5772/5015.

27. Boeuf A., Cortés J., Siméon T. Motion Planning / A. Boeuf, J. Cortés, T. Siméon – Aerial Robotic Manipulation (pp.317-332), 2019 10.1007/978-3-030-12945-3_23.

28. Liu S., Liu P. A Review of Motion Planning Algorithms for Robotic Arm Systems / Liu S., Liu P. – The 8th International Conference on Robot Intelligence Technology, Cardiff, 2020.

29. Abdulkareem A. Slip detection with accelerometer and tactile sensors in a robotic hand model / A. Abdulkareem – IOP Conference Series: Materials Science and Engineering, 2015 – 10 p.

30. Kahlmann T., Oggier T., Lustenberger F., Blanc N., Ingensand H. 3D-ToF sensors in the automobile / T. Kahlmann, T. Oggier, F. Lustenberger, N. Blanc, Ingensand H. – Proceedings of SPIE - The International Society for Optical Engineering, 2005 10.1117/12.607261.

31. Wang X. A flexible slip sensor using triboelectric nanogenerator approach / X. Wang – Journal of Physics: Conference Series, 2018 – 7p.

32. Fraden J. Handbook of Modern Sensors: Physics, Designs, and Applications 4th ed / J. Fraden – Springer, New York, 2010.

33. Rethink Robotics Sawyer, the high performance collaborative robot / Rethink Robotics – https://www.rethinkrobotics.com/sawyer

34. Joseph L., Cacace J. Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System, 2nd Edition / L. Joseph, J. Cacace – Packt Publishing Ltd, 2018 – 580 p.

35. Foote T., Marder-Eppstein E., Meeussen W. ROS Transformation Tree / T. Foote, E. Marder-Eppstein, W. Meeussen – http://wiki.ros.org/tf

36. Intel Depth Camera D435 / Intel -- https://www.intelrealsense.com/depth-camera-d435/

37. Rusu R., Cousins S. 3D is here: Point cloud library (PCL) / R. Rusu S. Cousins -- IEEE International Conference on Robotics and Automation 2011 (ICRA 2011), 2011 -- 10.1109/ICRA.2011.5980567.

38. Redmon J., Divvala S., Girshick R., Farhadi A. You Only Look Once: Unified, Real-Time Object Detection / J. Redmon, S. Divvala, R. Girshick, A. Farhadi -- 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016 – 10 p.

39. Hornung A., Wurm K. M., Bennewitz M., Stachniss C., Burgard W. OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees / A. Hornung, K. M. Wurm, M. Bennewitz C. Stachniss W. Burgard -- Autonomous Robots, Springer, 2013 – 17 p.

40. Chitta, S. MoveIt!: An Introduction / S. Chitta -- Robot Operating System (ROS) (pp.3-27), 2016 – 25 p.

41. Moveit Tutorials – Kinetic Documentation http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/index.html

42. Görner M., Haschke R. MoveIt! Task Planning / M. Görner, R. Haschke -- ROSCon2018, 2018

43. Görner M., Haschke R., Ritter H., Zhang J., MoveIt! Task Constructor for Task-Level Motion Planning / M. Görner, R. Haschke, H. Ritter J. Zhang –2019 International Conference on Robotics and Automation (ICRA), 2019 – 7p.

44. Thinakaran K. 3D Visual System Navigation for Service Robotics: Implementing the MoveIt Motion Planner for a Simulated Robot Arm / K. Thinakaran -- Reutlingen University, 2020 – 97 p.

45. Pan J., Chitta S., Manocha D. FCL: A General Purpose Library for Collision and Proximity Queries / J. Pan, S. Chitta, D. Manocha -- Proceedings - IEEE International Conference on Robotics and Automation, 2012 – 8 p.

46. Gopal A. Employing Bionic Sensors for Deep Learning based Grasping Slip Detection / A. Gopal – Reutlingen University, 2019 – 59 p.

APPENDIX A

SOURCE CODE

saywer_gazebo.launch

```
<?xml version="1.0" encoding="utf-8"?>
<launch>

  <!-- these are the arguments you can pass this launch file, for
example paused:=true -->
  <arg name="paused" default="false"/>
  <arg name="camera" default="true"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>
  <arg name="head_display_img" default="$(find
sawyer_gazebo)/share/images/sawyer_sdk_research.png"/>

  <!-- This argument loads the electric gripper, for example
electric_gripper:=true -->
  <arg name="electric_gripper" default="false"/>
  <!-- This argument loads sawyer's pedestal URDF -->
  <arg name="pedestal" default="true"/>
  <!-- This argument fixes the robot statically to the world -->
  <arg name="static" default="true"/>
  <!-- This argument dictates whether gazebo should be launched in
this file -->
  <arg name="load_gazebo" default="true"/>
  <!-- This argument sets the initial joint states -->
  <arg name="initial_joint_states"
    default=" -J sawyer::right_j0 -0.27
              -J sawyer::right_j1 1.05
              -J sawyer::right_j2 0.00
              -J sawyer::right_j3 0.49
              -J sawyer::right_j4 -0.08
              -J sawyer::right_j5 -0.06
              -J sawyer::right_j6 0.027
              -J sawyer::head_pan 0.00"/>

  <param name="img_path_head_display" value="$(arg
head_display_img)"/>

  <!-- Load the URDF into the ROS Parameter Server -->
  <!-- This xacro will pull in sawyer.urdf.xacro, and
right_end_effector.urdf.xacro
       Note: if you set this to false, you MUST have set the
robot_description prior
             to launching sawyer_world -->
  <arg name="load_robot_description" default="true"/>
```

```
  <param if="$(arg load_robot_description)"
name="robot_description"
      command="$(find xacro)/xacro --inorder $(find
sawyer_description)/urdf/sawyer.urdf.xacro
      gazebo:=true electric_gripper:=$(arg electric_gripper)
      pedestal:=$(arg pedestal) static:=$(arg static)
camera:=$(arg camera)"/>
  <!-- Load Parameters to the ROS Parameter Server -->
  <rosparam command="load" file="$(find sawyer_gazebo)/config
/config.yaml" />
  <rosparam command="load" file="$(find
sawyer_description)/params/named_poses.yaml" />
  <rosparam command="load" file="$(find sawyer_gazebo)/config
/acceleration_limits.yaml" />
  <param name="robot/limb/right/root_name" value="base" />
  <param if="$(arg electric_gripper)"
name="robot/limb/right/tip_name"
         value="right_gripper_tip" />
  <param unless="$(arg electric_gripper)"
name="robot/limb/right/tip_name"
         value="right_hand" />

  <param name="robot/limb/right/camera_name"
value="right_hand_camera" />
  <param if="$(arg electric_gripper)"
name="robot/limb/right/gravity_tip_name"
         value="right_gripper_tip" />
  <param unless="$(arg electric_gripper)"
name="robot/limb/right/gravity_tip_name"
         value="right_hand" />


  <!-- We resume the logic in empty_world.launch, changing the
name of the world to be launched -->
  <include if="$(arg load_gazebo)" file="$(find
gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find
sawyer_gazebo)/worlds/sawyer.world"/>
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>

  <!-- Publish a static transform between the world and the base
of the robot -->
  <node if="$(arg static)" pkg="tf2_ros"
type="static_transform_publisher"
     name="base_to_world" args="0 0 0 0 0 0 1 world base" />

  <!-- Run a python script to the send a service call to
gazebo_ros to spawn a URDF robot -->
```

```
   <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
respawn="false" output="screen"
     args="-param robot_description -urdf -z 0.93 -model sawyer
$(arg initial_joint_states)" />

  <!-- ros_control sawyer launch file -->
  <include file="$(find
sawyer_sim_controllers)/launch/sawyer_sim_controllers.launch">
      <arg name="electric_gripper" value="$(arg
electric_gripper)"/>
      <arg name="gui" value="$(arg gui)" />
  </include>

  <!-- sawyer cameras launch file -->
  <arg name="wrist_camera" default="right_hand_camera" />
  <arg name="head_camera"  default="head_camera" />
  <include file="$(find
sawyer_gazebo)/launch/sawyer_sim_cameras.launch">
      <arg name="wrist_camera" value="$(arg wrist_camera)" />
      <arg name="head_camera" value="$(arg head_camera)" />
  </include>


  <!-- Publish a static transform between the world and the base
of the robot -->
  <node pkg="rosbag" type="play" name="io_robot" args="-l $(find
sawyer_gazebo)/share/bags/robot_io.bag" />

</launch>
```

sawyer_moveit.launch

```
<?xml version="1.0"?>
<launch>
  <arg name="config" default="true"/>
  <arg name="rviz_config" default="$(find sawyer_moveit_config
)/launch/moveit.rviz" />

  <!-- By default, we do not start a database (it can be large) --
>
  <arg name="db" default="false" />
  <!-- By default, we are not in debug mode -->
  <arg name="debug" default="false" />

  <!-- By default we do not overwrite the URDF. Change the
following to true to change the default behavior -->
  <arg name="load_robot_description" default="false"/>
  <!-- Add controller box collision shape to check for link
collisions if set to true-->
  <arg name="controller_box" default="true"/>
  <!-- Left and electric gripper arg. Set to true to check for
collisions for their links -->
  <arg name="electric_gripper" default="false"/>
  <!-- Set the kinematic tip for the right_arm move_group -->
```

```
  <!-- <arg name="tip_name"     if="$(arg electric_gripper)"
default="right_gripper_tip"/>
  <arg name="tip_name" unless="$(arg electric_gripper)"
default="right_hand"/> -->

  <node name="joint_trajectory_action_server"
pkg="intera_interface" type="joint_trajectory_action_server.py"/>

  <arg name="custom_gripper" default="false"/>
  <arg name="gpd" default="false"/>

  <node if="$(arg custom_gripper)" pkg="tf2_ros"
type="static_transform_publisher" name="link1_broadcaster" args="0
0 0 -1.5708 0 0 stp_021709TP00478_tip gripper_tcp" />
  <node if="$(arg custom_gripper)" pkg="tf2_ros"
type="static_transform_publisher" name="link2_broadcaster" args="0
0 -0.17 0 0 0 right_hand fake_ee" />
  <node if="$(arg custom_gripper)" pkg="tf2_ros"
type="static_transform_publisher" name="link3_broadcaster"
args="0.07 0 0 0 -1.9 3.14 right_hand camera_link" />
  <!-- <node pkg="tf2_ros" type="static_transform_publisher"
name="link3_broadcaster" args="0.02 -0 0.27 0 0.38 0 head
camera_link" /> -->

  <arg name="tip_name"     if="$(arg custom_gripper)"
default="stp_021709TP00478_tip"/>
  <arg name="tip_name" unless="$(arg custom_gripper)"
default="right_hand"/>

  <!-- GPD Grasping server launch -->
  <include if="$(arg gpd)" file="$(find
gpd_ros)/launch/server.launch"/>

  <!-- Add planning context launch file -->
  <include file="$(find sawyer_moveit_config
)/launch/planning_context.launch">
    <arg name="load_robot_description" value="$(arg
load_robot_description)"/>
    <arg name="electric_gripper" value="$(arg electric_gripper)"/>
    <arg name="tip_name" value="$(arg tip_name)"/>
    <arg name="controller_box" value="$(arg controller_box)"/>
    <arg name="custom_gripper" value="$(arg custom_gripper)"/>
  </include>


  <arg name="kinect" default="false" />
  <arg name="xtion" default="false" />
  <arg name="realsense" default="false" />
  <arg name="camera_link_pose" default="0.15 0.075 0.5 0.0 0.7854
0.0"/>
  <include file="$(find sawyer_moveit_config
)/launch/move_group.launch">
    <arg name="kinect" value="$(arg kinect)" />
```

```
    <arg name="xtion" value="$(arg xtion)" />
    <arg name="realsense" value="$(arg realsense)" />
    <arg name="camera_link_pose" default="$(arg
camera_link_pose)"/>
    <arg name="allow_trajectory_execution" value="true"/>
    <arg name="fake_execution" value="false"/>
    <arg name="info" value="true"/>
    <arg name="debug" value="$(arg debug)"/>
  </include>
  <include file="$(find sawyer_moveit_config
)/launch/moveit_rviz.launch">
    <arg name="config" value="$(arg config)" />
    <arg name="debug" value="$(arg debug)"/>
    <arg name="rviz_config" value="$(arg rviz_config)" />
  </include>
  <!-- If database loading was enabled, start mongodb as well -->
  <include file="$(find sawyer_moveit_config
)/launch/default_warehouse_db.launch" if="$(arg db)"/>
</launch>
```

Motorsteuerung_mit_Sensoren.ino

```
//AX-12 Servo Motor
#include "Arduino.h"
#include "AX12A.h"
#include <Wire.h>
#include "VL53L1X.h"
#include "fslp_lib.h"

#define DirectionPin   (10u)
#define BaudRate      (1000000ul)
#define ID        (1u)
#define IDB        (2u)
#define LEDred    33  //immer an
#define LEDgreen  32  //nur an, wenn Objekt gegriffen


void setup()
{

  pinMode(LEDred, OUTPUT);
  pinMode(LEDgreen, OUTPUT);

  Serial.begin(9600);
  delay(250);
  Serial.println("Starting Gripper");
  ax12a.begin(BaudRate, DirectionPin, &Serial2);

  Wire.begin();
  Wire.setClock(400000); // use 400 kHz I2C

  sensor.setTimeout(500);

  if (!sensor.init())
```

```
  {
    Serial.println("Failed to detect and initialize sensor!");
    while (1);
  }

  sensor.setDistanceMode(VL53L1X::Long);
  sensor.setMeasurementTimingBudget(50000);
  sensor.startContinuous(50);

}

void executeCommand(){
  if(SerialCommand != 0){
    //New Command

    //Open gripper
    if(SerialCommand == 1){
      Serial.println("Open Gripper");

      digitalWrite(LEDred, HIGH);  //hier rote LED an
      digitalWrite(LEDgreen, LOW);  //hier grüne LED aus

      int notClosed = 1;
      while(notClosed){
        currentPos -=5;
        currentPosB +=5;
        ax12a.move(ID, currentPos);
        ax12a.move(IDB, currentPosB);
          if(currentPos<450){
            notClosed = 0;
            currentPos = 450;
          }
          if(currentPosB>900){
            notClosed = 0;
            currentPosB = 900;
          }
          delay(25);
      }
      //Close Gripper
    }else if(SerialCommand == 2){
      Serial.println("Closing Gripper");

      digitalWrite(LEDred, HIGH);  //hier rote LED an
      digitalWrite(LEDgreen, LOW);  //hier grüne LED aus

      int notOpened = 1;
      while(notOpened){
        currentPos +=5;
        currentPosB -=5;
        ax12a.move(ID, currentPos);
        ax12a.move(IDB, currentPosB);
          if(currentPos>900){
            notOpened = 0;
```

```
            currentPos = 900;
          }
          if(currentPosB<450){
            notOpened = 0;
            currentPosB = 450;
          }
          delay(25);
      }
      //Print all data
    }else if(SerialCommand == 3){

      digitalWrite(LEDred, HIGH);  //hier rote LED an
      digitalWrite(LEDgreen, LOW);  //hier grüne LED aus

      //Print gripper data
      //reg = ax12a.readRegister(ID, AX_PRESENT_VOLTAGE, 1);
      //Serial.println(reg);
      for(int i = 1000;i>0;i--){
        printData();
        delay(10);
      }
    }
    //Gripp object
    else if(SerialCommand == 4){

      digitalWrite(LEDred, HIGH);  //hier rote LED an
      digitalWrite(LEDgreen, LOW);  //hier grüne LED aus

      int notOpened = 1;
      int pressureRight, pressureLeft;
      int *measurement;
      measurement=getFSLP_Right();
      pressureRight=measurement[0];


      measurement=getFSLP_Left();
      pressureLeft=measurement[0];
      int threshold = 35;
      while(notOpened && (pressureRight<threshold &&
pressureLeft<threshold)){
        int speed = 5;
        currentPos +=speed;
        currentPosB -=speed;
        ax12a.move(ID, currentPos);
        ax12a.move(IDB, currentPosB);
          if(currentPos>900){
            notOpened = 0;
            currentPos = 900;
          }
          if(currentPosB<450){
            notOpened = 0;
            currentPosB = 450;
          }
```

```
          delay(5);
        measurement=getFSLP_Right();
        pressureRight=measurement[0];
        Serial.println(pressureRight);
        measurement=getFSLP_Left();
        pressureLeft=measurement[0];
        Serial.println(pressureLeft);
        if (pressureRight or pressureLeft > 1){
          delay(10);
        }
      }
      digitalWrite(LEDred, LOW);  //hier rote LED aus
      digitalWrite(LEDgreen, HIGH); //hier grüne LED aus
    }

    SerialCommand = 0;
  }
}


void loop()
{


  //Get User Input
  if (Serial.available() > 0) {
    char rx_byte = 0;
    rx_byte = Serial.read();

    // check if a number was received
    if ((rx_byte >= '0') && (rx_byte <= '9')) {
      SerialCommand = rx_byte - '0';
    }
    else {
      Serial.println("Possible Commands :");
      Serial.println(" 1 : Open Gripper");
      Serial.println(" 2 : Close Gripper");
      Serial.println(" 3 : Print all data");
      Serial.println(" 4 : Gripp Object");
    }
  }
  delay(20);
  executeCommand();

}

int printData(){
  int *measurement;
  int pressRight,posRight;
  int pressLeft,posLeft;
  if (toggle==1){
    Serial.println("Rechts");
    measurement=getFSLP_Right();
```

```
    pressRight=measurement[0];
    posRight= measurement[1];
    toggle=!toggle;
    }
  else{
    Serial.println("Links");
    measurement=getFSLP_Left();
    pressLeft=measurement[0];
    posLeft=measurement[1];
    toggle=!toggle;
    }

  /*
  char report[180];
  sprintf(report, "pressure left: %5d position left: %5d pressure
right %5d position right %5d Tof Sensor: %5d\n",
  pressLeft,posLeft,pressRight,posRight,sensor.read());
  Serial.print(report);
  */

  char report[180];
  sprintf(report, "pressure: %5d   position: %5d\n",
  measurement[0], measurement[1]);
  Serial.print(report);

  if (sensor.timeoutOccurred()) { Serial.print(" TIMEOUT"); }

  sprintf(report,"Tof Sensor : %5d\n",sensor.read());
  Serial.print(report);

  delay(250);


}
```

ДОДАТОК Б

ВІДГУК

керівника економічного розділу

на кваліфікаційну роботу магістра

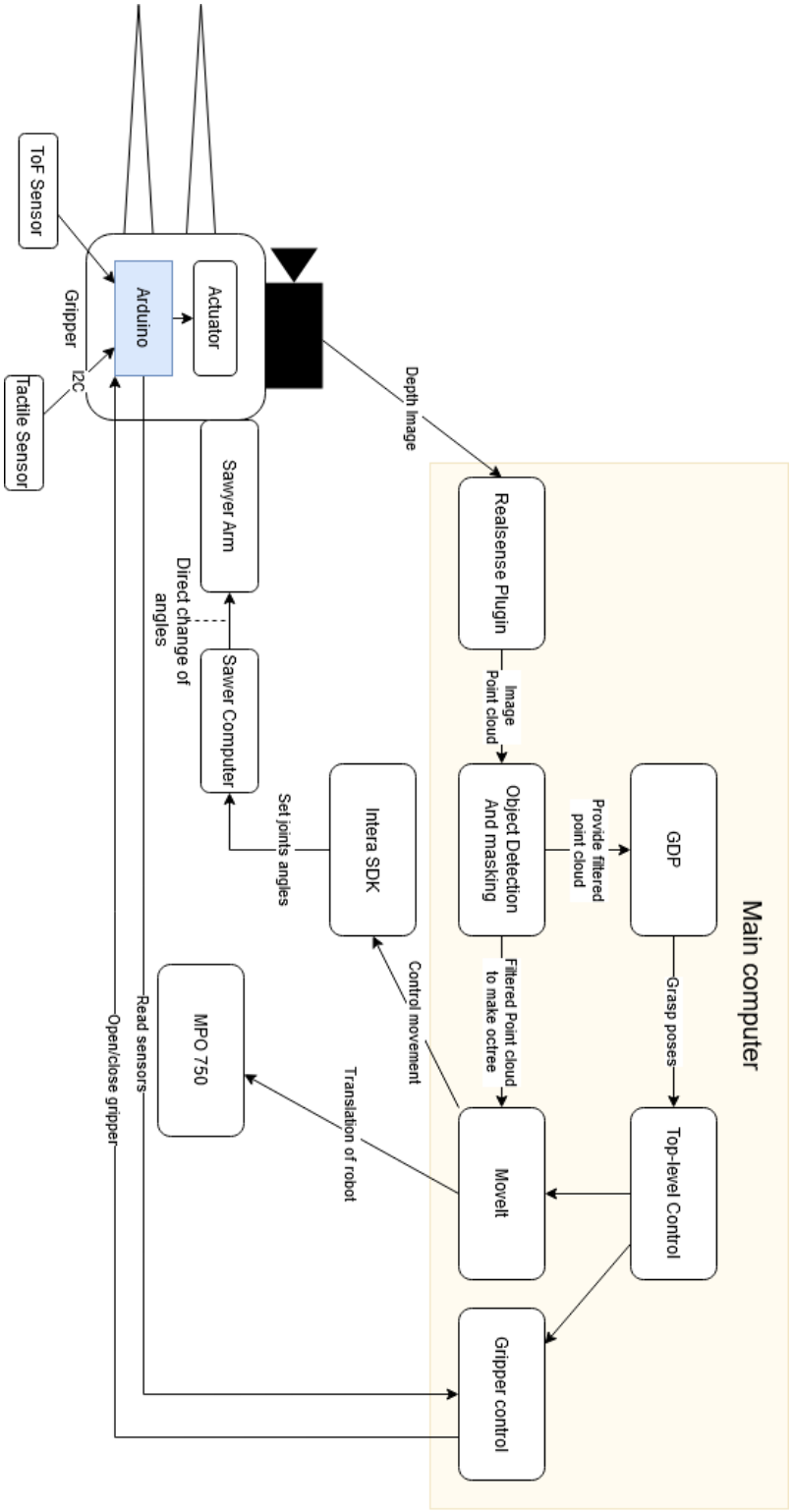
на тему: «Моделі, алгоритми та програмне забезпечення для планування шляху для навігації мобільних роботів з уникненням перешкод на основі дерева октантів»

студента групи 121м-19-1 Рудя Вячеслава Васильовича

Керівник економічного розділу
доцент каф. ПЕП та ПУ, к.е.н. **Л. В. Касьяненко**

APPENDIX C

STRUCTURE OF THE DEVELOPED SOLUTION



APPENDIX D

LIST OF FILES ON THE DISC

Filename	Description
Explanatory documents	
Thesis_RudViacheslav.docx	An explanatory note to the diploma project. Word document.
Thesis_RudViacheslav.pdf	An explanatory note to the diploma project in PDF format
Program	
Arm.zip	Archive. Contains program codes and a program
Presentation	
Presentation_RudViacheslav.pptx	Presentation of the diploma project