6CCS3PRJ Person Following Robot

Author: Allister Gough

Supervisor: Matteo Leonetti

Student ID: K21066336

April 12, 2024

Abstract

Robots are becoming increasingly popular in society and are being used to optimise workplace efficiency across many service sectors. Logistics and healthcare are some examples. A subset of these robots use person-following technology to achieve their service task. This project focuses on person-following technology and aims to implement a person-following robot that can contribute to this developing field of research.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Allister Gough
April 12, 2024

Acknowledgements

I would like to thank my supervisor, Matteo Leonetti, for his diligent support and advice during times of confusion and procrastination. I would also like to thank my peers for their technical tips throughout the course of this project.

Contents

Chapter 1

Introduction

Productivity is often limited by how much we can carry. This statement can be observed in many of our daily routines. For instance, when moving shopping bags from our cars to our homes, the amount of load we can transport is limited by how many bags we can move at one time. Hospitals and warehouses have gone about increasing this capacity by using devices like trolleys for their porters and movers. Whilst this solution provides a productivity improvement, it is heavily dependent on the performance of the human operative and their ability to physically push the trolley; humans can become tired. Robots, on the other hand, can perform tasks exhaustively without fatiguing and are also capable of carrying greater loads, making them a superior solution to both aspects of our productivity problem. ZMP, a Japanese robotics company, has already developed several robots designed for load-moving, with the most notable being CarriRo[1], a trolley-type logistics support robot capable of following people around. CarriRo has been deployed to numerous logistics companies, and in this case study[2], Yamato Transport Co., Ltd. describes how, by using CarriRo, they were "able to reduce the 'waste' of work" and were also "able to improve efficiency".

To fulfil their desired function, robots like CarriRo require person-following behaviour, a technology that combines tasks such as object-detection, point estimation, path planning, and whose scope of influence extends beyond the industrial. In the social domain, this technology is being used in personal luggage-carrying robots designed for daily excursions, with the most notable model being PFFs Gita robot[3]. In healthcare, research is being conducted on how this technology can be applied to wheelchairs to allow the incapacitated to autonomously follow nurses around hospitals[4].

Clearly, person-following robots can provide high value to many fields in society. Throughout this project, I intend on implementing my own person-following robot, that can be adapted to facilitate the functions mentioned above.

1.1 Aims and Objectives

The aims for my person-follower are as follows. The final product should be able to:

- **Detect** and **move** towards a person in the field of view.
- **Identify** and **re-identify** the target person by some physical attribute. In the event other people enter the field of view, the robot should recognise the target based on this attribute. I have chosen to use the colour of the person's clothing for this.
- Perform a **recovery scan** of the environment when the target person leaves the field of view.

Chapter 2

Background

This chapter aims to provide more context to the world of robots and robot applications, in hopes of giving the reader a better understanding of the later chapters of this paper.

2.1 Robots

Before we dive into the technicalities of a robot application, it's important to clarify what a robot is. Simply put, **robots are programmable machines designed to carry out a variety of tasks**. They can range from small vacuum cleaning devices, like iRobot's Roomba[5], to bigger machines capable of movement and vision, like Pal Robotic's TIAGo[6], which will be addressed more of later in this paper. In recent times, they have become more prevalent in society and can be seen used in the aforementioned logistics, social and healthcare fields, but also in manufacturing, agriculture and defense.

2.2 ROS

ROS[7], short for Robot Operating System, is an open-source robot development framework. Despite having "Operating System" in its name, ROS hardly resembles Windows, but instead a set of software libraries and tools that users can use to develop robotic systems. ROS was initially released in 2007 by Willow Garge but was later adopted by the Open Source Robotics Foundation. Since then, ROS has become widely used across academia and certain sectors of the robotics industry, so much so that I'd argue it has become the industry-standard platform for robot development. As ROS was used to develop this project, I will briefly explain its relevant concepts. ROS applications consist of **nodes**, which are simply processes that can perform

different tasks. These tasks can range from sensor data processing, to control algorithms and actuator control. The number of nodes used in an application depends on its complexity. Nodes will typically need to communicate with each other, and they do so in several ways. The simplest of these is through **topics**. Topics are channels that nodes can send and receive data to and from. Nodes do this by either publishing or subscribing to the topic. Nodes can also communicate by offering **services**, which other nodes can call to request actions or data. Lastly, nodes can also communicate through **actions**, which are used for long-duration tasks. **Messages** are the data structures exchanged through topics, services, and actions. They typically consist of multiple fields of varying types, with each field representing a different piece of data.

Chapter 3

Literature Review

This chapter dissects person-following technology into its main components, which were mentioned earlier in the introduction. Here, existing work in each domain will be analysed, and the advantages and disadvantages of each approach will be discussed.

3.1 Object Detection

Object detection is an integral component of person-following technology. Without the ability to recognise objects, the robot will not be able to detect people to follow, and not be able to detect objects to avoid. State-of-the-art methods for detecting objects make use of artificial intelligence. Here, AI models are trained against large data sets to make predictions on data they haven't seen, using what they've learned from the data they have seen. In this case, these models are fed data including images of the objects of interest, labels defining what objects are depicted in the images, and bounding boxes that wrap around the object(s) in the image. By doing this, these models should be able to make predictions of what objects are when fed **new** data.

Of these object detection models, there are two main types. Two-stage detectors work by firstly estimating where an object could be in an image, and secondly by classifying what the object itself could be. One-stage detectors, on the other hand, perform both of these steps simultaneously. Because of this, one-stage detectors offer speed of detection over accuracy, whilst two-stage detectors offer accuracy of detection over speed, as is explained in this paper published by the University of Nebraska - Lincoln, "One-stage detectors have high inference speeds and two-stage detectors have high localization and recognition accuracy"[8].

In existing research on person-followers, one-stage detectors are often picked over two-stage ones. This makes sense considering that, whilst more accurate detections would be desirable for person-followers to reduce the risk of error, the extra time spent using two-stage detection makes this method obsolete for detecting objects in real time. As person followers operate in real-time, one-stage detectors are generally what's used, as can be observed in this research paper where students used a one-stage detection model called YOLACT++[9] for their person-follower.

Figure 3.1: Diagram illustrating the different stages of object detection.

3.2 Point Estimation

Now that a person has been detected, the next step is to estimate where the person is. Existing research[9] on how to do this uses a three-step process. The first step involves using a person's segmentation mask to filter the point cloud. Point clouds are sets of points used to represent 3D environments, and segmentation masks are regions that outline the shape of the detected object. By applying the segmentation mask of a detected person to the point cloud, only the points that correspond to the person can be extracted. The second step then involves averaging all the person points to produce a centroid, which represents the average point of the person. The third step involves transforming this point to the appropriate frame of reference.

This approach to point estimation is resourceful as the person's segmentation mask is already made available by the detection model, eliminating the need to generate a fresh one.

3.3 Path Planning

Now that a person's location has been determined, a plan of how to get there needs to be created. Path planning is the subject field that addresses how robots move from their current position to a desired position or goal. This action can be achieved using a variety of planning algorithms. José Ricardo Sánchez-Ibáñez, Carlos J. Pérez-del-Pulgar, and Alfonso García-Cerezo review these different algorithms in their paper "Path Planning for Autonomous Mobile Robots: A Review"[10].

The first type of planning algorithms examined are Reactive-Computing-Based Path Planning algorithms. These algorithms use real-time sensor data to quickly react to changes in the environment, and are commonly used as 'local planners'. Local planners focus on short-path planning, while global planners focus on the entire path from start to finish. Typically, both types of planners are used together to achieve effective robot navigation. This is done by using a global planner to create the best overall route, the global plan, which is fed into a local planner that navigates the robot along this path, adapting the path along the way if obstacles are encountered. This real-time adaptability makes Reactive-Computing-Based Path Planning suitable for environments that require robots to react to changes quickly, such as the outdoors, hospitals, warehouses, etc.

The second type of planning algorithms examined are Soft Computing-Based Path Planning algorithms. These algorithms focus not on finding an optimal solution, but instead an approximation of it, allowing for a certain range of imprecision. This makes these algorithms good for situations that contain uncertainty and require robots to predict changes in the environment. However, this predictive behaviour comes at the cost of greater computation.

In general, reactive behaviour is more important than predictive for person-followers, as the environments in which they are deployed will contain unpredictable obstacles that need immediate reaction. As concluded in this review[10] of path planning algorithms, Reactive-Computing-Based algorithms "seem suitable for local obstacle avoidance path planning as they are easy and cheap to implement". For this project, a Reactive-Computing-Based approach has been considered. However, a hybrid of both reactive and predictive planning could offer the best solution for sophisticated person-following robots.

Chapter 4

Requirements

Following the literature review of the main person-following components in the previous section, this chapter lists the requirements for implementing a person-following robot. These requirements will be divided into four categories. User, Hardware, Functional, and Other.
User requirements highlight what the user should expect from the implementation, Hardware requirements highlight what hardware is needed from the robot, Functional requirements highlight what the robot should be able to do in response to the user, and Other requirements highlight how the software and hardware components should integrate.

4.1 User Requirements

- **U.1** The user should be able to have the robot follow them when they are moving within the robot's field of view.

4.2 Hardware Requirements

- **H.1** The robot should be able to see colour and depth data, therefore an RGB-D camera is needed.
- **H.2** The robot should be able to move autonomously, therefore the robot should have means of movement, e.g. wheels, and sensors to perform localisation and object avoidance, e.g LiDAR, ultrasonic, infrared.

4.3 Functional Requirements

- **F.1** The robot should use its camera to detect the user to be followed.
- **F.2** The robot should use its camera to interpret the colour of the user's clothing.
- **F.3** The robot should use its movement method to advance toward the user.
- **F.4** The robot should only pursue the user. Should other people enter the field of view, the robot should not pursue them.
- **F.5** The robot should stop pursuing the user when within a specified range of them (to prevent the robot and user from colliding).
- **F.6** The robot should use its sensors to avoid obstacles when pursuing the user.
- **F.7** The robot should perform a search scan when the user leaves the field of view.
- **F.8** The robot should re-identify the user by the colour of their clothing.

4.4 Other Requirements

- **O.1** The development computer's operating system must be compatible with the robot simulation software(s) and ROS.
- **O.2** The robot-specific libraries must be installed before coding.

Chapter 5

Specification and Design

This chapter details the specification and design choices made to satisfy the requirements listed in the previous chapter.

5.1 TIAGo

To fulfil H.1 and H.2 of the Hardware requirements for this implementation, a robot equipped with camera(s) capable of detecting objects and colour, movement device(s), and sensors for localisation and object avoidance was needed. One robot that fit this bill was SoftBank's Pepper[11]. This robot is equipped with 2 HD cameras and a 3D depth sensor that can be used to detect objects and colours, wheels for movement, and laser, sonar, and bump sensors that can be used for localisation and object avoidance.

Another robot of similar specification is PAL Robotics' TIAGo[6]. TIAGo is a manipulator robot equipped with an RGB-D camera for colour and depth detection, wheels for movement, and a LiDAR sensor for localisation and object avoidance.

Despite both robots possessing the required hardware, Pepper's main functionality lies in exceptional human interaction, which is unnecessary for the scope of this project. Referring back to U.1 from User requirements from the previous chapter, the user just needs to have the robot follow them. Because of this, features like Pepper's voice recognition, speech synthesis, and emotion recognition become redundant. Moreover, the Informatics Department at King's College London is fortunately in possession of a TIAGo robot, making real-life testing available. For these reasons, TIAGo was chosen as the robot for this implementation.

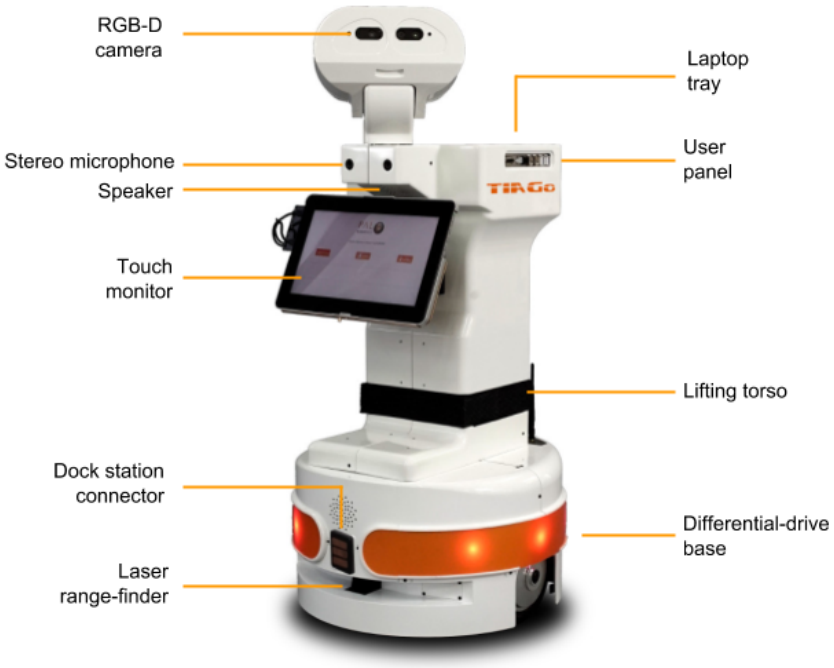

Figure 5.1: Diagram annotating TIAGo with its various sensors.

5.2 Ubuntu Linux

To fulfil O.1 of the Other requirements for this implementation, an operating system with support for ROS and its simulation software was needed. Linux[12] is an open-source operating system created in the 1990s by Finnish software engineer Linus Torvalds. While not as popular as other OSs like Windows and MacOS, mostly due to its non-user-friendly design, Linux's open-source nature and powerful command-line interface make it the OS of choice for more experienced computer users, including ROS developers.

Being open-source allows users to view and edit Linux's source code, which enables ROS developers to customise the OS in ways that support their robot applications. For instance, the OS can be optimised for performance and real-time operations, two potentially crucial aspects of robot applications. The inclusion of drivers and libraries for specific hardware is also supported. Additionally, Linux's command-line interface is another reason why ROS developers use this OS. Normally, users interact with their OS through a graphical user interface (GUI), which depicts the buttons and panels on a computer screen. The command-line interface, commonly referred to as the terminal, is another method of interaction with the computer that uses commands, which are scripts that perform specific tasks, such as listing the contents of a folder or showing all actively running processes. An OS will come with built-in commands but also allows

the creation of custom commands. In general, commands give users much more control over what they can do with their computer whilst also making computer interaction more efficient; a single command can be used to perform tasks that would need several button presses in a GUI. For ROS developers, the command line is needed to run essential commands for the ROS framework, such as ”rosrun” for starting nodes and ”roslaunch” for starting multiple nodes.
Linux distributions are operating systems built upon Linux. Each distribution has its target audience, with Ubuntu being the one that provides the best support for ROS and the simulation software used for this project. For these reasons, Ubuntu Linux was chosen as the operating system of this project.

5.3 Containers

To fulfil O.2 of the Other requirements for this implementation, all of TIAGo's dependencies needed to be acquired. A Container is an executable software package that encapsulates everything needed to run a piece of software. It provides a simple way of migrating software requirements and dependencies between machines. There are different platforms for creating Containers, with the most popular being Docker, but due to personal familiarity, Singularity[13] was chosen to create the Containers used for this project.
Initially, all of the needed dependencies were for development with TIAGo, hence an existing Container for TIAGo was temporarily used. However, more libraries became needed as development progressed, which motivated the creation of a custom Container that included all of TIAGo's dependencies and the new libraries as well.

5.3.1 Limitations

The development machine used for this project belongs to the Informatics Department of King's College London. For security reasons, student users are not given ”root” access and are, as a result, not permitted to perform certain actions. This became an issue when creating the Containers used for this project. In Singularity, Containers are defined in text files with the .def extension, which are then built into executable files with the .sif extension. Sif files are then run to start the Container. However, the action of building the Container requires root privileges.
To overcome this, the supervisor of this project, Matteo Leonnetti, kindly built the .def files on his own machine, allowing the subsequent Containers to be used for this project.

5.4 YOLO

To fulfil F.1, F.2, and F.8 of the Functional Requirements for this implementation, an object-detection model was needed. As discussed in the literature review of object detection models, one-stage detectors are more suitable for real-time systems than two-stage detectors due to their faster detection speed. YOLO[14], an acronym for "You Only Look Once", is a one-stage detection model named for its innovative one-stage approach to object detection that combines bounding box prediction and classification into one step. As a result, YOLO is capable of providing fast and accurate object detections.

Similarly to YOLO, the SSD (Single Shot MultiBox Detector) is another one-stage detector that combines prediction and classification into one step. However, when compared to YOLO in this article[15], it was found that YOLO processes more frames per second than SSD, making it the faster model of the two. It was also concluded that "YOLOv8 is ideal for real-time applications where real-time analysis is crucial". For these reasons, the latest version of YOLO was chosen as the detection model for this project, YOLOv8.

5.5 Move_base

To fulfil F.3, F.6, and F.7 of the Functional Requirements, a package capable of sending movement goals to the robot was needed. Move_base is a ROS package that combines several path planning algorithms to drive a robot from its current position to a goal position, whilst avoiding obstacles encountered along the way. As discussed in the literature review of path planning algorithms, Reactive-Computing-Based algorithms are suitable for real-time obstacle avoidance. Move_base uses this type of planning algorithm for its local planner, which works in tandem with its global planner to achieve obstacle-avoiding movement from the robot's current position to the set goal position. For these reasons, Move_base was chosen as the movement package for this project.

5.6 State Machines and SMACH

During the execution of the resulting application, the robot will exhibit different modes of operation, such as "Following Person", "Staying Still", and "Searching For Person". To fulfil all of the Functional requirements successfully, a medium for representing these different modes of operation and the transitions between them is needed. A state machine can be used to achieve

this. Formally, a state machine is a computational model compromised of a finite number of states, transitions between those states, and actions that result from the state changes. In this scenario, the states of a state machine can be used to represent the different modes of operation the robot will be in. To create a state machine inside the application, the SMACH[16] (State MACHine) ROS package was used.

The state machine used to control the robot's behaviour consists of five states: Initial, Following, Idle, Searching, and Final. On starting the application, the state machine begins at the Initial state and then transitions to the Following state. Here the robot detects a person to follow and starts following them. If the robot reaches the person, the state machine transitions to the Idle state. Here the robot checks if the person has started moving again. If the person has started moving again, the state machine transitions back to the Following state. If the person leaves the field of view, the state machine transitions to the Searching state. Here the robot performs a search scan of the environment to re-locate the person. If the person is found, the state machine transitions back to the Following state. If, at any point, the execution of the node running the state machine is terminated, the state machine transitions to the Final state.

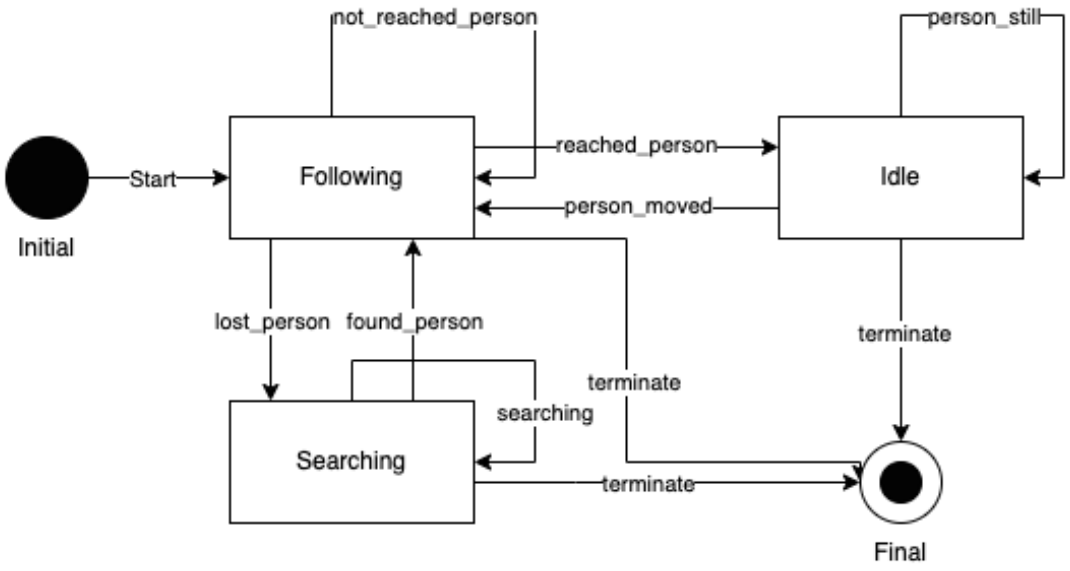

Figure 5.2: Diagram of the state machine used in this implementation .

5.7 Simulation Software

To test the functionality of the implementation during development, robotic simulation software was needed.

Gazebo and Rviz were used for this purpose. Gazebo[17] is a 3D simulation software that can be used to run new and existing 3D environments for testing robots. Rviz[18], short for ROS Visualisation, is also a 3D simulation software, but one that provides a range of tools for observing the robot's point of view during the simulation. On top of showing the view of TIAGo's RGB-D camera, Rviz also shows the coordinates of points in the 3D simulation. Both of these tools were used extensively throughout the development of this project.

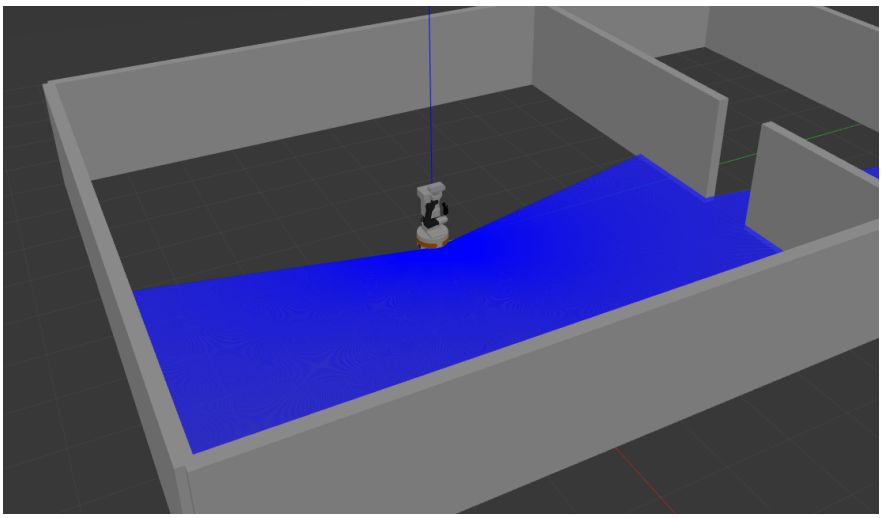

Figure 5.3: The world used for this project loaded into Gazebo.

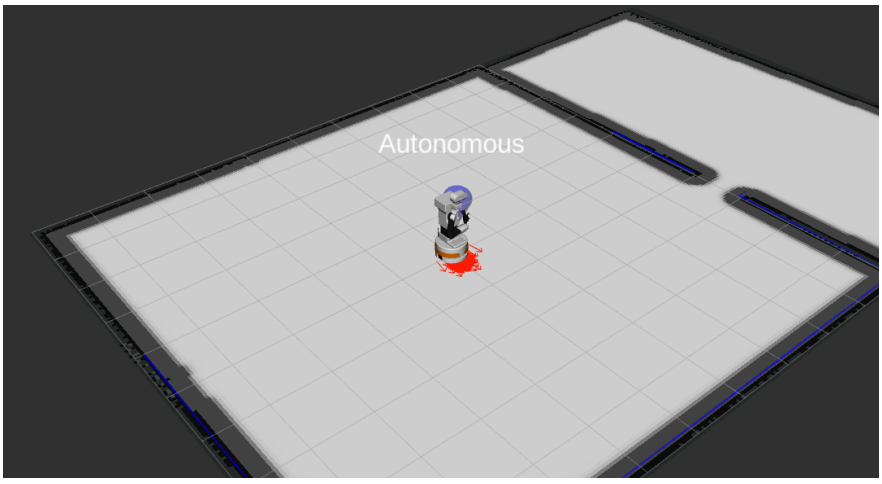

Figure 5.4: The world as seen in Rviz.

Chapter 6

Implementation

This chapter showcases the implementation of the specification and design choices explained in the previous chapter.

6.1 Container

The Container used for this project was made with Singularity[13], using the command `"sudo singularity build robocup_container_v2.sif robocup_container_v2.def"`. As mentioned in the previous chapter, this Container includes all dependencies needed for TIAGo, as well as ones added later in development.

The command needed to start the Container was `"singularity run robocup_container_v2.sif"`, which needed to be entered every time development on this project was carried out. To save time, the .bashrc file of the terminal was edited with the alias ”start=singularity run”, which made starting the Container faster. A .bashrc file is a configuration script that is run every time a terminal starts.

6.2 Programming Langauge

The code for the nodes of this project was written entirely in Python3[19] using the PyCharm[20] IDE. An IDE, short for Integration Development Environment, is a source code editor designed to provide a host of useful development tools, such as file exploration, intelligent code completion, and debugging. Due to the large nature of the workspace for this project, the ability to navigate the file structure from within the source code editor was very helpful.

6.3 Vision

Person detection and colour recognition were implemented as follows:

6.3.1 Person Detection

As mentioned in the previous chapter, YOLOv8 was chosen to handle the object detections of this implementation. This object detection model was integrated into the application as a service stored in a package called **lasr_vision_yolov8**. Within this package, a **yolo.py** script can also be found, which is imported by the service node to provide the functionality of the YOLO detection model. The service definition file, **YoloDetection.srv**, can be found inside a package called **lasr_vision_msgs**. This file defines the request and response messages for the service. The structure of these messages can be seen below:

```
1 sensor_msgs/Image image_raw
2 string dataset
3 float32 confidence
4 float32 nms
```

Listing 6.1: YoloDetection.srv request message

Above is the **request** message. The image_raw field stores the image to be examined. The dataset field stores the dataset to be used by the detection model. For this project, the yolov8n-seg.pt dataset was used. The confidence field stores the minimum confidence to be included in the resulting responses. The nms field stores a guiding value to remove overlapping bounding boxes.

```
1 lasr_vision_msgs/Detection[] detected_objects
```

Listing 6.2: YoloDetection.srv response message

Above is the **response** message. The detected_objects field stores an array of all the detected objects. These are of type Detection, another message from the **lasr_vision_msgs**, whose definition file is denoted by the .msg extension.

```
1 string name
2 float32 confidence
3 int32[] xywh
```

```
4 int32[] xyseg
```

Listing 6.3: Detection.msg

Above is the **Detection** message. The name field stores the classification of the detected object, e.g., person, cat, dog, etc. The confidence field represents how certain the detection model is that the classification is correct. The xywh field stores the dimensions of the bounding box around the detected object. The xyseg field stores the x and y coordinates of the points within the mask segment of the detected object.

To allow nodes to communicate with each other, the ROS Master first needs to be set up. This is achieved using the command `roscore`. Once this has been done, a sim.launch file is executed using the command `roslaunch move_tiago sim.launch` to start both this service and the Gazebo simulation. This launch file can be found inside the main package for this project, **move_tiago**. Once the service is up and running, the main node makes a request for the service. To do this, the main node instantiates the **Detect** class (found in detect.py) that contains the method **detect**. Within this method, the service request is created. A diagram illustrating the architecture of this setup can be seen below:

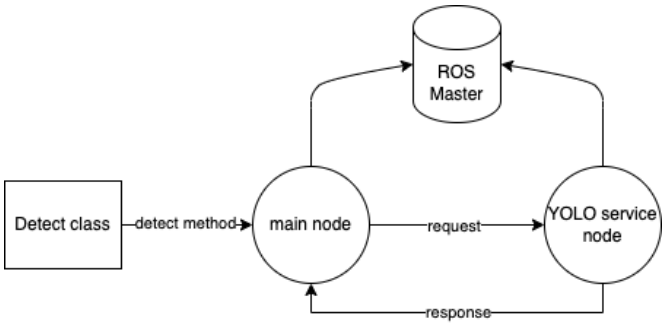

Figure 6.1: Diagram showing the interaction between the main and service nodes.

6.3.2 Colour Recognition

Colour is recognised by the `get_person_colour` method inside the Detect class. The process of detecting colour is similar to that described in the literature review on point estimation. Essentially, the mask segment received from the YOLO service response message is applied to the point cloud to filter out all of the points that correspond to the detected object, which in this scenario is the person to be followed.

Each of these person points possesses RGB data. RGB represents a tuple that consists of values of red, green, and blue that are between 0 and 255. These different values of red, green, and blue are combined to determine an overall colour.

The RGB values for each individual person point are then summed and averaged by the total number of person points. The averaged red, green, and blue values are then looked up in a dictionary of pre-defined colours to determine which colour matches the best. The point cloud data is received by subscribing to the topic **/xtion/depth_registered/points**.

```
1 pcl = rospy.wait_for_message('/xtion/depth_registeredpoints',
    PointCloud2)
```

Listing 6.4: Getting point cloud data

The use of `rospy.wait_for_message` allows for a one-time subscription to the topic, mitigating the need for a subscriber to be created. A demonstration of the `get_person_colour` method in action can be seen below:

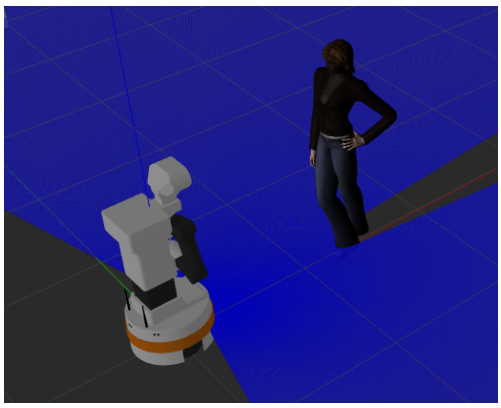

Figure 6.2: View of the world

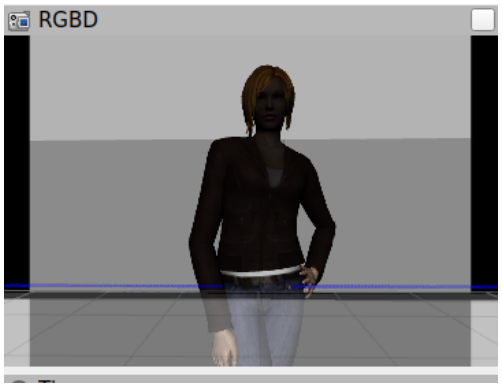

Figure 6.3: View from the RGB-D camera

```
[INFO] [1712079080.212544, 325.531000]: Found person, colour is black
```

Figure 6.4: Output to terminal.

6.3.3 Re-Identification

Re-identification is achieved by storing the colour returned from `get_person_colour` inside the ”person_colour” field of the **Detect** class. Initially, this field is set to empty but is filled when TIAGo detects the colour of the first person to be seen. If subsequent colour detections match that stored in ”person_colour”, TIAGo will advance towards the person. If this is not the case, TIAGo will remain in its current position. If the person leaves and re-enters the field of view, TIAGo will re-identify the person as the target person due to the detected colour and the stored colour being equal.

```
# initialise person colour
if not self.person_colour:
    self.person_colour = colour
    rospy.loginfo(f"Set person's colour to {self.person_colour}")
    self.found = True
    return x, y
    # check if detected colour matches stored colour
elif colour == self.person_colour:
    self.found = True
    return x, y
else:
    rospy.loginfo("Not our person")
    self.found = False
    return None
```

Listing 6.5: Re-identification code

6.4 Navigation

Point estimation and autonomous movement were implemented as follows:

6.4.1 Point Estimation

Estimating the detected person's relative position in the world is done by the **get_person_pose** method inside the Detect class. The process of estimating this relative position is similar to that described in the literature review on point estimation. Once again, the mask segment received from the YOLO service response message is applied to the point cloud to filter out all of the points that correspond to the detected object, which in this scenario is the person to be followed.

Each of these person points also possesses x, y, and z coordinates. The x, y, and z values for each individual person point are then summed and averaged by the total number of person points. The averaged x, y, and z values then constitute the point of the person. However, this point is relative to the **xtion_rgb_optical_frame** frame. Frames are essentially coordinate systems used to define the location and orientation of a point in 3D space with respect to something. For instance, a frame could be TIAGo's camera or the map TIAGo is in. The point needs to be relative to the **map** frame for it to represent the correct point in the world. Points can be adjusted to the desired frame through transformations.

To perform a transformation on the person point from the **xtion_rgb_optical_frame** frame to the **map** frame, the **tf2_ros** and **tf2_geometry_msgs** packages were imported into the Detect class. A TF Buffer was created to store all the transformations between frames, and a TF Listener was created to gather available transformations to populate the TF Buffer with.

```
1 self.tfb = tf2_ros.Buffer()
2 self.tf_listener = tf2_ros.TransformListener(self.tfb)
```

Listing 6.6: Tf Buffer and Listener

The TF Buffer is then queried for a transformation between **xtion_rgb_optical_frame** frame to the **map** frame:

```
1 point = Point(x=average_x, y=average_y, z=average_z)
2 point_stamped = PointStamped()
3 point_stamped.point = point
```

```
point_stamped.header.frame_id = pcl.header.frame_id
point_stamped.header.stamp = rospy.Time(0)

try:
    point_stamped = self.tfb.transform(point_stamped, "map")
    rospy.loginfo("Point transformation successful")
except TransformException as e:
    rospy.loginfo(f"Transform unavailable {e}")
    pass

return point_stamped
```

Listing 6.7: Transformation code

The returned point_stamped represents the person point relative to the map frame.

6.4.2 Movement to Point

Movement to the point returned from the **get_person_pose** method is conducted using the **move_to_person** method, which can be found in the **Move** class (found in move.py).

move_to_person takes the output of **get_person_pose** as input and uses its x and y coordinates to create a Pose. A Pose is a data type that stores information about the position and orientation of an object.

```
def move_to_person(self, coords):

    dest = Pose()

    dest.position.x = coords[0]
    dest.position.y = coords[1]
    dest.position.z = 0.0
    dest.orientation.w = 1.0
    rospy.loginfo(f"Going to {coords[0]}, {coords[1]}")
```

```
        self.b.sync_to_pose(dest)
```

Listing 6.8: move_to_person

Lines 5 and 6 show the x and y components of the person point, stored as a tuple called "coords", being inserted into the Pose.

This Pose, "dest", represents the desired point TIAGo should move towards (the goal). This goal is sent to the **Move_base** action server through the **sync_to_pose** method of the **BaseController** class.

Action servers facilitate the execution of long-duration tasks that might require intermittent feedback. Action servers are called by action clients. Below is the action client for the **move_base** action server created by the **BaseController** class.

```
1 self.__client = actionlib.SimpleActionClient("move_base",
    MoveBaseAction)
```

Listing 6.9: move_base action client

Through this client, the goal position is sent to the move_base action server.

```
1 goal = MoveBaseGoal()
2     goal.target_pose.header.frame_id = 'map'
3     goal.target_pose.header.stamp = rospy.Time.now()
        goal.target_pose.pose = pose
4
5     rospy.loginfo("base is going to (%.2f, %.2f, %.2f) pose",
        pose.position.x, pose.position.y, pose.position.z)
6
7     self._goal_sent = True
8     self.__client.send_goal(goal, done_cb=done_cb)
```

Listing 6.10: goal getting sent to move_base action server through action client

6.4.3 Search Scan

The search scan is implemented through the **recovery_scan** method of the **Move** class. This method calculates a rotation of $\frac{\pi}{4}$ radians (45 degrees) and uses it as input to the **rotate** method. **rotate**, a method also from the **BaseController** class, creates a pose containing this rotation that is sent to the MoveBase action server. The result is a rotation of TIAGo by 45 degrees. After each rotation, TIAGo scans its field of view.

6.5 State Machine

The state machine of this implementation is stored within the main node and is started with the command `rosrun move_tiago main.py`. The states are described as follows:

6.5.1 INITIAL state

The aptly named state INITIAL is the first state visited upon the state machine's start-up. This state has a single outcome, "start", which transitions the state machine to the state FOLLOWING. Upon execution of this state, "start" is returned, transitioning the state machine to the FOLLOWING state.

6.5.2 FOLLOWING state

FOLLOWING is the state that performs the main functionality of this implementation, the following behaviour. Here, both the **Detect** and **Move** classes are instantiated to make use of the **detect** and **move_to_person** methods. **detect** is called to acquire the point of the detected person, which is given to **move_to_person** to move TIAGo to the person. This state has three outcomes, "reached_person", "not_reached_person", and "lost_person". If the target person leaves the field of view, "lost_person" transitions the state machine to the state SEARCHING. If TIAGo reaches the target person, due to a temporary or permanent stop in the person's movement, "not_reached_person" transitions the state machine to the state IDLE. To determine if TIAGo has reached the target person, the **is_tiago_within_range** method of the **Detect** class is called. If the target person is still within the field of view and TIAGo has not reached them, "not_reached_person" keeps that state machine at state FOLLOWING.

6.5.3 IDLE state

IDLE is the state that keeps TIAGo stationary when it is within close proximity to the target person. This state has two outcomes, ”person_moved” and ”person_still”. If the target person has started moving again, indicated by the **is_tiago_within_range** returning false, ”person_moved” transitions the state machine back to the state FOLLOWING. If the person has not moved, indicated by **is_tiago_within_range** returning true, ”person_still” keeps the state machine at state IDLE.

6.5.4 SEARCHING state

SEARCHING is the state that makes TIAGo perform a recovery scan of the environment when the target person leaves the field of view. This state has two outcomes, ”found_person” and ”searching”. If the **detect** method re-identifies the target person, ”found_person” transitions the state machine to state FOLLOWING. Whilst the target person remains lost, ”searching” keeps the state machine at state SEARCHING.

Chapter 7

Evaluation

This chapter aims to evaluate the project by assessing how well its objectives have been met. To recap, the objectives outlined in the Introduction of this paper were as follows:

- **Detect** and **move** towards a person in the field of view.
- **Identify** and **re-identify** the target person by some physical attribute. In the event other people enter the field of view, the robot should recognise the target based on this attribute. I have chosen to use the colour of the person's clothing for this.
- Perform a **recovery scan** of the environment when the target person leaves the field of view.

The Implementation chapter of this paper showed how each of these objectives was implemented. All of these objectives have been evaluated. However, as the first objective,

- **Detect** and **move** towards a person in the field of view.

defines the most essential functionality of this implementation, it has been evaluated the deepest. This has been achieved by breaking it down into two main aspects: person detection and autonomous navigation. Tests on both of these aspects have been conducted, the results of which have been discussed below:

7.1 Person Detection Evaluation

YOLOv8 accurately classified 3D Gazebo person models when placed within TIAGo's field of view. Not once during real-time testing in Gazebo were the models incorrectly classified as anything other than people. However, TIAGo's RGB-D camera was not always able to detect the person models when placed at different distances from it. A test of the detection success rate of a Gazebo person model placed at 30 distances from TIAGo was conducted. The results can be seen below:

Distance 1 = 0 - 0.8 m Distance 2 = 0.9 - 80m Distance 3 = 81m +

	Distance 1 (close)	Distance 2 (middle)	Distance 3 (far)
Person detected	4	10	1
Person not detected	6	0	9
Success rate	40%	100%	10%

Figure 7.1: Results of testing detection success rate from different distances.

Fortunately, the low detection success rate at Distance 1 is not an issue for this implementation, as TIAGo should become stationary when within this distance anyway, and no longer need to detect the target person. For the majority of person-following applications, the robot will operate within the range of Distance 2, as can be seen from Gita's[3] user manual[22]. However, for some person-following applications where the person being followed needs to operate at distances within the range of Distance 3, such as those employed in the logistics industry, its low detection success rate can be a limiting factor. This limitation stems from the hardware rather than the software. Therefore, a solution to this limitation would be to use a camera capable of seeing further than TIAGo's RGB-D camera.

7.2 Autonomous Navigation Evaluation

As mentioned in the Specification and Design chapter, Move_base was used to navigate TIAGo from its current position to the position of the target person, whilst avoiding obstacles encountered along the way.

Obstacles were avoided regularly when placed between TIAGo and the person model. However, TIAGo rarely pursued the person model afterwards and would instead become stationary. To resolve this, the person model would have to be moved back in front of TIAGo so the following could start again. Whilst damage to TIAGo and the obstacle were both prevented by moving around it, TIAGo was unable to maintain its following behaviour. Unfortunately, this is a significant limitation that limits the scope of this implementation. In its current state, this person-follower would struggle in dynamic environments where fast obstacle avoidance is a necessity, such as the outdoors.

Prior to the design explained in the Navigation section of the Implementation chapter, TIAGo was moved using a different method. Initially, the **detect** method would publish the x and y values of the detected person to a custom topic called "coordinates". The **Move** class would subscribe to this topic and send the goal Pose to the Move_base action server through a callback function. This method allowed TIAGo to react quickly to changes in the target person's position. However, due to the constant publication of data to the "coordinates" topic, the callback function would constantly be called. This meant that when TIAGo would reach a stationary target person, instead of stopping, it would continue to move to random points around the map.

This motivated the design mentioned in the Implementation chapter. By switching to the **move_to_person** method, movement of TIAGo could be paused whilst in close proximity to the stationary target person. As a result, this method was superior at moving TIAGo to the correct position. A test of the accuracy of this design was conducted, and the results can be seen below:

	Forwards	Backwards	Left	Right
Correct position	(1.45, 0.040)	(-1.45. 0.040)	(1.04, -3.01)	(1.04, 3.01)
Actual position	(1.45, 0.040)	(-1.45. 0.040)	(1.04, -3.01)	(1.04, 3.01)
Percentage error	0%	0%	0%	0%

Figure 7.2: Results of testing the accuracy of the move_to_person method.

Despite providing significantly more accurate movement, this design takes longer to move TIAGo than the previous one. This is an unfortunate characteristic of this design, as it limits

this implementation's real-time capabilities. In practice, a person may have to wait for TIAGo to catch up with them. However, it can be argued that the improved safety this design provides by making TIAGo's navigation more accurate and ensuring TIAGo stops within close proximity to the target person, still makes this a better design.

7.3 Re-Identification Evaluation

Using the colour of the person model's clothing for re-identification worked successfully in scenarios with two person models wearing different-coloured clothes. However, in scenarios with two person models wearing the same-coloured clothes, TIAGo would pursue either person model. This is the biggest limitation of this method; any person will be identified as the original person if they are wearing the same-coloured clothes. For person-followers that operate in social environments like the outdoors, this can be a big problem as it is very probable at least two people will be wearing the same-coloured clothing.

One potential workaround to this problem would be to require the person being followed to wear an uncommonly worn colour, such as yellow, as stated by this article on colour attractiveness[23].

7.4 Search Scan Evaluation

By scanning the environment after every 45-degree rotation, TIAGo successfully re-found person models in scenarios when they were placed in the same room. However, if a person model were placed in a different room or hidden behind some structure, this search scan would not be able to find them. This could be resolved by implementing search behaviour that moves TIAGo to known rooms in the vicinity. However, moving from its current position may cause TIAGo to become lost to the person operating it. As a result, the implemented search scan seems suitable for situations where a robot's safety and security are of priority.

Chapter 8

Legal, Social, Ethical and Professional Issues

This chapter describes the different concerns surrounding the use of robots in the service industry. As stated by this article from EHL Insights [21], there are five main concerns. These concerns are listed below:

- **Social cues**: Robots should deliver a service that is as human-like as possible, and thus should possess social features. That being said, service robots should never hinder or replace interactions between people.
- **Trust and safety**: The safety of service bots to humans should always be considered when discussing their use in industry.
- **Autonomy**: The level of a service bot's autonomy should be controllable, especially when the consequences of a robot's actions are unpredictable.
- **Responsibility**: A robot's responsibility for its actions is important to consider before users engage with it. Therefore, before service bots are deployed, who is responsible for the robot's actions must be clearly defined. Moreover, for liability reasons, the actions of a service bot should always be traceable.
- **Human worker replacement**: Companies should consider the input of their employees on matters such as what kind of robot is used and what its defined tasks should be. In the event a service bot assumes a worker's job, they should be retrained for a new role.

For each of these concerns, this implementation provides a solution.

- **Social cues**: TIAGo's main functionality is following people in this implementation. As a result, it does not perform any intelligent, communicative functions that could potentially replace human-to-human interaction.
- **Trust and safety**: TIAGo operates at a top speed of 1 m/s. This speed allows it to navigate 3D spaces effectively whilst also prioritising human safety. Additionally, the inclusion of the IDLE state makes TIAGo stop moving when in close proximity to the target person. This prevents TIAGo from colliding with the target person when following them. As a result, the safety of users is adequately considered.
- **Autonomy**: The autonomy of TIAGo can be controlled by terminating the application. This can be done at all times by using `ctrl-c` on the main node.
- **Responsibility**: The actions of TIAGo can be traced using the ROS logs displayed in the terminal. These logs detail every action TIAGo performs.
- **Human worker replacement**: This implementation requires a human operative, a person to be followed. As a result, in the industries where this technology can be applied, there will always be a need for someone to operate the robot. Therefore, this implementation only assists human workers and does not replace them.

Chapter 9

Conclusion

This project ultimately achieved its primary goal of developing a robotic person-follower capable of person-following, identification and re-identification, and search behaviour. Whilst some functionalities remain sub-optimal, such as obstacle avoidance, all features outlined in the Introduction's objectives have been implemented. Here are the key findings from this project:

Using YOLOv8 as the detection model proved to be a good design choice. It provided fast and accurate results when detecting person models, and no false classification was produced during development. For these reasons, it seems suitable to recommend YOLO as an object detection model for person-followers.

When conducting tests to validate the accuracy of YOLO, it was discovered that TIAGo's RGB-D camera is most effective when detecting objects between 0.9 and 80 metres. The specifications of existing robotic person-followers, such as Gita[22], show that they tend to operate well within the 0.9 - 80 metre range. As a result, it seems suitable to suggest the use of TIAGo's RGB-D camera for the vast majority of person-following applications.

Using a state machine to model TIAGo's different modes of operation also proved to be a good design choice. During development, the need for TIAGo to become stationary when in close proximity to the target person arose. The state machine allowed this functionality to be smoothly integrated into the existing system. From this, state machines seem suitable for modelling robot behaviour that supports extendable functionality through smooth integration.

Chapter 10

Future Work

As mentioned in the Conclusion, some functionalities of this implementation remain sub-optimal. For future work, obstacle avoidance could be extended to allow TIAGo to resume following after avoiding an obstacle. Additionally, re-identification could be improved to prevent multiple people from being identified as the same person. This could be achieved by re-identifying characteristics that are unique to each person, such as biometric features like a person's face. There are many existing facial-recognition libraries to do this, such as OpenCV [24] and Python's Face Recognition [25]. Lastly, TIAGo's gripper could be utilised to extend this implementation with luggage-carrying capabilities.

References

[1] ZMP INC. Overview of the CarriRo logistics support robot [Internet]. Available from: https://www.zmp.co.jp/carriro/logistics-support [accessed: 21/3/24].

[2] ZMP INC. Yamato Transport Co., Ltd. case study [Internet]. Available from: https://www.zmp.co.jp/en/carriro/casestudy/detail_yamato.html [accessed: 21/3/24].

[3] Piaggio Fast Forward. Overview of the Gita Robot [Internet]. Available from: https://piaggiofastforward.com/ [accessed: 20/3/24].

[4] Sezer V, Salim Zengin R, Houshyari H, Cenk Yilmaz M. Conversion of a Conventional Wheelchair into an Autonomous Personal Transportation Testbed [Internet]. Service Robotics. 2020. Pages 1-2. Available from: http://dx.doi.org/10.5772/intechopen.93117 [accessed: 15/3/24]

[5] iRobot. Overview of the Roomba robot [Internet]. Available from: https://www.irobot.co.uk/en-gb/roomba.html [accessed: 29/1/24]

[6] PAL Robotics. Overview of the TIAGo robot [Internet]. Available from: https://pal-robotics.com/robots/tiago/ [accessed: 29/3/24]

[7] Ubuntu. Explanation of ROS [Internet]. Available from: https://ubuntu.com/robotics/what-is-ros [accessed: 14/3/24]

[8] Lohia, Aditya; Kadam, Kalyani Dhananjay; Joshi, Rahul Raghvendra; and Bongale, Dr. Anupkumar M. Bibliometric Analysis of One-stage and Two-stage Object Detection [Internet]. 2021. Pages 2-3. Available from: https://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=9123&context=libphilprac [accessed: 21/3/24]

[9] Rollo F, Zunino A, Raiola G, Amadio F, Ajoudani A, Tsagarakis N. FollowMe: a Robust Person Following Framework Based on Visual Re-Identification and Gestures [In-

ternet]. 2023. Pages 1-2. Available from: https://arxiv.org/pdf/2311.12992.pdf [accessed: 22/3/24]

[10] Sánchez-Ibáñez, J.R.; Pérez-del-Pulgar, C.J.; García-Cerezo, A. Path Planning for Autonomous Mobile Robots: A Review [Internet]. 2021. Pages 19-20. Available from: https://doi.org/10.3390/s21237898 [accessed: 22/3/24]

[11] SoftBank. Overview of the Pepper robot [Internet]. Available from: https://us.softbankrobotics.com/pepper [accessed: 29/3/24]

[12] Encyclopedia Britannica. Linux [Internet]. 2024. Available at: https://www.britannica.com/technology/Linux [accessed: 29/3/24]

[13] Singularity. Introduction to Singularity [Internet]. Available at: https://docs.sylabs.io/guides/3.5/user-guide/introduction.html [accessed: 29/3/24]

[14] Ultralytics. Overview of YOLOv8 [Internet]. Available at: https://docs.ultralytics.com/ [accessed: 16/1/24]

[15] Keylabs.ai. YOLOv8 vs SSD: Choosing the Right Object Detection Model [Internet]. 2023. Available at: https://keylabs.ai/blog/yolov8-vs-ssd-choosing-the-right-object-detection-model/ [accessed 30/3/24]

[16] ROS.org. SMACH documentation [Internet]. Available at: https://wiki.ros.org/smach [accessed: 30/3/24]

[17] Gazebo. Gazebo [Internet]. Available at: https://gazebosim.org/home [accessed: 30/3/24]

[18] Rviz. Rviz [Internet]. Available at: http://wiki.ros.org/rviz [accessed: 30/3/24]

[19] Python.org. Python3 [Internet]. Available at: https://www.python.org/downloads/ [accessed: 30/3/24]

[20] JetBrains. PyCharm [Internet]. Available at: https://www.jetbrains.com/pycharm/ [accessed: 30/3/24]

[21] Dr Reza Etemad-Sajadi. 6 main ethical concerns of service robots and human interaction [Internet]. 2022. Available at: https://hospitalityinsights.ehl.edu/service-robots-and-ethics [accessed: 30/3/24]

[22] Piaggio Fast Forward. Gita's user manual [Internet]. Page 14. Available at: https://f.hubspotusercontent10.net/hubfs/6286664/UserManual-online-only_v1.5.pdf [accessed: 28/3/24]

[23] Duje Kodžoman, Aleš Hladnik, Alenka Pavko Čuden, Vanja Čok. Exploring color attractiveness and its relevance to fashion [Internet]. 2021. Available at: https://onlinelibrary.wiley.com/doi/10.1002/col.22705#:~:text=Thus%2C%20the%20most%20attractive%20color,preferred%20color%20(Figure%201). [accessed: 6/4/24]

[24] opencv.org. OpenCV [Internet]. Available at: https://opencv.org/ [accessed: 6/4/24]

[25] pypi.org. Face-Recognition [Internet]. Available at: https://pypi.org/project/face-recognition/ [accessed: 6/4/24]