Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Nikolai Olav Persen

NTNU Cyborg Robot Vision - Implementing VSLAM and object detection in ROS

Master's thesis in Cybernetics and Robotics
Supervisor: Hendseth, Sverre
Co-supervisor: Knudsen, Martinius
June 2021

Cyborg Project

NTNU
Norwegian University of Science and Technology

Nikolai Olav Persen

NTNU Cyborg Robot Vision - Implementing VSLAM and object detection in ROS

Master’s thesis in Cybernetics and Robotics
Supervisor: Hendseth, Sverre
Co-supervisor: Knudsen, Martinius
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Problem description

The NTNU Cyborg project began in 2015 and involves an interdisciplinary development of a robot interfaced with biological neural networks. The project aims at having a robot (i.e., the Cyborg) roam around Glassgården at NTNU Gløshaugen autonomously while interacting with the students it encounters. An important contribution towards autonomy is the implementation of robot vision. Robots today can perform a large variety of tasks without the use of a camera. However, sometimes a camera is necessary when performing certain tasks, such as object detection and generating accurate 3D maps of its environment. To contribute towards making the Cyborg become autonomous, the student shall:

- Update the old Unified Robot Description Format (URDF) to visualize the current version of the Cyborg and display the model in RViz. This includes the body panels and the LED dome of the Cyborg and a separate model adding the ZED stereo camera.
- Implement and evaluate Visual Simultaneous Localisation and Mapping (VSLAM) with the Cyborg using the ZED stereo camera.
- Implement and evaluate object detection using You Only Look Once (YOLO) with the ZED stereo camera to detect students
- Demonstrate a navigation controller in Robotic Operating System (ROS) capable of detecting and following students using YOLO.

Abstract

The NTNU Cyborg project aims at having the Cyborg roam around Glassgården at NTNU Gløshaugen autonomously while interacting with the students it encounters. An essential contribution to this is the implementation of a robust and permanent robot vision module, enabling means of performing VSLAM and object detection. The latter has been implemented before in Robotic Operating System (ROS). However, it has not been developed using an updated Unified Robot Description Format (URDF) model and fully integrated with RViz. RViz is a popular 3D visualization tool in ROS used for task performance and displaying data such as camera images, lasers, robot models and more.

This report aims to create a more robust robot vision by integrating the ZED stereo camera with an updated robot model to perform object detection and VSLAM in RViz. 3D computer-aided design (CAD) models from SolidWorks, created by previous students, were exported as URDF files and added to the old robot model provided by Adept MobileRobots. The new robot model was used to perform VSLAM with Real-Time Appearance-Based Mapping (RTAB-Map) in ROS. Moreover, You Only Look Once (YOLO) object detection was implemented to detect students and publish depth measurements. The measurements were used as input to a navigation controller designed to follow students. A coordinate frame was created to track the position of a detected student in RViz continuously.

The new robot model was successfully created and used throughout the report. The position of the detected student was published to RViz and sent as navigation goals to the navigation controller. The Cyborg managed to follow the student inside the Cyborg office and in a simulated environment. Finally, VSLAM produced a 2D and 3D map of the Cyborg office with satisfactory accuracy.

Abstrakt

NTNU Cyborg prosjektet har som mål å ha en Kyborg vandre autonomt rundt i Glassgården ved NTNU, mens den har interaksjoner med studentene den møter. Et essensielt bidrag til dette er implementering av en robust robotsyn modul, som muliggjør utførelse av Visual Simultaneous Localisation and Mapping (VSLAM) og objekt detektering. Sistnevnte har blitt implementert i Robotic Operating System (ROS) før, men har ikke blitt utviklet ved bruk av en oppdatert Unified Robot Description Format (URDF) robot modell integrert i RViz. RViz er et populært 3D visualiseringsverktøy i ROS som blir brukt til utførelse av oppgaver og visualisering av data som kamera bilder, lasere, robot modeller og mer.

Denne rapporten har som mål å utvikle mer robust robotsyn ved å integrere et ZED stereo kamera med en oppdatert robot modell for utførelse av objekt deteksjon og VSLAM i RViz. 3D computer-aided design (CAD) modeller fra SolidWorks, lagd av tidligere studenter, ble eksportert som URDF filer og lagt til i den gamle robot modellen fra Adept MobileRobots. Den nye robot modellen ble brukt til å utføre VSLAM med RTAB-Map i ROS. I tillegg ble You Only Look Once (YOLO) objekt deteksjon implementert for å detektere studenter og publisere dybdemålinger. Disse målingene ble brukt som input til en navigasjonskontroller designet for å følge studenter. Et koordinat system ble lagd for å oppdatere posisjonen til den detekterte studenten i RVIZ kontinuerlig.

Den resulterende robot modellen ble vellykket og brukt gjennom hele rapporten. Posisjonen til den detekterte studenten ble publisert til RViz og sendt som navigasjonsmål til navigasjonskontrolleren. Kyborgen klarte å følge studenten på Kyborg kontoret og i et simulert miljø. Til slutt ble et 2D og 3D kart av Kyborg kontoret lagd ved bruk av VSLAM, med tilstrekkelig nøyaktighet.

Acronyms

AMCL Adaptive Monte Carlo Localization.

API Application Programming Interface.

CAD computer-aided design.

CNN Convolutional Neural Networks.

CUDA Compute Unified Device Architecture.

EiT Experts in Teamwork.

FOV Field of View.

GPU Graphics Processing Unit.

NTNU Norges teknisk-naturvitenskapelige universitet.

R-CNN Region Based Convolutional Neural Networks.

ROS Robotic Operating System.

RTAB-Map Real-Time Appearance-Based Mapping.

SDK Software Development Kit.

SLAM Simultaneous localization and mapping .

SVM Support Vector Machine.

URDF Unified Robot Description Format.

VSLAM Visual Simultaneous Localisation and Mapping.

XML Extensible Markup Language .

YOLO You Only Look Once.

Contents

Abstract i

Abstrakt ii

Acronyms iii

1 Introduction 1

1.1 The NTNU Cyborg project 1

1.2 Motivation 1

1.3 Limitation on the Cyborg battery 2

2 ROS - Robotic Operating System 3

2.1 What is ROS? 3

2.2 Nodes 3

2.3 Topics 3

2.4 Messages in ROS 4

2.5 Rosbag - Recording topics 4

2.6 Packages 4

2.7 Roscore 4

2.8 actionlib - Performing long term tasks 5

2.9 tf - How ROS keeps track of coordinate frames 6

2.10 How URDF works 7

2.11 The ROS navigation stack 8

3 Hardware and Software 13

3.1 Hardware 13

3.1.1 ZED stereo camera 13

3.1.2 Desktop computer 13

3.1.3 Jetson Nano Developer Kit 13

3.1.4 The NTNU Cyborg 14

3.2 Software 14

3.2.1 RosAria 14

3.2.2 RViz 14

3.2.3 MobileSim 17

3.2.4 zed_ros_wrapper - ROS wrapper for the ZED camera 17

3.2.5 zed_ros_examples - Collection of ROS examples with the ZED camera 17

3.2.6 ZED SDK 18

4 RTAB-Map - An RGB-D SLAM approach 19

4.1 rtabmap in ROS 19

5 Types of object detection algorithms 21

5.1 R-CNN, Fast R-CNN and Faster R-CNN 21

5.2 YOLO -You Only Look Once 22

6 Creating a Cyborg robot model for display in RViz 23

6.1 Creating the URDF file for the Cyborg . . . 23

6.1.1 Evaluation of the URDF file . . . 26

6.2 Adding the ZED camera to the robot model . . . 28

6.2.1 Evaluation . . . 29

7 Integrating Visual simultaneous localization and mapping (VSLAM) with the Cyborg 32

7.1 Specification and Requirements . . . 32

7.2 Implementing VSLAM . . . 32

7.3 Evaluation of the VSLAM . . . 35

8 Implementing object detection with the Cyborg 38

8.1 Implementing YOLO using the ZED camera . . . 38

8.1.1 Evaluating YOLO object detection . . . 40

9 ROS Navigation - Autonomous tracking of students using object detection 44

9.1 Specification and Requirements . . . 44

9.2 Creating a Navigation controller for the Cyborg . . . 45

9.2.1 Implementation the controller . . . 45

9.2.2 Evaluation of the controller . . . 54

9.3 Creating a Navigation controller in a simulated environment . . . 58

9.3.1 Specification and Requirements . . . 58

9.3.2 Implementation . . . 58

9.3.3 Evaluation of the simulated controller . . . 63

10 Discussion 66

11 Conclusion 67

11.1 State of the Cyborg as of writing this report . . . 67

A Dimensions 68

A.1 Pioneer LX . . . 68

A.2 ZED Stereo Camera . . . 69

B 3D CAD Models 70

B.1 The Cyborg assembly . . . 70

1 Introduction

1.1 The NTNU Cyborg project

The NTNU Cyborg project is an ongoing project, beginning in 2015, that in the end aims at being able to communicate between living nerve tissue and a robot [1]. Additionally, having a robot that serves as a platform for studying neural signaling properties, robotics and hybrid bio-robotic machines, the project also aims to bring NTNU to the forefront of international research within these areas. The project has the potential of integrating different disciplines to the project within areas such as robotics, mechatronics, sensor technology, artificial intelligence, psychology and much more.

Visually, the robot has undergone several changes over the years, as shown in figure 1. Figure 1a shows the 2016 contribution, with a skeleton frame and the motion-sensing device Kinect. In 2017, the ZED camera was used instead, as shown in figure 1b. From figure 1c, the size of the robot was compressed, adding a 3D printed casing and a dome on top. Figure 1d shows the robot as it stands today. The 3D printed casing has been given a paint job, and the NTNU logo was added to the front panel. Moreover, LED strips were added to the dome.

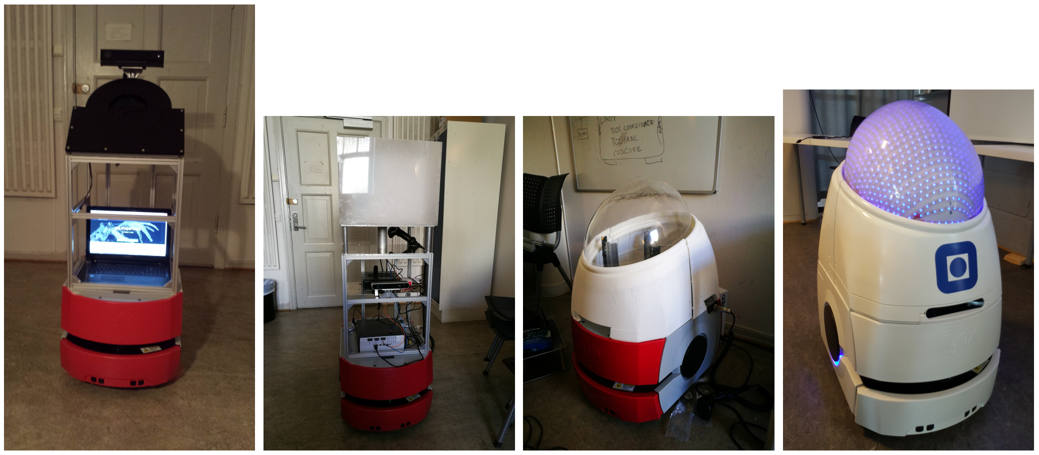

(a) 2016 (b) 2017 (c) 2018 (d) 2019 (current)

Figure 1: The different visual improvements of the Cyborg

1.2 Motivation

The Cyborg is currently intended to showcase the project by having it roam around the hallways at Glassgården at NTNU autonomously while interacting with the people it encounters. An essential contribution to this is the implementation of robot vision. This allows the robot, with the use of state-of-the-art object detection algorithms, to detect objects, e.g., humans, and autonomously navigate to them based on the receiving depth measurements. Additionally, having a camera mounted on the robot can enable 2D and 3D Visual Simultaneous Localisation and Mapping (VSLAM), an important tool towards autonomy. Additionally, having a permanent camera solution enables live monitoring of the robot, as long as it is connected to the internet.

Moreover, implementing an updated robot model for simulation and task performance has several benefits to the Cyborg project: It provides more accurate collision avoidance, an accurate visual representation of the robot and provides new coordinate frames that can be used as a frame of reference when expressing objects relative to it. For example, if a robot has a camera attached to it and wants to express an object detected by the camera relative to the robot, it must know the camera's location in the first place. If the required coordinate frame already has been implemented, any transformation of the coordinate frames can be done using existing packages in ROS.

1.3 Limitation on the Cyborg battery

As of writing this report, the Robot has had - and still has - some problems with its battery. The battery does not charge. Consequently, testing on the Cyborg is limited to the Cyborg office. The main reason is to prevent any excessive use of the battery. This will not be mentioned any further throughout the report.

2 ROS - Robotic Operating System

This section gives an introduction to Robotic Operating System (ROS), and some of its key elements such as ROS nodes, topics, messages and more.

2.1 What is ROS?

ROS is a collection of frameworks and tools, well suited for robots, that provide functionalities to that of an operating system. ROS provides services that are designed for a heterogeneous computer cluster [2], such as hardware abstraction and low-level device control [3]. Its initial release was 14 years ago in 2007 and has over the years released 13 different distributions [4]. The first distribution was released on March 2, 2010, called *ROS Box Turle*. Its most recent release (ROS1) is *ROS Noetic Ninjemys*, released on May 23. 2020. In 2014, ROS2 was announced [5]. The goal of ROS2 is to adapt to the changes that have been made to ROS over the years, leveraging what is great about its previous version and implement the necessary improvements. The latest ROS2 distribution is currently *Galactic Geochelone*, released on May 23. 2021.

2.2 Nodes

Nodes in ROS are processes that perform computation [6] and are important for the flow of information. If a node is sending out information, it means that it is a publisher. If a node is receiving information, then it a subscriber. A simple example using nodes in ROS could be to create a controller node that sends out velocity commands to another node that uses this information to command a robot to move. Nodes can be a publisher and a subscriber, or they can be by themselves. The active nodes can be found using `$ rosnode list` from terminal.

2.3 Topics

When a node publishes information, the information that is sent out is published to a *topic*. For topics to exist, an existing node has to publish to it in the first place. This way, the production of information is decoupled from its consumption, meaning that any existing node can subscribe to a particular topic, given that the information is of interest. There can be multiple publishers and subscribers for a single topic simultaneously, and a single node can publish and/or subscribe to multiple topics. `$ rostopic list` shows all topics that are being published. A simple diagram of two nodes and a topic can be found in figure 2

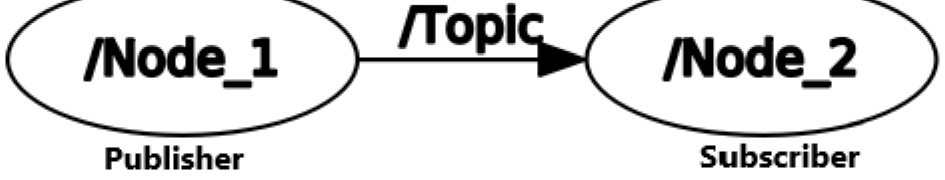


Figure 2: Simple graph of two nodes and a topic

2.4 Messages in ROS

The information that is published to a topic is structured as a ROS *message*. There exist multiple message types in ROS that are used to represent information such as velocity, position, orientation, and more. Each message is defined by a type, and a name, where the type is the type of message (e.g., an integer), and the name is the name of the message variable. The ROS community provides a large variety of messages that can be used in multiple projects. However, sometimes it can be convenient to create a custom message. This can be done if the project at hand requires some information that could be very specific to a type of task. In ROS, messages are labeled *name_ of_ message.msg* and are located in folders called *msgs*.

2.5 Rosbag - Recording topics

ROS provides a set of tools that enables recording of topics called a *rosbag*. This tool can listen to a topic and store all incoming messages. This can be done from terminal using `$ rosbag record <topic1>`. A rosbag can also play back to a topic. For example, if a rosbag has been recording data from a topic containing the string "/hello_world", it can publish its contents back to the topic it originated from with the same message. Rosbags can be very useful when creating a map of an unknown environment. For example, if a LiDAR is used to publish laser scans, it can be stored in a rosbag and played back later to construct the map in RViz. The content of a rosbag can be played from terminal using `$ rosbag play <name_of_bag>`

2.6 Packages

In ROS, the software is organized in packages and can contain nodes, a ROS-independent library, third-party software, configuration files and more that constitutes a module [7]. A ROS package is created by running `$ catkin_create_package`.

When creating a package, two important files are generated in the root of the package: *package.xml* and *CMakeLists.txt*. The package.xml contains information such as the package's name, the author, version numbers, description of the package and the maintainer. It also contains information about dependencies on other packages [8].

CMakeLists.txt is a file that describes how to build the code and where to install it to [9]. It uses commands such as *find_package()* to find other CMake/Catkin packages needed to build, *generate_ messages()* to invoke for example message generation and more.

2.7 Roscore

Roscore is an essential element when running ROS. Communication between nodes is established when roscore is running and is achieved by running `roscore` in terminal. This starts up the following [10]:

- A ROS Master
- A ROS Parameter Server
- A rosout logging node

The ROS Master has the role of enabling individual ROS nodes to locate one another to communicate with each other [11]. The Parameter Server enables nodes to store and retrieve parameters at run time. These types of parameters could be the ROS distribution and the version that is in use [12]. The last one is the rosout logging node, which is the name of the console log reporting mechanism in ROS [13].

2.8 actionlib - Performing long term tasks

Sometimes, when a request has been sent to a node, it is beneficial to receive a reply to the request. It could be the case of knowing whether or not a navigation goal request has been completed or not, or whether the goal is possible to perform or not. All of this is provided by the *actionlib* library, and is achieved through a *client*- and *server* application.

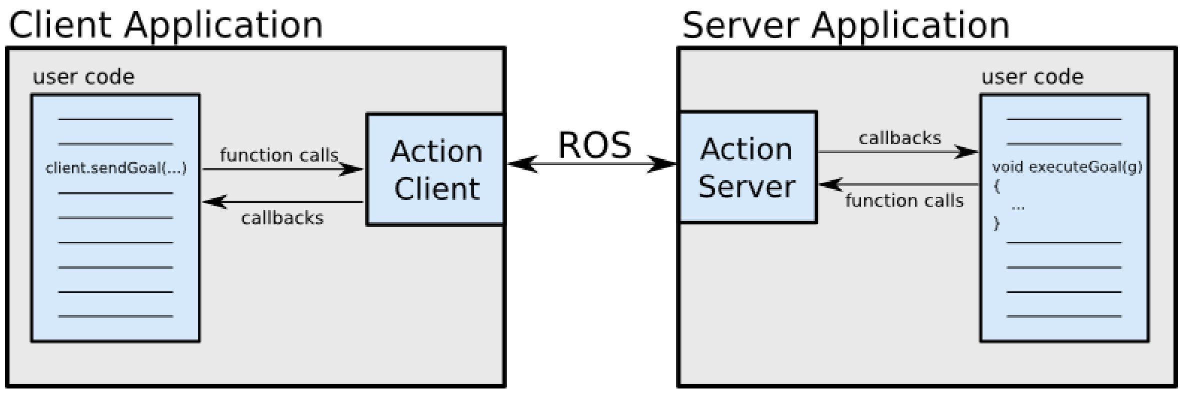


Figure 3: A client- and server application interface [14]

Figure 3 shows how actionlib uses the client and server to communicate. The client application consists of an *actionclient* which can receive function calls from user code and send callbacks to the user code. Similarly, the *actionserver* can send callbacks to user code, and receive function calls. The communication between the client- and server application is via a *ROS Action Protocol*. This protocol consists of ROS topics that have a specified ROS namespace to transport messages. The structure of this action protocol can be seen from figure 4.

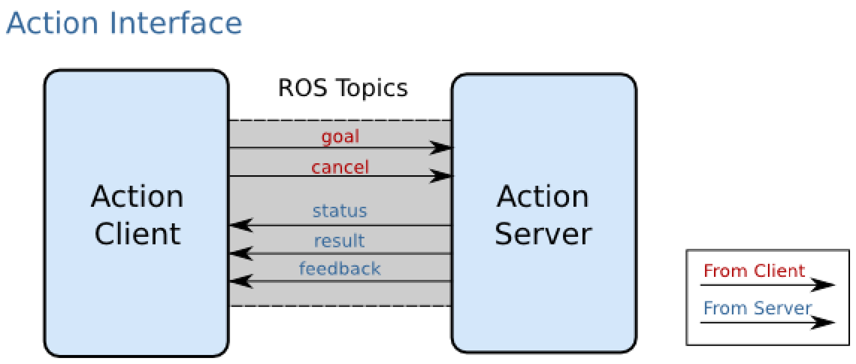


Figure 4: Communication between the ActionClient and ActionServer via a ROS Action Protocol [15]

Three important messages that the ActionClient and the ActionServer uses to communicate are:

- Goal
- Feedback

- Result

The **goal** message is sent to an ActionServer from an ActionClient. The message type of a goal depends on the type of project. In terms of robot navigation, it is common to use move_base_msgs/MoveBaseActionGoal:

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalID goal_id
  time stamp
  string id
move_base_msgs/MoveBaseGoal goal
  geometry_msgs/PoseStamped target_pose
    std_msgs/Header header
      uint32 seq
      time stamp
      string frame_id
    geometry_msgs/Pose pose
      geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
      geometry_msgs/Quaternion orientation
        float64 x
        float64 y
        float64 z
        float64 w
```

The target pose_pose defines the header, position and orientation of the target goal. The header and goal_id contain information about the goal itself.

The **feedback** is responsible for providing information about the process of the goal, e.g., the current position of the robot along a specified path generated from a given goal. This provides a valuable tool to monitor the progress of the goal. This information is sent from the ActionServer to the ActionClient.

The **result** is a message that is sent once, in contrast to the feedback which publishes several times. This message is only published upon completing a given goal, e.g., the final position of a robot after navigating to the end goal. This information is sent from the ActionServer to the ActionClient.

2.9 tf - How ROS keeps track of coordinate frames

Whenever a robot moves, all the corresponding coordinate frames will move relative to a frame of reference. To keep track of these changes, Robotic Operating System (ROS) uses a package called *tf*. This enables ROS to track multiple coordinate frames over time. It tracks all frames in a tree structure. The edges represent transforms, whereas the nodes represent the frames [16]. It uses composite transformations to find positions and orientations of different coordinate frames. An example of a composite transformation between three frames, going from node D via the edge to node B, then from node B via the edge to node A is given by the following equation:

$$\mathbf{T}_D^A = \mathbf{T}_B^A \cdot \mathbf{T}_D^B$$

This is an essential concept, as it allows ROS to constantly keep track of the different parts of the robot relative to the environment by having direct mathematical transformations between them. More information on this package, and how it works, can be found on the ROS Wiki [17] and from its Application Programming Interface (API) documentation [18].

2.10 How URDF works

The Unified Robot Description Format (URDF) is a format used to display a model, e.g., robot, in a simulated environment such as RViz, and is based on two important concepts: links and joints. A robot link is a component that constitutes the physical part of the robot. A joint allows for rotational- or translational movement of a link. A practical example of this is how the human arm works. The forearm is the link and can rotate because it is connected to the elbow, i.e., the joint. If the joint is taken away, then the forearm will no longer be able to move.

URDF files use Extensible Markup Language (XML), and describe some important properties that are needed when defining a robot model:

- *robot name*: Defines the name of the robot that is being created for the particular URDF file.
- *link name*: Defines the name of a particular link
- *visual*: Defines the *visual* properties of the robot, i.e. what the user sees during simulation.
  - *Geometry*: Defines geometric properties for a given link, such as a box or an imported mesh file
  - *origin* - rpy and xyz: Defines the position and orientation of the frame for a given link, respectively. This includes roll (r) pitch (p) and yaw (y), and the x-, y-, z coordinates of the frame. Changing these values will change the position and orientation of the entire link in the simulation.
- *Joint name*: Defines the name of a particular joint
  - *parent link*: Defines the parent link connected by the specified joint.
  - *child link*: Defines the successor link, i.e., the child for the given parent connected by the specified joint.
- *Inertial*: Defines properties such as the mass of the link and the inertia.

Although this is not a complete list that constitutes a fully defined URDF file, it does cover some of the basic elements that are used to define a robot model. The following is a short example of how to create a cylindrical robot model, extracted from the ROS Wiki [19]:

```
<robot name="myfirst">
  <link name="base_link">
    <visual>
      <geometry>
```

```
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
</robot>
```

The code starts with the opening tag <robot name"myfirst"> and closes it with </robot> on line 9. This means that any code between line 1 and line 9 is related to the robot called *myfirst*. This also applies for the the link, which has the opening tags on line 2 and the closing tags on line 8. This means that any code within these tags defines the link *base_ link*.

2.11 The ROS navigation stack

The navigation stack in ROS takes information from a robot's odometry, and sensor readings as input and outputs safe velocity commands for navigation [20]. The sensor readings must be of type *sensor_ msgs/LaserScan* or *sensor_ msgs/PointCloud*. Additionally, odometry must be of type *nav_ msgs/Odometry*, and tf must be running transforms for all coordinate frames. The navigation stack also requires a base controller. The base controller is responsible for sending velocity commands to the robot in the form of a *geometry_ Twist* message to the topic labeled */.../cmd_ vel*. The base controller contains linear and angular velocities in the x-,y- and z-direction. Optionally, a map can be included in the navigation stack as well. Figure 5 shows the configuration of the Cyborg with the navigation stack.

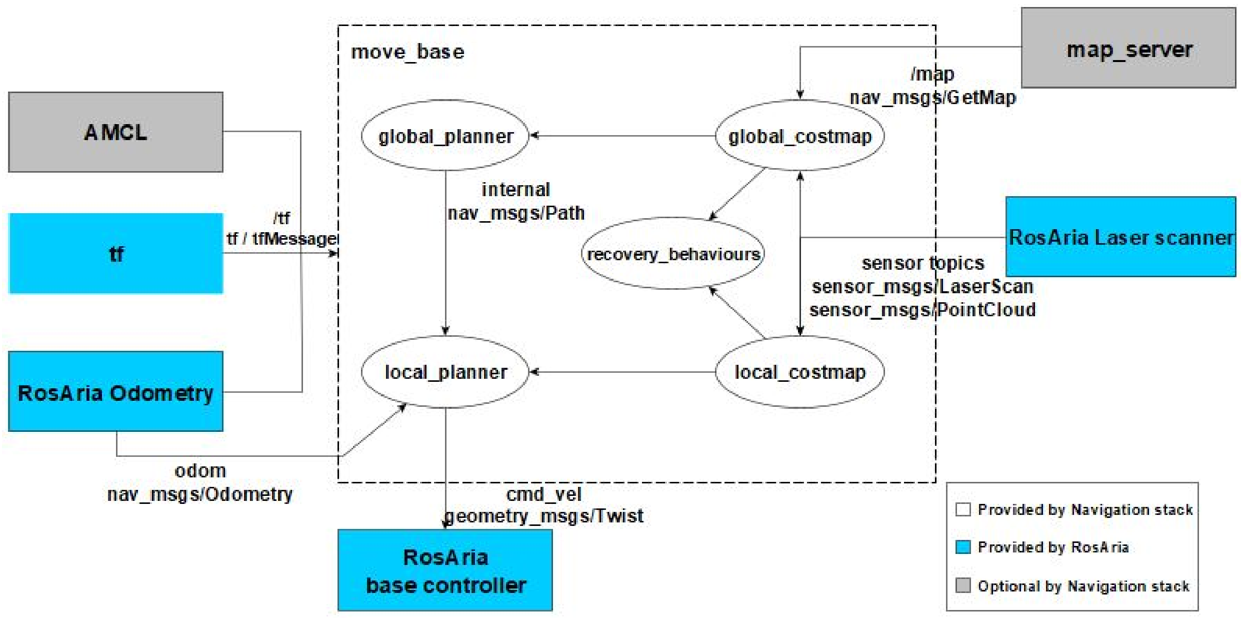


Figure 5: Navigation stack setup with RosAria

The white circles represent the configuration provided by the navigation stack, whereas the blue boxes are platform-specific. The latter corresponds to a controller, odometry source and laser scanner provided by the RosAria node and transform messages provided by tf. The grey boxes are optional data available from the navigation stack to improve the overall system, such as a map and Adaptive Monte Carlo Localization (AMCL). AMCL is a probabilistic localization system for robots moving in 2D and is included in the navigation stack. This algorithm takes a laser-based map, tf messages and laser scans as input and outputs pose estimates.

The node responsible for sending velocity commands to the base controller is *move_ base*. This node is divided into two different planners: *Local planner* and *global planner*. The global planner is responsible for long-term plans over the entire environment, whereas the local planner refers to obstacle avoidance and determining velocity commands to the robot for safe navigation. The *recovery_ behaviors* may be performed in situations where the robot is considered stuck. Moreover, the global- and local planner have two corresponding costmaps: local- and global *costmap* respectively. A costmap is a map that is divided into different cells, where each cell has been given a cost according to some cost function. The cost function determines whether there is an obstacle nearby or not.

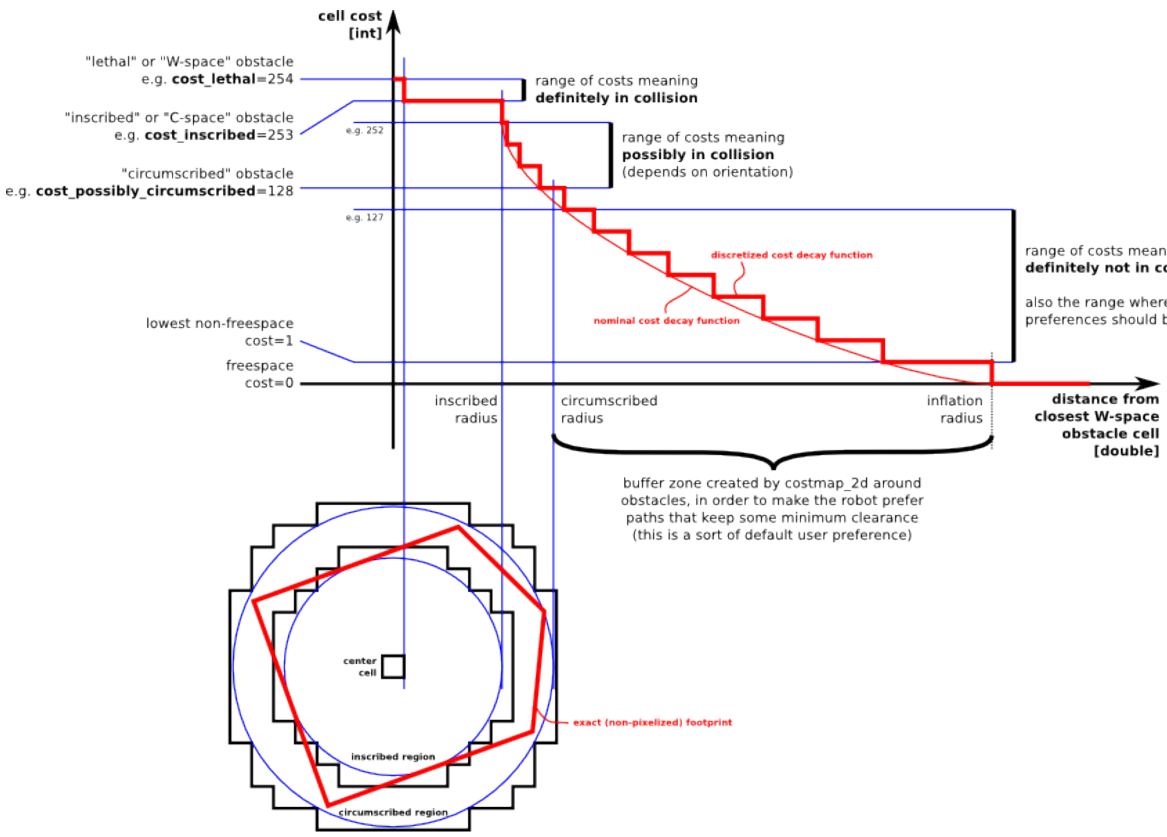

cell cost [int]
"lethal" or "W-space" obstacle e.g. cost_lethal=254
"inscribed" or "C-space" obstacle e.g. cost_inscribed=253
"circumscribed" obstacle e.g. cost_possibly_circumscribed=128
lowest non-freespace cost=1
freespace cost=0
e.g. 252
e.g. 127
range of costs meaning definitely in collision
range of costs meaning possibly in collision (depends on orientation)
range of costs meaning definitely not in collision
also the range where (most) user preferences should be expressed
discretized cost decay function
nominal cost decay function
inscribed radius
circumscribed radius
inflation radius
distance from closest W-space obstacle cell [double]
buffer zone created by costmap_2d around obstacles, in order to make the robot prefer paths that keep some minimum clearance (this is a sort of default user preference)
center cell
exact (non-pixelized) footprint
inscribed region
circumscribed region

Figure 6: Cell cost as a function of the distance from a robot to an obstacle

The move_base node is configured according to the *costmap_ 2d* package [21]. The costmap_2d node provides a method of propagating cost values outwards from each occupied cell that decreases with distance. This means that the further away an obstacle is, the lower the cell cost and vice versa. This is visualized in figure 6. This process is divided into five cell weight categories:

- Lethal
- Inscribed
- Possibly circumscribed

- Freespace
- Unknown

The categories are determined by the robot's footprint, its inscribed and circumscribed region and the center cell, which corresponds to a cell in the costmap. The cost function is a continuously exponentially decaying function or a discretized cost decay function. The lethal cost is the case when the robot is definitely in collision and must be avoided at all times. This is why it is given the highest cost value. Note that the cost values in the figure are arbitrary and may vary. The inscribed cost occurs when the center of a cell is within the radius of the inscribed circle, resulting in a collision. Possibly circumscribed is when the center cell is inside the circumscribed region, but outside the inscribed region. Whether the robot is in collision or not, in this case, depends on the robot's orientation. Freespace cost is assumed to be zero, meaning that the robot is safe from any collision in this space. Finally, the unknown cost is defined as the distance between the circumscribed region and the freespace. This is a buffer zone created by costmap_2d that allows the user to define a custom minimal distance that a robot is allowed to an obstacle. An example of how a 2D costmap looks like is shown in figure 7. The red area are cells that represent obstacles in the costmap, whereas the blue area is when the center of an obstacle cell is within the inscribed radius of the robot. The red polygon represents the footprint of the robot. From this costmap configuration, the blue cells must never intersect the inscribed region to avoid collisions. The gray area represents safe space for navigation.

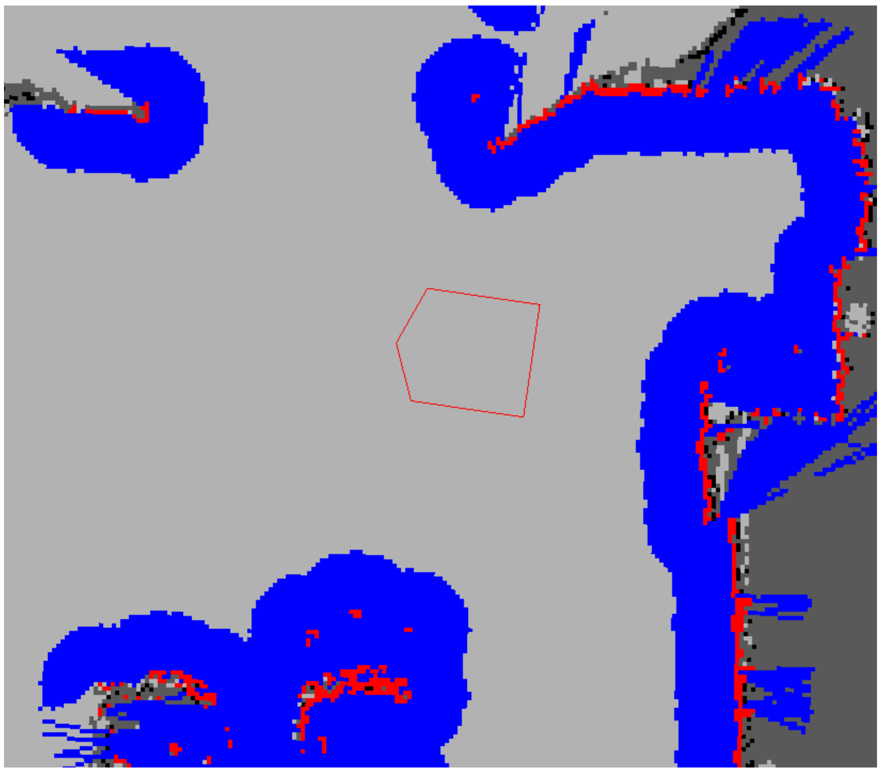


Figure 7: An example of a costmap

The costmap_2d package contains a set of user-specified parameters that define the local- and global costmap and the base controller. They are configured in 4 different .yaml files:

- costmap_common_params.yaml

- global_costmap_params.yaml
- costmap_common_params.yaml
- base_local_planner_params.yaml

The costmap_common_params are parameters that are used by both the local- and the global costmap. For the Cyborg, the file is defined by the following lines:

```
1  obstacle_range: 2.5
2  raytrace_range: 3.0
3  #footprint: [[x0, y0], [x1, y1], ... [xn, yn]]
4  robot_radius: 0.4
5  inflation_radius: 0.55
6  observation_sources: laser_scan_sensor point_cloud_sensor
7
8  # This configuration must be used when simulating the robot
9  laser_scan_sensor: {sensor_frame: frame_name, data_type: LaserScan,
10 topic: /RosAria/sim_S3Series_1_laserscan, marking: true, clearing: true}
11 point_cloud_sensor: {sensor_frame: frame_name, data_type: PointCloud,
12 topic: /RosAria/sim_S3Series_1_pointcloud, marking: true, clearing: true}
13 # This configuration must be used when using the real robot:
14 laser_scan_sensor: {sensor_frame: frame_name, data_type: LaserScan,
15 topic: /RosAria/S3Series_1_laserscan, marking: true, clearing: true}
16 point_cloud_sensor: {sensor_frame: frame_name, data_type: PointCloud,
17 topic: /RosAria/S3Series_1_pointcloud, marking: true, clearing: true}
```

Obstacle, on line 1, defines the maximum distance an object can be from the sensor and be added to the costmap. Setting this to 2.5 means that any objects within 2.5 meters of the sensor will be added to the costmap. Line 2 defines the *raytrace_range* to be 3.0 meters. This is the distance a robot requires to clear up from a given sensor reading. This is known as the freespace. Footprint, on line 3, defines the center of the Cyborgs' footprint. Line 4 defines the robot_radius for a circular robot. The *inflation_radius* defines the minimum distance for which all cells are given equal cost. This means that any obstacles with a distance of 0.55 meters or more will be given equal weights, whereas any cells less than this distance will be given weights according to figure 6. Line 6 defines the *observation_sources* which represents the laser scanners in this case. These sources are the *laser_scan_sensor* and *point_cloud_sensor* provided by the RosAria node. The sources are defined on line 9-12 for the simulated Cyborg using MobileSim and line 14-17 is for the real Cyborg.

The **global_costmap_params.yaml** is configured in the following way:

```
global_costmap:
   global_frame: /map
   robot_base_frame: base_link
   update_frequency: 1.0
   static_map: true
```

Where *global_frame* and *robot_ base_frame* are defined as the map- and base link frame. The *update_frequency* defines the frequency, in Hz, at which the costmap will run its update loop. Finally, the static_map is used to define how the costmap node will initialize itself. If it is set as true, it will initialize itself based on the map served by the *map_ server*. The parameters for the local costmap are defined similarly:

```
1  local_costmap:
2     global_frame: odom
3     robot_base_frame: base_link
4     update_frequency: 1.0
5     publish_frequency: 2.0
6     static_map: false
7     rolling_window: true
8     width: 6.0
9     height: 6.0
10    resolution: 0.05
```

The *global_frame* here is odom and the *robot_ base_frame* is base_link. The *update_frequency* and *static_ map* are the same as the previous file. The *publish_frequency* on line 5 defines the rate at which the costmap will publish visualization information. *rolling_ window* determines whether the costmap will be centered around the cyborg or not as it moves. The *width* and *height* defines, in meters, the size of the costmap, whereas *resolution* defines the resolution of the map in meters per cell. The last configuration file needed to set up the move_base node is the base_local_planner_params.yaml, which is defined as the following:

```
1  TrajectoryPlannerROS:
2     max_vel_x: 1.8
3     min_vel_x: 0.1
4     max_vel_theta: 3.1
5     min_in_place_vel_theta: 3.1
6     acc_lim_theta: 2.6
7     acc_lim_x: 0.5
8     acc_lim_y: 0.5
9     holonomic_robot: false
```

It defines the velocity and acceleration limits of the robot on line 2-5 and line 6-8, respectively. The last line defines whether the robot is holonomic or not.

The move_base stack uses actionlib (see section 2.8) to send or cancel goal requests to the robot. To request a navigation goal request, it has to be of the message type *geometry_ msgs/PoseStamped*.

3 Hardware and Software

3.1 Hardware

This section describes the relevant hardware used throughout this report.

3.1.1 ZED stereo camera

The ZED stereo camera, shown in figure 8, is a 3D Camera by Stereolabs for depth sensing, motion tracking and real-time 3D mapping [22]. It provides a path range from 0.3 meters up to 25 meters and has a maximum output resolution of 4416x1242 (2.2k). In addition, it has multiple third-party integrations such as Tensorflow, Matlab, OpenCV, ROS and more.

Figure 8: ZED stereo camera

3.1.2 Desktop computer

The desktop computer used in this report has the following hardware:

- NVIDIA GeForce GTX 1060 6GB
- Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz 3.40 GHz
- Motherboard: B75MA-P45
- 8 GB RAM
- ENERMAX MAXPRO II 600W PSU

3.1.3 Jetson Nano Developer Kit

The Jetson Nano by Nvidia, seen in figure 9 is a small and powerful computer capable of running multiple neural networks in parallel [23]. The following list contains the relevant hardware [24]:

- GPU: 128-core NVIDIA Maxwell™ architecture-based GPU
- CPU: Quad-core ARM® A57
- Memory: 4 GB 64-bit LPDDR4; 25.6 gigabytes/second

Figure 9: Jetson Nano developer kit

3.1.4 The NTNU Cyborg

The NTNU Cyborg consists of a Pioneer LX by Adept MobileRobots (went out of business in 2018) and student contributions (see section 1.1) The following list shows the relevant hardware:

- Intel D525 64-bit dual core CPU @1.8 GHz
- Intel GMA 3150 integrated graphics processing unit
- Intel 6235ANHMW wireless network adapter
- Ports for ethernet, RS-232, USB, VGA, and various other analog and digital I/O
- SICK 300 and SICK TiM 510 laser scanner
- A 60 A h battery, expected to power the robot continuously for up to 13 hours

3.2 Software

This section describes the relevant software used throughout this report.

3.2.1 RosAria

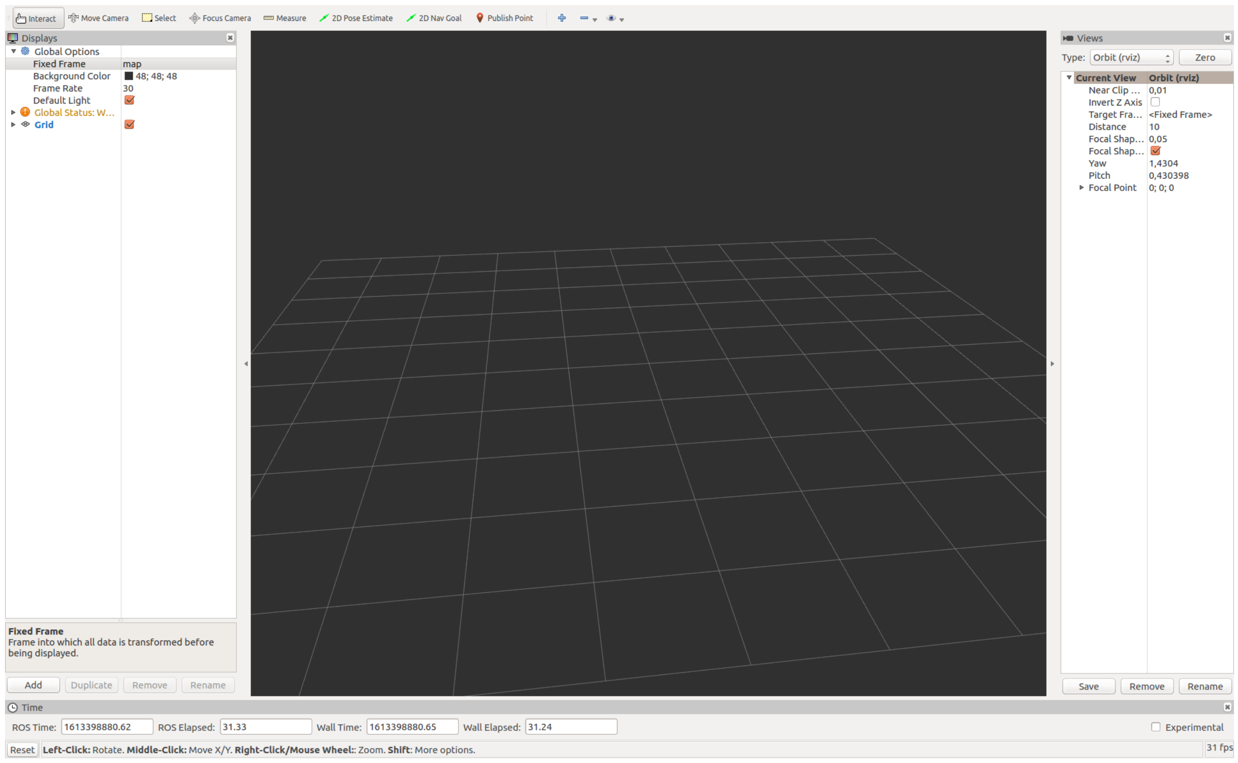
RosAria is a package that provides a ROS interface for most Adept MobileRobots, such as the Pioneer LX. The RosAria node publishes data from the robot's embedded controller by ARIA and can be used to send velocity commands to the robot via the RosAria/cmd_vel topic. It also publishes important topics such as the robot's pose, which contains odometry information of the robot, and the motor state defining whether the motor is on or off. The information from the laser scanner can also be obtained through the RosAria node.

3.2.2 RViz

RViz is a popular graphical tool in ROS and is used to display various information such as the robot model, path, odometry, maps and more. RViz can be launched using the following command in terminal, given that roscore is launched:

```
$ rosrun rviz rviz
```

This will open the RViz window. A new display can be added by pressing the *add* button on the lower-left corner of figure 10.



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
Global Status: W…	
Grid	✓

Fixed Frame
Frame into which all data is transformed before being displayed.

Type: Orbit (rviz)

Current View	Orbit (rviz)
Near Clip …	0,01
Invert Z Axis	☐
Target Fra…	<Fixed Frame>
Distance	10
Focal Shap…	0,05
Focal Shap…	✓
Yaw	1,4304
Pitch	0,430398
Focal Point	0; 0; 0

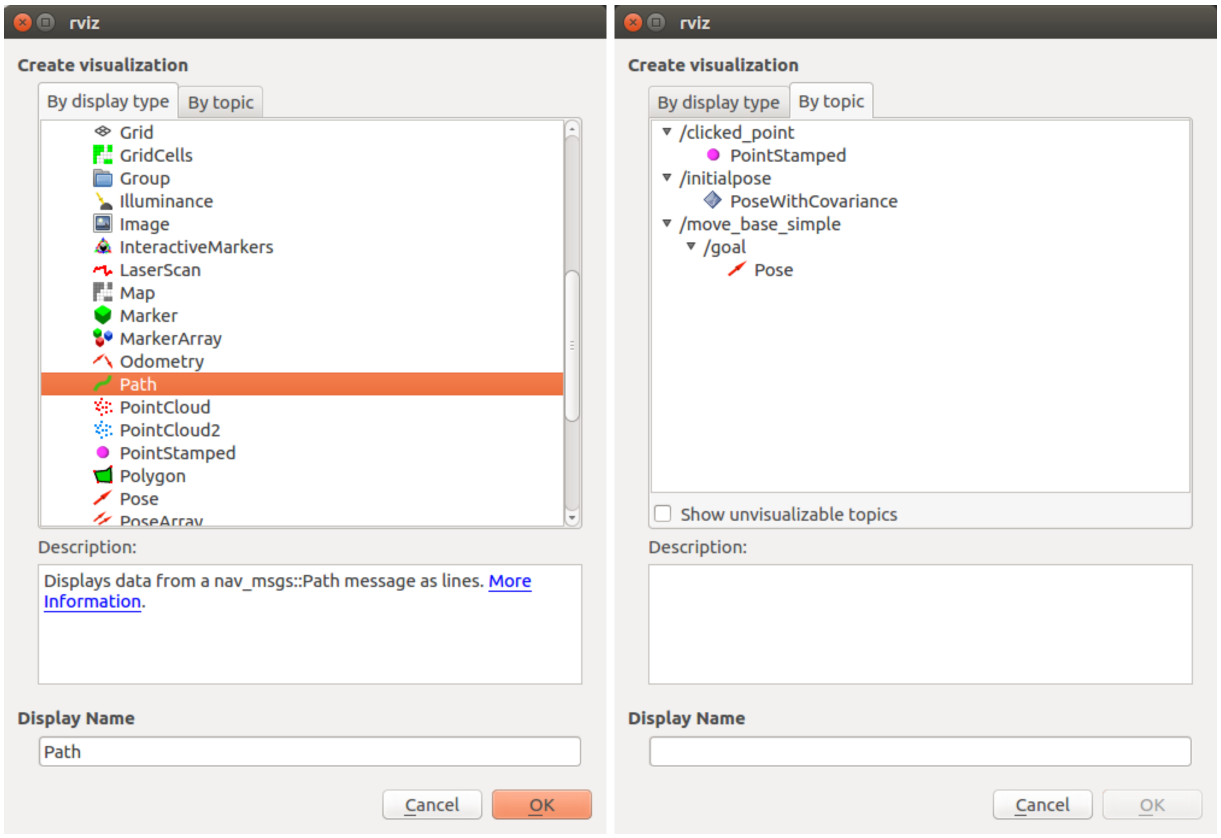
ROS Time: 1613398880.62 ROS Elapsed: 31.33 Wall Time: 1613398880.65 Wall Elapsed: 31.24

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options.

31 fps

Figure 10: Default view of RViz

This opens a new window where a new display can be added based on the display type (see figure 11a), or by a given active topic (see figure 11b). Most displays in RViz can visualize the given information as long as there is an active topic publishing the information needed. For example, the path of the robot can be displayed in RViz *only* if a topic is publishing to a topic containing the message nav_msgs/Path, as specified from the section called "Description" in figure 11



Displays data from a nav_msgs::Path message as lines. More Information.

(a) Adding visuals to RViz based on display type. (b) Adding visuals to RViz based on the active topic.

Figure 11: Adding a new display in RViz based on display type and by active topics.

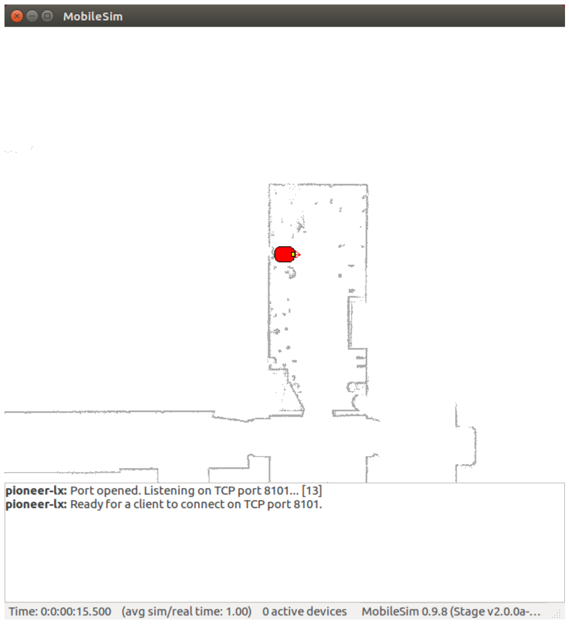
In the case of displaying the robot model, a URDF file of the robot must be defined. Additionally, the parameter called *robot_ description* must be set to contain the file path of the URDF file. Note that in this case, RViz is actively looking for a ROS parameter and not a topic. The parameter can be set using a launch file. The following lines show how such a launch file can be constructed:

```
1 <launch>
2 <!-- Structure:
3 <param name = "robot_description" textfile = "$(find name_of_ros_package)/
      path_from_package_to_urdf/robot_name.urdf" /> -->
4
5 <!-- Practical example: -->
6 <param name = "robot_description" textfile = "$(find robot_state_publisher)/amr-ros-config/
      cyborg_description/urdf/cyborg_fine.urdf" />
7 </launch>
```

The parameter name robot_description is defined on line 6 with the location of a text file from the robot_state_publisher package. This enables RViz to detect this file as the display type *RobotModel*, which can be added to the world. To view the model, a transform between the links that have been imported to RViz and the frame of reference must be active. This is done by running the *tf* package, described in section 2.9. The frame of reference, or the *Fixed frame*, is located under the *Global Options* display. By default, the frame of reference will be an empty map but can be changed as long as there exist other coordinate frames. Examples of different frames are the odometry frame and the different link frames, such as the frame of the base link. It is important to set the fixed frame as something that is considered stationary, such that everything moves relative to it. In the case of using SLAM with RViz, setting the map as the fixed frame will move the robot relative to the map that is being created.

3.2.3 MobileSim

MobileSim is a software that enables communication with a simulated robot by MobileRobots. It is a useful tool for testing code, as it uses the RosAria package, which is also used on the real Cyborg. This means that the same RosAria node will connect to the fake robot, subscribe to the cmd_vel topic, and publish fake laser scans and odometry information. Figure 12 shows what the simulator looks like with the map of Glassgården.



```
pioneer-lx: Port opened. Listening on TCP port 8101... [13]
pioneer-lx: Ready for a client to connect on TCP port 8101.
```

```
Time: 0:0:00:15.500   (avg sim/real time: 1.00)   0 active devices   MobileSim 0.9.8 (Stage v2.0.0a-...
```

Figure 12: Mobilesim - Displaying the Pioneer LX with a map of Glassgården.

3.2.4 zed_ros_wrapper - ROS wrapper for the ZED camera

The ZED ROS wrapper is a ROS package compatible for all ZED cameras and provides access to all camera sensors and parameters. Its primary node, *zed*, provides a variety of topics from its left and right lens, such as the image, and its sensors, such as IMU data. The package also provides a launch file that opens the camera with ROS and publishes all the necessary information to ROS topics.

3.2.5 zed_ros_examples - Collection of ROS examples with the ZED camera

The zed_ros_examples is a collection of examples demonstrating the various capabilities of the ZED camera, such as ZED with rtabmap (see section 4). This package includes the necessary launch files for running the different examples and can be modified to fit the requirements for a given project. This package uses the ROS wrapper and must therefore be installed first.

3.2.6 ZED SDK

The ZED SDK is software made by Stereolabs and is architectured around the *camera* class [25]. The Camera class is the main interface with the camera and can be opened using the class method `open()` and passing the function `InitParameters()` as the arguments. The `InitParameters()` function contains important initial parameters such as the camera resolution, frames per second, depth mode, input type and more. The input type is useful when another node has already opened the camera. Furthermore, by specifying the *set_from_stream* parameter from `InputType()`, with the IP address of the broadcasting computer and the port on which to listen to, allows for remote streaming.

Another important class method is `grab()`, which is used to grab the latest images. A set of runtime parameters are passed as arguments that defines the sensing mode, enabling depth and more.

4 RTAB-Map - An RGB-D SLAM approach

Real-Time Appearance-Based Mapping (RTAB-Map) is a SLAM method based on RGB-D, Stereo and Lidar graphs using incremental appearance-based loop closure detector [26]. Loop closure is the process of determining whether an agent, e.g., a robot, has returned to a previously visited location [27]. Loop closure detection is useful for correcting drift in odometry and correcting the map. It suffers, however, from significant dynamic changes to the environment, such as people or objects being present in areas previously empty. This should be considered when planning to map the environment to ensure robustness towards false positives. The map is created through map generation and exploration, i.e., by driving around and exploring new areas. RTAB-Map is based on occupancy grid. Occupancy grid mapping addresses the problem of generating maps from noisy and uncertain measurement data, assuming that the pose of the robot is known [28]. This method estimates the posterior probability:

$$p(m|z_{1:t}, x_{1:t}) \tag{1}$$

Equation 1 calculates the probability of map m, given the set of measurements $z_{1:t}$ from 1 to t, and state $x_{1:t}$ from time 1 to t. The state x represents the set of robot poses from 1 to t [29]. The map m can be split into a set of grid cells with index i, denoted m_i. Here, $p(m_i)$ represents the probability of a cell being occupied or not, e.g. if there is an obstacle there of not. The time complexity for equation 1 is therefor $\mathcal{O}(2^{|m|})$, where m is the number of grids, which can be 10 000 or much more. To reduce the time complexity, one can instead look at each cell m_i individually:

$$p(m|z_{1:t}, x_{1:t}) = \prod_i p(m_i|z_{1:t}, x_{1:t}) \tag{2}$$

Equation 2 now represents the posterior probability of the map as a product of each individual cell. Since each cell has the probability of being occupied or not, the time complexity for this case is the number of cells multiplied with the different states it can be in (i.e. occupied or not), or $\mathcal{O}(2|m|)$. The downside of this approach is that any dependency of neighboring cells cannot be modeled, as this approach assumes each cell are independent of each other.

4.1 rtabmap in ROS

RTAB-Map is available as two ROS packages: *rtabmap* [30] and *rtabmap_ros* [31]. The latter is a ROS wrapper of RTAB-Map. This package provides several useful nodes:

- rtabmap: The main node where the graph of the map is incrementally built and optimized when a loop closure is detected.
- rtabmapviz: This enables a visual interface of RTAB-Map, similar to RViz
- Visual and LiDAR Odometry: Uses visual and LiDAR odometry
- rgbd_odometry: Uses RGBD images to compute odometry. This approach uses visual features from the RBG images along with the info from depth images.
- stereo_odometry: Uses stereo images to compute odometry. The visual features are extracted from the left images, along with depth information from the depth images, and attempts to find similar features on the right images.

The rtabmap node can be configured in two ways:

1. Mapping mode
2. Localization mode

The mapping mode will continuously create a map through exploration of the environment. By default, any previously generated map will be deleted when the rtabmap is started. To avoid this, the argument "–delete_db_on_start" must be set to false. The second mode is the localization mode. This will not create any new map, nor will it expand on any existing map through exploration. This is useful when a map has already been made, and any further improvement is not necessary. However, this mode is limited to the size of the existing map, meaning that any localization will fail for the unexplored area.

5 Types of object detection algorithms

Object detection in terms of computer vision is identifying and locating one or several objects within an image or video. This section describes some popular methods of object detection.

5.1 R-CNN, Fast R-CNN and Faster R-CNN

Region Based Convolutional Neural Networks (R-CNN) is a method of detecting objects in images by using a method called *selective search* [32]. Selective search extracts 2000 regions from a given image which is defined as region proposals. These regions are used for classification and are fed into a Convolutional Neural Networks (CNN). CNN is an effective method of detecting patterns in images such as edges, shapes, colors etc., using different filters [33]. The CNN will take the 2000 region proposals as input, extract features and feed it into an Support Vector Machine (SVM). The goal of the SVM is to find a plane, called the hyperplane, capable of separating the different data points into classes based on the different features [34]. In other words, the hyperplane will be a plane in an N-dimensional space, where N denotes the number of features. If N $= 2$, the plane will be a line in 2D space. This method will create several hyperplanes, and the optimal hyperplane will be placed such that it has the maximum distance between the data points of all classes.

There are some disadvantages with this method of object detection:

- Using 2000 region proposals per image takes a long time in terms of training the network.
- Since it is rather slow at detecting objects, the method is *not* suitable for real time video.
- The selective search algorithm does not get "better" at detecting regions over time, meaning that the proposed regions might not be sufficiently good for every image.

In later years, improved versions of this algorithm have been implemented called *Fast R-CNN* and *Faster R-CNN*. These methods aim at fixing some of the disadvantages of its predecessor. Figure 13 shows a time comparison between the different R-CNN methods.

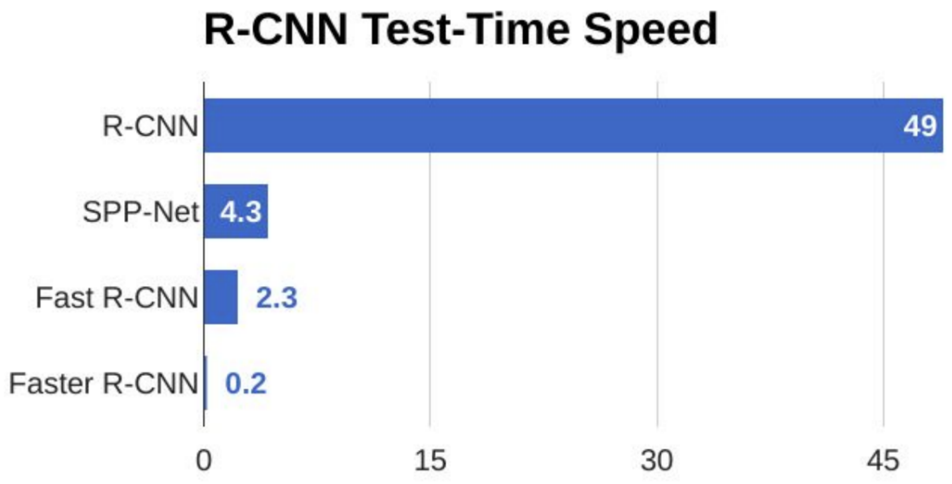


Figure 13: Speed comparison between R-CNN, Fast R-CNN and Faster R-CNN

5.2 YOLO -You Only Look Once

You Only Look Once (YOLO) is a method of detecting and locating objects using a single neural network on the entire image, in contrast to classification networks such as fast RCNN, which perform detection on the given region proposals. Consequently, YOLO does not have to perform prediction multiple times for various regions in an image. This also means that it requires only a single forward propagation through a neural network to detect the different objects [35]. The prediction that is being made is based on predicting a class of an object in the picture and the corresponding bounding box for specifying the object's location [36]. Images are typically split into a 19x19 grid cells, where each cell is predicting five bounding boxes. This approach detects more bounding boxes than there are actual objects in the image. Therefore, only the bounding boxes with high object probability are chosen. Figure 14 shows how objects are detected from an SxS grid as input.

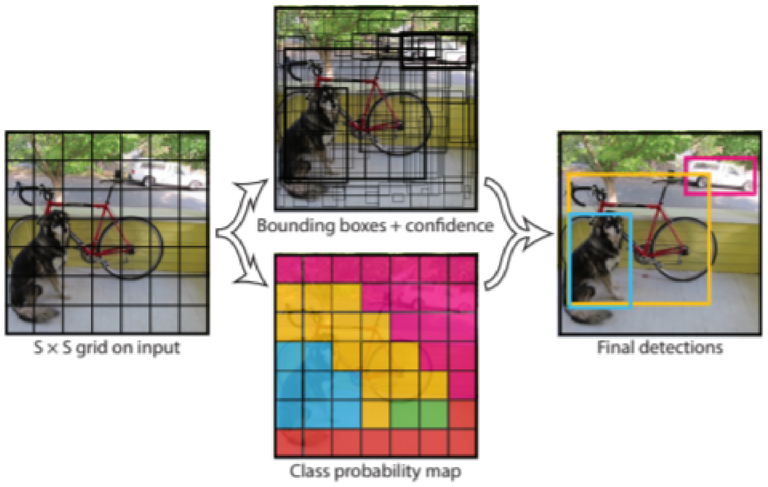


Figure 14: Process of detecting objects from an SxS grid image

A popular method of implementing YOLO is with Darkent, which is an open-source neural network framework. This is written in C and CUDA technology. This is a fast approach when using a Graphics Processing Unit (GPU) provided by Nvidia. Furthermore, YOLO is orders of magnitude faster than other object detection algorithms, which makes it suitable for live video.

6 Creating a Cyborg robot model for display in RViz

This section describes the process of using SolidWorks to export a 3D computer-aided design (CAD) model into a Unified Robot Description Format (URDF) file and combine it with an old robot model to display the current version of the Cyborg. This is done in two parts: One model shall display the robot without the ZED camera and one with the ZED camera.

6.1 Creating the URDF file for the Cyborg

The robot model that has been used in RViz over the years of ROS development on the Cyborg has been the pioneer-lx.urdf provided by Mobilesim (see figure 17a). This model did not accurately represent the current version of the Cyborg, which can be seen in figure 1d. The following parts were missing in the old model and had to be added:

- Front panel
- Back panel
- LED Dome

A complete 3D CAD model of the parts mentioned above was available in SolidWorks. A Solidworks plug-in for exporting 3D CAD models to URDF files was used. An Experts in Teamwork (EiT) project in the spring of 2018 had already assembled the current version into a complete assembly (see appendix B.1). Since only the three mentioned parts above were needed, any other parts were suppressed from the assembly.

Before the parts could be exported as a URDF file, two coordinate frames were created: *Body frame* and *Dome frame*. The purpose of these frames was to create frames of reference. These frames can be seen from figure 15.

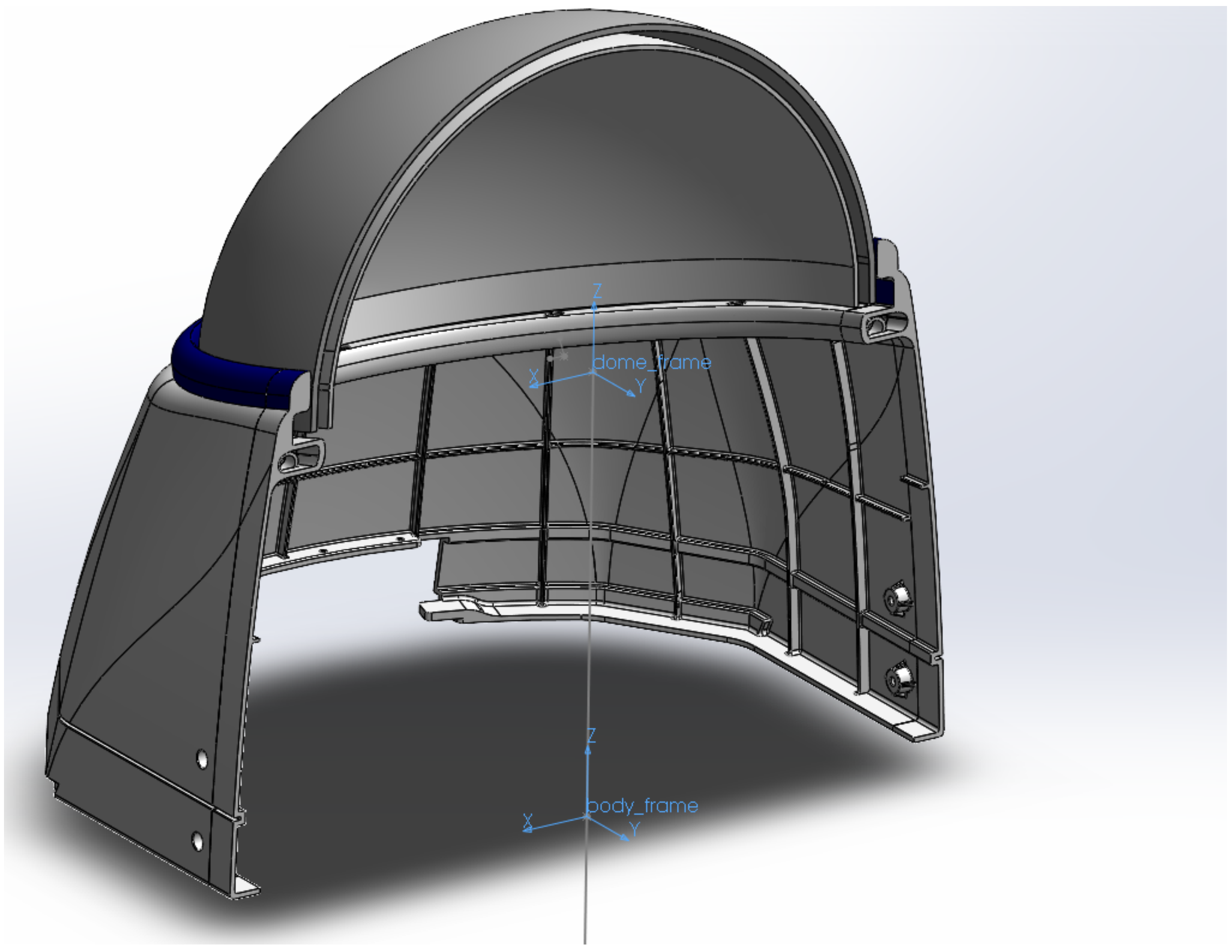
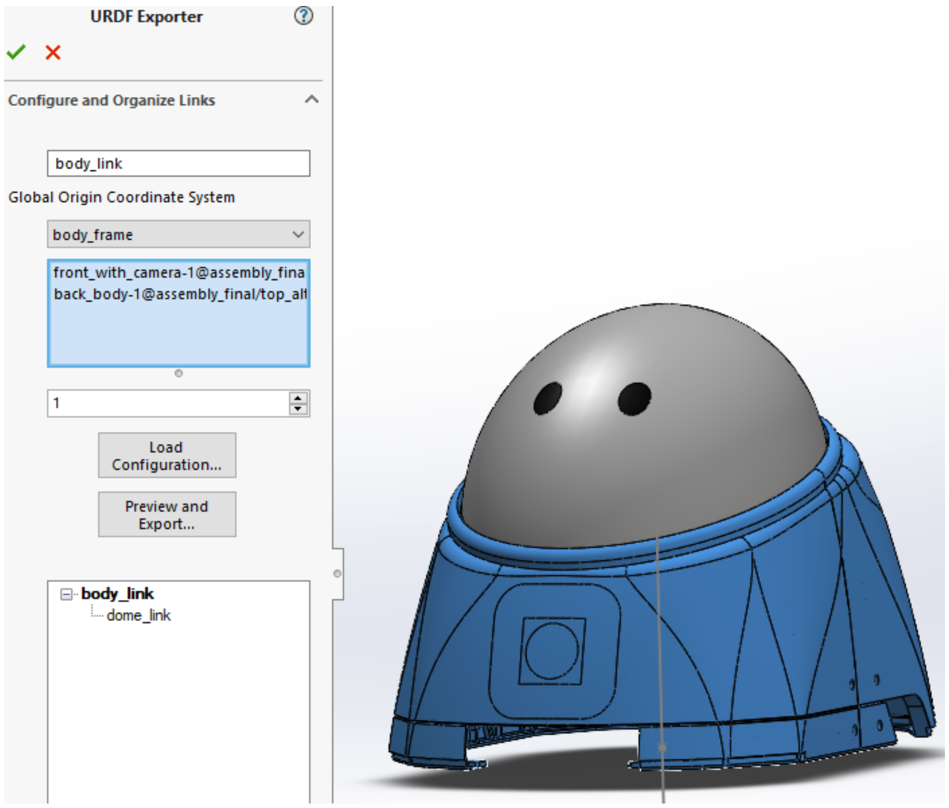


Figure 15: Reference frames for the body- and dome link

After the two frames were created, the *Export as URDF* tool was selected. The robot was configured with two links: The *body_link* and *dome_link*. Figure 16a shows the front- and back panel being selected as the body_link. Moreover, the global origin coordinate system was defined as the Body frame. The body_link was given one child link; the dome_link, defined in figure 16b. The child link consisted of the dome marked in blue or the head assembly as it was defined in SolidWorks. Additionally, a joint connecting the parent and child link was defined as the *dome_joint*. The reference coordinate system was the Dome frame. Since there were no more links to define, the dome_link had zero child links.

The resulting configuration had one joint and two links: Dome_joint and body_link and dome_link, respectively. The position and orientation of the origin of the joint were defined under the *configure joint properties* menu. The origin was set to $x = y = 0$ and $z = 0.26541$, which corresponded to the distance between the body_frame and dome_frame. The orientation of the joint was set to zero. The necessary link properties were defined in the *Configure Link Properties*. The position and orientation of the visual- and collision meshes were set to zero for both links. The inertial properties were unchanged. Finally, the URDF was exported with the necessary meshes from SolidWorks.



URDF Exporter

Configure and Organize Links

body_link

Global Origin Coordinate System

body_frame

front_with_camera-1@assembly_fina
back_body-1@assembly_final/top_alt

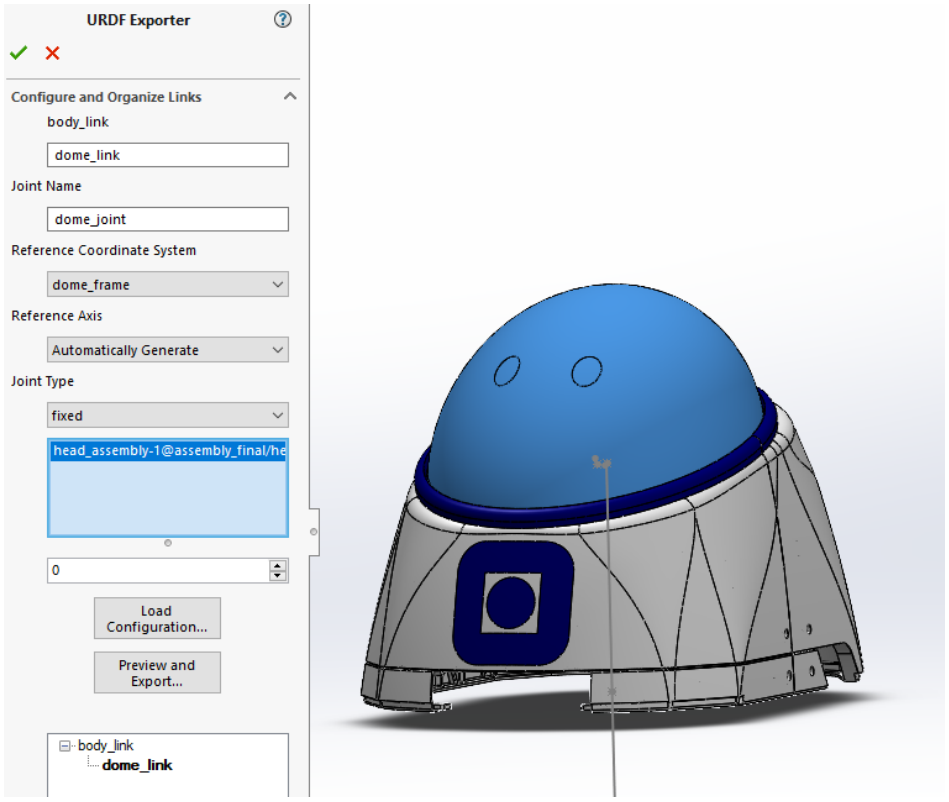
1

Load Configuration...

Preview and Export...

body_link
dome_link

(a) Defining the body link



URDF Exporter

Configure and Organize Links

body_link

dome_link

Joint Name

dome_joint

Reference Coordinate System

dome_frame

Reference Axis

Automatically Generate

Joint Type

fixed

head_assembly-1@assembly_final/he

0

Load Configuration...

Preview and Export...

body_link
dome_link

(b) defining the dome link

Figure 16: SolidWorks URDF Exporter

Since there already was a robot model for the Pioneer LX by Adept MobileRobots, the next step was to combine the old and new files into one URDF file. To achieve this, a custom joint had to be created. This joint was responsible for connecting the newly created body_link with the base link of the old model.

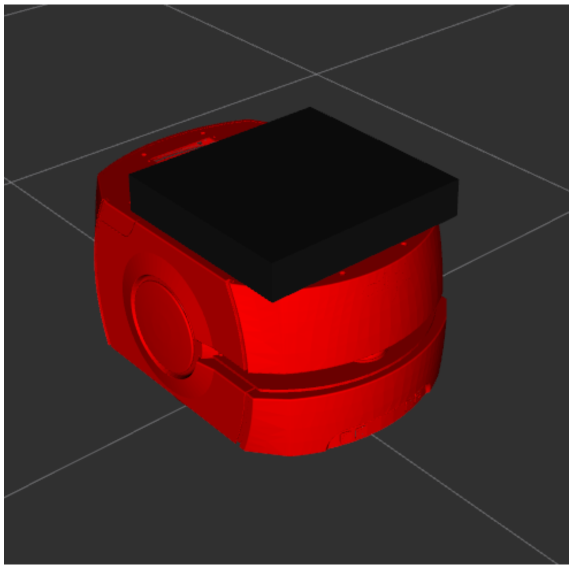
```xml
<joint
  name="body_joint"
  type="fixed">
  <origin
    xyz="0 0 0.0.372"
    rpy="0 0 0" />
  <parent
    link="base_link" />
  <child
    link="body_link" />
  <axis
    xyz="0 0 0" />
</joint>
```

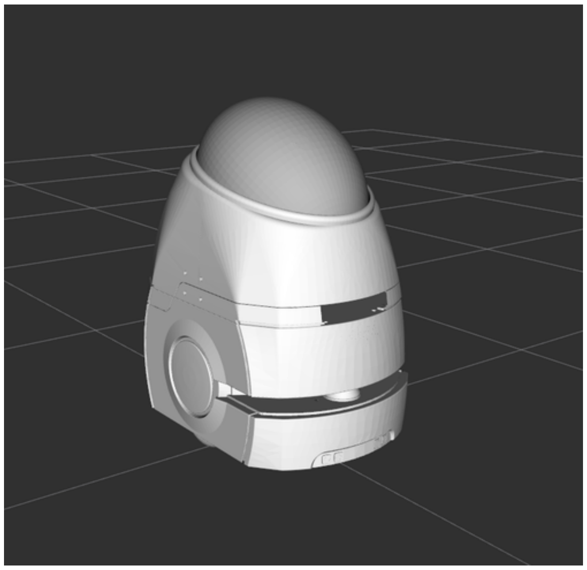
The joint type was defined as *fixed*, meaning that the links connected to this joint do not move relative to the joint. The origin of the joint was located 0.372 m above the floor. This corresponded to the height of the Pioneer LX (see Appendix A.1). Additionally, the parent and child link were set as base_link and body_link respectively. Next, the remaining code from the new URDF file was directly pasted into the old URDF.

Before the robot model was complete, changes to the body and dome joint orientation relative to the base_link had to be adjusted. A rotation of $\frac{\pi}{2}$ radians for both links aligned them with the rest of the robot. The same was done for the collision properties of the two links. Additionally, the robot's deck from the original pioneer-lx.urdf was removed, as the Cyborg had no such deck. The last finishing touch was to give the robot its white color. This was done by defining the red, green, blue and alpha (rgba) properties, where alpha defined the transparency of the links. These values were set to 1.

6.1.1 Evaluation of the URDF file

A comparison between the old and the new robot model can be seen from figure 17a and figure 17b, respectively. The deck on top of the old model was removed, and the two links: body_link and dome_link was added to the model connected by a custom joint called dome_joint. Moreover, the robot was given a white color to match the real Cyborg.


(a) Old model of the Pioneer-lx in RViz


(b) Complete model of the Cyborg in RViz

The joint- and link configuration can be seen from figure 18 and 19. The base_link had three children: r_wheel, l_wheel and body_link. Additionally, the dome_link was defined as the child link to the body_link. The resulting joints had the same orientation as the existing ones.

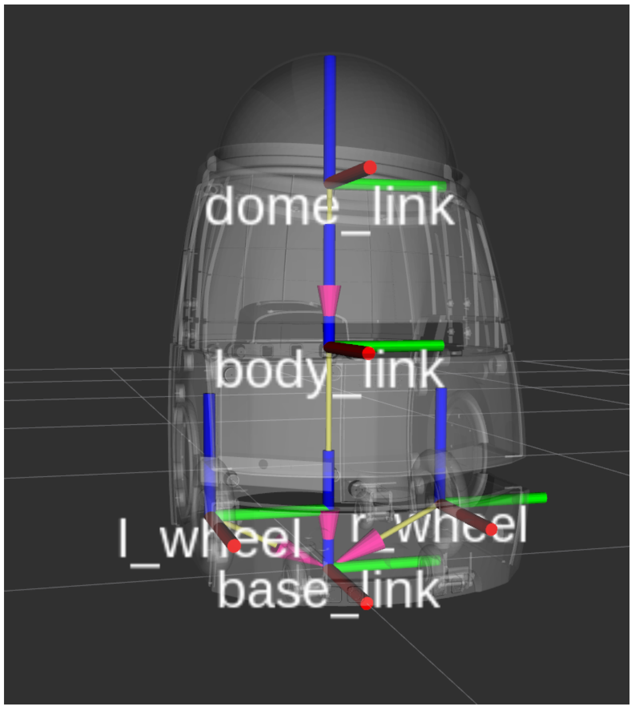


Figure 18: Joint configuration for the Cyborg

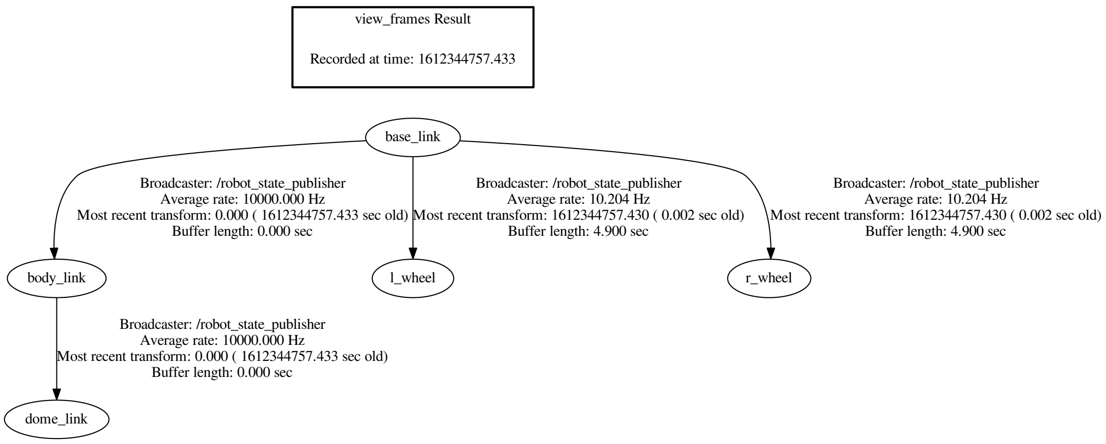

view_frames Result
Recorded at time: 1612344757.433
base_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1612344757.433 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10.204 Hz
Most recent transform: 1612344757.430 (0.002 sec old)
Buffer length: 4.900 sec
Broadcaster: /robot_state_publisher
Average rate: 10.204 Hz
Most recent transform: 1612344757.430 (0.002 sec old)
Buffer length: 4.900 sec
body_link
l_wheel
r_wheel
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1612344757.433 sec old)
Buffer length: 0.000 sec
dome_link

Figure 19: tf frames for the Cyborg model

6.2 Adding the ZED camera to the robot model

The ZED camera was also added and placed in front of the robot. The zed_wrapper package already had a URDF file for the ZED camera and was combined with the new robot model. Figure 20 shows the links of the ZED camera. The "main" link was the zed_camera_center. This was also the only link with any visual properties, i.e. the visual model represented in RViz was completely defined from zed_camera_center.

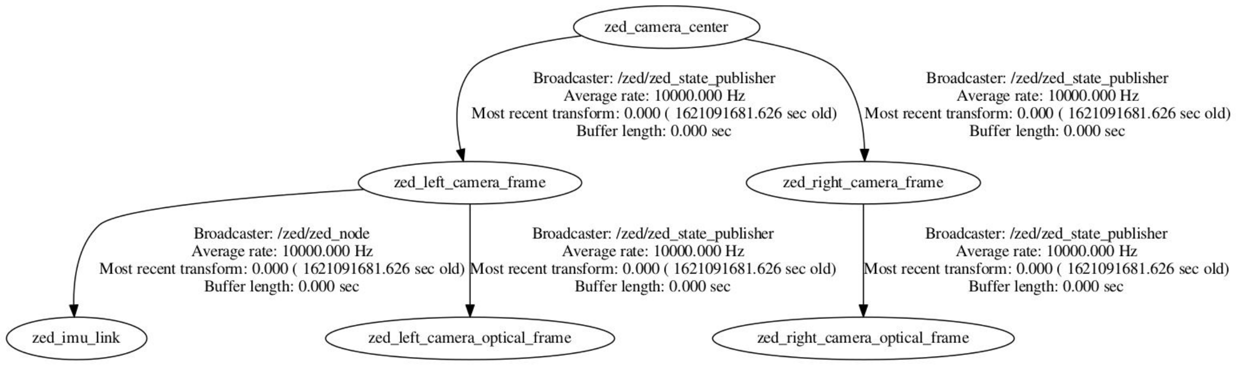

zed_camera_center
Broadcaster: /zed/zed_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1621091681.626 sec old)
Buffer length: 0.000 sec
Broadcaster: /zed/zed_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1621091681.626 sec old)
Buffer length: 0.000 sec
zed_left_camera_frame
zed_right_camera_frame
Broadcaster: /zed/zed_node
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1621091681.626 sec old)
Buffer length: 0.000 sec
Broadcaster: /zed/zed_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1621091681.626 sec old)
Buffer length: 0.000 sec
Broadcaster: /zed/zed_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1621091681.626 sec old)
Buffer length: 0.000 sec
zed_imu_link
zed_left_camera_optical_frame
zed_right_camera_optical_frame

Figure 20: The tf tree for the ZED URDF model, displaying the different frames.

To include the camera with the Cyborg URDF, a custom joint was created which connected the base link, i.e. the Pioneer LX, with the zed_camera_center:

```xml
<joint name="zed_camera_center_joint" type="fixed">
    <parent link="base_link"/>
    <child link="zed_camera_center"/>
    <origin xyz="0.299 0.0 0.387" rpy="0.0 0.0 0.0" />
</joint>
```

The joint name was labeled zed_camera_center_joint. The parent link was set as base_link, whereas the child link was set as zed_camera_center. The origin offset was set to $x = 0.299$, $y = 0.0$ and $z = 0.387$.

The coordinates were relative to the base_link frame. The x location was approximated to be 0.299, i.e., a little less than half the length of the robot base. The height of the camera, i.e. the z-direction, was $z = h_{robot} + \frac{h_{ZED}}{2}$ where h_{robot} was the height of the robot and h_{zed} was the height of the camera (see appendix A.2). These values were 0.372 meters and 0.030 meters, respectively.

6.2.1 Evaluation

The RViz model of the ZED camera is shown in figure 21, which displays its main link: zed_camera_center.

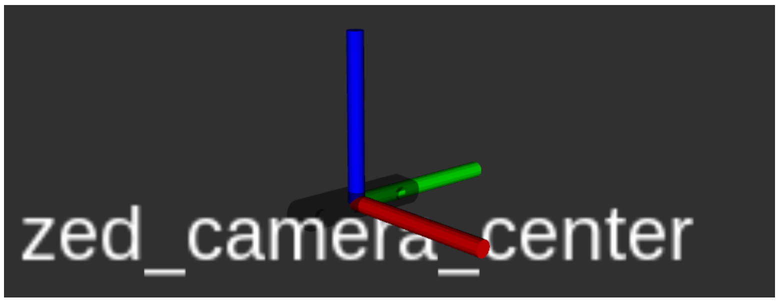
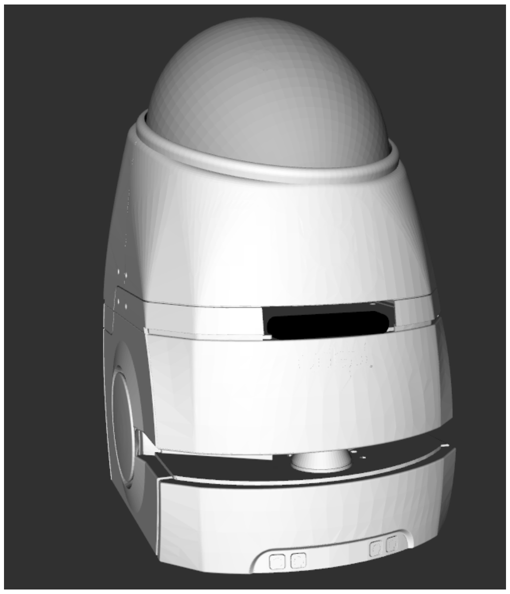
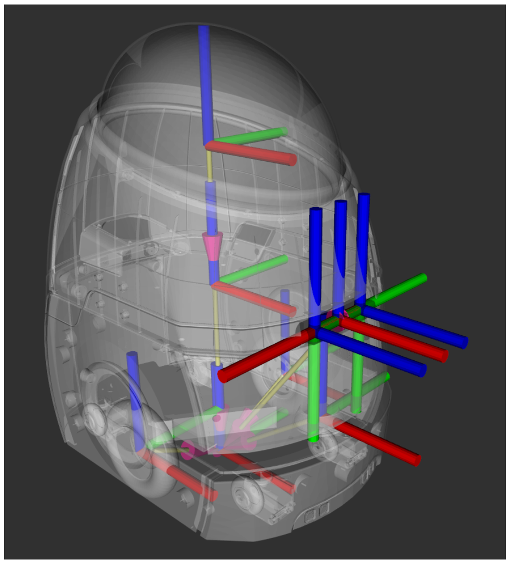


Figure 21: Location of the joint of the ZED camera in RViz

The resulting new robot model is shown in figure 22a. The camera was successfully placed on top of the base link, all the way to the front of the Cyborg. The joint configuration can be seen on figure 22b. The camera's location on the robot model was designed to match the camera's location on the real Cyborg.



(a) Cyborg robot model with ZED camera displayed in RViz.



(b) Joint location for the Cyborg model with the ZED Camera

Additionally, the tf frame tree can be seen from figure 23. The base_link was the parent link, whereas the zed_center was the child link. Moreover, the zed_center had two child links: the right- and left-camera links, which defined the location of the respective lenses. Moreover, the zed_left_camera had a child link called zed_depth_camera, the frame where all depth measurements were relative to.

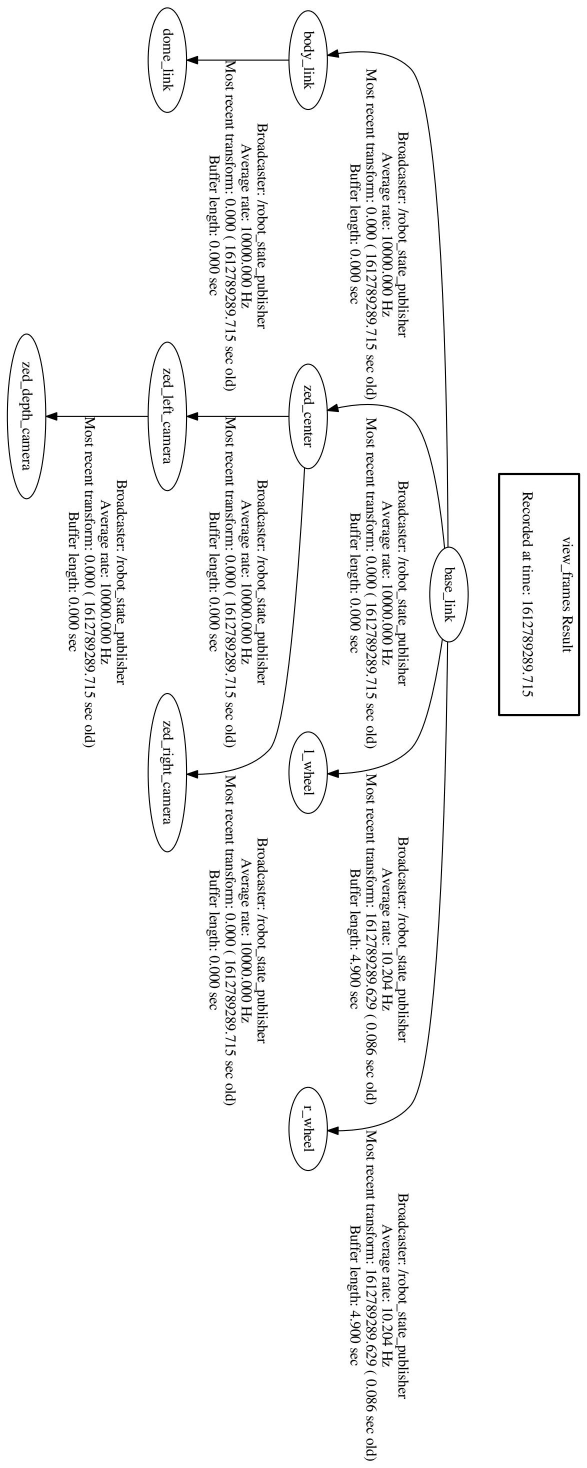

view_frames Result
Recorded at time: 1612789289.715
base_link
body_link
dome_link
zed_center
zed_left_camera
zed_depth_camera
zed_right_camera
l_wheel
r_wheel
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1612789289.715 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1612789289.715 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1612789289.715 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1612789289.715 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1612789289.715 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1612789289.715 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10.204 Hz
Most recent transform: 1612789289.629 (0.086 sec old)
Buffer length: 4.900 sec
Broadcaster: /robot_state_publisher
Average rate: 10.204 Hz
Most recent transform: 1612789289.629 (0.086 sec old)
Buffer length: 4.900 sec

Figure 23: tf frames for the Cyborg and ZED camera

7 Integrating Visual simultaneous localization and mapping (VSLAM) with the Cyborg

This section describes the process of using the ZED camera to create 2D and 3D maps using RTAB-Map in ROS.

7.1 Specification and Requirements

A framework for running VSLAM with the Cyborg shall be implemented. This includes:

- Creating a complete launch file using RTAB-Map running VSLAM with the Cyborg using the updated URDF file from section 6.2.
  - The ability to integrate laser scans provided by the RosAria node shall also be included.
- The RTAB-Map shall run on the desktop computer and listen to the ZED camera broadcast by the Jetson Nano.

7.2 Implementing VSLAM

Before any VSLAM software could be implemented, two necessary packages had to be installed: *zed_ros_wrapper* and *zed_ros_examples*. Both packages required installing the ZED Application Programming Interface (API) and its Compute Unified Device Architecture (CUDA) dependency. ZED SDK version 3.4.2 [37] was chosen, along with CUDA 11.1[38] on the desktop computer. ZED SDK for Jetpack 4.5 was installed on the Jetson Nano.

The zed_ros_wrapper contained a launch file which started the camera with ROS. This had to be modified to include the new Cyborg URDF created in section 6. The main launch file which launched the camera was zed.launch. This also launched the zed_camera.launch.xml, which included the URDF file. The following line was added to zed_camera.launch.xml:

```
<param name="$(arg camera_name)_description" textfile = "$(find zed_wrapper)/urdf/Cyborg.
    urdf" />
```

This specified ROS to set the parameter *zed_description* to contain the Cyborg URDF from the ROS package zed_wrapper for display in RViz. Changing the robot model required some changes to the parameters related VSLAM. The common.yaml, provided by zed_wrapper, set the *floor_alignment* parameter to true. This automatically calculated the camera/floor offset. Additionally, *two_d_mode* was set to true, which forced the robot to navigate along the horizontal plane. From rtabmap.yaml, provided by RTAB-Map, the parameter *frame_id* was changed from base_link to zed_camera_center. Originally, the base link of the ZED camera model had the same location as the zed_camera_center. Since the Cyborg model was used instead, the location of the new base_link was different and therefore had to be changed.

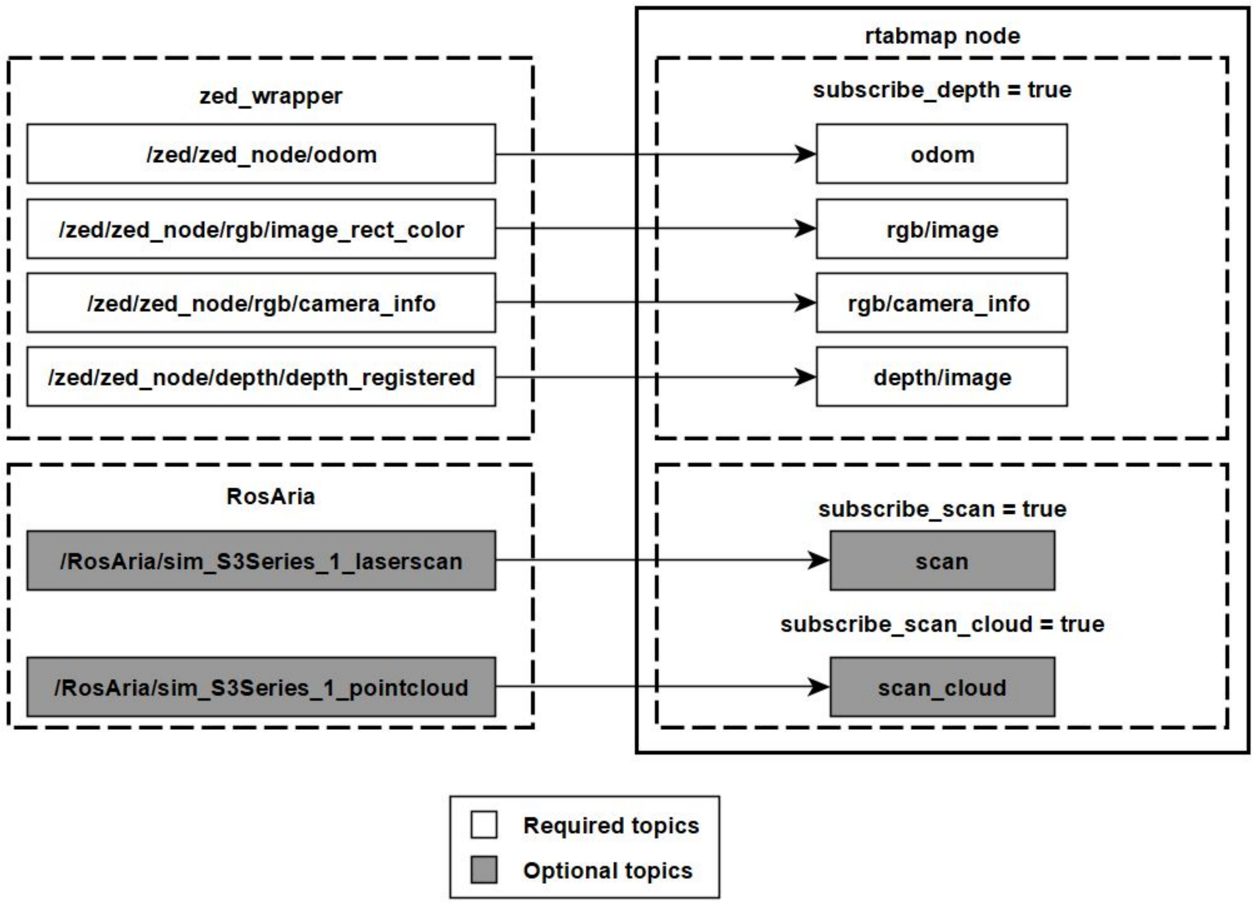

rtabmap node
zed_wrapper
subscribe_depth = true
/zed/zed_node/odom
odom
/zed/zed_node/rgb/image_rect_color
rgb/image
/zed/zed_node/rgb/camera_info
rgb/camera_info
/zed/zed_node/depth/depth_registered
depth/image
RosAria
subscribe_scan = true
/RosAria/sim_S3Series_1_laserscan
scan
subscribe_scan_cloud = true
/RosAria/sim_S3Series_1_pointcloud
scan_cloud
Required topics
Optional topics

Figure 24: Rtabmap launch configuration

The next step was to edit the launch files to work with laser scans provided by RosAria. Figure 24 shows the configuration of topics needed to run the rtabmap node using depth data and laser scans. The required topics, given that the parameter *subscribe_ depth* was true, were: odom, rgb/Image, rgb/camera_info and depth/Image. These topics had to be published by the zed_wrapper, which ran the camera. The direction of the arrows represent the required *remapping* of topics. The zed_wrapper node published to topics with a different name than the ones contained inside the rtabmap node. Remapping is a way to "trick" ROS nodes to think that they are publishing/subscribing to a topic of interest by using the remap argument. For example, remapping the zed/zed_node/odom topic provided by the zed_wrapper node to /odom causes the rtabmap node to subscribe to that information, even though the information is from a completely different topic.

The inclusion of laser scan or laser cloud readings required some modifications to sl_rtabmap.launch.xml:

```
    <!-- Laser scan topics -->
    <arg name= "laser_scan_topic"   default="/RosAria/S3Series_1_laserscan" />
    <arg name= "scan_cloud_topic"   default="/RosAria/S3Series_1_pointcloud" />
```

This stored the name of the laser topics into arguments, which was used later in zed_rtabmap.launch:

```
    <!-- RTABmapviz -->
```

```
2      <node name="rtabmapviz" pkg="rtabmap_ros" type="rtabmapviz" output="screen" args=""
       launch-prefix="">
3          <remap from="rgb/image"           to="$(arg rgb_topic)"/>
4          <remap from="depth/image"         to="$(arg depth_topic)"/>
5          <remap from="rgb/camera_info"     to="$(arg camera_info_topic)"/>
6          <remap from="odom"                to="$(arg odom_topic)"/>
7          <remap from="grid_map"            to="map" />
8          <remap from="scan"                to="$(arg laser_scan_topic)"/>
9          <remap from="scan_cloud"          to="$(arg scan_cloud_topic)"/>
10     </node>
11         <!-- RTABmap -->
12     <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen" args="$(arg args)"
        launch-prefix="">
13         <rosparam command="load" file="$(find zed_rtabmap_example)/params/rtabmap.yaml" />
14         <remap from="rgb/image"           to="$(arg rgb_topic)"/>
15         <remap from="depth/image"         to="$(arg depth_topic)"/>
16         <remap from="rgb/camera_info"     to="$(arg camera_info_topic)"/>
17         <remap from="odom"                to="$(arg odom_topic)"/>
18         <remap from="grid_map"            to="map" />
19         <remap from="scan"                to="$(arg laser_scan_topic)"/>
20         <remap from="scan_cloud"          to="$(arg scan_cloud_topic)"/>
```

This launched two different nodes: rtabmapviz and rtabmap on line 2 and 11, respectively. Line 3-7 and 14-18 were already implemented in the example launch files. Line 8-9 and 19-20 were added such that the nodes would subscribe to the laser topics.

VSLAM was tested within the Cyborg office, seen on figure 25. The camera was first launched on the Jetson Nano: `$ roslaunch zed_wrapper zed.launch`
`$ rosservice call zed/zed_wrapper/start_remote_stream` The parameter *start_ remote_ stream* was enabled, which allowed the Jetson Nano to broadcast to the desktop computer:
`$roslaunch zed_rtabmap_example zed_rtabmap.launch stream:=<ip_address>`
Which launched the nodes related to RTAB-Map and specified the IP address of the Jetson Nano.

Figure 25: The Cyborg office

7.3 Evaluation of the VSLAM

The resulting launch- and configuration files for implementing RTAB-Map with the Cyborg using laser scan measurements can be found in the attached zip file. The generated 2D map can be seen from figure 26, which displays the robot model in the 2D map generated by the rtabmap node. The map was accurate, despite it looking "noisy". The black areas represent obstacles, whereas the white areas represent free space. The map managed to capture the free space in the middle of the office, between the tables on the right and left side, and the area under the tables. The mapping process lasted approximately 1 minute by maneuvering the robot around the office.

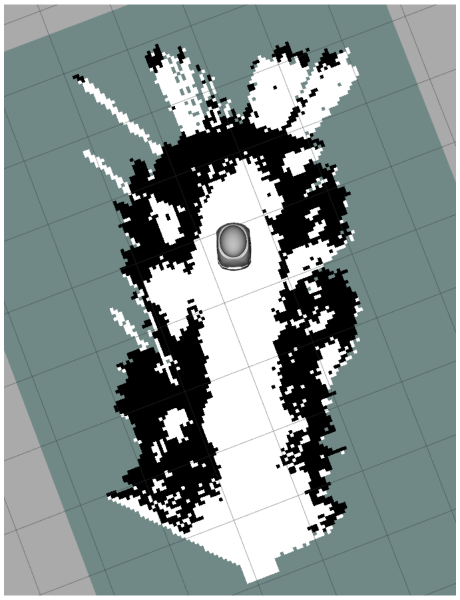
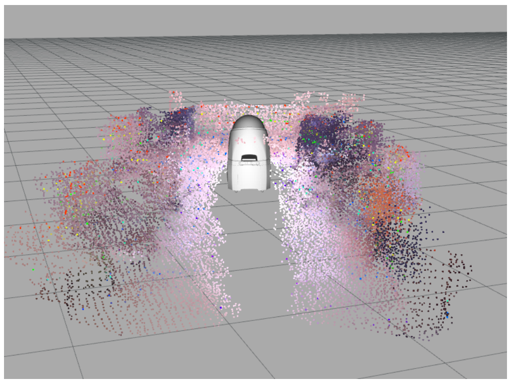


Figure 26: A 2D occupancy grid map of the Cyborg office, provided by the rtabmap node.

A 3D map was also generated and is shown on figure 27a. As comparison, figure 27b shows the correspondence between the 3D map and the actual office. It produced satisfactory results, where some of the features, such as the wooden cabinet, the wall at the end and the tables, are visible to some degree. The map could be improved by mapping for a more extended period and maneuvering more back and forth in the office to capture details it might have missed on the first run.



(a) 3D map provided by the rtabmap node in RViz.

(b) The corresponding 3D mapped part of the Cyborg office.

A comparison between the 2D and 3D map is also presented in figure 28. The 2D and 3D features overlap well, suggesting that both maps are working well at displaying obstructing objects.

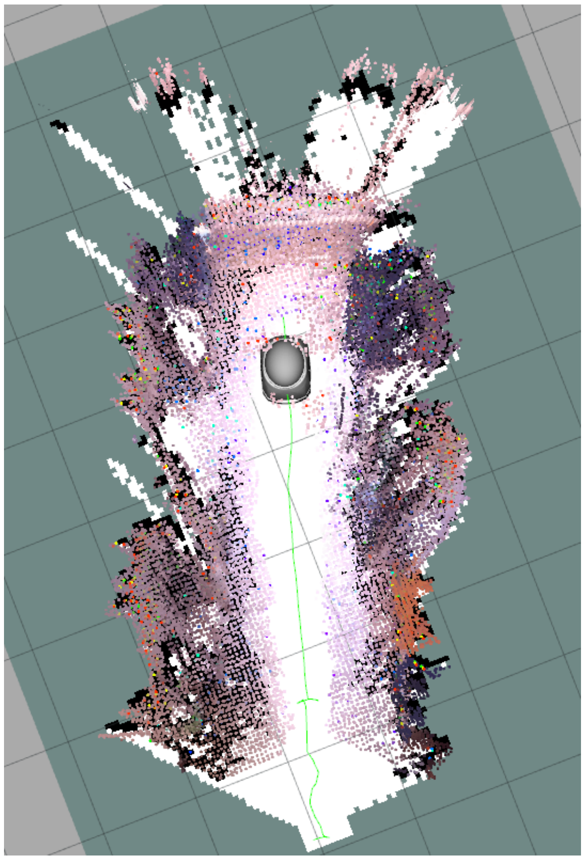


Figure 28: A 3D map on top of the 2D map of the Cyborg office during VSLAM.

The resulting 2D and 3D maps performed better than expected, as the cyborg office is a relatively small office with various obstructing objects. This meant that a large number of objects and obstructions had to be detected. Despite this, the unoccupied and occupied space was accurately detected. However, some mapping error did occur and is depicted in figure 29 by the black circle. The camera should have stopped mapping beyond the wall by the window. Instead, it kept adding to the map. The reason behind this could be due to rapid movements, tricking the rtabmap node into believing that a new area had been detected. Another explanation could be that it detected the window above and added that to the map. This error has no impact on the maneuverability of the Cyborg if used as a navigation map. The Cyborg has no chance of getting to that part, as it detected a wall there.

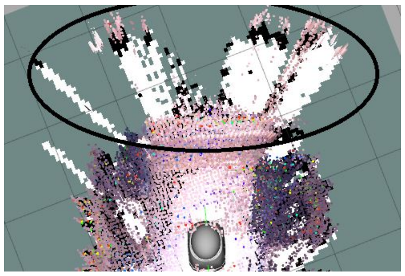


Figure 29: Error in the map, marked by the black circle.

8 Implementing object detection with the Cyborg

This section describes how YOLO object detection was implemented using the ZED stereo camera.

8.1 Implementing YOLO using the ZED camera

A python script called zed_darknet.py was included from the zed-yolo GitHub repository by Stereolabs [39]. This was a Python 3 wrapper for identifying objects in images. One of the imported classes was cv2. However, cv2 was imported under Python 2 by default with ROS Kinetic. To make cv2 work with Python 3, the following lines were added:

```
1  import sys
2  sys.path.remove('/opt/ros/kinetic/lib/python2.7/dist-packages')
3  import cv2
```

```
4  if __name__ == "__main__":
5      rospy.init_node("Object_detection_node", anonymous = False)
6      pub = rospy.Publisher("person_position", FloatVector, queue_size = 10)
7      floatvector = FloatVector()
8      while not rospy.is_shutdown():
9       main(sys.argv[1:])
```

To create an interface between ROS and YOLO, a ROS node was initialized on line 5 called object_detection_node. This node published to the *person_position* topic with the message type FloatVector. This was a custom ROS message containing a Point array. An object of this message was created on line 7. The **main()** function on line 9 was specified to run while the ROS node was active. This function was responsible for performing the object detection and depth calculations.

```
10     thresh = 0.6
11     config_path = "../libdarknet/cfg/yolov3.cfg"
12     weight_path = "yolov3.weights"
13     meta_path = "coco.data"
```

The first part of the main() function was to declare the detection threshold. This was set to 0.6, where the maximum threshold was 1. The threshold defined the lowest confidence score needed for an object to be detected, i.e., any objects with a confidence score greater than the threshold value would be accepted. The *config_path* defined where to find the structure of the neural network for YOLO version 3. The *weight_path* contained the weights for the given network and the *meta_path* contained information about the different classes, e.g. people and dogs.

```
14     init = sl.InitParameters()
15     init.coordinate_units = sl.UNIT.METER
16     init.set_from_stream(str("10.22.67.168"),30000)
17     cam = sl.Camera()
18     if not cam.is_opened():
19         log.info("Opening ZED Camera...")
20     status = cam.open(init)
21     if status != sl.ERROR_CODE.SUCCESS:
22         log.error(repr(status))
23         exit()
24     runtime = sl.RuntimeParameters()
25     # Use STANDARD sensing mode
```

```
26     runtime.sensing_mode = sl.SENSING_MODE.STANDARD
27     mat = sl.Mat()
28     point_cloud_mat = sl.Mat()
```

The next step was to open the ZED camera using the ZED Python API. The initial parameters for the camera were defined on line 14-16. The camera was specified to receive data from a broadcaster with a specified IP address and port. Line 17-23 created a camera object, passed the initial parameters to the open() function and checked if the camera was successfully opened or not. If the camera failed to open, e.g., the IP address of the broadcaster was wrong, the code was instructed to exit. The runtime parameters were defined on line 24. The sensing mode was set to standard, whereas the rest of the parameters were given their default values. Moreover, *mat* and *point_ cloud_ mat* were declared as objects of the `Mat()` class on line 27 and 28 respectively.

```
29         err = cam.grab(runtime)
30         if err == sl.ERROR_CODE.SUCCESS:
31             cam.retrieve_image(mat, sl.VIEW.LEFT)
32             image = mat.get_data()
33             cam.retrieve_measure(point_cloud_mat, sl.MEASURE.XYZRGBA)
34             depth = point_cloud_mat.get_data()
35             # Do the detection
36             detections = detect(netMain, metaMain, image, thresh)
```

The runtime parameters were passed to the `grab()` function on line 29 to grab the latest images from the camera. Line 30 checked if the camera returned successful behavior such that it could proceed with the code. Line 31 used the *cam* object to retrieve images from the left camera using the *mat* object. The *image*, on line 32, retrieved the data from the mat object in the form of a NumPy array. Next, on line 33, the depth measurements were retrieved and then converted into a NumPy array on line 34. Finally, the object detection took place on line 36, which used the `detect()` function.

The `detection()` function returned an array with information about the detection: index 0 returned the name of the detected object, e.g. a person. Index 1 returned the confidence score of the detected person, Index 2 gave information about the pixel location of the bounding box for the detected object and index 4 returned the index of the detected object from the coco.names file. For example, the person object had index 0 in coco.names.

```
37 if detection[0] == "person":
38     if xd != -1 and yd != -1 and zd != -1:
39         point = Point()
40         point.x = xd
41         point.y = yd
42         point.z = zd
43         floatvector.point.append(point)
44         pub.publish(floatvector)
```

An if-statement was implemented only to perform calculations if the label of the detected object was a person. This was done on line 37. Additionally, when a detected object was too close to- or too far from the camera, the given depth values were set to -1. A new if-statement was implemented on line 38 to only publish values when the camera received valid depth measurements. This stated that if any values in the x-, y-, or z direction was *not* equal to -1, it meant that new depth measurements were received. The depth values were then be published on line 44. The depth information was defined as a Point message, but published as a FloatVector.

8.1.1 Evaluating YOLO object detection

The object detection performance can be seen from figure 30. The best performing YOLO version in terms of frames per second (FPS) was YOLOV3-tiny. The FPS for the desktop computer was 27 FPS, whereas the Jetson NANO only managed 6 FPS. The detection itself performed well. YOLOV3 performed the worst in terms of FPS, but had the best detection performance. This made sense, though; Adding more layers to the CNN yielded better detection, but at the cost of being more computationally demanding and therefore causing a drop in FPS. The worst performing weight in terms of detection was YOLOV2-tiny. The detection was slower and sometimes failed to detect as it failed to reach the given threshold value.

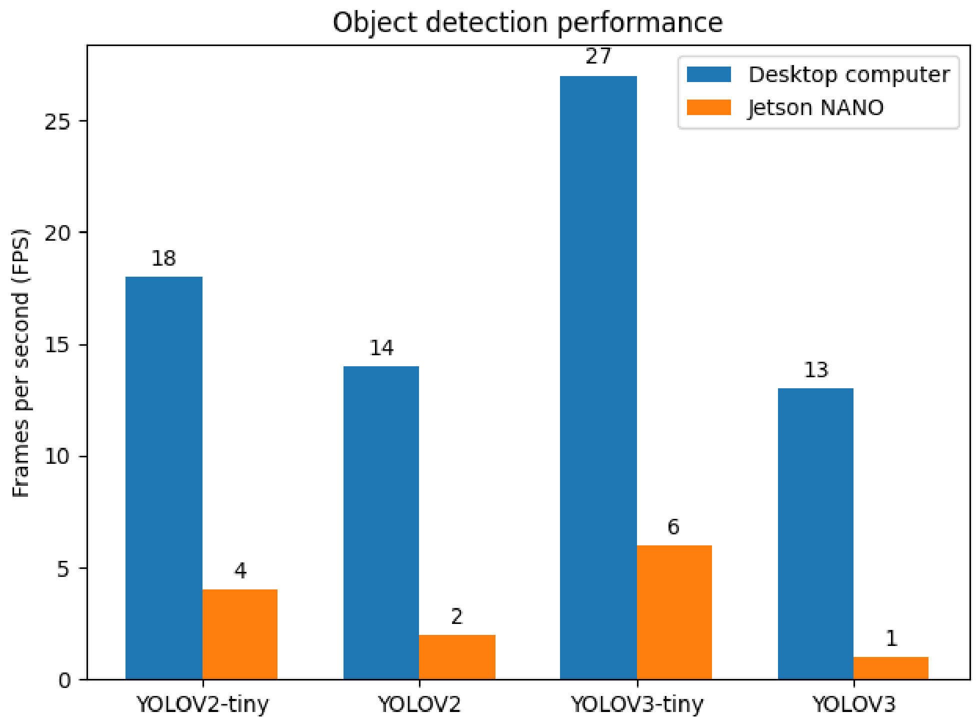
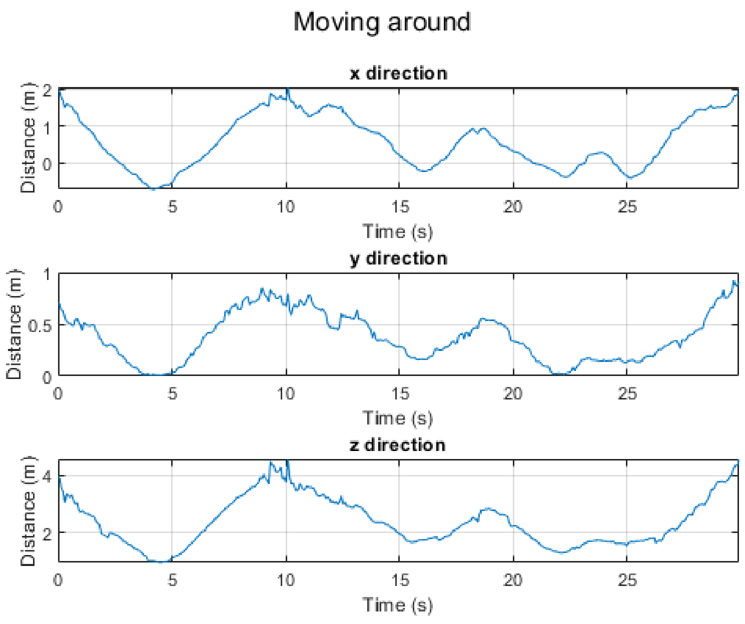
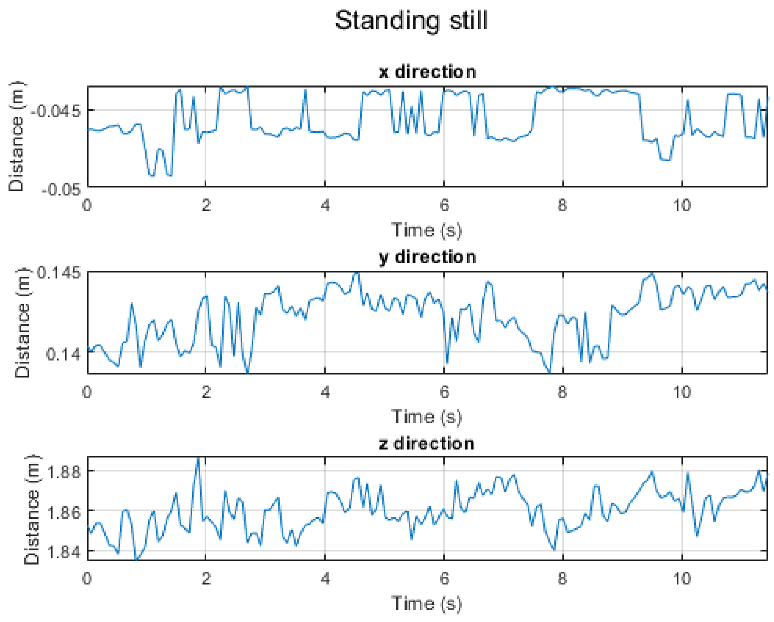


Figure 30: Object performance in terms of FPS for the Jetson Nano and the desktop computer.

Figure 31 shows two graphs of two different rosbags, each recording depth measurements of a detected student. Figure 31a shows the graph of a student that was constantly moving. The x-axis shows the time in seconds, whereas the y-axis shows the distance in meters from the camera to the person. The depth measurements seemed to be good, with some noise in the y-direction. Figure 31b shows the depth measurements of a person standing completely still. It was clear that this measurement contained a significant amount of noise. Note that the y-axis was scaled differently to that of figure 31a.



(a) Depth measurements while moving around.



(b) Depth measurements while standing completely still.

Figure 31: Depth measurement results.

The resulting confidence score from walking back and forth in the cyborg office with the camera directly pointed at the student can be seen from figure 32. This shows the confidence scores produced by the different weights. YOLOv3 yielded the best result with a confidence score close to 1 most of the time, whereas the minimum score was 0.8741. It had no problem detecting the person throughout the test. The small peaks happened when the person was changing the direction, i.e., turning around.

The second-best result came from YOLOv2, which had satisfactory results with a relatively small dispersion in the measurements. However, it got very close to a confidence score of 0.6. Any lower than this, and it would no longer be able to detect the student. Both YOLOv3-tiny and YOLOv2-tiny had less satisfactory results, with a higher disparity and a lower average than the previous two. Additionally, these weights would occasionally fail to detect the student walking, as the confidence score would reach below 0.6 on multiple occasions.

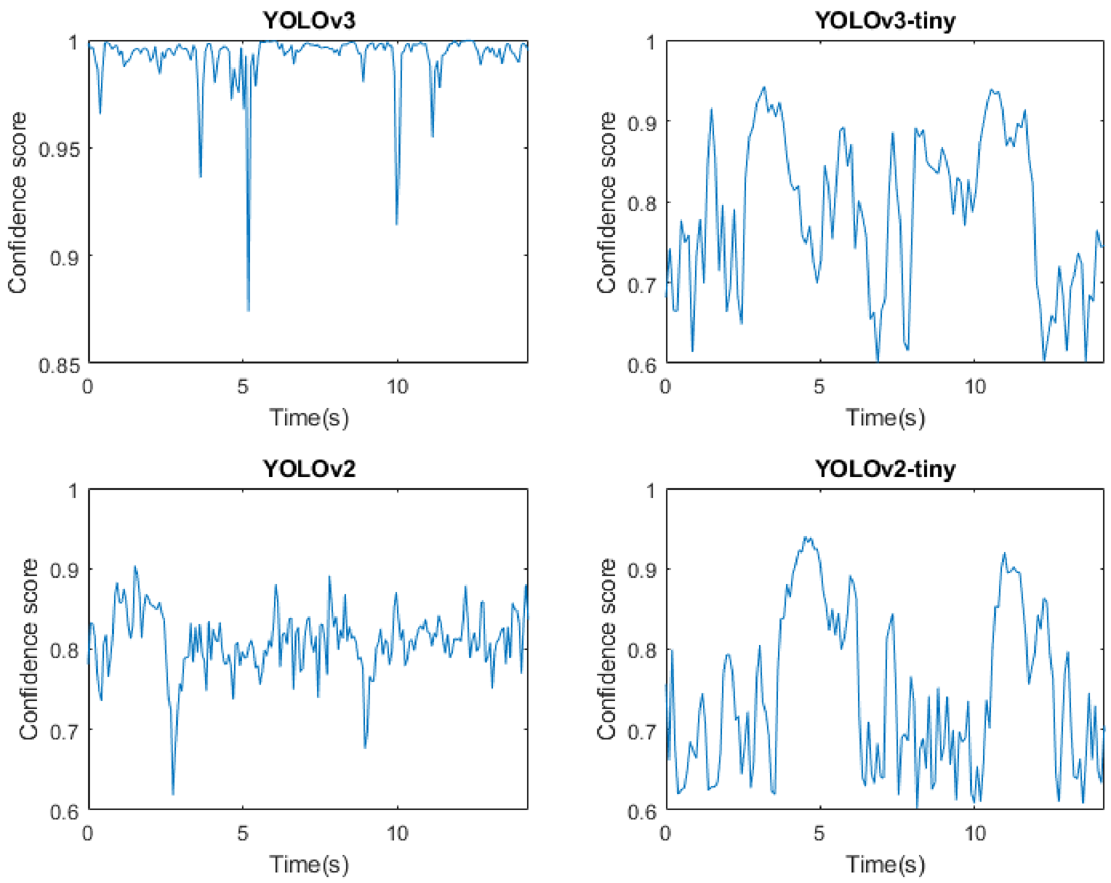


Figure 32: Showing how the confidence score for the different weights changed over time.

Table 1: The variance of the confidence score measurements

Weights	Variance(σ^2)
YOLOv3	0.00014646
YOLOv3-tiny	0.0091
YOLOv2	0.0016
YOLOv2-tiny	0.0094

The variance of the different weights can be found in table 1. It shows that YOLOv3 had a significantly smaller variance compared to the rest of the weights. YOLOv2 had a much higher variance than YOLOV3, but lower than both YOLOv2-tiny and YOLOv3-tiny, which both had very similar variance. The similarity in the variance for both the tiny weights came from the fact that both methods had 15 layers in the neural network. YOLOv3 had 106 layers, and YOLOv2 had only 31 layers, which explains the results.

9 ROS Navigation - Autonomous tracking of students using object detection

This section describes the procedures of creating a navigation controller capable of detecting a student using the object detection implemented in section 8 and follow the student.

9.1 Specification and Requirements

Based on the object detection module implemented in section 8, and the navigation stack described in section 2.11, the following specification and requirements are:

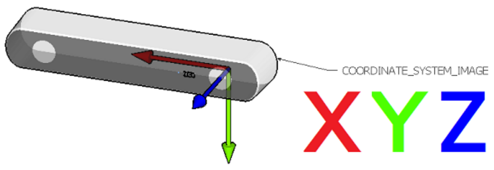
- The robot shall be capable of detecting people and receive navigation goals using the move_base package
- Two coordinate frames shall be implemented: *person_ detection_frame* and *person*. The former shall be used as the frame of reference for incoming depth measurements, and the latter shall track the person at all times by updating its position relative to person_detection_frame.
- Mount the camera and the Jetson Nano onto the Cyborg and enable streaming for the ZED camera. The desktop computer shall listen to the stream and run *all* ROS nodes related to the navigation controller and object detection. The Jetson Nano shall run only the ZED node.
- The Cyborg Computer shall only run the RosAria node, as well as costmap_2d navigation (see section 2.11)
- (Optional) Publish path information about where the student has moved and display it in RViz.

9.2 Creating a Navigation controller for the Cyborg

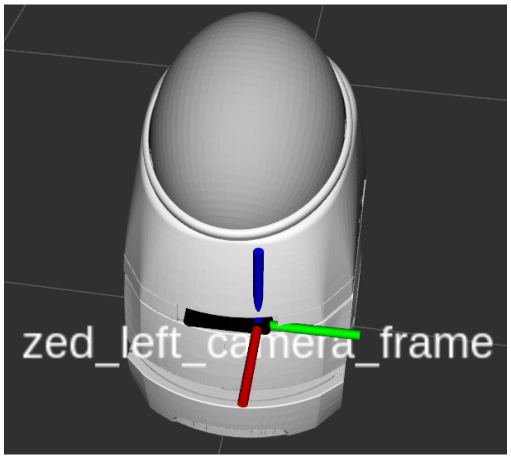
This section describes the implementation of a controller capable of navigating the Cyborg towards a detected student within the Cyborg Office.

9.2.1 Implementation the controller

Two new coordinate frames were implemented: *person_detection_frame* and *person*. The first frame was designed to match the orientation of the coordinate frame of the ZED camera. The latter was designed to represent the position of a student relative to a map. The default orientation of the axis provided by the ZED API is shown in figure 33a.



(a) ZED coordinate system.



(b) zed_left_camera_frame.

Figure 33: The default coordinate frame from the ZED API is represented on the left. The zed_left_camera_frame is represented in RViz on the right.

To create the person_detection_frame, the origin and orientation had to be defined. From figure 33b, the origin of zed_left_camera_center could be used as the frame of reference to represent the origin of the new frame. However, the orientation had to be adjusted to match figure 33a. Therefore, by creating a new frame relative to the zed_left_camera_frame, the orientation was adjusted by performing a roll angle of $\frac{3\pi}{4}$ to align the x-axis, followed by a yaw angle of $\frac{3\pi}{4}$ to align the y- and z-axis. Finally, the depth measurements from section 8 had to be averaged to avoid the noisy readings from figure 31b. The corresponding implementation in ROS was done in C++:

```cpp
class TFClass
{
public:
  TFClass()
  {
    pointsub_ = point_nh_.subscribe("/person_position", 1, &TFClass::caimplementedllback,
    this);
    optimized_point_pub_ = point_nh_.advertise<geometry_msgs::Point>("
    Optimized_person_distance",10000,true);
  }
```

A class called **TFClass** was created, with its constructor on line 4. Line 6 subscribed to the data provided by the /person_position topic and passed it to a callback function. A publisher was created on the subsequent line, which published a geometry_msgs Point message to the Optimized_person_distance topic.

```
 9  void callback(const ros_detection::FloatVector::ConstPtr& pointData)
10  {
11    quat_.setRPY(4.71238898, 0, 4.71238898);
12    quat_.normalize();
13    transform_.setOrigin(tf::Vector3(0,0,0));
14    transform_.setRotation(quat_);
15    br_.sendTransform(tf::StampedTransform(transform_, ros::Time::now(), "
      zed_left_camera_frame", "person_detection_frame"));
```

Here, the callback function was created, containing the subscribed data in *pointData*. Next, an object of the tf::Quaternion class used the *setRPY()* method to get the quaternion angles from the roll pitch and yaw angle. The roll and yaw angles were set as $\frac{3\pi}{4}$. The corresponding quaternion angles were normalized on line 12. The origin was set as $(0, 0, 0)$, which corresponded to the same origin as "$zed\_left\_camera\_frame$". On line 15, an object of class tf::TransformBroadcaster broadcast the data to $person\_detection\_frame$, relative to "$zed\_left\_camera\_frame$".

The subscribed data that was passed to the callback function was quite noisy and had to be averaged over a set of incoming measurements:

```
16    temp_point_.x = 0;
17    temp_point_.y = 0;
18    temp_point_.z = 0;
19    int count = 0;
20    current_size_ = pointData->point.size();
21
22    if (current_size_ >=10)
23    {
24      for (int i = current_size_-10; i < current_size_; i++)
25      {
26        temp_point_.x = temp_point_.x + pointData->point[i].x;
27        temp_point_.y = temp_point_.y + pointData->point[i].y;
28        temp_point_.z = temp_point_.z + pointData->point[i].z;
29        count++;
30      }
31      testpoint_.x = temp_point_.x/count;
32      testpoint_.y = temp_point_.y/count;
33      testpoint_.z = temp_point_.z/count;
34      optimized_point_pub_.publish(testpoint_);
35    }
36  }
```

On line 16-18, a class attribute called temp_point_ of message type geometry_msgs::Point had its x-, y- and z attributes set to zero. Moreover, a counter called *count* was set to zero on line 19. On line 20, a variable called current_size_ continuously kept track of the size of the incoming array of depth measurements. The if-statement on line 21 specified to average the incoming messages if the number of incoming messages was greater than or equal to 10. The for-loop on line 23 looped over the current index of the subscribed array subtracted by ten up to the current index. The values were added up on line 26-28 while updating the count variable on line 29. Outside this for-loop, a new variable of message geometry_msgs::Point, testpoint_ assigned its x-, y- and z-variables as the average of the data from the for-loop. Finally, the point was published on line 34.

```cpp
37 int main(int argc, char **argv)
38 {
39   ros::init(argc, argv, "personTF");
40   TFClass pathobj;
41   ros::spin();
42   return 0;
43 }
```

Inside the main function, the ROS node "personTF" was initialized. On line 40, an object of the TFClass was created and ran the class constructor. Finally, ros::spin() kept the program from exiting until the node was stopped.

A second frame was created to track the position of the detected student relative to the global /map frame. The procedure consisted of taking the measurements from the /Optimized_person_distance topic, which was relative to the person_detection_frame, and transform them into the /map frame using tf. Additionally, a publisher was created, which calculated the angle of a detected student relative to the horizontal Field of View (FOV) of the camera (see figure 34).

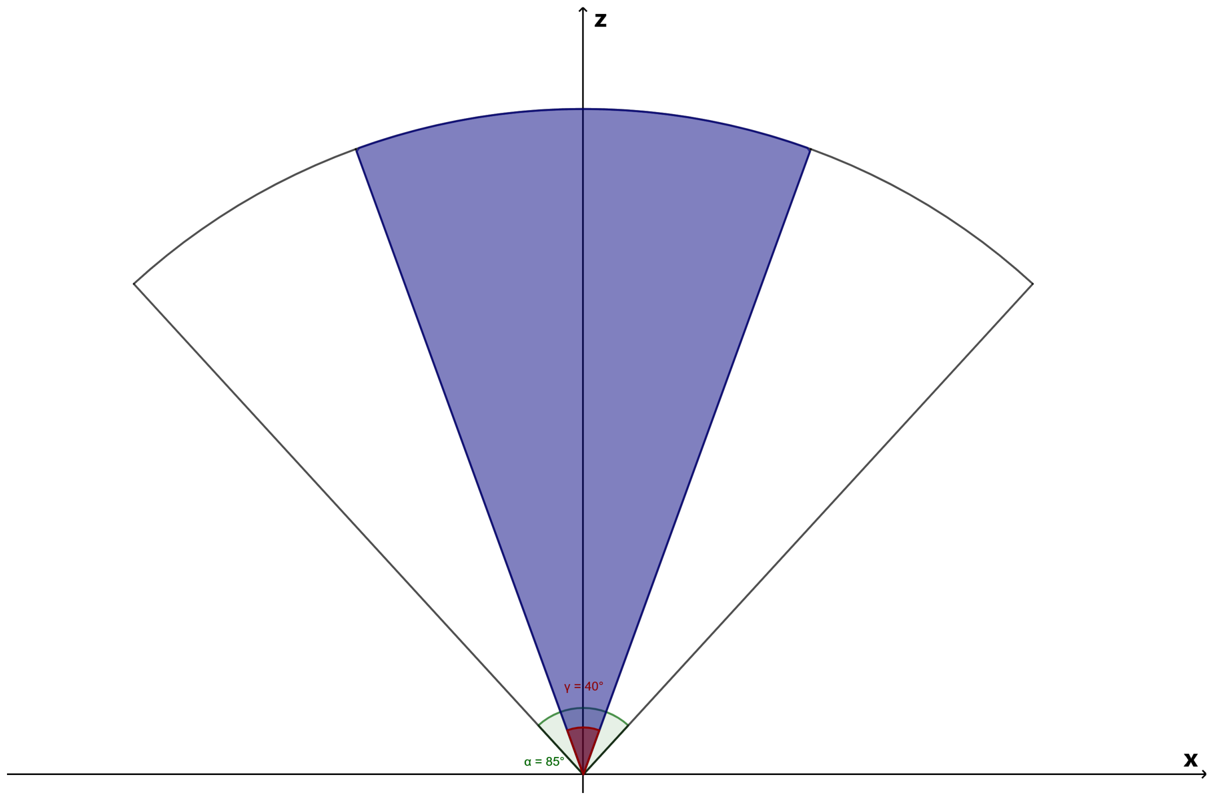


Figure 34: Horizontal FOV for the ZED camera.

The equation for the FOV was given by:

$$FOV\_angle = \pi - atan2\left(\frac{z}{x}\right) \tag{3}$$

Moreover, the distance between the Cyborg and the student was given by:

$$d = \sqrt{x^2 + z^2} \tag{4}$$

The implementations were made in C++:

```
1  class PersonBroadcaster
2  {
3    public:
4    PersonBroadcaster()
5    {
6      dist_sub_ = nh_.subscribe("/Optimized_person_distance",1, &PersonBroadcaster::callback,
       this);
7      path_pub_ = nh_.advertise<nav_msgs::Path>("person_path",1, true);
8      move_base_pub_ = nh_.advertise<geometry_msgs::Point>("move_base_goal_topic",1,true);
9      anglePub_ = nh_.advertise<std_msgs::Float64>("FOV_angle", 1, true);
10     distancePub_ = nh_.advertise<std_msgs::Float64>("distance", 1, true);
11   }
```

A class called *PersonBroadcaster* was created. Its constructor was defined on line 4, which created a subscriber to the /Optimized_person_position topic, and passed the data to a callback function. Moreover, four publishers were defined on line 7-10, which published to the following topics: *person_path* which contained the path of the detected person, *move_ base_ goal_ topic* which contained a navigation goal, the *FOV_ angle* containing the angle of the detected student and finally the */distance* topic which defined the distance between the Cyborg and the detected student.

```
12 void callback(const geometry_msgs::Point::ConstPtr& pointData)
13   {
14     personpose_.pose.position.x = pointData->x;
15     personpose_.pose.position.y = 0;
16     personpose_.pose.position.z = pointData->z;
17     personpose_.pose.orientation.w = 1;
18     personpose_.header.stamp = ros::Time(0);
19     personpose_.header.frame_id = "person_detection_frame";
```

Inside the callback function, the position of the detected student was assigned to personpose_, which was of type geometry_msgs::PoseStamped. The x and z-position were given by the corresponding x- and z- position provided by pointData. The y-position of the student was set to zero, corresponding to a constant height of 0 relative to person_detection_frame. As the Cyborg was confined to the horizontal plane, it made little sense to track the vertical changes of a detected student. Only the quaternion angle w was defined in this case, and was set to 1. On line 19, the frame_id was defined to be "person_detection_frame", which was the frame that the measurements originated from.

```
20     listener_.waitForTransform("map","person_detection_frame",ros::Time::now(),ros::Duration
       (3.0));
21     listener_.transformPose("map", personpose_, mappose_);
22     transform_.setRotation(tf::Quaternion(mappose_.pose.orientation.x,mappose_.pose.
       orientation.y,mappose_.pose.orientation.z,mappose_.pose.orientation.w));
23     transform_.setOrigin(tf::Vector3(mappose_.pose.position.x, mappose_.pose.position.y,
       mappose_.pose.position.z));
24     br_.sendTransform(tf::StampedTransform(transform_, ros::Time::now(), "map", "person"));
```

Line 20 specified ROS to wait for an available transform between the map and person_detection_frame. On the following line, **transformPose()** was used to represent the data provided in personpose_ to the map frame. The resulting transformation between the frames was assigned to the variable mappose_ of type geometry_msgs::PoseStamped. On line 22, the pose for a new coordinate frame was defined as the pose of mappose_. Finally, on line 24, the new coordinate frame *person* was created relative to the map frame.

```cpp
25     path_.header.stamp = ros::Time::now();
26     path_.header.frame_id = "map";
27     posestamped_.header.stamp = ros::Time::now();
28     posestamped_.header.frame_id = "map";
29     posestamped_.header.seq = seq;
30     posestamped_.pose.position = mappose_.pose.position;
31     posestamped_.pose.orientation = mappose_.pose.orientation;
32     path_.poses.push_back(posestamped_);
33     path_pub_.publish(path_);
34     move_base_pub_.publish(posestamped_.pose.position);
35     FOV_angle.data  = -std::atan2(pointData->z, pointData->x) * 180 / 3.141592 +90;
36     anglePub_.publish(FOV_angle);
37     dist = std::sqrt(std::pow(x,2) + std::pow(z,2));
38     dist_pub.publish(dist);
39     seq++;
```

Next, the path of the person frame was defined throughout line 25-32, and published on line 33. The stamp of the path message was defined on line 25, and the frame was defined on line 26. The position and orientation of the class attribute posestamped of type geometry_msgs::PoseStamped was defined on line 27-31. This variable was appended to the pose attribute of path_ by using the push_back function. Next, on line 33, the path was published to the person_path topic, and the navigation goal was published to move_base_goal_topic on line 34. The FOV_angle was defined according to equation 3 and the distance between the Cyborg and the student was defined according to equation 4. Finally, the variable seq was incremented for each time the callback was called on line 39.

A final C++ file implemented a navigation controller that sent out navigation goals to the Cyborg.

```cpp
int main(int argc, char** argv)
{
  ros::init(argc,argv,"ControllerNode");
  ros::NodeHandle nh;
  actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> myclient("move_base", true);
  ROS_INFO("Waiting for the move_base action server");
  myclient.waitForServer();
  ROS_INFO("Connected to the move base server");
  ROS_INFO("");

  tf::TransformListener listener;
  move_base_msgs::MoveBaseGoal goal;

  while(true)
  {
    movebaseaction(nh, listener, goal, myclient);
  }
  return 0;
}
```

Inside main() a ROS node called "ControllerNode" was created, as well as a node handle on line 4. An object called myclient of class actionlib::SimpleActionClient was created on line 5. The *waitForServer()* method was called on line 7, which specified the program to wait until a connection to the server had been established. On line 11, an object of the tf:TransformListener and an object of class move_base_msgs::MoveBaseGoal was created, which was used inside the *movebaseaction()* function on line 16. This function was inside a while loop that ran continuously until the program was forced to shut down. Four different arguments were passed to the function on line 16: The ROS node handle, the listener object, the goal object and the myclient object.

```
20 void movebaseaction(ros::NodeHandle& nh, tf::TransformListener& listener, move_base_msgs::
     MoveBaseGoal& goal,
21   actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>& myclient)
22 {
23 
24   geometry_msgs::Point point = *(ros::topic::waitForMessage<geometry_msgs::Point>("
     move_base_goal_topic", nh));
25   std_msgs::Float64 FOV_angle = *(ros::topic::waitForMessage<std_msgs::Float64>("/FOV_angle"
     , nh));
26   std_msgs::Float64 distance = *(ros::topic::waitForMessage<std_msgs::Float64>("/distance",
     nh));
```

The inputs to the function on line 20 was *called by reference*. In C++, this meant that the *address* of the variable was passed to the function, not the value itself. On line 24-26, the data from three different topics were obtained using ros:: topic:: waitForMessage. This specified ROS to wait for a *single* incoming message. The topics were: move_base_goal_topic, FOV_angle and distance. This function returned the address of the value stored in memory for the specific topic. The value was obtained using the dereference operator *asterix* (*).

```
27   tf::StampedTransform stamp;
28   listener.waitForTransform("map","base_link",ros::Time::now(),ros::Duration(3.0));
29   listener.lookupTransform("map","base_link",ros::Time(0), stamp);
30 
31   goal.target_pose.header.frame_id = "map";
32   goal.target_pose.header.stamp = ros::Time::now();
33   goal.target_pose.pose.position.x = point.x;
34   goal.target_pose.pose.position.y = point.y;
35   goal.target_pose.pose.position.z = 0;
36   goal.target_pose.pose.orientation.x = stamp.getRotation().x();
37   goal.target_pose.pose.orientation.y = stamp.getRotation().y();
38   goal.target_pose.pose.orientation.z = stamp.getRotation().z();
39   goal.target_pose.pose.orientation.w = stamp.getRotation().w();
```

The stamp object was created and passed to the lookupTransform function to obtain the pose of base_link of the robot relative to the map frame. On line 31-39, the navigation goal was defined (see section 2.8). The frame_id of the target was defined as the map frame, which was the inertial frame. The position of the target was the points x and y from move_base_goal_topic. The robot was incapable of moving vertically, such that the z-coordinate was set to 0. The orientation of the target was defined as the orientation of the Cyborg, or the base_link, at the time lookupTransfrom was called.

```
40   ROS_INFO_STREAM("Sending the following navigation goal: " << goal);
41   myclient.sendGoal(goal);
42   ROS_INFO_STREAM("Angle = "<<FOV_angle);
```

```
43   ros::Duration(1).sleep();
44     while (myclient.getState() == actionlib::SimpleClientGoalState::ACTIVE && std::abs(
       FOV_angle.data) < 20 && distance.data <= 1)
45     {
46       ROS_INFO_STREAM(myclient.getState().toString().c_str());
47       FOV_angle = *(ros::topic::waitForMessage<std_msgs::Float64>("/FOV_angle", nh));
48       ROS_INFO_STREAM("CURRENT ANGLE IS :" << FOV_angle);
49     }
50 }
```

The goal was sent using the sendGoal() method on line 41. The program was then instructed to wait for 1 second such that the robot had enough time to generate a plan and start executing it before running the while loop on line 44. The while loop ran as long as a goal was active, a detected person was within $\pm$ 20°of the maximum FOV and the student was within 1 meter of the Cyborg. Thus, the while loop acted more or less as a wait function; While the conditions were met, no new plans were generated. This allowed the Cyborg to move along the latest generated plan continuously. Otherwise, a new plan would be generated.

A flowchart of the navigation controller can be seen from figure 35.

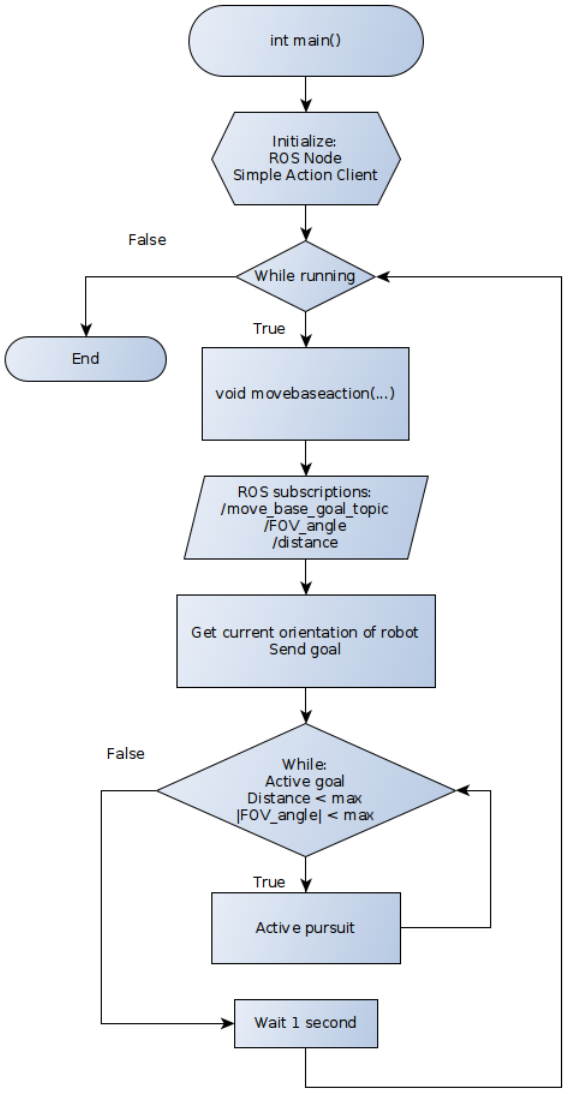


Figure 35: Flowchart for the navigation controller.

Before running the different nodes, the package.xml and CMakeLists.txt had to be modified to include the necessary imported classes. In CMakeLists.txt, the following changes were made:

```
find_package(catkin REQUIRED COMPONENTS
  message_generation
  roscpp
  tf
)
```

The message_generation had to be included for generating language bindings of messages [40]. Next, the roscpp was included as the nodes were written in C++ and the roscpp provided an interface with ROS topics, services and parameters [41]. Finally, *tf* was included to include the necessary functionalities from the tf package.

```
add_message_files(
  FILES
  FloatVector.msg
)
```

The custom ROS message FloatVector had to be specified inside the add_message_files() function to be considered a ROS message.

```
generate_messages(
  DEPENDENCIES
  tf
)
```

To generate any added messages (not including custom messages), they had to be specified in the generate_messages() function. After the messages and various libraries were specified, the nodes had to be made executable from:

```
add_executable(\${PROJECT_NAME}_node src/frames.cpp)
target_link_libraries(\${PROJECT_NAME}_node \${catkin_LIBRARIES})

add_executable(\${PROJECT_NAME}_tf  src/get_tf.cpp)
target_link_libraries(\${PROJECT_NAME}_tf \${catkin_LIBRARIES})

add_executable(\${PROJECT_NAME}_actionclient  src/action_client_interface.cpp)
target_link_libraries(\${PROJECT_NAME}_actionclient \${catkin_LIBRARIES})
```

Note that *PROJECT_NAME* was ros_detection. These changes made is possible to run the different C++ files from terminal using

```
$ rosrun ros_detection <node_name>
```

Additionally, the Package.xml had to be modified:

```
<build_depend>message_generation</build_depend>
<build_depend>message_runtime</build_depend>
<build_depend>roscpp</build_depend>
<build_depend>tf</build_depend>
<build_export_depend>roscpp</build_export_depend>
<exec_depend>message_runtime</exec_depend>
<exec_depend>roscpp</exec_depend>
<exec_depend>tf</exec_depend>
```

This built and/or executed the dependencies roscpp, tf message_generation and message_runtime.

After running YOLO, the nodes related to the navigation controller was started in the following order (for the desktop computer): ros_detection_node, ros_detection_tf and ros_detection_actionclient. This required having a student directly in front of the camera before starting the nodes.

9.2.2 Evaluation of the controller

The resulting coordinate frame person_detection_frame can be seen from figure 36. The position and orientation matched the coordinate frame from figure 33a.

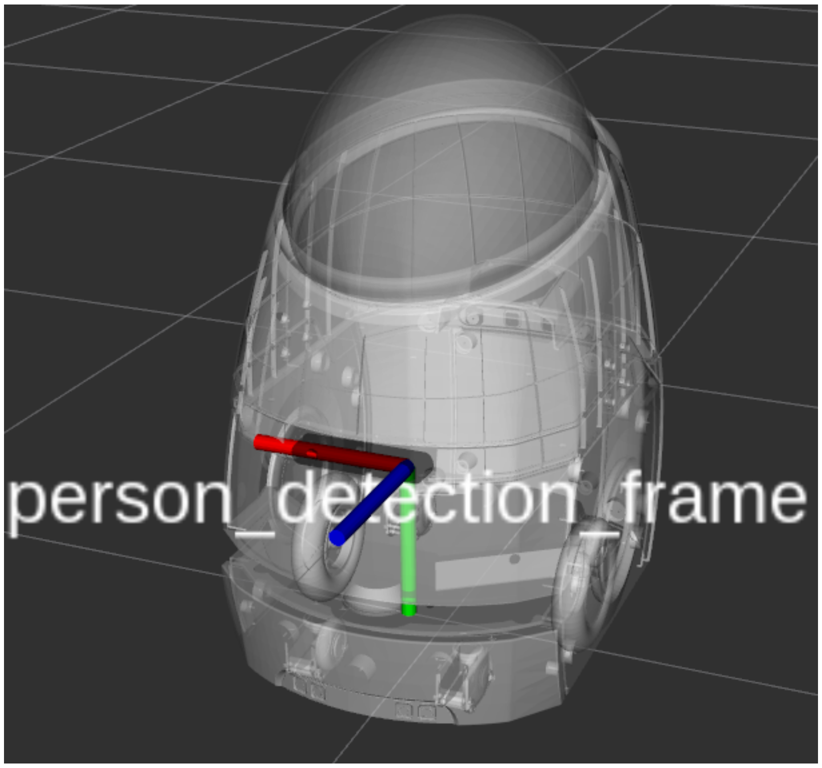


Figure 36: The new person_detection_frame displayed in RViz with the Cyborg model.

Moreover, the person frame was successfully created and is shown in figure 37

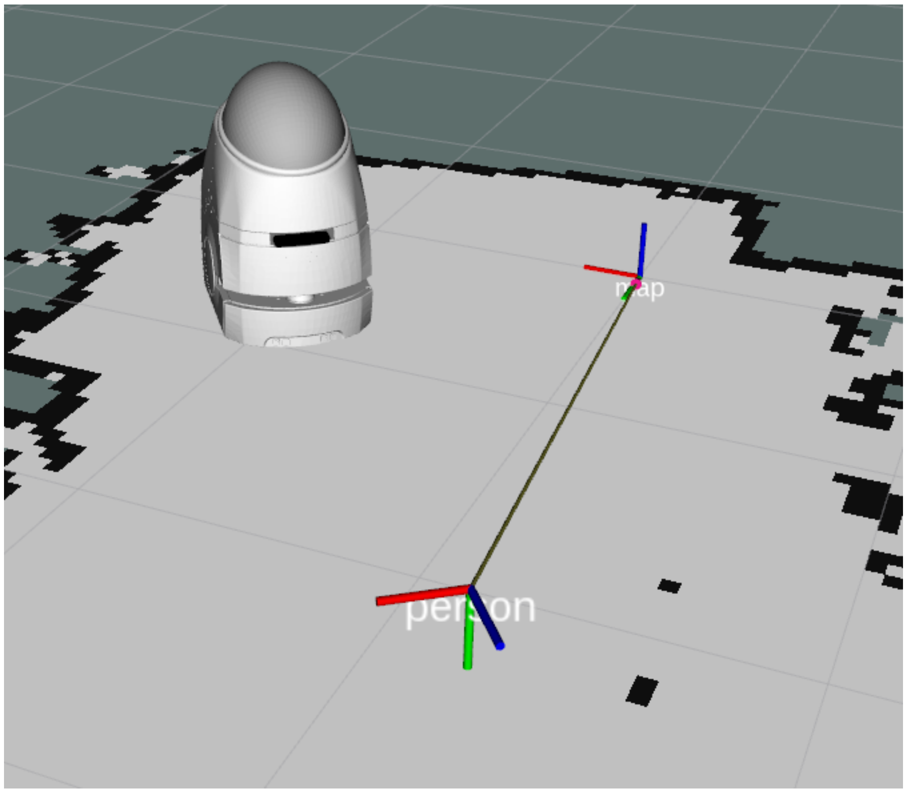


Figure 37: Showing the *person* coordinate frame in RViz.

The data used to broadcast the person frame was significantly less noisy than the data from figure 31b. However, the improved noise reduction came at the cost of having a less responsive frame. The larger amount of data it averaged, the less noisy the data became. However, this came at the price of having a less responsive frame, as it has to constantly wait for a certain amount of measurements before it can update its position. Having too small of a sampling size cause the data to become turbulent again, but more responsive. After testing with different sampling amounts, it was concluded that a sample of 10 measurements yielded a good combination of noise reduction and having a fast, responsive frame. The path of a student walking around the Cyborg office can be seen in figure 38. The path followed a smooth trajectory without jumping due to spikes in the measurements. Although these spikes in measurements might not matter when standing in a big room with few obstacles, inside the office, it did matter. When navigating in rooms like the Cyborg office, being able to track accurately is essential. A sudden jump in position might send the robot towards an obstacle, causing the robot to get stuck.

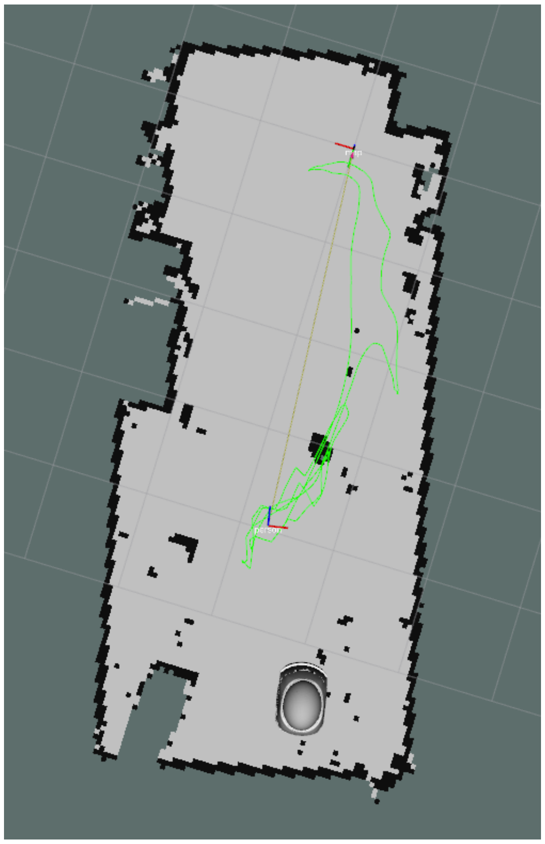
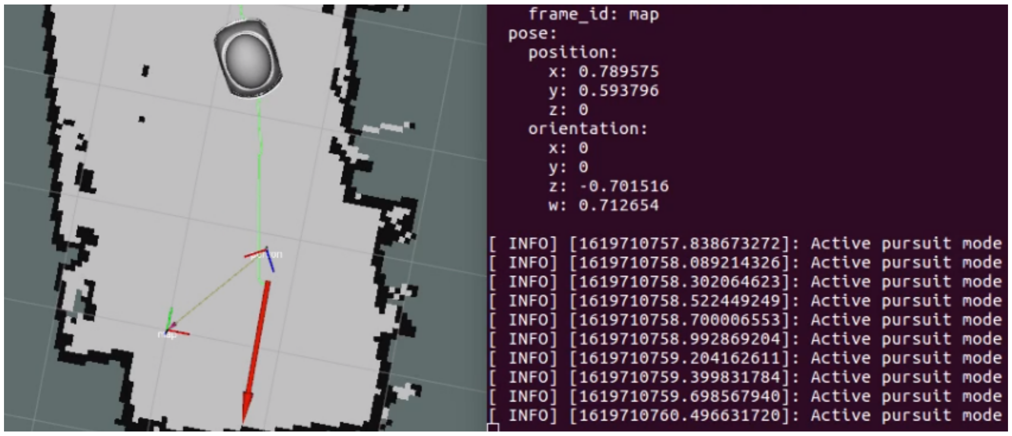


Figure 38: The green line shows the path taken by a detected student inside the Cyborg office.

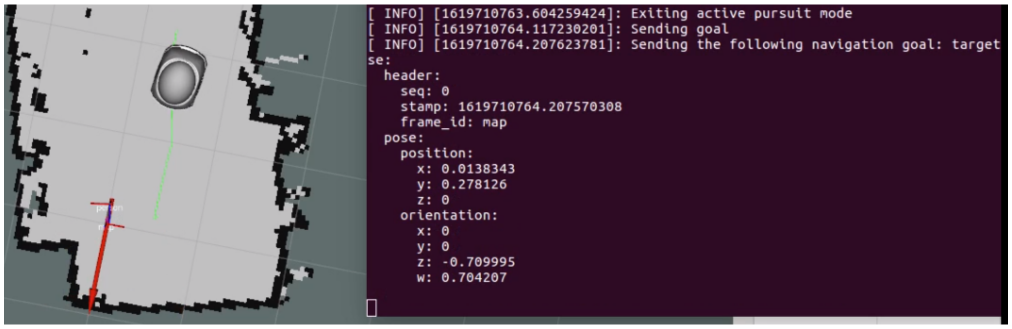
The navigation controller ran successfully and was capable of navigating the Cyborg towards a detected person using object detection implemented in section 8. Figure 39 shows a test of the Cyborg in the Cyborg Office.



```
  frame_id: map
pose:
  position:
    x: 0.789575
    y: 0.593796
    z: 0
  orientation:
    x: 0
    y: 0
    z: -0.701516
    w: 0.712654

[ INFO] [1619710757.838673272]: Active pursuit mode
[ INFO] [1619710758.089214326]: Active pursuit mode
[ INFO] [1619710758.302064623]: Active pursuit mode
[ INFO] [1619710758.522449249]: Active pursuit mode
[ INFO] [1619710758.700006553]: Active pursuit mode
[ INFO] [1619710758.992869204]: Active pursuit mode
[ INFO] [1619710759.204162611]: Active pursuit mode
[ INFO] [1619710759.399831784]: Active pursuit mode
[ INFO] [1619710759.698567940]: Active pursuit mode
[ INFO] [1619710760.496631720]: Active pursuit mode
```

(a) Active pursuit mode



```
[ INFO] [1619710763.604259424]: Exiting active pursuit mode
[ INFO] [1619710764.117230201]: Sending goal
[ INFO] [1619710764.207623781]: Sending the following navigation goal: target
se:
  header:
    seq: 0
    stamp: 1619710764.207570308
    frame_id: map
  pose:
    position:
      x: 0.0138343
      y: 0.278126
      z: 0
    orientation:
      x: 0
      y: 0
      z: -0.709995
      w: 0.704207
```

(b) Generating a new goal

Figure 39: Testing the navigation controller on the Cyborg.

Figure 39a shows the active pursuit mode, where the detected person was within 1 meter of the Cyborg, as well as being within ± 20°of the camera's FOV. The frame in front of the Cyborg was the student it was following, and the frame to the left was the inertial map frame. On the right side, a picture of the terminal window displayed the current goal and the message informing that it was running the active pursuit mode. The light green line from the robot to the red arrow displayed the path generated by the navigation controller from the Cyborg to the student. The red arrow shows the goal position and orientation. Figure 39b shows the case when the student stepped outside the maximum angle of ± 20 °. This caused the controller to generate a new goal.

The design of this controller performed well. The constraints ensured that a detected student would not leave the robot's FOV nor the maximum distance that the camera was capable of detecting people.

9.3 Creating a Navigation controller in a simulated environment

This section describes the implementation of a navigation controller in a simulated environment using MobileSim.

9.3.1 Specification and Requirements

Based on the object detection module implemented in section 8, and the navigation stack described in section 2.11, the following requirements and specifications are:

- Redesign the controller from section 9.2 to work with MobileSim.
- The speed of the Cyborg shall be regulated as a function of the distance between the Cyborg and the detected student

9.3.2 Implementation

The robot model was designed to have the camera in front of the Cyborg. This meant that in a real-life scenario, any object outside the camera's FOV would no longer be detected. Using a simulated robot, the camera would no longer move together with the Cyborg. For example, if the simulated robot was instructed to move towards a student being detected by the ZED camera, the robot would move. However, the camera would be stationary in real life. Therefore, the camera could not be used during the simulation. To work around this, a new design was implemented. The two nodes implemented in section 9.2, i.e. detection_frame_node and personTF, was used to set up the two coordinate frames person_detection_frame and *person*. A student was then instructed to move around while having its path recorded by a rosbag. The resulting path was then used as the target of the navigation goal.

When the rosbag was played, the information it contained was published to the topic it received it from in the first place. Since this path was prerecorded relative to a stationary camera, the distance- and fov_angle topic could not be used. The student's position relative to the camera during the simulation was not the same as the distance of the recorded path of the student in the simulation relative to the simulated robot. This meant that a new relative distance between the simulated robot and the student along the prerecorded path had to be calculated. This is depicted in detail in figure 40

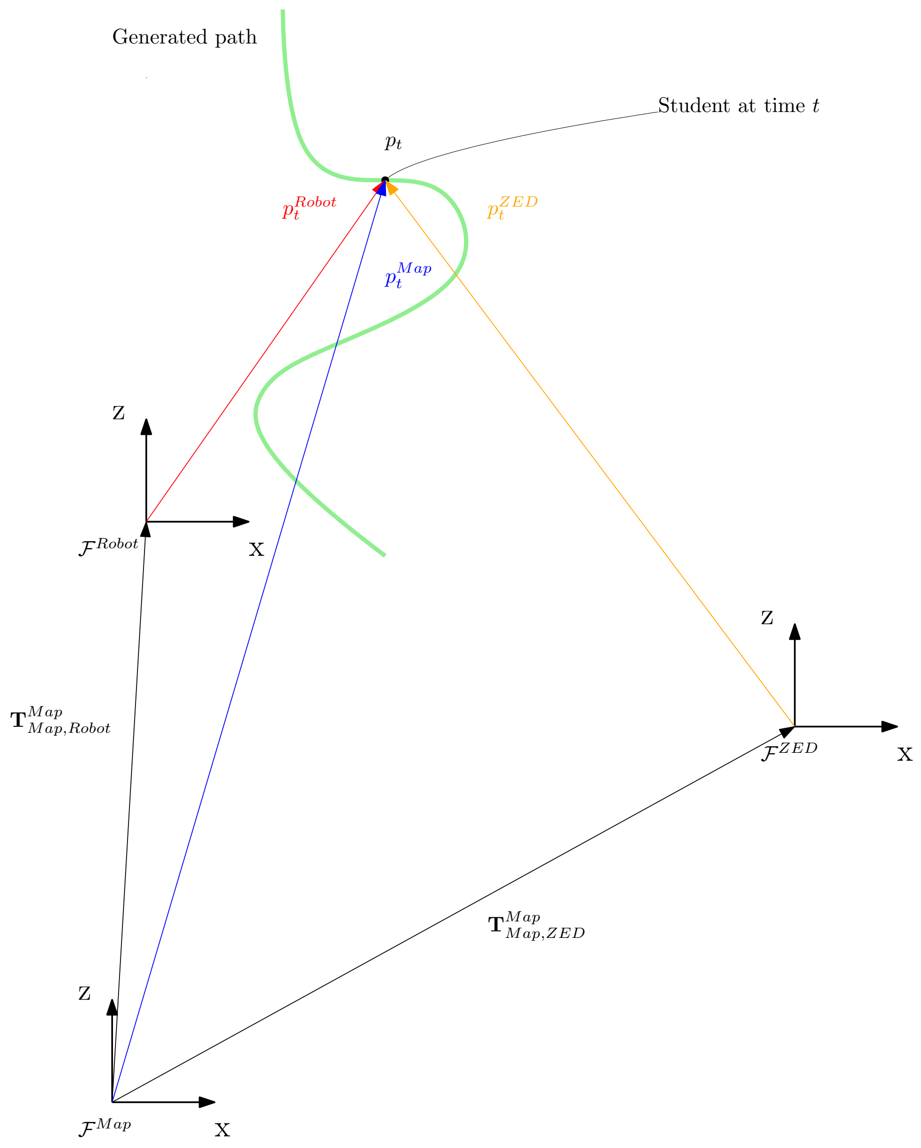


Figure 40: Representing the position of a student on an arbitrary path relative to different coordinate systems.

$\mathcal{F}^{Map}$ represented the stationary map frame, $\mathcal{F}^{ZED}$ represented the camera frame during the recording and $\mathcal{F}^{Robot}$ represented the simulated robot. The green line represented an arbitrary path of a student, where the point p_t represented the student along the path at time t. Moreover, p^{ZED} represented the position of the person at time t relative to the camera and p^{Robot} represented the position relative to the simulated robot. The position needed to represent the distance between the simulated robot and the student was therefore p^{Robot}.

The position and orientation of $\mathcal{F}^{ZED}$ and $\mathcal{F}^{Robot}$ relative to $\mathcal{F}^{Map}$ was represented by two homogeneous transformation matrices: $\mathbf{T}^{Map}_{Map,Robot}$ and $\mathbf{T}^{Map}_{Map,ZED}$, respectively. The position of the student relative to $\mathcal{F}^{Map}$ was given by:

$$\mathbf{p}_t = \mathbf{p}_t^{Map} = \mathbf{T}^{Map}_{Map,Robot}\mathbf{p}_t^{Robot} \quad (5)$$

Equation 5 can be rearranged to obtain an expression for $\mathbf{p}_t^{Robot}$:

$$\mathbf{p}_t^{Robot} = \left(\mathbf{T}_{Map,Robot}^{Map}\right)^{-1} \mathbf{p}_t^{Map} \tag{6}$$

The horizontal distance d, defined as the distance along the x and y axis, was given by:

$$d = \sqrt{x^2 + y^2} \tag{7}$$

Additionally, when $\mathbf{p}_t^{Robot}$ was known, the FOV angle could be calculated using equation 3. This was implemented in *distance.cpp*

```
1  class Distance
2  {
3    public:
4    Distance()
5    {
6      path_sub_ = nh_.subscribe("/person_path",1, &Distance::callback, this);
7      dist_pub_ = nh_.advertise<std_msgs::Float64>("relative_distance",10000,true);
8      FOV_pub_  = nh_.advertise<std_msgs::Float64>("FOV_angle",10000,true);
9    }
```

A class called Distance was created. Inside the constructor on line 4, a subscriber to the topic /person_path passed data to a callback function. Furthermore, two new publishers were created: "relative_distance" and "FOV_angle".

```
10   void callback(const nav_msgs::Path::ConstPtr& pathData)
11   {
12     seq = pathData ->header.seq;
13     posestamped_ = pathData -> poses[seq];
14     posestamped_.header.frame_id = "map";
15     posestamped_.header.stamp = ros::Time(0);
16     posestamped_pub_.publish(posestamped_);
17     listener_.waitForTransform("person_detection_frame","map",ros::Time::now(),ros::Duration
       (3.0));
18     listener_.transformPose("person_detection_frame", posestamped_,personpose_);
19     float x = personpose_.pose.position.x;
20     float z = personpose_.pose.position.z;
21     distance.data = std::sqrt(std::pow(x,2) + std::pow(z,2));
22     dist_pub_.publish(distance);
23     FOV_angle.data  = -std::atan2(z,x) * 180 / 3.141592 +90;
24     FOV_pub_.publish(FOV_angle);
25 }
```

The callback function contained the student's path, which was stored in posestamped_ on line 13. Next, the pose of the detected student at any given time was transformed into person_detection_frame on line 18. This implementation was equivalent to equation 6. The distance from the simulated robot and the student was calculated on line 21, according to equation 7 and published on line 22. The FOV was defined on line 23 according to equation 3 and published on line 24.

An exponential function was used to regulate the speed of the Cyborg. The function was given by the following equation:

$$f(x) = ab^x \tag{8}$$

Furthermore, the constants a and b from equation 8 could be found using two sets of two points (x_1, y_1) and (x_2, y_2):

$$\begin{aligned} a &= \frac{y_1}{b^{x_1}} \\ a &= \frac{y_2}{b^{x_2}} \\ b &= \left(\frac{y_1}{y_2}\right)^{\frac{1}{\Delta x}} \\ \Delta x &= x_1 - x_2 \end{aligned} \tag{9}$$

Using the points $(0.5, 0.2)$ and $(2, 0.8)$ gave the following constants:

$$\begin{aligned} \Delta x &= 0.5 - 2 = -1.5 \\ b &= \left(\frac{0.2}{0.8}\right)^{-\frac{2}{3}} \approx 2.5198 \\ a &= \frac{0.2}{2.5198^{0.5}} \approx 0.12599 \end{aligned} \tag{10}$$

Additionally, the function was designed such that if the distance from the robot and the student was less than or equal to 0.5, the output would be 0.2, and if the distance was greater than or equal to 2 meters, the output would be 0.8. A function, $g(x)$ was given by:

$$g(x) = \begin{cases} 0.2, & \text{if x < 0.5} \\ f(x), & \text{if } 0.5 \leq x \leq 2.0 \\ 0.8, & \text{if } x > 2.0 \end{cases}$$

Using the values from equation 10 in equation 8, the corresponding plot of $g(x)$ can be found in figure 41.

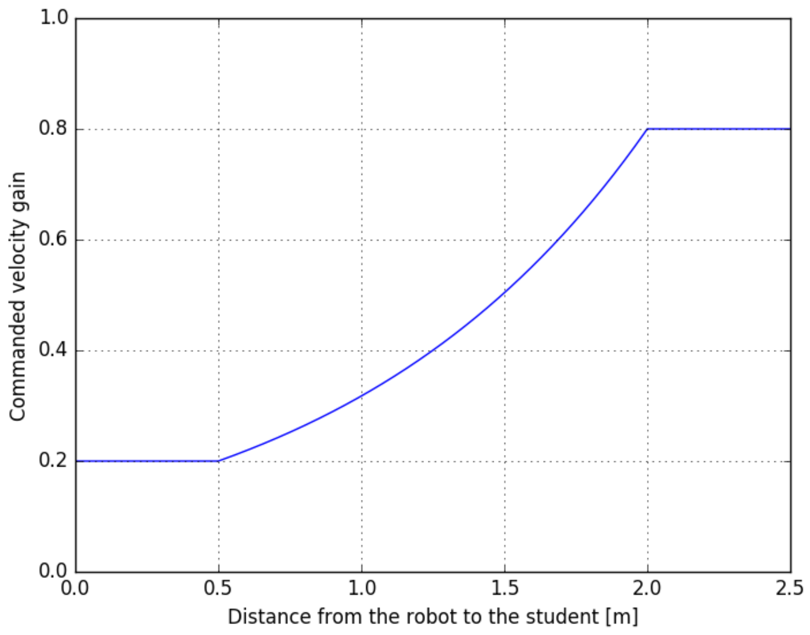


Figure 41: The resulting velocity function $g(x)$.

The output of $g(x)$ was used to tune the parameter max_vel_theta (see section 2.11), by using the *dynamic_reconfigure* ROS package. This package enabled means to update parameters at runtime without having to restart any nodes [42]. The complete implementation was done in a python file called active.py:

```python
1  #!/usr/bin/env python
2  import dynamic_reconfigure.client
3  import numpy as np
4  import math
```

Line 1 used a shebang referencing to the python interpreter. Line 2 imported the dynamic reconfigure client used to change ROS parameters at runtime. Line 3 imported the numpy library, and line 4 imported the math library. Next, the function $f(x)$ was defined:

```python
5  def f(x):
6      x1 = 0.5
7      x2 = 2.0
8      y1 = 0.2
9      y2 = 0.8
10     b = (y1/y2)**(1/(x1-x2))
11     a = y2/b**x2
12     return(a*b**x)
```

This function defined the constants calculated from equation 10 and returned the output of equation 8. Moreover, a listener function was created which initialized a node and subscribed to an active topic:

```python
13 def listener():
14     rospy.init_node('velocity_node', anonymous=True)
15     rospy.Subscriber("distance", Float64, callback)
16     rospy.spin()
```

The node was called "velocity_node" and subscribed to the "distance" topic. Line 16 kept python from exiting until the node was stopped.

```
17 def callback(data):
18     distance = data.data
19     if distance >= 0.5 and distance <= 2:
20         cost = f(distance)
21     if distance < 0.5:
22         cost = 0.1
23     if distance > 2:
24         cost = 0.8
25     client = dynamic_reconfigure.client.Client("move_base/TrajectoryPlannerROS", timeout=4,
       config_callback=None)
26     client.update_configuration({"max_vel_x":cost})
27     pub = rospy.Publisher('velocity', Float64, queue_size=10)
28     pub.publish(cost)
```

Line 17 created the callback function, which took the argument *data* containing the distance. Line 19, 21 and 23 contained the conditions of g(x), based on the distance. Line 25 created the object *client* of the class dynamic_reconfigure. The first argument of the class method client.Client was "move_base/TrajectoryPlannerROS", which contained the parameter *max_vel_theta*. The next line specified to update *max_vel_theta* with the variable *cost*, containing the output of $g(x)$. Finally, on line 27 and 28, a new publisher was created and published the velocity commands to the *velocity* topic.

9.3.3 Evaluation of the simulated controller

The path of the rosbag recording can be seen from figure 42.

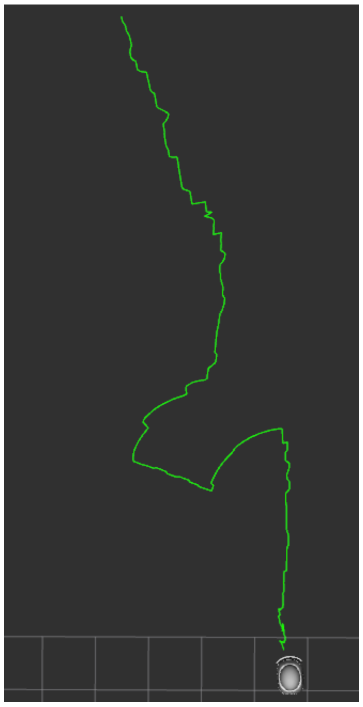


Figure 42: Path of the student, recorded by a rosbag.

Figure 43 shows three plots: The path of the Cyborg vs. the path of the student, the distance between them and the velocity commands sent to the robot. The Cyborg was capable of following the student throughout the simulation. However, it did not follow the line perfectly. This was because the Cyborg was instructed to follow different points along the student's path, not the actual line itself. The commanded speed also behaved as expected, showing that it never exceeded the limits of 0.2 and 0.8 (20% to 80%) of the maximum velocity. Finally, the Cyborg never lost track of the student.

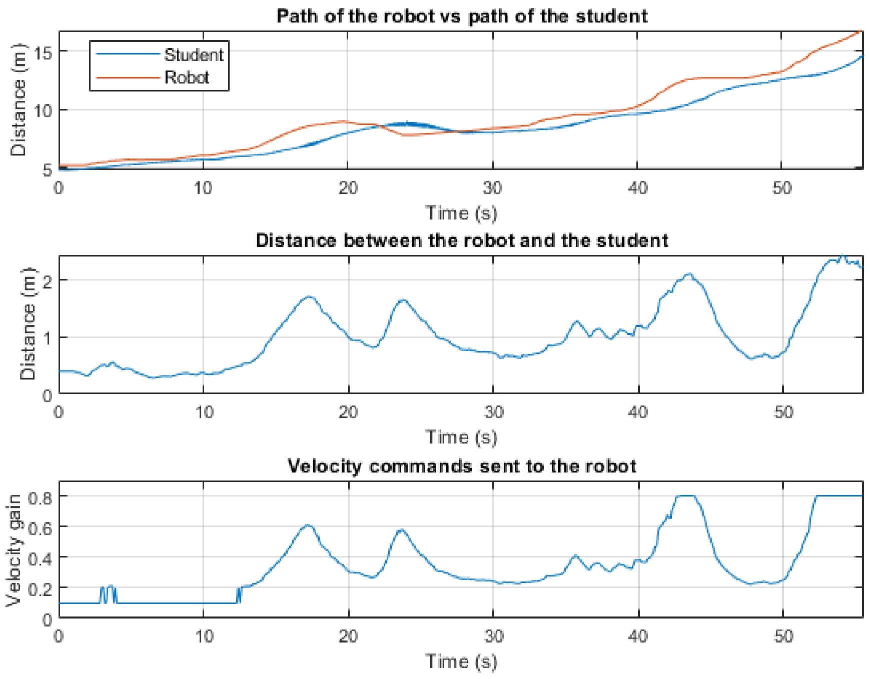


Figure 43: Results from simulations.

The simulation could not take place inside the Cyborg office, as it was too small. Moreover, any maps outside the office did not match the map used in MobileSim. This caused the Cyborg to either lose odometry information or get stuck on objects that were not supposed to be there in the first place. Consequently, the simulation ran without a map. This meant that the controller was tested without collision avoidance.

The different nodes and topics related to the detection of the student are shown in figure 44. The object_detection_node was publishing the position of the student to /person_position, which detection_frame_node subscribed to. The detection_frame_node created the person_detection_frame and published the optimized depth measurements to personTF, which finally created the person frame. Note that the topics FOV_angle and relative_distance did not appear here. The controller node did use the data from these topics, albeit using the waitForMessage function, which is not equivalent to subscribing to a topic.

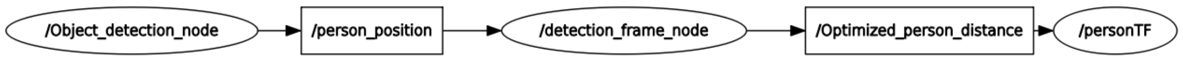


Figure 44: ROS computational graph related to object detection.

Figure 45 shows how the ControllerNode worked. The controller node published to move_base/goal and move_base/cancel, which was provided by the SimpleActionClient interface. The navigation goals were then interfaced with the move_base node, which published the velocity commends to /cmd_vel. Additionally, the laser scan and point cloud data from the RosAria node were used for localization.

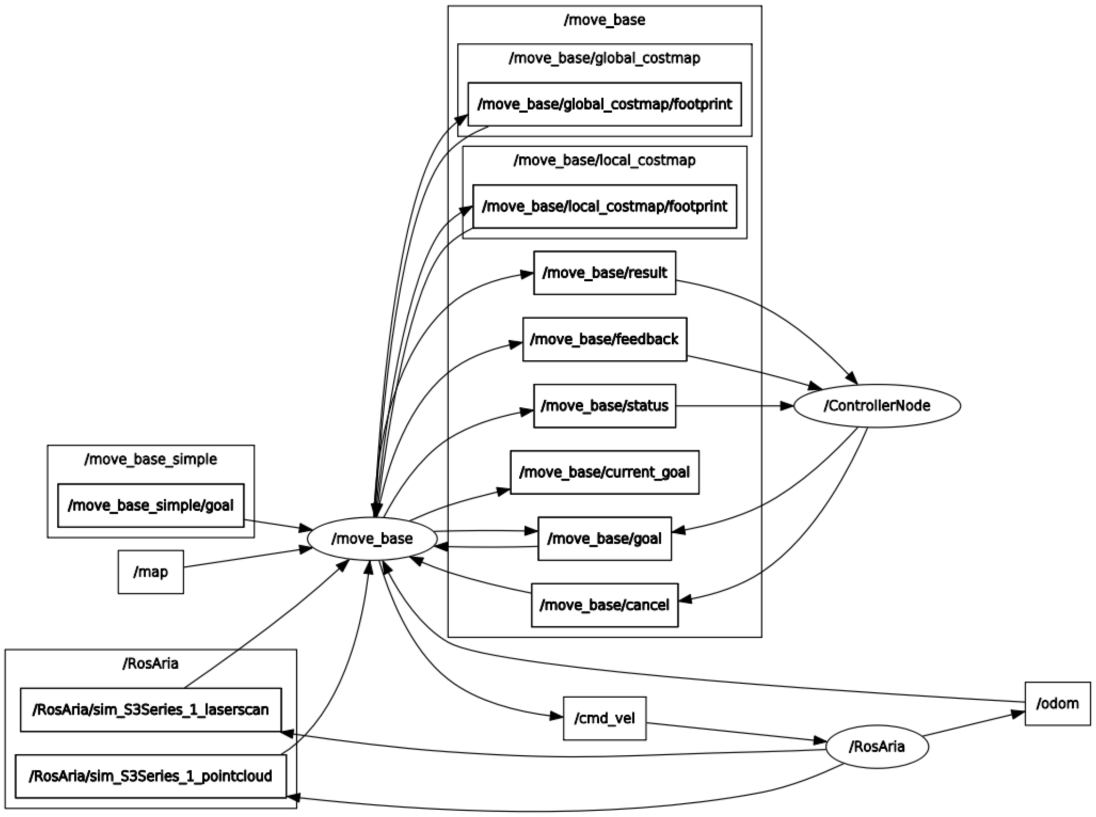


Figure 45: ROS computational graph related to the navigation controller.

10 Discussion

Performing VSLAM using RTAB-Map in section 7 has two important advantages: It provides 2D and 3D maps, as well as being accurate. However, it had problems detecting the edges of the Cyborg Office, compared to using only LiDAR, which does this well. Additionally, it is more computationally demanding compared to a method that only provides 2D maps.

YOLO object detection from section 8 has a significant advantage, compared to other methods of object detection: It is orders of magnitude faster at detecting objects. Additionally, YOLO is accurate and regularly improved for each new version (the newest version as of writing this report is YOLOv4).

However, it does not distinguish between different students. Consequently, if two students are standing in the same room at different locations, YOLO would not distinguish between the two. Consequently, the */person* topic would contain the location of two different students.

The controller design, implemented in section 9 had the advantage of always being able to navigate towards a student as long as two conditions were met: A student was detected and there was a safe maneuverable passage between the Cyborg and the student. No human intervention was needed.

Section 7, 8 and 9 uses the approach of having a desktop computer doing the most computationally demanding tasks (such as object detection or VSLAM), whereas the Jetson Nano and the Cyborg embedded computer ran a small selection of nodes. This approach has significant advantages: It is much cheaper, as most people have laptops or desktop computers more powerful than the embedded computer and Jetson Nano combined. Additionally, it is much faster to run object detection or VSLAM on a remote desktop computer than on hardware the size of the Jetson Nano (without completely blowing up the budget). Previous projects have used the Nvidia Jetson TX2 developer kit, capable of running object detection, but at about ten times the cost of the Jetson Nano.

Proposed future work

The following is a list of proposed future work for students interested working on the Cyborg project in the future:

- **Create a new map of Glassgården**: The RTAB-Map ros wrapper proved to be good at performing VSLAM, even in areas such as the Cyborg office. Therefore, creating a new map of Glassgården using RTAB-Map is recommended.
- **Upgrade the ROS distribution**: The embedded Cyborg computer uses the Kinetic Kame ROS distribution. However, this is no longer supported as it reached its End Of Life (EOL) in 2021. It is therefore recommended to upgrade to a newer distribution, e.g., ROS Noetic Ninjemys, which reaches its EOL in May 2025.

11 Conclusion

In section 6, two URDF models were created: Section 6.1 created a complete model of the current version of the Cyborg and evaluated in section 6.1.1. The second model, presented in section 6.2, included the ZED camera (see section 3) and was evaluated in section 6.2.1. Both models were created successfully and visualized in RViz, displaying the necessary coordinate frames.

The new robot model was later used in section 7, which implemented VSLAM with RTAB-Map in ROS. A framework for adding LiDAR measurements, provided by the Cyborg, was implemented. The VSLAM capabilities were evaluated in section 7.3, which provided a 2D and 3D map of the Cyborg office.

In section 8, object detection with YOLO and Darknet was implemented and evaluated in section 8.1.1. The YOLOv3 provided the best results and managed to detect a student within the Cyborg office successfully.

Along with the new robot model, YOLO was used to develop a navigation controller in section 9 aimed at following a student within the Cyborg office. The implementation was done in two parts: A controller was created and tested on the Cyborg in section 9.2. From this, two coordinate frames were created: *person_detection_frame* and *person*. The former was used as a frame of reference for the detected student, and the latter contained the position of the detected student. The Cyborg was instructed to navigate whenever a student was detected. A constraint was implemented to prevent the student from leaving the cameras FOV: The controller updated the navigation goal whenever the student was more than 20 °of the FOV. The Cyborg managed to follow the student during testing successfully.

The second controller, from section 9.3 was tested in a simulated environment using MobileSim. A velocity regulator was implemented, which adjusted the speed of the Cyborg as a function of the distance between the Cyborg and the student. The Cyborg successfully managed to follow the student along a pre-recorded path.

11.1 State of the Cyborg as of writing this report

The main concern regarding the Cyborg and the Cyborg project is the batteries. There are three different batteries available at NTNU, as well as three separate charging ports. However, for some unknown reason, the battery does not charge most of the time. When it does, it will charge to a certain point (sometimes 11%) and proceed to drain the battery. Fixing the batteries should therefore be the project's main priority. Moreover, the rack rails on the Pioneer LX are currently loose and must be locked to the robot such that the 3D printed casing can slide on and off with ease.

The robot is currently using the embedded computer to connect to the internet via Eduroam, meaning that it has a dynamic IP address. This is inconvenient for remote access to the Cyborg, as the connection will be lost as soon as the Cyborg is connected to another router (which changes the IP address). The router provided to the Cyborg project is a Linksys WRT54GL router, a model soon to be 20 years old. It is recommended to upgrade to a newer router and mount it to the Cyborg to set up a static IP. It is also recommended to invest in a Jetson Nano. This report proved that expensive hardware is not required, as long as a computer is powerful enough to perform any demanding processes remotely. However, it is recommended that the remote computer has a graphics card provided by Nvidia when performing object detection. Alternatively, the Cyborg project could invest in more expensive hardware, such as the Jetson TX2, to perform all (or most) of the necessary processes.

A Dimensions

A.1 Pioneer LX

The height of your payload will affect the center of gravity, covered in the next section.

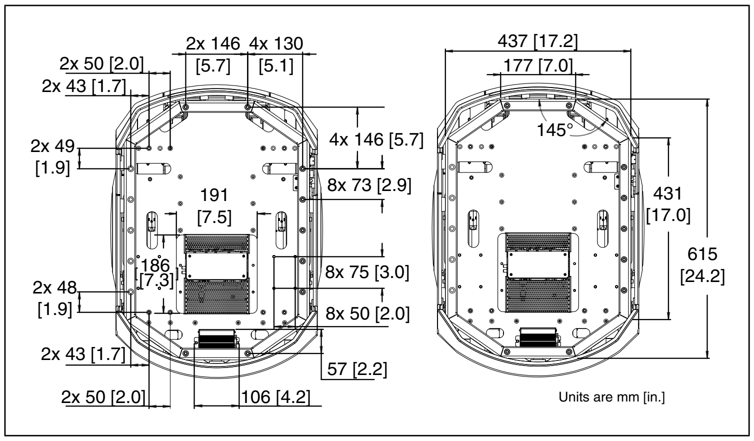
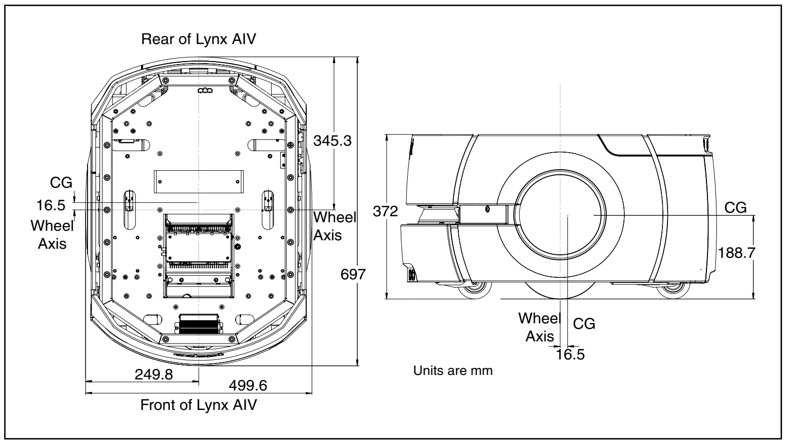


Figure 6-1. Platform Frame Dimensions, for Attaching Payload

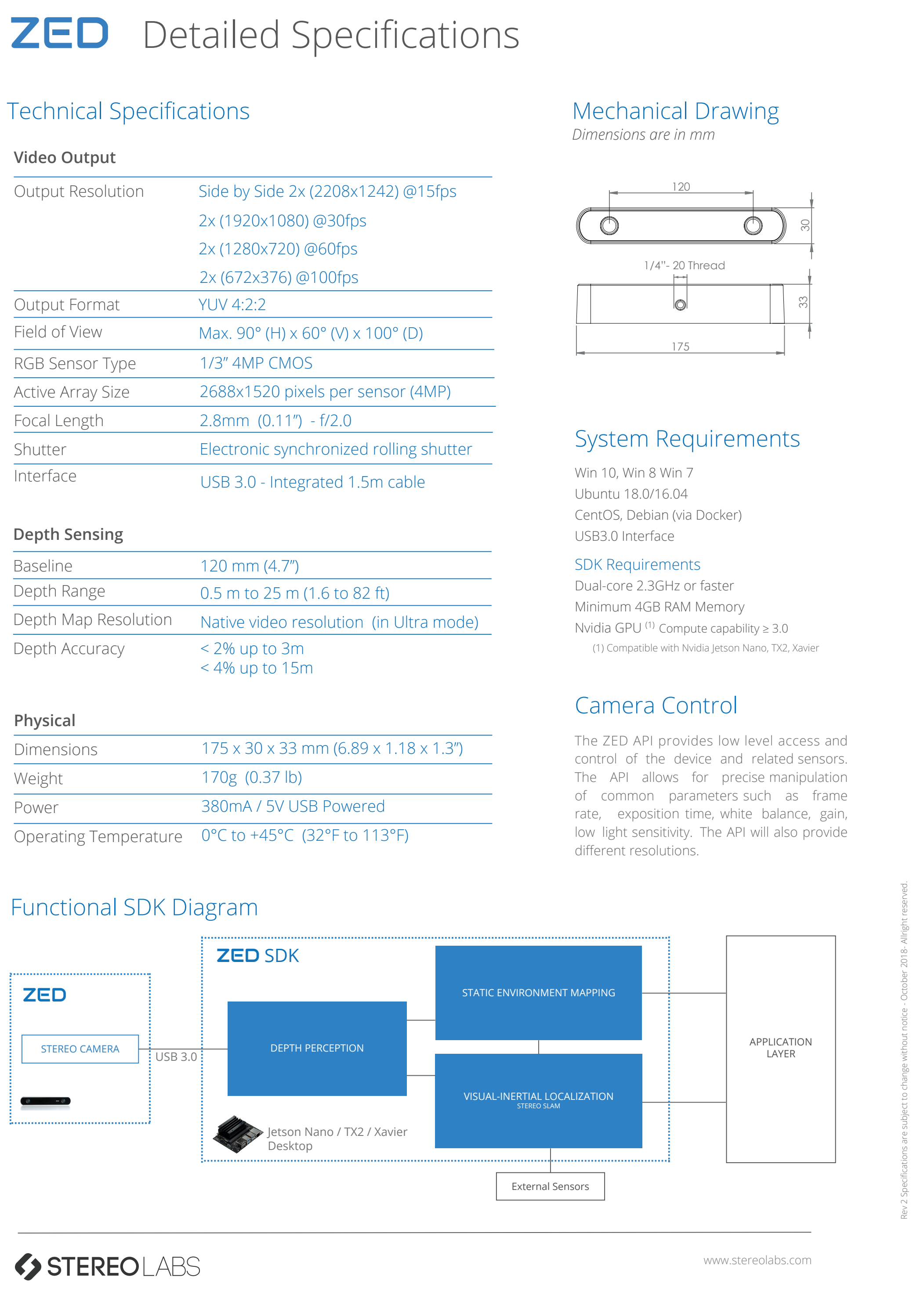
Center of Gravity

As much as possible, you should keep the payload center of gravity centered on the Pioneer LX, and as low (close to the platform top) as possible. This will give you the best stability, particularly when crossing thresholds or irregularities in the floor.

The following figure shows the center of gravity of the platform, without payload.



A.2 ZED Stereo Camera



ZED Detailed Specifications

Technical Specifications

Video Output

Output Resolution	Side by Side 2x (2208x1242) @15fps 2x (1920x1080) @30fps 2x (1280x720) @60fps 2x (672x376) @100fps
Output Format	YUV 4:2:2
Field of View	Max. 90° (H) x 60° (V) x 100° (D)
RGB Sensor Type	1/3" 4MP CMOS
Active Array Size	2688x1520 pixels per sensor (4MP)
Focal Length	2.8mm (0.11") - f/2.0
Shutter	Electronic synchronized rolling shutter
Interface	USB 3.0 - Integrated 1.5m cable

Depth Sensing

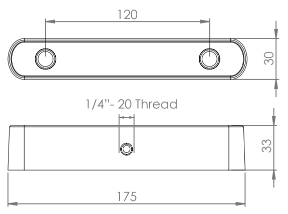
Baseline	120 mm (4.7")
Depth Range	0.5 m to 25 m (1.6 to 82 ft)
Depth Map Resolution	Native video resolution (in Ultra mode)
Depth Accuracy	< 2% up to 3m < 4% up to 15m

Physical

Dimensions	175 x 30 x 33 mm (6.89 x 1.18 x 1.3")
Weight	170g (0.37 lb)
Power	380mA / 5V USB Powered
Operating Temperature	0°C to +45°C (32°F to 113°F)

Mechanical Drawing

Dimensions are in mm



System Requirements

Win 10, Win 8 Win 7
Ubuntu 18.0/16.04
CentOS, Debian (via Docker)
USB3.0 Interface

SDK Requirements

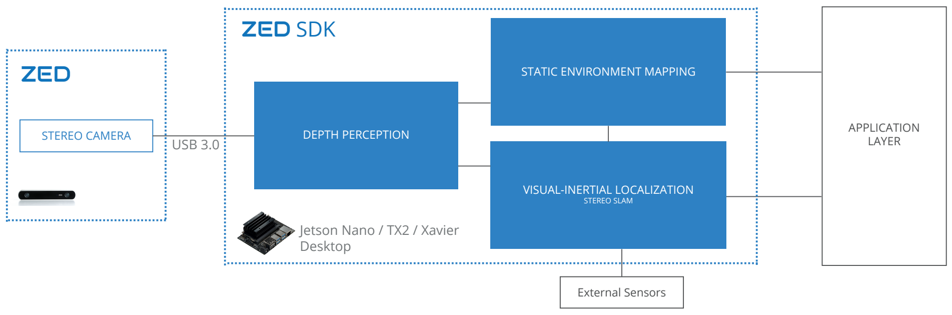
Dual-core 2.3GHz or faster
Minimum 4GB RAM Memory
Nvidia GPU (1) Compute capability ≥ 3.0

(1) Compatible with Nvidia Jetson Nano, TX2, Xavier

Camera Control

The ZED API provides low level access and control of the device and related sensors. The API allows for precise manipulation of common parameters such as frame rate, exposition time, white balance, gain, low light sensitivity. The API will also provide different resolutions.

Functional SDK Diagram



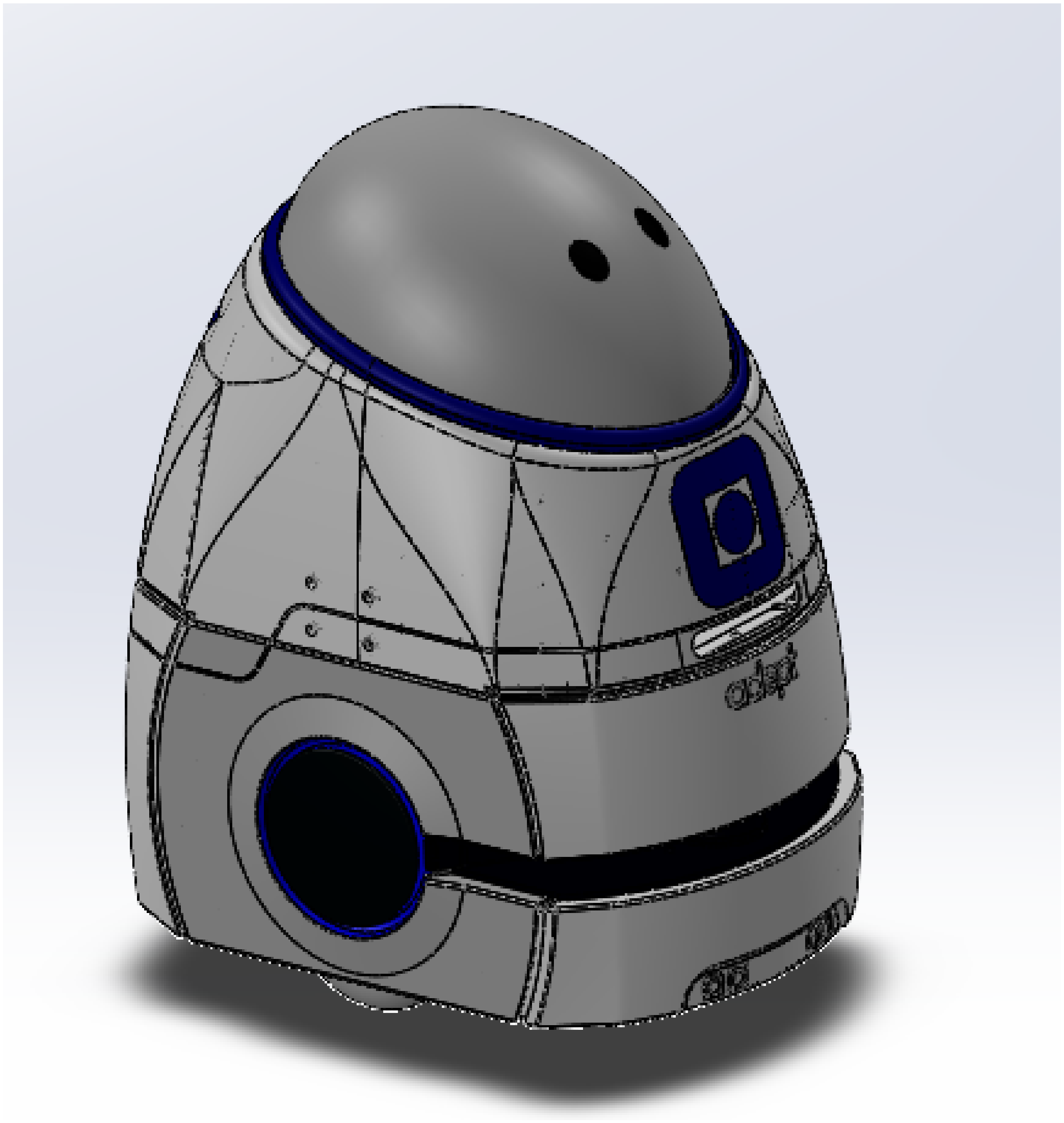
STEREOLABS

www.stereolabs.com

[H]

B 3D CAD Models

B.1 The Cyborg assembly



References

[1] NTNU, “About ntnu cyborg - ntnu,” https://www.ntnu.edu/cyborg/about, (Accessed on 05/15/2021).

[2] A. Ademovic, “An introduction to robot operating system | toptal,” https://www.toptal.com/robotics/introduction-to-robot-operating-system, (Accessed on 05/25/2021).

[3] Wikipedia, “Robot operating system - wikipedia,” https://en.wikipedia.org/wiki/Robot_Operating_System, (Accessed on 05/25/2021).

[4] R. Wiki, “Distributions - ros wiki,” http://wiki.ros.org/Distributions, (Accessed on 05/25/2021).

[5] ROS, “Ros 2 documentation — ros 2 documentation: Foxy documentation,” https://docs.ros.org/en/foxy/, (Accessed on 05/25/2021).

[6] R. Wiki, “Nodes - ros wiki,” http://wiki.ros.org/Nodes, (Accessed on 06/09/2021).

[7] R. wiki, “Packages - ros wiki,” http://wiki.ros.org/Packages, (Accessed on 10/13/2020).

[8] R. Wiki, “catkin/package.xml - ros wiki,” http://wiki.ros.org/catkin/package.xml, (Accessed on 10/15/2020).

[9] http://roswiki.autolabor.com.cn/catkin(2f)CMakeLists(2e)txt.html, “catkin/cmakelists.txt,” http://roswiki.autolabor.com.cn/catkin(2f)CMakeLists(2e)txt.html, (Accessed on 10/15/2020).

[10] R. Wiki, “roscore - ros wiki,” http://wiki.ros.org/roscore, (Accessed on 10/16/2020).

[11] ——, “Master - ros wiki,” http://wiki.ros.org/Master, (Accessed on 10/16/2020).

[12] T. Construct, “What is ros parameter server? - the construct,” https://www.theconstructsim.com/ros-5-mins-012-ros-parameter-server/, (Accessed on 10/16/2020).

[13] R. Wiki, “rosout - ros wiki,” http://wiki.ros.org/rosout, (Accessed on 10/16/2020).

[14] M. A. Eitan Marder-Eppstein, Vijay Pradeep, “actionlib - ros wiki,” http://wiki.ros.org/actionlib, (Accessed on 04/19/2021).

[15] ——, “actionlib/detaileddescription - ros wiki,” http://wiki.ros.org/actionlib/DetailedDescription, (Accessed on 04/19/2021).

[16] T. Foote, “tf: The transform library,” http://wiki.ros.org/Papers/TePRA2013_Foote?action=AttachFile&do=view&target=TePRA2013_Foote.pdf, April 2013, (Accessed on 01/23/2021).

[17] ——, “tf - ros wiki,” http://wiki.ros.org/tf, (Accessed on 01/23/2021).

[18] R. Wiki, “tf: Main page,” http://docs.ros.org/en/kinetic/api/tf/html/c++/, (Accessed on 06/10/2021).

[19] ——, “urdf - ros wiki,” http://wiki.ros.org/urdf/Tutorials/Building%20a%20Visual%20Robot%20Model%20with%20URDF%20from%20Scratch, (Accessed on 01/26/2021).

[20] E. Marder-Eppstein, “navigation - ros wiki,” http://wiki.ros.org/navigation, (Accessed on 04/12/2021).

[21] R. Wiki, “costmap_2d - ros wiki,” http://wiki.ros.org/costmap_2d, (Accessed on 04/12/2021).

[22] Stereolabs, “Zed stereo camera | stereolabs,” https://www.stereolabs.com/zed/, (Accessed on 05/20/2021).

[23] Nvidia, "Nvidia jetson nano developer kit | nvidia developer," https://developer.nvidia.com/embedded/jetson-nano-developer-kit, (Accessed on 06/06/2021).

[24] ——, "Nvidia announces jetson nano: $99 tiny, yet mighty nvidia cuda-x ai computer that runs all ai models | nvidia newsroom," https://nvidianews.nvidia.com/news/nvidia-announces-jetson-nano-99-tiny-yet-mighty-nvidia-cuda-x-ai-computer-that-runs-all-ai-models, (Accessed on 06/06/2021).

[25] Stereolabs, "Api documentation | c++ api reference | stereolabs," https://www.stereolabs.com/docs/api/, (Accessed on 04/13/2021).

[26] M. Labbé and F. Michaud, "Rtab-map | real-time appearance-based mapping," http://introlab.github.io/rtabmap/, 2017, (Accessed on 03/12/2021).

[27] P. Newman and K. Ho, "Slam- loop closing with visually salient features," https://www.robots.ox.ac.uk/~mobile/Papers/NewmanHoICRA05.pdf, (Accessed on 03/12/2021).

[28] Wikipedia, "Occupancy grid mapping - wikipedia," https://en.wikipedia.org/wiki/Occupancy_grid_mapping#cite_note-1, (Accessed on 03/15/2021).

[29] D. Bagnell, "Occupancy maps," https://www.cs.cmu.edu/~16831-f14/notes/F14/16831_lecture06_agiri_dmcconac_kumarsha_nbhakta.pdf, (Accessed on 03/15/2021).

[30] R. Wiki, "rtabmap - ros wiki," http://wiki.ros.org/rtabmap, (Accessed on 03/15/2021).

[31] ——, "rtabmap_ros - ros wiki," http://wiki.ros.org/rtabmap_ros, (Accessed on 03/15/2021).

[32] R. Gandhi, "R-cnn, fast r-cnn, faster r-cnn, yolo — object detection algorithms," https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e, July 2018, (Accessed on 03/24/2021).

[33] P. Bhavsar, "Object detection with convolutional neural networks | by panth bhavsar | datadriveninvestor," https://medium.datadriveninvestor.com/object-detection-with-convolutional-neural-networks-dde190eb7180, Jan 2019, (Accessed on 03/24/2021).

[34] R. Gandhi, "Support vector machine — introduction to machine learning algorithms," https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47, June 2018, (Accessed on 03/24/2021).

[35] G. Karimi, "Introduction to yolo algorithm for object detection | engineering education (enged) program | section," https://www.section.io/engineering-education/introduction-to-yolo-algorithm-for-object-detection/, April 2021, (Accessed on 06/05/2021).

[36] J. Świeżewski, "Yolo algorithm and yolo object detection: An introduction - appsilon," https://appsilon.com/object-detection-yolo-algorithm/, May 2020, (Accessed on 06/05/2021).

[37] Stereolabs, "Zed sdk 3.4 - download | stereolabs," https://www.stereolabs.com/developers/release/3.4/, (Accessed on 05/16/2021).

[38] Nvidia, "Cuda toolkit 11.3 downloads | nvidia developer," https://developer.nvidia.com/cuda-downloads, (Accessed on 05/16/2021).

[39] Stereolabs, “Github - stereolabs/zed-yolo: 3d object detection using yolo and the zed in python and c++,” https://github.com/stereolabs/zed-yolo, (Accessed on 03/25/2021).

[40] R. WIKI, “message_generation - ros wiki,” http://wiki.ros.org/message_generation, (Accessed on 05/10/2021).

[41] R. Wiki, “roscpp - ros wiki,” http://wiki.ros.org/roscpp, (Accessed on 05/10/2021).

[42] ——, “dynamic_reconfigure - ros wiki,” http://wiki.ros.org/dynamic_reconfigure, (Accessed on 05/08/2021).