Exploration of 2D/Visual SLAM and Autonomous Navigation with Parallax Eddie Robot

Zahra Niazi

Contents

1. Eddie final setup and bringup

1.1 Getting to know Eddie

Eddie is a versatile mobile robotics development platform. Eddie is equipped with 3D vision capabilities, utilizing the Microsoft Kinect, allowing it to perceive its surroundings in three dimensions. The platform is versatile, supporting applications like autonomous navigation, machine vision system development, tele-presence robots, personal assistant robots, security and surveillance robots, and crowd interaction and advertising.

1.1.1 Hardware Subsystems

Eddie is a differential drive robot featuring a microcontroller with an integrated PID, two optical wheel encoders, two PWM driven motors, and two lead-acid batteries. The robot's structure comprises a pair of top and bottom decks, along with an undercarriage designed to hold the batteries securely.

Figure 1: Eddie Platform

Range Sensors. On its base, there are five distance sensors placed for collision avoidance. This sensor array comprises three infrared (IR) sensors and two ultrasonic sensors.

Wheel Encoders. The 36-position Quadrature Encoders offer a reliable method to track the position and speed of each wheel continuously. When used with the included Encoder disks, each Encoder assembly provides a basic resolution of 36 positions per rotation. However, since there are two sensors in each assembly, the potential resolution increases to 144 positions per full tire rotation. This equates to a linear resolution of approximately 0.13 inches of travel per drive assembly when using the standard six-inch tires. It's important to note that the actual resolution may be limited by the amount of backlash in the gear motor assemblies, which is approximately +/- 0.175 linear inches. The Position Controllers are compatible with any microcontroller via a single-wire half-duplex asynchronous serial communication (UART) bus.

Board. The Eddie Control Board provides a complete single-board solution to control the Eddie Robot Platform. It utilizes the powerful Propeller P8X32A microcontroller, boasting eight 32-bit cores for impressive performance and versatility. The board incorporates high-current motor drivers, an eight-channel 10-bit ADC, and ample digital I/O options. It communicates via a USB mini-B connector, which enumerates as a serial COM port.

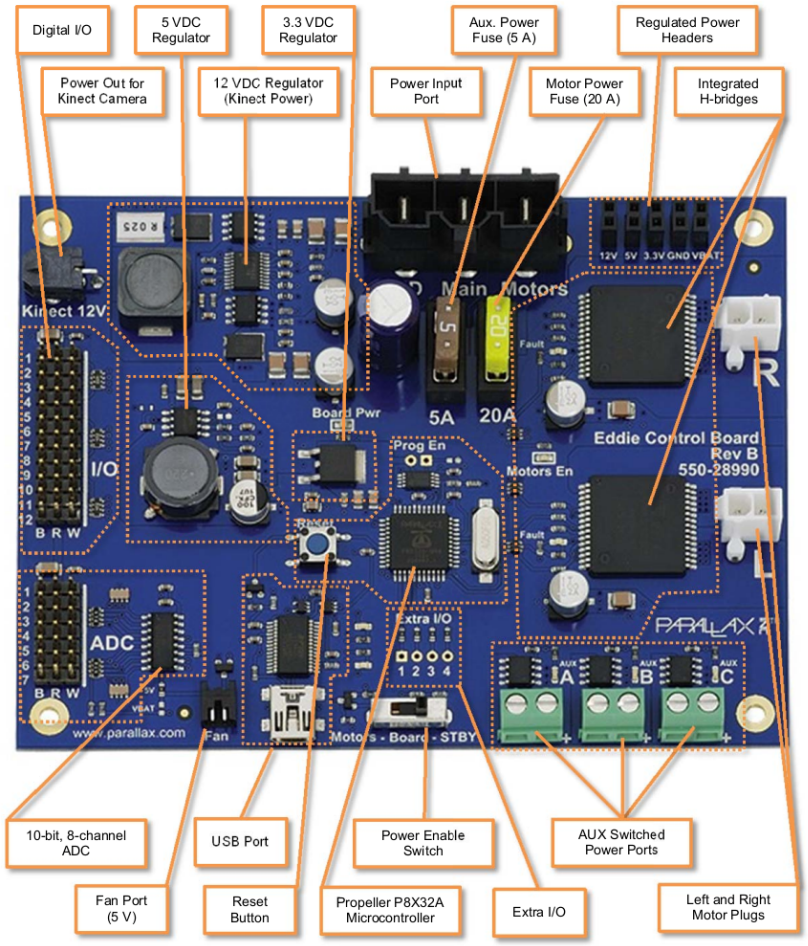

Figure 2: Board Overview

Most of the Propeller's 32 general purpose I/O pins are used to interface with on-board peripherals, or connect to headers for sensor interfacing. The Digital I/O pins are brought out to twelve 3-pin headers near the edge of the board. The order of the pins from left to right is: Ground, 5V, Signal.

- The 1st and 2nd pins are connected to our two ultrasonic sensors.

For the analog to digital converter, the board is equipped with the Microchip MCP3008 which is an 8-channel, 10-bit ADC.

- The left-most IR Sensor cable is connected to "AD1", the center IR sensor to "AD2", and the right-most IR Sensor to "AD3".

The three-pin cable from the right Position Controller is connected to Pin-set 11 in the "Encoders" section of the Control Board.

Camera. The Kinect for Xbox 360 is a combination of software and hardware, featuring a range chipset technology by PrimeSense, which uses an infrared projector and camera to determine the location of nearby objects in 3D.
The Kinect sensor consists of an RGB camera, depth sensor, and microphone array, allowing for full-body 3D motion capture, facial recognition, and voice recognition capabilities. The depth sensor employs an infrared laser projector and monochrome CMOS sensor to capture 3D video data under any lighting conditions, with an adjustable sensing range and automatic calibration for different environments.

The software technology of Kinect enables advanced gesture recognition, facial recognition, and voice recognition, making it capable of simultaneously tracking up to six people. The Kinect's various sensors output video at frame rates ranging from approximately 9 Hz to 30 Hz, with different resolutions and color formats. The sensor has a practical ranging limit of 1.2–3.5 meters and a field of view of 57° horizontally and 43° vertically, providing a high resolution of over 1.3 mm per pixel. To power the motorized tilt mechanism of the Kinect sensor, a proprietary connector combining USB communication and additional power is used, while older models require a special power supply cable that splits the connection into separate USB and power connections.

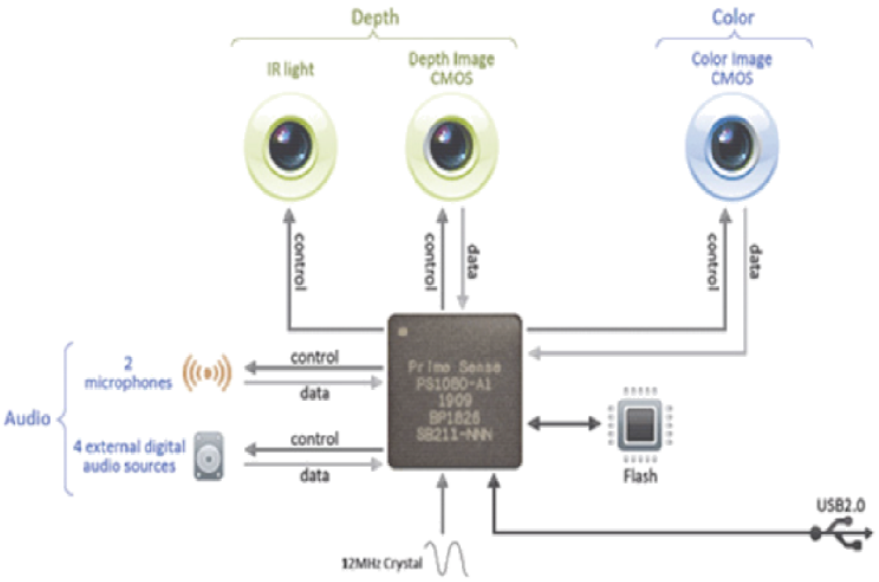

Figure 3: Kinect Components

1.1.2 Control Board Firmware Command Set

The Eddie Control Board is a complete robot controller and sensor-interface solution. Parallax's ready-to-go Eddie Control Board firmware, designed for the Eddie Robot Platform provides an easy-to-use serial command interface to control and manage all of the on-board pe-

ripheral electronics such as motor drivers, digital I/O, and analog to digital converter (ADC) channels.
The Eddie Control Board communicates over USB; and when connected to a PC, the board enumerates as a serial COM port. Configure the COM port to use these settings of 115.2 kBaud, 8-bit character size, 1 stop bit and no parity.
All commands adhere to the same general format which is shown below:
Input: <cmd>[<WS><param1>...<WS><paramN>]<CR>
Response (Success): [<param1>...<WS><paramN>]<CR>
Response (Failure): ERROR[<SP>-<SP><verbose_reason>]<CR>

In this format:

- <cmd> represents the command mnemonic.
- <param1>...<paramN> are optional parameters required by the command or mode. Numbers are always entered as hexadecimal values, and signed values use two's complement.
- <WS> refers to one or more whitespace characters, which can be spaces (ASCII 32) or tabs (ASCII 9).
- <CR> is a single carriage-return character (ASCII 13).
- <SP> is a single space character (ASCII 32).
- <verbose_reason> is an optional error message displayed when verbose mode is enabled (using the VERB command).

Allowed characters are in the ASCII range from 32 to 126, except for carriage return (ASCII 13) and tab (ASCII 9). Commands can have up to 254 characters, including the terminating carriage return character. Anything beyond this limit is ignored as an invalid command. The command handler only processes and responds after receiving the carriage return character.

Table 1: Eddie Command Set Description: *Interface*

Cmd	**Input Params**	**Return Params**	**Values**	**Description**
HWVER		***<version>***	***version***=0..FFFF	Get hardware version
VER		***<version>***	***version***=0..FFFF	Get firmware version
VERB	***<mode>***		***mode***= 0(off), 1(on)	Set verbose mode
WATCH	***<mode>***		***mode***= 0(off), 1(on)	Set watch mode
BLINK	***<pin><rate>***		***pin***=0.. 1F ***rate***=0..FFFF	Toggle pin at a specified rate in increments of 0.1Hz

Table 2: Eddie Command Set Description: *I/O Control*

Cmd	Input Params	Return Params	Values	Description
IN	***<bitmask>***		***bitmask***=0..7FFFF	Set GPIO pins in bitmask to inputs
OUT	***<bitmask>***		***bitmask***=0..7FFFF	Set GPIO pins in bitmask to outputs
LOW	***<bitmask>***		***bitmask***=0..7FFFF	Set GPIO pins in bitmask to low (only applies to output pins)
HIGH	***<bitmask>***		***bitmask***=0..7FFFF	Set GPIO pins in bitmask to high (only applies to output pins)
INS		***<bitmask>***	***bitmask***=0..7FFFF	Get GPIO pins currently set as inputs
OUTS		***<bitmask>***	***bitmask***=0..7FFFF	Get GPIO pins currently set as outputs
LOWS		***<bitmask>***	***bitmask***=0..7FFFF	Get GPIO pins currently set as low
HIGHS		***<bitmask>***	***bitmask***=0..7FFFF	Get GPIO pins currently set as high
READ		***<bitmask>***	***bitmask***=0..7FFFF	Get current state (high/low) of all GPIO pins

Table 3: Eddie Command Set Description: *Sensor Interfacing*

Cmd	Input Params	Return Params	Values	Description
SPNG	***<bitmask>***		***bitmask***=0..FFFF	Set pins in bitmask to act as GPIO pins
SGP	***<bitmask>***		***bitmask***=0..7FFFF	Set pins in bitmask to act as GPIO pins
PING		***<value1>***[***<value2>*** ...***<valueN>***]	***value***=0,12..B54	Get PING))) sensor sonar measurements (one 12-bit value per sensor)
ADC		***<value1>***...***<value8>***	***value***=0..FFF	Get all ADC values (12-bit values)

Table 4: Eddie Command Set Description: *Motor Control*

Cmd	Input Params	Return Params	Values	Description
GO	***<left><right>***		***left/right***=80..7F	Set motor power (signed byte)
GOSPD	***<left><right>***		***left/right***=8000..7FFF	Set motor speed (signed word)
STOP	***<dist>***		***dist***=0..FFFF	Slow to a stop over specified distance
TRVL	***<dist><speed>***		***dist***=8000..7FFF ***speed***=1..7F or 1..FF	Travel a specified distance in a straight line, ramping up to a maximum specified speed
TURN	***<angle><speed>***		***dist***=8000..7FFF ***speed***=1..7F or 1..FF	Rotate in place by a specified angle, ramping up to a maximum specified speed
STOP	***<rate>***		***rate***=1..7F or 1..FF	Set rate of acceleration/deceleration
SPD		***<left><right>***	***left/right***=8000..7FFF	Get the current average speed (positions per second) for both wheels
HEAD		***<angle>***	***angle***=0..168 (decimal 0..359)	Get the current heading (in degrees) relative to start
DIST		***<left><right>***	***left/right***=80000000..7FFFFFFF	Get the current average speed (positions per second) for both wheels
RST				Reset the distance and heading values to 0

1.1.3 Software Subsystems

Robot Communication: `eddiebot_bringup`
The communication with the board involves establishing a connection between the robot control board and our ROS2 programs, enabling seamless data exchange. This software acts as a pathway that conveys and receives data between two of our major hardware components. Specifically, this sub-system serves as the means to convey the desired velocity information from the control system to be executed by the robot's firmware board.

Description: `eddiebot_description`
This package contains files and data used to describe the structure of eddie. It includes URDF files to define the physical properties and kinematic structure of eddie. This package enables accurate simulation of Eddie's structure and movement within the ROS environment. This is crucial for visualizing and controlling the robot in a virtual space before deploying it in the physical world.
The `diff_drive` plugin is a component added to the URDF files to simulate the differential drive mechanism that eddie uses.
The `joint_state_publisher` plugin is another component added to the URDF files. It is used to simulate the movement of Eddie's wheels. `joint_state_publisher` helps in updating the positions and orientations of robot joints based on the data received from the robot's actuators.

In the launch file we run two nodes:

- `robot_state_publisher`: The `robot_state_publisher` reads the URDF description file and generates static transformations (tf static transforms). These transformations define the spatial relationship between different components of the robot, such as its base, wheels and different sensors.
- `joint_state_publisher`: The `joint_state_publisher` node sends dynamic transformations from the wheels to the `robot_state_publisher`. These dynamic transformations update the joint positions based on the movement of the wheels, allowing the `robot_state_publisher` to accurately track and represent the robot's current state.

Simulation: `eddiebot_gazebo`

The `eddiebot_gazebo` package enables us to test and experiment with Eddie's behavior in the Gazebo simulation environment.
When we run the main launch file in this package, it sets up the Gazebo simulator, creating a simulated environment where Eddie can exist and interact. Once Gazebo is up, we also spin a node to "spawn" Eddie inside the simulated world. Meaning we place the robot model, represented by a URDF file, into the Gazebo environment. We do this by launching the `eddiebot_description` launch file mentioned before.
This step essentially brings Eddie to life in the simulation, and we can now control and observe its actions as if it were a real physical robot. To ensure communication between Gazebo and the ROS2 ecosystem, the `eddiebot_gazebo` package establishes a connection between their respective topics.

Custom Interfaces: `eddiebot_msgs`

In the `eddiebot_msgs` package, we define the custom interfaces for Eddie, the robot. This package allows us to define the specific topics and services through which we can communicate information about Eddie's current state and control its movements.
Firstly, we define the topics where information about the robot's current state will be published. This includes data such as its current speed, angle, and sensor inputs. By publishing this information on specific topics, other nodes in the ROS2 ecosystem can subscribe to these topics and receive real-time updates on Eddie's state. This enables different components of the system to access and utilize this information for various purposes, such as perceiving the environment, making decisions or performing actions based on Eddie's current state.
Additionally, we define services that can be used to communicate with the robot and instruct it to move or turn. Services in ROS2 provide a request-response mechanism, allowing nodes to send requests to specific services and receive corresponding responses. For Eddie, these services can be used to send commands to the robot, such as specifying a desired movement or rotation. The robot can then receive these commands, process them, and respond accordingly, enabling control over Eddie's actions through the defined services.
By defining these custom interfaces in the `eddiebot_msgs` package, we establish a standardized and consistent way to exchange information and control commands between different

components of the system, facilitating effective communication and interaction with Eddie.

Odometry: `eddiebot_odom`

The `eddiebot_odom` package plays a crucial role in enabling Eddie to understand its position and movement within the environment. It is responsible for publishing odometry information, which is an estimation of Eddie's position and orientation based on the motion of its wheels. Odometry is essential for mapping and localization tasks in robotics.

Since Eddie doesn't have an IMU (Inertial Measurement Unit), which could provide direct information about its orientation and acceleration, the package relies on wheel encoders. Wheel encoders are sensors mounted on the robot's wheels that measure the rotation of the wheels. By analyzing the data from these encoders, the `eddiebot_odom` node can calculate how far each wheel has turned and, consequently, estimate Eddie's overall movement.

The process starts with the `eddiebot_odom` node subscribing to the topic where the wheel encoder data is published. As the wheels move, the encoders send real-time information to the node about the rotations. Using this information, the node calculates the incremental movement of Eddie in terms of distance and angular change.

To keep track of Eddie's pose (position and orientation) over time, the node continuously updates the transformation between two frames: the `"/odom"` frame and the `"/base_footprint"` frame.

- The `"/odom"` frame represents Eddie's initial position in the world, usually set at the robot's starting point. The coordinate frame called odom is a world-fixed frame. The pose of a mobile platform in the odom frame can drift over time, without any bounds. This drift makes the odom frame useless as a long-term global reference. However, the pose of a robot in the odom frame is guaranteed to be continuous, meaning that the pose of a mobile platform in the odom frame always evolves in a smooth way, without discrete jumps. The odom frame is useful as an accurate, short-term local reference, but drift makes it a poor frame for long-term reference.

- The `"/base_footprint"` frame is attached to Eddie's base chassis. The coordinate frame called base_link is rigidly attached to the mobile robot base. The base_link can be attached to the base in any arbitrary position or orientation; for every hardware platform there will be a different place on the base that provides an obvious point of reference.

By knowing how far Eddie has moved from its initial position and the angular changes during its motion, the node can estimate the transformation between these frames. This allows Eddie to understand its current pose relative to where it started, effectively providing odometry information.

Having accurate odometry enables Eddie to navigate its environment more effectively. It can use this information to create a map of the environment. However, it's worth noting that odometry estimates may drift over time due to inaccuracies and wheel slippage, especially in complex environments. To improve localization accuracy, other sensor modalities like visual odometry or an IMU can be used in conjunction with wheel encoders.

Visualization: `eddiebot_rviz`

The `eddiebot_rviz` package is responsible for visualizing Eddie's current state and movement in a visualization environment like RViz.

We run the `eddiebot_description` launch file within the `eddiebot_rviz` package, which loads the robot's URDF model into RViz. This URDF model includes information about Eddie's physical structure, joints, sensors, and other components, allowing RViz to create a visual representation of the robot accurately.

The `eddiebot_rviz` package subscribes to various topics that publish real-time data about Eddie's state and movements. It subscribes to topics that provide information about joint angles and sensor data. With access to the subscribed topics, the `eddiebot_rviz` package can update the visualization of Eddie's state in real-time within the RViz environment. As the robot moves, the URDF model's joints and links are updated according to the received data, enabling you to see Eddie's current pose and joint angles dynamically.

The real-time visualization in RViz allows us to interactively analyze Eddie's behavior. We can observe how the robot responds to different commands and paths, assess its trajectory, and verify whether its movements align with the intended behavior. In addition to joint angles and pose, the `eddiebot_rviz` package can also display other sensor data within RViz. For example, it can visualize data from range sensors, cameras, or any other sensors equipped on the robot. This feature is valuable for understanding how the robot perceives its environment and how it responds to various stimuli.

Velocity: `eddiebot_vel_controller`

The `eddiebot_vel_controller` package subscribes to the `"/cmd_vel"` topic, where commands to move or rotate Eddie are published. These commands are in the Twist format which consists of linear and angular velocity components that define how Eddie should move in the environment.

Upon receiving the Twist messages from the `"/cmd_vel"` topic, the `eddiebot_vel_controller` extracts the linear and angular velocity values from the Twist message and converts them into simple velocity commands. Then it publishes these commands on the `"eddie/simple_velocity"` topic.

SLAM and Navigation: `eddiebot_nav`

The `eddiebot_nav` package is the central location for managing all the launch files configuring the navigation stack and SLAM tools used in the Eddie robot's autonomous navigation capabilities. Eddie uses the `slam_toolbox` for 2D SLAM. SLAM is a critical process that allows the robot to create a map of its environment while simultaneously determining its own position within that map. The `slam_toolbox` is a powerful library that performs SLAM algorithms to achieve accurate mapping and localization using sensor data from onboard sensors like LIDAR and odometry. Because Eddie is not equipped with a LIDAR, we launch the `"depthimage_to_laserscan"` package to extract laser scans.

In addition to 2D SLAM, the `eddiebot_nav` package also explores vSLAM using RTAB-Map. RTAB-Map is a popular vSLAM library that enables the robot to construct 3D maps of the environment using both visual and depth information from RGB-D cameras. This advanced vSLAM technique enhances the accuracy and richness of the mapping process, enabling Eddie

to navigate more efficiently in complex environments. For autonomous navigation, Eddie utilizes the Nav2 framework. The navigation stack is a collection of algorithms and components responsible for planning and executing the robot's path from its current location to the target destination. In this launch file we also run the static transformations from the robot's frame to the RGB and depth frames.

1.2 Setting up the Kinect

The `kinect_ros2` package is a component that allows the Eddie robot to interface with a Microsoft Kinect sensor and utilize its RGB and depth images for perception tasks in ROS2. This package spins a node responsible for receiving the RGB and depth images from the Kinect sensor and storing them in memory. Then, at regular intervals, it publishes the data from these images on specific ROS2 topics. The topics it publishes are:

- `"image_raw"`: This topic contains the raw RGB image data captured by the Kinect sensor. It provides a continuous stream of color images representing what the camera perceives from its viewpoint.
- `"camera_info"`: This topic carries calibration and intrinsic parameters of the camera. It includes information about the camera's focal length, optical centers, and distortion coefficients. This data is essential for performing accurate transformations and geometric calculations in image processing.
- `"depth/image_raw"`: Here, the package publishes the raw depth image data captured by the Kinect sensor. The depth image provides information about the distance of objects from the camera. It is represented as a 2D array of depth values corresponding to each pixel in the RGB image.
- `"depth/camera_info"`: Similar to the `"camera_info"` topic, this topic contains calibration and intrinsic parameters specific to the depth camera. These parameters enable accurate depth mapping and 3D reconstruction from the depth image.

By publishing these topics, the `kinect_ros2` package enables other nodes in the ROS2 environment to subscribe to and access the Kinect sensor's data for various perception and navigation tasks.

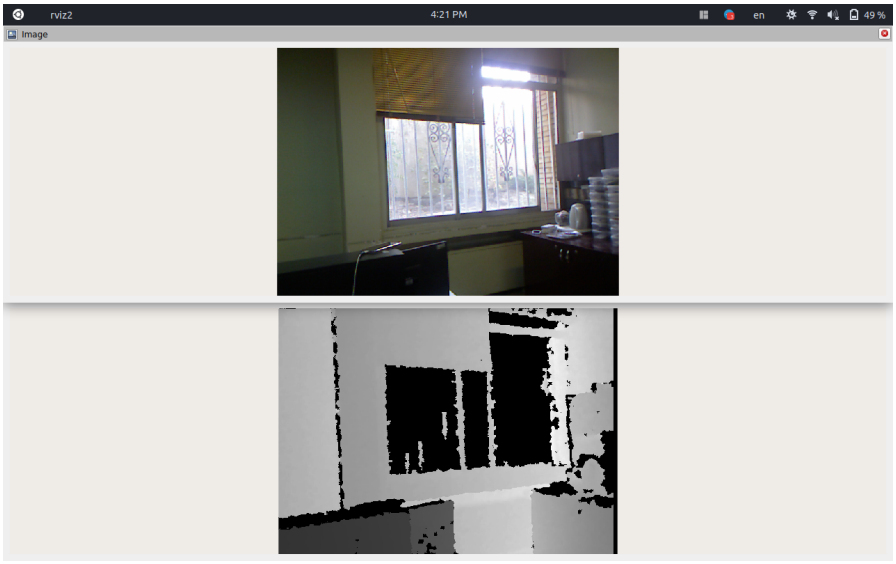

Figure 4: Testing kinect

One of the issues we encountered was the presence of out-of-sync timestamps between the `"camera_info"` and `"depth/camera_info"` topics that were being published. These two topics provide important calibration and intrinsic parameters for the RGB camera and the depth camera, respectively. Having these parameters synchronized correctly is crucial for accurate perception and 3D reconstruction. When performing coordinate transformations between RGB and depth images or other perception tasks, using unsynchronized timestamps can lead to misalignments and inaccurate results. This affects the quality of visual data and hinders the performance of perception algorithms. Also in applications like mapping and navigation, incorrect timestamps can introduce errors in robot localization and obstacle avoidance. These errors can potentially lead to collisions or inefficient path planning. Accurate timestamps are crucial for correctly associating corresponding data from different sensors or perception modules. With out-of-sync timestamps, the associations can become incorrect, leading to flawed data interpretations.
To address this issue, we fixed the out-of-sync timestamps. We ensured that both topics publish their data with matching timestamps, ensuring that the calibration and intrinsic parameters correspond accurately to the corresponding RGB and depth images. By resolving the timestamp synchronization problem, we improved the quality and reliability of our perception and mapping processes. It allowed our robot to better perceive and interpret the environment, leading to more accurate navigation and decision-making capabilities. As a result, the overall performance and robustness of our robot's perception system were greatly enhanced, allowing it to operate more effectively and efficiently in its environment.

1.3 Bringup

After successfully setting up the Kinect package, we proceeded to bring up Eddie and test his movement. The steps we followed are as follows:

1. Granting USB permissions: We connected the USB cable between Eddie and our laptop and executed the command "`sudo chmod a+rw /dev/ttyUSB0`" to give the necessary permissions for communication with Eddie's control board.

2. Bringing up Eddie: To initiate the robot's functionality, we ran the command "`ros2 launch eddiebot_bringup eddie.launch.yaml`" This command launched the required ROS2 nodes responsible for communicating with Eddie's control board and enabled communication within the ROS2 environment.

3. Launching Navigation Stack: To enable navigation capabilities, we executed the command "`ros2 launch eddiebot_nav eddiebot.launch.py`" This step published static transforms specified in the eddiebot_odom package and established transformations between the chassis and the base link of our sensors.

4. Teleoperating Eddie: Finally, to control Eddie's movement, we ran the command "`ros2 run teleop_twist_keyboard teleop_twist_keyboard`" This command allowed us to teleoperate Eddie by sending velocity commands to the robot, thus enabling movement in the desired direction.

By following these steps, we were able to successfully bring up Eddie, utilize the Kinect package for sensor data, and teleoperate the robot for movement using ROS2 commands. We could also bring up RViz to get a better sense of how Eddie is being represented in the ROS2 environment. You can see our attempt at teleoperating Eddie here.

1.4 Networking

To enable teleoperation of Eddie using two laptops, we need to ensure that both laptops are connected to the same network. This allows them to communicate with each other. In ROS2, the default middleware for communication is DDS (Data Distribution Service). DDS utilizes the concept of Domain ID to enable different logical networks to share a physical network.
In ROS2, nodes within the same domain can discover and exchange messages with each other, while nodes in different domains cannot. By default, all ROS2 nodes use domain ID 0. The domain ID is used by DDS to calculate the UDP ports for discovery and communication. When a node starts, it broadcasts its presence to other nodes on the network within the same ROS domain, which is determined by the `ROS_DOMAIN_ID` environment variable. Nodes respond with their information, allowing the necessary connections to be established for communication between nodes.
By connecting the laptops in this manner, we can proceed with teleoperating Eddie. We can bring up rviz on the remote laptop, and configure it to display the RGB image from the Kinect sensor. This enables visualization of the camera feed on the remote laptop, providing a convenient way to monitor Eddie's environment during teleoperation.

2. Odometry

2.1 Motor Control

As each wheel spins the sensor will gather data about the angular position change of the wheel. Once the encoder input has been captured it must be converted to linear velocity and used by a robotic kinematic model to create an estimate of the distance traveled and possible error. Two wheel encoders are used to deduce the speed and travel direction by measuring the number of pulses registered from each wheel. Each wheel encoder has two sensors and is capable of registering a distance resolution of 1/36th of the robot's wheel circumference. The Position Controllers on Eddie use a quadrature encoder system to reliably track the position and speed of each wheel at all times. With the included plastic encoder disks, each Position Controller has a resolution of 36 positions per rotation; this equates to approximately 0.5 inches of linear travel per position using the included 6 inch tires. The Position Controllers calculate and report position and average speed data on command. Knowing that the sensor has 1/36th resolution of wheel circumference, the sensor produces 36 pulses for every complete revolution of the wheel. Based on this, the distance traveled in the duration of one pulse is given below:

$$d = \frac{2\pi r}{36} \tag{1}$$

```
// COUNTS_PER_REVOLUTION   36

// WHEEL_RADIUS    0.1524   Wheel radius in meters
// // the distance of a wheel move forward when encoder increased by 1
// DISTANCE_PER_COUNT      ((TWOPI * WHEEL_RADIUS) /
    COUNTS_PER_REVOLUTION)

// WHEEL_SEPARATION      0.3    two wheels center-to-center distance


// Called from the velocity callback.
// Set the values for the left and right wheels' speeds
// so that Eddie can do arcs
void EddieController::moveLinearAngular(float linear, float angular) {

  // Calculate wheel velocities and convert meter per second to position
    per second
  double left_command_speed  = ((linear - angular * WHEEL_SEPARATION /
    2.0) / WHEEL_RADIUS) / DISTANCE_PER_COUNT;
  double right_command_speed = ((linear + angular * WHEEL_SEPARATION /
    2.0) / WHEEL_RADIUS) / DISTANCE_PER_COUNT;

  sem_wait(&mutex_interrupt_);
  left_drive_speed  = left_command_speed;
  right_drive_speed = right_command_speed;
  // this is not a pure rotation
  rotate_ = false;
  process_ = true;
  // cancel other moving commands in favor of this new one.
```

```
26   interrupt_ = true;
27   sem_post(&mutex_interrupt_);
28 }
```

Listing 1: eddie_controller.cpp

2.2 Wheel Odometry Model

With the rotation data, alongside information on the encoder, such as the radius or circumference, we can estimate the distance traveled by the wheel. Since each slit represents some angle of rotation, knowing the number of slits passed informs us about the amount of rotation between time steps. For an optical encoder, where all the slits are equally spaced, we could get the total angle of rotation between time steps by multiplying the number of slits passed by the amount of rotation represented by a single slit. After we determine the angle of rotation, we can multiply it by the circumference of the encoder to get the distance traveled by the wheel. The goal of an odometry model is to estimate the position and orientation of the robot. To achieve this, we'll leverage data from the rotary encoders, the dimensions of our robot, and geometry. The encoder will inform us of the distance traveled by each wheel at each time step. In terms of using the dimensions of our robot, the only dimension we need is the distance of the point from the left and right wheels. Since we defined our reference point to be located equidistant between the two wheels, we only need to keep track of one number.

Now let's define some variables to keep track of these ideas:

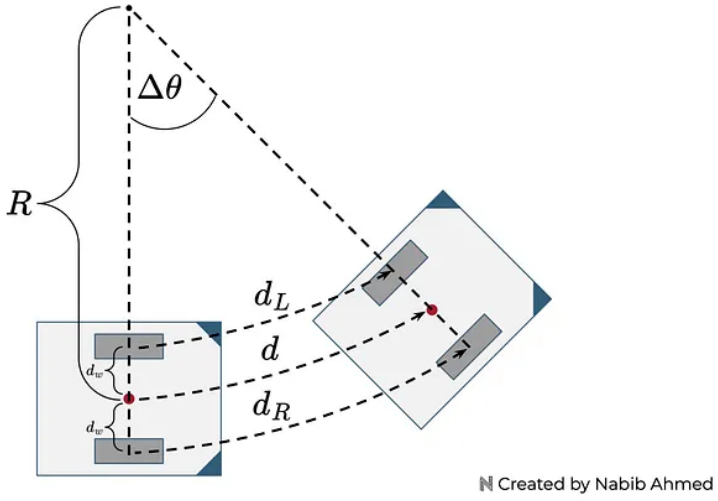

Figure 5

d_L = distance traveled by the left wheel.
d_R = distance traveled by the right wheel.
d_w = distance between the reference point and the wheels.

d = distance traveled by the reference point.
$\Delta\theta$ = A change in the angle of rotation.
R = radius of the curve containing the reference point.

d_L and d_R correspond to the distance traveled by the wheel at a certain time step. This information will come from our rotary encoder. d_w can be derived by measuring the distance between the two wheels and dividing it in half since the point is equidistant from the two wheels. The last three variables are not directly measurable — instead, we need to use geometry to relate these variables to the measurable quantities.

We can start by using the arc length formula. The path for the left wheel, right wheel, and reference points are arcs. They all share the same angle and the radius for each can be expressed in terms of the radius of the curve containing the reference point and the distance between the reference point and the wheels.

$$d = R\Delta\theta \tag{2}$$

$$d_L = (R - d_w)\Delta\theta \tag{3}$$

$$d_R = (R + d_w)\Delta\theta \tag{4}$$

Now we solve for the change in the angle of rotation in terms of measurable quantities, getting the following relationship:

$$\Delta\theta = \frac{d_R - d_L}{2d_w} \tag{5}$$

Now we solve for the radius of the curve containing the reference point by rearranging equations and plugging in what we know. Then, we solve for the distance travelled by the reference point.

$$R = d_L \frac{2d_w}{d_R - d_L} + d_w \tag{6}$$

$$d = \frac{d_R + d_L}{2} \tag{7}$$

We solved for all variables in terms of measurable quantities. Since we're interested in the robot's position and orientation, the key variables of interest would be the distance traveled by the reference point and the change in the angle of rotation. The distance traveled by the reference point informs us of the position and the change in the angle of rotation informs us of the orientation. The radius of the curve containing the reference point, while useful for derivation, is not really needed anymore.

Now, we know the distance traveled, but not the direction. We know how much the orientation angle changed, but not the new orientation angle. So we start modifying our odometry model.

To simplify our model, we will represent the distance traveled by the reference point as a line instead of a curve. We can make this simplification because typically, in wheel odometry with encoders, the data sampling is very high. What this means is that our encoders are able to collect data very frequently so the time window between measurements is very small. Since

the time window is very small, the amount of motion captured by each time step will also be small. For our model, that means the curvature of the arc will be very small and resemble a straight line. Thus, it's a safe assumption and simplification to represent our distance now as a straight line. We then calculate the angle of the distance in terms of a previously solved variable, and the new orientation of the robot as shown in Figure 6:

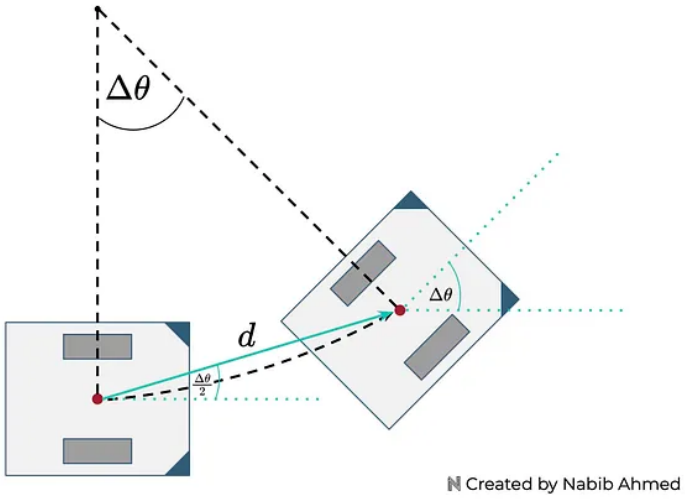

Figure 6

In Figure 7, the odometry model at time t will add the absolute orientation angle from the previous time step. Notice that adding the orientation from the previous time step won't change the distance traveled by the reference point or the change in angle of rotation as the formulas we derived from earlier don't rely on the orientation angle (only the traveled wheel distances). Instead what does change is the orientation of the robot, from being relative between time steps to now being absolute on the coordinate plane. Thus, the absolute orientation angle at any time step can be defined by:

$$\theta_t = \theta_{t-1} + \Delta\theta_t \tag{8}$$

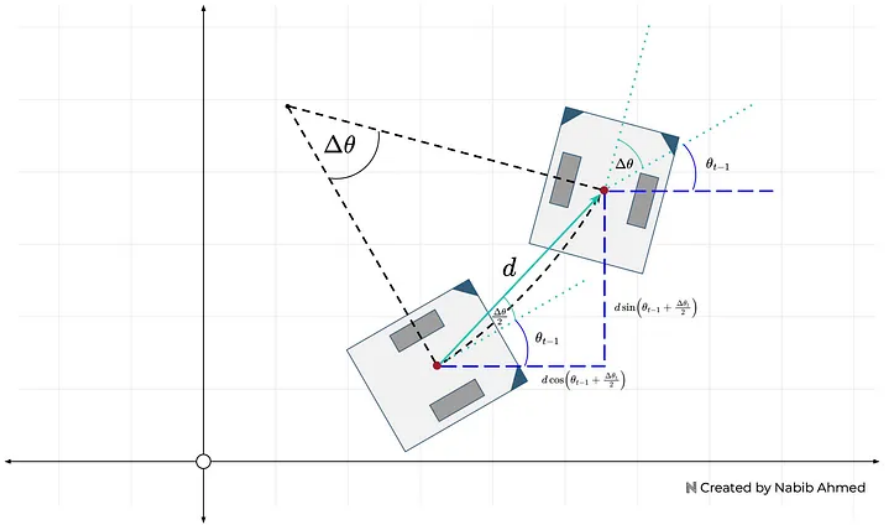

Figure 7

Using the distance traveled by the reference point and the angle of orientation from the previous time step plus the angle that results from motion, the amount of distance traveled along the x and y directions can be calculated.

$$x_t = x_{t-1} + dcos(\theta_{t-1} + \frac{\Delta\theta_t}{2}) \tag{9}$$

$$y_t = y_{t-1} + dsin(\theta_{t-1} + \frac{\Delta\theta_t}{2}) \tag{10}$$

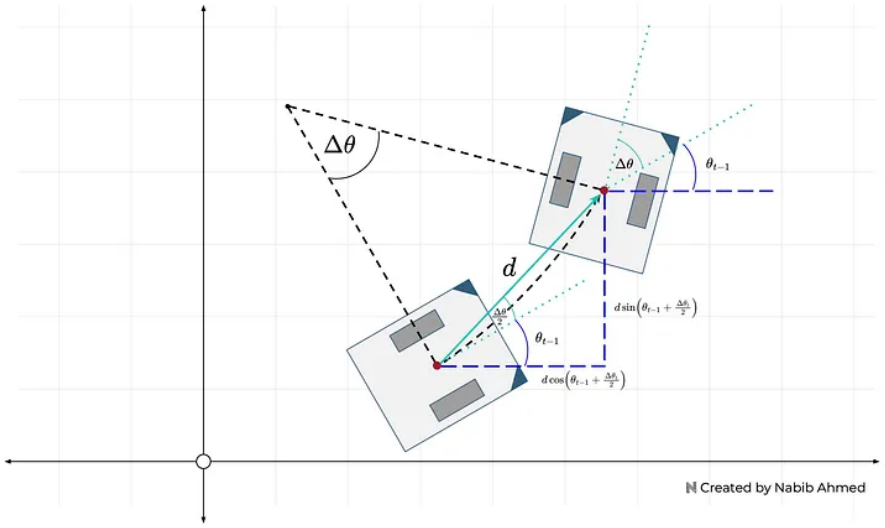

Figure 8

```
EddieOdomPublisher::EddieOdomPublisher(std::shared_ptr<rclcpp::Node>
    node_handle) : nh_(node_handle)
{
    odom_pub_ = nh_->create_publisher<nav_msgs::msg::Odometry>("odom",
    50);
    encoders_sub_ = nh_->create_subscription<eddiebot_msgs::msg::Encoders
    >("/eddie/encoders_data", 1,
      std::bind(&EddieOdomPublisher::encoder_cb_, this, std::placeholders
    ::_1));

    x_ = y_ = th_ = 0.0;

    prev_left_encoder_cnt_ = prev_right_encoder_cnt_ = 0;

    odom_broadcaster_ = std::make_unique<tf2_ros::TransformBroadcaster>(
    nh_);

    current_time_ = last_time_ = nh_->get_clock()->now();
}

void EddieOdomPublisher::encoder_cb_(const eddiebot_msgs::msg::Encoders::
    ConstSharedPtr msg)
{
    current_time_ = nh_->get_clock()->now();
    double dt = (current_time_ - last_time_).seconds();

    // msg->left(right) is to the total tick of the left(right) encoder
    // delta_left_cnt represents the increment of the left encoder ticks
    int delta_left_cnt = msg->left - prev_left_encoder_cnt_;
    int delta_right_cnt = msg->right - prev_right_encoder_cnt_;

    double delta_th = 1.0 * (delta_right_cnt - delta_left_cnt) *
```

```
    DISTANCE_PER_COUNT / WHEEL_BASE;
    double delta_dist = 0.5 * (delta_right_cnt + delta_left_cnt) *
    DISTANCE_PER_COUNT;
    double delta_x = delta_dist * cos(th_);
    double delta_y = delta_dist * sin(th_);

    x_ += delta_x;
    y_ += delta_y;
    th_ += delta_th;

    if(th_ > TWOPI)
        th_ -= TWOPI;
    else if(th_ <= -TWOPI)
        th_ += TWOPI;

    // printf("x = %lf, y = %lf, th = %lf\n", x_, y_, th_);

    prev_left_encoder_cnt_ = msg->left;
    prev_right_encoder_cnt_ = msg->right;
    last_time_ = current_time_;
    RCLCPP_DEBUG(nh_->get_logger(),
                 "got x = %lf, y = %lf, th = %lf, dt = %lf\n", delta_x,
    delta_y, delta_th, dt);

    publish_odom_(delta_x, delta_y, delta_th, dt);
}

void EddieOdomPublisher::publish_odom_(double dx, double dy, double dth,
    double dt)
{
    //since all odometry is 6DOF we'll need a quaternion created from yaw
    geometry_msgs::msg::Quaternion odom_quat = createQuaternionMsgFromYaw
    (th_);
    //first, we'll publish the transform over tf
    geometry_msgs::msg::TransformStamped odom_trans;
    odom_trans.header.stamp = current_time_;
    odom_trans.header.frame_id = "odom";
    odom_trans.child_frame_id = "base_footprint";
    odom_trans.transform.translation.x = x_;
    odom_trans.transform.translation.y = y_;
    odom_trans.transform.translation.z = 0.0;
    odom_trans.transform.rotation = odom_quat;
    //send the transform
    odom_broadcaster_->sendTransform(odom_trans);

    //next, we'll publish the odometry message over ROS
    nav_msgs::msg::Odometry odom;
    odom.header.stamp = current_time_;
    odom.header.frame_id = "odom";
    //set the position
    odom.pose.pose.position.x = x_;
    odom.pose.pose.position.y = y_;
    odom.pose.pose.position.z = 0.0;
    odom.pose.pose.orientation = odom_quat;
```

```
76      //set the velocity
77      odom.child_frame_id = "base_footprint";
78      odom.twist.twist.linear.x = dx / dt;
79      odom.twist.twist.linear.y = dy / dt;
80      odom.twist.twist.angular.z = dth / dt;
81
82      //publish the message
83      odom_pub_->publish(odom);
84  }
```

Listing 2: eddie_odom.cpp

3. SLAM and Navigation

3.1 2D SLAM

slam_toolbox

The slam_toolbox package incorporates information from laser scanners in the form of a LaserScan message and TF transforms from `odom->base_link`, and creates a map 2D map of a space. This package will allow you to fully serialize the data and pose-graph of the SLAM map to be reloaded to continue mapping, localize, merge, or otherwise manipulate. We allow for SLAM Toolbox to be run in synchronous (process all valid sensor measurements, regardless of lag) and asynchronous (process valid sensors measurements on an as-possible basis) modes.

slam_toolbox provides various tools and capabilities to address the problems with SLAM, and it offers several features:

- ***Basic 2D SLAM for Mobile Robotics***: slam_toolbox allows users to perform standard point-and-shoot 2D SLAM, where a mobile robot explores the environment, builds a map, and saves it in PGM (Portable Graymap) file format. The library also includes utilities to facilitate map saving.
- ***Continuing and Refining Mapping***: slam_toolbox allows users to continue mapping from a saved pose-graph. This means a previously serialized map can be loaded, and the robot can continue exploring and refining the map.
- ***Life-Long Mapping***: The library supports life-long mapping, where a robot can load a previously saved pose-graph and continue mapping in a space while intelligently removing extraneous information from newly added sensor scans.
- ***Optimization-Based Localization***: The library provides an optimization-based localization mode that utilizes the pose-graph. It allows the robot to determine its pose accurately based on the map and sensor data. Additionally, it offers the option to run localization mode without a prior map, using "lidar odometry" mode with local loop closures for localization.

- ***Synchronous and Asynchronous Modes***: slam_toolbox offers both synchronous and asynchronous modes for mapping, giving flexibility in how data is processed and utilized.

slam_toolbox's precess consists of four important steps:

1. **ROS Node**: SLAM toolbox is run in synchronous mode, which generates a ROS node. This node subscribes to laser scan and odometry topics, and publishes map to odom transform and a map.
2. **Get odometry and LIDAR data**: A callback for the laser topic will generate a pose (using odometry) and a laser scan tied at that node. These PosedScan objects form a queue, which are processed by the algorithm.
3. **Process Data**: The queue of PosedScan objects are used to construct a pose graph; odometry is refined using laser scan matching. This pose graph is used to compute robot pose, and find loop closures. If a loop closure is found, the pose graph is optimized, and pose estimates are updated. Pose estimates are used to compute and publish a map to odom transform for the robot.
4. **Mapping**: Laser scans associated with each pose in the pose graph are used to construct and publish a map.

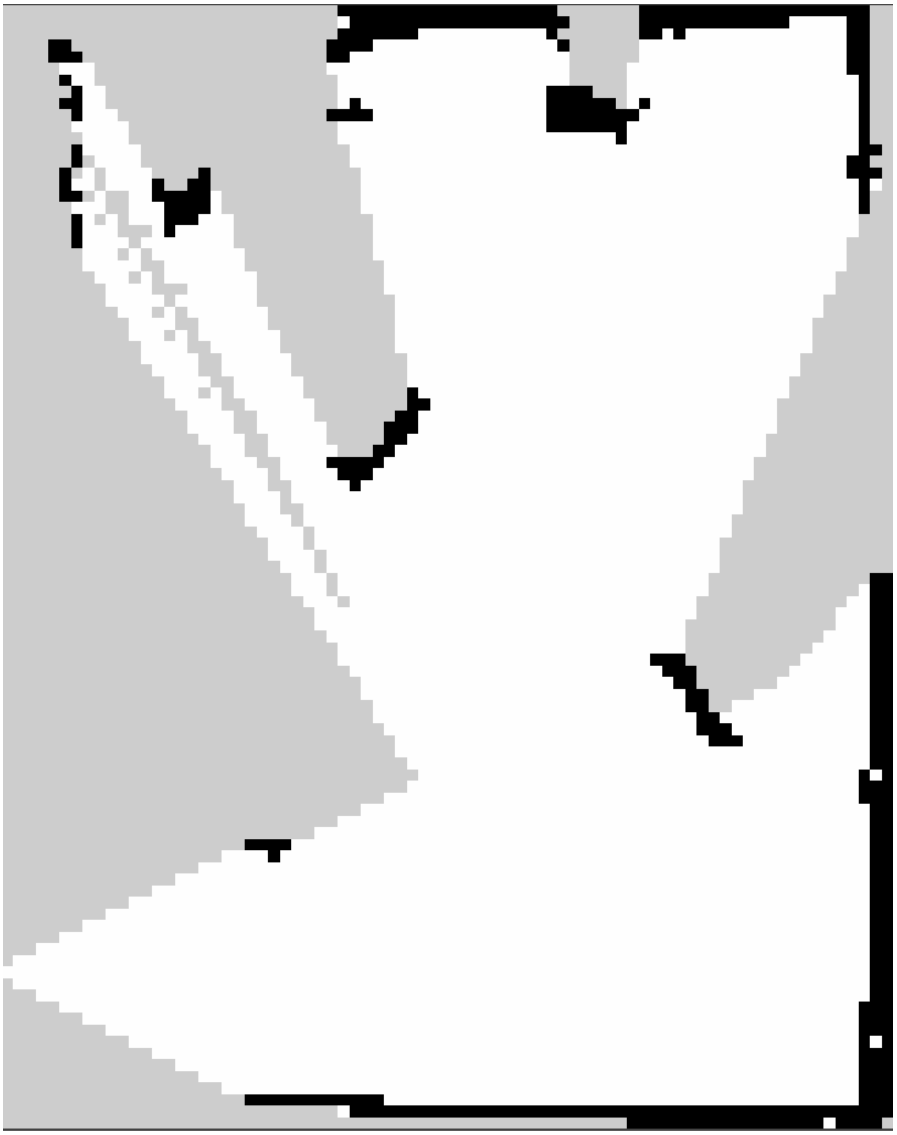

Figure 9: The map obtained in the video below

Here is our attempt at testing slam_toolbox alongside nav2.

3.2 Nav2

Navigation in robotics refers to the ability of a robot to move from one location to another in an environment while avoiding obstacles and reaching its destination safely. The Nav2 project leverages ROS2 for building its navigation stack, which includes various components to enable mobile robot navigation.

1. **Action Servers**: In the context of ROS2, action servers are used to manage long-running

tasks that may take a significant amount of time to complete, such as navigation. They allow clients to request specific tasks, and the server provides feedback and results during the execution of the task. The Nav2 stack utilizes action servers extensively to handle navigation tasks, enabling efficient execution of complex actions like path planning and control.

2. **Lifecycle Nodes and Bond**: ROS2 introduces the concept of lifecycle nodes, which are nodes that follow a state machine-based lifecycle. This helps in ensuring deterministic behavior during the startup and shutdown of ROS2 servers. The bond connection in Nav2 is used to ensure the active status of servers after they transition up. If a server crashes, the bond notifies the lifecycle manager, preventing critical failures.

3. **Behavior Trees**: Behavior trees are a way of organizing complex robotics tasks in a hierarchical manner. In Nav2, the BehaviorTree CPP V3 library is used to construct behavior trees. These trees consist of various nodes representing specific behaviors or tasks. Behavior trees provide a formal structure for navigation logic and allow for the creation of complex systems while maintaining verifiability and validation.

4. **Navigation Servers**: The Nav2 project employs several action servers to handle different aspects of navigation:
   - **Planner Server**: Responsible for computing a valid path from the robot's current position to a goal location based on the global environmental representation.
   - **Controller Server**: Handles local control efforts to follow the global plan or execute specific local tasks, like docking or avoiding obstacles.
   - **Smoother Server**: Refines the path computed by the planner to improve its smoothness and overall quality.
   - **Behavior Server**: Executes various recovery behaviors to deal with unknown or failure conditions, making the system more fault-tolerant.

5. **Waypoint Following**: Waypoint following is a fundamental feature of navigation systems. The Nav2 stack includes a waypoint following program with a plugin interface for executing specific tasks at multiple waypoints. It is useful for completing tasks like taking pictures, picking up objects, or waiting for user input at specified locations.

6. **State Estimation**: State estimation involves determining the robot's pose (position and orientation) relative to a global reference frame. In Nav2, two main transformations are essential: `map->odom` and `odom->base_link`. Global positioning systems (like GPS or SLAM) provide the `map->odom` transformation, while the odometry system (wheel encoders, IMUs, etc.) offers the `odom->base_link` transformation.

7. **Environmental Representation**: The environmental representation is how a robot perceives and models its surroundings. In Nav2, costmaps are used for this purpose. A costmap is a regular 2D grid that assigns costs to cells representing different types of areas (unknown, free, occupied, or inflated cost). Various costmap layers, implemented

as pluginlib plugins, buffer information from sensors into the costmap to provide a comprehensive representation of the environment.

8. **Costmap Filters**: Costmap filters in Nav2 are used to apply spatial-dependent behavioral changes based on annotations provided in filter masks. Filter masks contain data about specific areas in the environment where certain behaviors or restrictions should be applied. Costmap filters read this data and update the underlying costmap to alter the robot's behavior in those areas.

Overall, Nav2 provides a powerful and flexible framework for mobile robot navigation within the ROS2 ecosystem, with support for various navigation-related concepts and components.

We specifically focus on optimizing the `nav2_velocity_smoother` for our task. In order to do that, we will need to understand the key parameters that influence the smoothing behavior. The `nav2_velocity_smoother` is a lifecycle-component node that is part of the Nav2 navigation stack. Its main purpose is to take in velocity commands from Nav2's controller server and apply smoothing to these commands before sending them to the robot's hardware controllers. It achieves this by taking input commands from the cmd_vel topic and producing a smoothed output on the smoothed_cmd_vel topic.

Key features and design choices of the `nav2_velocity_smoother` node:

1. **Lifecycle and Composition Management**: The node utilizes the ROS 2 lifecycle manager for state management, which ensures predictable behavior during startup, shutdown, and runtime. Additionally, it utilizes composition for process management, allowing it to work seamlessly with other components in the Nav2 stack.

2. **Timer-based Smoothing**: Instead of simply computing a smoothed velocity command in the callback of each cmd_vel input from Nav2, the node operates on a regular timer running at a configurable rate. This allows it to interpolate commands at a higher frequency than Nav2's local trajectory planners can provide. By running at a higher frequency, it provides a more regular stream of commands to the robot's hardware controllers and performs finer interpolation between the current velocity and the desired velocity. This results in smoother acceleration and motion profiles, leading to better overall performance.

3. **Open and Closed Loop Operation Modes**: The node supports two primary operation modes:
   - ***Open-loop***: In this mode, the node assumes that the robot was able to achieve the velocity sent to it in the last command, which has been smoothed. This assumption is valid when acceleration limits are set properly. It is useful when robot odometry is not particularly accurate or has significant latency relative to the smoothing frequency. In open-loop mode, there is no delay in the feedback loop.
   - ***Closed-loop***: In this mode, the node reads from the odometry topic and applies a smoother over it to obtain the robot's current speed. This current speed is then

used to determine the robot's achievable velocity targets, taking into account velocity, acceleration, and deadband constraints using live data.

The `nav2_velocity_smoother` node plays a crucial role in enhancing the performance of robot motion control. By smoothing the velocity commands, it reduces jerky movements and wear-and-tear on robot motors and hardware controllers. The ability to operate in different modes allows flexibility in adapting to various robot setups and odometry accuracy levels. Overall, this node is a valuable component in the Nav2 navigation stack, contributing to the smooth and efficient navigation of mobile robots.
The `nav2_velocity_smoother` is responsible for smoothing velocities to reduce wear-and-tear on robot motors and hardware controllers by mitigating jerky movements resulting from some local trajectory planners' control efforts. The main `nav2_velocity_smoother` parameters that we adjust on are:

- `'smoothing_frequency': 0.1`
  This parameter controls how quickly the velocity is adjusted to smooth out accelerations and jerky movements. A lower smoothing frequency will result in slower adjustments, while a higher smoothing frequency will make the smoothing more responsive.
- `'scale_velocities': True`
  When set to true, this parameter adjusts other components of velocity proportionally to a component's required changes due to acceleration limits. It tries to make all components of velocity follow the same direction, but still enforces acceleration limits to guarantee compliance, even if it means deviating off the commanded trajectory slightly. This can help in maintaining a smoother and more coordinated motion of the robot.
- `'feedback': 'OPEN_LOOP'`
- `'max_velocity': [1.4, 0.0, 0.2]`
- `'min_velocity': [-1.4, 0.0, -0.2]`
  In some cases, you may want to set a minimum velocity to ensure the robot continues moving even in low-speed situations. This can be useful for preventing the robot from coming to a complete stop during trajectory planning.
- `'max_accel': [1.125, 0.0, 1.125]`
  This parameter limits the rate at which the linear and angular velocities can change. By constraining the accelerations, you can prevent sudden and abrupt changes in velocity.
- `'max_decel': [-1.125, 0.0, -1.125]`
- `'deadband_velocity': [0.0, 0.0, 0.0]`
  The deadband is a small range around zero velocity where no adjustments are made. This parameter sets the minimum velocities (m/s) to send to the robot hardware controllers to prevent small commands from damaging hardware controllers if that speed cannot be achieved due to stall torque. It helps avoid applying very small velocities that might lead to wear-and-tear on the robot hardware.

3.3 vSLAM

The development of 2D laser scanners was a game-changer for robotics, but their high cost has become a bottleneck for widespread adoption. To enable the next wave of robot applications, more affordable RGB-D sensors and cameras are crucial, especially in challenging environments where 2D techniques are not suitable. These advancements will allow for broader robot deployment and address new problems in various industries.
Some vSLAM approaches that we considered include:

- **ORB-SLAM3**: ORB-SLAM3 is a versatile visual SLAM method designed to work with various sensors, including monocular, stereo, and RGB-D cameras. ORB-SLAM has often adapted to enhance odometry systems for service robots. The latest version, ORB-SLAM3, introduces significant improvements, such as visual-inertial SLAM, allowing sensor fusion of visual and inertial sensors for more robust tracking in challenging environments with fewer point features. It also supports multi-map continuation and fisheye camera compatibility, expanding its applicability to different camera types.

- **RTABMap**: RTABMap is one of the oldest mixed-modality SLAM approaches. It stands out for its flexibility in accepting various input sensors, including stereo, RGB-D, fish-eye cameras, odometry, and 2D/3D lidar data. Unlike traditional feature-based SLAM methods, RTABMap creates dense 3D and 2D representations of the environment, making it unique and highly compatible as a semi-visual SLAM replacement. This characteristic allows for seamless integration into existing systems without additional post-processing, offering a compelling advantage.

- **OpenVSLAM**: OpenVSLAM is a well-structured implementation of ORB feature-based Visual graph SLAM. It offers optimized feature extractors and stereo matchers, alongside a unique frame tracking module for fast and accurate localization, competing with state-of-the-art solutions. The versatility of OpenVSLAM allows it to work with various camera models, including equirectangular and fisheye, making it suitable for a wide range of applications.

In the end, we opted for RTABMap for our ground robot experiments in a low-featured indoor facility equipped with an RGB-D camera.

RTAB-Map

RTAB-Map (Real-Time Appearance-Based Mapping) is a Graph SLAM approach that utilizes RGB-D data for real-time mapping. It employs a global Bayesian loop closure detector to efficiently detect and close loops in the mapping process. RTAB-Map can function independently with a handheld Kinect or stereo camera to perform 6DoF (degrees of freedom) RGB-D mapping. Alternatively, it can be used on a robot equipped with a laser rangefinder for 3DoF mapping with a 2D LiDAR or 6DoF mapping with a 3D LiDAR.

In the context of ROS, RTAB-Map is available as a package that serves as a wrapper for the RTAB-Map SLAM approach. This ROS package enables seamless integration and use of RTAB-Map with ROS-based systems. Various additional packages are available that can work in conjunction with RTAB-Map to generate 3D point clouds of the environment or create 2D occupancy grid maps, which are useful for navigation tasks.
In our specific case, where we have an RGB-D camera, odometry information from wheel encoders, and fake laser scans generated from depth images, we fine-tuned RTABMap's parameters to achieve optimal performance and accuracy. Here's how we adjusted the parameters.

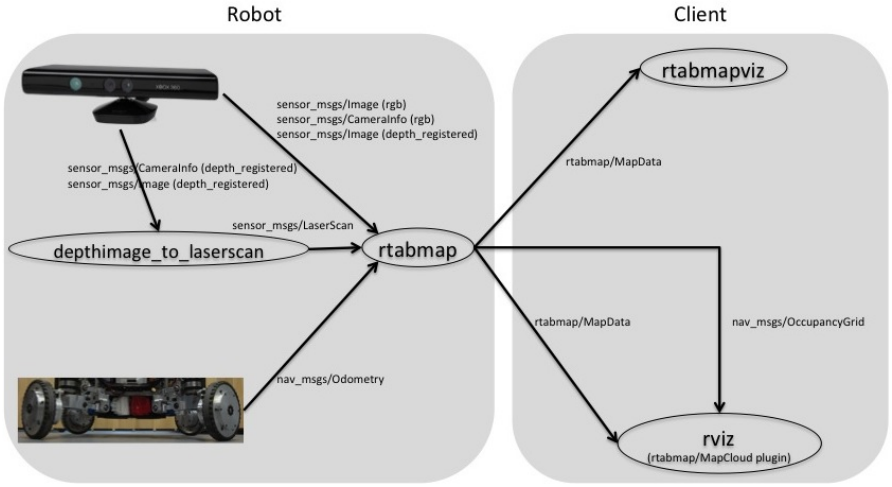

Figure 10: Kinect + Odometry + Fake 2D laser from Kinect

For RTABMap's ROS wrapper parameters:

- `'subscribe_depth': True`
  Subscribe to depth image
- `'subscribe_scan': True`
  Subscribe to laser scan
- `'subscribe_rgbd': False`
  Subsribe to rgbd_image topic.
- `'qos_image': LaunchConfiguration('qos')`
- `'qos_scan': LaunchConfiguration('qos')`
- `'use_action_for_goal': True`
  Planning Use actionlib to send the metric goals to move_base. The advantage over just connecting goal_out to move_base_simple/goal is that rtabmap can have a feedback if the goal is reached or if move_base has failed.

- `'approx_sync': True`
  Use approximate time synchronization of input messages. If false, note that the odometry input must have also exactly the same timestamps than the input images.

- `'tf_delay': 4.0`
  Rate at which the TF from /map to /odom is published

For most of UGV, the vehicle only runs on a flat ground, in this way, you can force the visual odometry to track the vehicle in only 3DOF (x,y,theta) and increase the robustness of the map. For rtabmap, we can also constraint to 3 DoF loop closure detection and graph optimization: For RTABMap's parameters:

- `'Optimizer/Strategy': '1'`
  Graph optimization strategy: 0=TORO, 1=g2o, 2=GTSAM and 3=Ceres.

- `'RGBD/ProximityBySpace': 'false'`
  Detection over locations (in Working Memory) near in space.

- `'Reg/Force3DoF': 'true'`

- `'Vis/MinInliers': '12'`
  Minimum feature correspondences to compute/accept the transformation.

- `'RGBD/AngularUpdate': '0.01'`
  Minimum angular displacement (rad) to update the map. Rehearsal is done prior to this, so weights are still updated.

- `'RGBD/LinearUpdate': '0.01'`
  Minimum linear displacement (m) to update the map. Rehearsal is done prior to this, so weights are still updated.

- `'RGBD/OptimizeFromGraphEnd': 'false'`
  Optimize graph from the newest node. If false, the graph is optimized from the oldest node of the current graph (this adds an overhead computation to detect to oldest node of the current graph, but it can be useful to preserve the map referential from the oldest node). Warning when set to false: when some nodes are transferred, the first referential of the local map may change, resulting in momentary changes in robot/map position (which are annoying in teleoperation).

By fine-tuning these parameters, we were able to tailor RTABMap to our specific setup. This process allowed us to achieve reliable and accurate SLAM results in our low-featured indoor facility.