Swiss Federal Institute of Technology Lausanne

Master project 2018-2019

Microengineering Section

Robot-elevator interaction

Author:
Paul Alderton

Professor & supervisor:
Alexandre Alahi

Laboratory:
VITA

Lausanne, 15 February 2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Project details

We want to benchmark the current methods for robot-elevator interaction. Four modes will be studied: a) waiting in front of the elevator, b) entering the elevator, c) waiting in the elevator, and d) exiting the elevator.

1. Provide a Literature review on these topics (download all relevant papers in a folder and summarize).
2. Define an evaluation protocol for reproducibility. How well are existing robots tackling the 4 modes?
3. Datasets review. Is there any data on human interacting with elevator for any of the modes (e.g., from youtube)?
4. Get familiar with the Loomo API/Framework. Write a tutorial / documentation on how to use it.
5. Implement a live demo of Loomo in each of the 4 modes. Identify all failure cases.
6. Suggest ideas/methods to address failure cases.

Abstract

In this work, we propose to transfer human-awareness to robots, for the specific scenario where they share elevators with people. Our objective is to teach robots to mimic natural human behaviors in order to reduce the amount of stress and annoyance people feel when sharing elevators with current state-of-the-art robots. This is important because robots need to navigate in elevators to accomplish tasks in multi-story buildings, but they must first learn the social etiquette before they are accepted by people. To solve this issue, we propose a framework that builds a path planning module using inverse reinforcement learning, to infer a cost function from real human behavior, as this has proven successful in other crowd navigation tasks. The intuition is that it is more accurate and straightforward to infer a cost function from humans that are subconsciously aware of such behaviors than to manually craft a function with a long list of elusive factors. We expect that a robot will understand the social and cultural norms that are expected of human beings and that they will be able to seamlessly integrate public spaces.

Acknowledgements

I would like to thank **Prof. Alexandre ALAHI** for the opportunity to work on this project, for his help and time in supervising me during this work and for providing feedback in the writing of this report.

I would also like to thank **Mr. Yuejiang LIU**, for helping me in understanding and utilizing the *Loomo* robot, and also for his contributions in writing the *Loomo* tutorial.

Contents

1 Introduction

As robots are asked to perform tasks in complex human environments such as hotels and hospitals, it is becoming more and more crucial that they learn to navigate these busy buildings in a socially aware manner, without disturbing people any more than other humans. These environments now require robots to share tight spaces such as elevators and corridors with people, but before robots can be accepted in these areas, they must be capable to respect the same social constraints that we impose on other people.

Figure 1: *Loomo* robot entering an elevator

One of the inherent problems of traditional path planning algorithms is that they treat people as mere obstacles and do not take into account any of the social variables, which can make interactions with robots awkward and uncomfortable. Most of these social variables, that are naturally learned by humans, are difficult to define, and attempting to implement and optimize a cost function that takes all social constraints into account is an arduous task, making the use of conventional algorithms for path planning inconvenient in these cases. However, inferring such a cost function from real humans that are naturally aware of the required behaviors seems like a much more feasible task. As we wish to solve the problem of robots navigating between floors using

elevators shared with people, we propose to take the approach of learning a cost function using inverse reinforcement learning and apply it to the elevator scenario as it has already been proven successful in other crowd navigation tasks, such as walking past people in the streets.

Sharing constrained spaces such as elevators between humans and robots is still a challenging task. The problem remains a crowd navigation issue, however it comes with its own unique set of constraints that are a result of the amount of space available during navigation and the various unique behavioral rules that exist inside the elevator, whether it has to do with placement, waiting or entering. Furthermore, this problem differs by the goal we are attempting to reach as we are moving towards the same target area as people (the elevator), rather than a far away destination where the exact location of the target is of little importance. As a result, we propose to use a framework where the data used for training is specifically chosen to represent the unique behavior we wish to model.

The result of this work is not a fully implemented human-aware navigation system, but a partially implemented solution that can be built upon to fully complete our ambitious goal. The current system is comprised of a local obstacle map constructed by the robot, that helps with obstacle avoidance and path planning. It uses a rudimentary path planning module that is not the human-aware module we plan to build. To move along the required path, robot locomotion is managed, where the robot can localize from odometry, and the current framework is able to enter and exit elevators in the presence of people, while avoiding collision with the help of the obstacle avoidance module. Finally, a person detector is tested to add important information to the local map, but it is not integrated to the overall system, and an evaluation protocol is established to assess the performance of the robot.

In this report, I first write about other research related to this project to help understand the approach taken for this problem. Then I explain the current implementation of the solution, its remaining issues and the next steps required to democratize robots in elevators. In parallel with this project, I have also worked on documentation explaining how to use the *Loomo* robot and its C++ development kit, offering tips discovered while getting familiar with the robot, as there is still no documentation available. The tutorial is available in annex.

2 Related Work

There are many papers focusing on navigation and trajectory prediction in human populated environments. [1] attempts to predict the trajectories of humans of many different classes (pedestrians, bikers, cars...) in different outdoor spaces without the need of classifying each person by learning from a large data set of over 185'000 instances of human interactions. [2] is an attempt to learn and reproduce real human motions from a large-scale bird's view surveillance data set using the same initial conditions as real scenarios for validation and [3] proposes a framework that uses optical flow for feature detection and inverse reinforcement learning to learn a cost function of social behavior with an implementation on a real robot that navigates with ease in a busy hallway. However, none of these focus on constrained spaces such as the elevator scenario where social rules are different; we must give priority to people exiting, move to the back of the elevator to allow others to enter, or leave a different amount of personal space between people depending on the situation; invalidating their implementation for this specific scenario.

In the elevator scenario, [4] [5] propose a method that plans a path towards the maximum fitness position in the elevator found after creating an occupancy grid map, but humans are treated as simple obstacles and no human-aware behavior is considered. Many focus on robots capable of using elevators, sometimes recognizing buttons and pressing buttons but they assume that the robot is the only user of the elevator and do not implement socially-aware behaviors. On the other hand, we would like to allow a robot to share an elevator with people seamlessly, adhering to social rules while making interactions with people feel comfortable and natural.

Robots capable of sharing an elevator with people is not a new concept and does exist. *Aethon*'s TUG [6], a commercial delivery robot employed in the healthcare and hospitality industries, is capable of operating elevators and although it occasionally has its own specific elevator that is not shared with people, this is not always the case and the robot is capable of sharing the space with people. Newer versions of the robot (TUG T4) are equipped with omnidirectional wheels allowing it to rotate on itself in the elevator in a tight area (although it has a rectangular shape) and it uses a stereo camera, as well as sonar, lasers and infrared sensors for efficient navigation. However, specifics on the TUG T4's navigation have not been published by *Aethon* as this is a commercial project. To the best of my knowledge, videos of older versions of *Aethon* TUG navigating in a hotel showed it stop abruptly at the sight of people in proximity, which would lead to assume that the behavior is not human-aware and mostly reactive although this might not be the case for newer versions with a stereo camera.

Human-aware navigation in elevators and other such constrained spaces (excluding corridors) have been largely unexplored, and publicly available datasets of humans interacting in elevators do not exist. As such, to the best of our knowledge, there has yet to be an attempt to create human-aware robots in the specific elevator sharing scenario, by inferring human behavior to a robot from data of people in elevators.

3 Support material

3.1 *Loomo*

The robot that is used to implement the elevator framework during this project is the *Loomo*[1] robot that is produced by *Segway Robotics* and visible in figure 2. It is a small two-wheeled autonomous robot with many sensors, measuring 64x57x28cm (height x length x width) and weighing 17.5kg.

Figure 2: *Loomo* robot
Source: `https://www.ctvnews.ca/sci-tech/robotics-on-the-agenda-for-ces-2018-1.3660628`

3.1.1 Locomotion

The most important features relating to movement is that *Loomo* is non holonomic as it is a differential wheeled robot and as a consequence, it cannot move sideways like humans which can be a critical constraint in an elevator. Also, it does not have any additional wheels to remain balanced; thus, the main priority of the robot is to adjust wheel speed to tilt the robot and avoid falling down, although a future version of the robot will allow the addition of extra wheels to turn off the self-balancing mode. This is important as delays are caused when accelerating and stopping to keep the robot balanced.

[1] `https://www.loomo.com/en/`

3.1.2 Sensors

Loomo is equipped with a large variety of sensors. One of its important features is the RGB-Depth camera that finds its use in vision related tasks such as detection and tracking of humans, which is essential for this project. It also has an HD camera with a wide field of view and ultrasonic sensors to help detect obstacles although it cannot detect obstacles that are closer than 25cm away (range of detection is approximately 5m). Additionally, hall sensors in the wheel motors provide odometry data, and 5 microphones are used for voice recognition and localization of the sound source.

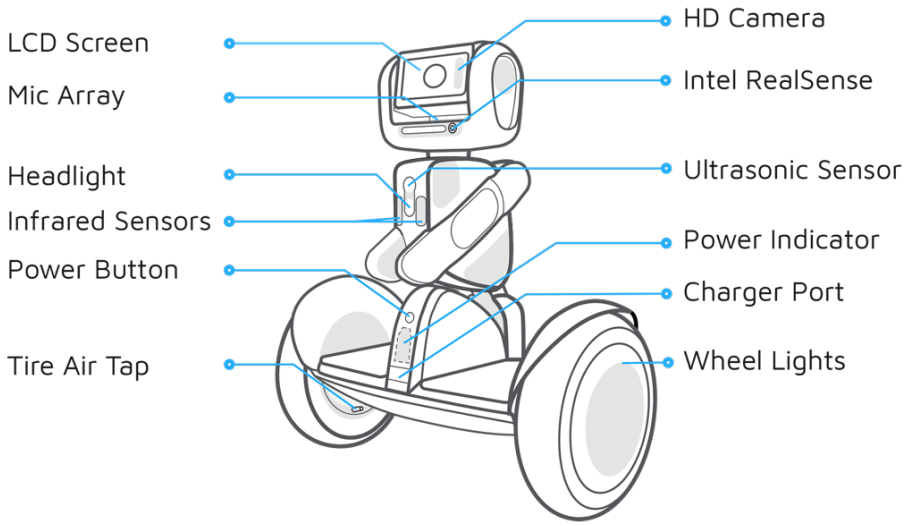


Figure 3: *Loomo* hardware
Source: `https://www.loomo.com/en/spec.html`

The position of the sensors of the robot can be visualized in figure 3. Most sensors are limited in what they can detect because they cannot move freely, such as the ultrasonic sensors or Intel RealSense RGB-D camera that cannot rotate up and down. As such, *Loomo* can have trouble detecting small obstacles on the ground that prevent it from moving. There can also be issues detecting obstacles on the sides of the robot, especially when moving, as the head has a 60° field of view and must be well oriented to detect obstacles.

3.1.3 Computation

For computation, *Loomo* is equipped with Intel Atom Z8750 processor, a 4 core CPU with a 2.4GHz frequency. This is not always optimal for compute intensive algorithms, so it is sometimes preferable to opt for communication with a server that computes the most demanding tasks.

3.1.4 Software development kit

The official SDK for *Loomo* is in Java. However, for the purpose of this project, we will be using a C++ development kit for *Loomo* that uses the Android NDK toolset for Android Studio to allow to code some parts in C and C++.

During this project, I have written a tutorial document for using the C++ development kit as it is still undocumented. The tutorial linked in annex contains notes and examples that are helpful to get started with coding on *Loomo* in C++.

4 Evaluation protocol

An evaluation protocol must be defined to determine how well the robot performs the various tasks of using the elevator. These tasks can be separated in 4 categories:

1. Waiting in front of the elevator
2. Entering the elevator
3. Waiting inside the elevator
4. Exiting the elevator

Each task requires a different set of metrics to evaluate performance although there are strong similarities between the two waiting tasks and between both the entering and the exiting tasks.

4.1 Tasks and evaluation metrics

4.1.1 Waiting in front of the elevator

The process of waiting seems straightforward as it does not require any movement at first glance, however, to comply with cultural norms, it will be necessary to respect a few criteria. The following metrics will be used for evaluation:

- Obstacle avoidance: no collisions while reaching the desired waiting area.
- Ethical sensitivity: waiting on the sides to allow people to exit.
- Human comfort: are people comfortable waiting with the robot?
- State observation: determine the door state at all times or in sufficient intervals, usually by facing the door.

The most important part of this task is to wait on the sides, away from the elevator doors, as depicted in figure 4, in order to allow people to exit the elevator when the doors open. The robot must also be able to evaluate the state of the door from a distance and stay away while the door is closed to avoid moving back and forth between the waiting area and the door. It must also stay behind people that are already waiting to give them priority, where the presence of people may indicate that the elevator doors are closed. During this time, the robot can look around at other people in order to gain more information that may be of use when entering the elevator (number of people, age, mobility, waiting order, elevator button...).

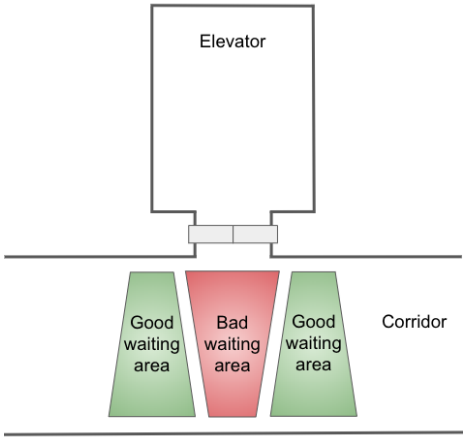


Figure 4: Examples of good and bad waiting areas. These are suggested areas and may not be correct. There is not necessarily a good and bad answer to this problem as there can be many different correct areas that can be learned and that are dependant on culture.

4.1.2 Entering the elevator

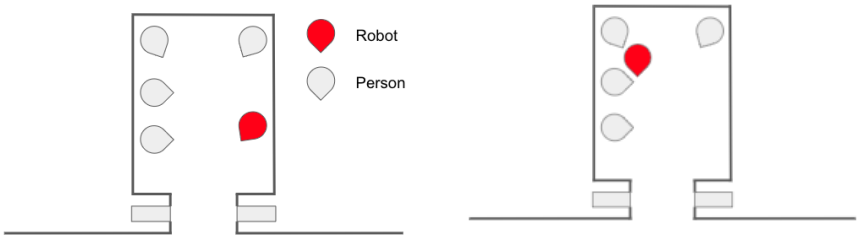
When it comes to entering the elevator, more variables must be taken into account. The metrics for evaluating this task are the following:

- Obstacle avoidance: absence of collisions.
- Ethical sensitivity: placement according to the position of people in the elevator (figure 5), allowing others to exit first, giving entrance priority to people that deserve it, understanding that it is acceptable to force the entrance when there is space in the back (figure 6b).
- Human comfort: how comfortable people are around the robot.
- Time: the robot must enter the elevator in a timely manner once doors are open.

4.1.3 Waiting inside the elevator

Waiting in the elevator has similar metrics for evaluation as waiting outside. Once it is well positioned inside the elevator, the robot will need to face the door in order to know when it can leave, but it will also need to move around depending on the intentions of other occupants. For example, if a person wishes to exit and the robot is blocking the path, it may need to move to the side (figure 7), or sometimes exit the elevator to let a person pass. The same goes as people attempt to enter, the position of the robot will have to be adapted based on the current occupancy of the elevator and the amount of people entering (figure 8). Here are the evaluation metrics:

- Obstacle avoidance: absence of collisions.



(a) The robot is placed at a good position in the elevator and does not disturb others.

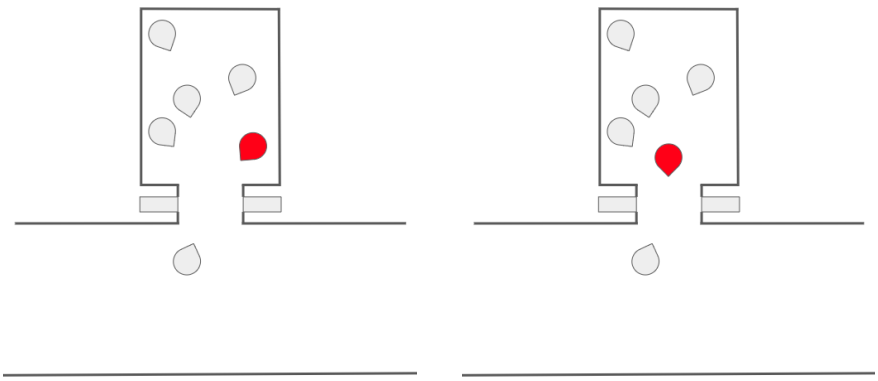
(b) The robot is stopped at a bad position in the elevator. It is waiting in a position that is too close to other occupants and invades their personal space.

Figure 5: Examples of good and bad placements

(a) The robot is trying to enter a full elevator. This is a bad decision and the robot should have waited outside.

(b) The robot is entering an elevator that appears to be full in the front but that has plenty of free space in the back. The decision to enter is good because the other occupants can move to the back.

Figure 6: Examples of good and bad entry decisions



(a) Good waiting position. The robot is waiting in a position that allows the other occupants to exit.

(b) Bad waiting position. The robot is waiting in a position that does not allow other occupants to exit the elevator.

Figure 7: Examples of good and bad waiting positions

- Ethical sensitivity: does the robot adapt placement to allow occupants to exit and enter more easily?
- Human comfort: how comfortable people are around the robot.
- State observation: determine the door state at all times or in sufficient intervals, usually by facing the door. Examine the state of people inside the elevator.

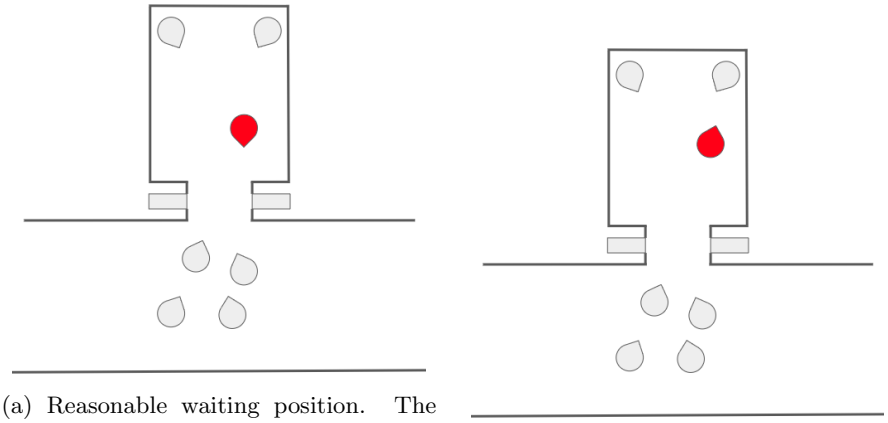
4.1.4 Exiting the elevator

Exiting the elevator is similar to entering, but with a few simplifications. Again, as with all movements, it must be collision-free and relatively quick to exit. The robot will have to give priority to people closer to the door and let them pass first if they wish to leave, and if there is no space to pass, it should make a sound to express intent before attempting to move. The metrics are as follows:

- Obstacle avoidance: absence of collisions.
- Ethical sensitivity: giving priority to people near the door, asking occupants to move when there is no space to exit (figure 9).
- Human comfort: how comfortable people are around the robot.
- Time: exiting the elevator in a timely manner.

4.1.5 Evaluation metrics

Most of the evaluation metrics such as time, obstacle avoidance and state observation are easy to evaluate and can be simply observed by a person in charge of evaluating performance. They will be evaluated on a scale of 1 to 5 based on how well the tasks are performed. Human comfort on the other hand, is



(a) Reasonable waiting position. The robot is waiting near the door, causing minor disturbances in flow, but occupants have enough space to enter and leave the elevator, so the robot positioning is acceptable. However, it would be more socially acceptable to move to the back.

(b) Good waiting re-positioning. At the sight of people entering, the robot moves to the back, offering space to others entering. This is a social behavior we wish to learn.

Figure 8: Examples of an acceptable waiting position, and of an improved and more social waiting position

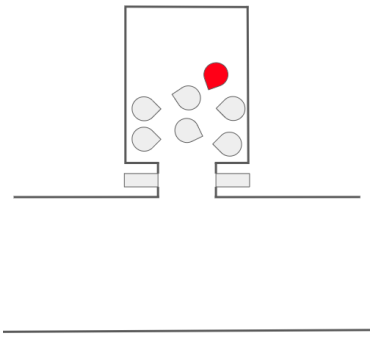


Figure 9: The robot is exiting the elevator, which is a bad decision because there is no space. As a social robot, it should ask and wait for people to move out of the way.

very subjective and will depend on people, thus, it is necessary to have a large amount of participants enter the elevator with the robot in different levels of elevator occupancy and to create a survey to understand where the issues lie.

The human comfort metric, that is defined as "the absence of annoyance and stress in interactions with the robot" in [7], shall be judged on a scale from 1 to 5, where the values range from stressful (1), to annoying (3), to natural (5). The participants may also comment on issues they wish to address, and rationalize the their scores.

The ethical sensitivity metric is a measure of the adherence to social rules that are different for each scenario, such as giving priority, or asking for space to

exit. Like other metrics, this will also be measured on a scale from 1 to 5 where the score will represent how well the various rules are met in each scenario.

4.2 Scenarios

4.2.1 No human interaction

In a first step, we will evaluate the robot performance in an empty elevator and corridor. Here, we will verify some of the most basic metrics such as time, obstacle avoidance and state observation. Ethical sensitivity is mainly dependant on human presence, but when waiting for the elevator, it can be evaluated as the robot must wait on the sides, regardless of whether there are people in the elevator or not.

In a next step, we will add obstacles in and around the elevator for extra complexity, to make sure that the robot places itself according to space and only enters when the space is adequate. This second scenario will help verify that the robot scales to different levels of complexity. Again, collision avoidance is a must in this situation with static objects and the robot must not attempt to enter if there are non-human obstacles blocking the entrance. Obstacles should impact the decisions of the robot and we must check that the decisions are correct before placing humans in the equation.

4.2.2 With human interaction

Finally, to evaluate all metrics, the robot must be placed in an environment with real people, entering and exiting the elevator at different times. Aside from having the same criteria as the previous cases without people, it is now necessary to evaluate the human comfort associated with the presence of the robot, by using human evaluations. With people, we can now also verify all aspects of the ethical sensitivity metric of the robot.

Overall, a social robot must respect personal space, feel non-threatening, have natural, human-like non-jittery movement, avoid touching anyone and anything, and should be able to understand the dynamics of people; leaving space for those entering/exiting, as well as giving priority to certain people while understanding that it can be acceptable to push its way in if there is sufficient space at the back.

Some of these criteria cannot be measured mathematically, hence the need of a survey. We will need to test the robot in different cases of crowded and spacious elevators to fit all scenarios and gather as many participants as possible to obtain meaningful results as this evaluation is very subjective.

4.3 Limitations

4.3.1 Data and human-aware planner

As we do not currently have the data of people using elevators, we are still unable to create the human-aware planner we envision. In the meantime, we will use the planner from the C++ development kit for *Loomo* that is not human-aware, but we can start work on the framework with tasks such as obstacle avoidance and general elevator usage.

4.3.2 Mapping

Eventually, the robot will use a global map indicating the position and size of the elevator, however, as no map is provided yet, the robot is unable to locate the elevator on its own and we must manually give the robot the necessary checkpoints to help it navigate. The absence of localization also means the robot cannot be placed anywhere, and the success of our scenario is dependant on starting conditions. With the size of the elevator not given, the robot will search for the best position to stop in a preset area of the elevator, chosen before entering. This area can be smaller, or slightly bigger than the elevator but the robot should choose a spacious location to stop. This is not human-aware placement as sometimes it is preferable to stop close to people, but this is the intermediate solution until the human-aware planner and a full map are created.

5 Solution

The most closely related area of research that is still human-aware navigation and that has successful implementations is socially aware navigation in a crowd, where the most reliable and efficient solutions seem to converge to the use of vision sensors and reinforcement learning. The use of real human data to infer behavior is not surprising as there are many rules with multiple solutions that cannot all be written down explicitly. Since the case of the elevator is similar in the sense that it is also human-aware navigation in a crowd, we have decided to adopt a similar approach. However, as the elevator scenario comes with its own set of rules and constraints, it was deemed necessary to train a new network for the robot, specifically for use in the elevator, only using data from elevators for training.

This solution can be decomposed into several key steps. First, the robot needs to extract features from the scene to recompose the different parts such as the location of the walls, obstacles and people; it is key to create a reliable map for the path planning step. Eventually, this feature extraction step will serve for detecting non-static objects like humans, and for localization in a prior map of the area, however, this map is not yet provided but should later considerably help for path planning, where the entrance and the size of the elevator are given. Next, it is important to train a network with inverse reinforcement learning for path planning, by using data of humans in elevators in order to learn the appropriate cost function that will model the behavior of humans. Finally, by combining the trained model with the data obtained from feature extraction (that should be similar to the data used for training), we should be able to run through the network that will act as the path planning module for the robot and use it to control all movements appropriately.

With everything implemented, it is important to test the functionalities following a strict evaluation protocol that will determine whether the robot's behavior is good enough.

5.1 Feature extraction

For feature extraction, we want to create a map of the scene using the built-in depth camera and ultrasonic sensors to place the walls and other obstacles

around the robot. As there is no prior map of the area given at the moment, the robot will combine sensor data and odometry to create a consistent map for navigation, but when a map is available, the detections should help for localization and for adding obstacles that are not yet in the map. With the RGB-D camera and an off-board person detector, we can place people in the current map, adding their orientation if deemed necessary, and predict their movement by using optical flow to extract crowd density and velocity. The different components of the map will serve as input data for the path planning module but will also be used for obstacle avoidance.

5.2 Path planning & inverse reinforcement learning

The goal of reinforcement learning is to learn the behavior that will minimize a cost function that must be manually defined beforehand. However, this would entail including each and every behavior we wish to learn in the cost function and weighing them appropriately which can be a very challenging task. As the name suggests, inverse reinforcement learning switches the problem and tries to learn the cost function from observed behavior in order to mimic what is seen. In this scenario, a robot does not have to be explicitly told what it must do but can derive certain instructions from the observed behavior, which is much more advantageous when there are complex tasks that are difficult to transcribe.

To obtain acceptable behavior, it is important to train a network with real data of humans using an elevator as part of their daily life with different levels of elevator occupancy. This data can be collected as videos but they should be labeled in order to have data that will be of the same format as what the robot creates after feature extraction. The data should contain the dimensions of the elevator, the position of people (and their size) as well as their movement speed, and the location and state of the door.

Using this module, the robot should become aware of the important factors and dynamics that must be considered when sharing an elevator, such as leaving personal space for people, or moving out of the way to allow those at the back to exit. In the end, the robot should know that the act of entering the elevator is not acceptable if its execution in doing so is not human-aware.

6 Implementation

6.1 Feature extraction

6.1.1 Obstacle detection & mapping

Obstacle detection is performed using the ultrasonic sensors to detect obstacles directly in front of the robot, or using the depth camera that has more flexibility in orientation and a wider field of view in height. As there is no prior map, we use the data from the depth camera to create and update the local map, rather than for self-localization in a map, which should be one of its objectives once a full map is available. Hence, the local map created is used for all planning tasks but is usually incomplete in some parts due to the limited field of view of the sensors. The update of the local map is performed with the *getDepthMapWithFrontUltrasonic()* function.

The field of view of the depth camera is approximately 60°, so the head must be controlled based on the robot's rotational velocity to detect obstacles when moving. It is also useful to scan the head around the body to update the parts of the map that are not in the field of view; this is only done when the robot is not moving to avoid missing the detection of obstacles in the path.

An example of an obstacle map is visible in figure 10.

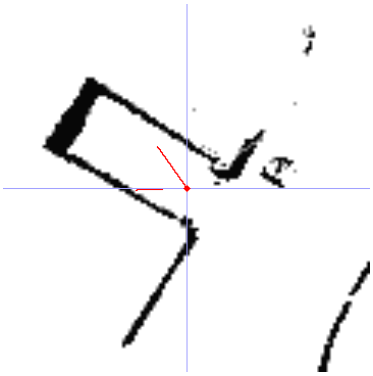


Figure 10: Local map of the robot inside an elevator. The red lines are the field of view, according to the orientation of *Loomo*'s base. The robot is at the center of the image where the lines meet. The black pixels are the obstacles detected by *Loomo*.

6.1.2 Person detection

Detecting people and placing them on the local map is an essential step for this project as we will need to place people in the elevator, possibly with their orientation as well, before we can apply a human-aware path planning module.

On a separate application from the elevator application, a person detector is implemented. This person detector, that is described in the tutorial in annex, is too slow to run on the on-board *Loomo* CPU as it takes a full second to run on a 640x480 pixel image. A sample code has been written to transfer the image to a cloud computer for computing the person detector, however, due to issues with the *Tensorflow* installation in C++, the detector was not able to run on the cloud computer, but computations would be considerably faster on a high-end desktop GPU and should run in less than 100ms. Examples of this detector are visible in figure 11.

Another person detector using the local map is implemented in the C++ development kit, it is very fast and simply uses the image of the local map as input, which is a 120x120 pixel image. People are detected by analyzing the size of connected components in the obstacle map, however, it is very inaccurate and has many false and missed detections. The previous person detector, that is slower but more precise, is a more favorable alternative to this one.

(a) Full body detection (b) Lower body detection.

Figure 11: Person detector examples. Detection bounding box is in pink. The detector can detect people regardless if only a leg is visible or the full body, which is very important for a small robot close to the ground.

6.2 Path planning & obstacle avoidance

6.2.1 Path planning

In the current version of the implementation, the robot is given a goal, or a set of waypoints, that serve as inputs to the path planner, that can in turn calculate the path towards the goal with the help of the local map, where some variables can be adjusted. Some of the variables of the path planner include the closest distance that the robot can approach an obstacle (set at 2cm) or the distance it can deviate from the path assigned (set at 30cm to avoid deviating too much). Although the planner allows us to control a wide range of variables, it does not react differently to people than it does to obstacles and thus, it does not satisfy the necessary requirements to be a human-aware planner.

The planner takes in as parameter the current pose and velocity of the robot, as well as the local map with obstacles. Based on these values, we can retrieve the velocities of both wheels from the planner but the planner is limited in the speed it can give the wheels. For example, the robot is unable to make very sharp turns or move backwards, which can be necessary readjustments to take a new path or to move away from obstacles. This is probably forced behavior due to the inability to detect obstacles behind the robot, but to smoothly move like humans in an elevator, it is also necessary to be able to move backwards when good assumptions of the current scene can be made.

6.2.2 Obstacle avoidance

The planner should be capable of navigating between obstacles in the map, however, for safety, obstacle avoidance is implemented in the case that unexpected obstacles appear or that the planner fails to stop in time. Obstacle avoidance uses the ultrasonic sensor to determine whether an object is in front of the robot, but we only receive a single value that gives no indication as to where the obstacle is oriented but generally, objects are more easily detected when directly in front of the robot.

When the ultrasonic sensor detects an obstacle over a certain threshold, the robot passes in safety control mode where it can start slowing down before

reaching the obstacle. It is important to detect obstacles well before reaching them because *Loomo* has to adjust its tilt before it can stop while remaining balanced, causing a delay when decelerating. Furthermore, before deciding to decelerate to avoid an obstacle, the desired linear and rotational speeds are compared to determine whether the robot is moving straight (towards the obstacle) or turning (away from the obstacle). If the rotational speed of the robot is higher than the linear speed, then the robot is less likely to stop as it is turning away from obstacles that are likely to be located straight ahead. In the event that the robot is turning towards another obstacle that is not yet detected and not used by the path planner, the robot should still be able to detect it in time with the depth camera and cancel the path before hitting the new object.

6.3 Locomotion

6.3.1 Base

The movement of the base is exclusively controlled with calls to the *safeControl()* function where we must provide the linear and rotational velocities we wish to give the robot, which is sometimes just the output of the path planner. In this function we execute the command that will make the wheels move but we also use the speed of the robot to control the head and perform general obstacle avoidance in the event the path planner is insufficient.

6.3.2 Head

The position of the head is crucial in detecting obstacles since it is where the depth camera is located and it is currently controlled based on the rotational speed of the base. The head can be controlled in speed or in position, however, in position mode, an unknown and undocumented controller is implemented that causes a delay in head movement whenever the base turns and as a result, the robot has trouble detecting obstacles in the path. Consequently, the head has to be controlled using speed which allows more flexibility, and a speed control function was implemented to detect obstacles more reliably.

The controller implemented for commanding speed is a PD controller since it satisfies the necessary requirements to detect obstacles successfully, namely a fast response time with little to no static error and a stable response. With w the rotational speed of the base, the speed of the head s is set at:

$$s = w + (w - HeadYawAngle)$$

In other words, we want the head to move at the same speed as the base, but with a slight offset in position to anticipate obstacles in the direction it is turning. The error in position is corrected with speed since we want the position of the head to vary according to the speed of the base, because the turns are much sharper with a higher rotational velocity. The speed of the head has been set experimentally, as these values seemed to position the head well enough to detect obstacles when cutting corners. A robot turning with its head well oriented is visible in figure 12b.

When the robot is not moving, the head turns to scan the environment around the robot to record any changes. This is especially useful when the environment changes around the robot, where a path to the side of the robot opens up, usually caused by humans moving out of the way.

(a) *Loomo* moving in a straight line (b) *Loomo* turning

Figure 12: Pictures of *Loomo* moving in a straight line and turning. (12a) *Loomo* is moving forward and looking straight ahead, (12b) *Loomo* is turning with the head slightly at an angle to detect obstacles.

6.4 Localization

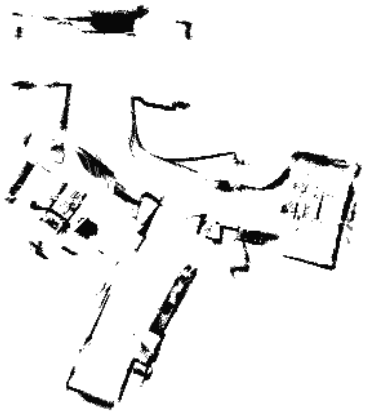
The current state of localization is very rudimentary; it simply consists of odometry to estimate position with respect to past detections. Since odometry inherently accumulates error, it becomes very imprecise with time and when the robot returns to a previously explored location, the map is replaced with new detections. As there is no prior map available, the implementation of a more complex localization algorithm to estimate location in a map such as Monte-Carlo was not considered. It is currently possible to build and save a map in memory from exploration as seen in figure 13a, but without any loop closure algorithm, the map can become completely distorted.

6.5 Elevator framework

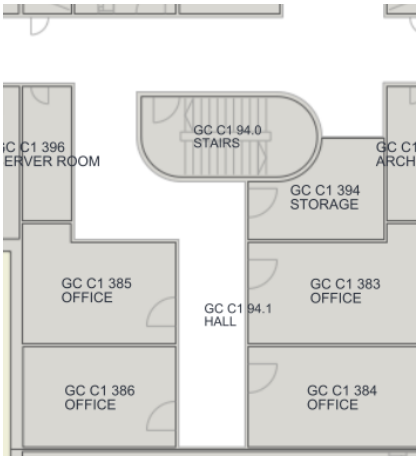
The current series of events for entering and exiting the elevator is manually programmed and starts by pressing the 'B' button on the *Loomo* screen. In the framework, the robot is given a series of tasks to accomplish or goals it must reach, before it is allowed to perform the next task of elevator usage. Figure 14 shows a photo collage of the entire framework where *Loomo* is entering and exiting the elevator.

6.5.1 Movement towards the door

The first step is to move to the door of the elevator. Here we tell the robot where the door is located and it must plan a path towards it and move. As we have no map of the area, and no door recognition algorithm, the location of the door



(a) Fixed map constructed from sensor measurements. The fixed map is typically 30m x 30m but is cropped here based on the area explored. The odometry error has caused the map to be completely distorted where some rooms are superposed.



(b) Floor plans of the office area mapped

Figure 13: Obstacle map created by the robot (13a) and its associated floor plans (13b). The map is hard to correlate with the floor plans because of the odometry error that has created distortions. There are some differences with regards to clutter, and only 2 office rooms are visited in the obstacle map, but a well reconstructed map should allow both maps to be superposed easily.

Figure 14: Collage of the robot entering and exiting the elevator with a person inside, holding the door open. Order is from left to right, top to bottom. The robot chooses the resting location in the elevator based on the space available.

.

is given at launch with respect to the starting position of the robot (in meters), so the robot must be relatively well positioned when starting the algorithm to

locate the door. The robot is pre-assigned 2 waypoints equally distributed on a straight line between the starting and endpoints, but it may deviate from these waypoints to a maximum of 0.3 meters away while still validating the waypoints. It is not necessary to give the robot multiple waypoints but when a complex path is known, waypoints are useful to guide the robot in a certain direction.

In this step, we also initialize the elevator position and the starting position that are saved as global variables. Since all positions are tracked in space by odometry, there can be slight offsets compared to where they are initially placed; this is especially visible when returning to the starting position. These positions appear on the *Loomo* screen as squares of different colours as seen in figure 15.

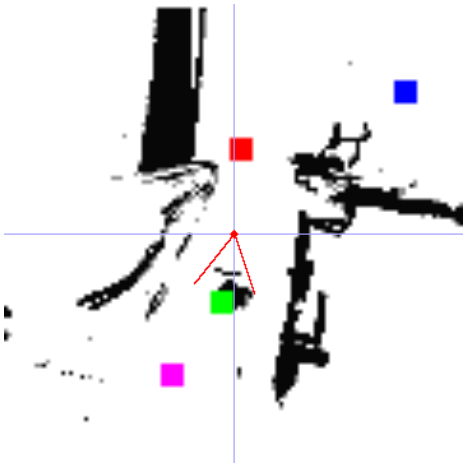


Figure 15: Local map of the robot where the elevator is simulated with a room. In blue is the starting location of the robot, in red is the door location of the elevator, in green is the position of the interior of the elevator, in purple is the calculated point in the elevator furthest from obstacles. The circular line just in front of the robot is the detection of the ultrasonic sensor.

6.5.2 Entering the elevator

Once the robot has reached the door, it moves to its next goal that is inside the elevator. Since the goal is set at the beginning and takes no account of the current state of the elevator, the robot creates a 1m x 1m square centered on the goal and it will search for the pixel furthest from obstacles within this square to set it as the final destination, by using the obstacle map and calculating the L2 distance from the nearest obstacle for each pixel. As the map can change quite often, if the current goal is still quite spacious, the robot does not attempt to replace it with a better position. The current goal that must be reached inside the elevator appears on screen in purple and it can be seen changing position depending on the update of obstacles located around.

6.5.3 Facing the door

Having reached the final destination inside the elevator, the robot then turns around to face the door. In this step, there is no path planning and the robot

turns no matter the surroundings, towards the angle it believes that the elevator door is located, which is determined by odometry. A function makes sure that the robot turns in the shortest direction and that it stops when reaching a threshold of orientation.

6.5.4 Returning to the starting point

Although there is a possible 4th step that can be used as waiting time, for example while the door closes, the final essential step is to return directly to the starting point, which is a simplified way of exiting the elevator. In this situation, the robot is already facing the door and after entering the elevator, it should have enough knowledge of the obstacles in between both locations to fully plan a consistent global path. As a result, if the door is closed, it knows that there is no possible path back and does not attempt to move. Unfortunately, the same goes for people in front of the door; if they are blocking the exit, the robot waits until the path is clear whereas it should move towards them and either follow them if they are exiting, or ask them to move.

In this situation, only one goal is given with no intermediate waypoints, as opposed to entering the elevator, and the robot has no problem finding its way back since it knows the area already. This highlights the importance of a global map as it improves the planning capabilities considerably. However, the robot is not robust at recognizing that objects can have different states such as doors or people, and as consequence, it treats all previously seen obstacles as permanent once they are no longer in the field of view. This issue can be solved with a global map that informs of the multiple states doors can have, and a human-aware planner that can help predict the movement of people.

6.6 Alternative framework

An alternative scenario for *Loomo* that uses prior knowledge of the map is available. In this scenario, *Loomo* must be moved in the elevator to save its location, and at the elevator door to save the door location. After this, the robot can be put outside of the elevator, and knows exactly where both goals are located, so it is no longer dependant on the starting position, but this requires a few extra steps for setup. When inside and outside of the elevator, it is good practice to scan the environment by manually moving *Loomo*'s head to have the most complete map possible. However, a closed elevator door must not be scanned because the planner does not know that doors can open and it will not make the robot move if the local map shows a goal surrounded by obstacles. If a closed door is scanned, the robot must start with the door visible from the robot's location to detect it when it opens.

This scenario uses the same steps as the previous scenario, with the sole difference that door and elevator locations are no longer estimated but are shown to the robot. The success of entering the elevator is no longer dependant on the starting position, offering more flexibility and autonomy to the robot.

To use this scenario, the user must press the 'C' button on the *Loomo* screen to save the current location as the elevator location. Next, the robot must be rolled outside of the elevator, where the user must save the door location by pressing the 'A' button; the order of locating the elevator or the door first is unimportant. Once in the corridor, by pressing the 'B' button as usual, the

robot will head to the elevator using the new mode (the 'C' button makes the robot change to this alternative scenario).

7 Current issues

7.1 Mapping & localization

7.1.1 Mapping

One of the limitations of the current system is the absence of a global map. As such, there is no prior knowledge of the location of the elevator, nor of its dimensions, and the robot must start at a given position and be told exactly where to go, losing some of its autonomy. There is a small margin for error allowed when initially placing the robot, but when the robot is badly placed and the location of the elevator does not correspond well in the real world, it is likely to fail as it is told to go through walls.

Currently a global map can be saved in memory, but due to odometry errors when building the map, it can be very inconsistent as seen previously in figure 13; a loop closure algorithm is a solution that will help when building a map. For better efficiency in all steps of the scenario, it is necessary to use a correct map of the environment for localization and planning, as it will help solve a few planning issues.

7.1.2 Localization

Odometry is a poor method for localization because of error accumulation and must be replaced in the future. On the alternative scenario where the door and elevator positions are recorded by *Loomo*, localization with odometry becomes problematic when the robot moves at a far distance from the elevator, as the doors will no longer be aligned and the robot will wait in front of a wall, thus failing at entering the elevator.

Segway Robotics has implemented a very competitive visual localization system for *Loomo* that can also help with map building, however, it is only available in the Java SDK and has not been ported to the C++ development kit. It may be possible to access Java functions from the C++ code but this has not been explored.

7.2 Path planning issues

7.2.1 Waiting in front of the elevator

During the step "waiting in front of the elevator", the robot is supposed to wait at a distance from the elevator door in order to avoid disturbing people exiting the elevator or other people also waiting. At the moment, the solution found to this issue is to set an intermediate goal on the door and the robot shouldn't move when the planner status isn't normal (path is blocked, goal inaccessible...). Unfortunately, depending on the angle at which the robot reaches the elevator and with no prior knowledge of the map, it sometimes does not locate the door with its sensors until it is too close and stops in front of the door. Furthermore, it does not know that there is only one entrance and may look around for another passage depending on the distance it can deviate from the path. A solution for

waiting that would work once a full map of the scene is available is to set an intermediate goal at the sides of the elevator entrance, with visibility on the door, then scan the scene, and set the next goal inside the elevator, where the robot now knows that there is no path leading inside if the door is closed and consequently, it will wait until it opens.

7.2.2 Entering and exiting the elevator

When exiting the elevator, the robot has already fully mapped the elevator. As a result, it knows that there is only one exit, however, when the path is blocked, either by the door or a person, the planner knows it cannot reach the goal and does not attempt to make the robot move. This is problematic because the robot needs to follow people that are exiting rather than wait until they are outside or that they move out of the way. This behavior is not human-aware, and is annoying to people because it is unpredictable and because time is wasted while waiting for the robot to exit. This over conservative waiting would also be problematic in the scenario where the robot enters the elevator and waits for everyone to enter before moving.

7.2.3 Planner failure

The main issue with the current planner is the inability to recover when the robot is close to walls, even when there is enough space to move and turn. The planner seems unable to make sharp turns which may be intentionally programmed to prevent the robot from turning towards obstacles outside the field of view, even though the depth camera can be controlled. As such, when the robot is close to a wall and needs to make a sharp turn to avoid it, the robot will remain static. An example of a robot that cannot move due to proximity with obstacles is visible in figure 16.

Another factor that contributes to the planner failure is the closest distance that the robot is allowed to approach obstacles (on the sides), which is set at 2cm only; the robot will avoid moving closer than this distance to obstacles and will stop all movement when under the threshold. This distance cannot be removed since it is necessary to add an offset from obstacles to ensure that they are avoided.

It must be noted that sometimes, the inability to recover is caused by the obstacle avoidance module that will prevent the robot from moving when it is close to obstacles. The path planner and obstacle avoidance modules need to be balanced perfectly in order to avoid all obstacles, while taking all paths that are spacious enough for the robot and allowing the robot to recover from closeness to obstacles. At the moment, the robot is not capable of balancing these tasks well enough in certain situations.

7.2.4 Obstacle avoidance

One of the issues in the obstacle avoidance module is that it can prevent the robot from moving as a precaution, even though there is enough space to move. For obstacle avoidance, only the ultrasonic sensor is used and it outputs a single value, giving no indication as to where the object is located. In such cases where the ultrasonic sensor outputs dangerous values because an object is close to the robot, but the object can be avoided, the robot will remain static.

Figure 16: Image of a robot unable to recover due to proximity with an obstacle. A slight clockwise rotation is enough to allow the robot to recover and enter the elevator.

Another issue with obstacle avoidance arises when the sensors are incapable of detecting obstacles in front of them. Indeed, the ultrasonic sensor needs a minimum range of 25cm to detect obstacles; anything closer is undetected by both the ultrasonic sensor and the depth camera. When the robot is closer to an obstacle than this threshold distance, either because it did not stop in time or because another object approached the robot, it will detect nothing and will start going forward and will collide with the obstacles.

7.2.5 Turning towards the door

When turning to face the door, the robot does not look around to see whether there are obstacles, nor does it use the planner. The function implemented simply makes the robot turn at a slow speed until it faces the direction of the door and is in no way human-aware. If the robot is close to an obstacle when turning, it will collide with it. This section of the framework should be taken care of by the human-aware planner, however, since the current planner is unable to give a specific orientation to the robot, it had to be taken care of manually.

7.2.6 Elevator position planning

The current resting position of the robot in the elevator is determined by calculating the distance of each pixel in the local map to the nearest obstacle pixel,

and choosing to stop at the furthest position from obstacles. This is not human-aware planning because there are many more variables than distance to take into account such as the amount of people entering the elevator or the space wasted by not resting close to a wall; the maximization of distance is by no means how a normal person would behave in similar situations. As such, it is necessary to use a planner that will give the robot a goal by using many more variables and a more complex obstacle map that includes people.

7.3 Human-awareness

7.3.1 Human-aware planner

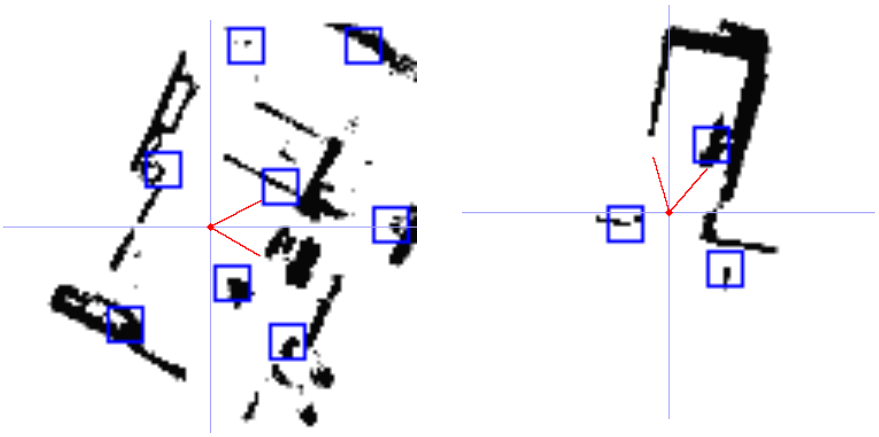
The current planner uses a basic algorithm that treats all obstacles similarly. It is a reasonable solution in static scenes but it is unable to take any social constraints into account and will have to be discarded for a better solution if we wish to achieve our goals. As the robot-human interaction is of a very high importance for us, it is necessary to detect people and modify the robot's behavior according to the actions of people.

The new human-aware planner we wish to create cannot be integrated with the current planner as it only takes waypoints as parameters, allowing very little flexibility, and we must be able to control many variables for movement such as speed, which is very important in human-aware navigation. Furthermore, the current planning module may get stuck when the robot is close to a wall as it cannot move forward safely and it is not allowed to move backwards and turn on itself, but as it can be inferred from human behavior, the new planner should know that it is acceptable to move backwards given prior knowledge of the scene and space, or with the required tilt in head orientation.

7.3.2 Person detection and global positioning

Currently there is no person detector implemented that is capable of running on the robot in real time, that also has a strong detection accuracy and that can place people on a 2D map. Since human-awareness depends on person tracking, it is a necessary requirement for our planner. The current person detector may be capable of running in real-time when sending images to an off-board GPU that detects people and sends the results back to *Loomo*, however, this has not been tested yet. The speed gain between a high end desktop GPU and the *Loomo* CPU should be at least tenfold, decreasing the processing time for a frame to under 100ms on the GPU. As this time is still too slow to run at 30 frames per second, it is necessary to use a tracker that can follow people between frames.

The other person detector from the C++ development kit for Loomo is much faster, but its drawback is in accuracy. As visible in figure 17a, this person detector has many false detection, particularly in the presence of clutter, and it is also possible that it fails at detecting people. The elevator scenario has the advantage of being clutter-free so the detector performs reasonably well (figure 17b) but some people can be overlooked easily when they are leaning on walls or when they are close to other big obstacle clusters. As a result, a more accurate person detector should be favored.



(a) Person detection in a cluttered office. There is only 1 person just below the robot, the rest is clutter such as a bin, a bag, tables, bad wall detection, other...

(b) Person detection in an elevator. The person is well detected but this is not always the case. Few errors: one is a previously detected post, the other is a wall that is not mapped well.

Figure 17: Person detection using the local obstacle map, obstacles labeled as people have a blue square.

8 Current performance evaluation

The performance results of the current implementation using the defined evaluation metrics are presented in figure 18. Metrics such as human comfort have not been evaluated since we have not conducted any surveys. Before doing the surveys, the human-aware planner should be implemented as it is the key solution to improving the human comfort metric. Evidently, the current version is still far from the performance we aim to achieve but most problems should be solvable with localization in a map and a human-aware planner.

The metrics that perform the best are obstacle avoidance and state observation because they are not directly dependant of the human-aware planner. The time metric, although it does not yet yield great results, is dependant on human-awareness for improvements, as the robot will learn to follow people, accelerating the speed of entering and exiting.

Human comfort and ethical sensitivity are the key metrics that are currently lacking in state-of-the-art systems, and these are what we aim to improve in our framework by creating a human-aware planner. As it is not implemented yet, they still yield poor results.

8.1 Score discussion

Obstacle avoidance is generally rated high, but because we are unable to detect objects that are closer than 25cm from the robot, causing collisions in this case, we are unable to give a perfect score. Also, in the situation where the robot is waiting inside the elevator, there is no obstacle avoidance while turning, so the robot may collide with nearby obstacles with a very weak force.

State observation is accurate since *Loomo* can scan the environment to detect

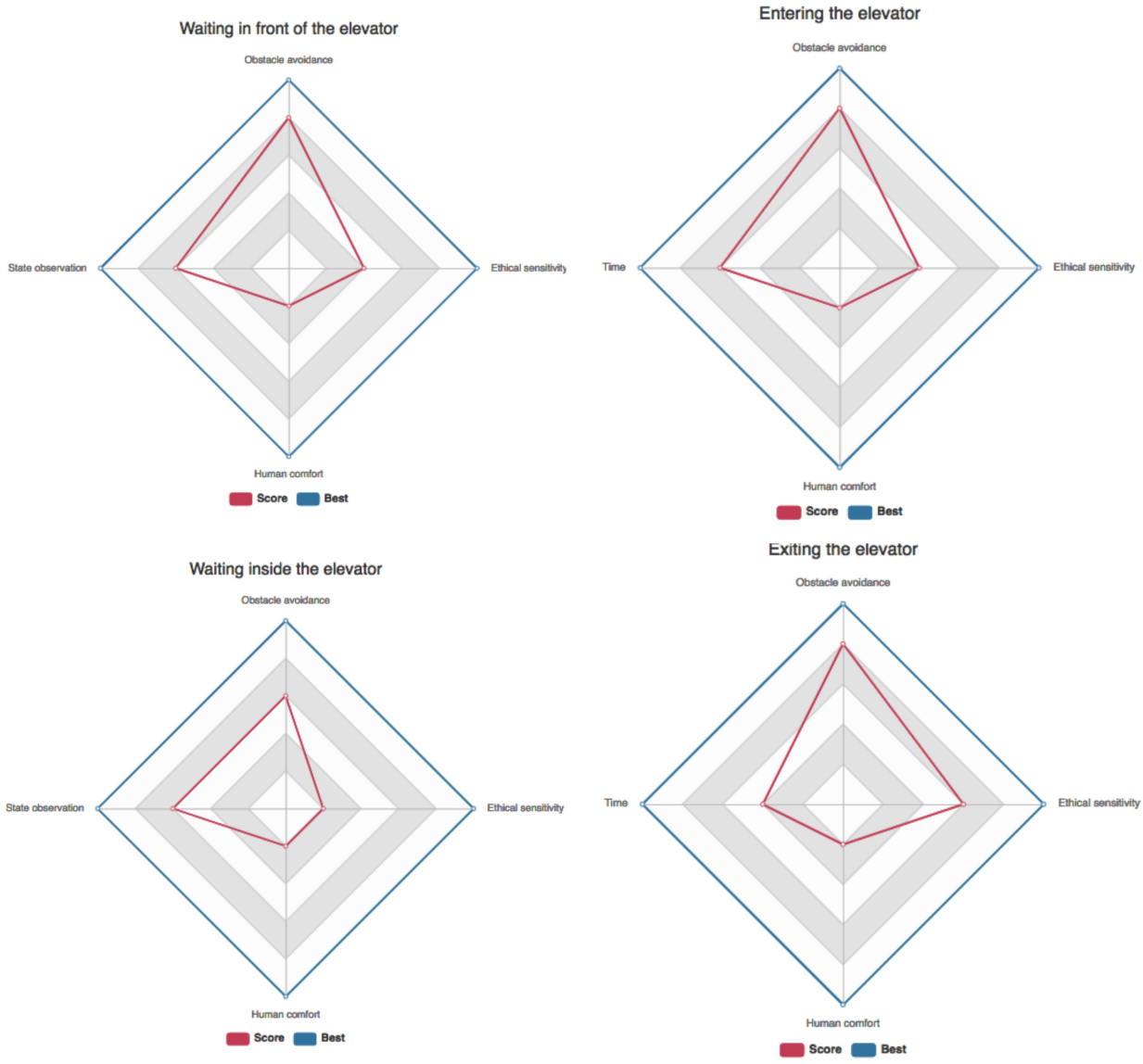


Figure 18: Performance evaluation in the 4 elevator tasks. Performance ranges from 1 to 5, where human comfort is rated as 1 because the survey has not been conducted. Depending on the scenario, the tasks use different metrics: the movement focused tasks use the time metric, while the waiting focused tasks use the state observation metric.

what is happening around, however, scanning is slow and the robot can take a while to detect an open door, which is inefficient.

Finally, although the pace of the robot's movement is good, there are two reasons for the low score of the time metric. Firstly, the robot can be slow at entering and exiting the elevator if a person is in front because it waits until

a full path is clear before moving. Secondly, *Loomo* can be overly cautious when detecting obstacles. As a result, *Loomo* can get stuck for a short while, particularly in tight spaces such as in the elevator.

9 Conclusions and future work

9.1 Future work

Before a new human-aware planner can be used, it is necessary to detect people with *Loomo* and place them on a 2D map constructed around the robot as our framework does not work if we cannot track people in real-time. Next, it is important to gather data of people in elevators, annotate the people with position and orientation at each frame and train a model with the data. This is a necessary step to create our human-aware path planning module, and it is the key distinction of our approach. Finally, there is no map of the scene available for the robot yet, but it would be very useful for localization and path planning, and would help to improve current performance. This is not the most pressing step but would offer more flexibility, autonomy and realism in our scenario and should be considered in the near future. Once a global map is available, it would be wise to use an algorithm capable of self-localization as we rely solely odometry at the moment.

9.2 Conclusion

The objectives of this project were to propose a framework that would enable *Loomo* to share an elevator with people while respecting social constraints, and implement a live demo for *Loomo*.

Ultimately, we have defined a new framework that will help *Loomo* achieve human awareness when sharing elevators with people, which is a topic that is still unexplored, but that is imperative to increase robot acceptance in public spaces. An evaluation protocol has been outlined to judge the performance in the four important steps of elevator usage and methods are suggested to resolve the issues of this implementation.

The current solution for *Loomo* is able to enter elevators relatively fast and without collisions, it can choose to position itself according to space, but it is not human-aware yet. There is still much work to be done with the human-aware planner before our objectives are met but the implementation serves as a baseline to work upon and improve.

References

[1] Alexandre Robicquet, Amir Sadeghian, Alexandre Alahi, and Silvio Savarese. Learning social etiquette: Human trajectory understanding in crowded scenes. In *European conference on computer vision*, pages 549–565. Springer, 2016.

[2] Matthias Luber, Luciano Spinello, Jens Silva, and Kai O Arras. Socially-aware robot navigation: A learning approach. In *Intelligent robots and sys-*

tems (IROS), 2012 IEEE/RSJ international conference on, pages 902–907. IEEE, 2012.

[3] Beomjoon Kim and Joelle Pineau. Socially adaptive path planning in human environments using inverse reinforcement learning. *International Journal of Social Robotics*, 8(1):51–66, 2016.

[4] Jeong-Gwan Kang, Su-Yong An, and Se-Young Oh. Navigation strategy for the service robot in the elevator environment. In *Control, Automation and Systems, 2007. ICCAS'07. International Conference on*, pages 1092–1097. IEEE, 2007.

[5] Jeong-Gwan Kang, Su-Yong An, Won-Seok Choi, and Se-Young Oh. Recognition and path planning strategy for autonomous navigation in the elevator environment. *International Journal of Control, Automation and Systems*, 8(4):808–821, 2010.

[6] Aethon. https://aethon.com/. Accessed: 2018-10-29.

[7] Thibault Kruse, Amit Kumar Pandey, Rachid Alami, and Alexandra Kirsch. Human-aware robot navigation: A survey. *Robotics and Autonomous Systems*, 61(12):1726–1743, 2013.

Annexes

A Code links

- Elevator code: `https://github.com/vita-epfl/loomo-algodev-mirror/tree/paul/elevator/algo_app/src/main/jni/app_elevator`
- Person detector code: `https://github.com/vita-epfl/loomo-algodev-mirror/tree/paul/elevator/algo_app/src/main/jni/app_tensorflow_person_detector`

B *Loomo* tutorial

Getting started with Loomo

VITA, EPFL

February 2019

1 Installation

1. Install *Android Studio*: `https://developer.android.com/studio/install`
2. Import project: File > New > Project from Version Control > Git > https://github.com/segway-robotics/loomo-algodev/tree/yuejiang/crowd
3. Add 3rd party librairies (follow *README.md* instructions).

Here is a list of possible issues:

- Missing NDK: File > Project Structure > SDK Location > Android NDK Location > Install
- *CMake* missing error: click on install *CMake* in *Android Studio* 'sync' window (easier than installing *CMake* before building)
- *Python* missing error: install *Python3* and add it to environment path
- *Ndk* toolchains error: if *mips64* is missing, download an archived version of *Ndk* that contains *mips64*, such as android-ndk-r16b

2 Development Kit

2.1 Classes

2.1.1 AlgoBase

This is the base class for the algorithms that should run on *Loomo*. Here are defined the important functions common to most projects such as:

1. *start()*,
2. *run()*,
3. *exit()*.

Most projects will need to create a separate class that is derived from this one and where the specific functionalities will be defined.

2.1.2 Algorithm class

The main algorithm shall be written in a class that inherits from the *AlgoBase* class. Here the parameters unique to the application are initialized and the main loop must be defined in the following function:

- *step()*.

This function is called at every iteration.

The C++ development kit for *Loomo* contains several example applications for:

- displaying sensor information,
- local mapping,
- object recognition.

Each of these applications are defined in their own source and header files where the main loop is defined in the *step()* function from the source file. A call to the *start()* function from the base class (*AlgoBase*) starts the loop and most applications should take inspiration from these examples.

2.1.3 RawData

The *RawData* class is used to retrieve sensor information from the robot and send commands to the actuators. Only one instance of this class must be created where all calls will be done. This instance shall be sent to the algorithm class as a parameter when it is instantiated.

As there is no official documentation for the functions in this class, it is necessary to refer to the *RawData.h* file[1] for information on the functions available and their parameters.

2.2 Launching an application

2.2.1 Connecting to *Loomo*

A computer can connect to the *Loomo* via the *Android Debug Bridge* (adb) command-line tool if they are on the same Wi-Fi network. To do this, it is necessary to obtain the IP address of the robot (Settings > System > Status > IP address). Once connected, it is possible to upload an application to the robot and receive debug information. Below are a few example commands to connect to a device and check for status, where the IP address must be replaced by the *Loomo* IP address.

```
adb connect 128.179.176.227
adb devices
adb shell
```

[1] '*dependency/algobaseinclude/interface/RawData.h*'

```
adb push / pull
adb disconnect 128.179.176.227
```

For more information about adb commands, please visit `https://developer.android.com/studio/command-line/adb`

2.2.2 *Loomo* screen

When running an application from the C++ development kit for *Loomo*, a screen as seen in figure 1 will appear on the *Loomo*. Here, the application running is the *AlgoTest* app that displays sensor information, but when launched, only the buttons on the top are visible and they represent several functions that can be started.

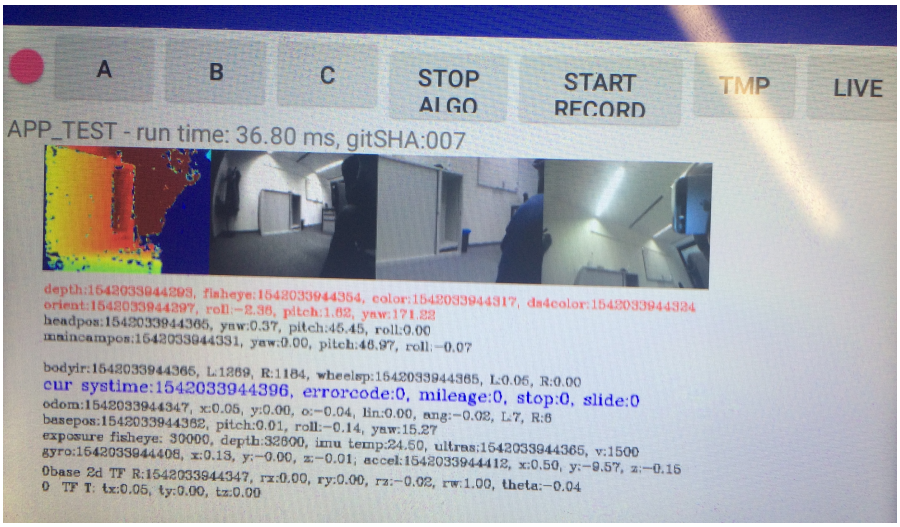


Figure 1: Loomo screen example running *AlgoTest* app from the development kit

2.2.3 Structure

The buttons on the *Loomo* screen are functions defined in '*core/AlgoImpl.cpp*'. To begin launching the application, the user must press the red button at the top left of figure 1 that will also run the *testAlgoStart(...)* function:
START ALGO/STOP ALGO buttons.
This function is the start of the main loop; here, an instance of *RawData* and an instance of the application algorithm class are created and the loop is started on a new thread with a call to the *start()* function.

Other functions can also be called via the *Loomo* screen (*funcA, funcB, funcC*) and can be customized by the user for each application in *AlgoImpl.cpp*.

Also in this file, it is necessary to define the cameras that must be used by the application (*USE_DEPTH*, *USE_FISHEYE*...).

2.2.4 CMakeLists.txt

The build process of the C++ code is managed by *CMake*, hence, the presence of the '*CMakeLists.txt*' file[2] in the project. Each application must be included in this file for compilation, where the macros must be modified to switch between applications. Also, for each application, it is necessary to list the libraries that must be used such as *Tensorflow* and *LocalMapping*.

2.3 External libraries

Third party libraries such as *OpenCV* and *Tensorflow* are included in the project and can be easily used. An example of a *Tensorflow* application that also uses *OpenCV* functions is included in the development kit.

2.4 ROS (Robot Operating System)

With this kit, you can build ROS-like threads. As such, some features from ROS can be found such as ROS request messages. An example of this is the *tf_message*[3] that is based on ROS's '*geometry_msgs::TransformedStamped*' messages. Here a user can send a request message to obtain the position of a robot part in a specific coordinate frame. Messages are received in a ROS format that use quaternions and sometimes must be converted, such as for 2D pose with the *tfmsgTo2DPose()* function. Using these messages is an efficient way to obtain the position of the robot parts such as the base or the head.

2.5 Movement

2.5.1 Robot base

Moving the robot's base is done with a call to:

- *ExecuteCmd(...)*.

This function takes in as parameters the translational velocity, the rotational velocity and the current timestamp. Similarly, stopping the robot is done with the same call but with zero-valued velocities and it is possible to use negative values for movement in the opposite direction.

It can be useful to use the robot odometry or other cues to stop the robot, however, given that the primary objective of the robot is to remain balanced, there will always be a delay while it tilts to adjust balance before stopping or moving.

An example of the use of the function is implemented in '*AlgoTest.cpp*' where the movement is started by pressing the B function button on the robot.

[2] '*algo_app/src/main/jni/CMakeLists.txt*'
[3] defined in '*dependancy/algobaseInclude/interface/tf_data_type.h*'

2.5.2 Robot head

The robot head is able to turn 150° each way for yaw and 180° to -90° for pitch. The head can be controlled in position or in speed for both yaw and pitch using the *ExecuteHeadMode()* function to change between speed and position control, and both *ExecuteHeadSpeed()* and *ExecuteHeadPos()* for the actual movement. With the position control, we are unable to control the head speed and the head moves slowly when close to its desired position due to the controller implemented, which is problematic when turning the base since the head is not looking in the correct direction to detect obstacles. As a result, it is more reliable to control the head by speed, while adding a certain position offset from the base using a PD controller, to look ahead in a good direction for object detection. A simple proportional controller cannot be used as it will accumulate error.

2.6 Socket-based computation offloading

The example application '*socket*' demonstrates how to send information between a client and a server, useful to process computationally heavy algorithms on a device that does not have the hardware limitations of the robot. It is also recommended to build fast and flexible prototypes using the robot-cloud socket.

Here the server (robot) sends the position of people detected, encoded as floats, to a client (cloud computer) that can track people with the information and send back the results to the robot.

On the cloud side[4], it is necessary to verify that the correct IP address of the robot is entered and that the executable is built again.

The example sends floats but there are several functions for sending chars or images as well, however, sending heavy loads over the network such as multiple images might come with a longer delay.

3 Basic walkthrough

3.1 Algo Test

AppTest is a sample application that extracts, displays and tests all basic elements of a robot such as sensors, robot joint positioning, camera inputs, odometry...

3.2 Algo Local Mapping

AlgoLocalMapping is a sample application that builds and maintains an occupancy map with sensory measurements. A local map centered at the robot frame can be queried by calling the function *getDepthMap()* or *getDepthMapWithFrontUltrasonic()*. For more details, refer to *LocalMapping.h* file.

[4] `https://github.com/vita-epfl/socket-loomo`, private link for now. Use *git checkout tags/cpp_standalone* to switch to an easy to run version

A local map can be initialized with a single call to the following example function:

```
bool AppName::initLocalMapping()
{
    if(m_p_local_mapping) {
        delete m_p_local_mapping;
        m_p_local_mapping = NULL;
    }
    float mapsize = 6.0;
    float m_map_resolution = 0.05;
    m_p_local_mapping = new ninebot_algo::local_mapping::
        LocalMapping(mapsize, m_map_resolution);
    if(m_p_local_mapping == NULL) return false;

    CalibrationInfoDS4T calib;
    mRawDataInterface->getCalibrationDS4T(calib);
    float fx = calib.depth.focalLengthX;
    float fy = calib.depth.focalLengthY;
    float px = calib.depth.pricipalPointX;
    float py = calib.depth.pricipalPointY;

    m_p_local_mapping->setLidarRange(5.0);
    m_p_local_mapping->setLidarMapParams(0.6, true);
    m_p_local_mapping->setDepthCameraParams(px, py, fx, fy, 1000);
    m_p_local_mapping->setDepthRange(3.5, 0.35, 0.9, 0.1);
    m_p_local_mapping->setDepthMapParams(1.0, 10, false, -1);
    m_p_local_mapping->setUltrasonicRange(0.5);

    m_map_width =  mapsize / m_map_resolution;
    m_map_height = mapsize / m_map_resolution;

    return true;
}
```

3.3 Algo Tensorflow

AlgoTensorflow presents an example that uses a *Tensorflow* model with all processing done with the on-board CPU. The pre-trained model[5] needs to be downloaded online and pushed to the corresponding folder of the robot.

3.4 Algo Socket

AlgoSocket demonstrates a recommended method for flexible prototyping and computation offloading. Efficient components such as constructing a local map

[5] *inception_v3_2016_08_28_frozen.pb*

and detecting blobs of persons are carried out on the robot, whereas core algorithms for tracking and planning are offloaded to the cloud. The socket latency between a robot and a cloud computer through the campus wifi is typically around 50 ms, fast enough for most real-time experiments.

4 Advanced examples

4.1 Running a multi-person detector

Modern person detectors are almost exclusively deep convolution neural networks as they yield more precise and consistent results than past methods. Existing models can be found online, however, since CNN-based object detectors are capable of classifying multiple objects, it is more common to find object detectors that happen to be capable of detecting people, rather than simple person detectors. As such, the model chosen for this example is capable of detecting 182 objects, including humans. The model, named "*ssd_mobilenet_v1_coco*"[6], is trained on the COCO dataset and is one of the most computationally efficient, while remaining precise enough for the *Loomo*, that only needs to locate people at a close range ($\approx$ 5m). However, this model does not use the depth information available for detection, and it usually is not necessary for person detection, but it can be useful for global positioning of a person with respect to the robot.

Other algorithms[7] using depth information for multi-person detection and tracking exist, but they are complicated to integrate (requiring full installation of ROS) and/or have limitations such as requiring manual user inputs, a specific height of the robot camera for better results or they require extra sensors (2D lasers) not available to *Loomo*. As such, a *Tensorflow* model is used for its simpler implementation, fast computation and precision in the scenario of a robot close to the ground locating people in a small radius.

The multi-person detector code[8] is inspired from the *Tensorflow* sample application, where the model is changed as well as its inputs/outputs and a few operations resulting from the different model. It can detect people at a range of approximately 5 meters from the robot depending on their size and runs in 0.9-1s on the *Loomo* CPU (with 640*480 images). This model is capable of detecting people in many different postures or with isolated body parts, such as having only legs visible, or only arms.

[6] `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md`

[7] `https://github.com/lagadic/pepper_launch/wiki/People-detection-using-RGBD`

[8] `https://github.com/vita-epfl/loomo-algodev-mirror/tree/paul/elevator/algo_app/src/main/jni/app_tensorflow_person_detector`

4.2 Local mapping and path planning

4.2.1 Local mapping

Local mapping is performed using the *LocalMapping* class[9]. Here the user can create and update a local occupancy map around the robot using the depth camera only (*getDepthMap(...)*) or a combination of depth camera and ultrasonic sensors (*getDepthMapWithFrontUltrasonic(...)*) as performed in the example application. The ultrasonic sensors only help to detect obstacles at a short distance directly in front of the robot. To create a new map, the user can call *clearMap()* to remove all prior knowledge, otherwise the map is updated with the new sensor information in the robot field of view. The map created is typically a 6x6m image due to sensor range and is a probability map of encountering obstacles. Some limitations of the map include the robot's field of view which can require turning the head to obtain information on the sides, or the detection range for close objects which is of 25cm for the ultrasonic sensor, and there is also an inability to detect very small objects on the ground that can obstruct robot movement, such as the tip of feet, because the depth camera and ultrasonic sensors cannot orient themselves towards the ground.

4.2.2 Path planning and movement

In terms of path planning and movement, the *ModelBasedPlanner* class[10] can be used. This class allows the user to give the desired goal location for the robot (*reset_global_reference(...)* function) as a vector of x-y checkpoints and the class will take care of the path planning, movement and obstacle avoidance for the robot. However, it might be necessary to create multiple waypoints along the path as the class and its functions may not always know how to get to a goal far away with no clear path. There are a few issues with this class. For example, the planner does not allow the robot to make sharp turns when close to obstacle, which is sometimes necessary to avoid getting stuck. Furthermore, if the robot is quite close to an obstacle, it stops moving even if there is sufficient space because the planner stops all movement when it is very close to obstacles, which can be over-conservative planning in certain situations.

The functions use map coordinates that change at every new run when using local mapping because they depend on the starting position and orientation, so it is best to convert the robot pose and orientation to the local map coordinates before using the occupancy map and attempting to set a goal location.

A function implemented for movement is *safeControl(...)*, that takes as inputs the linear and rotational velocities for the robot. This function not only makes the robot move, but has also been modified to control the robot head[11] so that it turns to have objects on the path in the field of view of the depth camera, as there are many collisions otherwise. Furthermore, the function contains

[9] 'dependancy/algo_include/general/LocalMapping.h'

[10] 'dependancy/algo_include/general/model_based_planner.h'

[11] Only located in the elevator scenario application: *https://github.com/vita-epfl/loomo-algodev-mirror/tree/paul/elevator/algo_app/src/main/jni/app_elevator*

a part that will prevent the robot from moving as part of obstacle avoidance by using the current robot velocity and ultrasonic sensors to detect obstacles and stop before it is too late.

4.2.3 Map format and information

The local map constructed by the robot is in an array format visible in figure 2.

Figure 2: Local mapping example in an office.

The local map can be modified in size but is a 2D map, typically 6m x 6m and is constructed with the robot at the origin (center of the map). The map represents the probability of encountering obstacles where obstacles in the map appear in gray and the values of each pixel can range from 0 to 255. It is best to loop through all pixels to acquire information, where we can typically consider obstacles to be pixels with a value superior to 20 as seen in the following code:

```
for (int i = 0; i < localMap.rows; ++i) {
    for (int j = 0; j < localMap.cols; ++j) {
        if(localMap.at<uchar>(i,j) > 20){
            // obstacle
        }
        else {
            // not obstacle
        }
    }
}
```

The local map is a crop of a bigger fixed map that is of size 30m x 30m. However, the bigger the map, the more it is falsified by the accumulation of odometry error. The fixed map can be acquired with the *getFixMap(...)* function and an example map is visible in figure 3. In this map, we can observe

walls that become larger and unaligned with respect to the real world due to the odometry error. This map was constructed with a single carefully planned passage, in a straight line, with a 90° turn, but with more chaotic movements, the map can become completely distorted and must be corrected with loop closure algorithms.

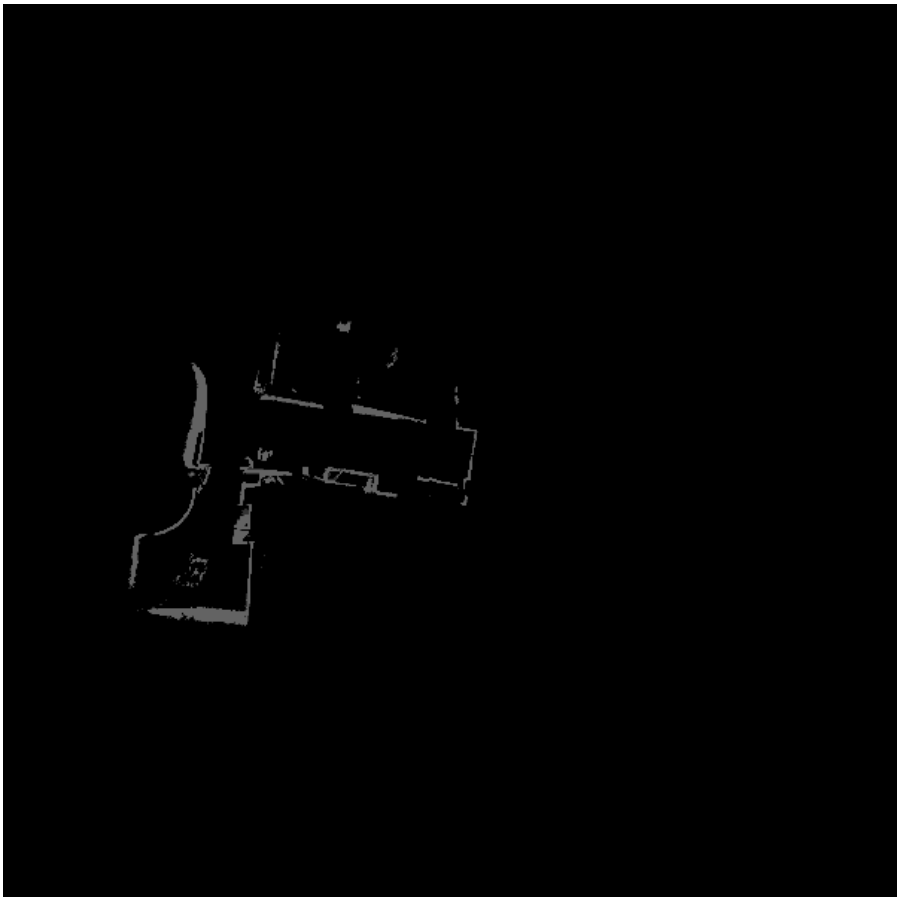


Figure 3: Fixed map example.

The maps can be saved in the robot permanent memory as png files with a single call to the *cv::imwrite(...)* function. At the moment, there is no code to localize in a map in C++; a competitive visual localization algorithm is implemented for *Loomo* in the base Java development kit but has not yet been ported to C++. In the meantime, Monte-Carlo localization is a good alternative but must be implemented.