Design of a mobile robot for navigation and sample collection

Ignat Georgiev

June 21, 2020

Abstract

We have developed a differential drive robot capable of autonomously localising, navigating and sample collection with a known environment. It is based on the popular Turtlebot3 robot and uses lidar data in combination with prior motion estimates to localise within a known environment. It then uses an A* path planning and a waypoint follower to traverse the environment. The robot is also equipped with a robotic arm which it uses to press buttons, clear paths and collect rock samples using an inverse kinematics based controller. All of these features are packaged together to create a modular and scalable solution which can be applied to different environments.

1 Introduction

Robots are one of the most fascinating machines that humans have ever developed. They are complex machines which involve the integration of many different bodies of knowledge - dynamics, kinematics, artificial intelligence, computer vision, control theory, and many more. This makes robotics as interdisciplinary a field as there can be. Having mastered stationary robotic arms in the past decades, interests have shifted towards a more interesting, albeit more difficult, topic - mobile autonomous robots. These captivating machines have raised hopes of exploring extraterrestrial planets (Figure 1), replacing humans workers in dangerous environments and offering assistance to humans with laborious tasks [1].

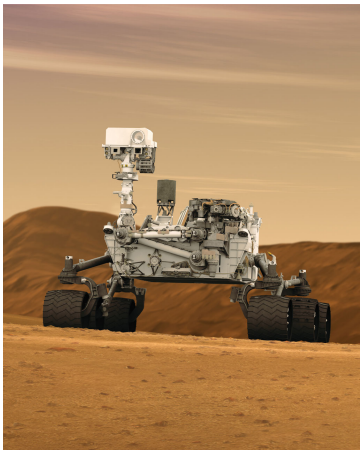


Figure 1: Curiosity, the NASA Mars Rover.

Despite many advancements in recent years, fully autonomous robots are still a few decades away from entering day-to-day human life. As such most piratical implementation of mobile robots are done in limited lab environments with the hopes of advancing the robotics field. As such, this report aims to showcase the design of a simple wheeled mobile robot tasked with navigating autonomously around a known environment, interacting with objects and collecting rock samples with a robotic arm. This was done purely for educational purposes in collaboration with Alex Roy as part of the Robotics Science and Systems course at the University of Edinburgh and as such is focused more closely on the software.

In more detail, the robot operates in the environment shown in Figure 2 and is tasked with:

1. Navigating to the button and pressing it.
2. Navigating to the boxes and clearing them away from the path. The box position is randomly chosen between two possible locations.

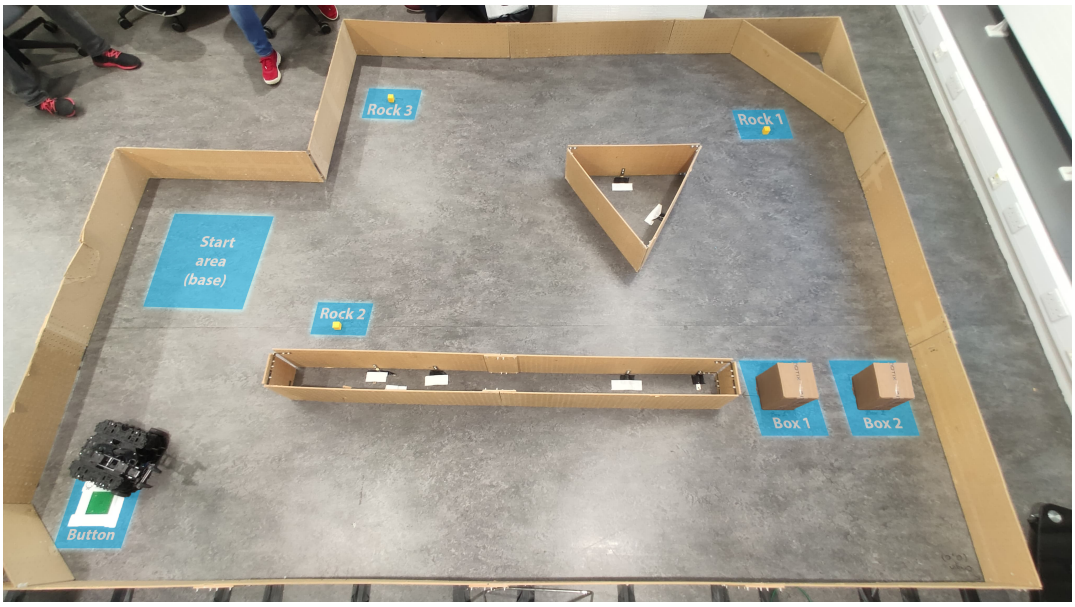


Figure 2: The robot environment

3. Navigating to one of the predefined sample collection points and picking up a rock sample.
4. Navigating back to its base in less than 5 minutes since the start.

Note that due to the scope and short time span of the project, computer vision has not been used and instead has been accounted for by predefined positions of objects.

2 Robot design

2.1 The robot

The robotic platform supplied for this report was the Turtlebot3 which is a small and affordable differential drive robot. It is equipped with 2 DC Dynamixel XM430 motors which can be controlled independently and provide accurate wheel odometry. Additionally, the robot is equipped with a 9-axis IMU, Raspberry Pi Camera and an LDS-01 360°Lidar. For computation, the robot uses a Raspberry Pi 3 Model B and an OpenCR board which is an embedded ARM Cortex M7 controller. Additionally, the robot is equipped with a 4 degree of freedom Interbotix PX100 robot arm which is controlled with servo motors and used for manipulation and grasping objects with the attached gripper.

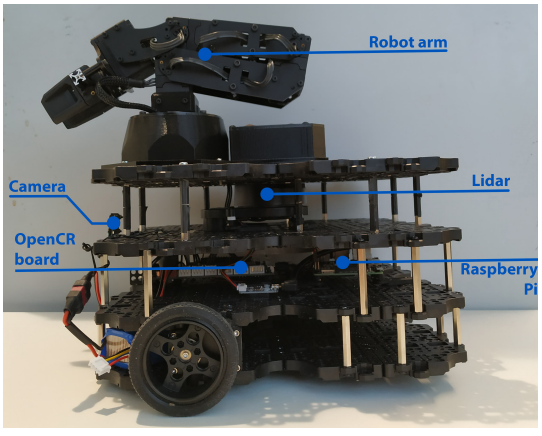


Figure 3: Turtlebot3

2.2 Software concept

Overall, our robot followed the standard see-think-act mobile robot cycle and was designed

with modularity and scalability in mind. The 3 major building blocks of the system being:

- **Particle filter localization** which requires a predefined discrete map of the environment and works with the motion estimate and lidar data from the robot.
- **Navigation** consisting of an A* pathfinder and a PID waypoint controller which sends motion commands to the robot directly.
- **Arm controller** which executes predefined arm movements and computes the arm joint angles with inverse kinematics.

These modules are grouped using a **High-level State Machine** which boots up the whole system, waits for all required modules to start, distributes tasks from a user-defined *action list* and waits for their completion. Tasks consist of:

- **move action** - move to a given 2D pose - x, y coordinates and orientation.
- **arm action** - move arm end effector to a given 3D position (x,y,z) position. Additionally, a movement type can be provided to alter the execution of the task.

The architecture of this system can be seen in Figure 4. Note that the feedback from modules to the state machine have been omitted for clarity.

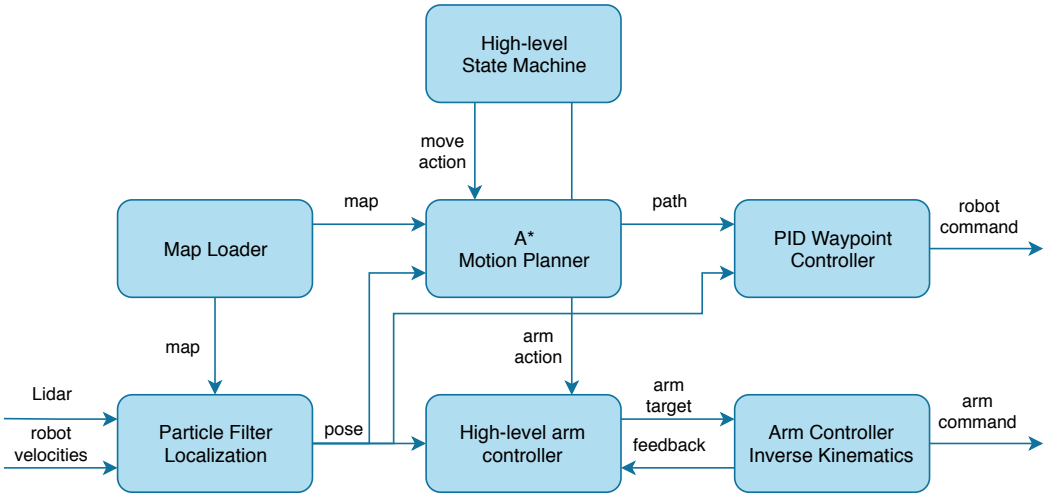


Figure 4: Software architecture

This whole system runs on the Raspberry Pi on top of ROS Kinetic and Debian Stretch [2]. Although, we were not allowed to use existing ROS packages, using ROS allowed us to run our system distributed, troubleshoot the system effectively and visualise data. All of the different modules seen in Figure 4, except the state machine and the map loader, have been implemented as standalone C++ ROS nodes as they are required to run in approximately real-time.

To interface with the robot, we have used the provided `turtlebot3_core` ROS node which runs on the OpenCR board in real-time and is connected to the Raspberry Pi over serial USB. This node provides fused position and velocity information from the wheel odometry and IMU, Lidar data and control interfaces for the robot itself and the arm.

2.3 Simulation

Although the robot is robust and easy to use, real-world experiments are still time-consuming and troublesome due environment availability, battery life and hardware issues. For those reasons, we

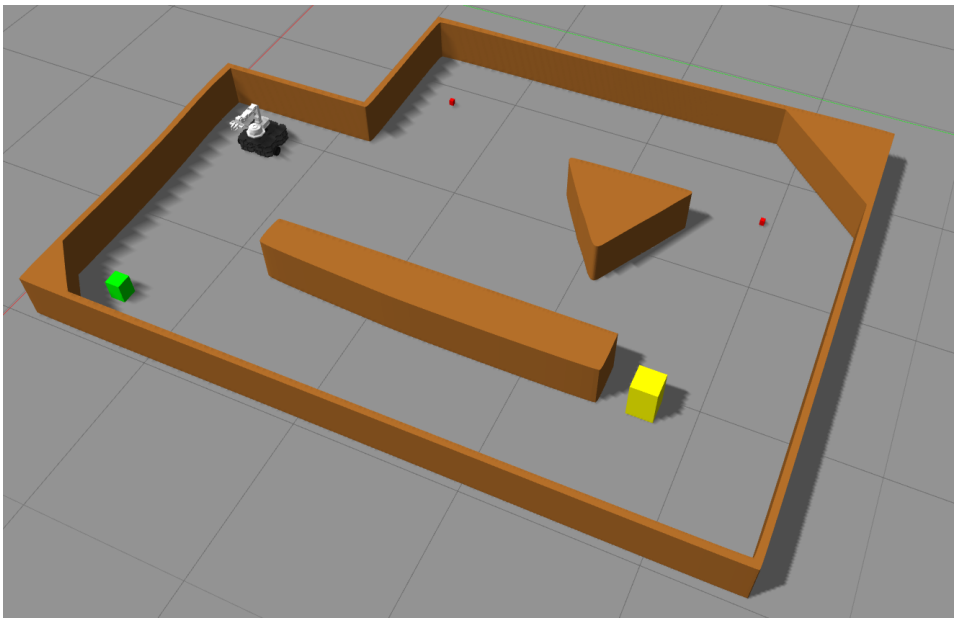


Figure 5: Gazebo simulation

decided to create a Gazebo simulation [3] of the task and only test the robot in the real world if it achieves the desired behaviour in simulation.

The simulation was built with the existing tools available in Gazebo and the open-source turtlebot3 plugin which simulates the robot, its sensors and interfaces. However, due to the non-traditional pairing of the robot and the Interbotix robot arm, no easy method was found of simulating the arm and instead, we opted to build it from the ground up. As the arm use servos, we chose to use PID position controllers from the ROS `position_controllers` and `controller_manager` packages to simulate the arm joints. The arm was then dynamically and visually modelled from CAD files and specifications available from the manufacturers, resulting in an accurate representation of the arm. Finally, the arm controllers in the simulation were tuned to match the transfer function of the arm in the real world, see Table 1. However, we could not simulate the gripper behaviour in Gazebo due to no existing functionality for it and implementing that ourselves, seemed out of the scope of the project.

Joint	P	I	D
waist	100.0	5.0	0.5
shoulder	500.0	10.0	0.5
elbow	300.0	10.0	0.0
wrist	100.0	3.0	0.0

Table 1: Arm controller gains in simulation

The final part of the simulation is the environment in which the robot is said to operate. To achieve that we created simple 3D meshes in Blender based on the specifications of the task and integrated them into our Gazebo simulation using sensible dynamical properties. However, doing this for the walls of the map proved to be a bad idea as the specifications of the environment did not match the environment seen in Figure 2. To address that issue we created a new 3D model of the environment based on the 2D discrete map used in the localisation module. The other objects in the environment were modelled as boxes using the specifications of the course. The final Gazebo simulation can be seen below in Figure 5 and was used to fully simulate the task described in this report except for the rock collection part.

2.4 Localisation

For the localisation of the robot, we chose to implement a particle filter as it is easy to develop, scalable and multi-modal. It is used only for estimating the pose of the robot which consist of its x, y positions and orientation - $\mathbf{x} = [x, y, \theta]$. To provide spatial information we chose to have a static discrete map instead of developing a full SLAM algorithm. Although the performance of the SLAM might have been superior, the complexity did not justify the additional development time.

Algorithm 1: Particle filter

N: Number of particles
$M(\cdot)$: motion model; m motion sample
$Z(\cdot)$: measurement model; z measurement sample

Given: starting position $\mathbf{x}_0$
Given: starting uncertainty σ_0

X_0 = initialise_particles($\mathbf{x}_0$, σ_0)

for *new pair* $[m_t, z_t]$ **do**
  for $n = 1$ *to* N *particles* **do**
    $x_t^{[n]} = M(m_t, x_{t-1}^{[m]})$
    $w_t^{[n]} = Z(z_t, x_t^{[n]})$
    add particle $[x_t^{[n]}, w_t^{[n]}]$ to $\bar{X}_t$
  for $n = 1$ *to* N *particles* **do**
    draw i with probability w_t^i
    add $x_t^{[i]}$ to X_t

The pseudocode of the algorithm can be seen in Algorithm 1

For the motion model (Algorithm 2), we explicitly decided to use the feedback from the robot instead of the commands sent to it as the feedback is more inherently a more realistic estimate and naturally bound to the capabilities of the robot. However, the robot does not inherently provide motion uncertainties, which are vital to the motion model. To address that issue we collected data from the robot while driving it manually in different circumstances and estimated the noise to be $\sigma_x = 0.00246$ and $\sigma_\theta = 0.02175$. During these experiments, it was also noticed that the motion estimation from the `turtlebot3_core` also gave erroneous measurements with hue values (eg. 175 rad/s rotational speed), which put off the particle filter and in some cases made it fail. To protect for such cases, we added filtering for the velocity limits of the robot.

Finally, for the measurement model $Z(\cdot)$ we decided to use a likelihood-field model instead of a mathematically formulated model. This decision simplified the tuning of the particle filter and offloaded some of the computation. The image of the map can be seen in Appendix 7.

Algorithm 2: Motion model

Given: $\mathbf{x}_{t-1} = [x_{t-1}, y_{t-1}, \theta_{t-1}]$: prior position estimate
Given: $m = [\dot{x}_t, \dot{\theta}_t]$ motion estimate of forward velocity and yaw rate
Given: σ_x and σ_θ motion uncertainty

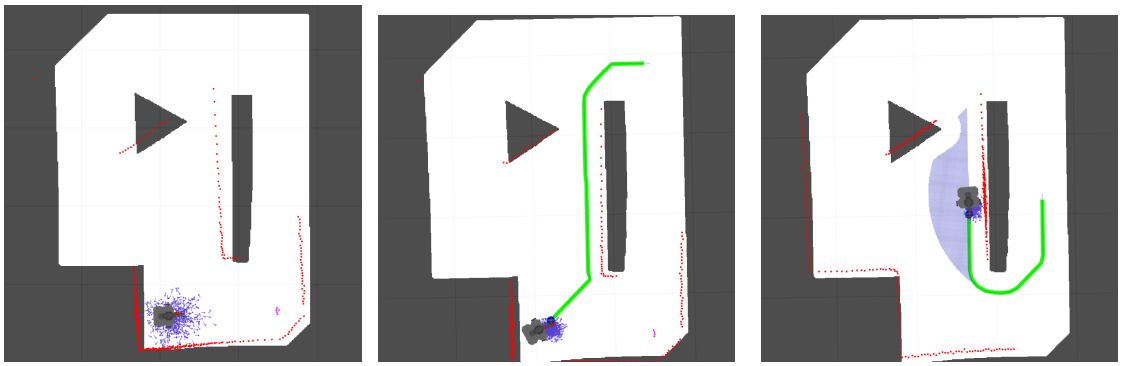
$x_t = x_{t-1} + (\dot{x}_{t-1} + \mathcal{N}(0, \sigma_x)) \cdot \cos\theta_{t-1}$
$y_t = y_{t-1} + (\dot{x}_{t-1} + \mathcal{N}(0, \sigma_x)) \cdot \sin\theta_{t-1}$
$\theta_t = \theta_{t-1} + \dot{\theta}_{t-1} + \mathcal{N}(0, \sigma_\theta)$
$\mathbf{x}_t = [x_t, y_t, \theta_t]$

return $\boldsymbol{x}_t$

In the end, the localisation algorithm ran well and reliably estimated the position of the robot. Some fine-tuning was necessary to guarantee real-time performance. We decided to run the algorithm at 10Hz with 250 particles using every other lidar hit. Additionally, we also added a moving average pose smoothing at the output of the localisation which stabilised the estimates of the algorithm.

2.5 Navigation

The navigation stack in this report can be split into two stages: *path planning* and *waypoint following*. As per Figure 4, we use a discrete A* path planning algorithm [4] which plans a path for the robot to follow based on a target position, current position and a discrete map of the environment which encodes whether there is an obstacle in a given position. This method was chosen due to to its simple implementation and robustness. The algorithm uses $f = g + h$ to evaluate the *score* of a given position on the map, where g gives how close to the goal we are and h is a combined function for other costs



(a) Starting position (b) Simple path planning (c) Complex path planning

Figure 6: A* path planning visualisation in RViz. Green path is the final path found. Blue overlay is the explored nodes. The other blue points in close proximity to the robot are the particles of the particle filter.

of the path. For the task in this report, g was set to be the Euclidean distance to the target position and f was set to be high if the explored position would result in the robot crashing into a wall. For the development of the algorithm, it was important to visualise its operation for debugging purposes, therefore RViz-compatible visualisations have been made as seen in Figure 6. The algorithm was developed as a C++ ROS node which allowed it to execute plan paths swiftly. In the worst case for the task (Figure 6c), a path was made in 210ms onboard the robot.

After a path is found, waypoints are calculated by taking every 10th point of the discrete path found by the path planner and fed to the waypoint follower, which uses a simple PID control loop for the steering and controls the forward velocity in an empirically derived way which allows the robot to follow the waypoints in a sequential order without significant deviation. The full control algorithm can be seen in Algorithm 3. The parameters used for this report were empirically tuned for a good balance between fast and smooth control - $K_P = 2.5$, $K_I = 0.0$, $K_D = 0.4$, $\dot{x}_{desired} = 0.1m/s$.

2.6 Arm control

To control the arm, we chose to choose one of the most tried and true methods - inverse kinematics (IK). Before attempting anything complex, we had the important insight that the arm can be seen as a 3-link planar manipulator on a rotating base. Due to the relative simplicity of this, we chose to do *Analytical IK* instead of the more common iterative optimisation IK. The solution for this was based on [5] and adopted to the use case of this project. If the desired end-effector position is 3D point and an orientation for the wrist with variables $[x, y, z, \phi]$, then we can split the solution into stages. The first "shoulder" joint angle is trivially computed to align the arm to the target position, then the rest of the system can be constrained to a single plane and solved analytically as per [5]. The final analytical solution is shown below:

Algorithm 3: Control method

K_P, K_I, K_D: steering parameters
$\dot{x}_{desired}$: desired forward speed
Given: $\mathbf{w}_m = [x, y]$: waypoint in the map frame
Given: $\mathbf{x} = [x, y]$: robot position
$\mathbf{w}_r = TransformToRobotFrame(\mathbf{w}_m)$
$\theta_{command} = K_P e + K_I \int_0^t e dt + K_d \frac{de}{dt}$
if $\theta_{command} > \frac{\dot{x}_{desired}}{2}$ **then**
  $\dot{x}_{command} = \dot{x}_{desired} - \theta_{command}$
else
  $\dot{x}_{command} = \dot{x}_{desired}$
$sendCommands(\dot{x}_command, \theta_{command})$

$$
\begin{aligned}
\theta_1 &= \arctan \frac{y}{x} \\
\gamma &= \arctan \frac{-y/\sqrt{(x^2+y^2)}}{-x/\sqrt{(x^2+y^2}} \\
\sigma &= \frac{x^2+y^2+L2^2-L3^2}{2L2\sqrt{x^2+y^2}} \\
\theta_2 &= \gamma - \arccos \sigma \\
\theta_3 &= \arctan \frac{y - L2\sin\theta_2/L3}{x - L2\cos\theta_2/L3} - \theta_2 \\
\theta_4 &= \phi - \theta_3 - \theta 2
\end{aligned}
\tag{1}
$$

where Ln is the arm link lengths. In our case, we extracted these from the specifications of the arm: $L1 = 0.04225$, $L2 = 0.105948101$, $L3 = 0.1$ and $L4 = 0.0685$; all in meters. This always results in 2 possible solutions; the one with the elbow pointing upwards is selected. In addition to this, the controller was constrained to not attempt to reach positions outside of the reachable area or positions in the area of the robot. This controller worked fine by itself and the arm was able to achieve all desired behaviour for the task, however, the hardware controller of the arm was pre-configured for a static environment (where the arm is mounted on a solid object), which was not the case in our task. Thus, the robot arm moved surprisingly fast, resulting in large inertias for the robot, causing to turn over.

Due to this issue, we decided to also implement a high-level arm controller which would sequentially feed *arm animations* to the actual controller. Increasing the resolution of these animations allows the robot to perform more smooth and controlled actions. Sadly, due to time limitations, these animations were derived manually offline and selected via the high-level state machine.

This strategy of using the robot arm proved more than adequate for the task of pushing the button and moving the boxes, however, it did not prove reliable enough for picking up the sample towards the final stage of the task. This was mainly due to inaccuracies in the localisation system and the lack of any visual feedback when picking up the sample (it is not seen by the lidar). To address this issue, we opted for a simple strategy of using the arm as a broom and sweeping the area in front of the robot towards its centre. This allowed us to account for the inaccuracies of the localisation and position the sample in a more probable position. With this strategy, we achieved 80% success ration on picking up Rock 1 (see Figure 2).

3 Results

Overall the robot was successful in all of the criteria of the task as defined in Section 1 except for achieving all of the desired tasks in less than 5 minutes. This resulting from user operation and no prior validation of the task.

Particle filter localisation worked very reliably after all of the tuning and incremental improvements. This was essential for the success of the robot as its position is the backbone of all other modules. After all of the fine-tuning, noise filtering and performance optimisations, the particle filter never failed and always localised the robot precisely down to an 8cm radius. Sadly, there was no robust method of evaluating the performance of this algorithm due to the lack of ground truth.

The navigation system of the robot also worked correctly all of the time and managed to navigate the robot to all reachable parts of the environment within a 5cm radius. There were some small issues with the navigation stack which resulted from the particle filter becoming inaccurate after a long duration of standstill. Luckily, this was not a probable scenario in the task and was ignored.

The arm controller was the least reliable part of the software of the robot. This was mainly due to bad hardware, bad decisions, and short development time. As such the arm was very over-tuned to the task at hand and relied on exceptional localisation and navigation. As such any time, those systems failed within a small threshold, the arm would fail as well. Luckily, the other systems were reliable enough.

4 Discussion

Although the robot hardware was more than adequate for the task, it also had some limitations as all robotics systems. The major issue was the bad compatibility between the Turtlebot and the robot arm which were not designed to be used together. This manifested itself in drastically reduced operational time of the robot which decreased from 2.5h to 0.5h, the high-centre of gravity, bad motion inertias and obstruction of the lidar. These issues were overcome with careful operations planning, less dynamic motion control and sensor filtering. In the future, the choice and placement of such a robotic arm should be considered more carefully.

The particle filter from Section 2.4 worked for the task, however, there is still room for improvement and more efficient robot deployment; namely the measurement model. In this report we used a likelihood field model, however, that is difficult to tune, thus a ray-casting type model as the one in [6] would be a better option.

One of the major limitations of the A* path planner was the jigsaw-like packages that it produced in difficult scenarios due to the exploration methodology. This worked for the task of this report but should be better optimised in a real-world application with a graph optimisation of the path after it has been found, this would smooth the path and make the robot more efficient in its navigation.

Although we chose to use analytical IK in Section 2.6, that was probably not the best solution to the problem due to the low-level controller of the arm and the resulting high inertias of the robot. In this report the issue was fixed with the introduction of manually derived "animations", however, a better solution to this problem would be to do Iterative IK such as page 19 of [7]. This would result in automatically smooth arm movements with an artificially added delay.

References

[1] Roland Siegwart et al. *Introduction to autonomous mobile robots*. MIT press, 2011.

[2] Jason M O'Kane. "A gentle introduction to ROS". In: (2014).

[3] Nathan Koenig and Andrew Howard. "Design and use paradigms for gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*. Vol. 3. IEEE. 2004, pp. 2149–2154.

[4] Nicholas Swift. *Easy A* Pathfinding*. `https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2`. Accessed: 2019-11-29.

[5] V. Kumar. *Planar kinematics*. `https://www.seas.upenn.edu/~meam520/notes/planarkinematics.pdf`. Accessed: 2019-11-29.

[6] Corey H Walsh and Sertac Karaman. "CDDT: Fast Approximate 2D Ray Casting for Accelerated Localization". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 1–8.

[7] Alex Li. *Planar kinematics*. `http://wcms.inf.ed.ac.uk/ipab/rss/lecture-notes-2018-2019/10%20RSS-Kinematics.pdf`. Accessed: 2019-11-29.

Appendices

A Supplementary images

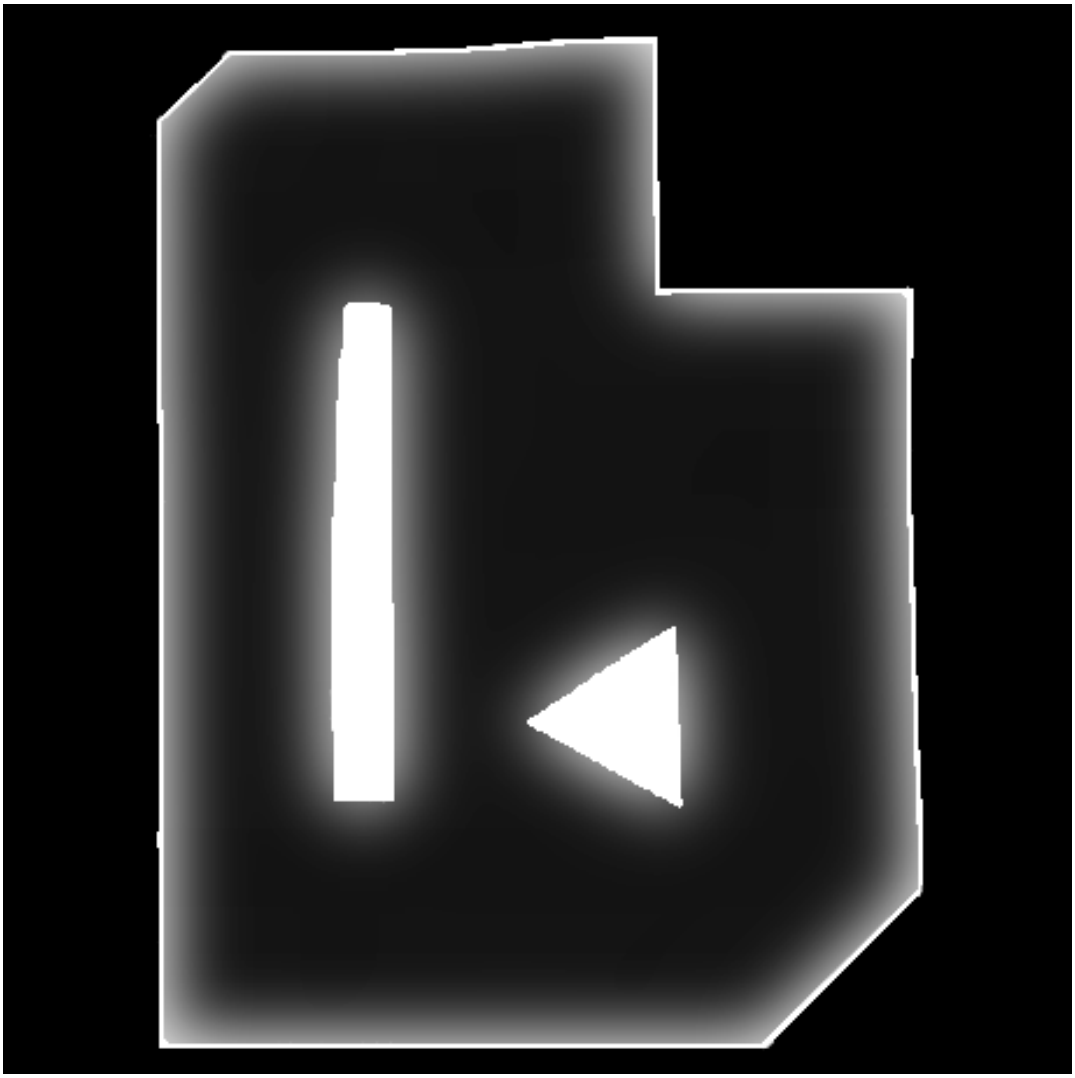


Figure 7: Localisation likelihood field map