Resumé

Dette projekt udforsker anvendelsen af reinforcement learning (RL) til at forbedre navigation og opgaveløsning for droner i miljøer såsom tekniske rum ved kundelokationer. Udført i samarbejde med Grundfos, en dansk virksomhed specialiseret i pumpesystemer, har projektet til formål at automatisere inspektioner og opgaver, der involverer lokalisering af pumper og kortlægning af ukendte omgivelser.

Problemet blev formaliseret som en Euclidean Markov Decision Process (EMDP). Ved at bruge en EMDP blev dronens beslutningsproces modelleret. Denne formalisering hjalp med at definere de forskellige komponenter af problemet, herunder tilstande, handlinger, overgange og belønninger, hvilket gjorde det muligt at udvikle en robust læringsalgoritme til dronens navigation og opgaveløsning.

Der anvendes en kombination af flere teknologier og metoder. UPPAAL STRATEGO anvendes til strategisyntese, som integreres med Robot Operating System (ROS) for at styre droneaktuationen i et simuleret miljø. Ved hjælp af Stochastisc Model-Predictive Control (STOMPC) kan systemet integrere realtidsdata fra en virtuel verden for løbende at syntetisere og tilpasse strategier. Dronen lærer at navigere ved at interagere med miljøet og modtage belønninger for at nå bestemte mål, såsom at finde pumper. To forskellige belønningsstrukturer blev testet for at eksperimentere med, hvilken tilgang der resulterede i at pumper i rummet findes hurtigst.

Mapping-algoritmen udviklet i simulationsmiljøet bygger på en 3D punktsky fra dronens dybdekamera, som bruges til at opbygge et 2D kort af miljøet. Punktskyen analyseres for at identificere vægge og frie områder, hvilket hjælper dronen med at navigere effektivt uden at kollidere med forhindringer. En abstraktion over en pumpedetektion model er implementeret som skal forestille hvordan et billede fra et kamera kunne processeres og returnere en sandsynlighed baseret på om billedet ser ud til at indeholde en pumpe og derved markere interessante områder af kortet. Pumpedetektionen opdaterer dronens viden om miljøet i realtid og hjælper med at øge sandsynligheden for korrekt identifikation af pumper.

Eksperimenterne blev udført i simulerede miljøer med forskellige konfigurationer af granularitet af kortet (coarse og fine), observationsområder og strategisynshorisonter. Resultaterne viste, at den belønningsstruktur, der fokuserede på hurtig udforskning (Reward Type 1), generelt førte til hurtigere og mere konsistente resultater sammenlignet med den struktur af belønning, der fokuserede på grundig og systematisk udforskning af nærområdet (Reward Type 2).

Den udviklede løsning er designet som en Proof of Concept (POC) og omfatter flere forenklinger og antagelser for at demonstrere det grundlæggende koncept. Dronen har sin bevægelse begrænset til to dimensioner for at forenkle navigation og kortlægning. Antallet af pumper i miljøet er kendt på forhånd, hvilket optimerer udforskningen ved at tillade dronen at afslutte søgningen, når alle pumper er fundet.

Projektet demonstrerer, at UPPAAL STRATEGO og STOMPC kan forbedre dronekontrol i komplekse og ukendte miljøer betydeligt. Dette kan potentielt anvendes til at automatisere rutineinspektioner og reducere behovet for manuel arbejdskraft, hvilket øger effektiviteten og sikkerheden i tekniske installationer.

Reinforcement Learning-Driven Drone Navigation in Simulated Environments Using Stochastic Model-Predictive Control and UPPAAL STRATEGO

Kasper L. H. Pedersen and Philip C. Greve

Department of Computer Science
Aalborg University

Abstract. Drones are increasingly utilized in industrial applications, including field service management for service and repair in complex environments. In collaboration with Grundfos, a Danish company specializing in pump systems, this project aims to investigate the possible application of reinforcement learning (RL) to enhance drone navigation and task execution within technical rooms at customer locations. The relevant tasks for the drone are to localize pumps in the unknown environment, while also building a map. Our approach involves the use of UPPAAL STRATEGO for strategy synthesis, integrated with Robot Operating System (ROS) to manage drone actuation within a simulated environment. Utilizing co-simulation, we integrate real-time data from a virtual environment to continually synthesize strategies. Building on previous applications of the Stochastic Model-Predictive Control (STOMPC) framework in other domains, this study demonstrates the potential of RL for this use case through a Proof of Concept implementation. Our experiments show that our RL-based approach outperforms a baseline method, highlighting the effectiveness of our proposed solution. Using one ideal configuration, these experiments show that our solution completes the task of locating a single pump in an average of about 4 minutes compared to 14 minutes of the baseline. We further experiment with two different reward engineering approaches to assess which yields the fastest task completion times.

1. Introduction

As autonomous systems and machine learning technologies advance, unmanned aerial vehicles (UAVs), commonly known as drones, have become increasingly prevalent in a variety of industrial applications. Drones are particularly valuable in field service management for tasks such as installation, service, and repair of equipment in technically complex environments that are often risky or inaccessible to humans. This reduces both the time and costs associated with manual labor.

This project explores the use of reinforcement learning (RL) as a method for enhancing drone control in navigating and performing tasks within complex spaces like technical rooms at customer locations. The aim is to utilize RL not just for learning optimal paths but for dynamically adapting to new information encountered during flight, thus improving the efficiency and safety of operations.

This thesis is a case study involving a simulation setup where a virtual drone operates within a digitally recreated environment of a possible technical room. Using UPPAAL STRATEGO for strategy synthesis, the project tests the drone's decision-making

capabilities in real-time scenarios. Robot Operating System (ROS) is integrated to manage the virtual drone's actuation, providing a realistic and controlled testing ground. The central question of this study is to assess how effectively reinforcement learning can be applied to the task of locating and interacting with specific targets like pumps in the simulated environment, considering the adaptive challenges posed by the dynamic conditions of field service locations.

Motivational Case

Grundfos is a Danish company specializing in pump systems. In an ongoing collaboration, Grundfos has presented a case contemplating the use of drone technology to automate routine inspections of technical rooms containing pump systems. Currently, a Grundfos technician or salesperson conducts these inspections using a proprietary mobile application developed by the company. This process involves manually scanning the room to identify and record the locations of pumps, which is time-consuming and requires the physical presence of staff. Once a pump is located, details need to be recorded about the pump. Here, Grundfos has trained a pump detection model to classify pump models from their assortment. See for example the wall with different pump products in Figure 1.

Figure 1: Wall with Grundfos pump products

Grundfos is interested in a drone autonomously inspecting a room and reporting back its findings. The proposed drone system aims to not only locate pumps but also to gen-

erate a detailed map of the environment. Such advancements would enable technicians to prepare more effectively before visiting the site or to reduce time spent on-site, thus enhancing efficiency and allowing for more direct interaction with customers.

However, the implementation of this drone technology would come with several constraints. The drone must operate efficiently in environments often devoid of internet or GPS connectivity, challenging its navigation and data transmission capabilities. Furthermore, it must complete its tasks swiftly to align with the operational timelines of on-site visits. The information gathered by the drone should include the locations of pumps within the room and a comprehensive environment map, which will aid in the planning of maintenance and sales visits.

The automation of inspection tasks through drone technology would represent a significant shift from Grundfos' current practices, offering an innovative solution to on-ground challenges faced by technicians and sales staff.

To address this Grundfos case we aim to develop a solution that leverages RL for drone navigation and pump localization in simulation environments, modelled as a Markov Decision Process, utilizing UPPAAL STRATEGO for strategy synthesis, ROS for drone control, and STOMPC for co-simulation between Gazebo and UPPAAL STRATEGO.

The rest of this report is structured as follows. Section 2 helps scope the problem to be tackled in this work. Section 3 covers preliminary concepts of RL and EMDPs. In Section 4, we model the case as a drone EMDP. Section 5 then takes this EMDP and creates a UPPAAL model which enables learning strategies using UPPAAL STRATEGO. Section 6 presents the concept of co-simulation and Stochastic Model-Predictive Control. Section 7 discusses Gazebo simulation and ROS and elements of our solution including drone control, mapping, and pump detection mechanisms. Section 8 describes experiments conducted for examining the solution's applicability, evaluating it under various parameter configurations, and comparing it against a baseline. Section 9 concludes with final remarks on the developed solution and obvious next steps. Section 10 emphasizes related work done previously including other machine learning techniques for drone control and the use of UPPAAL STRATEGO and STOMPC across various domains.

2. Scoping the Grundfos Case

Grundfos envisions an end-to-end solution in which a drone autonomously maps an intricate, unknown environment in high detail, identifying pump locations, classifying pump types, and determining relationships between technical components within a space. In the following we scope the problem, thus focusing on the aspects of this vision that our project will address.

Simplification to Two-Dimensional Space We constrain the drone's operational space to two dimensions rather than three. Starting in a two-dimensional space provides a simpler foundation to start working on while preserving the core ideas of the motivational case. Thus, it provides an adequate proof of concept. This decision simplifies several aspects of the project:

- **Movement**: The drone will navigate using only the four cardinal directions: North, South, East, and West, eliminating the need for vertical movement. As such, the drone can only move and turn in these directions.

- **Mapping**: The drone will generate a 2D map of the environment. This reduces the complexity of mapping algorithms and the actions required from the drone.
- **Resolution**: The granularity of the map, defined by the size of its cells, will be coarser. While finer detail can be achieved by reducing cell size, using larger cells simplifies the mapping process and reduces computational demands.

Assumptions on Number of Pumps We assume the number of pumps within the environment is known. While this is unlikely in real-world scenarios, it enables the drone to conclude its search once all pumps are accounted for. Otherwise, full exploration of the room would be required to be sure all pumps were found.

Utilization of Grundfos's Pump Detection Model Grundfos has developed a pump detection model that identifies pumps within an image, outlining them with bounding boxes and assigning confidence scores and classifications. This is the obvious tool to employ and let the drone use this model. However, our preliminary experiments, as discussed in [1], reveal that the model's effectiveness is confined within a specific interval, likely due to the training set images being taken from that distance. Consequently, we will abstract this task to enhance the detection range. Our adapted model will identify potential pump-like objects across broader areas but will not classify these objects or read their nameplates as initially envisioned by Grundfos.

3. Reinforcement Learning

Reinforcement Learning (RL) is a machine learning approach where an agent learns to make sequential decisions by interacting with an unknown environment through actions and receiving rewards over a series of discrete time steps [2, p. 1-3]. The agent can sense some aspects of the environment's state, take actions that influence this state, and have a certain goal state that it tries to reach. The learning process can be formalized as optimal control of a Markov Decision Process (MDP), which is a classic formalization of sequential decision-making [2, p. 47-48]. The MDP must capture the relevant properties of a learning agent facing a real problem, such as a state representation, possible actions, transition probabilities, rewards, and goal state(s).

Traditional MDPs provide a mathematical framework for modelling sequential decision-making problems, but they rely on discrete state representations. Euclidean Markov Decision Processes (EMDPs) provide an extension to traditional MDPs by representing the state space as Euclidean space, allowing for continuous state representations [3]. This extension bridges the gap between standard MDPs and the complexities of real-world environments, making EMDPs more suitable for modelling problems with continuous characteristics, such as robotics.

Based on the formalism in [3], an EMDP is defined as follows.

Definition 1 (EMDP). *An EMDP is a tuple $\mathfrak{M} = \langle \mathcal{S}, \mathcal{A}, s_0, T, R, \mathcal{G} \rangle$, where:*

- $\mathcal{S} \subseteq \mathbb{R}^K$ *is a (K-) bounded and closed subset of the Euclidean space,*
- $\mathcal{A}$ *is a finite set of actions,*
- $s_0 \in \mathcal{S}$ *is the initial state,*

- $T : \mathcal{S} \times \mathcal{A} \to (\mathcal{S} \to \mathbb{R}_{\geq 0})$ *is the transition function yielding a density function over* $\mathcal{S}$ *for each state-action pair,*
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ *is the reward function for state-action-state triples, and*
- $\mathcal{G} \subseteq \mathcal{S}$ *is the set of goal states.*

Several techniques are commonly employed in RL to solve these MDPs, such as dynamic programming (DP), Monte Carlo methods, and Temporal Difference (TD) learning [2, p. 23]. TD learning is particularly interesting as it is a combination of ideas from both Monte Carlo methods and DP, that allows agents to learn directly from experience in an online and iterative manner [2, p. 119]. Q-learning is a TD algorithm used to learn the optimal action-value function (Q-function), which is a function denoting how useful a given action is in achieving future rewards. It does so by iteratively updating this function based on the temporal difference between the expected and actual reward when taking an action in a particular state [2, p. 131-132].

The advantage of formalizing the learning process as an (E)MDP is that such a model enables the generation and execution of strategies while allowing for accumulated reward estimation of said strategies [3].

Definition 2 (Strategy). *A strategy for an EMDP* $\mathfrak{M}$ *is defined as* $\sigma : \mathcal{S} \to (\mathcal{A} \to [0, 1])$*, namely a mapping of a state to a probability distribution over* $\mathcal{A}$*.*

Being able to estimate the reward associated with individual strategies is imperative for finding the best strategy amongst multiple ones. The reward expectation of a strategy can be estimated using the following definition.

Definition 3 (Reward expectation of a strategy). *Let* $\mathcal{G}$ *be a set of goal states and* σ *a strategy. When starting in state* s *and reaching* $\mathcal{G}$*, the expected reward of doing so is the solution to the following system of equations:*

$$\mathbb{E}_{\sigma}^{\mathfrak{M}}(\mathcal{G}, s) = \sum_{a \in Act} \sigma(s)(a) \cdot \int_{t \in \mathcal{S}} T(s, a)(t) \cdot \left(\mathcal{R}(s, a, t) + \mathbb{E}_{\sigma}^{\mathfrak{M}}(\mathcal{G}, t)\right) dt$$

when $s \notin \mathcal{G}$ *and* $\mathbb{E}_{\sigma}^{\mathcal{M}}(\mathcal{G}, s) = 0$ *when* $s \in \mathcal{G}$*.*

The objective is to compute a strategy σ that optimizes $\mathbb{E}_{\sigma}^{\mathfrak{M}}(\mathcal{G}, s_0)$ according to the reward function used. Analytic solutions to these integral equations are often unobtainable, and for this reason, approximation approaches based on finite partitionings of the state space are employed [3].

4. Case as an EMDP

In our setting, the main objective is to find the pumps in the environment. To do so, the drone must explore the environment by moving around and collecting information about safe areas and possible pumps. Therefore, we define a *drone EMDP* D relevant to our specific problem case:

$$D = \langle \mathcal{S}_D, \mathcal{A}_D, s_0, T_D, R_D, \mathcal{G}_D \rangle$$

The following definitions are to capture the underlying properties of the drone and environment as parts of the D semantics.

An important aspect of the problem is for the drone to utilize a map denoting partial knowledge about the environment during learning to improve the decision-making process in navigation and collision avoidance when exploring. Drone movement is limited to two dimensions, which is reflected in the environment map that is defined as follows.

Definition 4 (Environment Map). *An environment map $\mathcal{M}_e$ is an $n \times m$ matrix, where a cell of $\mathcal{M}_e$ by default represents a square meter of the environment and $\mathcal{M}_e^{i,j} \in \{\texttt{unknown}, \texttt{free}, \texttt{wall}, \texttt{pump}\}$ is a value denoting current information of the cell at row i and column j of $\mathcal{M}_e$. The set of all environment maps is denoted by E.*

The drone should be able to localize pumps, i.e., determine the position of pumps within the environment. We define Points of Interest (POIs). A POI is a location in an explored part of the map that has interest. The main idea behind this is that in a realistic environment, there may be things or structures that look like pumps, but upon further inspection might turn out not to be. A POI is defined as follows.

Definition 5 (Point of interest (POI)). *A POI $\rho \in \Gamma$ is a tuple $\rho = \langle (x, y), e, v \rangle$, where:*

- $(x, y) \in \mathbb{N} \times \mathbb{N}$ *is the coordinates of ρ within the environment map $\mathcal{M}_e$,*
- $e \in \mathbb{B} = \{tt, ff\}$ *indicates whether ρ has been examined by the drone,*
- $v \in [0, 1]$ *indicates the current probability of the POI being a pump. Here, $v = 0$ if it was concluded not to be a pump, and $v = 1$ when the POI has been concluded to be a pump. Any value of v between 0 and 1 signifies the current probability of the POI being a pump.*

The set Γ denotes the set of currently observed POIs.

We can now define the state of our drone EMDP D capturing the current state of the system, including positional information about the drone and current map knowledge of the environment and POIs.

Definition 6 (State set). *The set of states S_D consists of tuples of the form $s = \langle (x, y), dir, \mathcal{M}_e, \Gamma \rangle$, where:*

- $(x, y) \in \mathbb{N} \times \mathbb{N}$ *is the coordinates of the drone within the environment map $\mathcal{M}_e$,*
- $dir \in Dir = \{\texttt{north}, \texttt{east}, \texttt{south}, \texttt{west}\}$ *is the direction of the drone,*
- $\mathcal{M}_e \in E$ *is the current state of the environment map,*
- Γ *denotes the current set of observed POIs.*

The initial state s_0 is part of the state set, such that $s_0 \in S_D$.

The next part of the drone EMDP D is an action set. The set of possible actions is constrained to turning and moving forward, as this simple set allows the drone to move in any two-dimensional direction.

Definition 7 (Action set). *The action set $\mathcal{A}_D = \{forward, turn\_left, turn\_right\}$ denotes all possible actions.*

An important aspect of exploring the environment is that the drone remains safe, i.e., avoids collisions with walls. To enforce exactly this, we define a shielding function $check\_move$ that ensures that the drone never moves into a wall cell of the environment map $\mathcal{M}_\text{e}$.

$$check\_move(x, y, dir, \mathcal{M}_\text{e}) = \begin{cases} \textit{tt} & \text{if } dir = \texttt{north} \text{ and } {\mathcal{M}_\text{e}}^{x-1,y} = \texttt{free} \\ \textit{tt} & \text{if } dir = \texttt{east} \text{ and } {\mathcal{M}_\text{e}}^{x,y+1} = \texttt{free} \\ \textit{tt} & \text{if } dir = \texttt{south} \text{ and } {\mathcal{M}_\text{e}}^{x+1,y} = \texttt{free} \\ \textit{tt} & \text{if } dir = \texttt{west} \text{ and } {\mathcal{M}_\text{e}}^{x,y-1} = \texttt{free} \\ \textit{ff} & \text{otherwise} \end{cases}$$

The shield protects the drone from taking a forward action when the cell immediately in front of it has been mapped as a wall. Note that only the forward action is shielded as it changes the position of the drone unlike simply turning, which happens in-place.

The transition function of our drone EMDP D should capture how the action set influences the probability of transitioning to another state. Furthermore, we define a transition-relation for each action, signifying how a particular action affects the next state. We denote the use of a function $next$, which given a drone position and direction, returns the coordinate of the cell in front of the drone in the environment map.

$$next((x, y), dir) = \begin{cases} (x-1, y) & \text{if } dir = \texttt{north} \\ (x, y+1) & \text{if } dir = \texttt{east} \\ (x+1, y) & \text{if } dir = \texttt{south} \\ (x, y-1) & \text{if } dir = \texttt{west} \end{cases}$$

Definition 8 (Transition function). *$T_D : \mathcal{S}_D \times \mathcal{A}_D \to (\mathcal{S}_D \to \mathbb{R}_{\geq 0})$ is a transition function yielding a density function over $\mathcal{S}_D$ for each state-action pair given below.*

The transition function is deterministic, i.e., for a state s and action a, there exists a state s' such that $T_D(s, a)(s') = 1$ if a is allowed according to $check\_move$. As the drone can always turn, $T_D(s, turn\_left)(s') \neq 0$ and $T_D(s, turn\_right)(s') \neq 0$. Further, $T_D(s, forward)(s') \neq 0$ if $check\_move((x, y), dir, \mathcal{M}_e) = \textit{tt}$, and 0 otherwise for a state $s = \langle (x, y), dir, \mathcal{M}_e, \Gamma \rangle$.

In order to convey the relation between a state s and a new state s' when taking an action a, we define the following transition-relations for each action.

$$
\begin{aligned}
((x,y),dir,\mathcal{M}_e,\Gamma) \xrightarrow{\textit{forward}} &((x',y'),dir,\mathcal{M}_e{}',\Gamma) \\
&\textit{if } check\_move(x,y,dir,\mathcal{M}_e) = tt \\
&\textit{and } x' = \begin{cases} x & \textit{if } dir \in \{\texttt{east},\texttt{west}\} \\ x+1 & \textit{if } dir = \texttt{south} \\ x-1 & \textit{otherwise} \end{cases} \\
&\textit{and } y' = \begin{cases} y & \textit{if } dir \in \{\texttt{north},\texttt{south}\} \\ y+1 & \textit{if } dir = \texttt{east} \\ y-1 & \textit{otherwise} \end{cases} \\
&\textit{and } \mathcal{M}_e' = \mathcal{M}_e \textit{ such that } \mathcal{M}_e^{p,q} = \texttt{free} \\
&\quad \textit{where } p,q = next((x',y'),dir) \\
&\quad \textit{if } \mathcal{M}_e^{p,q} = \texttt{unknown}
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
((x,y),dir,\mathcal{M}_e,\Gamma) \xrightarrow{\textit{turn\_left}} &((x,y),dir',\mathcal{M}_e{}',\Gamma) \\
&\textit{if } dir' = \begin{cases} \texttt{north} & \textit{if } dir = \texttt{east} \\ \texttt{east} & \textit{if } dir = \texttt{south} \\ \texttt{south} & \textit{if } dir = \texttt{west} \\ \texttt{west} & \textit{if } dir = \texttt{north} \end{cases} \\
&\textit{and } \mathcal{M}_e' = \mathcal{M}_e \textit{ such that } \mathcal{M}_e^{p,q} = \texttt{free} \\
&\quad \textit{where } p,q = next((x,y),dir') \\
&\quad \textit{if } \mathcal{M}_e^{p,q} = \texttt{unknown}
\end{aligned}
\tag{2}
$$

$$
\begin{aligned}
((x,y),dir,\mathcal{M}_e,\Gamma) \xrightarrow{\textit{turn\_right}} &((x,y),dir',\mathcal{M}_e{}',\Gamma) \\
&\textit{if } dir' = \begin{cases} \texttt{north} & \textit{if } dir = \texttt{west} \\ \texttt{east} & \textit{if } dir = \texttt{north} \\ \texttt{south} & \textit{if } dir = \texttt{east} \\ \texttt{west} & \textit{if } dir = \texttt{south} \end{cases} \\
&\textit{and } \mathcal{M}_e' = \mathcal{M}_e \textit{ such that } \mathcal{M}_e^{p,q} = \texttt{free} \\
&\quad \textit{where } p,q = next((x,y),dir') \\
&\quad \textit{if } \mathcal{M}_e^{p,q} = \texttt{unknown}
\end{aligned}
\tag{3}
$$

Executing any action will result in the cell in front of the new cell to be set as free, hence the use of $\mathcal{M}_e^{p,q} = \texttt{free}$ in all of the above transition relations if that cell is not a wall already. This enables the system to set a single free cell in the environment map when doing an action, which will help drive the exploration process of unknowns by providing rewards for doing so. The reason that it is specifically the cell in front that is set as free is because this means that when we move into a cell, we know it is free as it has been set

prior. This is of course under the assumption that the immediate unknown cell is indeed free.

Note that the set of POIs Γ do not change in any transition and thus remain static. The reason for this will be outlined following the remaining definitions.

The next part of the drone EMDP D is to define the reward function, that captures the immediate rewards associated with individual decision moments dependent on a state s and an action a.

For the following, we define two helper functions: max_poi_pos that given a set of POIs, returns the position (x, y) of the POI with the highest value v provided that it is unexamined, and max_poi_val that given a set of POIs, returns the highest value v of any unexamined POI.

$$\textit{max\_poi\_pos}(\Gamma) = (x, y) \qquad \text{where} \quad ((x, y), \textit{ff}, v) = \underset{((d,f),b,z)\in\Gamma}{\arg\max}\ z$$

$$\textit{max\_poi\_val}(\Gamma) = v \qquad \text{where} \quad ((x, y), \textit{ff}, v) = \underset{((d,f),b,z)\in\Gamma}{\arg\max}\ z$$

Together, max_poi_pos and max_poi_val return the position of the unexamined POI with the highest probability of being a pump and its corresponding probability.

Definition 9 (Reward function). *$R_D : \mathcal{S}_D \times \mathcal{A}_D \times \mathcal{S}_D \to \mathbb{R}$ is the reward function for state-action-state triples consisting of an exploring and POI reward as given below. As such the reward function is defined as $R_D(s, a, s') = R^e_D(s, a, s') + R^p_D(s, a, s')$.*

An exploration reward $R^e_D(s, a, s') = 100$ is provided if $\mathcal{M}^{p,q}_e = \texttt{unknown}$ where $p, q = next((x', y'), dir')$ given a state $s = \langle (x, y), dir, \mathcal{M}_e, \Gamma\rangle$ and $s' = \langle (x', y'), dir', \mathcal{M}_e{}', \Gamma\rangle$. As such, the exploration reward is given when the drone arrives in a state s' of which the cell in front of it in the environment map is unknown. This reward incentivizes the drone to explore unknown parts of the map.

A POI reward $R^p_D(s, a, s') = max\_poi\_val(s') \times 1000$ if $next((x', y'), dir') = max\_poi\_pos(s')$ given the state $s' = \langle (x', y'), dir', \mathcal{M}_e{}', \Gamma\rangle$. As such, the POI reward is given when the drone moves to a cell that is directly in front of the POI with the highest probability of being a pump. This reward incentivizes the drone to move closer to those POIs most likely to be pumps as the reward is relative to the probabilities of POIs.

The rewards are intentionally engineered so that the reward for uncovering POIs greatly outweighs the reward for exploring to capture the overall goal of finding pumps within the reward engineering. As such, the drone will almost always prefer uncovering POIs rather than exploring. However, note that these particular reward values could be changed according to needs depending on the behavior desired.

The final part of the drone EMPD D is to define what constitutes a goal state. The objective is to find all the pumps in the environment, and as denoted in Section 2 we assume the total amount of pumps to be known and constant. We let $\delta \in \mathbb{N}$ denote the known total amount of pumps within the environment. We then define $pumps(\mathcal{M}_\text{e}) = \sum_{i,j}[\mathcal{M}^{i,j}_\text{e} = \texttt{pump}]$ as a function that counts the number of cells with value $\texttt{pump}$ in the environment map $\mathcal{M}_\text{e}$, i.e., counts the number of pumps currently found.

Definition 10 (Goal states). *The set of goal states is defined as $\mathcal{G}_D \subseteq \mathcal{S}_D$. A goal state $s = \langle (x, y), dir, \mathcal{M}_e, \Gamma\rangle \in \mathcal{G}_D$ is a state where $pumps(\mathcal{M}_e) = \delta$.*

As such, a goal state s is reached when the number of pumps that has been found in the environment map $\mathcal{M}_e$ of state s equals the known total amount of pumps δ.

This completes our definition of the drone EMDP D tailored to explore and move closer to POIs. Our drone EMDP D might seem rather simple compared to the complexities of the problem case, but it is in fact intentionally designed to minimally mock how a state evolves. One might for example notice, that we in Definition 4 specify that the cells of the environment map can be walls and pumps, yet we do not show these being set in Definition 8. Likewise, for the POIs we do not specify how the probability or examined value of these are set or can change. We have realized our solution such that wall mapping and pump detection are not something that is directly captured by our drone EMDP D. In other words, D has no prediction of how these will evolve from state to state like it for example does with moving forward and turning. Instead, these state variables are to be dynamically updated by external algorithms outside of the drone EMDP D. These external algorithms involve mapping and pump detection mechanisms based on sensory data from the drone in the simulation environment.

This approach also justifies why we can settle for a deterministic EMDP. The more complex dynamics involving uncertainty and non-determinism are managed by external algorithms and the resulting state information is then dynamically integrated into the EMDP. Therefore, D can remain straightforward, focusing on the core learning task.

Having modeled the problem as an EMDP enables us to employ RL techniques to navigate the environment and optimize exploration. We utilize the tool UPPAAL, and its learning extension called UPPAAL STRATEGO, for this task.

5. Implementing Drone EMDP *D* in UPPAAL STRATEGO

UPPAAL is a tool that provides a platform for the modelling, validation, and verification of system models. These system models consist of networks of timed automata extended with discrete variables over user-defined types being updatable by user-defined functions [4]. The tool comprises a description language, a simulator, and a model checker. The description language allows for the formal creation of such models. The simulator enables dynamic system executions to be tested, while the model checker examines the system's behaviour, verifying invariant, reachability, and liveness properties through exhaustive state-space searches [4]. UPPAAL STRATEGO is an extension to the UPPAAL tool that allows for the synthesis/learning of strategies for Stochastic Priced Timed Games (SPTG), which can be semantically represented by EMDPs [3]. An SPTG is a probabilistic game specified by timed automata with continuous price expressions which allow estimating the distribution of cost within the system. For a SPTG, a strategy is a sequence of transitions for any state that wins the game. The default learning algorithm in UPPAAL STRATEGO is Q-learning, distinguished by its ability to perform partition/refinement of the observable state variables [3]. This enables the tool to generalize observations onto sets of states, thus expanding the learning capabilities of UPPAAL STRATEGO to the continuous domain, a necessity for learning near-optimal strategies for EMDPs [5]. Naturally, this leads to representing our drone EMDP D as an SPTG, i.e., a timed automaton, that UPPAAL STRATEGO can solve. Note that for the rest of this report, we will refer to UPPAAL STRATEGO as a term for both UPPAAL and UPPAAL STRATEGO.

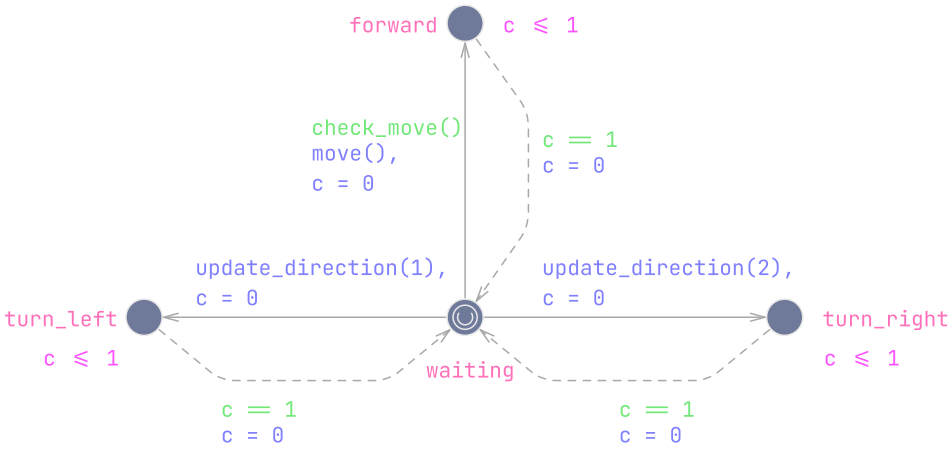


Figure 2: Drone EMDP as UPPAAL STRATEGO timed automaton

Figure 2 shows the timed automaton representation of our drone EMDP D created in UPPAAL STRATEGO. It encapsulates all aspects of the drone EMDP D as presented in the previous section Section 4. For example, the state is defined by the variables: `env_loc_x`, `env_loc_y`, `dir`, `map`, and `poi`, denoting the drone position and direction, environment map, and POIs, respectively. As such, these denote the state variables of the drone EMDP.

Figure 2 shows that the automaton has four locations; `waiting`, `forward`, `turn_left` and `turn_right`. These denote the actions of our drone EMDP D. Here, `waiting` is the center location with transitions going to and back from the other locations. Each edge from `waiting` is extended with transitioning logic that guards it and/or updates the state upon traversing the edge. For example, the edge going to `forward` has a guard `check_move()`, which is a UPPAAL STRATEGO realization of the $check\_move$ function used by our drone EMDP D in Section 4. As such, it is a shielding mechanism that uses the environment map to see if it allows a forward move, i.e., moving forward does not put the drone in a cell of the environment map deemed to be a wall.

```
bool check_move(){
    if (direction == 1)  predicted_x = predicted_x - 1;
    else if (direction == 2) predicted_y = predicted_y + 1;
    else if (direction == 3) predicted_x = predicted_x + 1;
    else if (direction == 4) predicted_y = predicted_y - 1;

    if (map[predicted_x][predicted_y] != 2)
        return true;

    return false;
}
```

Code Listing 1.1: Function *check_move()*

Listing 1.1 shows code for the function `check_move()` used as a guard on the action that would move the drone forward. Note that the direction is represented by numbers, where `north` = 1, `east` = 2, `south` = 3, and `west` = 4. The guard is enabled if the cell the drone predicts it will move to within the environment map is not a wall, i.e. does not have value 2, which we use to denote walls. Additionally, we check if the depth camera permits it to act as an extra shield in case of map incorrections using `check_front`.

If the guard of `check_move()` is satisfied, the edge can be traversed resulting in an update of the state through the update function `move()` which is based on Equation (1) of Section 4.

Listing 1.2 shows code for the function `move()` used as an update on the action moving the drone forward. Based on the current direction the cell predicted the drone moves to is set. The `action` variable is set to 1 to signify a forward move. Further, we make a function call to check if a reward should be given as a result of the action.

```
void move() {
    if (direction == 1) env_loc_x -= 1;
    if (direction == 2) env_loc_y += 1;
    if (direction == 3) env_loc_x += 1;
    if (direction == 4) env_loc_y -= 1;

    action = 1;
    check_rewards();
}
```

Code Listing 1.2: Function *move()*

The edges going to `turn_left` and `turn_right` are not guarded. Traversing the edges to these locations updates the state through the update function `update_direction(1)` or `update_direction(2)`, depending on left or right, based on Equation (2) or Equation (3) of Section 4.

```
void update_direction(int side){
    if(side == 1){
        if(direction == 1) direction = 4;
        else direction = direction - 1;
        action = 3;
    }
    if(side == 2){
        if(direction == 4) direction = 1;
        else direction = direction + 1;
        action = 2;
    }
    check_rewards();
}
```

Code Listing 1.3: Function *update_direction()*

Listing 1.3 shows code for the function `update_direction()` which is used as an update when taking the actions for turning left and turning right, respectively. Similarly, when the drone moves in the model, rewards are checked. Note here, that `side` variable determines whether it is a left turn or right turn by the use of value 1 or 2, respectively. If it is a left turn, we set the `action` variable to 3, while it is set to 2 for a right turn.

Rewards are implicitly provided according to Definition 9 through the updating logic of any action transition and can also be seen from the appendix of Section 10.

Listing 1.4 shows the code for checking for and supplying rewards, i.e., the UPPAAL STRATEGO realization of Definition 9. Firstly, we check if the cell in front of the drone in the environment map is 0, a value used to denote `unknown`, and if it is, we set it to 1 denoting it being `free`. We then provide the exploration reward of 100. Secondly, we check for the POI reward by seeing if the cell in front of the drone within the POI map is different from 1, a value used to denote a pump. This captures the idea that if a POI has already been concluded to be a pump we do not want to give reward for it anymore. If it is not 1, it must either be 0 or a value between 0 and 1. If it is 0, it means no POI is in the cell and the reward itself will therefore become 0 as we multiply. Contrary, if it is between 0 and 1, we provide a reward relative to the v value of the POI. As such, the reward will be higher when facing POIs with a higher probability of being pumps.

```
void check_rewards(){
    // Exploration reward
    if (map[env_next_x][env_next_y] == 0) {
        map[env_next_x][env_next_y] = 1;
        rp += 100;
    }

    // POI reward
    if (poi[env_next_x][env_next_y] != 1.0) {
        rp += poi_map[env_next_x][env_next_y] * 1000;
    }
}
```

Code Listing 1.4: Function *check_rewards()*

Executing an action, equivalent to being in `waiting` and going to one of the other locations and returning, takes 1 time unit. Independent of the edge traversed to one of these locations, an `action` variable is set denoting the specific action taken. This variable becomes an important part of clearly communicating strategies, i.e., having a compact and concise representation of action sequences, that can be easily interpreted.

Modelling the drone EMDP D in UPPAAL STRATEGO allows it to observe how the state evolves when different actions are taken. Having represented the EMDP as an SPTG creates a foundation for learning strategies that solve it. We utilize the Q-learning available in UPPAAL STRATEGO. The tool facilitates the specification of strategies in the form of queries. A query consists of maximizing some reward, setting the partial observability by the learning algorithm, and specifying the goal that should be reached.

Based on the aforementioned, we define a UPPAAL STRATEGO learning query as follows:

```
strategy fp = maxE(rp) [<=10]
    {Drone.waiting} -> {x,y,dir} : <> time >= 10
```

which learns a strategy that maximizes the estimated reward `rp` within a horizon of 10 time units, where only the location `waiting` and state variables `x,y` and `dir` are observable. We use the horizon `time >= 10` as the goal state in the query. A horizon of 10 means that the Q-learning of UPPAAL STRATEGO has 10 time units to find a goal state. In this case, the goal state is the state in which the reward during those 10 time units is maximized. Note that 10 time units translate to 10 actions, meaning that during learning we always maximize the reward within the next 10 actions. As such, the reward is always maximized within the 10 actions, whether that be due to the exploration reward or POI reward. Eventually, any POI that is a pump will be found.

In continuation of the previous learning query, the following simulate query can be defined:

```
simulate[<=10;1]{action} : time >= 10 under fp
```

which creates a single stochastic simulation run of the system. The simulation runs until a horizon of 10 is attained, signifying the same horizon as for the learning query above. During this simulation, the previously mentioned `action` variable is tracked thus denoting the sequence of actions learned during the horizon of 10, i.e., the next 10 actions.

As previously mentioned, we wish to iteratively update parts of our drone EMDP, namely these UPPAAL STRATEGO state variables, according to real-time information recorded by the drone in a simulation environment. This includes populating the environment map with wall and free locations using a mapping algorithm based on the data from the onboard depth camera and pump locations using an abstraction of a pump detection model. These will be detailed in the coming sections. By doing this we aim to always learn based on an updated drone EMDP, that captures the most recent state information. This makes co-simulation interesting as it will exactly allow for this real-time data exchange between UPPAAL STRATEGO and a simulation engine thus enabling online learning.

6. Co-simulation using Stochastic Model Predictive Control (STOMPC)

By integrating real-time information from a simulation environment, co-simulation allows us to learn from an updated drone EMDP D, making the solution more fitting to a real-world application. We employ the STOMPC framework for this matter as it facilitates the creation of Model-Predictive Control (MPC) designs using UPPAAL STRATEGO as the controller synthesis engine. Here, co-simulation is between UPPAAL STRATEGO and an external simulator allowing for a feedback loop between between the two, which enables continuous strategy synthesis [6] and therefore online learning. The external simulator performs the strategy and provides feedback to UPPAAL STRATEGO which is used to synthesize a new strategy.

MPC is characterized by its use of system models for finite-horizon prediction and strategy synthesis for objective optimization involving iterative calculations. It relies on

feedback loops from the true state following the execution of initial control actions [6]. The continuous recalculation of the optimal strategy is due to the natural likelihood of discrepancy between the predicted future state and the actual true state at any point. The process involves predicting the future state based on optimal actions, executing that action, observing the real true state, and using this real-time information to recalculate the next optimal action.

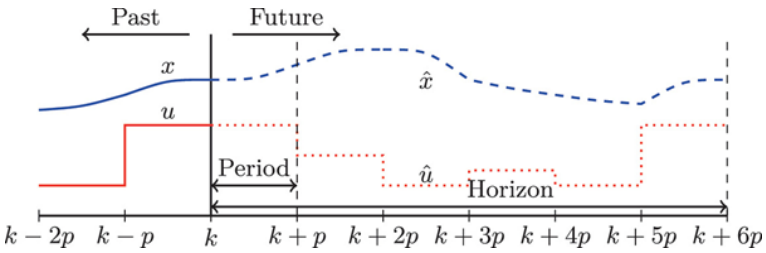


Figure 3: Model-Predictive Control conceptual sketch [6]

A conceptual sketch of how MPC works in STOMPC is shown in Figure 3. The solid blue line denotes the continuous evolution of the system state whereas the solid red line denotes the periodically adjusted control signal (action). The dotted lines denote the future prediction of their solid counterparts. As such, full trajectories denote the past, present, and future. The latter is up until some finite horizon [6]. Up to time $t = k$ we track the actual state of the system x and apply control inputs u. Leveraging a model of the system, a finite horizon prediction of the future system state $\hat{x}_k$ can be calculated. The future state is dependent on the control sequence $\hat{u}_k$. When the optimal control sequence (strategy) has been calculated, the first control input (action) hereof is applied. At time $t = k + p$, where p is the duration of the current period, the true state of the system $x(k + p)$ is then observed. This true state is likely deviating from the predicted state $\hat{x}_k(k + p)$. The new true state is used to recalculate a new optimal control sequence $\hat{u}_k$, thus adapting the strategy based on real-time system behavior and information [6].

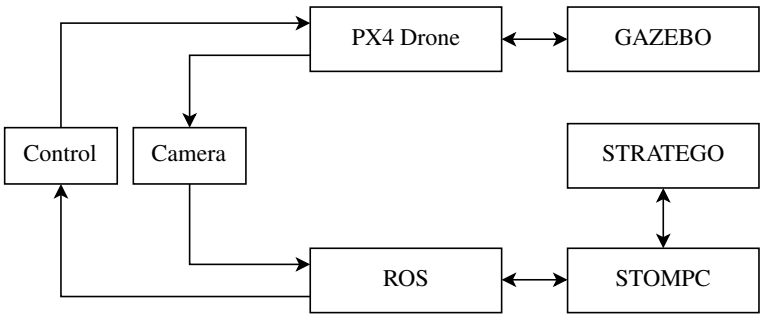


Figure 4: Component diagram of our co-simulation solution

For our solution, we utilize STOMPC to iteratively update the drone EMDP D state within UPPAAL STRATEGO based on data from the simulation environment after each action. Figure 4 shows a component diagram of our solution pipeline. By iteratively querying UPPAAL STRATEGO, STOMPC enables online learning using the queries specified earlier in Section 5. As such, co-simulation is done between UPPAAL STRATEGO and Gazebo simulator. Here, drone actions are realized by the use of Robot Operating System (ROS).

The UPPAAL STRATEGO variables, including those relevant to the EMDP state, that STOMPC works with can be seen by the use of "//TAG_X" in the UPPAAL STRATEGO model. This placeholder will be swapped out with the actual value when a copy of the XML file representing the UPPAAL STRATEGO model is created through the integration with STOMPC. As such, there will be a variable X on the STOMPC side of the solution, that will get pasted into the UPPAAL STRATEGO model each time the STOMPC loop queries UPPAAL STRATEGO for a strategy. See Listing 1.5 showing central state variables to the model.

```
// Cell that the drone is in within environment map
int env_loc_x = //TAG_env_loc_x;
int env_loc_y = //TAG_env_loc_y;

// Direction drone is facing (North, South, East, West)
int dir = //TAG_dir;

// Environment map and POIs
int map[//TAG_env_dim_x][//TAG_env_dim_y] = //TAG_map;
double poi[//TAG_poi_dim_x][//TAG_poi_dim_y] = //TAG_poi;

// How many pumps currently found and the known total amount
int pumps_reached = //TAG_pumps_reached;
int num_pumps = //TAG_num_pumps;
```

Code Listing 1.5: Model variables

STOMPC setting these state variables allows them to be determined by external factors independent of UPPAAL STRATEGO itself. To highlight our implementation of STOMPC at a high level consider the following.

1. Set initial state variables denoting the initial positional information of the drone in the simulation environment, empty environment map, and empty set of POIs.
2. STOMPC queries UPPAAL STRATEGO for a strategy.
3. Execute the first action of strategy in the Gazebo simulation environment using ROS.
4. Based on the outcome of action, update state variables in UPPAAL STRATEGO.
   - Positional information: `env_loc_x`, `env_loc_y`, `dir`.
   - Environment map based on sensor data: `map`.
   - POIs based on pump detection mechanism: `poi`.

The above defines the main STOMPC loop for our implementation. Given some initial state values, STOMPC queries UPPAAL STRATEGO for a strategy, which is a sequence of

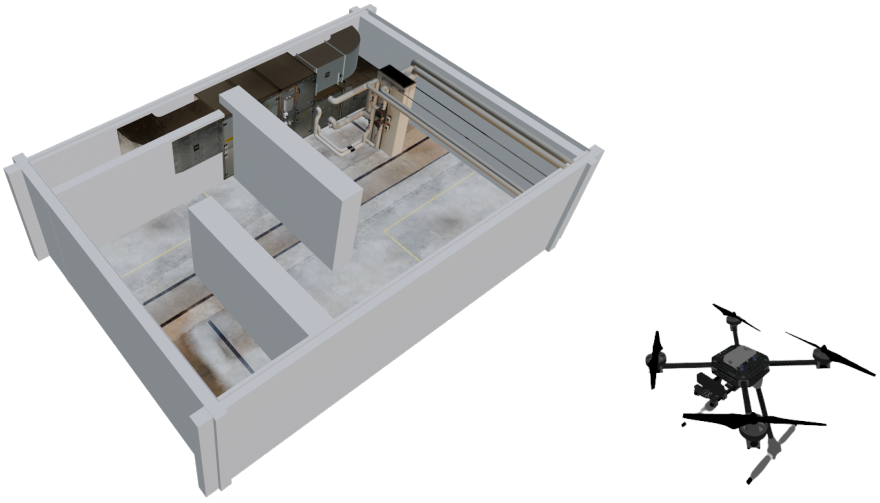
actions. Only the initial action is handled and translated into drone behaviour for execution in the Gazebo simulation environment using ROS. Executing an action results in a new and updated drone EMDP D, which is used to re-query UPPAAL STRATEGO for a new strategy, and thus action to be executed. The new drone EMDP D contains updated state information, i.e. positional information, updated environment map based on depth camera data gathered from the Gazebo environment, and POIs as a result of the applied action. This STOMPC loop continues until the drone has found all pumps in the room.

7. Drone Simulation

7.1. Gazebo

Gazebo is a robotics simulator, that provides a rich simulation environment featuring rendering of scenes, physics-based dynamics, and a diverse range of sensors, enabling accurate representation of real-world scenarios. It allows for simulating and interacting with custom environments and virtual robots, making it an ideal tool to evaluate performance and behavior under various conditions [7].

Gazebo provides a comprehensive simulation environment for our solution. It offers the capability to utilize realistic environment models, and sensors, such as the depth camera used for mapping, which can interact with and perceive the simulated environment. The sensory feedback gathered by the drone in the simulated environment is part of the state representation that is iteratively updated and sent to UPPAAL STRATEGO through STOMPC.



(a) (b)

Figure 5: (a) Simulation environment in Gazebo, (b) virtual drone (MODEL: X500 V2)

The main simulation environment used in our simulations, seen in Figure 5a, is derived from a model initially provided by Grundfos, which we have further modified with

walls using the 3D computer graphics tool Blender. The virtual drone, seen in Figure 5b, came by default through the chosen PX4 configuration. PX4 is a flight control software designed for drones, and it provides various virtual drone models that can be controlled by interfacing with their software [8]. Here, we use ROS to interface with PX4 to control our drone. These customized virtual models are imported into Gazebo, allowing us to create realistic simulation scenarios for the learning agent.

7.2. Robot Operating System (ROS)

ROS is a collection of tools and libraries aimed at simplifying the task of creating robot behavior across various robotic platforms. To enable the exchange of data, ROS uses nodes. At its core, nodes represent specific functionalities of a robot system, that need to be shared with other nodes. Data is shared between nodes using communication channels called topics. Nodes can publish data to a topic by sending data to it, or subscribe to a topic, receiving data from it [9].

Using ROS, it is possible to translate and realize the actions received from STOMPC into actual drone movement. ROS is integrated into the main STOMPC loop, meaning that actions are executed in synchrony with each iteration of strategy querying to UPPAAL STRATEGO. This information exchange between the learning and execution phases of the pipeline is the foundation of online learning. It is here, that we implement an approach to map the environment based on sensor data and to perform our own version of pump detection to emulate how the probabilities of POIs change. After having executed an action, these are state variables that are inserted back into the drone EMDP of UPPAAL STRATEGO by STOMPC, such that the next action we learn is based on an updated drone EMDP.

In the following, we describe our implementation of drone movement, environment mapping, and pump detection.

Drone movement has the logic for moving forward and turning using the *offboard control node* that translates actions into motor actuation on the virtual drone. It gets real-time positional information about the drone by subscribing to various topics that the internal sensor nodes publish data to.

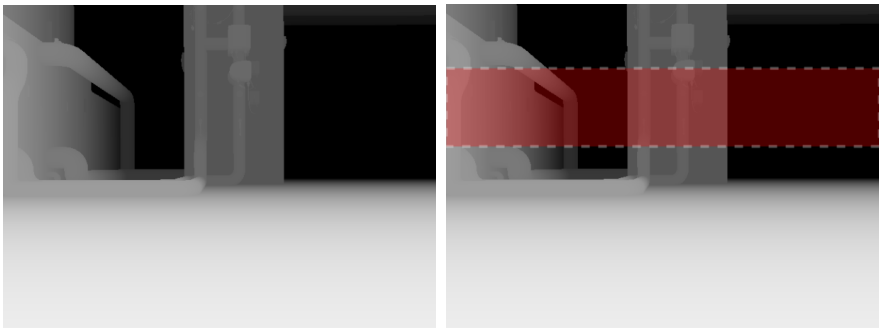
Turning is done based on yaw rotation. To turn, the drone has an internal yaw representation that is incremented or decremented depending on a right or left turn. As the possible directions are north, east, south, and west, there are four corresponding yaw values.

Moving forward is waypoint-based, meaning that the drone will move to a (x, y)-coordinate supplied as input based on its own odometry data. Several factors can lead to inaccuracies relying on odometry such as sensor drift. Therefore, in reality, the distance travelled is often slightly less or more than specified. Over time, these inaccuracies will likely accumulate. We convert positional information from the odometry into an equivalent environment map position after each action. This information, i.e. position, and direction of the drone, is part of the state, that is inserted back into the drone EMDP of UPPAAL STRATEGO by STOMPC. We deem this approach sufficient for a Proof of Concept and the goals we aim to validate in this project. The moving dynamics can be tailored to specific needs in terms of how far we want the drone to move for forward actions. By

default, a forward move translates to moving one meter in the simulation environment, which equates to moving one cell in the environment map.

Wall mapping is the logic for determining wall positions within the environment. We present our mapping algorithm, which is used to populate the environment map with walls. The process starts with a 3D point cloud and ends with a corresponding 2D environment map. This is exactly the environment map that STOMPC inserts into the UPPAAL STRATEGO drone EMDP state after each action.

The *depth camera control node* is used, subscribing to the topic where the internal depth camera sensor node publishes its data. The 3D point cloud is captured using the depth camera (OAK-D Lite) onboard the drone. The camera has a 640x480 pixel resolution. The point cloud has a data point for each pixel. As a result, the captured point cloud has more than 300.000 data points at any given time. We capture a depth image, with a corresponding point cloud, after each action executed by the drone. An example of such a depth image can be seen from Figure 6a. Here, it is clear that a large part of the image contains floor (the space at the bottom of the image), which we are not interested in as we want to produce a 2D map. For this reason, only a confined region of the whole depth image is used, as per Figure 6b. This region captures exactly the data points adequate to map walls in 2D space. The choice of confining the data region also significantly reduces the amount of data needing to be processed. We can now begin the process of calculating distances to each data point and finally use triangulation to assess data point positions within the environment map.



(a) (b)

Figure 6: (a) full depth image, (b) confined (640x120 pixel) region of depth image

Each data point is represented in a (x_p, y_p, z_p)-format denoting the pixel position and the depth to that specific point within the depth image, respectively. Note, here that the z value is not the Euclidean distance from the camera to the point, but is the distance from the camera to the point along the camera's view direction, which is perpendicular to the image plane. In order to transform a data point into an equivalent wall coordinate in the environment map we use triangulation, as per Figure 7.

The first step is to calculate the hypotenuse of the right triangle formed between any data point and the depth camera. This is also the distance between the data point and the depth camera. We do this as follows using the x_p, y_p, and z_p of a data point.

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

Calculating the distance using Equation (4) automatically accounts for phenomena like horizontal displacement. That is, using all the spatial information about the data point allows for the correct distance calculation. Horizontal displacement is especially important to account for when a data point lies in the outskirts of the depth image.

The next step of the triangulation process is to calculate the euclidean angle θ using the x_p and y_p of a data point as follows.

$$\theta = \arctan\left(\frac{y}{x}\right)$$

Based on the distance and Euclidean angle θ, the opposite ω and adjacent α sides of the right triangle can be calculated as follows:

$$\omega = |distance \times \sin\theta|$$
$$\alpha = |distance \times \cos\theta|$$

The opposite ω and adjacent α sides can then be used to determine the position of the data point, relative to the position of the drone, within the environment map. Here, the position of a data point is determined by adding or subtracting the opposite ω and adjacent α by the position (x, y) of the drone depending on its direction. The resulting pair of coordinates is then rounded to the nearest integer, thus denoting the coordinate of the data point within the environment map as a wall.

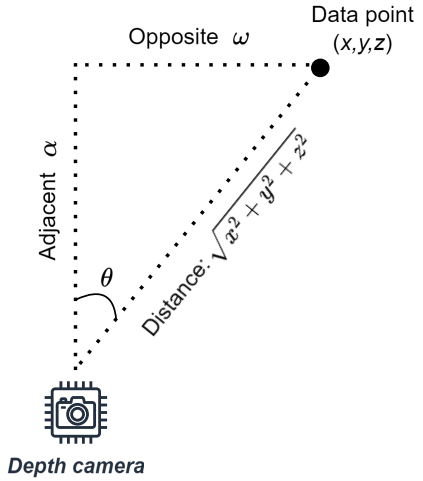


Figure 7: Data point placement by triangulation

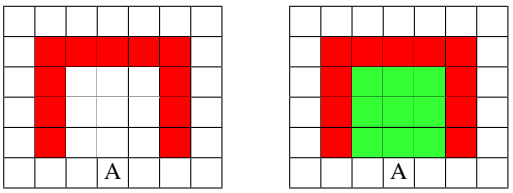
For example, consider a drone with a position $(x, y) = (8, 1)$ and a direction $dir = north$, and a data point with an opposite $\omega = 3.75$ and adjacent $\alpha = 1.8$. The coordinate of the data point within the environment map is then:

$$(x - \alpha, y + \omega) = (8 - 1.8, 1 + 3.75) \approx (6, 5)$$

Therefore, cell (6,5) of the environment map is deemed to be a wall.

The above procedure proceeds for every data point within the confined region shown in Figure 6b. As thousands of data points go through this process, naturally many of these will get rounded to the same integer coordinate. Here, we keep track of how many data points get rounded to each individual integer coordinate. This allows for defining a confidence threshold used to prune inaccuracies and coordinates with a low amount of data points rounded to it.

Free cells in the environment map are mapped last. By default, the wall-mapping mechanism extends beyond the drone's observation area. As such, any cell in the observation area, that has not been mapped as a wall, must therefore naturally be a free cell, as per Figure 8. Thus, we first map walls as seen in Figure 8a, then we use the result hereof to map free cells as seen in Figure 8b. The only exclusion is if an unknown cell is behind a wall within the observation area. In this case, the drone must find a way to gain adequate visibility of the cell in order to map it.



(a) (b)

Figure 8: (a) mapping `wall` cells, (b) then mapping `free` cells

Green cells: free, *red cells*: walls, *white cells*: unknown, *A*: drone

As such, after each action, we build out an environment map with walls and free cells based on the depth camera readings within the simulation environment. This concludes the description of our mapping algorithm that produces an environment map that is inserted back into the drone EMDP of UPPAAL STRATEGO through STOMPC after each action. It is the `map` variable of UPPAAL STRATEGO that receives this update. This allows us to always learn from an up-to-date representation of the simulation environment.

Pump detection is an important part of our solution, enabling the dynamic updating of POIs within our drone EMDP state. As defined in Definition 5, a POI has a position, an examined value, and a probability value. Our pump detection model updates the examined- and probability values of POIs, integrating these updates into the drone EMDP state of UPPAAL STRATEGO via STOMPC after each action.

Our pump detection approach takes inspiration from the experimental results from [1] on a real pump detection model. We define our own abstraction of such an object detection model, that would detect pumps through the use of the onboard camera on the drone. To create an abstraction of such a camera, the drone has a pump detection observation area of size $c \times k$ referring to the number of cells that the drone can see in front of it, as per Figure 9. This observation area is used as a way for the drone to detect POIs. A POI is only visible if it has been within the observation area, i.e. observed.

We define a ground truth set of POIs, that other than a position, examined value, and probability, also has a truth value. This allows us to define how many POIs we want in the environment, and also how many of these should be regarded as actual pumps. Note that UPPAAL STRATEGO never has access to this ground truth set, but only those POIs that have been observed by the drone.

The probability calculation of a POI is based on a measure of the distance between the drone and the POI weighted by the truth value. The weight of the truth value becomes stronger as the distance between the drone and POI becomes smaller. This should reflect, that the confidence in the POI being a pump or not increases as the drone moves closer. As such, if we have defined the POI to be a pump, the probability is more likely to increase as the drone moves closer. If we have defined the POI not to be a pump, the probability is more likely to decrease as the drone moves closer. Noise is built into this process such that the amount the probability decreases or increases can vary. Therefore, sometimes the drone will be able to immediately conclude a POI to be a pump, other times it will have to move closer. The same goes for being able to immediately conclude a POI to not be a pump.

When a POI is within the drone's observation area, our pump detection determines the probability of it being pump as follows:

- Upon seeing the POI for the first time, it is assigned a probability value between, but not including, 0 and 1. This value denotes the probability of it being a pump and is weighted by the distance from the drone and the truth value of the POI.
- With each subsequent observation, this probability is recalculated to reflect the updated probability of it being a pump weighted by the distance from the drone and the truth value of the POI.
- If at any point the probability exceeds 80%, the POI is confirmed as a pump.
- If at any point the probability drops below 20%, the POI is confirmed not to be a pump.
- If at any point the drone is one forward move away, the POI can be directly concluded based on its truth value.

The latter point seeks to abstract over the experiment results of [1], where it was discovered that within a certain distance, the real object detection model would be 100% accurate on positive examples. In our case, that distance is a single forward move away, which translates to approximately one meter.

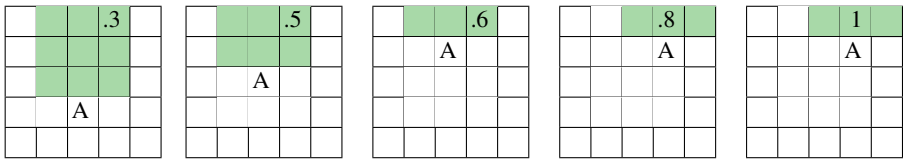


Figure 9: Value of POI increasing as drone A moves closer until determined as pump.
Note: 3x3 observation area (marked by green cells)

Figure 9 shows an illustration capturing how the pump detection works with a $3x3$ observation area. Here, the drone observes a POI that in fact is a pump. The value of the POI, i.e. its probability of being a pump thus increases when it moves closer to it. It does so until the value reaches at least 80%, where the POI gets concluded to be a pump denoted by the use of a probability value equal to 1. Naturally, we could also have shown the opposite example of a POI not being a pump, and its probability value decreasing.

After each action, the pump detection model runs, updating any POIs within the observation area and reinserting them into the drone EMDP state of UPPAAL STRATEGO via STOMPC. The following denotes the UPPAAL STRATEGO variables that are updated by this process. The `poi` state variable receives these updates. If a POI is confirmed as a pump, the environment map is updated to reflect its position, and the `map` state variable is modified accordingly. Additionally, the `pumps_reached` variable is updated to track the number of pumps identified by the drone.

Maintaining an updated view of POIs allows the learning process to effectively guide the drone based on the rewards defined in Definition 9. Furthermore, the pump detection mechanism is essential for determining when all pumps have been located, thereby playing a central role in completing the objective.

8. Experiments

We conduct several experiments to test the applicability of our developed solution against the motivational case of Section 1. Specifically, the experiments should prove or disprove that the scoping done in Section 2 provides enough to serve as a successful Proof of Concept. The objective is for the drone to find the pump(s) in the room, and the main metric is the time taken to complete this task. This includes the time UPPAAL STRATEGO takes to synthesize strategies and time for the virtual drone to execute actions in the simulation environment. As the Gazebo simulation runs in real-time, the times noted for the experiments reflect a real-world counterpart.

The goal of the experiments is firstly to test that our solution works and performs as intended. Secondly, to assess how different configurations of parameters affect the task objective for two different types of reward engineering. Finally, we measure our solution against a baseline implementation that does not use RL. We will also present computation times for strategy synthesis, and how our solution performs in different simulation environments.

For each experiment, we run a simulation five times and note the average time and standard deviation amongst these. As a result of the stochastic nature of UPPAAL STRATEGO, each run can produce a different strategy, which in turn leads to varying simula-

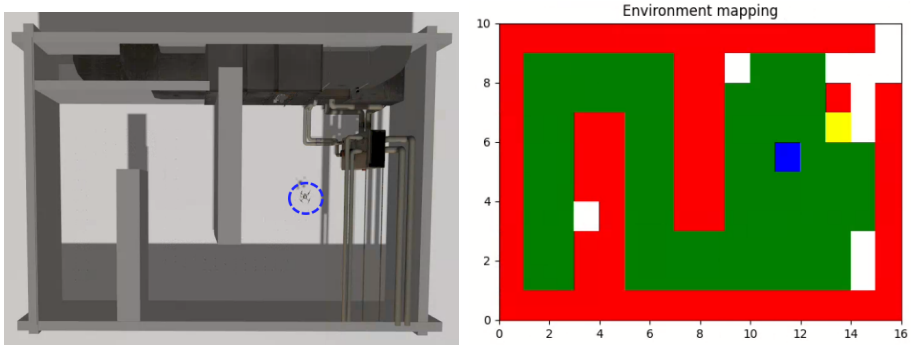
tions. We acknowledge that more simulations per experiment would be ideal, however, using only five allows us to carry out more, and diverse, experiments. As such, the average times and standard deviation are calculated from what should be viewed as a limited sample.

Specification of desktop PC used for experiments: RYZEN 7 5700X 3.4GHz, 32GB RAM. The specifications of such a desktop PC are not comparable to what would often be seen on a drone. However, this setup will suffice to demonstrate the Proof of Concept.

Lastly, note that we use the default learning parameters in UPPAAL STRATEGO.

8.1. Demonstrating the solution pipeline

To perform a run of an experiment, we simply launch PX4 with the Gazebo simulation with a single command. The simulation will load in the configured 3D room of Figure 5a. Then our Python code runs which spins up ROS nodes, defines initial state values, and then starts the STOMPC loop, retrieving new strategies from UPPAAL STRATEGO using updated state variables based on various algorithms as detailed in Section 6. After these supplied control actions are actuated in the simulation environment by the drone, a matrix is visualised representing the current state of the environment map that is built using our mapping algorithm described in Section 7.2. Figure 10 shows screenshots from a run, where we see the simulation environment from above, with the drone marked by the blue circle, and finally, the environment map built out at that point of the simulation. Figure 10a shows a screenshot from the simulation environment in Gazebo at a point where a pump has just been found, and Figure 10b shows the state of the environment map up until this point using the coarse granularity. Here, red denote walls, green denotes free, white denotes unknown, yellow denotes the pump, and blue denotes the drone itself.



(a) (b)

Figure 10: (a) Simulation environment with drone and (b) resulting environment map
Green cells: free, *red cells*: walls, *white cells*: unknown, *yellow cells*: pumps, *blue cell*: drone

See the appendix for more screenshots of this particular run, which shows how the environment map is progressively built out by the drone using our mapping algorithm while executing learned actions provided by UPPAAL STRATEGO through STOMPC. The

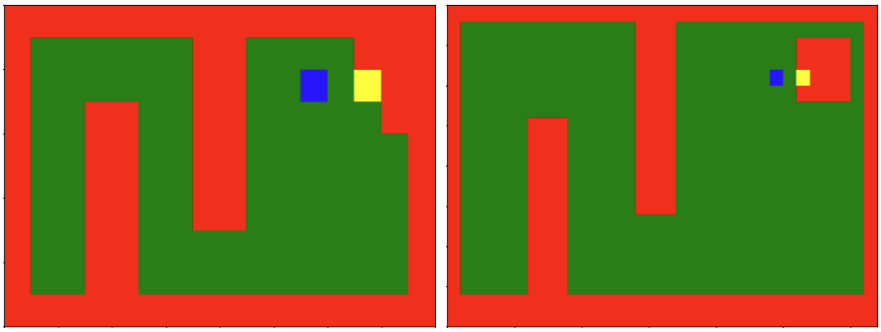
appendix also shows a similar run using the fine granularity for the environment map. This also includes links to YouTube videos of both runs, showcasing the mapping process for coarse and fine granularity.

Most maps produced are subject to minor inaccuracies. There are multiple reasons for this, with a major factor being our reliance on the odometry sensor, which has limitations in accurately estimating the drone's true position. This inaccuracy negatively impacts the mapping mechanism, as the mapping of any cell is done relative to this estimated position of the drone. Due to the relatively short duration of such a simulation, the amount of errors remains insignificant to the Proof of Concept. Most important is that the built-out environment map provides a sufficiently detailed representation of the simulation environment, which provides adequate foundation for learning from it within the drone EMDP.

8.2. Experimental parameters

As evident from Section 6, there are numerous parameters that can be tuned. These include the granularity of the matrix used for mapping, the drone's observation area size used for mapping, the drone's observation area size used for pump detection, and the horizon of the calculated strategy used by STOMPC. Moreover, multiple POIs including pumps, can be added at random locations to test the robustness of the solution being able to find them all in adequate time. We conduct experiments for different combinations of these parameters to assess and understand the effect on the time to complete the objective. For the following, we briefly detail each configuration parameter.

Granularity We work with the two granularities *coarser* and *finer*. The granularity denotes the size of the matrix that we use as the environment map. As such, using a coarser granularity means that the environment representation is more simple, whereas using a finer granularity means that is it more detailed. Here, a more detailed mapping refers to a more accurate representation of the structural layout and properties of the environment.



(a) (b)

Figure 11: (a) coarser environment map, (b) finer environment map
Green cells: free, *red cells*: walls, *yellow cells*: pump, *blue cell*: drone

Figure 11 shows two environment map representations of the simulation environment Figure 5a, and highlights the core difference between the two granularities. As such, the difference between the two maps lies in the granularity of the matrix size used to represent the environment. Here, Figure 11a shows a map using a 10x16 matrix (coarser granularity), whereas Figure 11b uses a 20x32 matrix (finer granularity). By experimenting with these two granularities, we wish to assess the trade-off between the time taken to find the pump(s) and the granularity used.

Pumps/POIs We wish to place multiple POIs, including pumps, within the environment. However, first we experiment with only a single POI that is also a pump. Then we experiment with a multitude of both. This should provide an interesting insight into how the number of POIs effect the time spent to find the pump(s). Additionally, how the time spent finding a single pump compares against that of multiple under different parameter configurations.

Observation areas For the experiments, we use two different observation areas. We use a *mapping area* to map free cells between the drone and mapped walls and model it such that it can be $1x1$, $3x3$, or $5x5$ essentially denoting a decreasing level of pessimism or caution. For example, if we use $1x1$ then we only trust the mapping certainty of the cell immediately in front of the drone. As the mapping area is used to mark free cells, the size of it also determines when the drone is able to move away from an area after having explored it. As such, a larger value means that the drone can move quickly through areas as it explores them faster.

For the pump detection, we have created an abstraction of the real pump detection model. In reality, such a model would utilize a color camera. We model the observation area, *pump area*, of the pump detection to be either $3x3$ or $5x5$ denoting two different ranges of detection capability as detailed in Section 2. As such, this captures the idea of using two different cameras with different qualities. Here, a pump area of $5x5$ denotes the better quality camera. This allows us to experiment with the idea of how the quality of the camera, i.e., pump area value, affects the pump detection.

Horizon For the experiments, we utilize different values for the horizon. The horizon denotes the planning depth used by UPPAAL STRATEGO through STOMPC, i.e., the time frame of which UPPAAL STRATEGO has to learn actions until reaching a goal state. This goal state denotes the state of maximized reward up until that point as detailed in Section 5. Experimenting with different horizons allows us to assess how the planning depth affects the time to find the pump(s). This becomes increasingly interesting when experimenting in combination with different values of granularity and observation area sizes as these affect the size of the space to be mapped and how much is mapped at each step.

Comparing reward engineering approaches For different values of the above configuration parameters, we evaluate the solution with two types of reward engineering. The difference between the two reward types is the exploration reward component. We denote the two types of different reward engineering by *Reward Type 1* and *Reward Type 2*. Here, Reward Type 1 is exactly the one we have described in Section 4. On the other hand,

Reward Type 2 works by rewarding the drone for exploring the unknowns closest to its starting point. As such, the drone should progressively exhaust the immediate unknown state space before moving on to explore farther areas. In essence, the exploration strategy of Reward Type 2 behaves reminiscent of a breadth-first search. Contrary, the exploration strategy of Reward Type 1 is more reminiscent of a depth-first search. By comparing these two, we should get an idea of how two different approaches under the same solution work in reaching the objective. When we write Reward Type 1 and Reward Type 2 it refers to our solution under the reward engineering of these, respectively.

Multiple simulation environments To evaluate our solution further, we wish to conduct experiments by testing our solution in three different Gazebo simulation environments; Figure 5a, Figure 12, and Figure 13. We denote these *Room 1*, *Room 2*, and *Room 3*, respectively.

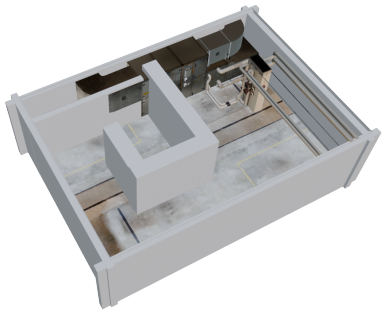


Figure 12: Room 2 - A simulation environment similar to Room 1 but with another structural layout - a ”dead end”

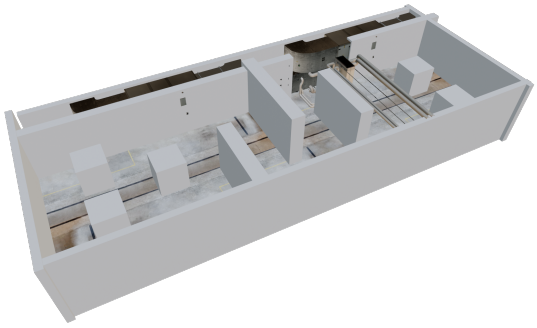


Figure 13: Room 3 - A larger simulation environment

Note that not all experiments carried out for Room 1 will be done for the two others. Instead, we will pick and choose those we find most relevant to test out in the other environments. Experimenting with different simulation environments will test the robustness of the solution and possibly highlight shortcomings.

8.3. Results comparing two reward engineering approaches in Room 1

Table 2 shows the experiment results for various parameter configurations, and two different mechanisms for giving reward, within Room 1. For each experiment, we denote the average time and standard deviation to find the pump(s) within the environment. The average is over the end-to-end real time for the whole pipeline to run, including strategy synthesis, action execution, and data processing. Here, a low average with a low standard deviation is ideal as it indicates that the drone consistently finds the pumps quickly and reliably across multiple runs. Note also that for each time average and standard deviation, we mark in bold the reward type with the better timely outcome.

	Parameter configurations					**Reward Type 1**	**Reward Type 2**
No.	***Granularity***	***Pumps/ POIs***	***Mapping area***	***Pump area***	***Horizon***	***Avg. time ± std. dev***	
1	Coarse	1/1	1x1	3x3	5	**3m53s ± 32s**	4m55s ± 24s
2	Coarse	1/1	1x1	3x3	10	**3m04s ± 23s**	5m49s ± 50s
3	Coarse	1/1	3x3	3x3	10	3m43s ± 53s	**3m47s ± 34s**
4	Coarse	1/1	5x5	3x3	10	**3m35s ± 09s**	3m37s ± 19s
5	Coarse	1/1	5x5	5x5	10	**3m26s ± 06s**	3m41s ± 45s
6	Coarse	1/4	5x5	5x5	10	**3m17s ± 06s**	3m52s ± 13s
7	Coarse	3/4	5x5	5x5	10	**4m21s ± 42s**	5m02s ± 57s
8	Fine	1/1	1x1	3x3	20	**5m45s ± 48s**	21m50s ± 314s
9	Fine	1/1	3x3	3x3	20	**5m03s ± 42s**	6m14s ± 38s
10	Fine	1/1	5x5	3x3	20	**4m18s ± 29s**	4m59s ± 53s
11	Fine	1/1	5x5	5x5	20	**4m01s ± 35s**	4m19s ± 27s
12	Fine	1/4	5x5	5x5	20	**4m08s ± 10s**	5m59s ± 38s
13	Fine	3/4	5x5	5x5	20	6m11s ± 74s	**5m34s ± 16s**

Table. 2: Reward Type 1 and 2 under different parameter configurations in Room 1

For the case of coarser granularity, we see that Reward Type 1 generally outperforms the Reward Type 2 learner with the exception of one instance. For Reward Type 1, we see that the average time to find the pump(s) does not vary considerably. This is likely due to the depth-based exploration strategy of our solution, which allows for exploring faster while looking for POIs. For Reward Type 1, we also see a general pattern in the standard deviation decreasing as the size of observation areas increases. In this case, a smaller standard deviation indicates that there is less variability from the average, i.e. our solution produces increasingly more stable results under Reward Type 1. This stability is due to two factors. Firstly, an increased observation area for mapping allows the drone to cover more ground and map free cells faster, i.e. explore quicker. Secondly, an increased

observation area for pump detection detects POIs at a greater distance, which ultimately allows the drone to find any pump(s) faster.

For experiment number 6, we added more POIs to the environment while keeping the number of pumps the same. Interestingly, both reward engineering types remained rather unaffected by the addition of more POIs compared to the previous experiments with only a single POI. This partly suggests that no time is wasted on POIs with a sufficiently low probability of being a pump, i.e., the POIs that end up not being pumps. Alternatively, it might be that the drone's exploration path naturally brings it close enough to these POIs to examine them incidentally, meaning the presence of additional POIs doesn't significantly impact the overall exploration time. This captures the ideal behaviour of a real pump detection model, which we aim to emulate with our implementation as detailed in Figure 9. This behaviour is attributed to our reward engineering, as per Definition 9, that aims to make it more attractive to follow POIs with a high probability of being a pump. However, if this value decreases, it becomes less attractive to follow it compared to exploring.

In the last experiment with coarse granularity, more pumps were added, purposefully spread out to force the drone to explore a substantial part of the environment. Naturally, this led to an increase in the average time for both Reward Type 1 and Type 2, as more time was required to locate and reach these pumps. Additionally, we observed an increase in the standard deviation for both reward types. This is likely because the randomness inherent in the simulations sometimes resulted in the drone taking a faster overall route, while other times it followed a slower one.

Moving on to the finer granularity, we see that Reward Type 1 on average finds the pump faster than Reward Type 2 in all cases. This average decreases when the observation areas increase as the drone is able to explore more at a time, which in turn allows it to explore more quickly and thus find the pump faster. The clear difference between the exploration strategies of Reward Type 1 and Reward Type 2 becomes evident with experiment number 7, where Reward Type 2 on average takes almost 4x the time of Reward Type 1. Here, with a small mapping area of $1x1$, the drone can only map one free cell at a time. Combining this with the coarser granularity and the exploration strategy of Reward Type 2 drives up the average time exceedingly. As for the coarser granularity, we see that once the observation areas reach a certain size, the average times of Reward Type 1 and Reward Type 2 become increasingly similar. As such, it becomes evident that as the size of the observation areas increases, the core effect of the exploration strategies decreases. For experiment 11 we added more POIs to the finer granularity setting. We see here, that Reward Type 1 remains rather unaffected as it did for the coarse granularity.

In the last experiment with fine granularity, more pumps were again added and spread out to force the drone to explore a substantial part of the environment. Reward Type 1 recorded the lowest single time to find all pumps (not shown), but Reward Type 2 had a lower average time overall and standard deviation. Thus, when pumps are widely dispersed and more thorough exploration is required, Reward Type 2 proves to be the better option. This is due to its BFS-like exploration strategy, which in this case leads to more consistent and efficient results.

8.4. Results for computation times: our solution

Having performed multiple experiments showing average run times to complete the objective, we now shift to computational times and amount of actions performed. For the

following, we use the same mapping area and pump area for all, but we change the granularity and horizon to assess the affect hereof on the computational times, i.e. strategy synthesis times.

	Parameter configurations					**Our solution**	
No.	***Granularity***	***Pumps/ POIs***	***Mapping area***	***Pump area***	***Horizon***	***Total avg. comp. time (sec)***	***Avg. comp. time (sec)***
1	Coarse	1/1	5x5	5x5	5	4.99	0.13
2	Coarse	1/1	5x5	5x5	10	10.72	0.25
3	Coarse	1/1	5x5	5x5	20	23.15	0.49
4	Fine	1/1	5x5	5x5	5	6.21	0.14
5	Fine	1/1	5x5	5x5	10	9.59	0.26
6	Fine	1/1	5x5	5x5	20	20.01	0.51

Table. 3: Our solution in Room 1: average computational times and actions

In Table 3 we present two metrics: the average total computation time spent on strategy synthesis during a run and the average computation time for a single strategy synthesis call. These should give an idea of how much time out of a run is spent on the computation of strategies. Evident from these results, is that when the horizon increases the computation times increase. This is as expected as it requires computing a strategy consisting of more actions.

Interestingly, as each computation takes fairly little time across all the experiments above (under a second) in the kind of granularity we are working on in this case, it would allow for the synthesis of strategies during action execution instead of when the action has been executed. This could be possible as an action takes approximately between 1 and 2 seconds to execute meaning that the strategy synthesis likely could fit into here.

8.5. Baseline implementation and results

This project has hypothesized that RL is an effective technique for tackling the problem of a drone autonomously mapping and localizing pumps in an unknown environment. Therefore, we should evaluate our solution against a baseline controller not employing RL. We consider a baseline inspired by the Wavefront algorithm[10]. Our algorithm works by propagating a ”wave” from an initial cell in the room, systematically considering neighboring cells in an outward manner. It ensures that the closest cells to the starting position are explored first, expanding the frontier uniformly. Closest cells are determined with Manhattan distance and paths through wall cells are discarded. The algorithm terminates when the drone lands in the proximity of a POI concluded to be a pump. As the frontier expands relative to its starting position in a Breadth-first search (BFS) manner, the further from the starting position the pump is, the longer the task will naturally take. However, this is also the case for our solution.

We experiment with this baseline using various values of the parameter configurations used for our solution in Section 8.3. Note that the baseline also uses the mapping algorithm we have developed, and the same pump detection mechanism as our solution.

The experiments are conducted in Room 1 for both coarser and finer granularity of the environment map with the pump placed in the same place as the previous experiments. Results can be seen in Table 4.

	Parameter configurations				**Baseline**
No.	***Granularity***	***Pumps/ POIs***	***Mapping area***	***Pump area***	***Avg. time* $\pm$ *std. dev***
1	Coarse	1/1	1x1	3x3	16m08s $\pm$ 80s
2	Coarse	1/1	3x3	3x3	5m45s $\pm$ 20s
3	Coarse	1/1	5x5	5x5	3m40s $\pm$ 07s
4	Fine	1/1	1x1	3x3	73m39s
5	Fine	1/1	3x3	3x3	19m33s $\pm$ 35s
6	Fine	1/1	5x5	5x5	14m05s $\pm$ 92s

Table. 4: Baseline experiments performed in Room 1

In the coarse granularity setting, the smallest configuration of observation areas in Experiment 1 resulted in an average time of 16m08s to find the pump. As the size of the observation areas increased, the average time to locate the pump decreased. Notably, when the observation areas reached their maximum size in Experiment 3, the average time of the baseline became increasingly similar to our solution. This is inherently due to the larger observation areas enabling faster exploration combined with a coarser environment representation, which means fewer cells to explore. In this setting, the behavior of our solution and the baseline becomes so similar because there is little room for their algorithmic differences to manifest due to the small amount of environment to be explored and the large observation area covering much of it at once. The results become more interesting when increasing the granularity from coarse to fine.

In the fine granularity, we see an exceedingly high time to find the pump for experiment 4, namely 73m39s. The reason for this is due to the core behaviour of the baseline algorithm, which seeks to exhaustively explore the state space in an outward manner. When combining this behaviour with a fine granularity and mapping fewer free cells at a time, it means that the exploration process becomes significantly slower. The fine granularity results in a higher number of cells the drone must examine, while the small mapping area limits the number of free cells that can be mapped in a single step. Using such a small observation area essentially means that the time taken to find the pump runs linearly in the amount of cells of the environment map. Due to this inherently time-consuming process, this experiment was only conducted once, thus presenting no standard deviation. For the two remaining experiments 5 and 6, we see the recurring pattern in that increasing the size of the observation areas decreases the time taken to find the pump. This aligns with most of the behaviour seen for our solution, under both Reward Type 1 and Type 2 in Section 8.3.

In general, for both our solution and the baseline, increasing the observation areas results in less time taken to find the pump. This is expected, as larger observation areas allow the drone to explore more quickly and detect pumps from farther away. The main takeaway is that while the best configurations of both our solution and the baseline are

relatively similar, the worst of each is very much different in favor of our solution. It is evident from the baseline that there is a significant increase in time when moving from coarse to fine granularity. This is not the case for our solution, where the average time only increases slightly. Once we switch to the coarser environment representation, our solution remains remarkably better in all configurations compared to the baseline.

It becomes clear that a learning-based method like the one we have proposed is much more effective in completing the objective compared to a more naive approach, like the baseline. In such a naive implementation, where the drone is not given rewards to reinforce the desired behavior, the exploration is less efficient and driven by exhaustively covering the environment rather than making intelligent and informed decisions. The advantages of RL become prevalent when dealing with more complex environments and finer granularity, where the drone's ability to learn and adapt leads to significantly more efficient exploration and task completion.

8.6. Results in Room 2

Several runs were performed in Room 2, with the pump placed in the same location as in Room 1. However, here the room layout permits the drone to enter a dead end if it continues the path up from its starting position. When it thereafter needs to explore the rest of the room it needs to navigate back through many cells that have been marked free. This challenges the reward mechanism which gives a reward for exploring unknown cells. The experiments were run with both a mapping area and pump area of 5x5 and a horizon of 30.

For this setup, we recorded an average time of 7m to locate the pump, with a standard deviation of 208s. The lowest recorded time was 2m18s, whereas the highest was 9m57s. From the start, the drone can either move towards the dead end or the remaining part of the environment. This choice will be random as the drone does not know there is a dead end, but only that there is unknown space to explore in both directions. If it happens to go down the dead end, the time to find the pump will increase significantly as seen from the times presented. Important to note too, is that we needed to use a horizon of 30 to be able to escape the dead end. As such, the horizon must be high enough to be able to synthesize a strategy that gets out of the dead end and through the prior explored space. As a result of the increased horizon, the computation times related to strategy synthesis will increase, as evident from Table 3. This could also drive up the time to find the pump.

8.7. Results in Room 3

We further conducted experiments in Room 3, a much larger and more complex simulation environment. Here, we placed two pumps at each end of the environment to test how the solution would be affected by having to go through a largely explored part of the environment in order to get to the other side, that had not been explored yet. We encountered problems when the drone had to leave a heavily explored area in order to get to an unexplored area.

What we found was that in such a large environment, the horizon becomes a very important variable. We had to dynamically adjust it during runs such that the drone would be able to get out of certain explored regions. Figure 14 shows the map generated from

one of these runs in Room 3. Here, one pump has already been found (marked by yellow) by the drone (marked by blue). However, there is another pump located at the left-most unexplored part of the room (the white cells left-most in the map). Here, the drone has to travel through many cells of which it will not get rewarded during its path over to the unexplored part.

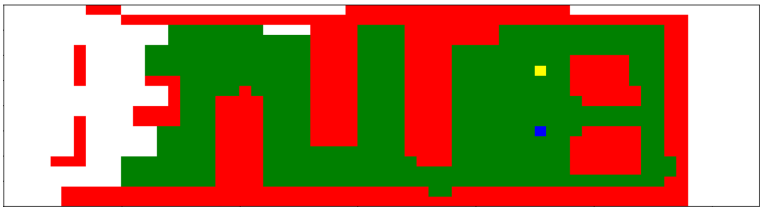


Figure 14: Room 3: A screenshot of environment map from a run in the larger room

This imposed a central challenge in using our solution for such a large environment. It becomes clear, for our solution to be applicable for a situation where pumps are placed so far apart in such a large environment, we have to manage the horizon dynamically, possibly in combination with an extension to the current reward engineering approach.

9. Conclusion

In this thesis, we explored the integration of UPPAAL STRATEGO with a Gazebo simulation environment to enable autonomous drone navigation and pump detection. Our approach utilized an Euclidean Markov Decision Process (EMDP) to model the problem. By leveraging STOMPC as a co-simulator between UPPAAL STRATEGO and Gazebo, we enabled continuous strategy synthesis and online learning.

The STOMPC co-simulation approach allowed for real-time updates to state information based on external algorithms, simplifying the modeling of the EMDP. We implemented our own SLAM-like mapping using depth camera data and created an abstraction for a pump detection mechanism. This was suitable for a Proof of Concept, where most components of the solution are simplified but could be scaled up.

Our approach enabled the drone to accurately navigate the environment and identify pumps, continuously adapting its strategies as new data was gathered. Experiments demonstrated the effectiveness of this approach, showing improvements in task completion time compared to a baseline method. However, the experiments also indicated that the solution requires further refinement to be applicable in larger and more complex environments. These preliminary experiments have been an investigation of how different approaches to giving the agent reward impact its task, as well as the quality of what the drone can sense about the environment at a given time. These investigations are obvious to continue and understand what would reach a solution that makes RL an efficient method for the problem case. As this is a Proof of Concept, there are naturally several areas that could be developed further to build a more comprehensive solution.

In conclusion, this thesis has shown that co-simulation between UPPAAL STRATEGO with Gazebo through STOMPC for continuous strategy synthesis and online learning can significantly enhance autonomous drone navigation and pump detection. This co-simulation approach offers an interesting foundation for future research and possible real-world applications.

10. Related work

In recent years, there has been growing interest in using learning-based methods for the control and navigation of autonomous robots in uncertain environments, as per [11], [12], and [13]. Most of these studies prioritize collision avoidance and ensuring the safety of the agent during operation. Yet they do not involve building or utilizing maps of the environment in their methods.

The paper [11] presents an autonomous navigation and obstacle avoidance algorithm for unmanned aerial vehicles (UAVs) in complex environments, addressing the challenge of achieving accurate path planning. The proposed algorithm utilizes deep reinforcement learning (DRL) within an actor-critic framework. A set of reward functions is designed to adapt to autonomous navigation and obstacle avoidance tasks in complex environments. Additionally, to mitigate decision-making bias stemming from incomplete observability, a gate recurrent unit network is employed to enhance perception ability, improve perception representation, and enhance real-time decision-making. Experimental results validate the algorithm's effectiveness in achieving adaptive UAV flight adjustment during navigation and obstacle avoidance tasks while demonstrating enhanced generalization ability and training efficiency in complex environments.

Other studies, like [14], have integrated established mapping techniques like SLAM (Simultaneous Localization and Mapping) into RL algorithms to improve navigation in unknown environments. The paper introduces a path-planning algorithm for mobile robots to reach target locations in unknown environments, leveraging onboard sensors. Specifically, a deep reinforcement learning motion planner is designed and evaluated, employing continuous linear and angular velocities for navigation, based on the deep deterministic policy gradient (DDPG) approach. Furthermore, the algorithm incorporates environmental knowledge from a grid-based Simultaneous Localization and Mapping (SLAM) algorithm to shape the reward function. This enhancement aims to improve the convergence rate, escape local optima, and reduce collisions with obstacles. A comparative analysis is conducted between reward functions shaped based on SLAM-derived maps and those without map knowledge. Results demonstrate promising results, including reduced learning time, fewer collisions, and higher success rates in reaching target locations compared to standard RL approaches without mapping.

The application of UPPAAL STRATEGO for strategy synthesis across diverse problem domains is increasingly being studied. The STOMPC framework, with UPPAAL STRATEGO as its controller synthesis engine, allows for continuous online strategy synthesis. Despite the capabilities of the framework, it remains widely unused and has few documented use cases.

The work presented in [5] considers dynamic route planning for fleets of autonomous robots doing tasks in a shared environment. Their problem is formalised as an Euclidean Markov Decision Process (EMDP) realised in UPPAAL STRATEGO for near-optimal route

planning synthesis using Q-learning. They present an online and distributed approach, using Model-Predictive Control (MPC), to facilitate fast and scalable replanning, demonstrating significant improvements in makespan over conventional planning methods.

In [15], the authors address the challenges of managing stormwater detention ponds to minimize pollution and prevent emergency overflows. They apply formal methods to synthesize safe and optimal control strategies for detention ponds, ensuring better utilization of current infrastructure while minimizing pollution. Detention ponds are modelled as hybrid Markov decision processes and UPPAAL STRATEGO is used for control strategy synthesis minimizing pollution and guaranteeing no emergency overflows. STOMPC enables online and continuous synthesis of control strategies, allowing for adaptive responses to changing weather conditions and possibly other operational requirements. Furthermore, the STOMPC allows for combining strategy synthesis with simulation, as demonstrated by the integration with the SWMM simulator in this study. Simulation results demonstrate the efficacy of their contribution by UPPAAL STRATEGO being able to learn optimal strategies that prevent emergency overflows and improve sedimentation periodically.

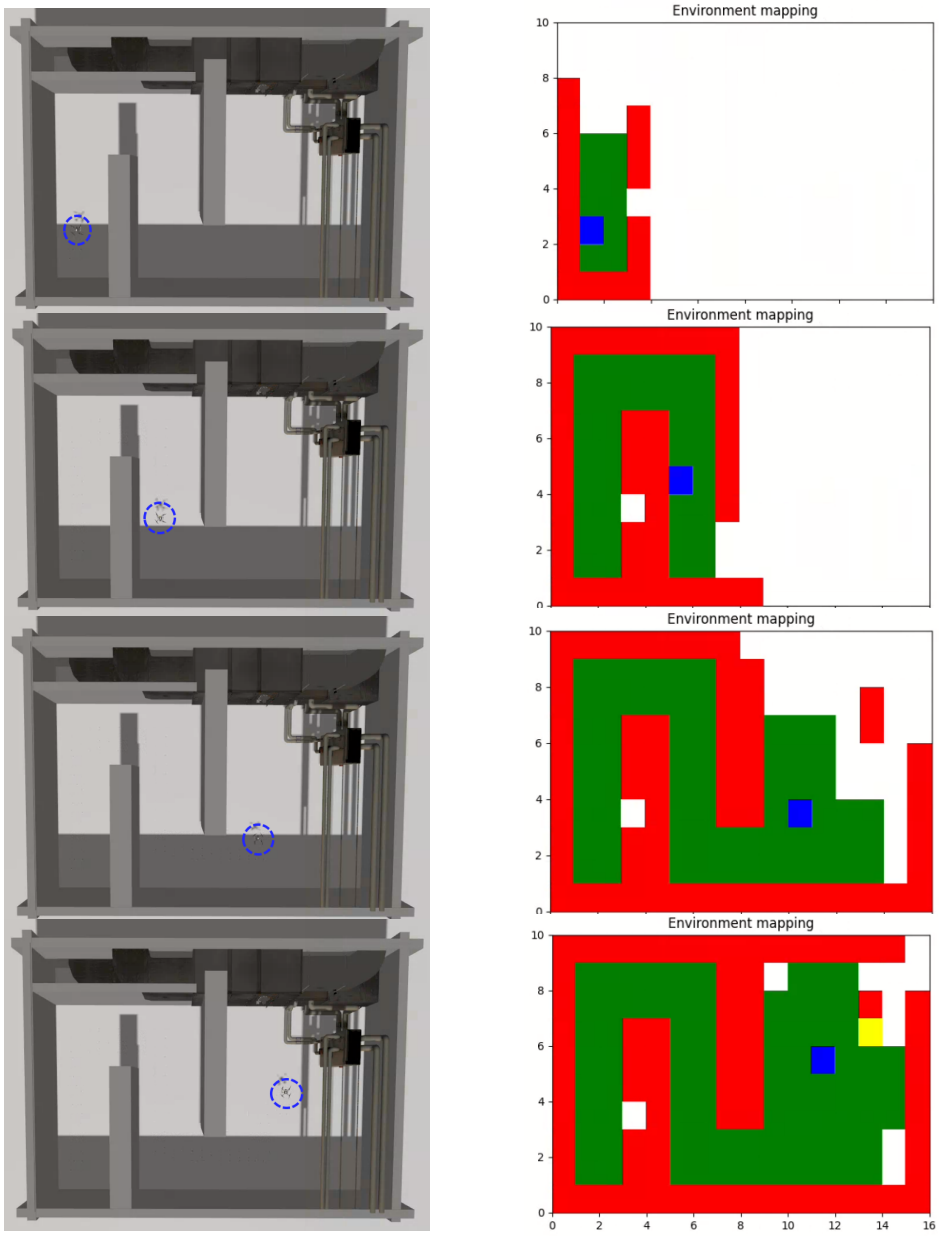
References

1. K. L. H. Pedersen, M. K. Axelsen, P. C. Greve, and T. G. S. Lauritsen, "Reinforcement learning through uppaal in simulation environments: Pump identification by drones in technical rooms," 2023.
2. R. Sutton and A. Barto, *Reinforcement Learning: An introduction*. The MIT Press, 2018.
3. M. Jaeger, P. G. Jensen, K. G. Larsen, A. B. Legay, S. Sean, and J. H. Taankvist, "Teaching stratego to play ball," 2019.
4. G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," 2004.
5. S. Bøgh, P. G. Jensen, M. Kristjansen, K. G. Larsen, and U. Nyman, "Distributed fleet management in noisy environmentsvia model-predictive control," *ICAPS 2022*, 2022.
6. M. Goorden, P. Jensen, K. G. Larsen, M. S. J. Srba, and G. Zhao, "Stompc: Stochastic model-predictive control with uppaal stratego," *Automated Technology for Verification and Analysis. ATVA 2022*, 2022.
7. "Gazebo." `https://gazebosim.org/home`, last accessed 08.05.2024.
8. "Open source autopilot for drones - px4 autopilot." `https://px4.io/`, last accessed 08.05.2024.
9. "Ros: Home." `https://www.ros.org/`, last accessed 08.05.2024.
10. "Wavefront expansion algorithm." `https://en.wikipedia.org/wiki/Wavefront_expansion_algorithm`, last accessed 04.06.2024.
11. S. Zhao, W. Wang, J. Li, S. Huang, and S. Liu, "Autonomous navigation of uav through deep reinforcement learning with sensor perception enhancement," *Mathmatical Problems In Engineering Vol. 2023*, 2023.
12. D. Du, S. Han, N. Qi, H. Ammar, J. Wang, and W. Pan, "Reinforcement learning for safe robot control using control lyapunov barrier functions," 2023.
13. D. Tompos and B. Nemeth, "Safe trajectory design for indoor drones using reinforcement-learning-based methods," *SACI 2023*, 2023.
14. K. Mustafa, N. Botteghi, B. Sirmacek, M. Poel, and S. Stramigioli, "Towards continuous control for mobile robot navigation: A reinforcement learning and slam based approach," *XXIV ISPRS 2019*, 2019.
15. M. A. Goorden, K. G. Larsen, J. E. Nielsen, T. D. Nielsen, M. R. Rasmussen, and J. Srba, "Learning safe and optimal control strategies for storm water detention ponds," *IFAC 2021*, 2021.

Appendix

Simulation run - coarse granularity[1]

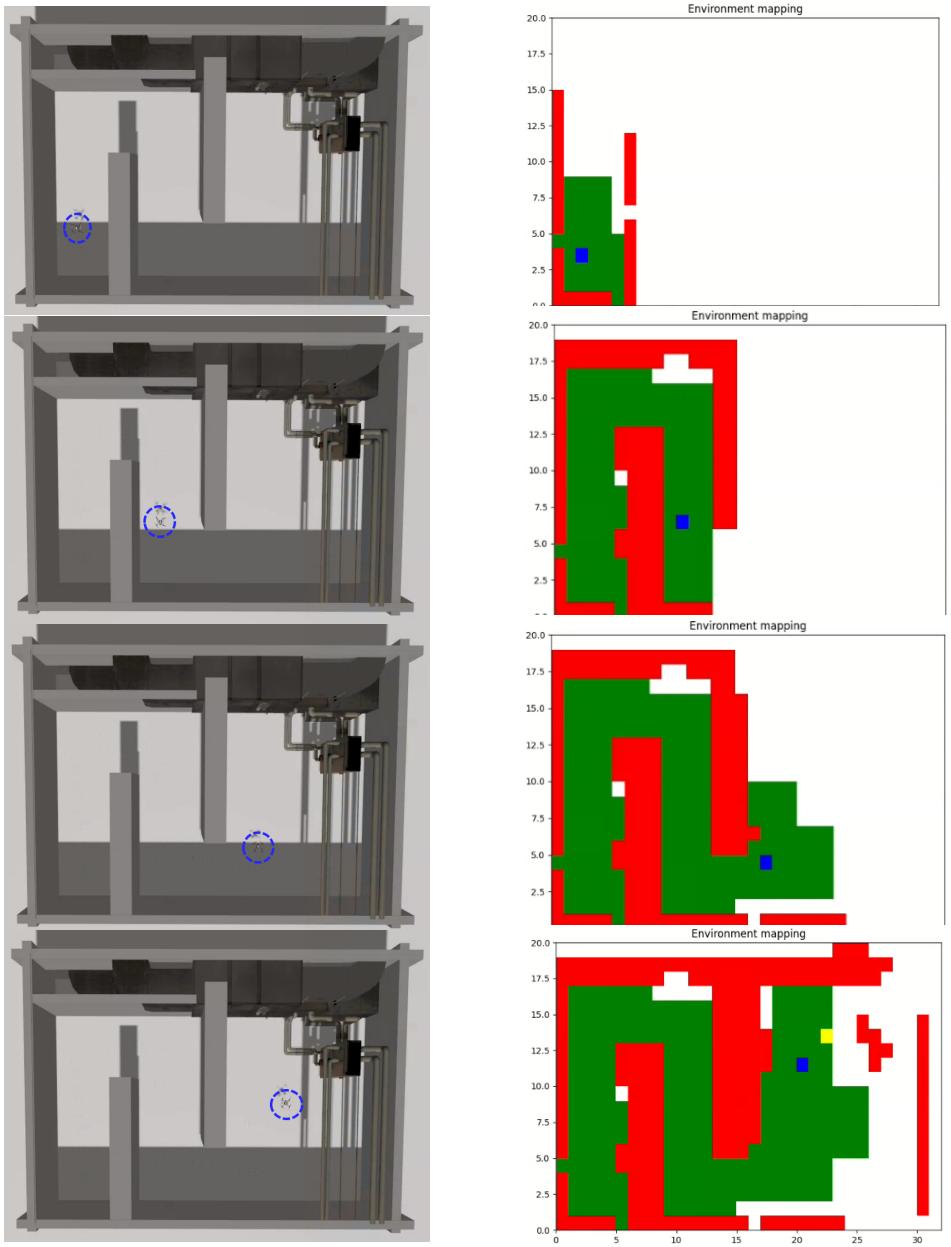
Below shows the progress of the environment map during a run in the Gazebo. The left side are screenshots from Gazebo and the corresponding environment map to the right.



[1] https://youtu.be/HQdq0NcSaHA

Simulation run - fine granularity[2]

Below shows the progress of the environment map during a run in the Gazebo. The left side are screenshots from Gazebo and the corresponding environment map to the right.



[2] https://youtu.be/VaMy4KareaU