A Game-theoretical Approach for Distributed Cooperative Control of Autonomous Underwater Vehicles

Thesis by
Yimeng Lu

In Partial Fulfillment of the Requirements

For the Degree of

Masters of Science

King Abdullah University of Science and Technology
Thuwal, Kingdom of Saudi Arabia

May 2018
The thesis of Yimeng Lu is approved by the examination committee

Committee Chairperson: Jeff S. Shamma
Committee Members: Bernard Ghanem, Burton Jones
For my beloved mother and father
ABSTRACT

A Game-theoretical Approach for Distributed Cooperative Control of Autonomous Underwater Vehicles

Yimeng Lu

This thesis explores a game-theoretical approach for underwater environmental monitoring applications.

We first apply game-theoretical algorithm to multi-agent resource coverage problem in drifting environments. Furthermore, existing utility design and learning process of the algorithm are modified to fit specific constraints of underwater exploration/monitoring tasks. The revised approach can take the real scenario of underwater monitoring applications such as the effect of sea current, previous knowledge of the resource and occasional communications between agents into account, and adapt to them to reach better performance.

As the motivation of this thesis is from real applications, in this work we emphasize highly on implementation phase. A ROS-Gazebo simulation environment was created for preparation of actual tests. The algorithms are implemented in simulating both the dynamics of vehicles and the environment. After that, a multi-agent underwater autonomous robotic system was developed for hardware test in real settings with local controllers to make their own decisions. These systems are used for testing above mentioned algorithms and future development of other underwater projects.

After that, other works related to robotics during this thesis will be briefly mentioned, including contributions in MBZIRC robotics competition and distributed control of UAVs in an adversarial environment.
ACKNOWLEDGEMENTS

I would first thank Prof. Jeff Shamma for his advising during this thesis and throughout my master’s study at KAUST. His insightful advice, broad knowledge, patience, and tolerance have always encouraged me to go further and aim higher during these two years and to the future. It’s my honor being advised by you.

I also would like to thank Prof. Bernard Ghanem and Prof. Burton Jones for serving as my committee. Thank you for your valuable advice and further discussions about this research.

Thanks to all RISC lab members for your supporting, by all means. I would like to especially thank Dr. Mohamed Abdelkader, Fadl Abdellatif, and Kuat Telegenov for your help during my experiments. Thanks to Siqing Hou for his valuable advice during software design phase. Thanks to KAUST Island Recreation Center for providing me test field. Thanks to Thomas Hoover from CMOR for the help of marine robot engineering. Without your help, I might take much more time to finish my work.

I would also like to take this opportunity to thank all my friends and colleagues here at KAUST. Thanks for those amazing memories with you. I learned here, had fun here, and the most important, I grew up here. Let this thesis conclude my two years at KAUST, and be a start of promising tomorrow.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examination Committee Page</td>
<td>2</td>
</tr>
<tr>
<td>Copyright</td>
<td>3</td>
</tr>
<tr>
<td>Abstract</td>
<td>5</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>6</td>
</tr>
<tr>
<td>List of Figures</td>
<td>10</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>12</td>
</tr>
<tr>
<td><strong>2 Game-theoretical Approach Background</strong></td>
<td>17</td>
</tr>
<tr>
<td>2.1 Components for game-theoretical formulation</td>
<td>17</td>
</tr>
<tr>
<td>2.1.1 Players</td>
<td>17</td>
</tr>
<tr>
<td>2.1.2 Actions</td>
<td>17</td>
</tr>
<tr>
<td>2.1.3 Utility</td>
<td>18</td>
</tr>
<tr>
<td>2.1.4 Learning mechanism</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Essential concepts from game theory</td>
<td>18</td>
</tr>
<tr>
<td>2.2.1 Nash Equilibrium</td>
<td>19</td>
</tr>
<tr>
<td>2.2.2 Efficiency of equilibrium</td>
<td>19</td>
</tr>
<tr>
<td>2.2.3 Potential games</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Game-theoretical formulation</td>
<td>20</td>
</tr>
<tr>
<td>2.3.1 Utility design</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2 Learning mechanism</td>
<td>22</td>
</tr>
<tr>
<td><strong>3 Game-theoretical Approach for Distributed Coverage Problem</strong></td>
<td>23</td>
</tr>
<tr>
<td>3.1 Problem setup</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Utility and learning mechanism design</td>
<td>25</td>
</tr>
<tr>
<td>3.3 Applying the method to underwater sensor coverage</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Modifications of the algorithm</td>
<td>27</td>
</tr>
<tr>
<td>3.4.1 Adapt to effect of sea current</td>
<td>28</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>A.3.1 Data recording and representation</td>
<td>69</td>
</tr>
<tr>
<td>A.3.2 List of software packages and OS images</td>
<td>70</td>
</tr>
<tr>
<td>A.3.3 Carrying list for outdoor test</td>
<td>70</td>
</tr>
<tr>
<td>A.3.4 Equipment list and backups</td>
<td>71</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

3.1 A model of sensor coverage problem ........................................ 24
3.2 Multiple drifting resources. Agent locations at $t = 0$ and $t = 8000$ 27
3.3 Sea current reaction. Left picture operates the original algorithm while right picture uses modified utility ........................................ 29
3.4 Comparison of global utilities over time between with/without occa-

sional information exchange .................................................... 30
3.5 Search in convex area ......................................................... 31
3.6 Search in convex area: result ............................................... 31

4.1 Simulation environment screenshot ........................................ 35
4.2 Simulation software architecture ............................................ 36
4.3 Multiple drifting resource simulation result ............................. 37
4.4 Adapt to sea current simulation result ..................................... 38
4.5 Vision-based localization system ......................................... 41
4.6 Hardware system architecture ............................................. 42
4.7 Hardware system overall look ............................................. 43
4.8 AUV trajectory in open-loop test ........................................ 44

5.1 Feature extraction and box bounding ...................................... 46
5.2 Laser width detector system ............................................... 48
5.3 Quadcopters used for experiments ....................................... 49

A.1 Class diagram of algorithm in ROS ...................................... 54
A.2 ROS/Gazebo nodes .......................................................... 56
A.3 x and y directions in Gazebo ............................................... 57
A.4 USBL localization system ................................................... 62
A.5 Serial adapter ................................................................. 63
A.6 Transponder installed BlueROV ......................................... 64
A.7 DJI matrice 100 with localization system ......................... 65
A.8 Close view of camera ....................................................... 66
A.9 Outdoor test of the markers ............................................... 67
A.10 Hardware system ROS node structure .......................... 68
A.11 Network structure ............................................. 69
Chapter 1

Introduction

There is a significant number of ocean monitoring applications performed by autonomous vehicles on or under the sea. These include resource exploring, leakage surveillance of submarine pipelines, study of marine animals, archeology, oceanography and rescue, and even harbor defense [1]. The smart ocean concept which utilizes intelligently controlled networks of mobile sensors to optimize the information gathering in the ocean can support marine research by creating a dynamic model for real-time observations and predictions. All of these applications require autonomous agents such as an ASV (Autonomous Surface Vehicle) or AUV (Autonomous Underwater Vehicle) carrying different types of sensors operating in the ocean [2], as these missions often last long and require responding rapidly to changing environments.

For applications in the ocean, there are a few different features and constraints compared with the ground and aerial vehicles. For AUVs operating underwater, as water disables or deteriorates many types of communications (WiFi, radio, etc.), exchanging information or even sending commands to the agents becomes energy-consuming, time-inefficient and unreliable. The underwater world is full of uncertainties, such as sea current [2] and obstacles [3], and even the resources we are looking for might not be static but change over time [1]. GPS is the primary localization approach for both ground and aerial applications, but it cannot be used underwater. Dynamics of the vehicle and interaction with the environment is different from UAVs (Unmanned Aerial Vehicles), including the effect of sea current which might dramatically increase the energy consumption of the vehicles. Here we consider a group of
agents rather than a single agent as typically the area to explore is big and the sensor range of underwater mobile sensors is small so that sometimes even a group of agents cannot monitor every part of it at the same time. Sometimes we model the environment as an uncertainty heatmap where uncertainty grows with time, and we want to keep the uncertainties of the whole map small. Thus, they need to move smartly to guarantee persistent coverage [4, 5].

From the specifications above, our objective is to efficiently explore and consistently cover resources or information we are interested in within a dynamical underwater environment. Viewing this problem from the perspective of persistent sensor coverage, we recommend to deploy sensors on multiple agents distributively and use possibly less communication between them as communication in water can be costly and unreliable. Rather than having a central node to plan and send commands to all agents, which is impractical because of above issues, agents will make their own decisions only based on their own local sensory information. We want our system to still meet optimal and robust coverage performance while each agent is actually locally controlled. As the resource heatmap itself can drift over time, and we might not have this information as a priori, the network of agents should be able to adapt to this changing environment. Moreover, there might be different scenarios for different applications, and the algorithm for controlling the agents should be flexible to adapt to these different specifications. Since we perform marine exploration tasks in big open water areas, energy consumption of AUVs needs to be taken into consideration while planning their motion.

Many works deal with optimal sensor coverage problem [6] as a paradigm for the above scenario.

Reference [6] proposed both central and distributed algorithms for coverage control of mobile agents, where Voronoi partitions are used to find optimal locations and Lloyd descent is used to drive the agents to those locations. Agents with compu-
tation, communication and control capabilities compute and maintain their Voronoi cells while sensing resources and other agents nearby. This algorithm can be performed distributedly in an asynchronous fashion, but it is designed for a structured environment where the resource is static. Also, to perform Voronoi partition, the range of sensor and communication need to be adjustable to locate all the neighbors of an agent. Decentralized Voronoi partition with information exchanged during sharing event, which will be requested by vehicles occasionally, is performed to adaptively sample the environment with uncertainties [7], which improves the previous method. Later we will try to modify our algorithm with a similar idea.

As mentioned in [8], investigating consensus among distributed multi-agents plays an important role in many multi-agent systems. The idea of consensus is to have distributed agents agree on a common value quantity through local interactions. There is a widely studied type of consensus algorithm called the "gossip algorithm" which leads to consensus in stochastic settings. However, it needs pairwise communication in a connected network of agents, which does not meet our requirements. Reference [9] integrated topological constraints into a multi-robot allocation problem and solved it with decentralized matroid optimization. This method also utilized a max-consensus auction [10], which requires heavy communication between robots.

One way to tackle uncertain environments is to perform multi-agent persistent surveillance [5]. This work associated the environment with a parameter ascending from zero every time after being discovered. By solving an optimization problem minimizing the maximum age of all the cells in the area while considering vehicle dynamics, the author claimed that in experiments the agents tend to search and cover in different locations. The inspiring part of this work is that it introduced a more realistic dynamics model of the vehicle where turning is not a sharp but gradual, and this affects the path planning process. But this work assumes other agents' positions are correctly known, and it suggests as future work that limited location
exchange should be considered and investigated. We will study this specification in our approach later in this thesis.

In [2], multiple robots are cooperating to sweep a field and prevent its uncertainties from growing unbounded anywhere. In their setup, AUVs moving on pre-defined trajectories can adjust their speed according to the environment to spend more time where uncertainty is high. A centralized LP is formed to solve this problem without knowing locations of other agents. However, the pre-assumed constant growing speed of the uncertainties makes the environment actually known by the planner, and the LP is solved in a centralized fashion. Accordingly it is hard to apply to robots whose sensing, computing and decision making are all limited on their own.

To summarize, if we assume expensive and unreliable communication in terms of latency and energy consumption as in underwater missions, we should avoid using any communication or centralized planner while doing multi-agent cooperative control. Instead, we need to design and apply distributed sensing, computation, and control.

In this thesis, we view the optimal sensor placement problem as a multi-player game, in which each agent is acting as self-interested player optimizing its utility. Then the problem will be reduced to designing agent utilities and learning mechanisms. Considering energy consumption and time delay during communication, we prefer to design locally computable agent utilities. Modifying the approach in [11], we can design agent utilities differently based on the needs of the applications as long as they are aligned with global utility. For learning mechanisms, we favor an algorithm that reduces unnecessary information exchange among agents but still converges to at least near optimal.

The rest of this thesis is organized as follows: in Chapter 2 we will introduce some background on game-theoretical methods and related results. In Chapter 3 we will formulate a sensor coverage problem and propose a game-theoretical approach that meets the specifications of different scenarios in our applications. Simulation and
hardware implementation will be described in Chapter 4. Other works done during the course of this thesis related to robotics are included Chapter 5. We will conclude and discuss the results and suggest future works in Chapter 6. Finally, detailed implementation notes are attached as Appendix A.
Chapter 2

Game-theoretical Approach Background

As our objective is to control multiple agents distributedly, i.e., each agent is making its own decisions only with local information. Game theory is well suited for this purpose. In game theory, let each agent be a player of the game, and its outcomes are affected by their own decision and decisions of others. We will then discuss essential elements of game theory for this thesis. From now on, we will use the terms robot, agent, vehicle, mobile sensor and player equally as they play the same role in our following discussion.

2.1 Components for game-theoretical formulation

To form a game, we need to define players, actions, utility and learning mechanism.

2.1.1 Players

Players are individual decision makers. Each player can choose its action based on some decision protocol. The set of players is denoted as

\[ \mathcal{P} = \{P_1, P_2, ..., P_{|\mathcal{P}|}\} \]

2.1.2 Actions

For each player \( P_i \), the set of actions it can choose is denoted by \( A_i \). The exact action chosen by \( P_i \) in a one-stage game is \( a_i \in A_i \). All the actions chosen by a group of
players \( \mathcal{P} \) is denoted as \((a_1, \ldots, a_P)\), which is called an action profile. Each action profile belongs to a family of action profiles, i.e., \( a \in \mathcal{A} \), where \( \mathcal{A} := \mathcal{A}_1 \times \cdots \mathcal{A}_{|\mathcal{P}|} \)

### 2.1.3 Utility

As we stated before, players are making decisions based on preferences over joint actions of all players. One possible way of expressing preferences is through a utility function. The utility function of player \( P_i \) is:

\[
U_i(a) = U_i(a_i, a_{-i}) : \mathcal{A} \rightarrow \mathbb{R}
\]

where \( a_{-i} \) denotes the actions chosen by other players. Larger utility represents a better outcome obtained by this joint action. Also, each action profile \( a \in \mathcal{A} \) corresponds to a global utility function defined as:

\[
U(a) : \mathcal{A} \rightarrow \mathbb{R}
\]

which can be seen as the objective of the player group or the central planner.

### 2.1.4 Learning mechanism

Within the setting presented above, the players are facing a multiplayer game, where they sequentially negotiate their actions to form an action profile satisfying all of them. Learning mechanism here means each player chooses its next action based on its utility, which depends on the actions of all the players in the current or previous rounds.

### 2.2 Essential concepts from game theory

Here we review some concepts which will be used later from game theory.
2.2.1 Nash Equilibrium

An action profile $a^*$ is called a pure Nash equilibrium if, for all players $P_i \in P$,

$$U_i(a_i^*, a_{-i}^*) = \max_{a_i \in A_i} U_i(a_i, a_{-i}^*)$$

If a Nash equilibrium is reached among a group of players, that means each of them is choosing its best action with respect to the actions of others. Note this does not necessarily mean a global optimal is reached in terms of the global utility $U$. Rather, it just means everyone’s choice is optimal given the choices of others. A famous example is the prisoner’s dilemma, where each player is acting optimally and reaches the Nash equilibrium is not the optimal global action for either player.

Also we need to point out that a pure Nash equilibrium might not always exist, but for our setting, there will be at least one pure Nash equilibrium.

2.2.2 Efficiency of equilibrium

Notice that there might exist more than one Nash equilibrium in a game. We introduce the concept of efficiency to distinguish them. An efficient action profile gives the highest global utility that can be reached for all the players.

2.2.3 Potential games

Here we introduce potential games because they have good convergence properties and suit our problem setup. Player action sets $\{A_i\}_{i=1}^n$, together with player objective functions $\{U_i : A \to \mathbb{R}\}_{i=1}^n$, constitute a potential game if, for some potential function $\phi: A \to \mathbb{R}$

$$U_i(a_i'', a_{-i}) - U_i(a_i', a_{-i}) = \phi(a_i'', a_{-i}) - \phi(a_i', a_{-i})$$

for every player $P_i \in P$, for every $a_i', a_i'' \in A_i$, and for every $a_{-i} \in A_j$ where $j \neq i$.

To form a potential game, individual utilities of agents should align with the
potential function, which means that if any player changes its action, his utility and
the potential function will have the same amount of change. So if we could formulate
the problem as a potential game, then we can take advantage of game theory literature
on learning in this group of games to establish algorithms with guaranteed optimal
convergence [12].

2.3 Game-theoretical formulation

Now we introduce a way to form multi-agent distributed control problems into a game
theory framework [11, 12, 13].

If we can assign action set $A_i$ and utility function $U_i : A \rightarrow \mathbb{R}$ with a proper learn-
ing mechanism, then the multi-agent distributed control problem can be interpreted
as a game. The problem is reduced to properly designing following components and
letting them fit some category of games which has desired properties:

- Utility function that each agent can obtain locally.
- Learning mechanism by which players can negotiate a final action profile that
yields optimal global utility.

The objective of our design is to reach the optimal global utility while each agent
is only interested in optimizing its own utility. A critical factor for utility design
is that it should somehow be aligned with global utility which is the main interest
of the central planner. If we design the utility in this way, it can be shown that
the game designed falls into the class of games named ordinal potential games. For
this category of games, certain learning algorithms could lead to an optimizer of the
potential function with arbitrarily high probability. A straightforward approach is to
use global utility as each agent’s utility. As we still want to make it locally accessible
for each agent, using global utility for player utility directly is impractical.

Next, we will state the design of the game to guarantee optimal convergence.
2.3.1 Utility design

All the utility functions used in this study are defined here. They are arranged in this order because some utilities are derived from others.

2.3.1.1 Target utility

We introduce target utility to define global utility in a clear way. The utility for a target is defined based on the action profile \( a \in \mathcal{A} \). In our sensor coverage setting, we can set each target’s utility to be the value of that target on the heatmap if it is within the sensory range of at least one agent.

2.3.1.2 Global utility

With the definition of target utility, we can define global utility to be the sum of all the target utilities denoted as \( U_{\tau_j} \), i.e.,

\[
U(a) = \sum_{\tau_j \in \tau} U_{\tau_j}(a)
\]

where \( \tau \) is the set of targets.

2.3.1.3 Agent utility

As stated before, agent utility should be aligned with global utility to make sure the agents agree on an optimal action profile maximizing the global utility. Here we use marginal contribution utility as agent utility, which is an agent’s marginal contribution to global utility, i.e.,

\[
U_{\mathcal{P}_i}(a_i, a_{-i}) = U(a_i, a_{-i}) - U(\bar{a}_i, a_{-i}), \quad \text{where } \forall \mathcal{P}_i \in \mathcal{P}
\]
Here $\tilde{a}_i$ means null action, by which agent $P_i$ takes no action as if it turns off its sensing ability. By the assumptions that follow, this utility can be obtained by agents with local information.

Moreover, it turns out that if we assign agent utility in this manner, we are forming a potential game with the global utility $U(a)$ serving as a potential function \[1\].

**2.3.2 Learning mechanism**

Learning mechanism enables players to learn to play desired joint action profile by repeatedly playing the game. We describe this process as follows: at each time step, each player selects its action $a_i(t) \in A_i$ with a probability computed by some mechanism. Then it obtains its utility $U_i(a_i(t), a_{-i}(t))$ and modifies the probability distribution for choosing its action for the next round. We are to create rules to adjust those probabilities for each player.

Here we will apply binary log-linear learning \[13\], which chooses action from a constrained action set. It operates as follows: at each time step, select one player $P_i \in P$ to update its action and other players keep their actions as the last round. Selected player $P_i$ chooses one action $\hat{a}_i$ from its available action set uniformly. Then the player chooses its next action by following probability distribution:

\[
\Pr_{a_i}^a(t) = \begin{cases} 
\frac{\exp\left(\frac{1}{T} U_i(\hat{a}_i(t-1))\right)}{\exp\left(\frac{1}{T} U_i(\hat{a}_i(t-1))\right) + \exp\left(\frac{1}{T} U_i(\tilde{a}_i, a_{-i}(t-1))\right)} & a_i(t) = a_i(t-1) \\
\frac{\exp\left(\frac{1}{T} U_i(\hat{a}_i, a_{-i}(t-1))\right)}{\exp\left(\frac{1}{T} U_i(\hat{a}_i, a_{-i}(t-1))\right) + \exp\left(\frac{1}{T} U_i(\tilde{a}_i, a_{-i}(t-1))\right)} & a_i(t) = \hat{a}_i \\
0 & \forall a_i \neq \hat{a}_i, a_i(t-1)
\end{cases}
\]

where $T$ is some factor specified by us which reflects how we differ the utilities of different situations.

From \[13\] and \[13\], we know that under binary log-linear learning, the global utility can be driven as close to the optimal as desired.
Chapter 3

Game-theoretical Approach for Distributed Coverage Problem

In this chapter, we will form an underwater sensor coverage problem and provide an algorithm based on the method introduced in Chapter 2. Then modifications of the method will be made to fit this approach to different application scenarios.

3.1 Problem setup

We will model our underwater coverage problem as dynamic sensor coverage problem as in [15], whose goal is to place some sensors in space to maximize resource coverage in given space. The whole setup can be seen in Fig 3.1.

We use discrete grids to represent the space. Consider a group of vehicles moving in constrained field $\mathcal{F} \in \mathbb{Z}^d$, with their positions $\mathcal{P} = \{P_1, P_2, \ldots, P_{|\mathcal{P}|}\}$. Agents are marked as small submarines in Fig 3.1. Any location in $\mathcal{F}$ is described as $l = l(x, y)$, where $x$ and $y$ are the coordinates corresponding to that location. Resource or information is formed as a heat map $\mathcal{M} : \mathcal{F} \mapsto \mathbb{N}$ where the value of the heat map indicates amount of resource or information. Their locations are $\mathcal{R} = \{R_1, R_2, \ldots, R_{|\mathcal{R}|}\}$. Here resources are marked as pink dots in Fig 3.1 and their sizes indicates different values of the heat map. Define the distance between two locations $l_i$ and $l_j$ in $\mathcal{F}$ as $d_{l_i l_j}$.

There are two types of sensing range for agents. Sensing range of agent for resources is indicated as red circles with radius $r$ surrounding the submarines. Sensing range of other agents has radius $2r$. We assume this for two reasons. Firstly it is needed
for the following algorithm to work. Also, it is reasonable as agents can have some protocol to sense other agents in a broader range. We assume that each robot knows its own location $p_i$ perfectly for simplicity. Each agent is capable to move in the field $\mathcal{F}$ and sense the environment within its range as well as nearby agents. For each agent, its actions are constrained set including going one grid to four directions and staying at the original place. Dynamics of agent $\mathcal{V}_i$ at time step $t$ (assuming discrete time) is

$$p_i(t) = p_i(t - 1) + u_i(t - 1)$$

where $u_i$ is the command of taking an action to move within its reach in one time step.

Figure 3.1: A model of sensor coverage problem
3.2 Utility and learning mechanism design

We will then design utility and learning mechanism based on the concepts in Chapter 2. Target utility is defined as

\[ U_{Rj} = \begin{cases} M(R_j) & \text{if any } d_{P_iR_j} \leq r, \forall i \\ 0 & \text{otherwise} \end{cases} \]

where \( r \) is the sensing radius of the agents. Then global utility can be expressed as

\[ U(P) = \sum_j U_{Rj}(P) \]

Our objective is to maximize this value, i.e.,

\[ \max_P U(P) \]

Then we can define agent utility as the marginal contribution to global utility, i.e.,

\[ U_{P_i}(a_i, a_{-i}) = U(a_i, a_{-i}) - U(\bar{a_i}, a_{-i}), \text{ where } \forall P_i \in P \]

For learning mechanism, each time one agent is randomly chosen, then it will uniformly choose an action within its constrained action profile and compute the probability as the last chapter. Its next movement will be obtained from this distribution.

To summarize the proposed approach, we conclude the above approach as algorithm 1.

Note that as we set sensing range of other agents twice as the sensing range of the resources, this marginal utility of each agent can be obtained using only the local sensing information of the agent.
Algorithm 1 Distributed Mobile Sensor Coverage

Require: Field $F$, Heat map $M$, Agents $P$, Actions $A$, Resources $R$

1: for $t = 1$ to MaxTime do
2: randomly choose an agent $P_i$ and one action $a_i$ in its action profile $A_i$
3: construct $U_{P_i}$ using marginal contribution
4: compute $Pr_{P_i}^a(t)$
5: let $P_i$ move to the position chosen from above multinomial distribution
6: end for

3.3 Applying the method to underwater sensor coverage

The goal of this resource exploring setup is that overall resources under coverage by all the sensors equipped with agents should be maximized. That means the agent should not only find and cover the richest resource but also avoid covering the same resource with the others, because according to the definition of agent utility, covering the same resource will not increase the agent’s utility.

One standard assumption for game theoretic setups is that the learning environment is stationary. As one of the primary assumption for marine applications, we do not assume the resource heat map we are covering is static. If the resource location has uncertainties or is drifting, the agents should be able to tackle them with a still persistent coverage. We will then apply the algorithm mentioned above directly to this scenario and claim it suits our purpose with both theoretical results and experiments. It is shown in [13] that if the environment varies ”slow”, such performance retains.

We will then apply this algorithm to a slowly drifting version of sensor coverage problem. In below Matlab simulation, nine mobile robots ran above algorithm in a $20 \times 20$ grid. Resources with different location and density are marked as colored areas on the map.
Figure 3.2: Multiple drifting resources. Agent locations at $t = 0$ and $t = 8000$

The left picture in Fig. 3.2 shows the initial position of both resources and portable sensors. Blue arrows indicate drifting direction of the resources. This drifting is slow concerning the dynamics of the agents, i.e., resources drift every 1000 steps of agent action. From the right picture in Fig. 3.2 we can see after enough long period mobile sensors reached their optimal location distribution to cover the resources and this coverage is robust to drifting resource locations. The agents need to be robust against the possible changes of the environment parameters such as direction and speed of sea current and adjust its exploring algorithm.

### 3.4 Modifications of the algorithm

In this section, we will modify this original algorithm to adapt it to specific requirements for underwater applications and use simulation experiment to testify their effectiveness. As the method used is from game theory, we can utilize the flexibility of utility design to adapt the algorithm to different scenarios.
3.4.1 Adapt to effect of sea current

One important factor affecting underwater robots is sea current. As known, the tide is changing over the day and sea current condition underwater might have rapid change over time as well, which will both affect the dynamics and energy consumption of the robots.

Here we assume agents are equipped with sensors that can sense the direction of the sea current and try to adapt their actions to the current to save energy while exploring. This idea is applied by adjusting the utility function as follows.

$$U_P(a_i, a_{-i}) =$$

- 1.4 × (U(a_i, a_{-i}) − U(\tilde{a}_i, a_{-i})) if $a_i$ is left
- 0.6 × (U(a_i, a_{-i}) − U(\tilde{a}_i, a_{-i})) if $a_i$ is right
- $U(a_i, a_{-i}) − U(\tilde{a}_i, a_{-i})$ otherwise

Here we punish the action against the sea current by multiplying a factor smaller than one to the original marginal contribution utility and reward the action following the sea current similarly. With this modification, agents will search and cover the resource in an energy-saving way. The simulation in Matlab is shown in Fig. 3.3. Here the white dot indicates the start position of the agent while white lines indicate the trajectories of the agent. We can see although both agents at last found the high resource area and stayed there, the right one consumes less energy by following the sea current as possible.
3.4.2 Adaptive to occasional communication between agents

Based on a practical scenario where AUVs need to float to the surface to either recalibrate their GPS locations or change their batteries, this period can be used to exchange some information between the agents to accelerate the exploration process as this information exchange happens in the air so we can use WiFi or radio to do it cheaply. As [S] pointed out, group behavior can be controlled better if we can exchange some information between agents.

We design the occasional information exchange event as follows:

- Set one round out of 100 time steps for information exchange ("occasional").
- Assume each agent is up to the surface during this round and broadcast their locations and utilities obtained to all the other agents.
- All the agent augment their action profile with an additional action which is "go directly to the agent with the highest utility", and the utility associated there will be discounted based on the distance between them, i.e., possible actions of
each agent now become

\{up, down, left, right, stay, go to agent \v_{\text{HighestDiscountedUtility}}\}

- Then do usual game theoretical learning.

![Graph showing comparison of global utilities over time between with/without occasional information exchange.]

Figure 3.4: Comparison of global utilities over time between with/without occasional information exchange

From Fig 3.4 we can see the agents converge to optimal coverage locations faster than the original algorithm.

### 3.4.3 Utilize prior knowledge of environment

Sometimes we have some knowledge about the resource before starting exploration as mentioned in [2]. Here we assume a simple case in which we know resources are distributed in a convex area. This prior knowledge can be interpreted as once an agent enters the resource area, then that is the target it is looking for. As the best approach to search somewhere better when one is in a convex area is moving inside it, we adjust
the utility of the agents by changing the weighting of surrounded area after reaching anywhere within this convex resource zone, such that the following movement of this agent will remain in this convex area, i.e., for the situation in Fig. 3.5 adjust the utility of the blue agent to be

\[
\left\{ \frac{U_N}{100}, \frac{U_W}{100}, U_S, U_E, U_{Stay} \right\}
\]

at this time to prevent it from leaving the convex resource area. After that, once the agent reaches the edge, its utility will be adjusted accordingly to keep the agent inside.

![Figure 3.5: Search in convex area](image)

From Fig. 3.6 we can see agents converge faster when prior information is utilized.

![Figure 3.6: Search in convex area: result](image)

Above we made modifications of the original algorithm to satisfy different specifications for underwater applications utilizing the flexibility of utility design.
Chapter 4

Simulation and Hardware Implementation

4.1 Introduction

As this study is motivated by real application scenarios, our final goal is an executable system for implementation of ocean exploration. For this purpose, we will first create simulation environment using ROS and Gazebo to do software-in-the-loop tests. Then submarine robots are assembled and deployed with algorithm to do hardware test. Designing these systems is also one of the main contributions in this thesis.

Note as we don’t currently have sensors to sense the resources nor the proximation sensor to detect other agents, we will instead rely on localization systems to know all the locations of agents and resources both in the simulator and real implementation systems. We will then decide to release the positions of the resources and other agents only to those agents who are within the sensing radius set by us as if related sensors are equipped to know this information.

Please note that this part is just a brief introduction to the experiment testbed. Detailed setup and manuals are in Appendix A.

4.2 Simulation Testbed

4.2.1 Software packages used

Here we introduce software components used while creating this simulation environment.
• **ROS**: Short for Robot Operating System, which is a communication protocol lives between Linux system and software to exchange information between different parts of robot software and hardware. Utilizing standardized interfaces, ROS is commonly used in robotic system design and becomes a standard.

• **Gazebo**: A simulator directly communicates with ROS that can simulate robot sensor, actuator and outside environment. It also provides perfect localization as ground truth. It is commonly used for rapid tests of algorithms and pre-tests for hardware implementation.

• **UUV simulator package** [16]: A Gazebo/ROS package for underwater robot simulation. It contains AUV models that can be used directly in Gazebo and low-level controllers to send them waypoint commands and drive them to desired positions in Gazebo. It can also simulate water dynamics such as direction and speed of sea current.

• **MultiROV package**: ROS package created by me to execute distributed game-theoretical algorithm on each agent and send them corresponding commands. This package can be accessed at [https://github.com/luym11/multirov/](https://github.com/luym11/multirov/)

AUV model in simulation adds more realistic features to the experiment. For example it can’t operate at arbitrarily high speed, can’t take sharp turn and its dynamic model can be affected by environmental current flow.

### 4.2.2 Simulation setup

• **Area**: Considering the real testbed we have for this stage is a swimming pool, area size is set to be $10 \times 10$ grids with 100 sectors.

• **Agents**: We use AUV models from UUV package as agents. As we have three hardware kits in the lab, we use three agents in the simulation. Different from
previous Matlab simulation setup, now agents can go to any sector surrounding it in one time step, i.e., action set is now

\{up, down, left, right, up left, up right, down left, down right, stay\}

- **Resources**: Here we consider a static resource with size $3 \times 3$ and a drifting resource with size $5 \times 5$ whose location can be controlled by a human operator through command lines or joystick.

Notice here we use hard constraints to prevent agents from hitting the edge of the test area and bumping into each other. Once the algorithm gives an output direction-to-go for the agent, the agent will first test if this will lead to conflict either with the boundary or other vehicles. If it’s true, then the agent will stay at the original place at this round. This prevents possible collisions from happening. Also, it is claimed in the algorithm that the way it’s using to set the utility will automatically take care of collision avoidance, but we still made this modification regarding safety. Fig 4.1 is a screenshot of the constructed simulation environment. Notice the while box on the right bottom corner indicates one of the resources.
4.2.3 Simulation software architecture

The simulation system architecture is as shown in Fig. 4.1. It’s designed as a layered structure. Black arrows indicate information exchange between different parts. From bottom to top, it can be divided into:

- **Distributed game-theoretical algorithm class**: takes local information as input and send out corresponding commands generated by the algorithm for the agent to execute in next round.

- **Distributed AUV ROS node**: a node written to bridge the C++ algorithm with simulator node using communication protocols in ROS.

- **AUV simulator node**: contains the AUV model from UUV package, takes input command and execute as waypoint following in Gazebo simulator.
• **Gazebo environment**: the simulator of a marine test field and robot dynamics with localization of robot and resource positions.

With this simulation setup, we can execute our algorithm in a simulated marine environment.

![Figure 4.2: Simulation software architecture](image)
4.2.4 Experiment results

Here two sets of experiments are executed to examine the correctness and performance of the game-theoretical algorithm. Note although all the algorithms are running on the same workstation, each AUV node runs separately and do not talk to each other.

Simulation 1: Multiple drifting resources

For this experiment, we set two virtual resources in the simulation environment. One is $3 \times 3$ static, and the other is $5 \times 5$ drifting. Drifting resource is moving randomly within test range($10 \times 10$) with $\frac{1}{4}$ of AUVs’ speed. Three AUVs are used in this experiment. From Figure 4.3, both static and drifting resources are well covered consistently by three agents after enough long time. Also, we can see from the past visited positions of AUVs that the whole area is well explored.

![Figure 4.3: Multiple drifting resource simulation result](image)
Simulation 2: Sea current adaptation

In this experiment, we activated the sea current in Gazebo environment and let the AUVs be aware of it and change their utilities as described in the algorithm. Their trajectories can be seen from Fig. 4.4. From Fig. 4.4, firstly we can conclude the AUVs are trying to explore the environment following the sea current to save energy from vehicle 1 and 3. Also, notice that vehicle 2 has found a resource and covered it with the other agents, we can see the agents aren’t blindly following the sea current but still executing search mission.

![Adapt to sea current simulation result](image)

Figure 4.4: Adapt to sea current simulation result

4.3 Hardware Testbed

4.3.1 Introduction

After simulation test, we built hardware testbed to testify our algorithm above in real underwater environment and prepare for other testing purposes in the future. The algorithm parts are the same as in simulation, except the simulator parts are replaced with real underwater vehicles. Here newly added components are listed below:
• **Pixhawk**: Pixhawk is an autopilot module that can install open source firmware for different vehicles including drones, helicopters, robot cars and so on. It takes care of low-level controls such as stabilization, depth hold as well as controllers for actuators on the robot. In our case, it controls motors on an underwater vehicle.

• **ArduSub**: Autopilot firmware belongs to AuduPilot family specifically for underwater vehicles. It builds a bridge between the vehicle and ground control station software to enable controls from outside and mission planning. It’s also compatible with ROS using MavROS package to communicate with different components within ROS architecture.

### 4.3.2 Hardware setup

• **Area**: Considering the real testbed we have for this stage is a swimming pool, area size is set to be 10 × 10 grids, with size of 0.6m × 0.6m for each grid.

• **Agents**: We use underwater robot kit BlueROV as agents, and action set is now \{up, down, left, right, stay\}.

• **Resources**: Here we consider a resource with size 3 × 3 and controlled by a human operator through commands.

### 4.3.3 Localization methods

As stated before we don’t have any local sensor except water pressure sensor for BlueROV to hold its depth, we will use the same idea as in simulation, i.e., a central computer knows all the locations of both vehicles and resources, and releases related information to vehicles if they are within the pre-set sensing radius.

But here the experiments will be done in reality, which means we don’t have perfect localization from Gazebo. So it’s essential to build a localization system to represent
the positions of vehicles and resources. As mentioned, previous easy localization methods don’t suit our testing environment, e.g., GPS can’t be used underwater, motion capture with markers might not work properly underwater especially when water is moving. So other methods should be considered. Two different methods are implemented and tested. And they are proper for different testing conditions.

4.3.3.1 Ultra-short baseline (USBL)

USBL is an underwater acoustic positioning system. It includes a transceiver mounted on a platform near the surface, and transponders mounted on underwater vehicles. Then a computer is used to calculate the positions of each vehicle based on the signals received by the transceiver using “phase-differencing” method.

This method has a wide range, stable rates and 0.1m level accuracy so that it’s suitable to deploy in real ocean applications. But as it’s a method based on acoustics, one apparent shortage is that it can be significantly affected by reflection especially in constrained areas. The test result in the swimming pool showed that when the vehicle is near the walls of the pool, readings of the USBL is greatly fluctuated. It can be a suitable method for testing in the ocean, but for our first stage pool test, we need to find another localization method.

4.3.3.2 Vision-based localization

A vision-based method is chosen to localize AUVs for testing in the pool. As shown in Fig. 4.5, a quadcopter will monitor the whole testing range at a certain height with a camera, and each AUV will have an Apriltag set on top of it which can be seen by a downward faced camera. There are existing algorithms that can calculate relative positions of each marker with respect to the camera. If the camera is tuned properly and the area is carefully chosen, this can be used as a way to localize underwater vehicles.
This method also has its deficiencies, as the camera has resolution constraints, the localizable area can be covered is constrained as well. Testing results in seawater indicate that this method is not proper for unstable water as the markers will get tilted due to waves. But this is good enough for testing in swimming pool for this stage. To obtain a remapped coordinates as in simulation system, we set two markers to identify origin and x-axis. By differentiating relative positions with these two markers, coordinates are remapped as simulation.

This system reaches $0.1m - 0.2m$ accuracy and about 2Hz based on a comparison between the ground truth. As we set the grid size as $0.6m \times 0.6m$, this accuracy is enough for our localization purpose.

![Figure 4.5: Vision-based localization system](image)

### 4.3.4 Hardware system architecture

The overall system structure is shown in Figure 4.6. Algorithm node and ROS communication node (lowest and the second lowest) remain the same as in simulation. Above the communication node, AUV simulator node is now replaced with AUV controller node which converts high-level direction-to-go command to actual motor speeds and
sends to autopilot node through MavROS to drive the motors. Then the markers on top of the AUVs will be read by the camera on DJI Matrice 100, a ROS node running there will report remapped coordinates to AUV controller node, thus closing the control loop. Above described system is shown in Fig. 4.7.

Figure 4.6: Hardware system architecture
4.3.5 Hardware open-loop test result

Firstly we tested the overall system with open-loop control, i.e., the AUV is driven by a human operator through outside controller to examine dynamical properties of the system, including controller response and localization speed and accuracy. From Fig. 4.8 positions of the AUV is read and recorded with required accuracy. Video of this experiment can be accessed from https://youtu.be/dtm3MSz8M68.
Figure 4.8: AUV trajectory in open-loop test
Chapter 5

Other works related to robotics

5.1 Introduction

In this chapter, we will briefly talk about other works of related to robotics this thesis. They include contributions to MBZIRC robotics challenge 2 and cooperation with other lab members in distributed linear programming for multi-agent cooperation in an adversarial environment.

5.2 MBZIRC challenge 2

5.2.1 Feature based tool detection from video stream

As part of the challenge 2 task, we need to use a camera to detect multiple tools and keep tracking them in different frames. My task here was to write this algorithm using OpenCV library and integrate this to ROS as a part of the software running on the Husky ground robot.

The idea is as follows:

- Extract features of the initial frame (here we used SURF feature)
- Create bounding boxes by feature point clusters (as shown in Fig 5.1)
- Import next frame and keep tracking feature points
- Create bounding boxes by these feature points to track tools in different frames
There is one import issue of this method that it will not work if too many feature points are lost in one frame, but this situation didn’t appear too many times in indoor test. But this method is not used because other members developed a detector using neural networks and outperformed feature-based algorithm while also providing tool sizing function.

Code for this detector can be accessed at https://github.com/luym11/MulticascadeDetector

### 5.2.2 Laser width detector

Another issue during the challenge is to identify the size of the value the robot need to operate in order to determine the size of the tool to grab.

One possible method to measure the size of the valve is to use a laser sensor facing down and pass through the valve. Then recorded time of passing the valve can be used to compute the width of it if the moving speed is known.

From the design of this width detector, a SICK sensor will output current in some range indicating the distance between the laser point and the emitter. We will
then convert this current to a voltage and then send the voltage to an A/D converter. Converted voltage value will then sent to Arduino to filter and publish to a ROS topic for further process. The system is shown in Fig 5.2. This video shows the system in operation https://photos.app.goo.gl/eE3X5QfO70iGBXmH2.

Although the result was promising regarding identifying the valve size indoor, we still faced these issues:

- Valve’s reflective surface caused disturbance to distance readings.
- The laser we were using is level 2, which might not be strong enough under sunlight in Abu Dhabi.

Also at that time, we have no information about the initial orientation of the valve. As it might be tilted rather than horizontal, we need to develop a new algorithm to measure the orientation as well. But based on our indoor experiments, the accuracy of the laser beam is not acceptable for this algorithm. So this method was installed but prepared just as a backup. Finally, this sizing issue was solved using vision. Code of laser width detector can be accessed at https://github.com/luym11/ArduinoLaserReader
5.3 Hardware implementation for distributed linear programming

Here some work with other lab members on implementing an algorithm for indoor tests will be introduced. Main contribution of the author here was to develop this testing system, find a suitable size of the problem to execute in real-time, and record/representing some of the results in the paper. This setup consists of 4 25cm quadcopters that do computation onboard inside RISC flying arena. All the quadcopters are localized by motion capture system. PX4 installed Pixhawk autopilot is for low-level control and Odroid XU4 onboard computer with WiFi communication module is used to execute distributed algorithm in high level. All the quadcopters communicate with each other and the base station through a wireless router. Quadcopters used is shown in Fig. 5.3.
After testing for different scenarios, we finally chose $7 \times 7$, 3 VS 1 with prediction horizon 2, which executes at around 30Hz as our test setup. Indoor experiments showed the algorithm has good performance and high execution speed.

Figure 5.3: Quadcopters used for experiments
Chapter 6

Conclusion

6.1 Summary of this thesis

This thesis describes a developed algorithm, simulation and hardware test system, as well as all related software packages. We first introduce a game-theoretical algorithm for distributed multi-agent control and applied it to underwater coverage problem. Then modifications of this algorithm were made to adapt to different underwater application scenarios. For testing and future development purposes, a simulation and a hardware system for multiple vehicle underwater test system are built and tested.

6.2 Future works

This work can be extended in following directions:

- Based on testing results in constrained pool, do ocean test using USBL.

- Consider other real application constraints such as limited energy.

- More realistic assumptions on sensors (directional information of other agents might not be available).

- More reasonable model for information exchange.
REFERENCES


A.1 ROS/Gazebo simulation system manual

Main functions and operating manual of simulation system is in this section.

A.1.1 MultiROV

MultiROV contains game-theoretical algorithm, protocols to communicate with ROS environment, and other essential software components. It is created based on the principle of separating algorithm parts and ROS parts so that each part can be modified individually. Note that there are two branches in this repository. The master branch is for the simulator, and the BlueROV branch is for the hardware test. However, BlueROV branch is written more cleanly and abandoned many parts that are not necessary. So I recommend using BlueROV branch for both simulation and hardware. The user can refer to commit history of this repository for detailed development process as I have clear notes while creating this project for future reference.

It’s class diagram is shown in FigA.1L.
Figure A.1: Class diagram of algorithm in ROS

Note that only important members and methods of the classes are presented. Here are three main components:

- **explore_algo** class is the high-level algorithm class, deciding for an agent with local information of resources and other agents. It builds a heatmap with local resources and a coveragemap (not working as a class member but a local variable inside a method) with visible local agents. Last two steps of its computation are listed in its methods in the class diagram.

- **coveragemap** class builds the nearby coverage status for an agent with local information of other agents. Methods include
  
  - `set_coveragemap()`: based on nearby agents to compute coverage status

- **explore_algo_node** class has **explore_algo** as its member to high level algorithmic computations. Other subscribers are used to subscribe resource locations, agent locations and current direction from Gazebo topics. Publisher is used to publish computed command to a controller node to send incremental
control service for simulation and controller node for hardware case. Important methods include:

- `heatmap_update()`: function needed for `resource_location_Callback()`, update local heatmap of member `ex` with locations of sensed resources. Note static resource hardcoded at (3,4) here but still keeps secret for the agent when it’s out of the sensory range.

- `resource_location_Callback()`: triggered once resource location is received from Gazebo/other outside nodes. Update heatmap in `ex`.

- `agent_location_Callback()`: triggered once agent location is received from Gazebo/other outside nodes. Will execute algorithm and send command the controller, either in simulation or in hardware.

- `current_angle_Callback()`: triggered once current is changed and related topic is publishing. Will modify a parameter in `ex` to change the utility computation.

One thing to notice is the way we express commanded directions in this setting. We use

\[
\begin{align*}
2 & 5 & 8 \\
1 & 4 & 7 \\
0 & 3 & 6
\end{align*}
\]

i.e., 4 means stay, 7 means go one step in the x-direction, etc.

### A.1.2 UUV simulator

UUV simulator as mentioned before is an open source software package to simulate underwater robots and its working environments. As it’s a big complicated package, we will only discuss some issues when using its related parts. Notice this package is modified by me so please use the version at [https://github.com/luym11/uuv_simulator](https://github.com/luym11/uuv_simulator) as
it’s the version in RISC marine workstation. Also, there are some pre-requirement software packages for installing this package, they are installed correctly on RISC marine workstation. There will be some instructions about this in the last part of this appendix.

We will structure this part with ROS nodes need to run and related explanation. Then we will draw a ROS node graph as Fig A.2 to show the relation between these components.

Figure A.2: ROS/Gazebo nodes

- **roslaunch multirov lake.launch**: This launch file loads Gazebo world, it’s appearance and simulated time. After loading this file, Gazebo environment will be open. Note that x and y axes are already set in Gazebo, we also use
walls to indicate that as \[ A_3 \].

Figure A.3: x and y directions in Gazebo

- **roslaunch multiagent_simulation multiagent.launch**: this file loads AUV models, resource models and corresponding controllers. RViz can also be loaded from here. In details, we launched (in terms of namespace)
  
  - **rexrov1, 2,3**: spawn robot model in Gazebo; publish its states to Gazebo; publish its position to ROS topic (using `agent_listener` node running with it); simulated dp controller.

  - **rexrov0**: spawn resource model in Gazebo (originally at (5,5)); publish its states to Gazebo; joystick node to control it (notice `agent_listener` is not here because the code for resource was developed earlier and used another method for publishing the location).

  - **rexrov00**: spawn resource model in Gazebo (static at (3,4)); publish its states to Gazebo.

- **rosrund multirov resource_publisher_hd**: let the movable resource con-
trolled by publishing to the topic `resource_location_from_keyboard`. The way to change resource location is `rostopic pub /resource_location_from_keyboard geometry_msgs/Point "x: 4.0 y: 4.0 z: -30.0" -r 1` and different from the joystick that can also change the location of the movable resource, this change with a keyboard is instant.

- **rosrun multirov resource_listener_hd_node**: subscribe from above published topic and republish to the topic `resource_location`. Our previous method was a more complicated way of implementing `agent_listener` node by subscribing `rexrov0/base_stabilzed` and republish to our own topic `resource_location`. Now we move to this method for compatibility because in hardware phase we can’t get positions from Gazebo neither the existence of related topics. For vehicles they can be localized by our method, for virtual targets, this is the best way to write this so that it can be used both in simulation and hardware. More details can be found in commit comments in BlueROV branch.

- **roslaunch multirov explore_environment.launch** for three vehicles: Executes previous mentioned node `explore_algo_node_main` and a incremental controller which calls the service `ns/go_to_incremental`.

Also, this package supports useful topics and services, for example:

- **Add current**: `rosservice call /hydrodynamics/set_current_velocity "velocity: 1.0 horizontal_angle: 1.7 vertical_angle: 0.0"` and this will be published to related topics as if the ROVs have sensor to sense it.

- **go_to service**: command the vehicle to a specific position in Gazebo.
A.2 Hardware system manual

In this section, we will discuss the hardware implementation phase of this project. As this system consists of many parts, we will talk about them separately.

A.2.1 BlueROV

Kit Assembly and common issues

Please refer to their official website for assembly while noticing following points:

- It’s recommended to test each ESC and motor before sealing the enclosure. It will be very hard to change any of them if the ROV is fully assembled.

- Fathom-X Topside board always needs to be powered by Mini USB, or it will not work.

- Organize the tether wire cleanly and don’t let it twist when doing experiments, or much time will be wasted on untangling them.

- When opening the enclosure, remember to remove the penetrator first; when closing the enclosure, remember to close the penetrator after closing the cap. It’s for water proof sealing purpose.

- Use 7.0Ah, 14.8V batteries in the lab as they last much longer than the others.

- Do a vacuum test every time before submerging.

- Motor direction can be reconfigured through QGroundControl software and don’t need to change its wires on hardware.

Network setup and companion computer

Here we are using Fathom-X to extend the ethernet longer and communicate with the Raspberry Pi inside the BlueROV. BlueROV originally comes with a compan-
ion Raspberry Pi with a system image that only allows joystick control through QGroundControl ground station, which is not what we desire. So we reimaged the Raspberry Pi with an Ubuntu Mate system, then installed related software packages there, including ROS Kinetic and BlueROV ROS package (modified) from https://github.com/luym11/bluerov-ros-pkg.

We mainly use two parts of this package. For BlueROVs, we will launch bluerov
bluerov_r1.launch locally, which loads state publisher, MavROS that talks to ArduSub firmware, imu and camera equipped on the ROV. For controller from ROS via MavROS (both joystick and codes), we launch bluerov_apps teleop_f310.launch on ROS master machine because it needs a joystick for emergency operation, change of mode, arm/disarm, etc. This modified controller node can additionally take direction_to_go as input from ROS topic and control the ROV to go towards that direction with a pre-set speed by publishing to rc_override topic as the joystick does. Note this also means we can directly publish to this topic to control the ROV from the command line.

Note that for some version of ArduSub firmware, the ROV can not take commands from MavROS. For now only ROV1 associated with IP 192.168.0.111 has the correct version of firmware. This will be checked further.

For hardware basic testing, we have a water tank in RISC lab. To use it, please use the mountain climbing rope attached to both the ROV and the beam on top of the tank in case it sinks. Normally testing operation can be done by only one person as the ROV will automatically float on the surface when disarmed.

Instead of the network configuration used in their manual which can only control one ROV at a time, network interfaces of them are reconfigured and connected to RISC marine router with pre-assigned static IP addresses. Note that we will connect all the devices through this RISC marine router with static IP address. A detailed list will be included in the last part.
Raspberry Pi OS image (software packages configured) used here is stored in RISC Google Drive, after flashing, remember to change

- `.bashrc` for ROS_IP and ROS_MASTER
- `interfaces` in etc folder for IP address
- `bluerov1.launch` for ground station IP and target number which is used in accessing multiple ROVs from QGroundControl

### A.2.2 Localization system

As mentioned before, a localization system is essential for both knowing the positions of agents and resources. Also it’s needed for waypoint feedback control of the ROVs. We will introduce two methods we have so far.

**Tritech USBL**

For USBL method, we use Tritech USBL devices. Transponders will be installed on ROV as shown in Fig A.5 and powered from the battery there. Transceiver is powered by it base controlled by software on windows machine and data will be transferred to ROS master PC from serial port. Related ROS package is at [https://github.com/luym11/RISCusbl](https://github.com/luym11/RISCusbl). So the overall architecture is shown in Fig A.4.
When using this system, please use the specifically made serial port reader as Fig. A.6 for its voltage level.
Figure A.5: Serial adapter
Vision-based system

As the defects of USBL system mentioned before, we finally used a vision-based method for this stage of hardware test. Here we chose to use Apriltags to mark the ROVs and use a fisheye camera with related packages to give relative locations of each marker. Then we use a ROS node called `location_bridge` to publish these locations to `agent_locations[/]` topics as we did for Gazebo, thus close the control loop.

First, we need to choose a proper camera and calibrate it. After testing different kinds of camera, we finally chose the fisheye camera and calibrated it using a ROS
camera calibration package. This localization system is installed on a DJI matrice 100, with an on-board computer as shown in Fig A.7 and Fig A.8.

Figure A.7: DJI matrice 100 with localization system
Then package at https://github.com/luym11/apriltags2_ros is used to detect markers. Test indoor and outdoor showed its good performance as shown in Fig A.8.
We used Odroid with WiFi communication to RISC marine router to send detected locations to ROS master computer. Three software components are running on the odroid:

- The USB camera node to publish camera image camera
- `image_proc` package to do image rectification
- Detection code that gives relative location of each marker to the center of the camera

The odroid image is also stored in RISC Google Drive.

The software running on PC is a `location_bridge` node, remap these coordinates and publish them to `agent_locations/` topics instead of the Gazebo environment. With this architecture, we can create a closed control loop.

The overall system architecture is shown in Fig A.10.
All the commands need to run for one robot open-loop test with this set up are as follows, note the algorithm part is not included in the test now, but as we have the localization system, there is not too much work to close the loop as the architecture graph shows.

- **On ROS master machine**
  - roscore
  - roslaunch bluerov_apps teleop_f310.launch
  - rosrunc image/view image/view image:=/tag_detections_image: to monitor the view of the camera

- **On Odroid**
  - roslaunch apriltags2_ros rov.launch
• On BlueROV

  – `roslaunch bluerov bluerov_r1.launch`

So the network architecture of this system is Fig. A.11.

![Network structure](image)

Figure A.11: Network structure

### A.3 Others

#### A.3.1 Data recording and representation

It’s recommended to use rosbag and rqt_mplot to record and represent data, respectively.
A.3.2 list of software packages and OS images

Software packages

A list of all software packages used (with hyperlinks). They are all host on my account publically on Github. Will be forked to RISC account.

- MultiROV
- UUV simulator (modified)
- BlueROV packages (modified)
- Apriltags detection package
- USBL serial reader

OS images used

- Original OS image for BlueROV (just for archive purpose)
- Ubuntu 16 Mate with ROS, MavROS and BlueROV package: for Raspberry Pi
- Ubuntu 16 Mate with ROS and AprilTag package: for Odroid

A.3.3 Carrying list for outdoor test

As there will always be something forgotten, a list of carryings when going outdoor test is created and maintained.

- School bus key
- DJI Matrice 100, 2 batteries, RC, connection wire with the smartphone, attached Odroid (with WiFi stick and batteries) and camera, attached camera
- Odroid backup: with WiFi, power cable, a camera with USB cable
• Odroid console cable

• SD card reader

• Tapes

• Battery checker

• Ethernet cables

• ruler

• zip ties

• RISC marine router with battery and power cable

• Apriltag markers

• Linux PC (RISC marine laptop)

• ROVs with tether, Fathom-X power cable, ethernet cable, batteries

• Logitech joystick

A.3.4 Equipment list and backups

• Linux ROS Master risc@192.168.0.195, risc

• ROV1 risc@192.168.0.111, risc; gcs target 1

• ROV2 risc@192.168.0.112, risc; gcs target 2

• ROV3 risc@192.168.0.113, risc; gcs target 3

• ROV2 Test Pi with a ArduSub installed Pixhawk risc@192.168.0.112, risc; gcs target 2

• Camera Odroid odroid@192.168.0.190, odroid
• Camera Odroid backup odroid@192.168.0.180, odroid

A.3.5 UUV dependencies troubleshoot

Look at the log, reinstall essential packages, modify CMakeLists. Remember to source the bashrc everytime redo catkin build to make changes really effect.

Eigen 3 issues

Can’t find related CMakeLists

Change related CMakeLists as

-find_package(Eigen3 REQUIRED)
+find_package(PkgConfig)
+pkg_search_module(Eigen3 REQUIRED eigen3)

Can’t find eigen/core

• Make a new soft link to src

• modify include_directories(include catkin_INCLUDE_DIRS Eigen_INCLUDE_DIRS)

Other dependencies

teleop issue

Rebuild this package from source or use apt-get