Machine Intelligence

- Guide Lego Robot Towards Intelligence -

- SW10 report by d630a -

- Group d630a -
Christian Skalborg Dietz
Jonas Majlund Vejlin
Tharaniharan Somasegaran

Aalborg University

The Department of Computer Science
Aalborg University

Software Engineering, 10th semester

Title:

Guide Lego Robot Towards Intelligence

Project period:
February 1 2009- July 31, 2009

Theme:
Machine Intelligence

Group:
d630a

Authors:
Christian Skalborg Dietz
Jonas Majlund Vejlin
Tharaniharan Somasegaran

Supervisor:
Yifeng Zeng

Number printed: 6

Pages: 91

Finished: July 31, 2009

Abstract

This report deals with the implementation of a multi-agent system. The implementation is a realization of the theory about POMDP and I-POMDP. In the implementation agents are represented using Lego Mindstorm, Wii Remotes are used as a positioning system and graphical models are created using Hugin Expert. The system itself is programmed using conventional C++.
Focus has been placed upon implementing a chosen scenario using the theory, and the issues that arise when doing so. Examples of these are: Attaining a viable data structure that will be able to efficiently represent the exponentially increasing amount of data in a policy tree. The creation of a transition function, takes the uncertainty of actions into account. The choice, representation and implementation of an environment that would normally be too large to support higher time horizons.

Preface

This report documents a project made by group D630a at the Department of Computer Science at Aalborg University within the area of Machine Intelligence. The project period spans from Feburary 1st 2009 to July 31st 2009. The theme is multi-agent systems.

This report assumes that the reader has basic knowledge about probability theory, graphical modeling of POMDP and programming in C++.

References to literature are written in square brackets such as [DZC09], references to figures are written without brackets such as figure 3.7 and references to equations are written in parenthesis such as equation (3.3). Furthermore references to appendices are written as Appendix A. The appendices can be found just after the bibliography, which is located at the end of the report.

Project group d630a

Christian Skalborg Dietz Jonas Majlund Vejlin

Tharaniharan Somasegaran

Contents

Chapter 1

Resume

In the world of Artificial Intelligence (AI) multi-agent setting is a new branch. This report documents the process of implementing a multi-agent setting in a physical environment using two robots. The implementation of the multi-agents is built around the 'Follow the Leader' game. One robot is the 'Leader' and moves independently, whereas the 'Follower' tries to mimic the actions of the 'Leader'.

Theory

The Interactive Partially Observable Markup Decision Process (I-POMDP) is a framework for modeling multiple autonomous agents in the same environment. In contrast to similar framework, such as POMDP, each agent performs actions and observes the state, to update their own belief state. The framework is however a mathematical framework which cannot be implemented into code directly. By using graphical modeling, it is possible to create an Interactive Dynamic Influence Diagram (I-DID), which is an extension of the DID, over the multi-agent problem. The I-DID can be implemented in Hugin, and through Hugin's API, the model can be accessed and manipulated during runtime. The solution to the I-POMDP will be a policy tree which maximizes the accumulated reward.

The robots are built using the Lego Mindstorm Robotics kit, which is a toy consisting of multiple sensors and actuators. The actuators are electrical motors which can provide forward and backward propulsion. The sensors are Light, Compass and Distance, which can be combined to perform an observation in the environment. There are made experiments on the sensors and the actuators to estimate their accuracy, and the results are rather discouraging. All the sensors failed, but the actuators however perform rather well. We had to find another mean of performing observations, and turned towards the Wii Remote. We implemented our own tracking system using the Wii Remote and used a simple mathematical model to map the environment in the implementation. The tracking system can pin-point the position of each robot, in an area of $2.50 \times 1.10[m^2]$ with a precision of $0.50[cm]$. The positioning is to be considered as an observation for the robots, but due to the precision of the tracking system we had to implement a function to introduce error into observations.

Implementation

The environment is modeled as a 4×4 grid cell, where each cell can be considered a state and

where there is a unique observation in each cell. Using a regular tree structure to store the policy tree, the size quickly becomes a problem as the number of nodes increase exponentially for each time horizon. To accommodate for the exponential growth, we implemented a new data structure which address the problem and the result is a compact structure, which only grows linear in size as the time horizon increases. The idea is derived from the setup of tree structures; the total number of different nodes a parent can have, is equal to the number of actions defined in the framework. By only introducing the total number of nodes at each horizon, and drawing arcs matching the observations from the previous horizon, it is possible to encode the information of a tree structure in a more compact manner, review the comparison on Figure 1.1.

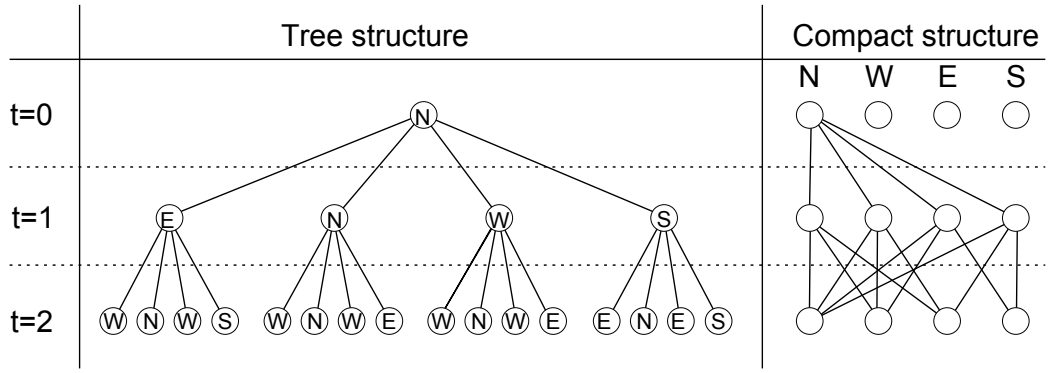

Figure 1.1: Comparison of a regular tree data structure and a compact data structure.

The I-POMDP framework includes models of other agents, and utilizes a CPT to derive which model each agent will act according to. The CPT grows exponentially as a result of the observations and actions of each agent. To address this problem, the environment is decomposed into four smaller sections, see Figure 1.2.

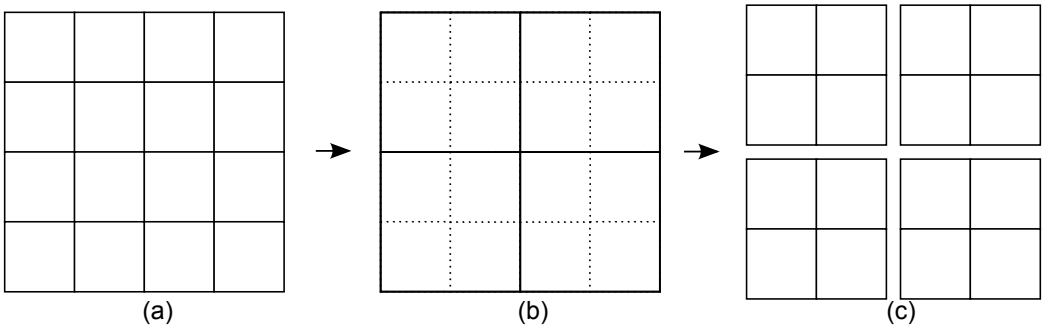

Figure 1.2: Decomposition of the environment into smaller sections.

The decomposition allows us to increase the number of time horizons of the policy tree, as well as scale the environment to a size normally not feasible. The 'Leader' is modeled using a DID, and the 'Follower' a I-DID which includes the model of the 'Leader'. There are four DID models for the 'Leader', one for each of the decomposed sections of the environment, but there are only two I-DIDs. There is a top-level I-DID and a low-level I-DID, the top-level considers the entire 4×4 environment as a giant 2×2 grid cell, see Figure 1.2 (b). The low-level I-DID considers each decomposed section as a 2×2 and is oblivious of the entire map. If the robots are in the same decomposed section, the low-level I-DID is used for the 'Follower' which will include the DID for the 'Leader', that corresponds to the decomposed section. If both robots are in different decomposed sections, the top-level I-DID is used. Both I-DID moves the 'Follower' close to the 'Leader'.

Results

We have achieved to successfully implement the 'Follow The Leader' game using I-POMDP modeled as an I-DID. There has been made experiments of the model and they verify the models and the implementation. Figure 1.3 is a screenshot of the GUI, which records the movements of both robots and illustrates the date in real-time for the user.

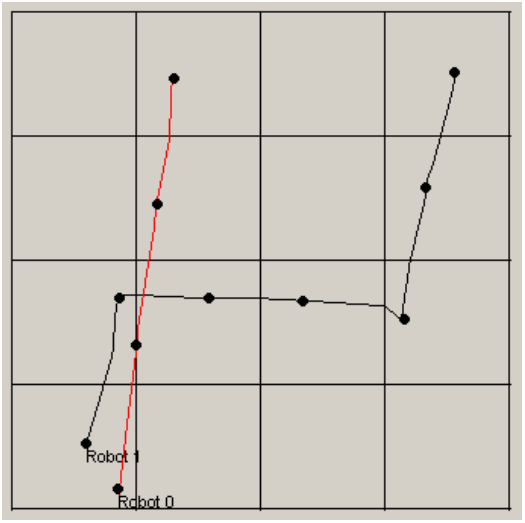

Figure 1.3: Full actual recorded path in the environment for 2 robots. The red path is the 'leader' which is using a DID model, and the black path is the 'follower' which is using an I-DID model.

The red line illustrates the movements of the 'Leader' and the black line the movements of the 'Follower'. Both robots move from the top of the environment to the lowest left cell which is the goal.

Chapter 2

Introduction

An Artificial Intelligence (AI) is commonly considered as a single entity, which controls one robot that performs different tasks in the environment. In the recent years scientists have expanded the definition of AI to include multi-agent systems. A multi-agent system has a number of autonomous agents interacting with each other, in order to solve a predefined task. This project is a study of how to implement the underlying theory into a practical implementation.

A multi-agent system has several applications, e.g. in computer games, where the individual AI-opponents can be configured to cooperate to perform complex strategic maneuvers, which ultimately will give the gamer a more realistic gaming experience. Our focus for multi-agent systems will be robotics, where robots are modeled to perform different tasks depending of each other. The *Interactive Partially Observable Markup Decision Process* (I-POMDP)[DZC09] is a mathematical framework for modeling multi-agent systems. Each agent is able to consider the preferences of other agents, and act accordingly.

The agents will be physical robots, but we do not have access to advanced equipment so they will be built using the Lego Mindstorm framework[Leg09]. Lego Mindstorm is a toy, and the robots will not be able to perform sophisticated or complex interactions with the environment, but we do have access to simple actions such as movement, distance readings, color readings and compass readings. We have decided to model a 'Follow the leader' game, where one robot moves and the other mimics its behavior. To perform an action, observations must be made, and for this purpose the environment must be encoded into a map, so each robot can induce its position from the observations. To perform observations the sensors supplied with Lego Mindstorm will be an obvious candidate. Lego Mindstorm has its own programming language, but it is not very expressive. Enthusiasts have however implemented a library, 'NXT++'[Com09] which allows us to implement the project using C++. The code can be compiled using a regular gcc compiler, but Bluetooth communication is so far restricted to Windows OS.

The focus of the project will be applying the I-POMDP theory to an implementation, and modeling the robots using said theory. I-POMDP is a mathematical framework, and cannot be implemented directly into code. By modeling the I-POMDP using Hugin Expert[Hug09], we can access the model during runtime, to update it based on the observations, and retrieve which action to perform. Hugin is a decision tool which support modeling decision problems as Dynamic Influence Diagram (DID), so the I-POMDP must be modeled into this context. There are API's for Hugin in different languages, including C++.

This report will covers the development of the I-POMDP and the modeling of the robots. The techniques used for the implementation along with related work will be presented, and finally the model will be tested and evaluated, with regards to whether or not it fulfills the goals. The report is structured as follows.

- Chapter 3 - Background

  Introduces the theoretical framework, the means of graphical modeling of the frameworks and finally how to encode the environment.

- Chapter 4 - Lego Mindstrom

  Investigate the precision of the sensors and actuators supplied by Lego.

- Chapter 5 - WiiRemote

  Covers the process of developing a position system using Wii Remotes.

- Chapter 6 - Implementation

  A overview of the design of the application, and description of the implmentation.

- Chapter 7 - Experimental results

  Experiments used to very the implementation and the models.

- Chapter 8 - Future Work

  A discussion of future developments and improvements that can be made to the model and the implementation.

Chapter 3

Background

Robot navigation is usually focused on a single robots interaction with its environment. Entering another robot into the same environment makes the scenario more complex, especially if the robots have to cooperate in order to achieve a common goal. This chapter will present the mathematical framework for modeling two robots, with focus on how to implement the framework using graphical modeling. The environment must be incorporated into the framework as well, so the robots can have a 'perception' of where they are, and thereby determine which action to perform. The final part of this chapter will look into means of modeling the environment.

3.1 Decision Process

The Markov Decision Process (MDP) is a mathematical framework for modeling a decision process where uncertainty regarding the outcome of the actions exist. The framework is a purely theoretical approach, which requires the agent to perform perfect observations of the state. Working with agents, the observations have to be made using sensors, which are prone to uncertainty, making it impossible to perform a perfect observation. The *Partial Observable Markov Decision Process* (POMDP) addresses this issue by utilizes a probability distribution of the current state. This approach is more suitable for agent navigation, but POMDP only models one agent. If other agents are introduced into an environment using POMDP, they are typically implemented as a static probability distribution over the agents preferences. The Interactive POMDP (I-POMDP) introduces the means of modeling several agents, where each agent observes and updates its own belief, to select the appropriate action. This section will describe POMDP and illustrate how it serves as a basis for I-POMDP.

3.1.1 POMDP

POMDP is a framework for observing the state of an environment and performing actions accordingly. In contrast to MDP, POMDP does not depend on exact observations of the current state. But similar to MDP, the environment must not change in between observing a state and performing the according action, limiting the interaction to be performed on a synchronous basis. Figure 3.1 illustrates this notion.

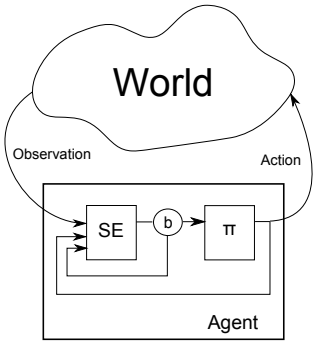

Figure 3.1: If the world changes inbetween perfroming an observation and before the action is made, the world state would have changed and the action might no longer be appropriate.

The belief state (b) is a probability distribution over all the states in the environment. A belief state with low entropy indicates the agent has a good 'awareness' of its current position. High entropy suggests that the agent is lost. The state estimator (SE) updates the belief state after an action has been performed. As it can be reviewed in Figure 3.1, the SE has three arguments, the old belief state, the action which has been performed and the resulting observation. The POMDP can be described as a tuple consisting of six variables $\langle$**S**, Ω, **A**, **T**, **O**, **R**$\rangle$.

- **S** is a finite set of states of the world.
- Ω is a finite set of observations which can be made in the world.
- **A** is a finite set of actions.
- **T**: **S** $\times$ **A** $\rightarrow \prod$(**S**) is the state transition function.

  Pr(s_{t+1}|s_t,a_t), denotes the probability of s_{t+1} conditioned on the action a_t and state s_t
- **O**: **S** $\times$ **A** $\rightarrow \Pi\,(\Omega)$ is the observation function.

  Pr(O_{t+1}|s_{t+1},a_t), denotes probability of O_{t+1} conditioned on the action a_t and state s_{t+1}
- **R**: **S** $\times$ **A** $\rightarrow \mathbb{R}$ is the reward function.

  $R(s, a)$ denotes the reward of performing action a in state s.

Similar to MDP, the policy is calculated using a value function. To this purpose the reward function is used, but in contrast to MDP the reward is calculated using the belief state.

Value function

A *t*-step policy can be considered as a tree. The top node depicts the first action, and given the observations which can be made, follows an arc to the next level, where the next action is determined. The tree depicts an agents conditional behavior[1] for *k*-steps, review Figure 3.2.

In the most simple case the policy is a 1-step tree, which is just an action. The reward for executing that action in state s can be described using the reward function.

$$V_\pi(s) = R(s, a_\pi) \tag{3.1}$$

[1]Conditional because it is based on the observation.

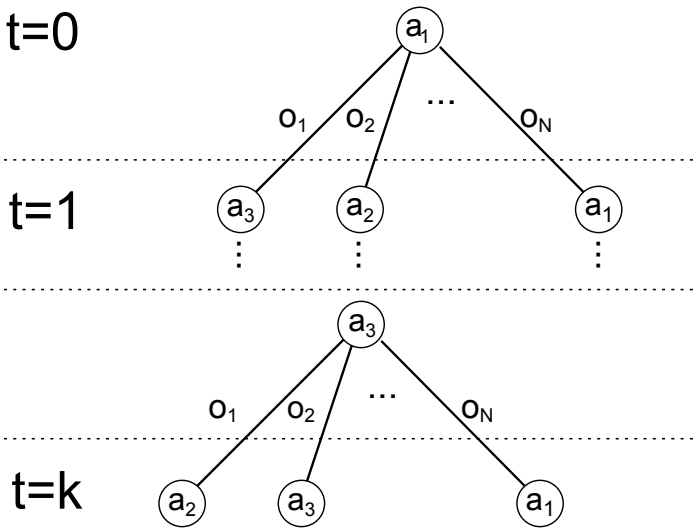

Figure 3.2: A policy tree of *k* horizons.

In Equation 3.1, a_π depicts the action suggested by policy π. If π was a t-step policy tree, we can describe the accumulative reward by adding the expected reward of future actions proposed by π, to the reward of performing the action at the root node of the policy tree.

$$\begin{aligned} V_\pi(s) &= R(s, a_\pi) + \text{Expected Value of the future.} \\ &= R(s, a_\pi) + \sum_{s' \in S} Pr(s'|s, a_\pi) \sum_{o_i \in \Omega} Pr(o_i|s', a_\pi) V_{o_i - \pi}(s') \\ &= R(s, a_\pi) + \sum_{s' \in S} T(s, a_\pi, s') \sum_{o_i \in \Omega} O(s', a_\pi, o_i) V_{o_i - \pi}(s') \end{aligned} \tag{3.2}$$

The expected value of the future is defined by considering all possible next states, s', and calculating the reward, $V_{o_i-\pi}(s')$, for performing the policy in those states. The reward is however depended of which policy subtree is executed, and the policy subtree is depended of which observation the robot receives, so this is also considered for all possible observations.

The agent does not know it accurate state, and must therefore compute the accumulated reward based on a belief state b, see Eq. 3.3.

$$V_\pi = \sum_{s \in S} b(s) V_\pi(s) \tag{3.3}$$

A belief state with high entropy will yield a low accumulative reward, and low entropy will yield a high reward. The policy which maximizes the accumulative reward is the optimal policy, and is the solution to the POMPD[LPKC95].

The reward function will be defined by the developers, and will depict the purpose of the agent's actions, e.g. treasure hunting by increasing the reward when the agent moves closer to the treasure. The reward is calculated on the basis of the belief state and having a good transition function and observation function becomes crucial for archiving a high reward.

A transition function depicts the accuracy of the actuators installed on the robot. A robot with e.g. motors which cannot drive straight will eventually get lost. The transition function is usually

defined by sampling numerous times over each action, deriving a probability distribution over the success rate for each action. Similarly the observations function is derived by sampling over which observations that can be made by an agent in each state, allowing the robot to induce its current state.

POMDP is suitable for modeling a single agent in an environment. It was not intended to model other agents in the same environment. Different agents can be introduced, but they will be a static probability distribution over the preferences of the agent.

3.1.2 Interactive POMDP

I-POMDP generalize POMDPs to a multi-agent setting by including the other agents' models as part of the state space. Agents have different beliefs concerning the models of other agents, and their belief about others. The state space with respect to I-POMDP is dubbed the *interactive state space*(IS). As the belief can become infinitely nested, we will only look in to finitely nested I-POMDPs.

A finitely nested I-POMDP with the strategy level l, for agents i and j, can be described as a tuple containing six variables $\langle \mathbf{IS}_{i,j}, \mathbf{A}, \Omega_i, \mathbf{T}_i, \mathbf{O}_i, \mathbf{R}_i \rangle$

- $\mathbf{IS}_{i,j}$ is a finite set of interactive states, which is a combination of the states for agent *i* and the models of agent j.

  $\mathbf{IS}_{i,j} = S \times M_{j,l-1}$

- $\mathbf{A} = A_i \times A_j$ is a finite set of joint actions which can be made by all agents.

- Ω_i is a finite set of observations which can be made in the world by agent *i*.

- $\mathbf{T}_i$ is the state transition function. It reflects the uncertainty effect of joint actions on the physical state.

  $\mathbf{S} \times \mathbf{A} \times \mathbf{S} \rightarrow$[0,1].

- $\mathbf{O}_i$ is the observation function. It describes the probability for agent *i* to receive an observation given the physical state and the joint actions.

  $\mathbf{S} \times \mathbf{A} \times \Omega_i \rightarrow [0, 1]$ is the observation function.

- $\mathbf{R}_i$ is the reward function. It describes agent *i*'s preferences over $\mathbf{IS}_i$ and $\mathbf{A}$.

  $\mathbf{IS}_i \times \mathbf{A} \rightarrow \mathbb{R}$

The interactive state is a combination of the states of agent *i* and the intentional and sub-intentional models of agent *j*. The intentional models describe belief, preference and rationality in action selection, whereas a sub-intentional model maps the history of *j*'s observations to probability distributions over its actions. The solution to an I-POMDP is the policy of agent *i*, which maps the belief over the interactive state to an action. Analogous to POMDPs, I-POMDPs are solved in two steps, updating the belief state, b, and computating the policy π.

I-POMDP belief update

There are differences, compared to POMDP, which complicates belief update in I-POMDP. The interactive state is depended of the joint actions of all agents in the environment. Agent *i*'s prediction of the state has to be based on a prediction of *j*'s action based on models contained in the *model* node of *i*. Furthermore if agent *j* acts according to its intentional models, its belief will be updated according to its actions and observations. Agent *i* will have to update its belief based on predictions of what agent *j* would observe. If agent *j* acts according to its sub-intentional models, its observations will be based on the history of observations which are contained in the model.

If agent *j* is modeled as an I-POMDP, it will have a model of agent *i* in its model. Invoking agent *j*'s belief update will in turn invoke the belief update of agent *i*'s belief update. The recursive call to each other's belief update will bottom out at strategy level 0, where the respective agent's IS only contains its own model, yielding a regular POMDP. Equation 3.4 formally describes the belief update of an I-POMDP.

$$\begin{aligned} Pr(is^t|a_i^{t-1}, b_{i,l}^{t-1}) \quad &= \beta \sum_{IS^{t-1}:\widehat{m}_j^{t-1}=\widehat{\theta}_j^t} b_{i,l}^{t-1}(is^{t-1}) \sum_{a_j^{t-1}} Pr(a_j^{t-1}|\theta_{j,l-1}^{t-1}) \\ &\times O_i(s^t, a_i^{t-1}, a_j^{t-1}, o_i^t) T_i(s^{t-1}, a_i^{t-1}, a_j^{t-1}, s^t) \\ &\times \sum_{o_j^t} O_j(s^t, a_i^{t-1}, a_j^{t-1}, o_j^t) \delta_K(SE_{\widehat{\theta}_j^t}(b_{j,l-1}^{t-1}, a_j^{t-1}, o_j^t) - b_{j,l-1}^t) \end{aligned} \tag{3.4}$$

The new belief is conditioned on the old action a_i^{t-1}, and the old belief $b_{i,l}^{t-1}$. The updated physical state of agent *i* is derived by the transition function T_i, and corrected using the observation function O_i. But both depend on the action of agent *j*, and because that action is uncertain, it is derived as a probability distributions over all actions conditioned on the old model $\theta_{j,l-1}^{t-1}$. As agent *j* acts and observes to update its own belief state, the belief state stored in agent *i* of *j* must be updated. The belief state is also dependent of the resulting observation, after having performed action a_j^{t-1}. The observation is however also uncertain and is derived as a probability distribution over all the observations. Given the new observation, old action and the old belief of agent *j*, it is possible invoke the state estimator $SE_{\widehat{\theta}_j^t}(b_{j,l-1}^{t-1}, a_j^{t-1}, o_j^t)$. $b_{j,l-1}^t$ is the correct belief, and given the updated belief is correct, the subtraction will return '0'. δ_K is the Kronecker delta and will return '1' if the argument is '0', and '0' otherwise. If the Kronecker delta returns '1', it indicates that the correct action and observation of agent *j* has been derived. But if the Kronecker delta returns '0', it indicates it is the incorrect action or observation, their influence is however neutralized as multiplying with '0' will return '0'. The results are finally normalized with β, which is a normalizing constant.

Value function

Each belief state of agent *i* has an associated value reflecting the pay-off the agent can expect for performing a given action. Equation 3.5 illustrates how the pay-off can be derived given the belief state and model of agent *i*.

$$\begin{aligned} U^n(\langle b_{i,l}, \widehat{\theta}_i \rangle) \quad &= \max_{a_i \in A_i} \Bigg\{ \sum_{is \in IS_{i,l}} ER_i(is, a_i) b_{i,l}(is) + \gamma \sum_{o_i \in \Omega_i} Pr(o_i|a_i, b_{i,l}) \\ &\quad U^{n-1}(\langle SE_{\widehat{\theta}_i}(b_{i,l}, a_i, o_i), \widehat{\theta}_i \rangle) \Bigg\} \end{aligned} \tag{3.5}$$

$ER_i(is, a_i)$ calculates the reward based on the interactive state is, and an action a_i. The is contains state of both agents, which is determined by their individual action. The action of agent *j* cannot be viewed directly and has to be described as a probability distribution conditioned on its model $m_{j,l-1}$. In Equation 3.5 $ER_i(is, a_i) = \sum_{a_j} R_i(is, a_i, a_j) Pr(a_j | m_{j,l-1})$.

For a finite case horizon the with the discount factor γ, the optimal action a^* for agent *i* is an element of the set of optimal actions for the belief state, review Equation 3.6. Similar to POMDP, the optimal policy is a mapping from a belief state to a set of optimal actions.

$$\begin{aligned} OPT(\langle b_{i,l}, \widehat{\theta}_i \rangle) &= \underset{a_i \in A_i}{argmax} \left\{ \sum_{is \in IS_{i,l}} ER_i(is, a_i) b_{i,l}(is) + \gamma \sum_{o_i \in \Omega_i} Pr(o_i | a_i, b_{i,l}) \right. \\ &\quad \left. U^n(\langle SE_{\widehat{\theta}_i}(b_{i,l}, a_i, o_i), \widehat{\theta}_i \rangle) \right\} \end{aligned} \tag{3.6}$$

3.2 Graphical Modeling

POMDPs and I-POMDPs can be modeled using Influence Diagrams (ID) for a particular state. IDs extend the BN with syntactic features which provides the means of encoding the decision problem as a graphical model, as well as representing the structure of the decision sequence[JN07].

3.2.1 Influence Diagram

Influence diagram (ID) is a well known graphical formalism used to describe decision problems. The ID framework can be considered a decision tool extending BNs, with different types of nodes. There are three types of nodes, a decision node, chance node and a utility node, each representing different aspects of a decision process[Lab09].

Decision Node
A decision node represents the decisions which are available for the decision maker. The node is used to model the impact of the different decision on the utility node. A decision node is illustrated by a rectangle.

Chance Node
A chance node represents the uncertainty of the different decisions, and it is illustrated by an oval.

Utility Node
A utility node is a measure of desirability, indicated by an integer value of the outcome of the decision process. A utility node is illustrated by a diamond.

To define the influence of the different nodes on one another, arrows are used, similar to BN's. But certain rules must be upheld by the graph. The formal description of the requirements is listed below.

- A directed acyclic path compromising all decisions nodes.

- Utility nodes, have no children.
- Decision- and Chance nodes have finite states.
- Utility nodes have no state.
- An ID is *realized* if the following quantities are specified.

  [-] *A conditional probability table P(A|pa(A))*[2]*, is attached to each chance node A.*

  [-] *A real-valued function over pa(U) is attached to each utility node U.*

Given the framework it is possible to model a decision problem, we have made a simple example based on an investment problem, see Figure 3.3.

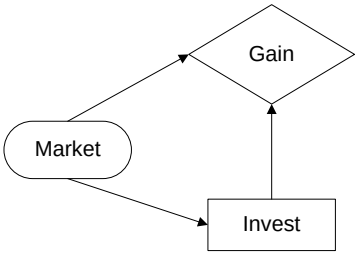

Figure 3.3: Problem of investment modeled using ID.

When investing in stocks it is important to observe the market, to know which stocks are eligible and which are not. In this example we have three stocks 'A', 'B' and 'C'. Of these three, 'A' is most likely to yield the highest profit, hereafter 'B' and finally 'C'. This is modeled[3] into the 'Market' node, review the table below.

Market	
A	0.7
B	0.2
C	0.1

The decision node is conditioned on the 'Market' node, and lists the actions available for the user. We wish to figure out which stock will yield the highest pay-off for the investor, and for the moment we just assigned uniform distribution to the node. Please notice that the investor can only invest in one stock at a time.

Invest			
Market	A	B	C
Invest_A	1	1	1
Invest_B	1	1	1
Invest_C	1	1	1

The Utility node is dependant of both 'Market' and 'Invest'. Its purpose is to indicate the gain of buying a given stock under the market conditions.

[2]The probability of A, given the parent of A.

[3]The probabilities are made up.

Gain									
Invest	Invest_A			Invest_B			Invest_C		
Market	A	B	C	A	B	C	A	B	C
Utility	1.0	-0.5	-0.2	-1.0	0.5	-0.2	-1.0	-0.5	0.2

Each stock has been assigned utility values according to their probability of yielding the highest profit, but the loss of not investing in that stock is also accordingly lower. This model is very naive and does in no way correspond to a credible scenario in the stock market. The model is implemented using Hugin, and saved as 'Market.oobn', it can be found on the enclosed CD (cd:/Hugin_Models/).

Giving evidence on the different decisions and propagating the model, will reveal the different utility values. We have calculated the values using Hugin, and the results are displayed in the table below.

Stock	Utility
A	0.58
B	-0.62
C	-0.78

The negative utility values do not mean the investor will lose their money, but investing in stock 'A' will yield the highest outcome. Stock 'B' is the runner-up and finally the 'C' will yield the lowest pay-off, for the investor.

3.2.2 Dynamic ID

A Dynamic ID extends the ID in terms of time series. Certain scenarios require a sequence of decisions to be made before the desired goal of the decision maker is achieved. Following the investment example, if a bank, e.g. wishes to invest in different stocks, there are a variety of aspects which must be considered before the stocks can be selected. First and foremost, the bank must assure, it does not invest in competing companies, as competition will eventually reduce the accumulative financial gain. In essence an investment will influence the market and the success rate of other investments. This problem can be represented as a POMDP, and modeled using DID.

The stocks can be considered as the state space, but the consequences of performing an investment on the stocks cannot be determined exactly. Observations can be used to assign probabilities over the different stocks, allowing the user to deduce the current state. The model will be extrapolated over three decision, where each decision represents an investment, review Figure 3.4.

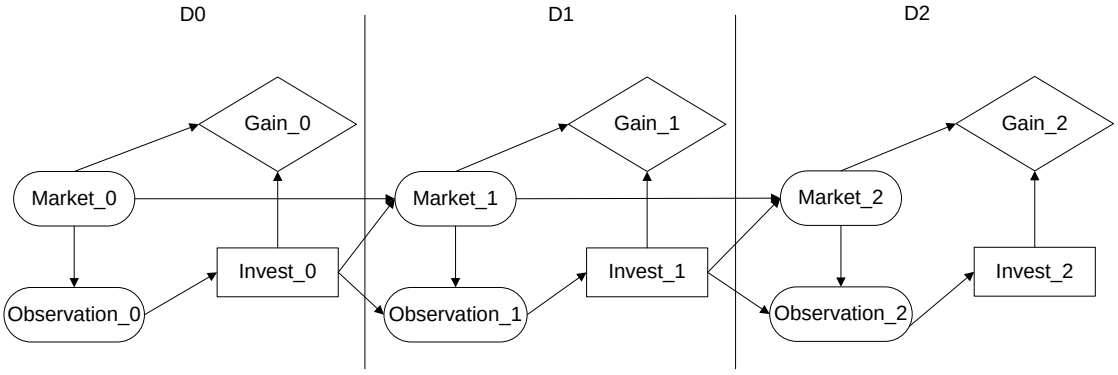

Figure 3.4: Problem of several investments modeled using DID.

The transition- and observation functions are modeled in the DID by the influences of the nodes. The transition function is described by $P(Market\_1|Market\_0, Inveset\_0)$ of the chance node '$Market\_1$', and the observation functions by $P(Observation\_1|Market\_1, Inveset\_0)$ of the chance node '$Observation\_1$'. The gain for the different combination of investments can be calculated by the utility function, and the sequence of investments which yield the highest utility award, will be the optimal policy.

A POMDP can be modeled using DID, but it does not solve the POMDP. By inducing evidence on the different actions it allows the user to derive the optimal policy. A DID only supports modeling of other agents as chance nodes, which will be a static probability distribution over the agents preferences.

3.2.3 Interactive DID

The Interactive DID (I-DID) is based on the Interactive ID (I-ID), which extends the syntactic features of the ID to include the models of other agents, by introducing the *model* node and the *policy link*. We will first look into I-ID and hereafter extrapolate to I-DID[DZC09].

Model node
The model node contains the models of other agents, which can be I-ID themselves. A model node is represented as a hexagon

Policy link
The policy link connects the model node to a chance node. The chance node will be a CPT over the actions of the agents conditioned on the models. The policy link is represented as a dashed link between the model node and the chance node.

If there are no other agents in the environment, the *model* node and *policy link* will vanish, reducing the I-ID a regular ID. Figure 3.5 illustrates a strategy level l I-ID for two agents. The model node contains the models of agent *j* at level $l-1$, which themselves can be I-ID. The chance node A_j holds a probability distribution over the actions of agent *j* conditioned on the model.

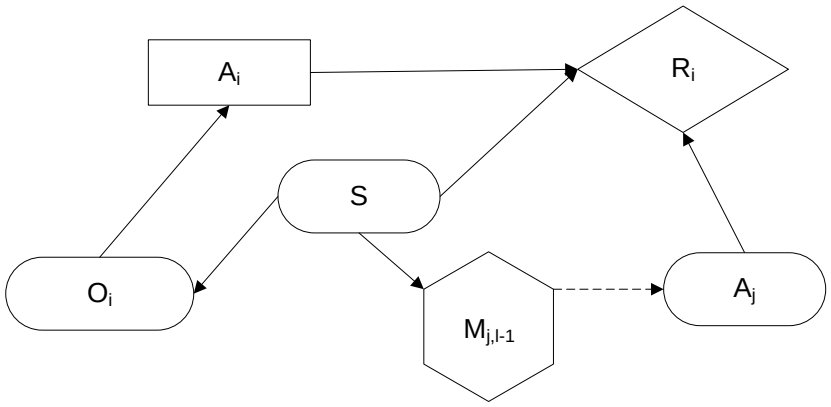

Figure 3.5: A general strategy level l I-ID for agents *i* and *j*.

Each model in $M_{j,l-1}$ can be an I-ID or ID, which when solved generates an optimal action in their decision node, for that given model. The decision node of an I-ID, $m^1_{j,l-1}$ in the model node, must be mapped to a corresponding chance node A^1_j. A^1_j will represent a probability distribution over the

actions of agent *j* given model $m^1_{j,l-1}$. Each level $l-1$ I-ID in the model node, will be mapped to its own chance node, and the actions which yield the highest value in each decision node are assigned a uniform distribution in the chance node, while the rest are assigned zero.

Figure 3.6(a) and 3.6(b) illustrates how an I-ID can be represented as an ID. The models $m^1_{j,l-1}$ and $m^2_{j,l-1}$ represent two models for agent *j*, which are mapped to their corresponding chance nodes A^1_j and A^2_j.

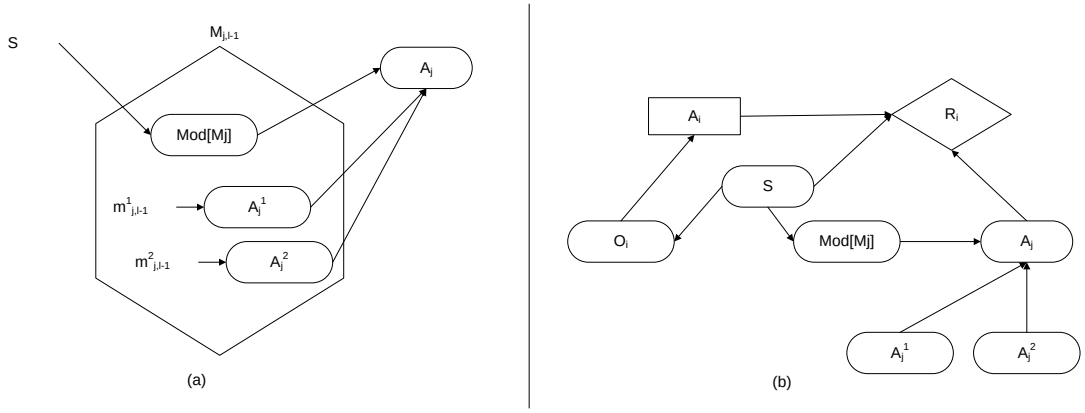

Figure 3.6: (a) The model node and the policy link using chance nodes and their dependencies. (b) The I-ID is represented as an ID.

The chance nodes A^1_j and A^2_j, holds the probability of performing different actions conditioned on their models. Along with $Mod[M_j]$ the chance nodes, A^1_j and A^2_j, form the parents of A_j. A_j is a multiplexer which assumes the values of either A^1_j or A^2_j conditioned on $Mod[M_j]$. The value of $Mod[M_j]$ denote the different models of agent *j*. In other words, if $Mod[M_j]$ is $m^1_{j,l-1}$, A_j will be equal to A^1_j, and if $Mod[M_j]$ is $m^2_{j,l-1}$, A_j will be equal to A^2_j. Notice that in Figure 3.6 the *model* node and *policy link* has been replaced with chance nodes and dependency links; the model is a regular ID.

An I-DID extends the I-ID in term of series to solve dynamic decision problems, similar to DIDs and IDs. Using the chance nodes and their influence on one another, it is possible to model an I-POMDP as an I-DID. Figure 3.7 illustrates an I-POMDP for two agents, modeled as an I-DID.

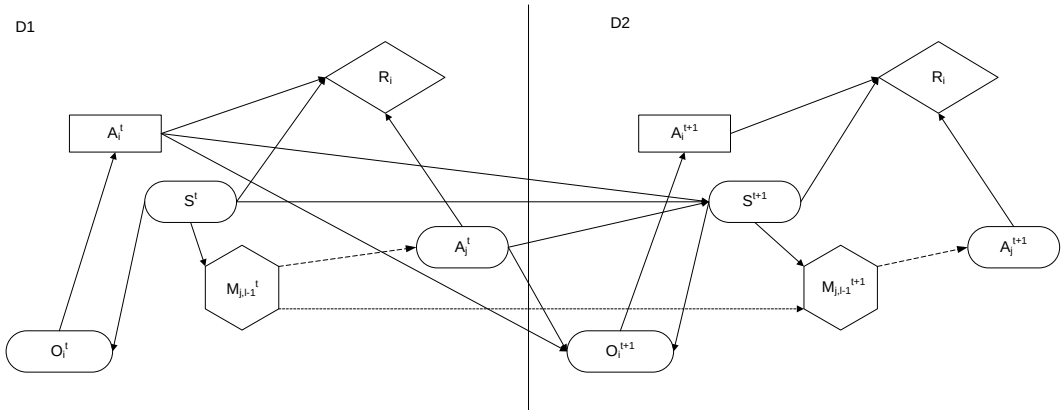

Figure 3.7: An I-POMDP for the agents *i* and *j* modeled as an I-DID.

The transition function of the I-POMDP is described by $P(S^{t+1}|S^t, A^t_i, A^t_j)$ of the chance node

S^{t+1} and the observation function by $P(O_i^{t+1}|S^{t+1}, A_i^t, A_j^t)$ of the chance node O_i^{t+1}.

The *model* node changes over times, and can be considered as the updated belief of agent *j*. In the I-DID it is illustrated using the *model update link*[4] between the model nodes, in Figure 3.7. The update of the model node involves two steps. First, given the candidate models at time t, identify the updated set of models in the *model* node at time $t + 1$. Secondly, compute the new distribution over the updated models given the original distribution and the probability of the agent performing the action and receiving the observation that led to the updated model. Figure 3.8 illustrates how the *model update link* can be implemented by chance nodes and dependency links.

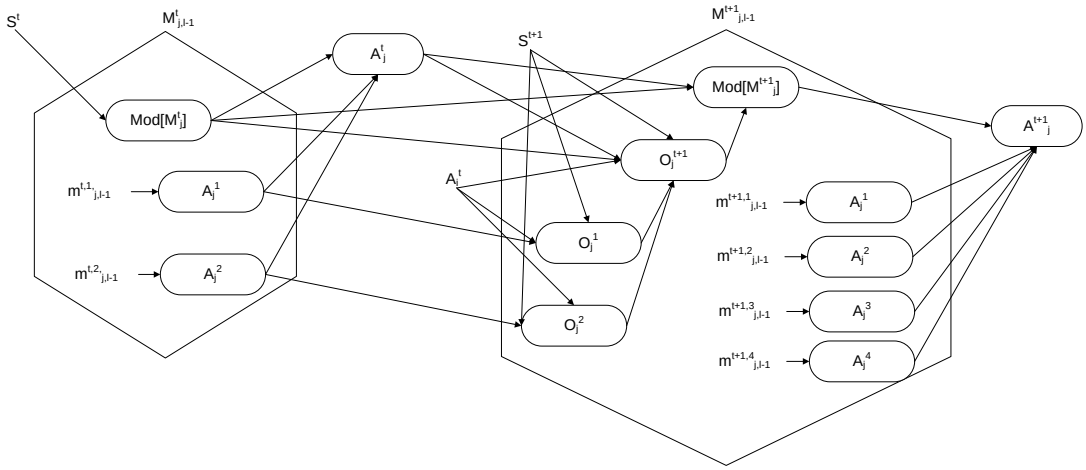

Figure 3.8: The *model update link* implemented using chance nodes.

The example illustrated in Figure 3.8 is an extension of the example in Figure 3.6. Given two models describing agent *j* at time step t, will result in one action and that agent *j* will then be able to perform two possible observations, the *model* node $M_{j,l-1}^{t+1}$ will contain four updated models[5]. Each of the models differ in their initial belief, each being a result of agent *j* updating its belief due to its action and a possible observation.

As previously mentioned, the chance node A_j^t and A_j^{t+1} assumes the distribution of the chance nodes over the actions, derived by the models, based on the value of $Mod[M_j^t]$ and $Mod[M_j^{t+1}]$. The probability of agent *j*'s updated model, e.g. $m_{j,l-1}^{t+1,1}$, is based on the previous action, the observation that led to the model and finally the prior distribution over all the models, $Mod[M_j^t]$. To retrieve the observation a chance node is introduced over the observations which can be made by agent *j*, O_j^{t+1}. Analogous to A_j^t and A_j^{t+1}, the O_j^{t+1} is a multiplexer modulated by $Mod[M_j^t]$. O_j^{t+1} assumes the distribution of the observation node, derived by acting according to the model suggested by $Mod[M_j^t]$. That is, given the value of $Mod[M_j^t]$, O_j^{t+1} will assume the distribution of either O_j^1 or O_j^2. The updated distribution over the models, $Mod[M_j^{t+1}]$ is conditioned on $Mod[M_j^t]$, A_j^t and O_j^{t+1}, and will represent the belief of agent *j*. Notice that the number of models increase exponentially, which will eventually become an issue, as the size of the model node becomes too big thus very time consuming to compute.

Similar to POMDP the action which maximizes the utility is the optimal action. The optimal policy is the set of actions which maximizes the accumulative reward, and will be the solution for the I-POMDP.

[4]The dotted line

[5]The models from $m_{j,l-1}^{t+1,1}$ to $m_{j,l-1}^{t+1,4}$.

3.3 Environment

In order to select an appropriate action, the state of the environment has to be observed. The observation depends on how the environment is encoded. There are different means of encoding an environment, each with advantages and disadvantages. This section will take a theoretical approach on defining a map, and how to implementation of a thereof.

3.3.1 Map

An observation can be single or a combination of readings from multiple sensors. An environment cannot be observed directly, but it can be encoded into a map, which specifies how observations can be made and how to interpret them. A map is a list of objects or features in the environment and their position. Formally a map m can be described as a set of objects and their properties, see Equation 3.7[TBF05].

$$m = \{m_1, m_2, \cdots, m_N\} \tag{3.7}$$

N is the total number of objects in the environment. Each m_n, where $1 \leqq n \leqq N$, defines an object and its property. The indexation of a map can either be done according the features or the location of features in the environment.

Feature based
A feature based map contains the location of features in the environment, all of which are uniquely identifiable. This method of representation is commonly used e.g. when triangulating the position of a cell phone according to the signal strength of different antennas, see Figure 3.9.

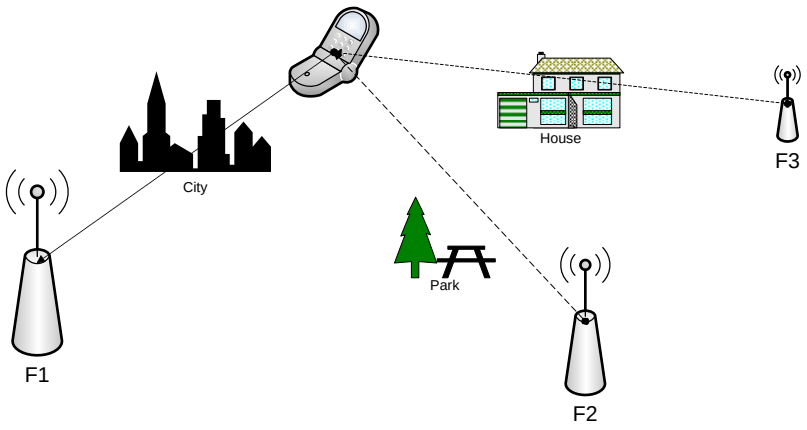

Figure 3.9: Triangulating the position of a cell phone, using the antennas.

Location based
A location based map is encoded such that every object is included, but the objects are not necessarily uniquely identifiable. This method of representation is useful in environments where objects are similar, but are distinguishable due to their location. This is the case for most hallways, thus is this method preferable for navigating inside buildings or within city borders, where no unique features

are available[6]. Figure 3.10 illustrates a robot in a hallway with 3 doors. The robot can distinguish the doors from each other, given the knowledge that Door 1 and 2 are close to one another while the 3^{rd} door is further away. Also notice that the 3 doors are identical, so it is not possible to use the feature based approach in this scenario.

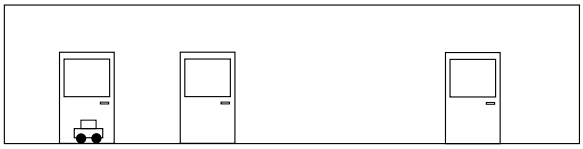

Figure 3.10: Robot using a location based approach for navigating a hallway.

Summery

Both encoding types have advantages and disadvantages. The location based encoding is volumetric[7], but allows for encoding pretty much everything, this includes areas without any objects. Feature based encoding only defines the shape of the environment, namely the uniquely identifiable objects. The encoding must be selected depending on the task at hand. If the robot is to create the map from scratch, the feature based encoding is suitable, as it does not require each location to be identified and encoded. On the other hand if the robot has knowledge of the environment, a location based map, allows for more precise navigation.

Our objective is to investigate I-POMDP, so a simple map is sufficient. The most simplistic representation is by the use of a grid cell map, see Figure 3.11.

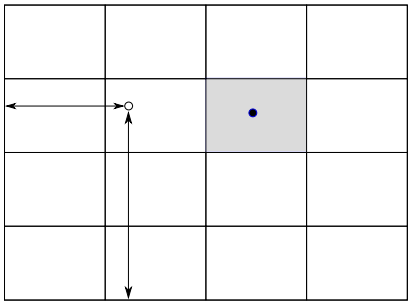

Figure 3.11: A 4x4 grid cell map.

A grid cell map can be encoded according to both features and locations. Each cell can be encoded using its distance according to the edges of the environment. If that is not suitable, the cells can be enumerated or given a different color, so each cell is uniquely identifiable.

We have examined at the Tribot Design[8], by Lego and established that each cell must be approx. $20 \times 20[cm^2]$ to fit the robot.

[6]This may not always be true, as smaller towns tend to have unique features.
[7]The size increase rapidly
[8]The Tribot is a basic robot design made by Lego, see [Gro09] for design details.

Chapter 4

Lego Mindstrom

Lego Mindstorm is a robotics kit consisting of sensors, actuators and an 'intelligent' brick, dubbed NXT. The NXT is a small computer, disguised as a brick, capable of executing code developed by the user. There exist several languages that can be compiled to the NXT, some object-oriented others following the functional paradigm. We have chosen NXT++ which is extension of C++ that is widely used in the world of NXT robotics and more importantly it is well documented[Com09].

There are a variety of sensors for Lego Mindstorm, the basic of these are included with the kit and other, more complex sensors can be bought separately. The distance, light, microphone, touch and compass sensor are at our disposal, and these can be utilized to perform localization in an environment.

The distance sensor can be used to measure the distance from the robot to the edge of the environment. The compass allows for the robot to perform turns in a precise manner and finally the light sensor can be used to distinguish cells of different color. The touch sensor and microphone are not applicable for performing localization and will henceforth not be considered. The actuators are two electrical motors that support both odometry and velocity motion.

The accuracy of localization is highly dependent of the integrity of each sensor, and the precision of actions are dependant of the accuracy of the actuators. Lego offers very sparse information regarding the sensors and none what so ever regarding the actuators. The integrity of the equipment has to be established so eventual uncertainty can be taken into consideration when performing localization or an action in the environment.

4.1 Distance sensor

The distance sensor is made of a speaker and a microphone. The speaker will emit a sound wave which is reflected by any objects in its path. The microphone registers the reflected wave and the distance can be derived given the time period between the wave is emitted and registered. The sound wave is emitted at a high frequency, and cannot be heard by humans nor does regular background noise interfere with it. Using sound waves to estimate the distance is prone to uncertainties, e.g. the spread of the waves, the object having indentations reflecting the sound wave in a different directions, etc.. The error coefficient for a measurement is set to +/-3 $[cm]$ by Lego, and the sensor can

measure up to 255 [cm]. The experiment investigates the validity of the limits set by Lego[Leg09c].

4.1.1 Setup

For the microphone to register the emitted sound wave, it must be reflected by an object. The direction of reflection is highly dependent of the surface of the object and the angle of approach; we wish to have it reflected directly back. Pointing the sensor straight at a wall will reflect the wave directly back and is under the circumstances the optimal setup. We will vary the distance to the wall from [3-220][cm] with different intervals, the setup is illustrated in Figure 4.1.

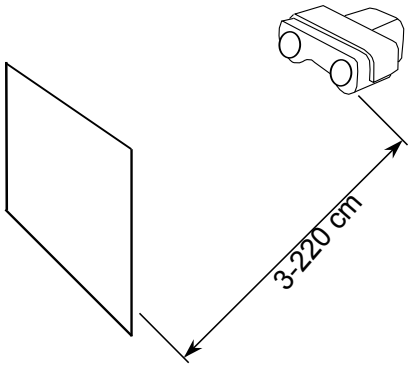

Figure 4.1: The setup of the experiment.

4.1.2 Results

The results of the experiment is illustrated in a Cartesian system, see Figure 4.2. There are 20 test cases, which are mapped on the X-axis and the sensor measurements are plotted using a curve. The sensor readings are depicted by the curve labeled 'Sensor', and 'Exact distance' is the correct distance. The curve 'Deviation' illustrates the difference between the sensor readings and the correct distance.

The sensor has problems performing close proximity readings, distances below 5 [cm] are consistently misinterpreted to 6 [cm]. From [10-100] [cm] it stays within the error coefficient stated by Lego, but measurements made above 100 [cm] have a deviation well above the error coefficient. When the distance is beyond 180 [cm] the sensor returns 255 [cm], indicating that it does not register the wall. This is in contrast to Lego's statement of the range being up to 255 [cm].

The distance sensor is only reliable when the object is between [10-100][cm] away, readings outside these boundaries are uncertain and cannot be relied upon. The short range makes the sensor difficult to work with, the distance sensor must be used in conjunction with other sensors, as its readings are unreliable to be used alone.

4.2 Compass Sensor

The compass sensor measures the heading according to the magnetic field of earth. The compass brick is composed of an electronic compass and a built-in calibration mechanism as means of re-

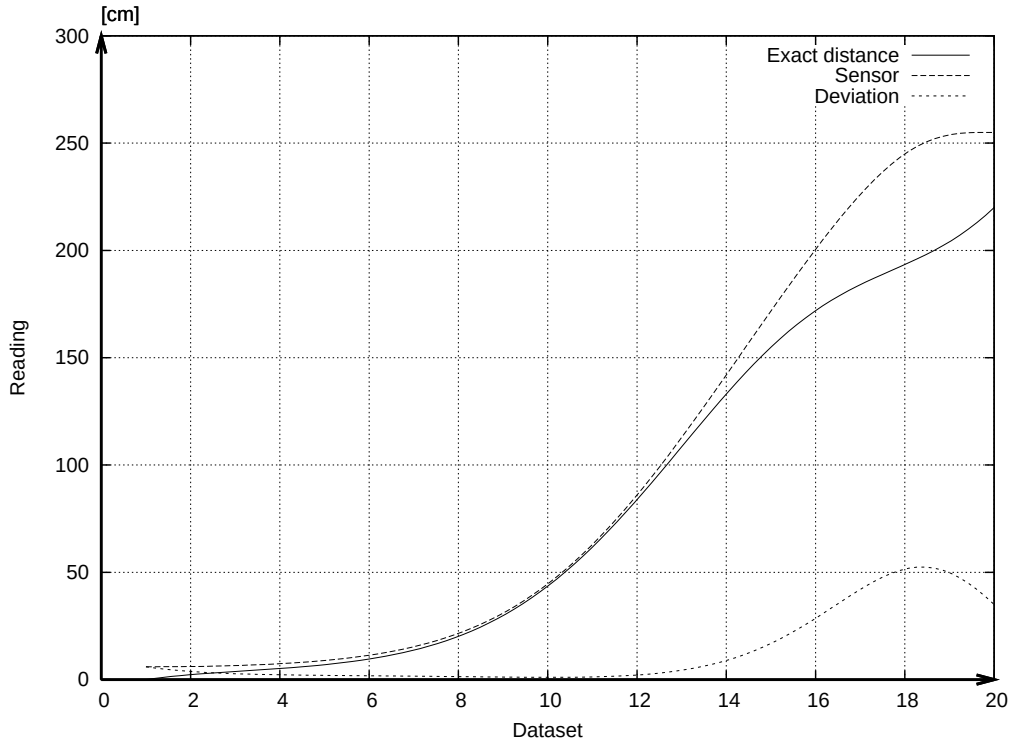

Figure 4.2: Close proximity readings and distant objects are difficult for the distance sensor to register.

ducing eventual magnetic interference from the surrounding environment. Even with the calibration mechanism too much magnetic interference can still cause erroneous readings. This experiment is to investigate the precision of the compass and the magnetic interference of the NXT[Leg09a].

4.2.1 Setup

To reflect a credible scenario the compass sensor will be mounted on the Tribot. The NXT brick is, as previously stated, a small computer. The electrical circuits in the NXT and the actuators, which are electrical motors, will generate a magnetic field. The influence of the magnetic field will be at its highest when the sensor is close to the NXT, but given an effective calibration mechanism it will be neutralized.

The experiment will be performed using two setups, one where the sensor is mounted close to the NXT, and one mounted far from it. When close to the NXT, the readings can illustrate the intensity of the magnetic field surrounding the Tribot and whether the calibration mechanism can cope with it. The optimal setup is however when the sensor is mounted far away from any magnetic interferences. The second setup will investigate the precision of the compass.

To determine the exact heading a compass will be used. When the heading has been established the Tribot, with the sensor mounted on it, will be placed facing that same direction. The Tribot will

be turned clockwise with an interval of 90° and the readings will increase for each turn, until a full turn is made. The readings of the sensor can be compared with the heading and any difference will eventually be due to magnetic interference.

4.2.2 Results

The results are depicted in graphs with the direction plotted on the X-axis and the sensor readings on the Y-axis, see Figure 4.3 and Figure 4.4. To give the readings a linear growth they are depicted in the following sequence: 'North', 'East', 'South' and 'West'. The readings are expected to grow with 90° for each turn. Figure 4.3 illustrate the results of the first experiment with the compass sensor mounted close to the NXT.

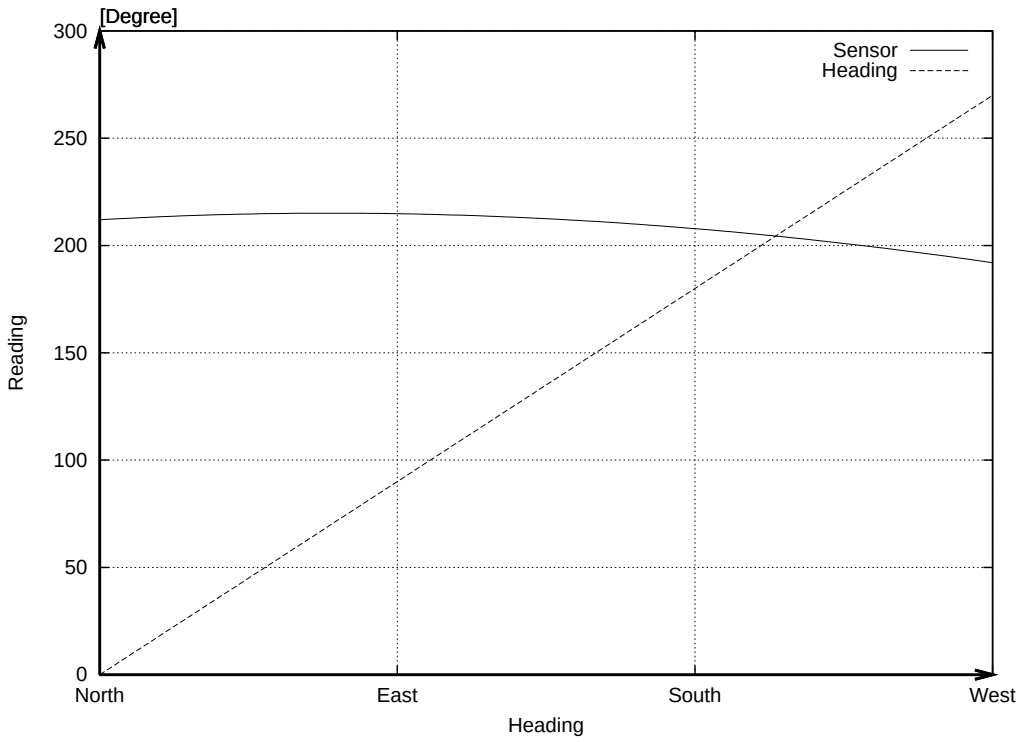

Figure 4.3: Compass mounted close to the NXT

The curve 'Sensor' depicts the sensor readings, and 'Heading' is the expected growth. 'North' represents the starting point at 0°, however the sensor reading is well above 200°. The readings of the compass are almost constant throughout the experiment, which is in contrast to the expected linear growth. The magnetic field of the Tribot is constant, and assuming it is around 200° could explain why 'North', 'East' and 'South' have the 'same' sensor reading. However at 'West' the reading is expected to reach 270°, which it does not. This could indicate an error with the sensor, but whatever the reason the conclusion of this experiment is clear. The built-in calibration mechanism is not capable of handling the magnetic field surrounding the Tribot, and mounting the sensor close to it will cause erroneous readings.

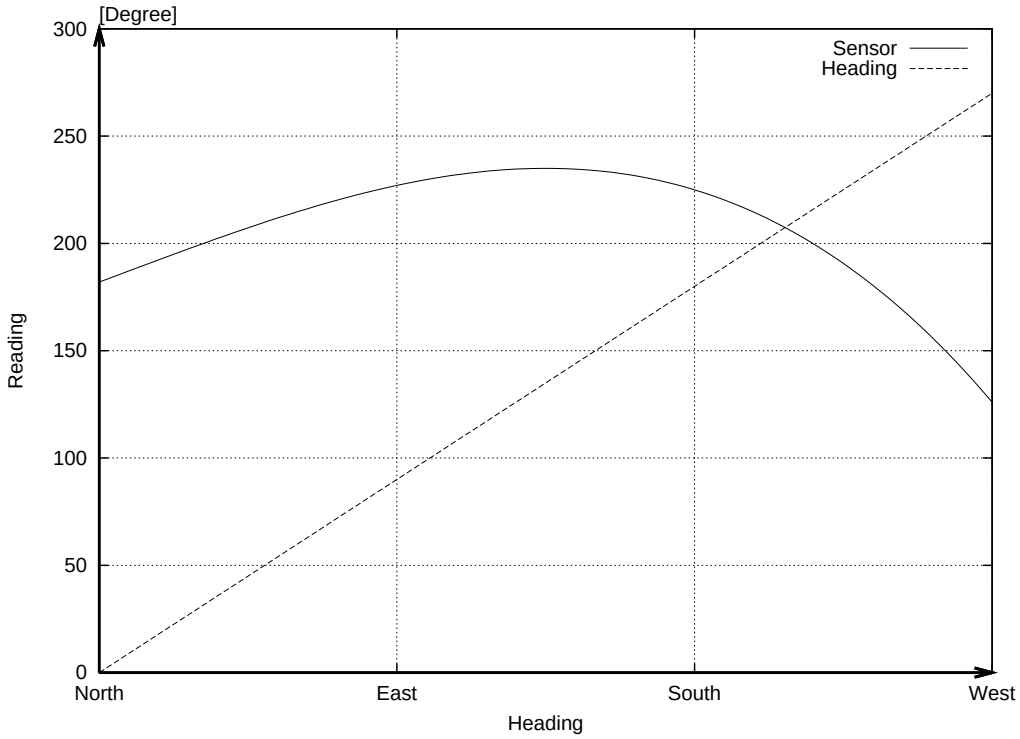

Figure 4.4: Compass mounted far from the NXT

Figure 4.4 illustrate the second experiment with the sensor mounted away far from the Tribot. The measurements made with this setup are somewhat peculiar. The growth between 'North' and 'East' is approx. $50°$, well below the expected $90°$. Between 'East' and 'South' the readings stagnate and even drop some degrees. 'West' has a sensor reading well below the expected $270°$, which suggest that the sensor has its magnetic 'North', close to 'West'. Neither the growth nor the direction suggested by the sensor matches the expectation or the heading.

The compass is intended to support the action of turning, and not to estimate the four corners of the world. To perform a turn it is important that the sensor can establish whether the heading has changed $90°$, but the results clearly indicate that it cannot perform this task. The inconsistent measurements make the compass sensor unreliable, and therefore not suitable to be used in the project.

4.3 Light Sensor

The light sensor can be applied to various aspects of navigation, e.g. to follow a line or differentiate cells of different color. The sensor is composed of a light emitting diode (LED) and a light sensitive camera. The LED illuminates the area, and the camera registers the reflected light. The sensor registers light in grayscale and cannot identify specific colors, but it can however distinguish them. There are some issues when utilizing the sensor in this manner. Ambient light will inevitably

interfere with the readings, but as long as it is consistent throughout the environment it does not impose a problem. If the environment is illuminated unevenly it will cause erroneous readings. By erroneous readings, we mean different color can give the same reading on the grayscale, or vice versa.

The distance between the sensor and its target will also influence the reading. If the sensor is mounted too close to its target the LED will overexpose the area, and the reading will become unnaturally high. Moving the sensor too far away from its target, the LED will become less intensive yielding a low grayscale value. In the latter case there is a great chance for an extern light source to influence the readings. The experiment is to investigate the sensors ability to distinguishing colors[Leg09b].

4.3.1 Setup

The initial problem is to have consistent lighting throughout the group room. The best result will undoubtedly be achieved in a completely blacked out room. This is however not a credible scenario, and further more we cannot see under such conditions. The experiment is performed with the blinds pulled and all the lights on. Given the circumstances, this is the optimal setup for an even illumination of the target.

The light sensor is mounted on the Tribot and the sensor is set to read the grayscale value of seven different colors. The distance between the sensor and its target will be varied from [1-6][cm] with 1 [cm] intervals. We have aimed at choosing colors which represent the color spectrum, and have settled on the following colors: 'White', 'Black', 'Red', 'Blue', 'Yellow', 'Pink' and 'Green'. The readings are expected to decrease as the height increases due to the light, emitted by the LED, will disburse.

4.3.2 Results

The experimental results are plotted in the graph below, see Figure 4.5. The distance between the sensor and colors is mapped on the X-axis and the grayscale readings on the Y-axis.

'White' and 'Black' are the two extremes and the other colors lie in between them. 'Green' is rather unique and is fairly in the middle of the two extremes. 'White', 'Yellow' and 'Pink' have grayscale values very close to one another when the height is between [1-3][cm], but grows apart hereafter. 'Red' is similar to 'Pink', but has a darker profile. 'Blue' and 'Black' have a similar patter as 'Red' and 'White' but at the very dark end of the scale.

At the distance of 4.5 [cm], the sensor is able to distinguish all the colors, however using 'White' and 'Yellow' in the same environment must be discouraged. Colors with grayscale values close to one another are more likely to be misinterpreted and are therefore very sensitive to ambient light.

Taking all seven colors in consideration, only 'Black' 'White' and 'Green' can be categorized as being 'safe'. They are far apart from one another, and cannot be misinterpret even with the influence of ambient light, the rest however can. Three colors are not enough to cover an entire grid cell map, as each cell must be assigned a unique color to perform exact localization. The colors can be reused, but this will increase the uncertainty when performing localization. Claiming the light sensor is imprecise would be wrong; on the other hand applying it to perform localization would be

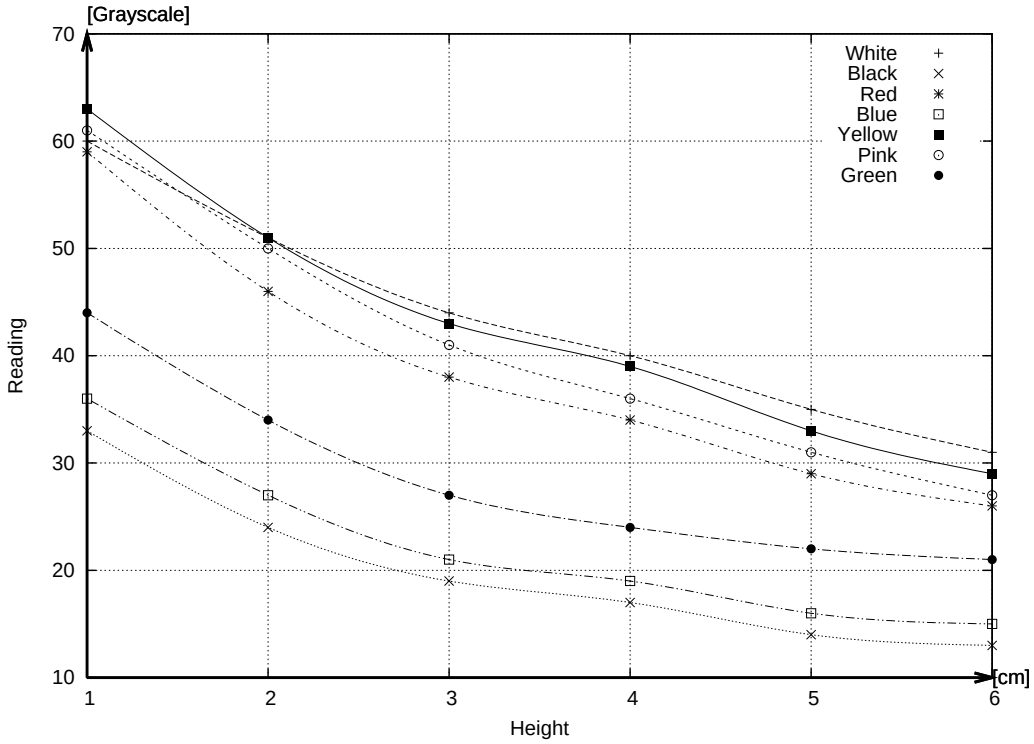

Figure 4.5: The grayscale value for different colors at different heights.

troublesome, as it cannot give a clear picture of the robots location.

4.4 Actuator

The actuators are two electric motors, each capable of clockwise and counter-clockwise propulsion. Motors in general can be controlled by either velocity- or odometry motion; the actuators support both motion control. To avoid misunderstanding here is a brief definition of the motion controls[TBF05].

Velocity motion : An action is translated into a number of rotations by each motor.

Odometry motion : An action is translated into a specific speed over a given time for each motor.

The NXT brick powers the motors, and the speed is determined by the voltage over each motor. The speed is set as an integer value between [0-100] in the implementation, '0' means the power is shut off and '100' is the maximal voltage over the motor.

4.4.1 Setup

To simulate a credible scenario the Tribot will be used for the experiment. The Tribot has two motors, one for each wheel and they are connected to port A and B on the NXT. The motor attached to Port A will henceforth be referred to as motor A, and the second will be referred to as motor B. The efficiency of the brakes on the motors is very poor, and the braking distance is proportional to the velocity of the Tribot. We have settled on a speed of '20', which is somewhat slow, but the Tribot will instantly perform a full stop when the brakes are engaged.

To investigate the precision of the motor and the motion controls, the Tribot will be set to move a predefined distance. Deviation from the goal can be considered an error, either due to the specific motion control or lack of precision by the motor. By performing the experiment using both motion controls it can be established whether the motors are unreliable or the motion control is the problem.
Velocity Motion
The Tribot will be set to drive 20 cm. The speed setting corresponds to an acceleration of approx. 540[$\frac{cm}{min}$][1]. With the given acceleration it will theoretically take 2.22 [sec]. for the Tribot to reach its destination. This will be the time period when performing the experiment using velocity motion.
Odometry Motion
The Tribot will be set to drive the same distance, as with velocity motion. Each motor has a built-in counter which keeps track of the rotations, but the rotations are of an internal shaft and not the ones made by the wheel. At the speed setting of '20', the Tribot moves approx. 4.95[$\frac{mm}{rotation}$][2], which corresponds to approx. 403 rotations to drive 20 cm. This will be the number of rotations when performing the experiment using odometry motion.

4.4.2 Results

Eleven test cases were made for each motion control, the results for velocity motion can be reviewed in Figure 4.6 and odometry motion in Figure 4.7. The test cases are depicted on the X-axis, and each test case contains the result for motor A and B. The straight line at 20 [cm] is the aim, and each motor should come as close to it as possible.

Using velocity motion, the motors behave very precise and the greatest deviation is merely 0.5 [cm][3]. It is noteworthy that the motors behave very differently; this is due to different torque and resistance of each motor. The Tribot generally does not drive far enough and the average deviation for motor A is 0.15 [cm] and 0.14 [cm] for motor B.

The results of odometry motion is depicted in Figure 4.7. The greatest deviation is 0.5 [mm][4], and the average deviation of motor A is 0.18 [cm] and for motor B 0.1 [cm]. The different torque and resistance still influence the motors even though this motion is based on rotations.

It seems impossible to ensure that motor A and B will cover the same distance, as both motion models suffer from the different torque and resistance of each motor. The different output of the motors will eventually change the heading of the robot, which will become a problem as it will accumulate over time. The greatest deviation is 0.6 [cm] at the 4^{th} test case under odometry motion.

[1]The Tribot was set to drive for a specific time duration and by measuring the driven distance, the acceleration was derived.

[2]The Tribot was set to drive for a specific number of rotation, and by measuring the driven distance, the measure was derived

[3]Motor A in test case 8 in Figure 4.6

[4]Motor A in test case 4 in Figure 4.7

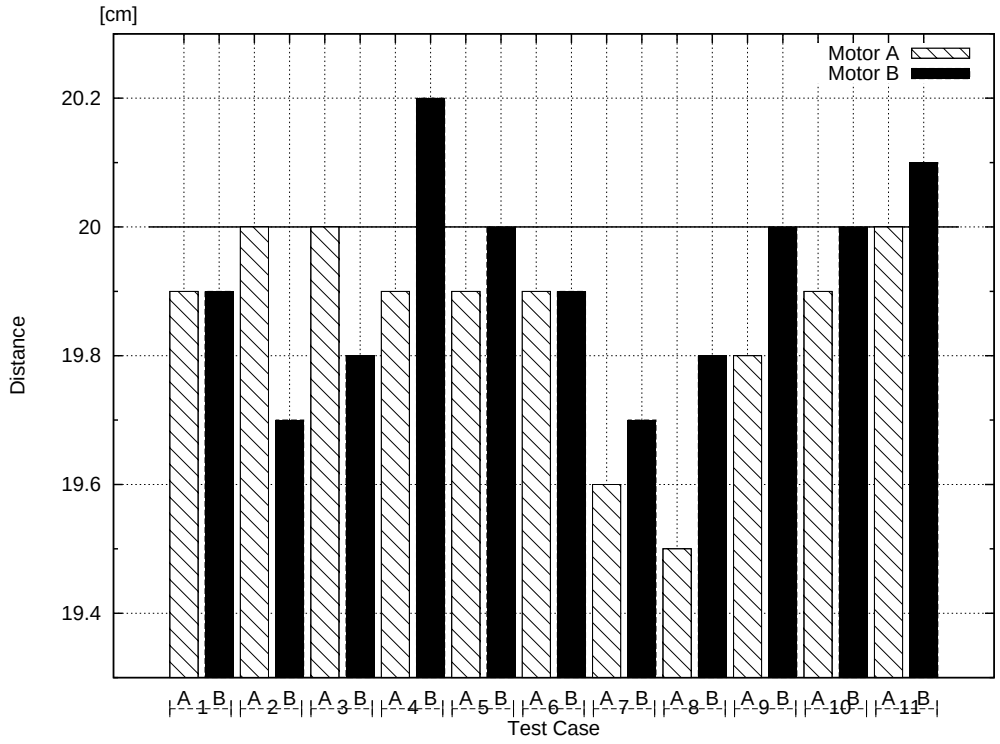

Figure 4.6: Results of the actuators using velocity motion.

The average deviation between the output of the motors is 0.26 $[cm]$ for odometry motion and 0.15 $[cm]$ for velocity motion. The only way to address this problem is to adapt the settings made in the implementation to each motor. This is not a lasting solution, and we chosen not to address this problem. We will use the same implementation for all motors.

In general terms the motors perform very well, one must keep in mind that the errors can be caused by other factors, e.g. the Tribot was not facing straight when it was set to drive. In contrast to the sensors the actuators are reliable, as long as an action does not require precision at millimeter level. It is surprising that velocity motion provides the most uniform output of both motors, where as odometry motion has a problem surrounding this issue. The difference between velocity motion and odometry motion is negligible, and they are both recommendable to be applied for motion control, velocity motion however has the edge, as it keeps a heading of the robot steady.

4.5 Summary

Lego Mindstorm is only a toy, made to entertain people and its sensors are not meant to perform exact measurements. The compass sensor does not perfrom sensible readings, and we are left with the distance sensor and the light sensor. The distance sensor is only reliable within [10-100][cm] of its target, and due to the limits of the light sensor, only a few colors can be used in the environment.

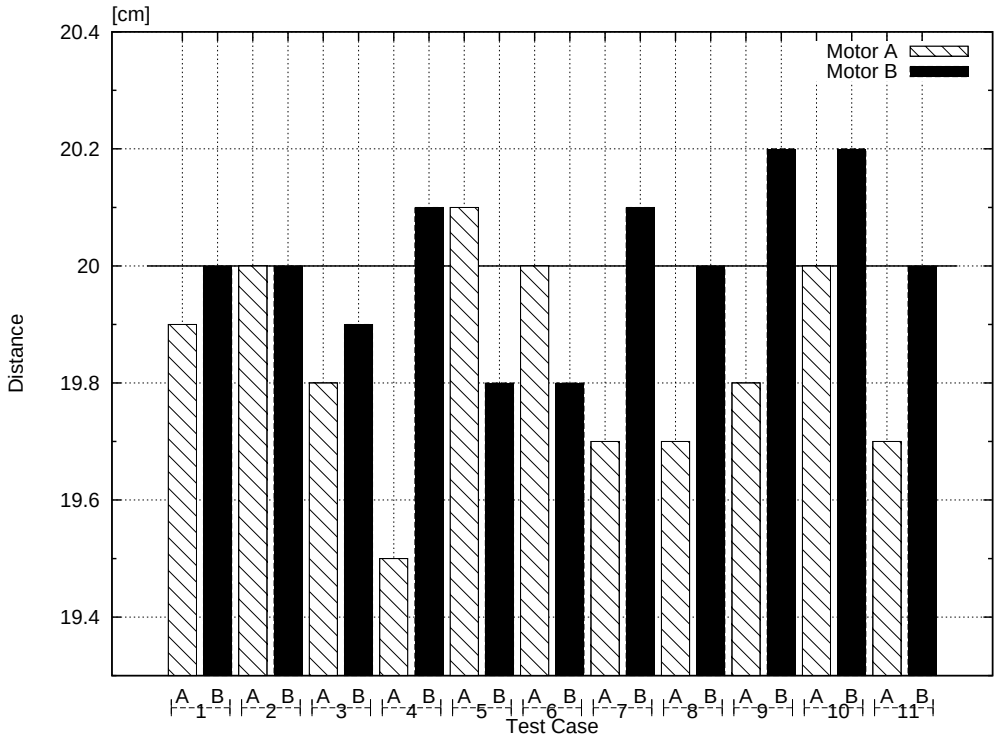

Figure 4.7: Results of the actuators using odometry motion.

Positioning the robot is a process which combines sensor readings with an internal logic, which will return the current posistion of the robot. This process will henceforth be referred to as localization. Performing localization with the sensor supplied by Lego is very difficult, and any results achieved using these sensors will be highly uncertain.

Chapter 5

WiiRemote

Performing localization of the robot using the LEGO Mindstorm sensors is a difficult task. Localization cannot be performed using just one sensor, and even if all were combined, the outcome will still be uncertain. Following this path further will not be prudent. There are however other means of localization, which can be utilized instead. One solution that has great potential is the Wii Remote.

The Wii Remote is a part of the Wii console, developed and produced by Nintendo. It is a gaming console which, in contrast to others, does not use a joy pad. Nintendo has utilized accelerometers in their human interface devices (HID) allowing for interaction using gestures.

The accelerometers are only used for gaming, during menu navigation a cursor similar to the mouse-courser is used to select the respective options. The cursor is controlled by the use of infrared light (IR). A bar made of IR diodes is placed at a static point, e.g. below the television. The Wii Remote is equipped with an IR camera which detects and tracks the IR diodes on the bar. As the bar is static, any movement of the IR diodes is caused by the user shifting the Wii Remote. The shift can be translated into cursor movement, somewhat similar to the principles of an optical mouse.

The Wii Remote is proprietary and Nintendo has not released any technical documentation regarding the API nor the hardware. Enthusiast around the world have however in some extend reverse engineered the Wii Remote, and created means of interacting with the Wii Remote. The Wii Remote can be used to perform localization by keeping it static, and using its camera to track an IR diode, that is mounted on a robot. The camera supports tracking of up to four individual IR diodes at any given time, so we have means of tracking 4 robots simultaneously. In contrast to the sensors supplied by Lego the IR camera provides almost exact readings.

5.1 Optical Tracking

Three Students at Cambridge University set out to create optical tracking using only commodity hardware[HNH08]. The Wii Remote can perform tracking of an IR diode, and the price of a single Wii Remote is affordable for the general population. At the top of each Wii Remote is the camera, see Figure 5.1, and IR diodes in its sight will automatically be fixated on.

The project supports tracking of an IR diode in a three dimensional (3D) environment by utilizing

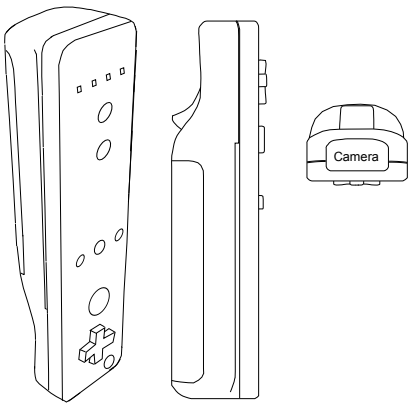

Figure 5.1: The Wii Remote from the front (left), the side (middle) and from the top (right).

two Wii Remotes. The Wii Remotes must be set orthogonally to one another allowing both Remotes to track the IR diode. One will be used to determine the X-coordinate and the other the Y-coordinate, both Wii Remotes can be consulted regarding the height of the IR diode. Figure 5.2 illustrates the basic concept of the setup.

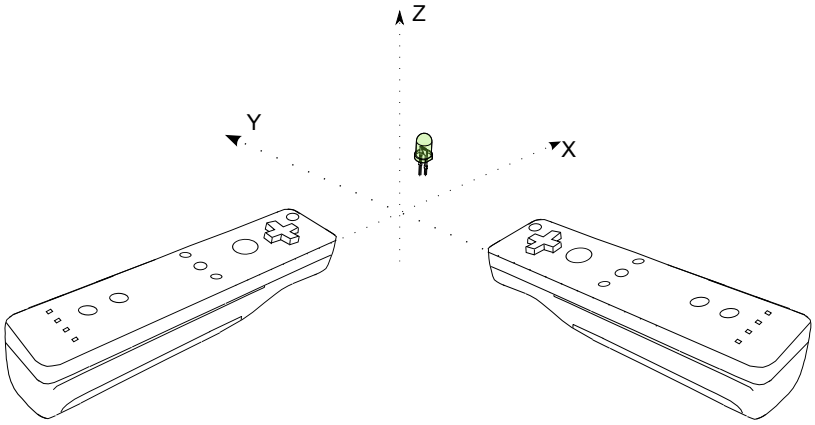

Figure 5.2: Localization in a 3D environment.

To function properly the software must be calibrated for each setup. The calibration is a process of defining the position of each Wii Remote with regards to each other, prior to performing localization. The calibration creates the 3D Cartesian system, in which the readings will be positioned. Without the calibration, the physical position of the Wii Remotes cannot be altered, making it difficult use the project in different scenarios.

The project has not reached its final release, and is currently under development. The documentation for the different aspects of the project is either under development, or has not been initiated yet. There is currently no documentation regarding calibration, and it must therefore be backtracked from the source code[SHH09].

The source code is available from their homepage, but once more, the lack of documentation regarding how and what is necessary to compile, makes it difficult to build the project. By reviewing the *'Make'* scripts it is however possible to reverse engineer a list of dependencies for the project. There are all together four *'Make'* scripts for the project, one for generating the GUI, one to build the library used to interact with the Wii Remotes and finally one to build the methods used for localization, which extend certain Matlab methods. These scripts are executed sequentially by a main

'Make' script from the root of the source folder.

The paths to certain application dependencies are hardcoded according to the local computer which was used to create the project and these must be changed for the project to compile. This problem is above all an issue with respect to Matlab as the paths for many header files, used during compilation, must be changed manually. The GUI is the last to be built, but it has a library dependency which we have not been able to determine. The application is set to output to the GUI and as it could not be built the project could not be compiled.

This was a great set back as we were forced to abandon the application, but the Wii Remote can still be utilized. Localization must however be performed in another manner.

5.2 Mapping

The solution created at Cambridge had to be abandoned, but the concept of using a Wii Remote to perform localization still has potential. We have access to libraries which provides the means of interacting with the Wii Remote, but the functionality to perform localization has to be developed. The easiest way to perform localization is to directly map a reading made with the Wii Remote to a position in the environment. We will show that direct mapping is not a possibility and a mathematical model has to be created to interpret the readings according to a virtual map, which represent the environment.

5.2.1 Direct Mapping

Mounting a Wii Remote on the ceiling with the camera facing down allows us to capture the pixel coordinate of the IR diode. The most simplistic mean of localization is direct mapping, which assumes the pixel coordinate can be directly converted into a position in the environment. Figure 5.3 is the grid cell environment, where each grid is a $20\times20[cm^2]$ square, as seen from the Wii Remote.

Each $+$ is a reading made with the camera at the intersection of the grids. The lines and the internal Cartesian system have been added later to ease the understanding of the illustration. Considering the horizontal edge at the bottom of the environment as an X-axis and the vertical edge on the left as an Y-axis, allows for any reading in the environment to be positioned according to the fixed origin $(0, 0)$, see Figure 5.3.

The IR camera and the environment is not aligned, which has a great influence on whether direct mapping is possible or not. The point (20,20) in the environment is the end of the first grid, and has the pixel coordinate (0.381,0.340). The point (80,20) in the environment is the end of the 4^{th} grid along the X-axis, which has the pixel coordinate (0.696,0.307). The points in the environment have the same y-coordinate, however the pixel coordinates deviate. This deviation can be mapped by a function, and thereby countered, but it is difficult as the deviation change depending on where along the Y- and X- axis the reading is made. For direct mapping to work seamlessly, the environment and the IR camera must be aligned to one another, see Figure 5.4.

Aligning the environment and the IR camera is very difficult, as the Cartesian system of the IR camera cannot be seen. The only approach is to try and estimate the alignment, by manually turning the Wii Remote gradually. The Wii Remote must however be taken down regularly to pair it with

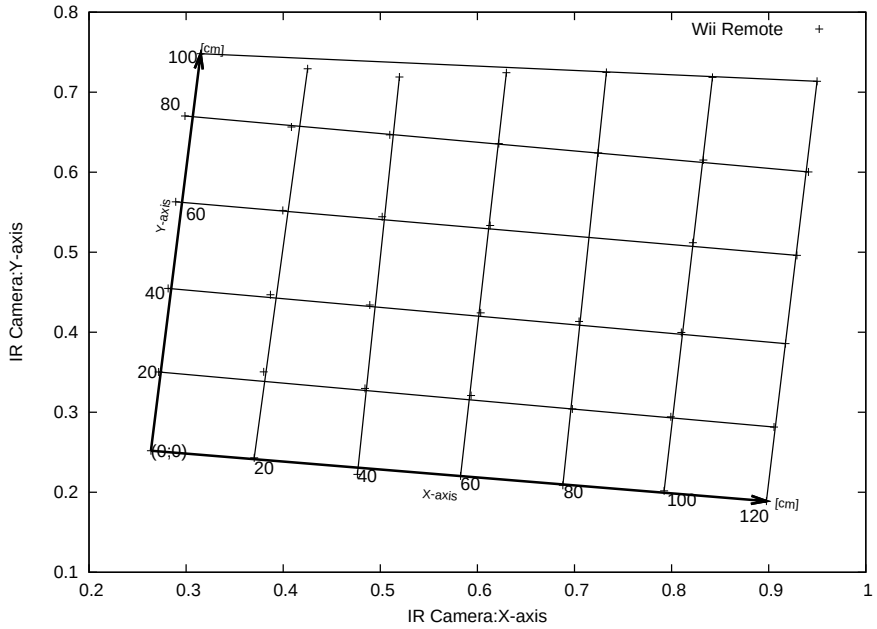

Figure 5.3: Mapping of the environment made with the IR camera.

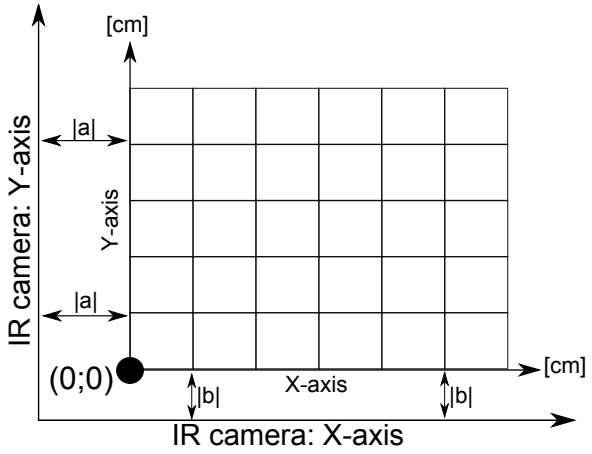

Figure 5.4: The environment and the IR camera aligned.

a PC, change batteries, etc., and each time alignment must then be repeated. Due to the difficult alignment process direct mapping is not a lasting solution.

5.2.2 Virtual Mapping

Even though direct mapping is not the solution, the setup can still be utilized and by using certain geometric properties the issues surrounding direct mapping can be solved.

For simplicity we will henceforth denote the edges of the environment, see Figure 5.3, for X- and Y-

axis. The edges can be considered as two straight lines, which exist in the Cartesian system[1] defined by the IR camera. A straight line can be described by a function, and by mapping the edges of the environment by two functions, any readings made in the environment can be described according to its relative position with respect to these functions, see Figure 5.5.

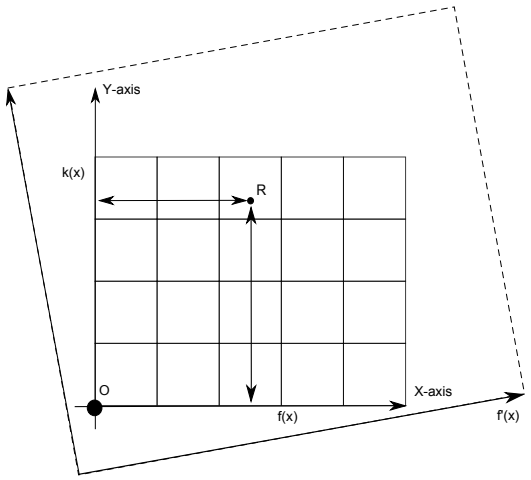

Figure 5.5: Distance to a reading, positioned according to the edges.

The area covered by a single the Wii Remote is not large enough and in cooperation with our counselor we have decided to extended setup with another Wii Remote, see Figure 5.6.

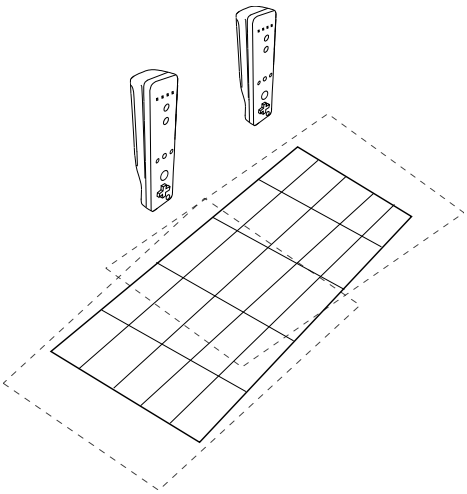

Figure 5.6: The setup of the Environment.

By the introduction of the second Wii Remote the problem becomes somewhat more complex. First and foremost each IR diode must still be localized, but now every reading made by second Wii Remote must be converted to fit the environment set by the first Wii Remote. Secondly, a handover mechanism must be implemented if more than one robot is to drive in the environment, to uniquely identify which robot is leaving the view of one Wii Remote and entering the other. The basic idea remains the same; depict the edges of the environment as two functions, but now one of the edges must be extrapolated into the view of the second Wii Remote.

[1]The Cartesian system is in fact the pixel array of the camera

Localization

Aligning the two Wii Remotes is not possible, and the views will be twisted according to one another, see Figure 5.7. The view of the first Wii Remote is denoted Q1 and the view of the second Q2; the displacement of the views is exaggerated, but this is to emphasize the point.

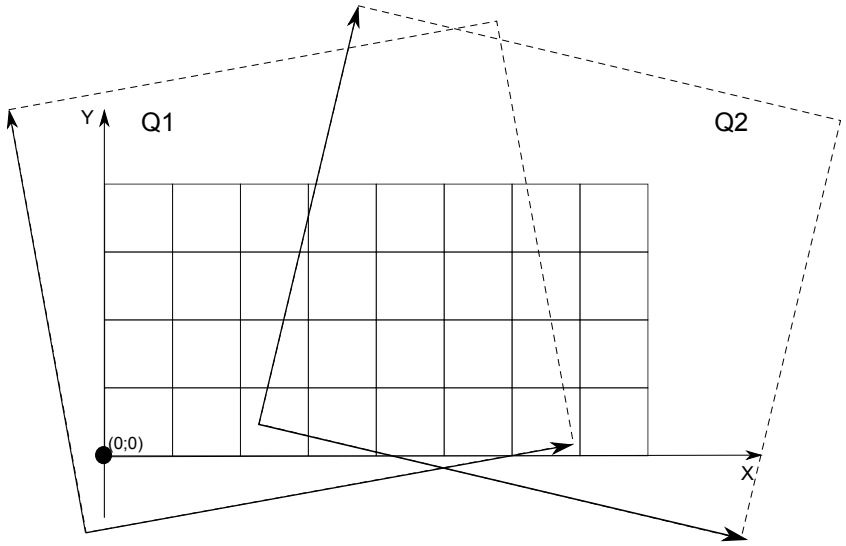

Figure 5.7: Overview of the environment from the Wii Remotes

The X-axis of the environment is prolonged from $Q1$ into $Q2$, but as the views are displaced, it cannot be described using a single function. It has be divided into two, one describing it in $Q1$ and one in $Q2$, as the Y-axis is orthogonal to the X-axis, it is also necessary to have two different Y-axis, one in $Q1$ and one in $Q2$. The localization process can be divided into two scopes, a global scope and a local scope.

The global scope considers the environment as a single entity, and is responsible for localization of an IR diode according to the fixed origin[2] of the environment. The local scope considers the environment as being divided into two, the beginning in $Q1$ and the end in $Q2$. Localization in the local scope will be according to each view, and not the fixed origin of the environment. The global scope is a high level interpretation of results from the local scope.

Local Scope

A function for a straight line in a Cartesian system is described using a numerical representation of it's in-/declination, α, and its intersection with the Y-axis, b, see Equation 5.1.

$$f(x) = \alpha \cdot x + b \tag{5.1}$$

A linear representation is the most appropriate as the edges of the environment are two straight lines, see Figure 5.3. To define such a function requires two readings made along the X-axis of the environment. As the X-axis is divided into two line segments, the first segment in $Q1$ and the final segment in $Q2$, four readings must be made, two in each view for each segment. Performing these readings will henceforth be referred to as calibration. The calibration is performed by moving an IR diode along the X-axis of the environment and performing readings using the Wii Remotes at given points. The IR diode must be moved manually and is therefore prone to uncertainty. To minimize the uncertainty and for practical reasons, two of the readings can be made in the same physical point, see Figure 5.8.

[2]The fixed origen of the environment is (0,0), see Figure 5.7.

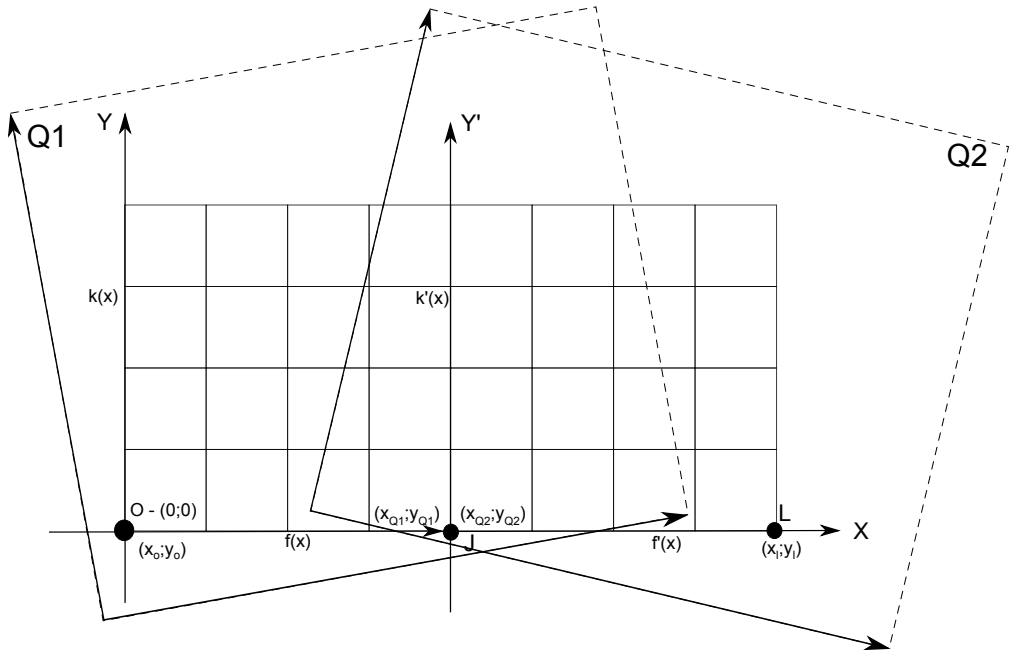

Figure 5.8: Calibration

There are three points marked in Figure 5.8, each depicting where the readings are to be made in the environment. Point O is the fixed origin of environment, but also the fixed origen when performing localzation in $Q1$. The physical point J lies in the view of both Wii Remotes and its position can be read using them both, reducing the number of times the IR diode has to be moved, and thereby reducing the uncertainty. At the point J, $J_{Q1} = (x_{Q1}, y_{Q1})$ will represent the reading made with the Wii Remote associated with $Q1$. $J_{Q2} = (x_{Q2}, y_{Q2})$ will represent the reading made with the second Wii Remote, and is the fixed origin of $Q2$. L is a random point on the X-axis in $Q2$, its position is of no crucial influence, but placing it at the far end of $Q2$ will increase the probability for a more precise depiction of the X-axis. The X- and Y-axis are depicted by respectively $f(x)$ and $k(x)$ in $Q1$, and in $Q2$ by $f'(x)$ and $k'(x)$.

Performing localization in each view is very similar, and it will only be illustrated using $Q1$ in the report. There is a comprehensive numerical example using both views in Appendix B.

Given a reading R in $Q1$ its position according to O can be defined as the orthogonal distance from the edges of the environment to R, see Figure 5.9.

The function $f(x) = \alpha_f \cdot x + b_f$ can be calculated using the O and J_{Q1}, which were established by performing the calibration.

$$
\begin{aligned}
f(x) &= \alpha_f \cdot x + b_f \\
\alpha_f &= \frac{\Delta Y}{\Delta X} \\
\alpha_f &= \frac{y_{Q1} - y_O}{x_{Q1} - x_O}
\end{aligned}
$$

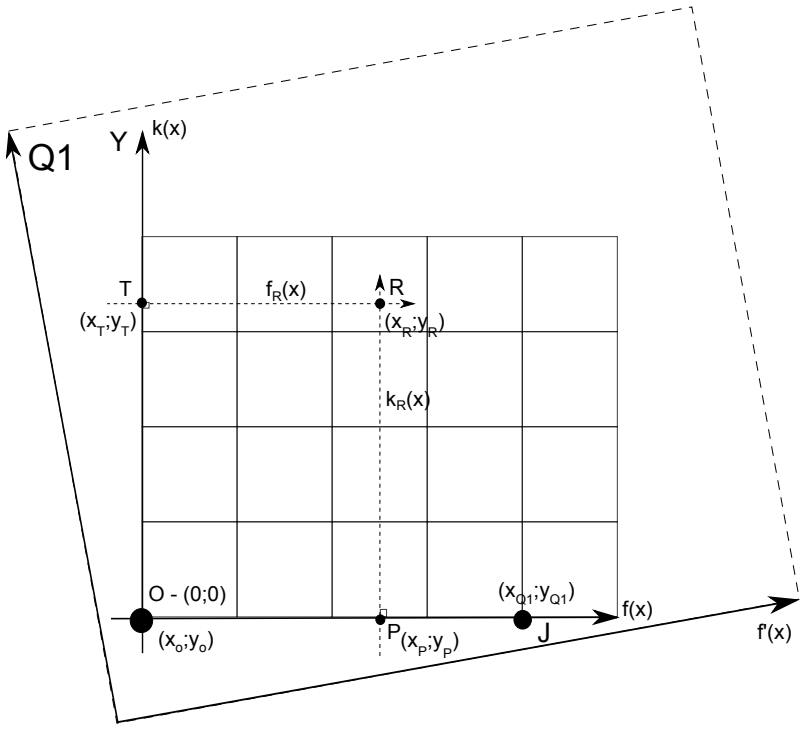

Figure 5.9: Localization of R in $Q1$

Given the α_f; b_f can be derived using either O or J_{Q1} as it will yield the same result, see Eq. 5.3

$$f(x_{_O}) = \alpha_f \cdot x_{_O} + b_f = y_{_O} \tag{5.2}$$
$$\Rightarrow b_f = y_{_O} - \alpha_f \cdot x_{_O} \tag{5.3}$$

The Y-axis is orthogonal to the X-axis, and $k(x) = \alpha_k \cdot x + b_k$ is therfore orthogonal to $f(x)$. Using this geometric aspect, $k(x)$ can be derived from the knowledge of $f(x)$.

The product of two orthogonal α values will yield -1.

$$\alpha_f \perp \alpha_k \Leftrightarrow \alpha_f \cdot \alpha_k = -1 \Leftrightarrow \alpha_k = \frac{-1}{\alpha_f}$$

Using this representation of α_k it is possible to derive b_k by the use of O.

$$k(x_{_O}) = \alpha_k \cdot x_{_O} + b_k = \frac{-1}{\alpha_f} \cdot x_{_O} + b_k = y_{_O}$$
$$b_k = y_{_O} - \frac{-1}{\alpha_f} \cdot x_{_O}$$

Having defined $f(x)$ and $k(x)$, the next step is to derive the orthogonal distance from each of the edges to R, see Figure 5.9. The point R represent a reading made by the Wii Remote, and has the coordinate $R = (x_{_R}, y_{_R})$. To derive the orthogonal distance, two orthogonal functions must be defined, one orthogonal to $f(x)$ and the other orthogonal to $k(x)$, which both goes through R. The distance from the intersection of $f(x)$ and its orthogonal function to R will be the y-coordinate, and

the distance from the intersection of $k(x)$ and its orthogonal function to R will be the x-coordinate. Having derived the point of intersection the respective distance can be calculated using Pythagoras.

Let $f_{R}(x) \perp k(x)$ and $k_{R}(x) \perp f(x)$.

$$\begin{aligned} \alpha_{f_R} \perp \alpha_k &\Leftrightarrow \alpha_{f_R} = \alpha_f \\ \alpha_{k_R} \perp \alpha_f &\Leftrightarrow \alpha_{k_R} = \alpha_k \end{aligned}$$

Similar to the previous calculations, $b_{f_R}(x)$ and $b_{k_R}(x)$ can be derived using $R = (x_{R}, y_{R})$.

$$\begin{aligned} f_{R}(x_{R}) &= y_{R} \\ \alpha_{f_R} \cdot x_{R} + b_{f_R} &= y_{R} \\ b_{R} &= y_{R} - \alpha_{f_R} \cdot x_{R} \\ k_{R}(x_{R}) &= y_{R} \\ \alpha_{k_R} \cdot x_{R} + b_{R} &= y_{R} \\ b_{R} &= y_{R} - \alpha_{k_R} \cdot x_{R} \end{aligned}$$

The functions $f_{R}(x)$ and $k_{R}(x)$ are the orthogonal functions to respectively $k(x)$ and $f(x)$, which goes through the point R. Using these, it is possible to calculate the intersection, with the edges and derive the orthogonal distance. The following calculations are divided with respect to each coordinate, so they cannot be misinterpreted.

X - coordinate
The x-coordinate is the distance between $|TR|$. T is found at the intersection of $k(x)$ and $f_{R}(x)$, see Figure 5.9.
T:

$$\begin{aligned} k(x) &= f_{R}(x) \\ \alpha_k \cdot x_{T} + b_k &= \alpha_{f_R} \cdot x_{T} + b_{f_R} \\ x_{T} \cdot (\alpha_k - \alpha_{f_R}) &= b_{f_R} - b_k \\ x_{T} &= \frac{b_{f_R} - b_k}{\alpha_k - \alpha_{f_R}} \end{aligned}$$

x_{T} is the x-coordinate for the point T, its respective y-coordinate can be calculated using either $k(x)$ or $f_{R}(x)$, as it will yield the same results.

$$\begin{aligned} k(x_{T}) &= \alpha_k \cdot x_{T} + b_k \\ &= y_{T} \end{aligned}$$

Having determined $T = (x_{T}, y_{T})$ it is now possible to establish the distance to R by the use of

Pythagoras.

$$|TR|^2 = (x_{_R} - x_{_T})^2 + (y_{_R} - y_{_T})^2$$
$$|TR| = \sqrt{(x_{_R} - x_{_T})^2 + (y_{_R} - y_{_T})^2}$$

Y - coordinate
The y-coordinate is the distance between $|PR|$. P is found at the intersection of $f(x)$ and $k_{_R}(x)$, see Figure 5.9.
P:

$$\begin{aligned} f(x) &= k_{_R}(x) \\ \alpha_f \cdot x_{_P} + b_f &= \alpha_{k_{_R}} \cdot x_{_P} + b_{k_{_R}} \\ x_{_P} \cdot (\alpha_f - \alpha_{k_{_R}}) &= b_{k_{_R}} - b_f \\ x_{_P} &= \frac{b_{k_{_R}} - b_f}{\alpha_f - \alpha_{k_{_R}}} \end{aligned}$$

$x_{_P}$ is the x-coordinate for the point P, its respective y-coordinate can be calculated using either $f(x)$ or $k_{_R}(x)$.

$$\begin{aligned} f(x_{_P}) &= \alpha_f \cdot x_{_P} + b_f \\ &= y_{_P} \end{aligned}$$

Having determined $P = (x_{_P}, y_{_P})$ it is now possible to establish the distance to R by the use of Pythagoras.

$$|PR|^2 = (x_{_R} - x_{_P})^2 + (y_{_R} - y_{_P})^2$$
$$|PR| = \sqrt{(x_{_R} - x_{_P})^2 + (y_{_R} - y_{_P})^2}$$

The coordinate of the point R is (|TR|,|PR|) according to the fixed origin of $Q1$. Most of the calculations presented here are basic, but they are repeated a number of times, which might become confusing. Please review the numerical example in Appendix B for a more comprehensive walkthrough.

Global Scope
The localization performed while in Local scope, does not necessarily correspond with the environment. The environment is divided into two views, and the localization performed in each view must be placed according to the fixed origin of the environment. The fixed origin of the environment is the point O, which is defined during calibration, see Figure 5.8. O and J_{Q1} are used to define the X-axis of view $Q1$, and in this view O acts as the fixed origin, thus making localization performed in $Q1$ match the environment. Localization performed in $Q2$ however does not match, as its fixed origin is J_{Q2}; this can however be converted by taken the geometric properties into consideration, see Figure 5.10.

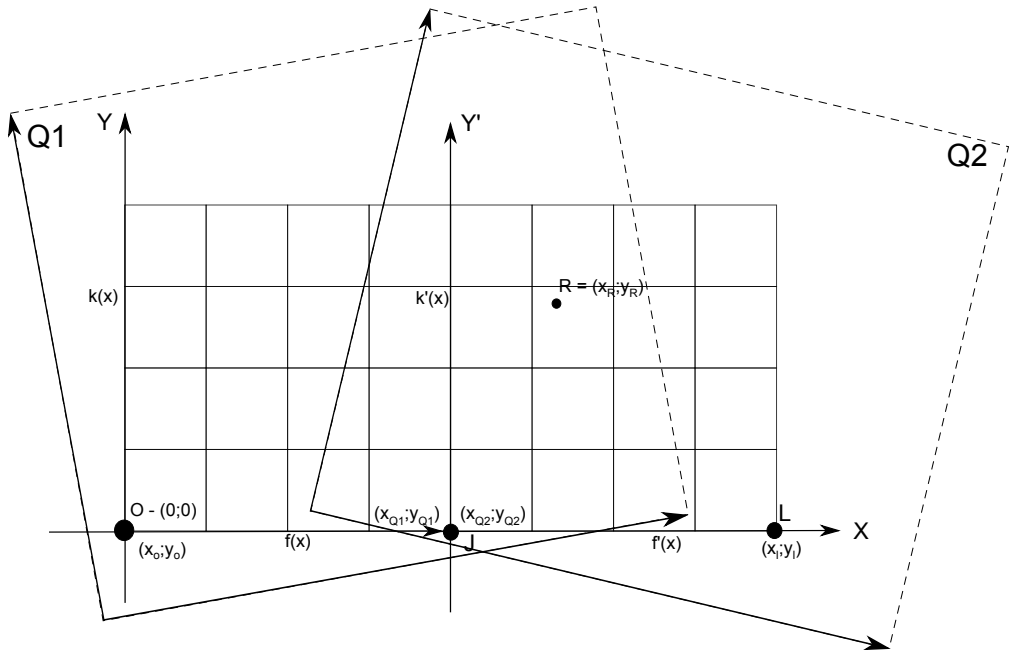

Figure 5.10: Converting readings from Q2 to fit the environment

Figure 5.10 illustrates a reading, $R = (x_{_R}, y_{_R})$, posistioned in $Q2$ according to its fixed origen J_{Q2}. The fixed origin of $Q2$ is shifted along the X-axis of the environment. The points J_{Q1} and J_{Q1} are physically the same point in the environment, therefore the displacement of J_{Q2} is $|OJ_{Q1}|$. To convert readings made in $Q2$ to fit the environment the displacement must be added to the position of R in $Q2$.

Y - coordinate
The view $Q2$ is shifted horizontally and not vertically according to O. The function $k'(x)$ which depicts the Y-axis of $Q2$, is parallel to $k(x)$, but is shifted along the X-axis. As there is no horizontal manipulations the y-coordinate retrieved from localizing in $Q2$ will match the y-coordinate of the environment.

X - coordinate
In contrast to the y-coordinate, the x-coordinate must be manipulated to fit the fixed origin of the environment. The point R has the coordinate set $R = (x_R; y_R)$ when positioned in $Q2$. The horizontal shift must be taken into account, by adding the shift to the x-coordinate. The position of R in the environment is $R = (x_R + |OJ_{Q1}|, y_R)$.

We are now able to perform localization of the robot in the environment in a very precise manner. The area covered by both Wii Remotes is approx. $2.50 \times 1.10[m^2]$, which is a great improvement compared to the Lego Mindstorm sensors. The results retrieved from the Wii Remote will be the observations. Observations are approximation of the current state, but the Wii Remote can deliver exact observations of the current state, which means it would be sufficient to model the robots using MDPs. To investigate the precision of the Wii Remote we have to perform an experiment.

Precision

To investigate the precision of the localization, we have performed an experiment which will establish eventual deviation. The numerical example presented in Appendix B was made using a drawn sketch and it validates the mathematical model for localization. Any deviation we might encounter in this experiment will very likely be due to the manual calibration.

A straight line, with the equation $f(x) = x$, is drawn in the environment and measurements are made along this function. All the measurements are expected to lie on the function, but due to the uncertainty of the calibration they will deviate.

Both views are calibrated in the same manner, and the mathematical conversion of the readings made in $Q2$ into a position in the environment does not impose any uncertainty. Performing the experiment in just $Q1$ will therefore be sufficient, to establish whether the calibration will impose a problem regarding the precision of localization. The results are depicted in Figure 5.11. The edges of the environment are mapped on the axis's and the linear function is depicted as a straight line, where the $+$ indicate the readings.

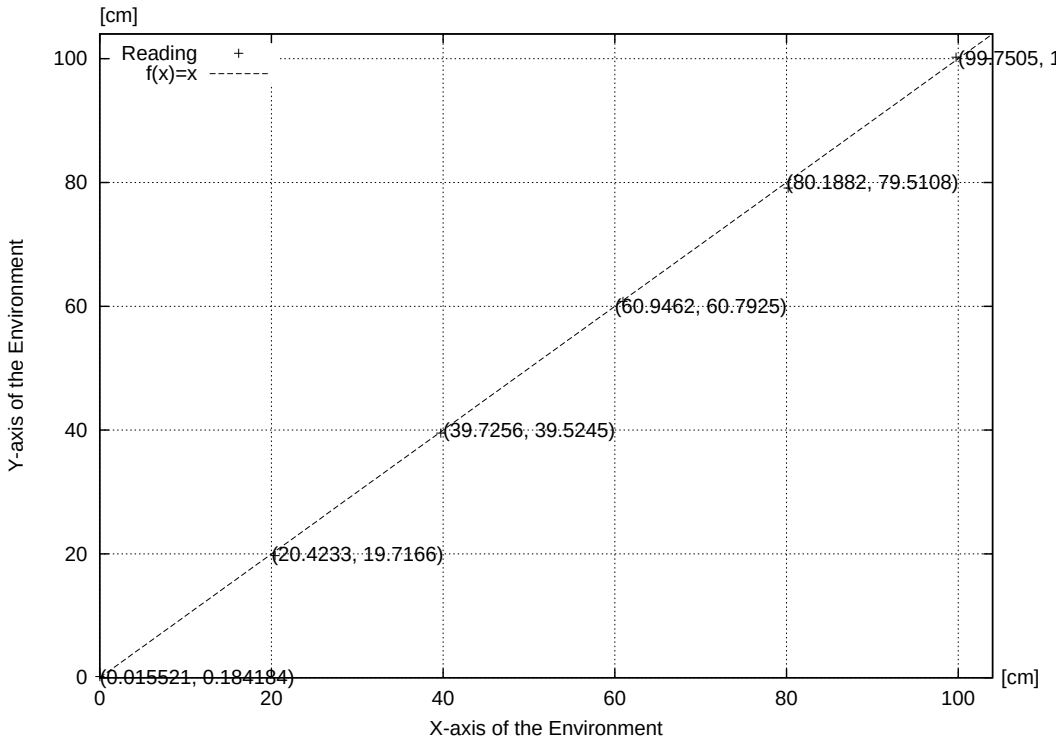

Figure 5.11: The measurements are very precise, indicating the calibration was performed well.

There has been made 6 readings along $f(x) = x$, and the deviation between the readings and the line are minimal. The greatest deviation is encountered at (60,60), where the x-coordinate of the reading almost deviate with 1 $[cm]$. The average deviation of the x-coordinates is 0.1749 $[cm]$ and 0.4106 $[cm]$ for the y-coordinate. The deviation is not spread evenly but accurse in a random pattern, which indicate that it might be due to the IR diode not being placed precisely

enough. In general the measurements are very precise, especially when considering the deviation which occurred using the sensors supplied by Lego. The Wii Remote can be applied to perform localization and it will provide almost exact positioning of the IR diode. The implementation is capable of observing the exact state, making I-POMDP redundant. We still wish to implement I-POMDP and have therefore implemented the 'ToFuzzyState' method which purposely returns the position with an error coefficient of +/- 3 $[cm]$[3].

Handover

When there are two or more robots in the environment, it is essential that they can be distinguished. If a robot moves from $Q1$ to $Q2$ or vice versa, it moves from one Wii remote to another. To retrieve its position the implementation must read from the Wii Remote currently fixated on it. In other words the implementation must dynamically adapt between reading from the different Wii Remotes, given the movement and location of the robot. This property is dubbed handover.

To understand how handover is to be implemented, we must review how the Wii Remote fixates and tracks an IR diode. Information presented in this section is achieved through experimenting and by reviewing the WiiYourSelf![4] library.

IR Tracking

The IR camera has a resolution of 1024x768$[pixel]$, and an IR diode will be registered as a cluster of pixels. The library converts the cluster to a single pixel coordinate which can be retrieved by the developer. There are API functions which allow the developer to get the size of the cluster, but these have no practical application regarding handover.

When an IR diode enters the view of the camera, it will automatically fixate upon the IR diode and begin tracking it. The camera is capable of tracking up to four IR diodes simultaneously. How the camera fixates on each diode and distinguishes them is still uncertain but it writes the pixel coordinate of each of the four IR diodes in 4 specific addresses on the memory of the Wii Remote. The IR diode that has the highest luminous intensity will be stored in the first address space, the second highest in the second address space and so forth. If there is only one IR diode, the second-, third- and forth address space will automatically be set to 0xff[5]. When an IR diode disappears from the camera's view the corresponding address space is 'set', and IR diodes which are not fixated upon will be assigned to that address space. In practical terms, the user can not always count on the IR diode with the highest luminosity to be assigned to the first address space. This will change over time and it will eventually become impossible to predict which IR diodes is assigned to which address space. It is only during initialization the address space is assigned according to luminosity.

The library reads from each of the address spaces, and stores the data from the first address space in an array at index number [0]. The second address space is stored the same array but at index number [1], and so forth. The array is accessible by the developer, and the pixel coordinate of e.g. the first IR diode can be obtained by reading at index [0] in the array. From a developers point of view, the IR diodes are simply enumerated, and assigned an integer value between [0-3], depending on the number of IR diodes in the view. Handover revolves around predicting which Wii Remote to read from and determining which index number to read the pixel coordinate from.

[3] We decided to use the same error coefficient as the Distance sensor.

[4] WiiYourSelf! is a library for interacting with the Wii Remote, see [gl.09]

[5] The address space is said to be 'set'. If the address space is 'set', nothing has been written to it

The environment is covered by two Wii Remotes, and a robot will move from one view to another and back again. If there only is one robot, the implementation can consequently read of the pixel coordinate stored at index number [0], as there only will be one IR diode. However if there are two or more diodes, the task becomes more complicated. The IR diodes are stripped of an IR bar for the Wii console, and we have no information regarding the wavelength or luminosity of the diodes. It is however reasonable to assume the wavelength and luminosity for all the IR diodes is the same. It is therefore not possible to predict which index number the IR diodes will be stored at during initialization.

The problem can be addressed by letting the robots move synchronously in a different pattern and thereby determine which index number is assigned to which IR diode. But this is a very complication solution, and we have decided to let the user input the IR diodes, by visual confirmation. This is only necessary when the implementation is initialized, tracking the robots afterwards is done automatically.

When several robots move from one view to another it is impossible to predict at what index number the new Wii Remote will store their pixel coordinate. The solution is to have an overlapping area, in which the implementation can track the diodes using both Wii Remote and identify which index number to read from, see Figure 5.12.

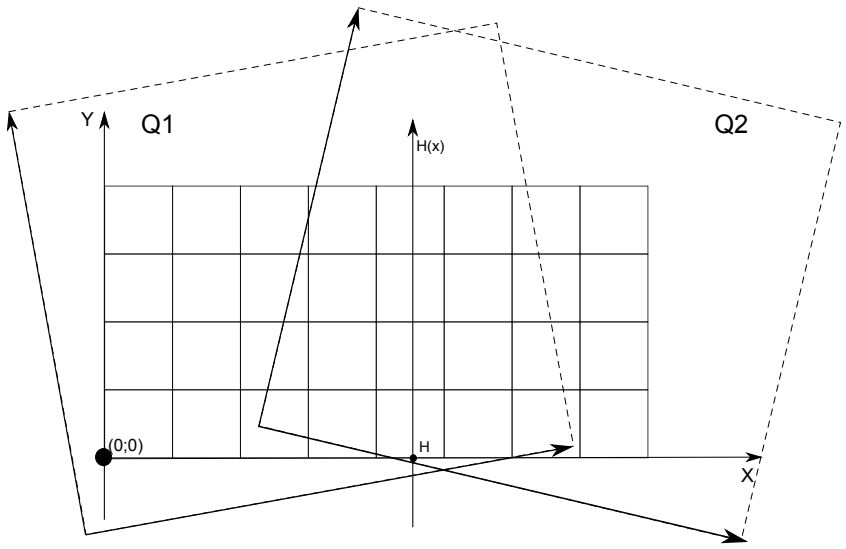

Figure 5.12: The environment seen from the Wii Remotes, where H is the handover point.

When a robot enters the overlapping area, both Wii Remotes will have fixated on its IR diode. The robot's position is already being monitored by one of the Wii Remotes, and the other Wii Remote will begin tracking it. By using localization, described earlier in this chapter, each of the pixel coordinates saved in the array of the second Wii Remote can be converted into a position in the environment. Comparing the robots position with each of the positions derived from the pixel coordinates of the second Wii Remote, the index number can be established. The implementation will then have to read from the second Wii Remote using the established index number.

There is however an issue, when utilizing this approach. The implementation has to shift reading from a one Wii Remote to another when a robot moves from one view to another. The robot can however still enter the overlapping area and then drive back to the view it came from. If the implementation has already shifted when the robot turns around and drives back, the implementation will not be able to track the robot.

The solution is to divide the overlapping area into two by inserting a virtual line, $H(x)$, see Figure 5.12. The area to the left of line is to be read by the Wii Remote associated with $Q1$ and the area to the right, with the Wii Remote associated with $Q2$. When the robot crosses the line, the implementation is to shift and begin reading from the appropriate Wii Remote. E.g. if the robot moves from $Q2$ and crosses the line, the implementation is to read using the Wii Remote associated with $Q1$. If it however turns back before crossing the line, the shift will not have been performed and nothing would happen. Does the robot cross the line and then drive back, the implementation will automatically shift back again, when the robot crosses the line. To implement this functionality the robot must have an 'awareness' of which Wii Remote it is currently reading from and its current position. The following high-level code depicts the concept of handover.

Foreach (robot)(

- **if**(robot->current-position **left of** *H(x)*[6])
  - **if**(robot->Wii-Remote == $Q1$->Wii-Remote)
    - The robot is being tracked using the correct Wii Remote.
  - **else**(
    - The robot is being tracking using the Wii Remote associated with $Q2$.
    - Establish the correct index number which has been assigned by the Wii Remote associated with $Q1$.
    - Perform the shift and begin reading from the Wii Remote associated with $Q1$.
)
- **if**(robot->current-position **right of** *H(x)*[7])
  - **if**(robot->Wii-Remote == $Q2$->Wii-Remote)
    - The robot is being tracked using the correct Wii Remote.
  - **else**(
    - The robot is being tracking using the Wii Remote associated with $Q1$.
    - Establish the correct index number which has been assigned by the Wii Remote associated with $Q2$.
    - Perform the shift and begin reading from the Wii Remote associated with $Q2$.
)

)

The idea behind handover is simple, but this code must run continuously to always update each instance of the robot, so it must be implemented as a thread. The point H has to be defined during calibration, so it is up to the user to find an appropriate middle in the overlapping area. This is due to the overlapping area will be different depending on the setup of the Wii Remotes, and a static implementation will therefore not be appropriate. In cooperation with our counselor it has been decided that two robot are sufficient to illustrate the theory, increasing the number of robots will only introduce higher computational complexity.

[6] See Figure 5.12
[7] See Figure 5.12

Chapter 6

Implementation

The robots are to move according to the rules of the 'Follow the Leader' game. This chapter will elaborate the implementation of the project. We will initially describe the scenario, that is, the rules of the game and how to model it. Hereafter the overall design will be presented to give an overview of the implementation. There are four aspects in the implementation: The GUI, Tracking, Core and finally the decision process. We will aim at presenting the implementation as either high-level code or as a description and avoid detailed walkthrough of the actual C++ implementation. But to understand this section it is necessary to have a basic understanding of the object-oriented paradigm and C/C++ syntax. We have chosen an evolutionary development method for the implementation, as this will accommodate any changes along the way.

6.1 Scenario

The scenario is a description of the 'Follow the Leader' game which is the foundation of the implementation. At the same time the scenario is a list of requirements, that describes the limits of the implementation. The list of requirements will be an abstract definition of problems posed by the scenario, where each element can have different levels of complexity.

Scenario: The environment is modeled as a 4×4 grid cell map, with a total of 16 cells where each cell represents a state. Following the definition from Section 3.3 the map will be approx. $80 \times 80[cm^2]$. The environment will be feature based, which means each individual state is distinguishable by observations. The movement from one cell to another is to be considered as the action. One robot acts as a 'Leader', and moves towards a designated goal in the environment. Observations made by the 'Leader' are very precise, so that it will move to the goal with very few errors. The 'Follower' is interested in following the 'Leader' to the goal. Observations made by the 'Follower' are very error prone. Because of this, the 'Leader' is better at finding the goal, which is additional incentive for the 'Follower' to mimic the actions of the 'Leader'.

From this description we can derive a number of requirements:

- The environment consists of 16 states, mapped as a 4×4 grid cells.
- There is a total of 4 actions in the environment; moving North, East, South and West[1]
- Each individual state is distinguishable by observation, which yield a total of 16 observations.
- Two robots performing actions in the environment.
- The first robot moves towards a goal using very precise observations.
- The second robot attempts to follow the first using observations prone to errors.

6.2 Design

The project is implemented as two applications. One responsible for tracking of the robots and performing the decision process, while the other is responsible for presenting the live data for the user. The overall design can be viewed in Figure 6.1.

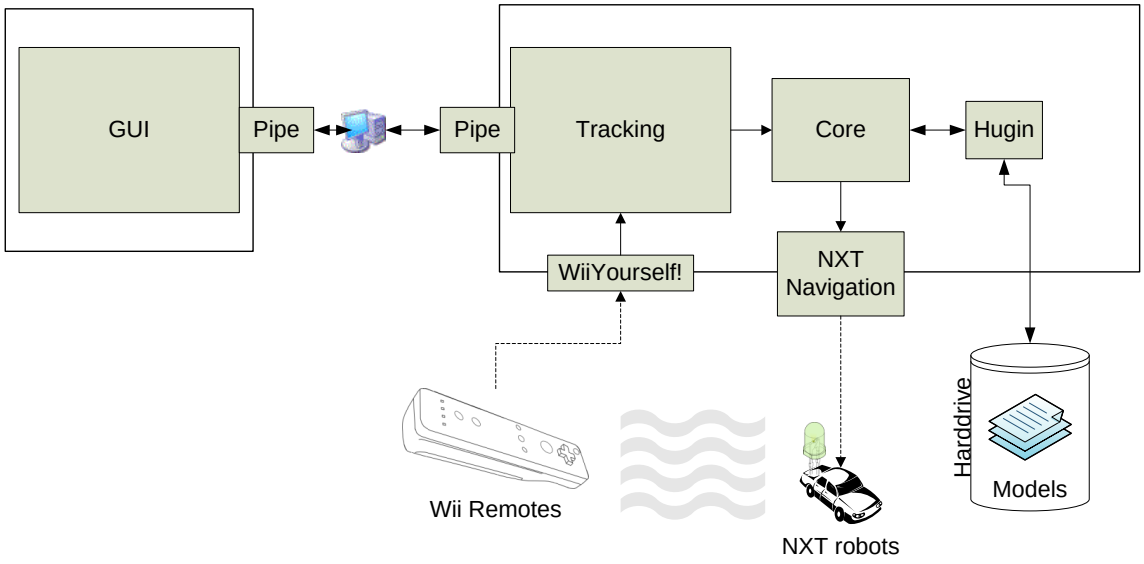

Figure 6.1: The design of the implementation

Figure 6.1 is an UML diagram of the implementation; the arrows indicate the direction of information exchange. The Graphical User Interface (GUI) has been separated from the tracking and decision logic ('Core' and 'Hugin' module). It is separated into an isolated application which is executed in parallel. The 'Pipe' is a mean of Inter Process Communication (IPC), which allows other processes to poll the results of localization. The GUI displays the data on screen, allowing the user to follow the movements of the 'NXT Robots'. The data can however be polled by any process which uses the correct API, providing the means of using the results of localization for other purposes than just to support decision making.

The 'Hugin' module calculates the policy from the underlying I-POMDP. The I-POMDP is implemented as an I-DID, which is stored on the local hard drive. The 'Main' module set the policy

[1]These represent the direction of movement.

in motion, by using the localization performed by the 'Tracking' module as observations to select the appropriate action. The action is parsed to the 'NXT Navigation' module, which implements velocity motion to control the actions of each NXT robot (the car). Each action is predefined and implemented as different functions, executing these will result in the NXT robot performing the associated action. The NXT bricks can be connected using either Bluetooth or USB, in Figure 6.1 the dotted line indicates a Bluetooth connection.

Each NXT robot has an IR diode mounted on them, and the pixel coordinate of each diode is read using the Wii Remotes. The data from the Wii Remotes are polled by the WiiYourSelf! library using a Bluetooth connection and stored locally[2]. The data is parsed by the library to the 'Tracking' module which performs localization of the pixel coordinates, according to the calibration.

The application containing the 'Tracking' module is to be initialized first. When executing the application the very first task is to perform the calibration of the Wii Remotes, which must be established before localization can be performed.

6.3 Decision

In this section we describe the implementation of the decision process. As described in the scenario[3], the 'Leader' receives almost exact observations and its task is to arrive at the goal in the least number of steps. The 'Follower' will mimic the actions of the 'Leader', and must therefore predict the actions of the 'Leader'. The 'Leader' will be modeled as a DID, and the 'Follower' as an I-DID, which contains the model of the 'Leader'

The decision process revolves around generating the policy trees from the I-DID and DID. There is however a number of problems that needs to be solved. The first issue is to select an appropriate data structure to store the generated policy tree. Using a tree structure to store the policy tree is unfortunate. The size of the tree structure can easily become too big to manage, as the size increases exponentially in the number of observations.

Another issue is modeling of the environment. A big model[4] will inevitable have a large observation set, making the policy tree very big, and thus affect the number of possible time horizons. This section will address these problems and present a solution.

6.3.1 Data structure

The data structure is used for storing the generated policy tree. The size of the tree grows exponentially in the number of observations for each time horizons. The number of nodes at a given time horizon, can be calculated as the number of observations to the power of the time horizon. In an environment with e.g. 8 observations, the number of nodes at time horizon 3 is $8^3 = 512$nodes. Table 6.1 illustrates how the number of nodes increase exponentially for each time horizon.

We wish to maximize the number of time horizons, and the exponential nature of trees is therefore undesirable. Thus we need to implement a more compact data structure, that is able to store the data, but without the exponential increase in nodes.

[2]Local in this context means the RAM.

[3]See Section 6.1.

[4]A model with a large state space.

Timehorizon	Number of nodes
0	1
1	8
2	64
3	512
4	4.096
5	32.768

Table 6.1: Number of nodes in a tree data structure for scenario with 8 observations.

Timehorizon	Number of nodes (tree)	Number of nodes (compact)
0	1	4
1	8	4
2	64	4
3	512	4
4	4.096	4
5	32.768	4

Table 6.2: Number of nodes at a given time horizon in a scenario with 8 observations and 4 actions.

In [SZ07], Seuken and Zilberstein illustrate how a policy tree can be represented in linear space. The concept behind their implementation is derived from the manner policy trees are constructed. A policy tree consists of actions and observations, where each node represents an action and the links between nodes represent observations. On each level of the policy tree, the maximum number of different nodes is equal to the number of actions. This knowledge can be utilized to construct a compact data structure, which has the same number of nodes on each level, but encodes an entire policy tree. Figure 6.2 illustrates the compact data structure in comparison to a normal tree structure.

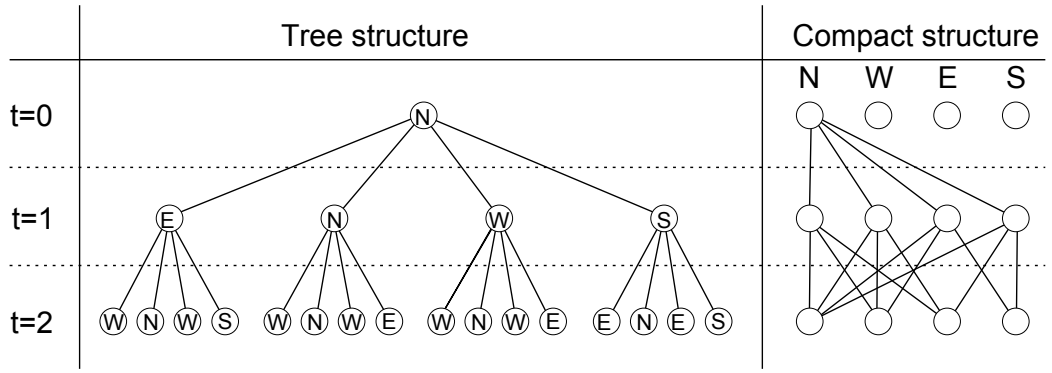

Figure 6.2: Comparison of a tree data structure and a compact data structure.

The compact data structure has some obvious advantages over a traditional tree structure. Notice the dramatically reduced space requirement. While the size of a traditional tree structure increases exponentially, the compact structure only increases in a linear fashion. Table 6.2 is a expansion of table 6.1, which shows the number of nodes at a given time horizon. Furthermore the compact data structure is simple in its construction, which makes traversing it simple.

The compact data structure is not without flaw though. It is possible to encode a policy tree in the compact data structure, but this is a one way process. It is not possible to reverse the process and create the policy tree from the data contained in the compact data structure. This is due to the simple nature of the model discards a lot of information when the policy tree is encoded. The discarded

Timehorizon	Model node CPT size
0	16
1	256
2	4.096
3	65.536
4	1.048.576
5	16.777.216

Table 6.3: The Mod node CPT size, when the number of initial models is 1 and the number of observations is 16.

information is mainly surrounding the parents of different nodes. It may not seem relevant, but if one tries to recreate the original policy tree, there can be made relations between nodes which should not occur. Because the information about parents is needed in the implementation, we need to argument the compact data structure with information about the parents of the nodes.

The resulting data structure is the compact data structure, as described, argumented with parent information. The addition of the parent information means that the compact data structure loses some of its advantage over the regular tree structure, but the remaining difference is still in the compact data structures favor.

6.3.2 Models

As stated earlier the 'Leader' will be implemented as a DID and the 'Follower' as an I-DID. Hugin will be used to model the implementation and to derive the optimal policy. There are however some inherent problems when creating the models, most notably is the size of the conditional probability tables of the Mod nodes in the I-DID models. The size of the CPT of the Mod node increases exponentially for each time horizon. A large CPT is difficult to compute, and will therefore limit the number of possible time horizons.

The environment is a 4×4 grid cell, with 16 states, 16 observations and 4 actions. Given the number of initial models is 1, the size of the CPT of the Mod node will be 16 at the first time horizon, and 256 for the next time horizon, etc.. Table 6.3 illustrates the CPT size of the Mod node, where the number of initial models is 1.

It is not possible to avoid the exponential nature of the Mod node CPT size, but by reducing the number of states in the model, we will reduce the number of observations and thus reduce the speed at which the CPT size grows. The idea is to decompose the problem into smaller, more manageable problems. The environment is represented as a 4x4 grid, by decomposing it into 4 smaller parts, each being a 2x2 grid, we will have reduced the state space and observation set. The decomposition is illustrated on figure 6.3.

The result of the decomposition is an environment with 4 states, 4 possible observations and 4 possible actions. This reduction in the number of states and observations, is reflected in the size of the CPT of the Mod node. Table 6.4 illustrates the Mod node CPT size for the decomposed problem, where the number of initial models is 1. The reduction in CPT size is exactly 75%.

Both of the previous examples have 1 initial model. This is only to illustrate the speed of which the CPT size increases. In practice the number of initial models should be as high as possible, to take more cases into account. Additional initial models multiplies the size of the Mod node CPTs. In

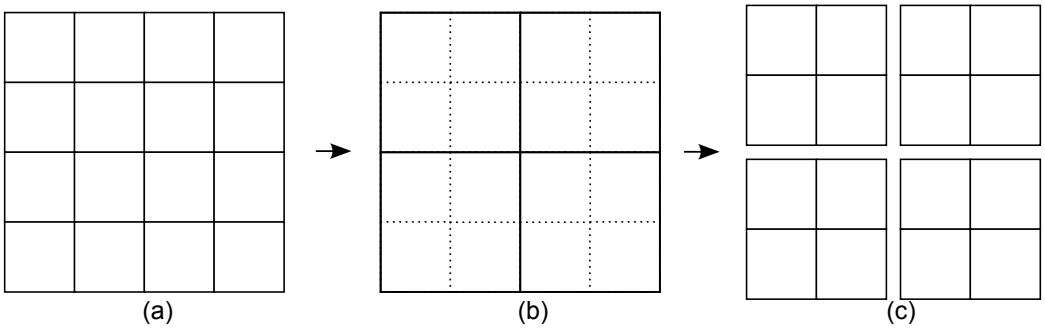

Figure 6.3: Decomposition of the problem into smaller problems.

Timehorizon	Model node CPT size
0	4
1	64
2	1.024
3	16.384
4	262.144
5	4.194.304

Table 6.4: The Mod node CPT size, when the number of initial models is 1 and the number of observations is 4.

such cases the decomposed problem has a big advantage over the full model, because it can contain 4 times the number of initial models in the same space.

Decomposing the problem is however not without drawbacks. The decomposition of the problem affects the number of DID and I-DID models that needs to be created. In the scenario there are 2 robots, a 'Leader' modeled by a DID and a follower modeled by and I-DID. Due to the decomposition, the models that represent the full problem must to be split into different models covering each of the smaller partial problems.

Four different DID models are needed to model the 'Leader'. The goal can only be in one state of the 4×4 grid cell environment and each model will only represent a part of the entire environment, see Figure 6.3 (c). Each model will map an action that will guide the 'Leader' closer to the goal. The implementations of the DIDs are stored in robot1.net, robot2.net, robot3.net and robot4.net, which can be found on the enclosed CD in folder Hugin_Models.

A rather naive approach is taken to model the 'Follower', as it is modeled using two I-DID models. There is a top-level I-DID and a low-level I-DID. The top-level I-DID is to be applied when the robots are in each part of the decomposed environment, whereas the low-level is to be applied when they are in the same decomposed environment. Using the two I-DID models, it is possible to generate 8 different I-DID models based on the DIDs of the 'Leader'. Both I-DID models will suggest which action to perform, in order to get close to the 'Leader', regardless of whether the 'Leader' is moving toward the goal or not. The implementation of the I-DIDs are stored in multiagent_robot0.net and multiagent_robot1.net, which also be found on the enclosed CD in folder Hugin_Models.

A1	North			
S1	State 1	State 2	State 3	State 4
State 1	0.05	0.05	0.05	0.05
State 2	0.05	0.05	0.05	0.05
State 3	0.85	0.05	0.85	0.05
State 4	0.05	0.85	0.05	0.85

Table 6.5: Snippet from the DID and low level I-DID transition function CPTs.

Transition function

The transition function will be elaborated in this section. We will look into how it is made, and explain special cases. As previously stated the map is feature based, and each cell has therefore been assigned a number. This numbering is illustrated on figure 6.4.

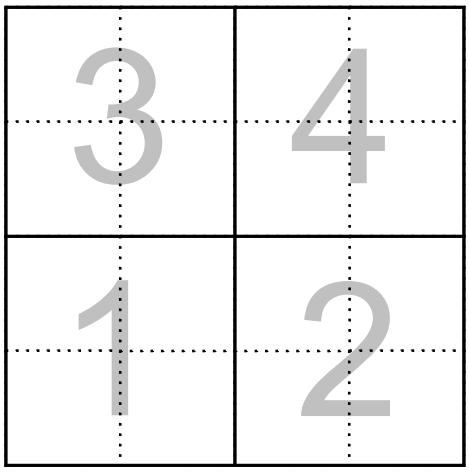

Figure 6.4: Numbering of states in the models.

In section 4.4.2 a series of tests is performed in order to determine the most efficient motion control for the robots. Selecting velocity motion directly affects the transition function, and we can use the test data to estimate a fairly precise transition function. The test results illustrates, that the worst test case of the velocity motion had a deviation of 0.5 $[cm]$ on a 20 $[cm]$ start-stop test (this is a deviation of 2.5%). A reliable implementation of the movement of the robot is very useful, as is helps reducing the overall uncertainty. But despite the precision of the implementation, it cannot prevent the deviation from accumulating an error, leading to eventual errors. We design the transition function to be a bit more erroneous than the test samples suggests, as experience dictate that the NXT is somewhat unreliable.

Table 6.5 contains a snippet of the CPT for the chosen transition function. It is quite clear, that the transition function is close to being deterministic. That is because of the very precise movement function. This transition function will be used in the DID models and the low-level I-DID model.

A special cases of transition function exists within the I-DID, where the transition function is different. Due to the decomposition of the environment described in section 6.3.2, we made 2 different I-DID models. The difference between these models is the transition function. Figure 6.3 illustrates the decomposition, but it also shows the 2 different I-DID models. The top-level I-DID consider the environment as illustrated in Figure 6.3 (b) that is a 2x2 grid where each cell contains a 2x2 grid. The low-level I-DID considers the environment as illustrated in Figure 6.3 (C).

A_1	North			
A1	North			
S1	State 1	State 2	State 3	State 4
State 1	0.53	0.01	0.001	0.001
State 2	0.01	0.53	0.001	0.001
State 3	0.45	0.01	0.948	0.05
State 4	0.01	0.45	0.05	0.948

Table 6.6: Snippet from the top level I-DID transition function CPT.

Given that the two robots are in the same decomposed part of the environment, the low-level I-DID will be used. The low-level I-DID will have the same transition function as the DID. The top-level I-DID differs in its transition function. When the robots are in each part of the decomposed environment, the top-level I-DID will be used. But as each decomposed part is a 2×2 grid cell, see Figure 6.3 (b), the chances of moving from one decomposed part to another, is low.

An example of this is, e.g. if the robot is in the top left cell of the top left cell in the top-level I-DID model. If the robot moves east, it will still be in the same state. But if the robot is in the top right cell of the same cell, and it moves east, it will leave the state. This example is illustrated on figure 6.5. As indicated by the test cases in Section 4.4.2, the robot does not always drive completely straight, and the chances are therefore slightly higher that it will stay in the same state in the top-level I-DID. The transition function for the top-level I-DID is illustrated in table 6.6.

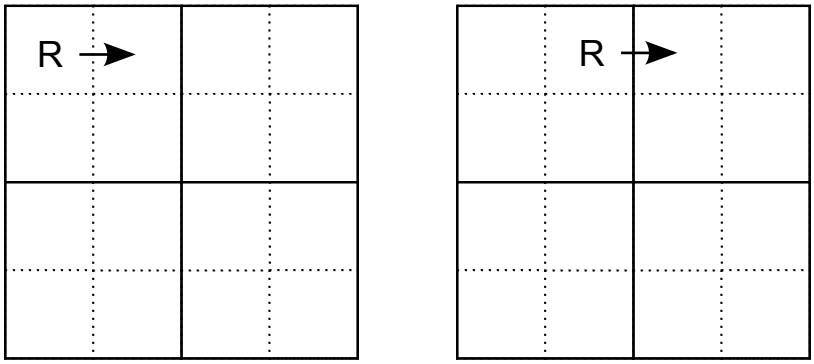

Figure 6.5: Special case in the transition function in the top level I-DID.

Observation function

The observation function is a central part of the scenario. The 'Leader' will receive precise observations, in that way it can navigate to the goal with little failure. Whereas the 'Follower' will receive less precise observations. The difference in observation precision serves as one of the incentives for the 'Follower' to mimic the actions of the 'Leader', as it is probably better at finding the goal. We use the Wii Remotes for positioning of the robots. The Wii Remotes are described in chapter 5, and the test results regarding the precision of them is described in section 5.2.2. The average deviation of the readings is 0.1749 $[cm]$ on the x-axis and 0.4106 $[cm]$ on the y-axis. In general it is a deviation of $+/-0.5[cm]$. This is sufficient for the 'Leader', whereas it is far too precise for the 'Follower'. Because of this we had to add some amount of error to the readings of the 'Follower'. Listing 6.1 contains a code snippet of the function `ToFuzzyState`, which adds error to the observations of the 'Follower'. The amount of error added is +/- 3 $[cm]$, for each coordinate.

```
1  int ToFuzzyState(Robot *r){
2     double x,y;
3     int xsign, ysign, xerror, yerror;
4     int length_Vertice = 20;
5     //seed the random generator.
6     srand((unsigned)time(NULL));
7     if(rand()%2){xsign=1;}else {xsign=-1;}
8     if(rand()%2){ysign=1;}else {ysign=-1;}
9     xerror = rand()%3*xsign;
10    yerror = rand()%3*ysign;
11    x = r->getPosistion()->x+xerror+20;
12    if(x<0){x=0;}else{x=x/length_Vertice;}
13    y = r->getPosistion()->y+yerror+20;
14    if(y<0){y=0;}else{y=y/length_Vertice;}
15    return static_cast<int> (x+(y-1)*4);
16 }
```

Listing 6.1: The error creation function `ToFuzzyState`.

6.4 Localization

To gain access to the Wii Remote the WiiYourself! library is used. It is written by 'gl.tter'[5] and can be downloaded freely from 'http://wiiyourself.gl.tter.org/'.[gl.09]
This section looks into the library and how the data is retrieved from the Wii Remote. The library is not documented, and we have tried back track as much information as possible, but certain parts to the implementation are still unclear for us. The source code can be found on the enclosed CD in folder *WiiYourself!_1.01a*.

6.4.1 WiiYourself!

Only the source code is included in the download package, the library must be compiled manually. To compile the library, it is necessary to understand how it interacts with the Wii Remote.

The library is a device driver for the Wii Remotes, and it will automatically goes through the Bluetooth stack in Windows searching for Wii Remotes. Each Wii Remote will be assigned an integer number[6], and the library supports interaction of up to 8 Wii Remotes simultaneously. The individual Wii Remotes can be accessed by associating an instance of the *class* `wiimote`[7] with the number assigned to that specific Wii Remote.

The Bluetooth stack can however deviate depending on the underlying hardware, and the library does not support all hardware configurations. We have compiled the project on three different computers, with different hardware specifications and have so far not encountered any problems.

To compile the library, the Windows Driver Development Kit (WinDDK) is necessary. The kit is an integrated driver development system that allows the user to create device drivers. The library uses certain headers and libraries from WinDDK to gain access to the Bluetooth stack through the OS. The WinDDK is specific for each kernel, and the compiled library is therefore OS specific. The

[5] The developer's name is not stated, neither in the library or the homepage.
[6] Beginning with 1.
[7] See l. 64 in wiimote.h

path [*WinDDK-install-directory*]/inc/wxp must be added to the path variable and [*WinDDK-install-directory*]/lib/wxp/i386 to the library path.

We have not been able to determine the calls made by the library to the underlying system. But they are not part of the Common Language Runtime (CLR), and the library must be compiled directly into native code, around the CLR.

Common Language Runtime

The CLR is part of .NET initiative and used to unify programs, making them executable across the series of Windows platforms. Its functionality is very similar to the Java Virtual Machine (JVM). Code, whether it is written in C# or C++, is compiled into the Common Intermediary Language (CIL). The CLR is an intermediary level between CIL and native code, which 'adapts' the CIL code to the specific OS, e.g. applying the look-and-feel of the GUI to match the OS it is executed on, see Figure 6.6. This architecture provides the means of distributing most application seamlessly between any Windows distribution, and having it run similarly on each of them.

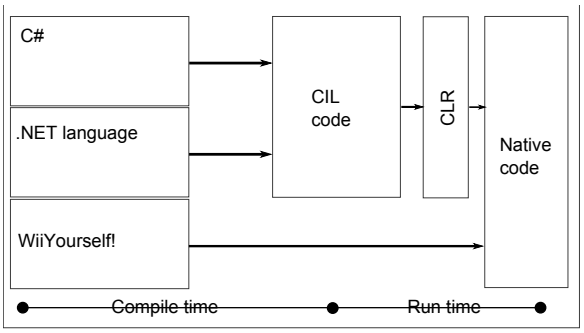

Figure 6.6: The structure of Common Language Runtime(CLR)

The WiiYourself! library requires functionality which is not supported by the CLR, and must be compiled directly to native code. This does not impose a problem, but GUI development is however affected by this. The drag-and-drop GUI development is based on the CLR, and without it, every aspect of e.g. a window must be defined manually by the developer. This is a time consuming and complicated process. This was one of the main arguments for separating the application in two. By separation of the GUI, it is possible to utilize the efficiency of the drag-and-drop development, while still using the library.

The compiled library can be added to any C++ project and used to communicate with the Wii Remotes. The library has a great deal of functionality which are irrelevant for the project. We will henceforth only consider those which are used retrieve the pixel coordinates from the Wii Remotes.

Data polling

The Wii Remote does not automatically send updated data, it must be polled from it. The library performs read and write operations at a specific frequency. When reading this section please confer with Appendix C.1 and C.2 to clarify argument type and return types of the methods presented in this section.

To retrieve the IR information, an instance of the *class* `wiimote` must be made for each Wii

Remote[8]. Each instance must be associated with a specific Wii Remote by executing the *bool* `Connect(...)` method. We will illustrate how to establish a connection later on in this chapter.

Given a successful connection has been established between an instance of *class* `wiimote` and the specified Wii Remote, the data can be polled from the Wii Remote. As stated earlier, the read operation is not continuous, but an action which reads the whole memory, that is repeated at a specific frequency. The data gathered from each read operation is dubbed a *'report'*. The user can not influence what is read form the Wii Remote, but can determine which information is extracted from the *'report'*.

Each instance of the *class* `wiimote` has to execute the *void* `SetReportType(...)`[9] method to establish which information is to be extracted from the report. The library consists of different parsers, and executing *void* `SetReportType(...)` defines which of these parsers to use, see Figure 6.7.

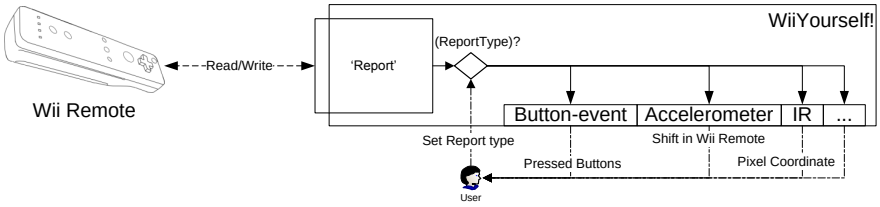

Figure 6.7: The report type, set by the user, determines which parser to use, which extract the data from the report and provides them for the user.

The report type *IN_BUTTONS_ACCEL_IR*[10] retrieves data regarding button-events, the accelerometers and finally the IR camera. We utilize button-events in the implementation to turn off the Wii Remote, but accelerometers are of no use. We could do without the information regarding of both button-events and accelerometers, but as we will see in a brief moment, the respective report type provides the most detailed encoding of the pixel coordinates.

IR parser

The implementation of the IR parser can be found in Appendix C.3, but this is only an excerpt, to review the complete code please review wiimote.cpp in the *WiiYourself!_1.01a* folder. The parser is executed with the *'report'* as argument.

The IR parser is a *switch case* with three cases. The case is selected depending which IR mode is set by the user. The IR mode is another report type, but is inherited from *struct* `wiimote_state`[11]. In *struct* `wiimote_state` there are defined four report types *OFF*, *BASIC*, *EXTENDED* and *Full*. *OFF* is self explanatory and *Full* has not been implemented yet. The difference between *BASIC* and *EXTENDED* is the number of bytes used to encode an IR diode.

BASIC use 5 bytes to represent two IR diodes, and the pixel coordinates are read two at a time, review l. 18-21 in Appendix C.3. The 5 byte representation is used to derive the offset, henceforth referred to as *offs* in the code. The pixel coordinates of the IR diodes are stored in different address

[8]There are 2 Wii Remotes in the setup.
[9]For details regarding the method see Appendix C.1.
[10]A global variable defined in wiimote.h l. 78
[11]See the inheritance in Appendix C.1, and *struct* wiimote_state in Appendix C.2.

space on the Wii Remote and the content of each address space is copied into different indexes in the *'report'*. The offset points to those indexes.

The report type stated eelier[12] also sets the IR mode to *EXTENDED*[13], so each IR diode will be encoded using 3 bytes, review l. 43 in Appendix C.3. The structure of *buff* is not very clear, and we have not been able to derive how the information surrounding each IR diode is represented in *buff*. Establishing whether an IR diode is visible is not only done by checking at the index matching the *offs* but also at index *offs +1* and *offs +2*, review l. 45. Furthermore we have not been able to determine why two indexes are accessed to derive both the x- and the y- coordinate on l. 48-49. Even though we do not understand the operations, they do however retrieve the correct coordinates of each IR diode.

When the pixel coordinates are derived they are converted into a numerical representation between [0-1] for both the x- and the y- coordinate, review l. 50-51. This however alters the unit scale so the edges have different unit measure. To solve this problem we have altered the divisor on l. 51 to *wiimote_state::ir::MAX_RAW_X*[14]. The divisor can however be deleted both on l. 50 and l. 51, during the implementation we were afraid that storing a value larger than 1 would cause an error in the library, in case the pixel coordinates were used in respect with other functionality. The subtraction performed on l. 50 alters the intersection of the edges to (1.F[15],0). Deleting the subtraction will restore the intersection to its natural position.

The alterations illustrated above have to be made to the library, otherwise the results of the localization will be erroneous. The implementation of the IR parser indicates that it is not intended to perform localization. The coordinates are only updated if the IR diode is visible, review l. 47 in Appendix refsec:IRParser. If the library loses its fixation of the IR diode, it will return the last known pixel coordinates, unless another IR diode can be fixated upon. We did not want to alter the library, unless it was absolutely necessary, but it can be modified to provide better means of localization.

6.4.2 Tracking

Most of the implementation is very straight forward and a basic understanding of C++ is sufficient to read the code. We will therefore mainly focus on the revolving the Wii Remotes. Code snippets presented here are from files that lie in the *project*-folder in the root of the implementation, see enclosed CD in folder *project*.

Wii Remote

Given that the library has been successfully compiled and added to a project, its classes and methods can be accessed. There are two functions which retrieve the data from the Wii Remotes, each function is associated with its specific Wii Remote. The functions are very similar so we will only present one of them.

```
1 wiimote mainREMOTE, extREMOTE;
2 ...
3 //setup of the mainREMOTE
4 void MainWII(void *){
```

[12] *IN_BUTTONS_ACCEL_IR*.

[13] Review l. 624-625 in wiimote.cpp

[14] This variable holds the integer value 1024, which represent the resolution of the X-edge of the IR camera.

[15] A Hex value for 31

```
5
6     mainREMOTE.
7        Connect(1);
8
9     mainREMOTE.
10       SetReportType(wiimote::IN_BUTTONS_ACCEL_IR);
11
12    // set the nr. 1 LED light on, to indicate its the main remote.
13    mainREMOTE.
14       SetLEDs(0x01);
15
16    wiimote::ir::dot &dot = mainREMOTE.IR.Dot[0];
17    wiimote::ir::dot &dot1 = mainREMOTE.IR.Dot[1];
18
19    while(!mainREMOTE.Button.Home()){
20
21       mainDotX = dot.X;
22       mainDotY = dot.Y;
23       mainDot1X   = dot1.X;
24       mainDot1Y   = dot1.Y;
25
26       while(mainREMOTE.RefreshState() == NO_CHANGE){Sleep(500);}
27    }
28    return ;
29 }
```

Listing 6.2: The implementation of the Wii Remote

The first line creates two instances of the *class* `wiimote`, the first instance is 'mainREMOTE' and the second 'extREMOTE'. The setup of the 'mainREMOTE' will be elaborated.

When working with the WiiYourself! library it becomes very clear, that the project still is in its early stages. Certain functionalities does not work, which has had an influence on the implementation. When an application starts reading from the Wii Remote, the link must not be broken. If the link is broken e.g. the implementation stops reading for a period, the program can not gain access to the Wii Remote any more. The only solution is to shut the program down, and start it over. If the Wii Remote has to be accessed several times, a thread must be made which continuously reads from the Wii Remote.

The method *void* `MainWII(`*void* `*)` is intended to be executed as a thread, which continuously read the pixel coordinates from the Wii Remote.

The connection between 'mainREMOTE' and the Wii Remote is made in l. 6, were it is set to establish a connection to the first Wii Remote the library encounters in the Bluetooth stack. Even though the method takes two arguments, the second argument is not implemented and is therefore by default set to false[16]. 'extREMOTE' will in contrast to 'mainREMOTE' use argument '2', to communicate with the second Wii Remote in the Bluetooth stack.

The report type is set on l. 9, and on l. 13 the Wii Remote is set to turn the LED nr. 1 on, to provide the means of visually confirming which Wii Remote is nr. 1 and nr. 2.

There are two instances of type *wiimote::ir::dot* on l. 16 and l. 17. The variable '*dot*' and '*dot1*' will be used to store the data of the first[17] and second[18] IR diode in the environment. Each time the library polls new data the IR parser extracts the information of each IR diode and store them in *dot* and *dot1*. The *wiimote::ir::dot* is a structure which holds all the information regarding an

[16] Review Appendix C.1 for further detail.
[17] mainREMOTE.IR.Dot[0]
[18] mainREMOTE.IR.Dot[1]

IR diode[19], we are however only interested in the x- and y- coordinates. The function is an eternal loop[20], that exits when the 'Home' button on the associated Wii Remote is pressed. To extract the pixel coordinates, four global variables are used, review l. 21-24. The pixel coordinates of each IR diode is stored in the global variables, which can be accessed from outside the thread, and localization can be performed. For future development, the use of global variables should be minimized.

The other parts of the tracking module are very straight forward, the handover function was implemented almost according to the high-level code presented in Section 5.2.2. It is however implemented as a function which is executed with each robot as argument. The function is executed continuously by a secondary function, with each robot as argument. Given the user wishes to introduce another robot in the environment, only the secondary function needs to be edited. The secondary function, *void* `Update(Robot)`, can be reviewed in Navigate.cpp, in the project folder, which lies in the root of the implementation.

During the implementation of the handover, an unforeseen problem did arise. During handover, the current position of the Robot is compared to the positions of the fixations belonging to the new Wii Remote. Each Wii Remote operates according to its own Cartesian system, and the bottom edge of the environment was to represent the X-axis of both systems. But due to the uncertainty of calibration the x-axis of the environment was not straight but became somewhat crooked. As the Cartesian systems were not aligned, localization performed on the same IR diode in the overlapping area would return different coordinates. This made it difficult to establish which fixation of the new Wii Remote, was fixated on the IR diode. We solved this by using Pythagoras to compute the distance between the current position and the fixations. The fixation which is closest will be correct one. The handover function[21] can be reviewed in Navigate.cpp, in the project folder, which lies in the root of the implementation.

6.5 Core

In this section we briefly describe the Core module of the application. Figure 6.1 is an UML diagram of the entire program. It illustrates the Core as the center of the program, that utilizes the other modules in order to make the robot move about correctly. It is the logic behind making the robot move 'correctly' that we will describe in this section.

Listing C.1 contains pseudo code of the most important aspect of the Core module, that is the logic that combines the three other modules of the program[22]. The code may seem redundant, but that is because, it is for both the 'Leader' and for the 'Follower'. The critical difference is in the creation of the model and the calculation of policy trees, otherwise it is the same. First off, the 'Leader' is supposed to move independently toward its goal. As described in section 6.3.2, the problem has been decomposed, and different models have been made for each of the subproblems. The idea is to begin with an observation, in order to determine which DID model to use. When this is done, if it is a different model than the last one, which is always true the first time, we insert the observation into the chosen DID as initial belief, calculate the policy tree and perform the first action.

If it is the same model, then we need to see whether the time horizon limit has been reached. If

[19] See Appendix C.2 for the full list

[20] Review l. 19.

[21] *void* Run(*Robot* *)

[22] We are not considering the GUI in this context.

this is the case, we need to insert a new initial belief and recalculate the policy tree, after which we perform an action. If we have not reached the limit, the next action can be performed according to the policy tree. As mentioned earlier the critical difference lies in the creation of the model and the calculation of policy trees, otherwise it is the same.

Depending on whether the robots are in the same part of the decomposed environment, we can determine which I-DID model to use. But before the policy can be calculated, we must determine which DID model the 'Leader' is using, and use that model to generate the I-DID model. Due to this, the I-DID model may need to be regenerated and the policy tree recalculated many times compared to the DID model. It is a drawback that is introduced because of the decomposition of the problem. This is not optimal, but since the decomposition allows us to work with a larger problem size, it is worth the drawback. Even more so because the decomposition method means that it is possible to scale the problem size to something that would otherwise not have been possible.

```
1  while( true ) {
2     // robot1
3     observation of robot1
4     determine which DID that should be used
5     if( it is the same model as the current? ){
6        if( is time horizon not at the limit? ){
7           perform next action
8        }else{
9           recalculate policy tree with new initial belief
10          perform action
11       }
12    }else{
13       insert observation as initial belief
14       calculate policy tree
15       perform action
16    }
17    //robot2
18    observation of robot1 and robot2
19    determine which I-DID that should be used
20    if( it is the same model as the current? ){
21       if( has the DID model changed? ){
22          set timehorizon to the limit
23       }
24       if( is timehorizon not at the limit? ){
25          perform next action
26       }else{
27          determine which DID that should be used for robot1
28          generate I-DID using chosen DID model
29          recalculate policy tree with new initial belief
30          perform action
31       }
32    }else{
33       determine which DID that should be used for robot1
34       generate I-DID using chosen DID model
35       insert observation as initial belief
36       calculate policy tree
37       perform action
38    }
39 }
```

Listing 6.3: Pseudo code of main logic

6.6 Pipe & GUI

The final modules of the implementation are the GUI and the pipe. The pipe is an IPC and to poll the data, the correct API is required. The API is very simple and we will give a brief overview of

it, in this section. Furthermore the data is presented by a GUI, which is mainly developed using drag-and-drop tools. We will give a brief description of how to interpret the signs on the GUI.

6.6.1 Pipe

Our implementation of the pipe is based on the 'server-client' architecture. The client sends requests for the data, and the server pushes it to the client, very similar to the FTP- and HTTP protocol. The server creates a channel, and any client that wishes to communicate with it must know the channel. The server constantly listens for request, and if it receives a request it does not understand, the request will be ignored. We will briefly elaborate how the channel is made, and which request to send, and what format to expect the data.

Server

The server creates the channel, and implements a function which listens for requests. Unless the developer knows when the request are inbound, the listing function must be implemented as a thread. The following code is a high-level description of the implementation of the server. The full implementation of the server can be found in pipe.cpp, in the project folder on the enclosed CD.

```
1  startPipe(){
2     npipe = CreateNamedPipeA('ChannelA');
3   }
4
5  pipeRunning(){
6   while(ConnectNamedPipe(npipe)){
7
8     while(ReadFile(npipe,buff)){
9
10     if(buff == '0'){
11      tmp = Robot1->position * 100000;
12      if(!WriteFile(npipe, tmp){return 'Error, failed to send position of Robot1';};
13     }
14     if(buff == '1'){
15      tmp = Robot2->position * 100000;
16        if(!WriteFile(npipe, tmp){return 'Error, failed to send position of Robot2';};
17     }
18    }
19   }
20   return;
21  }
```

Listing 6.4: Pseudo code of the pipe(server)

The method `startPipe()` creates a channel with the name 'ChannelA', and any client that wishes to communicate with the server must send request on this channel. The method `pipeRunning()` is intended to be executed as a thread, which exists until the channel is terminated, review l. 6. The *while* loop on l. 8, constantly reads incoming requests from the channel, and saves the data into the variable *buff*.If the request is a '0' the position of Robot 1 is transmitted, and if it is '1' the position of Robot 2 is transmitted. The API is very simple, the name of the channel is 'TestChannel' and the command '0' is interpreted as a request for the position of Robot 1 and '1' for the position of Robot 2. These are the only information necessary to retrieve the position of the robots.

Client

Given the API of the server the client must connect to the specific channel, and send the request. The following code is a high-level description of the implantation of the client. The full implementation of the client can be found in pipe.cpp, in the UI-2 folder on the enclosed CD.

```
1  startPipe(){
2
3     if(!WaitNamedPipeA(ChannelA)){return 'Error, cannot find TestChannel';};
4
5     npipe = CreateFileA(ChannelA);
6     if( npipe == INVALID_HANDLE_VALUE ){return 'Error cannot open Channel';};
7
8     return;
9  }
10 position Update(Robot){
11
12    if(Robot == Robot1){if(!WriteFile(npipe,0)){return 'Error transmitting request';};
13
14    if(Robot == Robot2){if(!WriteFile(npipe,1)){return 'Error transmitting request';};
15
16    if(!ReadFile(npipe,buff)){return 'Error receiving request';};
17    Robot->position = buff/100000;
18
19    return Robot->position;
20 }
```

Listing 6.5: Pseudo code of the pipe(client)

The method `startPipe()`, on the client side, is executed to setup the pipe, that is to associate the channel[23] with the pipe-handler, *npipe*. The method *position* `Update(Robot)` transmits the request for the position of a given robot and waits for response. The response from the server is stored in *buff*, review l. 15.

The high-level code only depicts the main steps of initializing, transmitting and receiving messages through a channel. There are however some issues using channels to transmit data. Only characters can be sent through a channel, and if decimal values are cast into characters, everything after the decimal is discarded. We have chosen 5 decimals to be sufficient, and by multiplying the position of each robot with 10^5, ensures that 5 decimals are sent along, this must of course be countered on the client side.

6.6.2 GUI

The GUI retrieves the results of localization using the pipe and presents the data for the user. The environment is mapped into a [4x4] square, which illustrates the states of the environment. The state space has however been altered but we kept this design for illustrative purposes, review Figure 6.8.

The history of each Robot is marked by the lines, and at the current position is marked by the title, that is 'Robot 0' is the first robot. This however only illustrates an approximate position, the precise position, is illustrated at the bottom of the window in the text field. The tick box, turns the history on and off, in case only the position of one of the robot is relevant.

The GUI is a simple example of how the pipe can be utilized to use the results of localization for

[23] ChannelA

Figure 6.8: A sample the robots movement, illustrated by the GUI

other purposes than only to support the decision process.

Chapter 7

Experimental results

We have performed experiments in order to verify that the robots perform as expected. This includes checking the generated policy tree against the actual performance. We conduct experiments with the DID model (one robot) and experiments with the I-DID model (two robots). We present one experiment for the DID model and one for the I-DID model, while the results of the other experiments are on the enclosed CD[1] [2].

7.1 DID model

The DID model is used to model a single robot. In this project it is used to model the 'Leader', as described in the scenario in section 6.1. Due to the decomposition of the problem, described in section 6.3.2, multiple models are used to represent the entire problem. There exist different policy trees for each of the models. We will break down a single experiment, from some state to the goal, into smaller parts corresponding to the decomposed problem sizes.

7.1.1 Experiment

In this experiment, the goal has been defined as the state in the lower left corner of the environment, and the starting position of the robot is the top right state. The initial orientation of the robot is `WEST`. This is illustrated on figure 7.1. Because of the decomposition, we start with the model for quadrant 4, as illustrated on figure 6.4. When the robot moves to another quadrant, the model for that quadrant is used. We repeat this until the robot reaches the goal.

Figure 7.2 illustrates the initial configuration of quadrant 4, as well as the policy tree generated for this. According to the policy tree, in best case, the robot should move `WEST` from state 4 into state 3 and then continue out the left part of the quadrant. Figure 7.3 illustrates the recorded path of the robot for the execution of the test. The robot moves `WEST` all the way out of the quadrant as we anticipated. The only comment is that it is not moving in a straight line, but the deviation is taken into account as uncertainty in the transition function, see Appendix A.1. The transition function

[1] cd:/Experimental_Results/DID/
[2] cd:/Experimental_Results/I-DID/

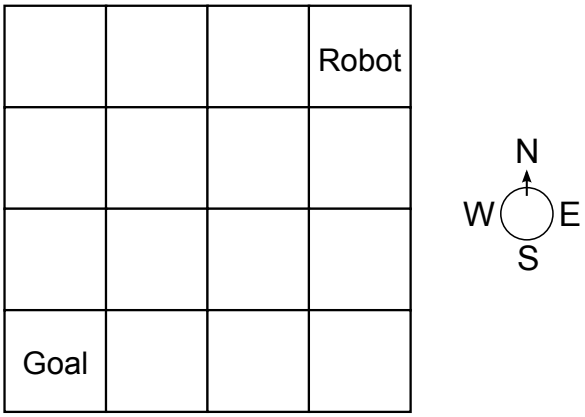

Figure 7.1: Illustration of the goal state, the initial robot position and the orientation reference.

has a 85% success rate on moving, so the first couple of moves should be without any significant mistakes. However as the robot moves, the error accumulate and the robot will eventually make a mistake. When the robot moves `WEST` out of quadrant 4, it enters quadrant 3.

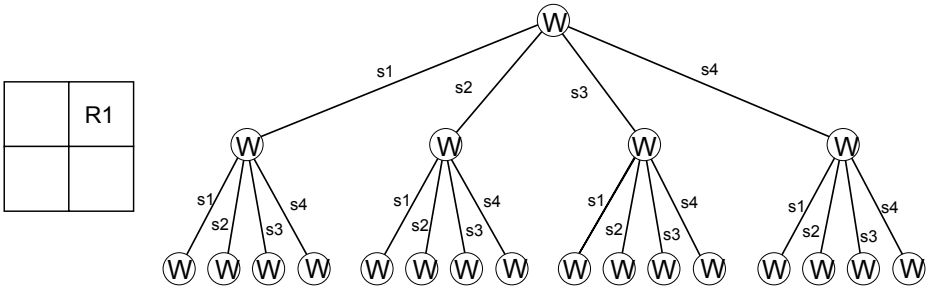

Figure 7.2: The initial configuration of quadrant 4 (orientation is `WEST`), and the generated policy tree.

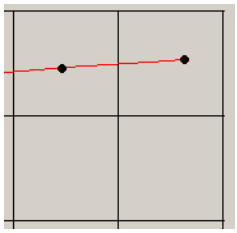

Figure 7.3: Recorded actual path of the robot in quadrant 4. Black dots shows where the robot performed its observation.

Figure 7.4 illustrates the initial configuration of quadrant 3, as well as the policy tree generated for this. According to the policy tree, in the best case, the robot should move `SOUTH` from state 4 into state 2 and then continue out the bottom part of the quadrant. Figure 7.5 illustrates the recorded path of the robot for the execution of the test. The robot moves `SOUTH` as we anticipated, followed by another move `SOUTH` out of the quadrant. The robot follows the policy without any mistakes, and the accumulating error from the transition function is still not big enough to become relevant. As the robot moves `SOUTH` out of quadrant 3, it enters quadrant 1.

Figure 7.6 illustrates the initial configuration of quadrant 1, as well as the policy tree generated for this. According to the policy tree, in the best case, the robot should move `WEST` from state 4 to state 3 and then `SOUTH` into state 1. As state 1 of quadrant 1 is the goal state, the experiment is over when

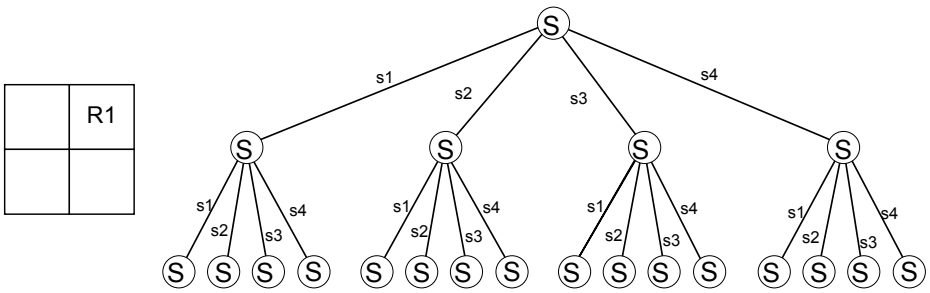

Figure 7.4: The initial configuration of quadrant 3, and the generated policy tree.

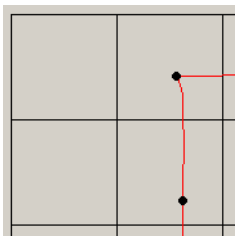

Figure 7.5: Recorded actual path of the robot in quadrant 3. Black dots shows where the robot performed its observation.

the robot reaches this state. Figure 7.7 illustrates the recorded path of the robot for the execution of the test. The robot moves `WEST` followed by a move `SOUTH` into state 1. Again the robot follows the policy tree flawlessly. The accumulated error of the transition function almost causes the robot to make a mistake by missing state 1, but as this is not the case no further comments on this.

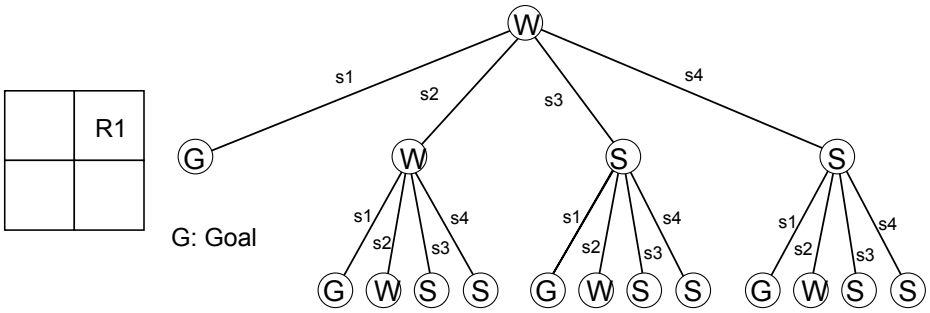

Figure 7.6: The initial configuration of quadrant 1, and the generated policy tree.

Figure 7.8 illustrates the full recorded path for the robot in the environment. The black dots shows where the robot performed its observations.

7.2 I-DID model

The I-DID model is used to model a robot, with knowledge of the model for a second robot. In this project it is used to model the 'Follower', as described in the scenario in section 6.1. According to the scenario, the 'Follower' is supposed to follow the 'Leader' to whichever goal it might have. This is done using observations prone to errors. As described in section 6.3.2, because of the decomposition of the problem, multiple models are used to represent the entire problem. A consequence of this is, there exists a different policy tree exist for each of these models. Because of this we will

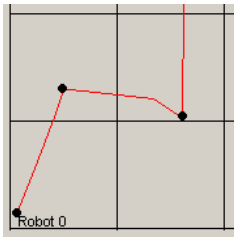

Figure 7.7: Recorded actual path of the robot in quadrant 1. Black dots shows where the robot performed its observation.

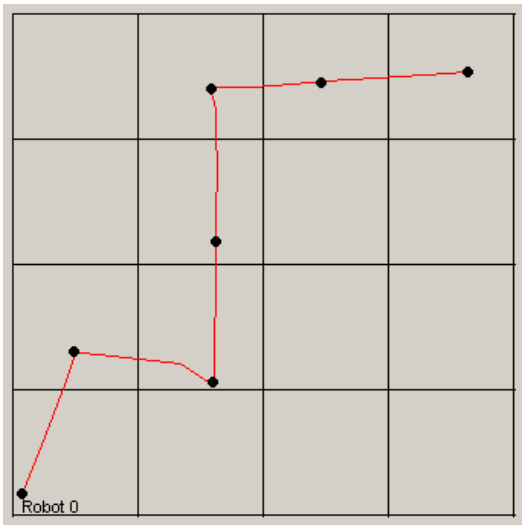

Figure 7.8: Full actual recorded path in the environment.

break down a single experiment, from some state to the goal, into smaller parts corresponding to the decomposed problem sizes.

7.2.1 Experiment

In this experiment, the goal has been defined as the state in the lower left corner of the environment, and the starting position of the 'Leader' is the top right state in quadrant 3. The initial orientation of the 'Leader' is `WEST`. The initial position of the 'Follower' is the top right corner of quadrant 4, and the initial orientation is `WEST`. This is illustrated on figure 7.9. As opposed to the experiment described in section 7.1.1, we now have 2 robots that are supposed to move around in the environment at the same time. Because of this we have to generate policy trees for both robots, and because the model for the 'Follower' is based on the current model for the 'Leader', generating and maintaining a policy tree for it is hard. The creation of the policy trees is described in section 6.3.2. Because of the decomposition, we start with the model for quadrant 3 for the 'Leader', and the top level model for the 'Follower'. Then as each robot moves around in the environment, each time any one robot moves to another quadrant, the model for both robots has to be reevaluated. We repeat this until both robots reach the goal.

Figure 7.10 illustrates the initial belief of each robot. On the left is quadrant 3, in which the 'Leader' is in the top right corner. On the right is the entire environment with the 'Leader' somewhere in the top left corner and the 'Follower' somewhere in the top right corner. We generate policy trees for

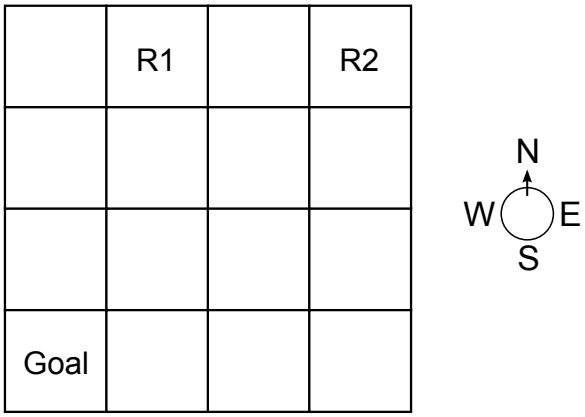

Figure 7.9: Illustration of the goal state, the initial robot position and the orientation reference. R1 is the 'Leader' and R2 is the 'Follower'.

each robot using these initial beliefs. The policy tree for the 'Leader' is illustrated on figure 7.4. The policy tree for the 'Follower' is too big to fit in the report, so it is on the enclosed CD[3]. According to the policy tree for the 'Leader', it is supposed to move `SOUTH` and then, regardless of the observation, `SOUTH` again. This would take the 'Leader' from quadrant 3 to quadrant 1. The motive for the 'Follower' is to mimic the actions of the 'Leader' using the model of the 'Leader' within its own model. In this way it 'predicts' the actions of the 'Leader', which is reflected in the policy tree for the 'Follower'. According to the policy tree, the 'Follower' is supposed to move `SOUTH` and then, regardless of the observation, `SOUTH` again. This would take it from quadrant 4 to quadrant 2. Figure 7.11 and figure 7.12 illustrates the recorded actual path of the 'Leader' and the 'Follower'. Both robots perform as expected, however the accumulating error in the transition function causes both robots to move in a skew line. After these actions both robots have changed quadrant, so the models and the policy trees need to be regenerated.

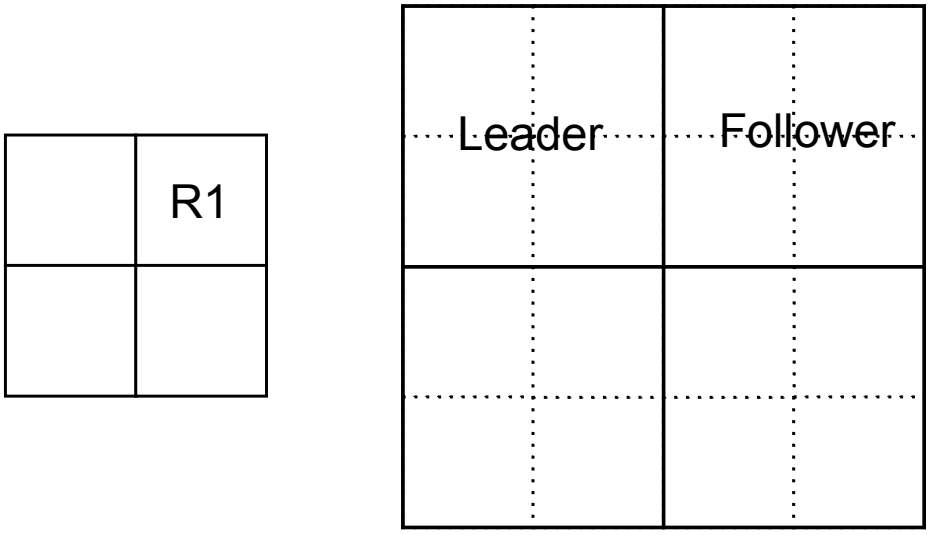

Figure 7.10: The initial belief of both robots. On the left is the 'Leader's' initial belief of quadrant3, and on the right is the 'Follower's' initial belief of the entire environment.

As the robots are moving in sequence, the new policy tree for the 'Leader' is recalculated first. The 'Leader' is in the top left corner, and has the orientation `SOUTH`. This information is used

[3] See cd:/Picture/Sampling/idid_policy_tree_big.pdf

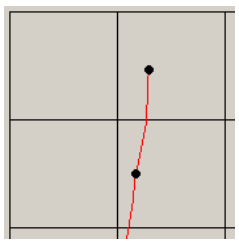

Figure 7.11: Recorded actual path of the ‘Leader’ in quadrant 3. Black dots shows where the robot performed its observation.

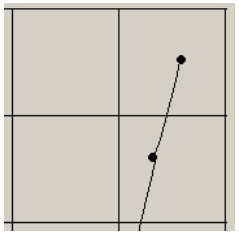

Figure 7.12: Recorded actual path of the ‘Follower’ in quadrant 4. Black dots shows where the robot performed its observation.

to generate the new policy tree, which is illustrated on figure 7.13. According to the policy tree, the ‘Leader’ is supposed to move `SOUTH` and hopefully end up in the goal. If it manages to miss the goal, the robot would probably end up in state 2, in which case moving `WEST` is the option. Figure 7.14 illustrates the recorded actual path of the ‘Leader’ when it enters quadrant 1 along with the path for the ‘Follower’ when it enters quadrant 1. The ‘Leader’ is the red path that enters the quadrant from the top, and the ‘Follower’ is the black path that enters the quadrant from the right. Because the robots are moving sequentially to simulate moving at the same time, the ‘Leader’ takes its action before the new policy tree is calculated for the ‘Follower’. This is important because when the ‘Leader’ reaches the goal state, the ‘Follower’ is no longer able to mimic the actions of the ‘Leader’. When that happens, the ‘Follower’ has to revert to using a DID model and rely on the bad observations to get to the goal. As illustrated on figure 7.14, the ‘Leader’ moves `SOUTH` and successfully enters the goal state, thus forcing the ‘Follower’ to revert to using a DID model. In this experiment the accumulated error from the transition function causes the ‘Leader’ to move in a skew path and it is therefore able to reach the goal state in 3 moves instead of 4 theoretically perfect moves. This proves the error related to the transition function, but in this case it is pure luck that it moved to the goal instead of away from the goal.

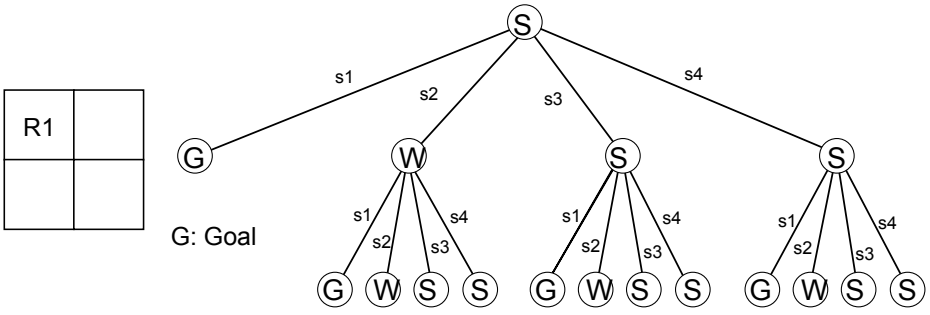

Figure 7.13: The initial configuration for the ‘Leader’ in quadrant 1, and the generated policy tree.

Before the policy tree for the ‘Follower’ is regenerated, the model is reevaluated. Because the

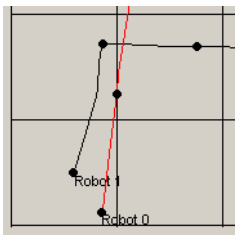

Figure 7.14: Recorded actual path of the ‘Leader’ and the ‘Follower’ in quadrant 1. Black dots shows where the robots performed observations.

‘Leader’ has reached the goal, the ‘Follower’ is forced to use a DID model along with erroneous observations to reach the goal. Figure 7.15 illustrates the initial configuration of the ‘Follower’ in quadrant 2 along with the generated policy tree. It is obvious that the ‘Follower’ will move `WEST`, followed by another move `WEST` regardless of the observation. Figure 7.16 illustrates the recorded actual path for the ‘Follower’ in quadrant 2. It moves `WEST` followed by another move `WEST` out of quadrant 2 and into quadrant 1. The path that the robot is moving in is crooked, but not enough to cause an error.

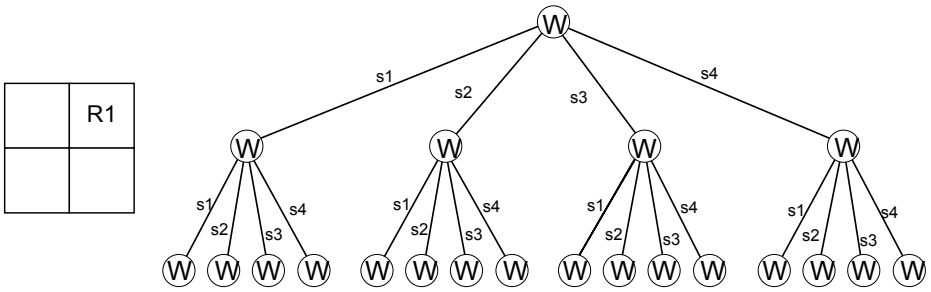

Figure 7.15: The initial configuration for the ‘Follower’ in quadrant 2, and the generated policy tree.

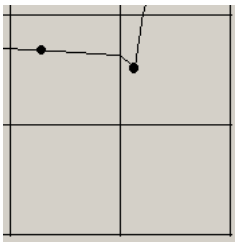

Figure 7.16: Recorded actual path for the ‘Follower’ in quadrant 2. Black dots shows where the robot performed its observation.

As the ‘Follower’ enters quadrant 1, it is positioned in the top right state with the orientation `WEST`. Figure 7.6 illustrates this initial configuration and the generated policy tree. According to the policy tree, it is supposed to move `WEST`, to the top left corner, wherefrom it will move `SOUTH` into the goal state. Figure 7.14 illustrates the recorded actual path of the ‘Leader’ and the ‘Follower’ in quadrant 1. The ‘Follower’ is the black path entering the quadrant from the right. It moves `WEST` and then `SOUTH` into the goal state. Again the error in the transition function is apparent, but is not big enough to actually cause any errors.

Figure 7.17 illustrates the full actual recorded path for both robots in the environment. The black dots show where the robots performed their observations.

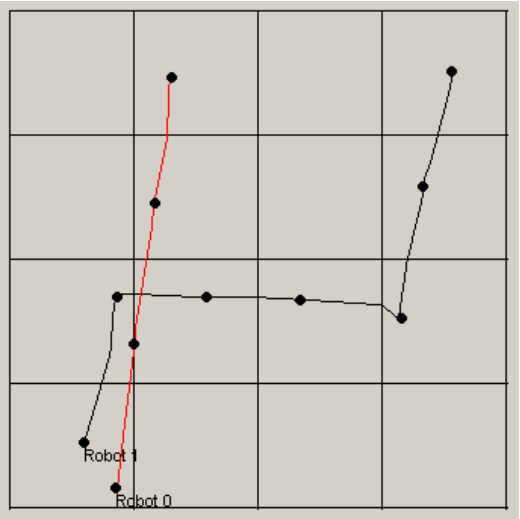

Figure 7.17: The full actual recorded path in the environment for both robots. The ‘Follower’ is represented by the black path starting in the top right corner of quadrant 4, and the ‘Leader’ is represented by the red path starting in the top right corner of quadrant 3.

Overall the robots performed as we expected them to. Errors due to the accumulated error in the transition function did happen, but to an extend that was within the expected acceptable margin.

Chapter 8

Future Work

The experience gathered during implementation and experimentation of the application, indicate that there is room for improvement. This chapter will elaborate the improvements and describe an eventual implementation of these. The improvements are categorized into three sections Decision, Localization and GUI & Pipe, refereeing to aspects of the Design.

8.1 Implementation

The overall implementation is successful, but not without room for improvements. For future work purposes a number of the key elements of the implementation can be improved in some way or an other.

The decomposition of the environment into smaller sections is a very good idea, as it allows us to work with an environment that would be too big for being computationally feasible. The solution however can be improved in the sense that it is a tradeoff between how optimal the actions are and the computational power needed. An improvement is possibly a more optimal decomposition or an entirely new way of dealing with large environments.

The data structure used in the implementation, is the compact data structure presented in section 6.3.1. It deals with the exponentially growing CPT size of Mod nodes in the I-DID models. The compact model itself is very good, reducing the number of nodes needed to represent a policy tree from an exponential amount to a mere linear amount. While this is good, it does not fully support actual implementation and issues related to this. In implementation we have to generate policy trees for the robots to follow, this part is somewhat simple. Following a compact data structure of a policy tree is not easy, because as the time horizons increase the number of arcs increase exponentially until the levels become fully connected. This is partially true for the 3 timehorizon example on figure 6.2. The problem is with a fully connected level, that it is impossible to figure out which action the robot should take. Because of that a conversion from compact data structure back to normal policy tree is needed, but this is not possible as several "'new"' paths appear that would not exist otherwise. Finally because of all of these peculiarities with the compact data structure we were forced to argument the data structure with data about parents, so that it is possible to make sense of the fully connected levels (backtrack). This means that while the number of nodes used to represent the data structure is linear in the time horizon, the amount of argument data is still very

much exponential in the time horizon, thus limiting the it. The improvement to the compact data structure seems nigh impossible to achieve, and that is to completely eliminate the exponential part of representing policy trees. Be that in the compact data structure or a entirely new data structure.

The transition function could use a lot of improvement, is on the hardware level. On the hardware level is the Lego Mindstorms robot, which is unreliable at best. While moving forward is done with high precision, turning is an other matter with deviations of $+/-30°$. This irregularity in the precision of actions made it hard to create a viable transition function. The improvement to the transition function can be reduced to a more reliable robot. As the robot gets more reliable, the transition function becomes easier to create and also more precise.

Finally in the current implementation issues related to collision detection have been omitted. This is because in our scenario we have a leader and a follower, where the follower mimics the action of the leader. In a scenario like that collision is a theoretical impossibility. This impossibility is based on the scenario stating that the robots are moving at the same time in parallel. In our implementation however we move the robots in sequence. This is because moving in parallel is much harder to implement and highly prone to error, while moving in sequence is easy to implement and not nearly as prone to errors, but because of this the theoretical impossibility becomes quite possible. Improvements here is the addition of collision detection to the implementation, and the possibility of moving both robots in parallel.

8.2 WiiRemote

Due to the Wii Remote and the mathematical models for positioning, we are able to perform exact localization. There is however an issue which we did not consider while developing the mathematical models. The position of an IR diode is calculated as the orthogonal distance to the edges. But the theory does not consider whether the Robot is inside or outside the environment, review Figure 8.1. The robot's physical position is illustrated by the point R, and it is positioned outside the environment. Using the current model, the position will appear as if the robot was inside the environment, mirrored in the Y-axis of $Q1$, and in position S.

The solution is however rather simple, to determine the distance the point T[1] must be calculated, see Figure 8.1. A comparison of the x-coordinates of the R and T will reveal whether R is within the environment. Using the proposed solution, an 'emergency brake' can be implemented to stop the robot, whenever it is about to cross over.

Another issue regarding the implementation of the Wii Remote is surrounding the pairing[2]. Each time the connection between the computer and Wii Remote is dropped, e.g. when the computer or the Wii Remote is turned off, it must be paired again. Each time they are paired, they are listed in a different sequence in the Bluetooth stack. The static implementation of the Wii Remotes[3] causes the instances of *class* `wiimote` to switch between the Wii Remotes. The calibration is a static implementation, and the first Wii Remote is bound to $Q1$ and the second to $Q2$. Currently it is easier to switch the argument to the `Connect()` method in the implementation[4] and recompile the project, than physically exchanging the Wii Remotes on the ceiling, but this is not a lasting solution.

[1] The intersection between the Y-axis and the orthogonal function to the Y-axis which goes through R.
[2] Establishing the Bluetooth connection
[3] Review l. 6 in Listing 6.2.
[4] Change the argument from 1 to 2, on l. 6 in Listing 6.2

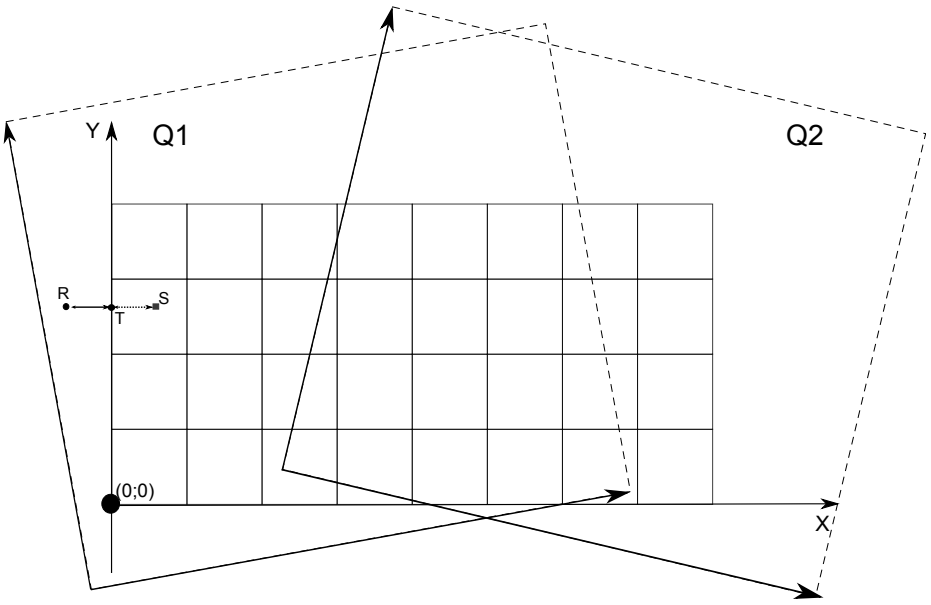

Figure 8.1: If the robot is outside the environment, its position is mirrored by the edges.

The current implementation can however be extended to request the user to identify which Remote is the first and second. Even though this a better design, the optimal solution is undoubtly to identify the Wii Remotes on their Bluetooth Mac-address, but the current implementation of the WiiYourSelf! does not support this.

8.3 GUI & Pipe

The GUI only serves to illustrate the position of the robots. It can however be expanded to give a much more detailed picture of the underlying models. The policy tree for each robot can be displayed together with the observation, allowing the user to see the future actions. This information can be viewed next to the already implemented environment, where the precise position can be illustrated. Using a small textbox in each state of the environment, it is possible to show the belief state of each robot. All these information are necessary to perform quick and efficient debugging of the underlying models. Developing the GUI is does not impose great difficulty, but the current implementation of the pipe must be extended. The current protocol, where the client requests for the data, will undoubtly become a bottleneck and implementing different channels for the different types of data will therefore be recommendable.

Implementing a Pipe to allow other applications to retrieve the results of the localization is in essence a good idea. But when we are dealing with large amounts of data, as suggested above, implementing the GUI together with the rest as one unified application will be the most efficient approach. To do so, we must implement the GUI without the use of drag-and-drop development tools, or use another library. The library has tied our hands with respect to the OS, and replacing it with another more versatile library would make the implementation attractive for a larger audience.

Chapter 9

Conclusion

Implementing multi-agent systems involves many aspects other than just modeling the I-POMDP. We will in this section look into some of the aspects, and finally summarize the total achievements.

The physical robots were built using Lego Mindstorm. The kit is, as previously stated, just a toy, and cannot be used in our context. The sensors were unreliable; the compass sensor in particular was directly erroneous. Readings made with this equipment is prone to high deviation and cannot be relied upon. The actuators are however another matter. They drive forward in a very precise manner, but they are very difficult to control. Theoretically the odometry motion should have given the most precise control, but this is not the case. Moving in a straight line was a success and could be done with high precision. But turning is another matter, the different torque and resistance among the different motors becomes very apparent. Fact is that they rarely perform a 90° turn, and even though the implementation is consistent, the outcome of the turns are varying, making it hard to define a good viable transition function.

By discarding the sensors supplied by Lego, we were forced to look elsewhere for the means of performing localization. The Wii Remote has turned out to be an excellent tool. The tracking can be done with and error coefficient of approx. $5[mm]$, and considering the cost of a single Wii Remote, this is very impressive. The implementation is made so that the environment is highly scalable, so other Wii Remotes can easily be added to increase the area. A limit however is that we can only track up to four robots simultaneously with a single Wii remote. The tracking system using the Wii Remotes has turned out to be a very successful by-product of this project.

The 'Follow the Leader' scenario is partially created using POMDP and partially I-POMDP. The 'Leader' is focused on arriving to the goal and even though it is modeled as a POMDP, we provide exact observation to it, so it performs as few wrong actions as possible. The 'Follower' mimics the movement of the 'Leader' using an I-POMDP to predict its movement. The experimental results indicate the modeling is correct. The wrong turns however is an indication of the accumulated error in the transition function, but this is not a cause for concern as the error accumulates slowly. Because the CPT size of the Mod node grows exponentially as a function of the time horizon and the number of observations, we have chosen to decompose the environment into smaller sections. This decomposition is highly scalable, but has a side effect of producing a lot of small DID models, as a separate model is required for each local area.

Because of the decomposition of the environment, the required time horizon needed to perform

actions within the local area of the robot is quite low. This decomposition and low time horizon means that the actions chosen by the robots will be locally optimal, but not necessarily globally optimal. E.g. on local scope moving `WEST` is the optimal action maximizing the expected reward, but on a larger scale `SOUTH` could be even more rewarding. Adding time horizons cannot fix this problem, as the robot only considers the environment it is currently in, which is limited by the decomposition. Arguments can be made for increasing the size of the decomposed section and increasing the time horizon to improve the quality of the robots actions. But doing this invokes another problem inherent to I-DID models, which is the exponential increase in the CPT size of the Mod node. We have investigated how to combat this increase using a compact data structure to represent the policy tree, but the benefits were limited. Ultimately it is a tradeoff between optimizing the actions and in turn the expected reward gained from actions and the computation power needed to calculate policy trees etc. In the end a robot that performs a single action and then pauses for 1 hour for calculations is no good compared to robots able to move in near real time.

Bibliography

[Com09] NXT++ Community. Nxt++. http://nxtpp.clustur.com/index.php/Main_Page, Last checked 13-06-09.

[DZC09] Prashant Doshi, Yifeng Zeng, and Qiongyu Chen. Graphical models for interactive pomdps: representations and solutions. *Autonomous Agents and Multi-Agent Systems*, 18(3):376–416, 2009.

[gl.09] gl.tter. Wiiyourself! - native c++ wiimote library v1.01. http://wiiyourself.gl.tter.org/WiiYourself!_1.01a.zip, Last checked 13-05-09.

[Gro09] LEGO Group. Meet the robots. http://mindstorms.lego.com/Overview/MTR_Tribot.aspx, Last checked 13-06-09.

[HNH08] S. Hay, J. Newman, and R. Harle. Optical tracking using commodity hardware. In *Mixed and Augmented Reality, 2008. ISMAR 2008. 7th IEEE/ACM International Symposium on*, pages 159–160, 2008.

[Hug09] Hugin. Hugin expert. http://www.hugin.com/, Last checked 29-07-09.

[JN07] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer Publishing Company, Incorporated, 2007.

[Lab09] Decision Systems Laboratory. Influence diagrams. http://genie.sis.pitt.edu/GeNIeHelp/Decision-theoretic_Modeling/IDs.htm, Last checked 13-06-09.

[Leg09a] Lego. Compass sensor. http://shop.lego.com/Product/?p=MS1034, Last checked 26-01-09.

[Leg09b] Lego. Light sensor. http://shop.lego.com/ByTheme/Product.aspx?p=9844&cn=17, Last checked 26-01-09.

[Leg09c] Lego. Ultrasonic sensor. http://mindstorms.lego.com/overview/Ultrasonic_Sensor.aspx, Last checked 26-01-09.

[Leg09] Lego. Lego mindstorms. http://mindstorms.lego.com/, Last checked 29-07-09.

[LPKC95] Michael L. Littman Leslie Pack Kaelbling and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1995.

[SHH09] Joseph Newman Simon Hay and Robert Harle. Optical tracking using commodity hardware - homepage. http://www.cl.cam.ac.uk/~sjeh3/wii/, Last checked 13-05-09.

[SZ07] Sven Seuken and Shlomo Zilberstein. Memory-bounded dynamic programming for dec-pomdps. In *In Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI*, pages 2009–2015, 2007.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

Appendix A

Transition function

A1	North				West			
S1	State 1	State 2	State 3	State 4	State 1	State 2	State 3	State 4
State 1	0.05	0.05	0.05	0.05	0.85	0.85	0.05	0.05
State 2	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
State 3	0.85	0.05	0.85	0.05	0.05	0.05	0.85	0.85
State 4	0.05	0.85	0.05	0.85	0.05	0.05	0.05	0.05
A1	East				South			
S1	State 1	State 2	State 3	State 4	State 1	State 2	State 3	State 4
State 1	0.05	0.05	0.05	0.05	0.85	0.05	0.85	0.05
State 2	0.85	0.85	0.05	0.05	0.05	0.85	0.05	0.85
State 3	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
State 4	0.05	0.05	0.85	0.85	0.05	0.05	0.05	0.05

Table A.1: Transition function in the DID. The low level I-DID transition function CPT is the same 4 times over, because actions of the first robot does not affect movement of the second robot.

A1	North				West			
S1	State 1	State 2	State 3	State 4	State 1	State 2	State 3	State 4
State 1	0.53	0.01	0.0010	0.0010	0.948	0.45	0.0010	0.01
State 2	0.01	0.53	0.0010	0.0010	0.0010	0.53	0.948	0.01
State 3	0.45	0.01	0.948	0.05	0.05	0.01	0.0010	0.45
State 4	0.01	0.45	0.05	0.948	0.0010	0.01	0.05	0.53
A1	East				South			
S1	State 1	State 2	State 3	State 4	State 1	State 2	State 3	State 4
State 1	0.53	0.0010	0.01	0.0010	0.948	0.05	0.45	0.01
State 2	0.45	0.948	0.01	0.05	0.05	0.948	0.01	0.45
State 3	0.01	0.0010	0.53	0.0010	0.0010	0.0010	0.53	0.01
State 4	0.01	0.05	0.45	0.948	0.0010	0.0010	0.01	0.53

Table A.2: Top level transition function in the I-DID is the same 4 times over, because actions of the first robot does not affect movement of the second robot.

Appendix B

Numerical Example

The numerical example will be illustrated using the readings presented in Figure B.1.

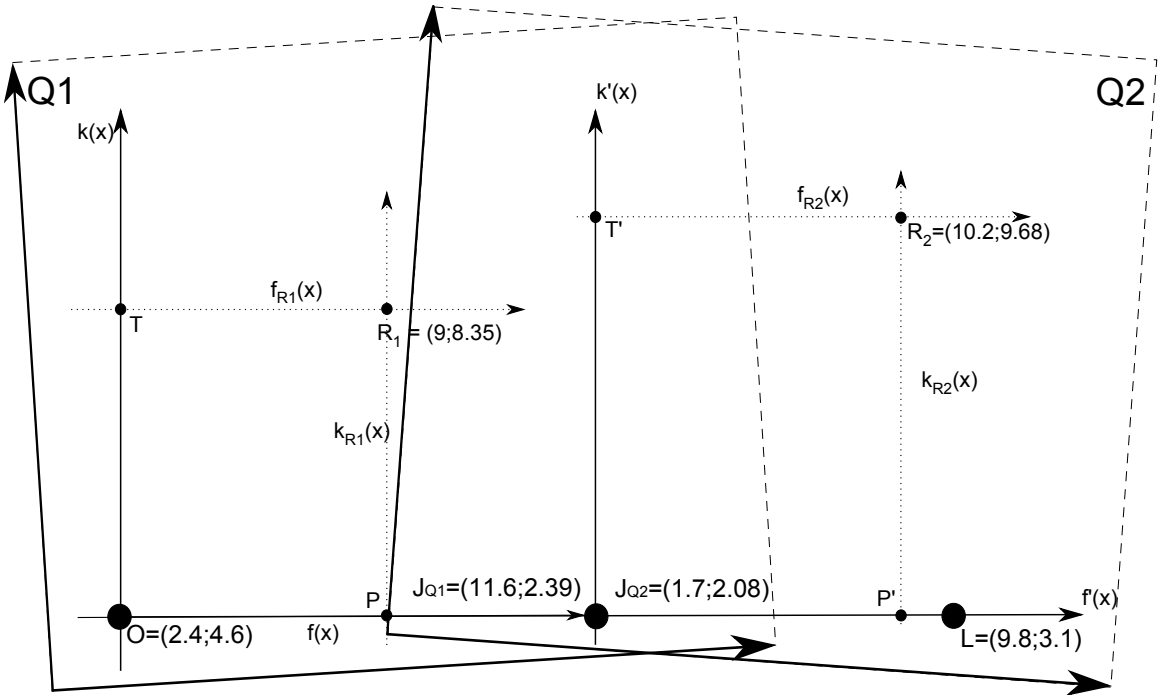

Figure B.1: Numerical Example

For the purpose of this example a life size model was made on paper; Figure B.1 is an illustration of that specific model. The illustration has not been scaled, so the interrelations of the points are not correct. The figure is only for illustrative purposes, and to avoid eventual misinterpretation the grid cell structure of the environment has been removed.

B.1 Positioning

In contrast to the theoretical proof in the report, we will here illustrate the theory using both views. R_1 and R_2 are readings made in the environment, each in its view. We will first calculate the position of R_1 in $Q1$ and thereafter R_2 in $Q2$, these position will be relative to the local scope. Hereafter they will be placed according to the global scope. The reading of Point J is made twice, J_{Q1} is the reading made with the Wii Remote for $Q1$, and J_{Q2} is made by the second Remote.

B.1.1 Local scope

Positioning of R_1

First off, the edges of the environment must be established, that is $f(x)$ and $k(x)$.
$\underline{f(x)}$:
The readings made at the points O and J_{Q1} will be used to define $f(x)$.

$$\begin{aligned} f(x) &= \alpha_f \cdot x + b_f \\ \alpha_f &= \frac{\Delta Y}{\Delta X} \\ \alpha_f &= \frac{4.6 - 2.39}{2.4 - 11.6} \\ \alpha_f &= -0.240217391 \end{aligned}$$

Given α_f we can use either O or J_{Q1} to derive b_f.

$$\begin{aligned} f(x_O) = \alpha_f \cdot x_{_O} + b_f &= y_O \\ b_f &= y_O - \alpha_f \cdot x_O \\ &= 4.6 - (-0.240217391 \cdot 2.4) \\ &= 5.17652174 \end{aligned}$$

The X-axis for the environment is defined by the function $f(x) = -0.2402 \cdot x + 5.1765$.
$\underline{k(x)}$:
The orthogonal function to $f(x)$ is derived from α_f and will intersect $f(x)$ in O.

$$\begin{aligned} k(x) \perp f(x) \Leftrightarrow \alpha_k \perp \alpha_f \\ \alpha_k \cdot \alpha_f = -1 \Leftrightarrow \alpha_k = \frac{-1}{\alpha_f} \\ \alpha_k = \frac{-1}{-0.240217391} \end{aligned}$$

Given α_k; b_k can be derived by using the reading made in O.

$$\begin{aligned} k(x_{_O}) = \alpha_k \cdot x_{_O} + b_k &= y_{_O} \\ b_k &= y_{_O} - \alpha_k \cdot x_{_O} \\ &= 4.6 - ((-1)/(-0.240217391) \cdot 2.4) \\ &= -5.39095024 \end{aligned}$$

The Y-axis for the environment is defined by the function $k(x) = \frac{-1}{-0.2402} \cdot x - 5.3910$.

The edges of the environment have now been established for $Q1$, and it is possible to derive the position of R_1 according to O.

X-coordinate of R_1
The x-coordinate for R_1 is the orthogonal distance from $k(x)$ to R_1. To calculate the distance, an orthogonal function to $k(x)$ must be defined which goes through R_1.

$$k(x) \perp f_{_{R1}}(x) \Leftrightarrow \alpha_k \perp \alpha_{f_{R1}}$$

As $f(x)$ is already orthogonal to $k(x)$; $\alpha_{f_{R1}} = \alpha_f$ and its respective $b_{_{R1}}$ can be derived by using R_1 and $f_{_{R1}}(x)$.

$$\begin{aligned} f_{_{R1}}(x_{R1}) = \alpha_{f_{R1}} \cdot x_{_{R1}} + b_{f_{R1}} &= y_{_{R1}} \\ b_{f_{R1}} &= y_{_{R1}} - \alpha_{f_{R1}} \cdot x_{_{R1}} \\ &= 8.35 - (-0.240217391 \cdot 9) \\ &= 10.5119565 \end{aligned}$$

The orthogonal function to $k(x)$ which goes through R_1 is defined by $f_{R1}(x) = -0.2402 \cdot x + 10.5120$. The intersection of $f_{R1}(x)$ and $k(x)$ will be denoted T. The distance between T and R_1 will be the orthogonal distance, and the x-coordinate.

$$\begin{aligned} f_{R1}(x) &= k(x) \\ \alpha_{f_{R1}} \cdot x + b_{f_{R1}} &= \alpha_k \cdot x + b_k \\ x \cdot (\alpha_{f_{R1}} - \alpha_k) &= b_k - b_{f_{R1}} \\ x &= \frac{b_k - b_{f_{R1}}}{\alpha_{f_{R1}} - \alpha_k} \\ x &= \frac{-5.39095024 - 10.5119565}{-0.2402 - \frac{-1}{-0.240217391}} \\ x &= 3.61175568 \end{aligned}$$

Given the x-coordinate the corresponding y-coordinate can be derived using either $f_{_{R1}}(x)$ or $k(x)$, as it will yield the same result.

$$\begin{aligned} f_{_{R1}}(3.61175568) &= -0.2402 \cdot 3.61175568 + 10.5119565 \\ f_{_{R1}}(3.61175568) &= 9.64441279 \end{aligned}$$

The point of intersection between $f_{R1}(x)$ and $k(x)$ is $T = (3.6118; 9.6444)$. The distance between T and R_1 can by calculated by Pythagoras.

$$\begin{aligned}
|TR_1|^2 &= (\Delta Y)^2 + (\Delta X)^2 \\
|TR_1| &= \sqrt{(\Delta Y)^2 + (\Delta X)^2} \\
|TR_1| &= \sqrt{(8.35 - 9.6444)^2 + (9 - 3.6118)^2} \\
|TR_1| &= 5.54149534
\end{aligned}$$

The x-coordinate for R_1 according to the fixed origen O is 5.5415.

Y-coordinate of R_1

The y-coordinate for R_1 is the orthogonal distance from $f(x)$ to R_1. To calculate the distance, an orthogonal function to $f(x)$ must be defined which goes through R_1.

$$k_{R1}(x) \perp f(x) \Leftrightarrow \alpha_{k_{R1}} = \frac{-1}{-0.240217391}$$

Given $\alpha_{k_{R1}}$; $b_{k_{R1}}$ can be derived using $k_{R1}(x)$ and R_1.

$$\begin{aligned}
k_{R1}(x_{R1}) &= \alpha_{k_{R1}} \cdot x_{R1} + b_{k_{R1}} = y_{R1} \\
b_{k_{R1}} &= y_{R1} - \alpha_{k_{R1}} \cdot x_{R1} \\
&= 8.35 - (\frac{-1}{-0.240217391} * 9) \\
&= -29.1160634
\end{aligned}$$

The orthogonal function to $f(x)$ which goes through R_1 is defined by $k_{R1}(x) = \frac{-1}{-0.2402} \cdot x + -29.1160$. The intersection of $k_{R1}(x)$ and $f(x)$ will be denoted P. The distance between P and R_1 will be the orthogonal distance, and the y-coordinate.

$$\begin{aligned}
k_{R1}(x) &= f(x) \\
\alpha_{k_{R1}} \cdot x + b_{k_{R1}} &= \alpha_f \cdot x + b_f \\
x \cdot (\alpha_{k_{R1}} - \alpha_f) &= b_f - b_{k_{R1}} \\
x &= \frac{b_f - b_{k_{R1}}}{\alpha_{k_{R1}} - \alpha_f} \\
x &= \frac{5.17652174 - (-29.1160634)}{\frac{-1}{-0.240217391} - (-0.240217391)} \\
x &= 7.78825858
\end{aligned}$$

Given the x-coordinate the corresponding y-coordinate can be derived using either $k_{R1}(x)$ or $f(x)$, as it will yield the same result.

$$\begin{aligned}
k_{R1}(7.78825858) &= \frac{-1}{-0.240217391} \cdot 7.78825858 - 29.1160634 \\
k_{R1}(7.78825858) &= 3.30564657
\end{aligned}$$

The point of intersection between $k_{R1}(x)$ and $f(x)$ is $P = (7.7883; 3.3056)$. The distance between T and R_1 is calculated by Pythagoras.

$$\begin{aligned} |PR_1|^2 &= (\Delta Y)^2 + (\Delta X)^2 \\ |PR_1| &= \sqrt{(8.35 - 3.3056)^2 + (9 - 7.7883)^2} \\ |PR_1| &= 5.18788861 \end{aligned}$$

We have now performed localization of R_1 in $Q1$. The position of R_1 accordign to O is (5.5415;5.1879).

Positioning of R_2

Positioning R_2 in $Q2$ is performed similarly, the greatest difference being the fixed origin is changed to J_{Q2}. The calculation presented here will not be as comprehensive as the previous example.

The edges of the environment for $Q2$ must first be defined. The readings made at the points J_{Q2} and L will be used to define the X-axis of the environment.

$$\begin{aligned} f'(x) &= \alpha_{f'} \cdot x + b_f \\ \alpha_{f'} &= \frac{1.7 - 9.8}{2.08 - 3.1} \\ \alpha_{f'} &= 0.125925926 \end{aligned}$$

Given $\alpha_{f'}$; $b_{f'}$ can be derived by using the reading made in L.

$$\begin{aligned} f'(x_{_L}) = \alpha_{f'} \cdot x_{_L} + b_{f'} &= y_{_L} \\ b_{f'} &= y_{_L} - \alpha_{f'} \cdot x_{_L} \\ &= 3.1 - (0.125925926 \cdot 9.8) \\ &= 1.86592593 \end{aligned}$$

The line depicting the X-axis of $Q2$ is defined by the function $f'(x) = 0.1259 \cdot x + 1.8659$. The Y-axis is an orthogonal function to $f'(x)$ and can be derived using $\alpha_{f'}$.

$$k'(x) \perp f'(x) \Leftrightarrow \alpha_{k'} = \frac{-1}{\alpha_{f'}}$$

$$\alpha_{k'} = \frac{-1}{0.125925926} = -7.94117647$$

Given $\alpha_{k'}$; $b_{k'}$ can be derived by using the reading made in J_{Q2}.

$$\begin{aligned} k'(x_{J_{Q2}}) = \alpha_{k'} \cdot x_{J_{Q2}} + b_{f'} &= y_{J_{Q2}} \\ b_{k'} &= y_{J_{Q2}} - \alpha_{k'} \cdot x_{J_{Q2}} \\ &= 2.08 - (\frac{-1}{0.125925926} \cdot 1.7) \\ &= 15.58 \end{aligned}$$

The line depicting the Y-axis of $Q2$ is defined by the function $k'(x) = \frac{-1}{0.1259} \cdot x + 15.58$. The intersection of $f'(x)$ and $k'(x)$ will be the fixed origin when performing localization in $Q2$.

X-coordinate of R_2
The x-coordinate for R_2 is the orthogonal distance from $k'(x)$ to R_2. To calculate the distance, an orthogonal function to $k'(x)$ must be defined which goes through R_2.

$$k'(x) \perp f_{R2}(x) \Leftrightarrow \alpha_{k'} \perp \alpha_{f_{R2}}$$

As $f'(x)$ is already orthogonal to $k'(x)$; $\alpha'_f = \alpha_{f_{R2}}$ and b_{R2} can be derived by using R_2.

$$\begin{aligned} f_{R2}(x_{R2}) &= \alpha_{f_{R2}} \cdot x_{R2} + b_{f_{R2}} = y_{R2} \\ b_{f_{R1}} &= 9.68 - (0.125925926 \cdot 10.2) \\ &= 8.39555555 \end{aligned}$$

The orthogonal function to $k'(x)$ which goes through R_2 is defined by $f_{R2}(x) = 0.1259 \cdot x + 8.3956$. The intersection of $f_{R2}(x)$ and $k'(x)$ will be denoted T'. The distance between T' and R_2 will be the orthogonal distance, and the x-coordinate.

$$\begin{aligned} f_{R2}(x) &= k'(x) \\ x &= \frac{b_{k'} - b_{f_{R2}}}{\alpha_{f_{R2}} - \alpha_{k'}} \\ x &= \frac{15.58 - 8.39555555}{0.125925926 - \frac{-1}{0.125925926}} \\ x &= 0.890585504 \end{aligned}$$

Given the x-coordinate the corresponding y-coordinate can be derived using either $f_{R2}(x)$ or $k'(x)$, as it will yield the same result.

$$\begin{aligned} f_{R2}(0.890585504) &= 0.125925926 \cdot 0.890585504 + 8.39555555 \\ f_{R2}(0.890585504) &= 8.50770335 \end{aligned}$$

The point of intersection between $f_{R2}(x)$ and $k'(x)$ is $T = (0.8906; 8.5077)$. The distance between T' and R_2 is calculated by Pythagoras.

$$\begin{aligned} |TR_2|^2 &= (\Delta Y)^2 + (\Delta X)^2 \\ |TR_2| &= \sqrt{(9.68 - 8.5077)^2 + (10.2 - 0.8906)^2} \\ |TR_2| &= 9.38292149 \end{aligned}$$

The x-coordinate for R_2 according to the fixed origin J_{Q2} is 9.3829.

Y-coordinate for R_2
The y-coordinate for R_2 is the orthogonal distance from $f'(x)$ to R_2. To calculate the distance, an orthogonal function to $f'(x)$ must be defined which goes through R_2.

$$f'(x) \perp k_{R2}(x) \Leftrightarrow \alpha_{k_{R2}} = \frac{-1}{0.125925926}$$

Given $\alpha_{k_{R2}}$; $b_{k_{R2}}$ can be derived using $k_{R2}(x)$.

$$\begin{aligned} k_{R2}(x_{R2}) &= \alpha_{k_{R2}} \cdot x_{R2} + b_{k_{R2}} = y_{R2} \\ b_{k_{R2}} &= 9.68 - (\frac{-1}{0.125925926} \cdot 10.2) \\ &= 90.68 \end{aligned}$$

The orthogonal function to $f'(x)$ which goes through R_2 is defined by $k_{R2}(x) = \frac{-1}{0.1259} \cdot x + 90.68$. The intersection of $k_{R2}(x)$ and $f'(x)$ will be denoted P'. The distance between P' and R_2 will be the orthogonal distance, and the y-coordinate.

$$\begin{aligned} k_{R2}(x) &= f'(x) \\ \alpha_{k_{R1}} \cdot x + b_{k_{R1}} &= \alpha_f \cdot x + b_f \\ x &= \frac{b_{f'} - b_{k_{R2}}}{\alpha_{k_{R2}} - \alpha_{f'}} \\ x &= \frac{1.86592593 - 90.68}{\frac{-1}{0.125925926} - 0.125925926} \\ x &= 11.0094145 \end{aligned}$$

Given the x-coordinate the corresponding y-coordinate can be derived using either $k_{R2}(x)$ or $f'(x)$, as it will yield the same result.

$$\begin{aligned} k_{R2}(11.0094145) &= \frac{-1}{0.125925926} \cdot 11.0094145 + 90.68 \\ k_{R2}(11.0094145) &= 3.25229667 \end{aligned}$$

The point of intersection between $k_{R2}(x)$ and $f'(x)$ is $P' = (11.0094; 3.2523)$. The distance between P' and R_2 is calculated by Pythagoras.

$$\begin{aligned} |PR_2|^2 &= (\Delta Y)^2 + (\Delta X)^2 \\ |PR_2| &= \sqrt{(9.68 - 3.2523)^2 + (10.2 - 11.0094)^2} \\ |PR_2| &= 6.4784609 \end{aligned}$$

We have now performed localization of R_2 in $Q2$. The coordinate set for R_2 according to J_{Q2} is $(9.3829; 6.4785)$.

B.1.2 Global scope

Given the position of R_1 and R_2, in their respective view, they can be positioned in the global scope, which is the final step of localization.

Positioning of R_1

Given the fixed origin of the environment is O, and that R_1 is localized in $Q1$ according to O, it is not necessary to perform further calculations. The position of R_1 in the environment is $(5.5415; 5.1879)$.

Positioning of R_2

In contrast to R_1, R_2 is placed in $Q2$, and positioned according to J_{Q2}. The displacement of J_{Q2} is horizontal, hence only the x-coordinate will be influenced. To position R_2 in the environment, the displacement must be compensated. As J_{Q1} and J_{Q2} are physically the same point in the environment, the displacement is the distance between O and J_{Q1}, which can be calculated by Pythagoras. Let $DispM.$ be the displacement.

$$\begin{aligned} DispM.^2 &= \Delta Y^2 + \Delta X^2 \\ DispM. &= \sqrt{\Delta Y^2 + \Delta X^2} \\ &= \sqrt{(x_{J_{Q1}} - x_{_O})^2 + (y_{J_{Q1}} - y_{_O})^2} \\ &= \sqrt{(11.6 - 2.4)^2 + (2.39 - 4.6)^2} \\ &= 9.4617176 \end{aligned}$$

The displacement must be taken into consideration, and as the displacement is horizontal, only the x-coordinate is influenced by the displacement. Let R_{2-O} be the position of R_2 according to fixed origin of the environment.

$$\begin{aligned} R_{2-O} &= (x_{R_{2O}}; y_{R_{2O}}) \\ y_{R_{2-O}} &= y_{_{R2}} = 6.4785 \\ x_{R_{2-O}} &= x_{_{R2}} + DispM. \\ &= 9.3829 + 9.4617176 \\ &= 18.8446176 \end{aligned}$$

The position of R_2 according to fixed origin of the environment is $(18.8446; 6.4785)$.

The readings of R_1 and R_2, which are made in an external Cartesian system, have been successfully converted into a position in the environment. Positioning using the Wii Remotes is now possible.

Appendix C

Wii Remote

There is no official documentation of the WiiYourself! library, the following documentation is made solely to clarify the methods used in the implementation. Please review the respective files in the *WiiYourself!_1.01a* folder for the full overview of the implementation.

C.1 *class* wiimote

This is a documentation of *class* `wiimote`. Please notice that this is not the full list of member function but only those we use in our implementation, see wiimote.cpp for details surrounding the other functions. The variable/return type is written in *itallic* and the successive word is the variable/method name. The *class* `wiimote` inherits from *struct* `wiimote_state`, which can be found in Appendix C.2.

+ *class* wiimote: wiimote_state

- To retrieve information of a Wii Remote, each Wii Remote must be associated with an instance of this class.

  + wiimote()

  - Is the default constructor. Each instance accesses the hid.dll in WinDDK, to access a Wii Remote.

  + *bool* Connect(*unsigned* wiimote_index, *bool* force_hidwrites)

  - The variable `force_hidwrites` is not implemented yet, and is always set to false.

  `wiimote_index` represents which Wii Remote to connect to. The values can either be an unsigned integer or 0xffffffff. The last picks a random Wii Remote in the Bluetooth stack.

  + *void* Disconnect()

  - Disconnect from the associated Wii Remote.

  + *void* SetReportType(*input_report* type, *bool* continuous)

  - The variable 'continuous' defines how the data read. If set to *true*, the library will perform a continuous read. This is however not implemented, and is set to *false*, which means an update requires the library to poll the data.

The report type can be one of the following types:

IN_BUTTONS

Only information regarding button events will be parsed form the report.

IN_BUTTONS_ACCEL

Both buttons events and accelerometer readings will be parsed.

IN_BUTTONS_ACCEL_IR

Buttons, accelerometer and IR information will be parsed.

IN_BUTTONS_ACCEL_EXT

Buttons, accelerometer and Extension[1] will be parsed.

IN_BUTTONS_ACCEL_IR_EXT

Readings regarding all aspects of the Wii Remote is parsed; this is however not implemented yet.

+ *void* SetLEDs(*BYTE* led_bits)

- Input can vary from [1-4], representing the LED lights on the Wii Remote.

[1] E.g. Nun chuck

C.2 *struct* wiimote_state

The *struct* wiimote_state holds the definition for the different aspects of the Wii Remote. We are only interested in those surrounding the IR diode, and have only included these in the documentation. To review the full list of definitions, see wiimote_state.h.

***struct* wiimote_state**

***struct* ir**

- Contains definition of the IR diodes,
  and the definition of the IR modes.

  ***struct* dot** [4]

  - The dot struct is a representation of an IR diode.
    Four dot structures are made, one for each diode the Wii Remote can fixate upon.

    ***bool* bVisible**

    - True if the IR diode visible, false otherwise.

    ***unsigned* RawX**

    - Stores the value of x-coordinate, read from the Wii Remote.

    ***unsigned* RawY**

    - Stores the value of y-coordinate, read form the Wii Remote.

    ***float* X**

    - Stores a numerical representation of RawX between [0-1].

    ***float* Y**

    - Stores a numerical representation of RawY between [0-1].

    ***int* Size**

    - Stores the cluster size of the IR diode. The size is determined by the number of pixels covered by the diode.

  ***enum* mode**

  - Four different modes types, which are used by the IR parser to encode each individual IR diode.[2]

    Off : No Report of IR.

    Basic : An IR diode is encoded using 2 bytes.

    Extended : Each IR diode is encoded using 3 bytes.

    Full : Each diode encoded using 8 bytes, but is not implemented.

[2] See Appendix C.3.

C.3 IR Parser

The following C++ method is the IR parser. It receives the 'report' as the argument and returns an integer value. We are however not interested in the return value, but how the method extracts the information regarding the IR. The report type *IN_BUTTON_ACCEL_IR* automatically sets *Internal.IR.Mode = EXTENDED*[3]. The mode defines how to encode each IR diode; the extraction of the IR diodes is performed on l. 38-55.

```
1  int wiimote::ParseIR (BYTE* buff)
2  {
3   if(Internal.IR.Mode == wiimote_state::ir::OFF)
4      return NO_CHANGE;
5   // take a copy of the existing IR state (so we can detect changes)
6   wiimote_state::ir prev_ir = Internal.IR;
7
8   // only updates the other values if the dots are visible (so that the last
9   //  valid values stay unmodified)
10  switch(Internal.IR.Mode)
11   {
12   case wiimote_state::ir::BASIC:
13    // 2 dots are encoded in 5 bytes, so read 2 at a time
14    for(unsigned step=0; step<2; step++)
15    {
16     ir::dot &dot0 = Internal.IR.Dot[step*2  ];
17     ir::dot &dot1 = Internal.IR.Dot[step*2+1];
18     const unsigned offs = 6 + (step*5); // 5 bytes for 2 dots
19
20     dot0.bVisible = !(buff[offs  ] == 0xff && buff[offs+1] == 0xff);
21     dot1.bVisible = !(buff[offs+3] == 0xff && buff[offs+4] == 0xff);
22
23     if(dot0.bVisible) {
24      dot0.RawX = buff[offs  ] | ((buff[offs+2] >> 4) & 0x03) << 8;;
25      dot0.RawY = buff[offs+1] | ((buff[offs+2] >> 6) & 0x03) << 8;;
26      dot0.X    = 1.f - (dot0.RawX / (float)wiimote_state::ir::MAX_RAW_X);
27      dot0.Y    =        (dot0.RawY / (float)wiimote_state::ir::MAX_RAW_Y);
28      }
29      if(dot1.bVisible) {
30      dot1.RawX = buff[offs+3] | ((buff[offs+2] >> 0) & 0x03) << 8;
31      dot1.RawY = buff[offs+4] | ((buff[offs+2] >> 2) & 0x03) << 8;
32      dot1.X    = 1.f - (dot1.RawX / (float)wiimote_state::ir::MAX_RAW_X);
33      dot1.Y    =        (dot1.RawY / (float)wiimote_state::ir::MAX_RAW_Y);
34      }
35     }
36    break;
37
38   case wiimote_state::ir::EXTENDED:
39    // each dot is encoded into 3 bytes
40    for(unsigned index=0; index<4; index++)
41     {
42     ir::dot &dot = Internal.IR.Dot[index];
43     const unsigned offs = 6 + (index * 3);
44
45     dot.bVisible = !(buff[offs  ]==0xff && buff[offs+1]==0xff &&
46                      buff[offs+2]==0xff);
47     if(dot.bVisible) {
48      dot.RawX = buff[offs  ] | ((buff[offs+2] >> 4) & 0x03) << 8;
49      dot.RawY = buff[offs+1] | ((buff[offs+2] >> 6) & 0x03) << 8;
50      dot.X    = 1.f - (dot1.RawX / (float)wiimote_state::ir::MAX_RAW_X);
51      dot.Y    =        (dot.RawY / (float)wiimote_state::ir::MAX_RAW_Y);
52      dot.Size = buff[offs+2] & 0x0f;
53     }
54    }
55   break;
56
57   case wiimote_state::ir::FULL:
58     _ASSERT(0); // not supported yet;
```

[3]Review l. 624-625 in wiimote.cpp

```
59      break;
60    }
61    return (memcmp(&prev_ir, &Internal.IR, sizeof(Internal.IR)) != 0)? IR_CHANGED : 0;
62  }
```

Listing C.1: Pseudo code of the pipe(client)