

**REINFORCEMENT LEARNING BASED
MOBILE ROBOT NAVIGATION**

by

Nihal ALTUNTAŞ

A thesis submitted to

the Graduate Institute of Sciences and Engineering

of

Fatih University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

March 2013
Istanbul, Turkey

APPROVAL PAGE

This is to certify that I have read this thesis written by Nihal ALTUNTAŞ and that in my opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Erkan İMAL
Thesis Supervisor

Assist. Prof. Dr. Nahit EMANET
Co-Supervisor

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Assist. Prof. Dr. Kadir TUFAN
Head of Department

Examining Committee Members

Prof. Dr. Erkan İMAL

Prof. Dr. Onur TOKER

Assist. Prof. Dr. Nahit EMANET

Assist. Prof. Dr. Kadir TUFAN

Dr. İhsan Haluk AKIN

It is approved that this thesis has been written in compliance with the formatting rules laid down by the Graduate Institute of Sciences and Engineering.

Assoc. Prof. Dr. Nurullah ARSLAN
Director

March 2013

REINFORCEMENT LEARNING BASED MOBILE ROBOT NAVIGATION

Nihal ALTUNTAŞ

M. S. Thesis – Computer Engineering
March 2013

Thesis Supervisor: Prof. Dr. Erkan İMAL

Co-Supervisor: Assist. Prof. Dr. Nahit EMANET

ABSTRACT

For last decades, reinforcement learning (RL) has been widely used in different research fields starting with psychology. Unfeasibility of sampling all possibilities for continuous-state problems such as mobile robot navigation and absence of an explicit teacher make RL algorithms be preferred for the supervised learning in machine learning (ML) area as the optimal control problem has become a popular subject of search.

In this thesis, after examining base logic and different solution classes of the RL algorithms, literature is reviewed on applications using various RL methods. As a result of the review, a system is proposed to solve mobile robot navigation by opting for the most popular two RL algorithms, Sarsa(λ) and Q(λ). The system, developed in MATLAB, can guide the mobile robot to a desired goal by avoiding obstacles with a high success rate on both simulated and real environments. Additionally, it is possible to observe effects of initial parameters used by the RL methods, e.g., λ , on learning, and also to make comparisons between the performances of Sarsa(λ) and Q(λ) algorithms.

Keywords: Reinforcement learning, autonomous navigation, mobile robot, obstacle avoidance, temporal-difference method, Sarsa, Q-learning, eligibility traces.

TAKVİYELİ ÖĞRENME TABANLI GEZGİN ROBOT NAVİGASYONU

Nihal ALTUNTAŞ

Yüksek Lisans Tezi – Bilgisayar Mühendisliği
Mart 2013

Tez Danışmanı: Prof. Dr. Erkan İMAL

Eş-Danışman: Yrd. Doç. Dr. Nahit EMANET

ÖZ

Son yıllarda, psikoloji alanından başlayarak takviyeli öğrenme (RL) değişik alanlarda yaygın olarak kullanılmaktadır. Optimal kontrolün popüler bir araştırma konusu haline gelmesiyle, gezgin robot navigasyonu gibi sürekli-durum problemleri için bütün ihtimallerin örneklendirilmesinin imkânsızlığı ve belirgin bir öğretmenin eksikliği makine öğrenmesi (ML) alanında RL algoritmalarının öğretmenli öğrenmeye göre daha çok tercih edilmesini sağladı.

Bu tezde RL algoritmalarının temel mantığı ve farklı çözüm sınıfları detaylı bir şekilde incelendikten sonra değişik RL metotlarını kullanan uygulamalara yönelik literatür taraması yapılmaktadır. Araştırma sonucunda en yaygın iki RL metodu, Sarsa(λ) ve Q(λ), seçilerek gezgin robot navigasyonu probleminin çözümü için bir sistem önerilmektedir. MATLAB ile geliştirilen sistem, hem simülasyon hem gerçek ortamda, yüksek bir başarı oranıyla gezgin robotu engellerden kaçırarak istenen hedefe yönlendirebilmektedir. Ayrıca, sistem sayesinde RL metotlarında kullanılan başlangıç parametrelerinin, örneğin λ , öğrenmeye olan etkisini gözlemlemek ve Sarsa(λ) ve Q(λ) algoritmalarının performanslarında karşılaştırmalar yapmak da mümkündür.

Anahtar Kelimeler: Takviyeli öğrenme, otonom navigasyon, gezgin robot, engellerden kaçınma, geçici-fark metodu, Sarsa, Q-öğrenme, nitelik izleri.

To my parents

ACKNOWLEDGEMENT

I express my appreciations to Prof. Dr. Erkan İMAL for giving me the opportunity to study on this research and for his motivations and encouragements through this thesis. I also wish to express my sincere gratitude to Assist. Prof. Dr. Nahit EMANET for his constructive comments and valuable suggestions.

I also would like to thank my family that this study would not have been completed successfully without their understanding and patience. Besides, I am thankful to my friends for their assistance and support during the research.

Finally, I gratefully acknowledge the financial support provided by the Scientific Research Fund of Fatih University under the project number P50061202_B and the Scientific and Technological Research Council of Turkey (TÜBİTAK) under National Scholarship Programme for M.Sc. Students.

TABLE OF CONTENTS

ABSTRACT.....	iii
ÖZ	iv
DEDICATION.....	v
ACKNOWLEDGEMENT	vi
TABLE OF CONTENTS.....	vii
LIST OF TABLES	ix
LIST OF FIGURES	x
LIST OF SYMBOLS AND ABBREVIATIONS	xii
CHAPTER 1 INTRODUCTION	1
1.1 Background.....	2
1.2 Problem Definition	3
1.3 Decision Making Algorithms	4
1.3.1 Markov Decision Process (MDP)	6
1.3.2 Partially Observable MDP (POMDP).....	7
1.3.3 Reinforcement Learning (RL).....	9
1.3.4 Solution Methods for The Algorithms	10
1.3.4.1 Dynamic Programming	11
1.3.4.2 Monte Carlo Method	13
1.3.4.3 Temporal-Difference Learning.....	14
CHAPTER 2 LITERATURE REVIEW	17
2.1 Mobile Robot Applications	21
2.2 Vision Based Applications	24
CHAPTER 3 IMPLEMENTATION.....	31
3.1 Proposed System.....	31
3.1.1 Graphical User Interface (GUI).....	32
3.1.2 Navigator.....	33
3.1.3 Reinforcement Learning Control	33

3.1.3.1	Action Set	33
3.1.3.2	Reward Function	34
3.1.3.3	State Set	34
3.1.4	Simulator	35
3.1.5	Simultaneously Localization and Mapping (SLAM)	35
3.1.6	Utils	36
3.2	Eligibility Traces	36
3.2.1	Sarsa(λ)	38
3.2.2	Q(λ)	39
3.3	Software interface of the System	40
CHAPTER 4 EXPERIMENTS AND RESULTS		43
4.1	Simulated Experiments and Results	43
4.2	Experimental Platform – The Robotino [®]	46
4.2.1	PID Control in Robotino [®]	48
4.2.2	Motion and Odometry Accuracy	49
4.2.3	Experimental Results	53
CHAPTER 5 CONCLUSION		55
5.1	Future Work	56
REFERENCES		57
APPENDIX A ROBOTINO		60
A.1	Drive Unit	60
A.2	Sensors	61
A.3	Controller Board	62
A.4	Software	63

LIST OF TABLES

TABLE

1.1	Comparison of decision making algorithms.....	10
1.2	Comparison for classes of methods to solve decision making algorithms.....	16
2.1	The parameters used by RL system (Shaker et al., 2010)	29
4.1	Single robot motion through x -axis for distance d and linear velocity v	50
4.2	Single robot rotation for amount of θ and angular velocity w	50
4.3	Single robot rotation for amount of $-\theta$ and angular velocity $-w$	51
4.4	Results of empirical cycles shown in Figure 4.5.....	52

LIST OF FIGURES

FIGURE

1.1	The agent (Robotino [®]) – environment interaction through sensors and actuators..	3
1.2	Transition graph for an MDP model with two states and three actions	7
1.3	Generalized policy iteration (GPI) cycle: policy and value interact until they are optimal (Sutton and Barto, 2005)	13
2.1	GUI for application of Mountain Car problem by Anderson (2012).....	18
2.2	$Ex\langle\alpha\rangle(\lambda)$ algorithm results ($k = 8$; $\lambda = 0.9$; $\varepsilon = 0$; $\alpha = 1.2$; $\gamma = 1.0$) for the Cart-Pole problem	20
2.3	The neural network structure used by Maček et al. (2002)	21
2.4	An example simulation for study of Maček et al. (2002)	22
2.5	(a) Pekee robot and (b) its infrared sensors clustered by Khriji et al. (2011).....	23
2.6	Example results in (a) simulated and (b) real platforms (Khriji et al., 2011)	24
2.7	A sample performance of the real robot used by Zhumatiy et al. (2006)	25
2.8	The system architecture used by Busquets et al. (2002).....	26
2.9	An example simulation of (a) environment and images perceived (b) right and (c) left stereo cameras (Zhu and Levinson, 2001).....	28
2.10	Image sequence of a simulated landing (Shaker et al., 2010).....	30
3.1	The block diagram of the proposed system	32
3.2	Variables of the State Set module are defined by robot distance and orientation .	35
3.3	The pseudocode for Sarsa(λ) algorithm used by the proposed system	38
3.4	The pseudocode for Q(λ) algorithm used by the proposed system.....	39
3.5	The GUI of the proposed system named RLC.....	41
3.6	The interface for environment configuration of the system.	42
4.1	Simulated Sarsa(λ) results for three different obstacle options	44
4.2	Simulated Q(λ) results for three different obstacle options	45
4.3	Degree of Freedom (DoF) of the Robotino [®]	47
4.4	PID controller loops of motors of the Robotino [®] are updated by EA09 board	48

4.5	One empirical cycle: (1) go forward for distance d , (2) rotate for 180° , (3) go forward for distance d , (4) rotate for -180°	52
4.6	The Robotino [®] can reach the target by avoiding obstacles in the test phase	53
4.7	The interface during the test execution in Figure 4.6	54
A.1	Drive unit with (1) motor, (2) encoder, (3) omnidirectional wheel, (4) planetary gear, (5) toothed belt	61

LIST OF SYMSBOLS AND ABBREVIATIONS

SYMBOL/ABBREVIATION

AI	Artificial Intelligence
API	Application Programming Interface
DoF	Degree of Freedom
DP	Dynamic Programming
EKF	Extended Kalman Filter
GUI	Graphical User Interface
GPI	Generalized Policy Iteration
I/O	Input/Output
k -NN	k -Nearest Neighbors
LSPI	Least-Squares Policy Iteration
MC	Monte Carlo
MDP	Markov Decision Process
ML	Machine Learning
NSM	Nearest Sequence Memory
PBF	Polynomial Basis Function
PC-NSM	Piecewise Continuous Nearest Sequence Memory
PEAS	Performance, Environment, Actuators, Sensors
PID	Proportional, Integral, Derivative
POMDP	Partially Observable Markov Decision Process
RBF	Radial Basis Function
RL	Reinforcement Learning
RLC	Reinforcement Learning Control
SIFT	Scale Invariant Feature Transform
SLAM	Simultaneously Localization and Mapping
SURF	Speeded Up Robust Feature
TD	Temporal-Difference

UAV	Unmanned Aerial Vehicle
VA	Vision Agent
VBAN	Vision-Based Autonomous Navigator
W-LAN	Wireless Local Area Network

CHAPTER 1

INTRODUCTION

Through developing technology, people started to prefer machines instead of human work in order to increase productivity. At the beginning, machines were only used to automate works not requiring intelligence but invention of computers made people think about machine learning (ML). Today, artificial intelligence (AI) is still a subject of study to provide machines with the learning ability. Besides, not only intelligence, but also mobility gets importance as the use of machines becomes more prevalent.

First of all, it is important to understand the nature of learning in order to achieve the goal of intelligent machines. Although there are great deals of algorithms that are developed as supervised and unsupervised learning methods in ML field to enable machines learn, the fundamental idea is learning from interaction. While infants are learning how to walk or talk, there is no explicit teacher, but they have ability of connecting their environment by observations. Observing is the main source of knowledge for humans about their environment. It is important to establish a cause-effect relationship, and to be aware of results of actions for learning from interaction. When people experience the results of their behaviors, they get a chance to improve their actions. Reinforcement learning (RL) brings such approach of learning from interaction for enabling machines to learn and to obtain interaction ability, various sensors providing capabilities of human sense organs are mounted on machines such as infrared, sonar, inductive, diffuse sensors and so on.

In addition, it is an undoubted fact that the wider the area in which the machines are used is, the more features they need to have. Therefore, the intelligence and mobility

have also become another essential capability for machines. It is possible to use machines for various purposes by providing them with mobility via wheels. Some of examples are exploring hazardous places or guiding people in specific environments.

1.1 BACKGROUND

This section gives some expressions for terminology used in the thesis, and explains which features the system needs to have in order to be called as rational and so successful.

The intelligent agent which is the learner and decision-maker is the mobile robot called Robotino[®], in this case (see Appendix A). Everything with which the agent interacts is called the environment. The agent uses its sensors to gather information from the environment and actuators to respond to it as illustrated in Figure 1.1. The Robotino[®] perceives the environment through its camera and acts according to those inputs it captures. The robot selects an action depending on entire percept sequence which is the complete history of inputs it perceives, and the environment responds to that action presenting new situation to the robot to perceive more. An agent function which maps any percept sequence to an action is necessary to describe robot behaviors. An agent program runs on the Robotino[®] for implementation of the agent function.

The robot is required to select the right action that maximizes expected performance for each percept sequence to be called rational. Here, it is necessary to measure the performance, and in this case, the closer the robot approaches to the target, the more it is rewarded as a result. While providing rationality, it is also necessary to be autonomous by using the percept sequence obtained by information gathering instead of any prior knowledge. To accept the agent successful, information gathering is not enough, it is also necessary for the agent to learn as much as possible from what it perceives and to exhibit better performance as it gains experience.

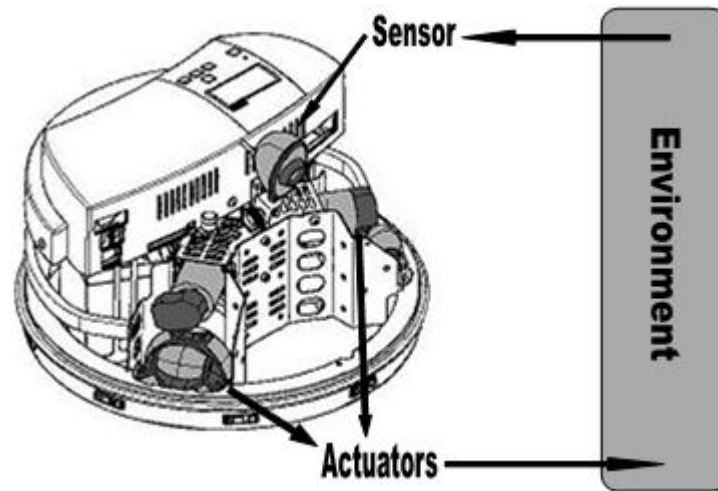


Figure 1.1 The agent (Robotino[®]) – environment interaction through sensors and actuators.

1.2 PROBLEM DEFINITION

In addition to simply explaining the system as enabling the mobile robot to navigate autonomously by using RL based control, it is necessary to examine all details of the system requirements for a successful construction. To make this examination, it is essential to specify the performance measure embodying criterion for success, the environment in which the robot is, and actuators and sensors of the robot – also called as PEAS (Performance, Environment, Actuators, Sensors) description (Russell and Norvig, 2003).

First of all, in terms of performance, the system should have some qualities like getting to the correct destination, minimizing the trip time, and maximizing the safety by avoiding obstacles. Clearly, while providing those qualities, there are some tradeoffs making the system complicated.

The second question is how the environment should be. There are some different criteria to specify the environment. Foremost, it is partially observable meaning that the robot cannot get the complete state of the environment because of noisy or inaccurate sensors and missing parts of a state from sensory data. Partially observability makes the environment stochastic; therefore, next state of the robot depends on not only the current state and some other variables such as inaccurate results of actions also affect

what the next state is. Besides, the environment is sequential; in other words, the current action of the robot influences all future actions. Although the environment itself does not change over time, the performance of robot is dynamic; thus, it is said to be semi-dynamic. Additionally, since the robot can be in any coordinates at any time, the environment is continuous. Finally, because there is only one robot as intelligent agent, it is a single agent environment.

The last two criteria of PEAS description are the actuators and sensors of the robot. Motors of omni-directional wheels of the robot are the actuators to provide the robot with actions that are decided by analyzing the inputs obtained from sensors, i.e., a single camera, odometer and bumper of the robot.

After examining the problem, it can be said that the system has a complex structure, and AI is necessary for the robot to learn how to navigate autonomously. Since it is not possible to design all possible situations to teach the agent, a supervised learning algorithm cannot be used. In the next section, it is explained in details why reinforcement learning is opted for the system.

1.3 DECISION MAKING ALGORITHMS

Besides the agent and the environment, it is necessary to define some components like transition model, reward function, policy and value function in order to explain the algorithms.

A transition model specifies the outcome probabilities for each action in each possible state. It means that the transition model predicts how the environment will respond to the agent's actions. Also since the system is stochastic, for a given state and action, the model has several next states and next reward with different probabilities. In this thesis, $P_a(s, s')$ denotes the probability of reaching state s' by choosing action a in state s .

A reward function maps each state or state-action pair to a single number called reward. It defines good and bad for just instant state not considering future states. The goal is to maximize total amount of reward received by the agent, and the reward

function must be unchangeable by the agent. They are accepted as a part of the environment and so computed there, not inside the agent in order to prevent the agent to change rewards. Even if the goal is related to parts of the robot like position of the robot arms, those parts are considered to be outside of the agent whereas they are inside of the robot. Since the main purpose is maximizing total reward, it is significant to determine the rewards accurately. The agent should be rewarded only if it achieves real goal. Subgoals should not be determined. Otherwise it becomes possible for the agent to achieve subgoals without reaching the actual goal. Rewards are expected to show the agent what to achieve not how to achieve (Sutton and Barto, 2005). The reward received by the agent is represented as $R_a(s, s')$ in the thesis, for transition from state s to state s' applying action a .

A policy defines how the agent acts according to perceived states of the environment. The policy itself is enough to determine the behavior, by determining which action to choose in each possible state. Also it can be changed to receive better rewards. Complexity of the policy depends on the systems of the agent. For some systems, it can be a simple function or lookup table, but for others, it can use complex supervised learning algorithms which need great computation. π denotes the policy and $\pi(s)$ represents the chosen action for state s according to policy π , $\pi_a(s)$ is the probability of choosing action a in state s for policy π and π^* denotes optimal policy.

The quality of a policy is determined by the expected value of all possible states, and there is always at least one policy whose expected value is greater than or equal to that of other policies for all states. The best policy having highest expected value is called as optimal policy, π^* . The closer to optimal policy used by the agent is, the closer to the goal the agent approaches.

A value function indicates what is good in the long run while the reward function specifies short-term benefit. The value of a state is total reward of expected state sequences starting from that state for the executed policy. Therefore, value of a state depends on the policy, and a low-rewarded state can have a high value since it is followed by states having high rewards or vice versa. Thus, values are more important than rewards while making decisions and choosing actions. However, it is much harder to determine values of states than their rewards. Whereas rewards are given to the environment, the values need to be recalculated after each gathered information is

interpreted. This makes the value iteration method the most essential component in the entire algorithm. V^π and V^* denote value function for policy π and optimal value function, respectively. $V^\pi(s)$ meaning estimated value of a state for executed policy is defined formally by Eq. (1.1).

$$V^\pi(s) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, s_{t+k+1}) \mid s_t = s \right\} \quad (1.1)$$

where $E_\pi\{\}$ is the expected value for policy π and γ is discount factor. The discount factor describes the effect of future rewards over value of current state, and is a number between 0 and 1. When discount factor is equal to zero, then the agent concerns to maximize only the next reward not future rewards, and the system becomes episodic not sequential. On the other hand, when it approaches to one, then future rewards get more important to decide next action for the agent.

Similarly $Q^\pi(s, a)$ denotes action-value function, also called Q-value, for policy π which means the expected value for selecting action a in state s under policy π , and Q^* is the optimal action-value function. Eq. (1.2) expresses the definition of the Q-value used for reinforcement learning algorithm explained in following subsections.

$$Q^\pi(s, a) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, s_{t+k+1}) \mid s_t = s, a_t = a \right\} \quad (1.2)$$

Finally, it is also an important question whether the system is built over finite horizon or infinite horizon. Finite horizon means there is a fixed number, N , of chance to reach the goal. Thus, for finite horizon, action decision in a state also depends on time. Eqs. (1.1) and (1.2) are defined over potentially infinite horizon.

1.3.1 Markov Decision Process (MDP)

Markov Decision Process (MDP) is the base algorithm for sequential decision problems. And there are some assumptions for MDP to work. For instance, some prior knowledge about the environment is required like transition model, and reward function meaning that the agent knows not only all possible states and actions, but also probability distribution over state – action pairs. This distribution relies only on current

state not preceding ones. Each state signal retains relevant information in order for deciding which action to select. Therefore, it is said MDP satisfies Markov property (Markov, 1954) referring memorylessness. Furthermore, the environment is fully observable allowing the agent to know in which state it exactly is, and to make decisions according to that information. Using current state s as parameter, the system defines action a which is $\pi(s)$ by the executed policy and then passes to the next state s' through the transition model.

Figure 1.2 displays example of an MDP system with two states and three actions. Probability distribution called transition model and reward function of action-state pairs are also shown in Figure 1.2.

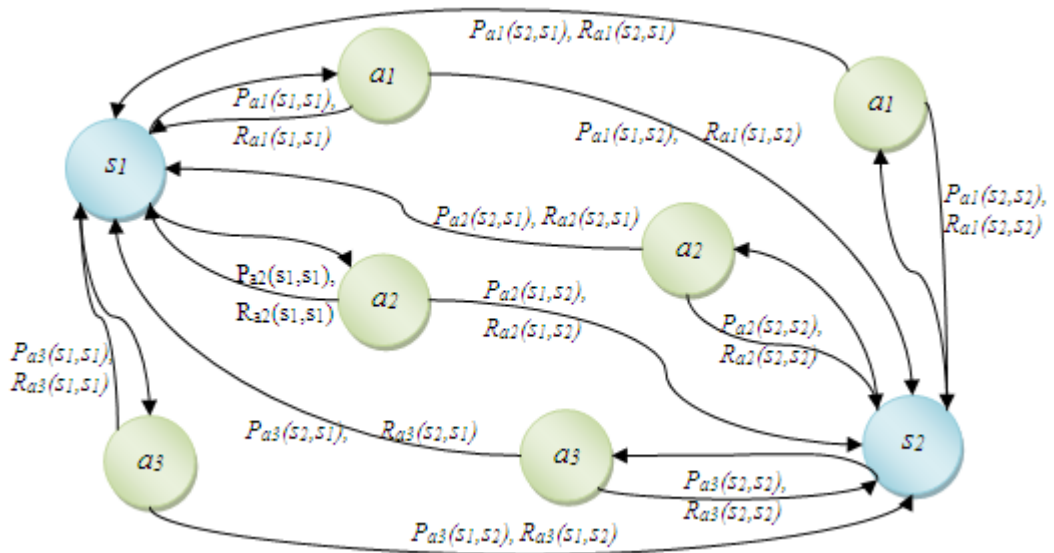


Figure 1.2 Transition graph for an MDP model with two states and three actions.

1.3.2 Partially Observable MDP (POMDP)

Unlike the MDP algorithm assumes, in the real world, it is not practical for the agent to fully observe whole environment in which it is. Because of this partial observability, the agent cannot know the current state but can make a guess about the state depending on its observations. By having all elements of MDP systems like transition model and reward function, in order to use those observations, partially

observable MDP (POMDP) systems also need to have observation model, $O(s, o)$ specifying the probability of observation o in state s .

As mentioned above, the agent needs a guess about where it is. Therefore it has a belief state representing its current belief about the possible states it might be in. POMDP systems use belief state as probability distribution over all possible states to detect which state the agent is likely in. How much the agent knows where it is determines how well it selects future actions. Thus, it updates belief state at the end of POMDP decision cycle. The decision cycle is that depending on the belief state, the agent decides the action $a = \pi(b)$, and passes to the next state s' , then new belief state b' is calculated as seen in Eq. (1.3) by using observation o in state s' and previous belief state b .

$$b'(s') = \alpha O(s', o) \sum_s P_a(s, s') b(s) \quad (1.3)$$

$$P_a(b, o) = \sum_s O(s', o) \sum_s P_a(s, s') b(s) \quad (1.4)$$

where $\alpha = 1 / P_a(b, o)$ is normalizing constant making the belief state sum to 1 and $b(s)$ is probability of agent's being in state s by belief state b . $P_a(b, o)$ seen in Eq. (1.4) denotes the probability of observation o for executing action a in belief state b .

Solving the POMDP problem on a real state space can be reduced to MDP problem on a corresponding belief state space by using $\tau_a(b, b')$ and $\rho_a(b)$ instead of $P_a(s, s')$ and $R_a(s)$, respectively (Russell and Norvig, 2003). Eq. (1.5) defines $\tau_a(b, b')$ denoting transition function in belief state space and Eq. (1.6) defines the reward function, $\rho_a(b)$, for belief states.

$$\tau_a(b, b') = \sum_o P(b'|o, a, b) \sum_{s'} O(s', o) \sum_s P_a(s, s') b(s) \quad (1.5)$$

$$\rho_a(b) = \sum_s b(s) R_a(s) \quad (1.6)$$

where $P(b'|o, a, b) = 1$ if b' is equal to the next belief state of the agent after POMDP decision cycle and $P(b'|o, a, b) = 0$ otherwise.

1.3.3 Reinforcement Learning (RL)

Although POMDP enhances MDP to be applicable for real problems by adding partial observability, it still requires prior knowledge about the environment like complete transition model and the reward function, yet there is no clue for either in this case. Reinforcement learning aims the agent to learn how to behave when it is placed in an unknown environment.

Not knowing the model in reinforcement learning algorithm causes a trade-off between exploration and exploitation. Selecting the action with greatest estimated value means that the agent exploits its current knowledge. Instead, selecting one of the other actions indicates that the agent explores in order to improve its estimate for values of those actions. Exploitation maximizes the reward in the short run, yet it does not guarantee to maximize the total reward in the long run. On the other hand, while exploration reduces short-term benefit, it produces greater reward in the long run because after the agent has explored better actions, it can start to exploit them. The agent can neither explore nor exploit exclusively without failing at the task, and it cannot both explore and exploit in one selection. Therefore, it is vital to balance between exploration and exploitation for converging to the optimal policy. The most common method to balance this trade-off is ϵ -greedy method. In this method, the action with greatest estimated value is called greedy action, and the agent usually exploits its current knowledge by selecting the greedy action but there is also a chance of probability ϵ for the agent to explore by selecting one of non-greedy actions randomly. This type of action selection is called as ϵ -greedy method (Sutton and Barto, 2005).

Reinforcement learning can be examined as passive or active depending on whether it uses a fixed policy or not. Passive learning aims only to learn how good the fixed policy is. The only challenge is that the agent knows neither transition model nor reward function since the environment is unknown. The agent executes a set of trials using the fixed policy to calculate estimated value of each state. As distinct from passive learning, active learning aims also to learn how to act in the environment. Since the agent does not know either the environment or the policy, it has to learn transition

model in order to find an optimal policy. But in this case, for each state there are more than one choice of actions because the policy is not fixed (Russell and Norvig, 2003). Here the balance between exploration and exploitation gets importance to decide which action to choose.

Although all algorithms covered so far can be used in order to learn complex problems, not all are suitable for the proposed system. Table 1.1 shows a comparison between the algorithms and their suitability for the proposed system. Since the intelligent agent, Robotino[®], has capacity to observe partially and has no clue about either of the environment and the policy, only active reinforcement learning is suitable for the system.

Table 1.1 Comparison of decision making algorithms.

	MDP	POMDP	REINFORCEMENT LEARNING	
			PASSIVE	ACTIVE
Environment	Known	Known	Unknown	Unknown
Observability	Full	Partial	Partial	Partial
Policy	Unknown	Unknown	Known	Unknown
Suitability for the System	NO	NO	NO	YES

1.3.4 Solution Methods for the Algorithms

In the following subsections, three fundamental classes of methods developed for solving complex decision making algorithms are described. Those classes of solution methods are dynamic programming, Monte Carlo methods and temporal-difference learning. Each class of methods has its own pros and cons.

1.3.4.1 Dynamic Programming

Dynamic programming (DP) algorithms are based on an assumption of complete environment model, in other words, it is a model-based solution class. Since DP requires considerable a prior knowledge, this solution class is proper only for MDP problems not for RL problems. DP algorithms are generally unpractical because of not only their perfect model assumption but also their great computational expense. However dynamic programming is still important theoretically since it provides an important foundation for other solution methods. It can be said that other types of methods have the purpose of achieving the same effect as DP with less computation and without complete model of environment. Many of the solution methods are based on DP and its algorithms such as policy iteration and value iteration.

Policy iteration consists of two interacting processes, policy evaluation and policy improvement. Policy evaluation part calculates the consistent value function for the current policy while policy improvement part finds greedy policy for the value function. After policy evaluation, the policy becomes no longer greedy for the changed value function, and policy improvement makes the value function inconsistent for the greedy policy. However, after some point, both evaluation and improvement processes stabilize meaning that value the function and the policy become optimal.

During the policy evaluation process, dynamic programming basically calculates value functions using their fundamental property of satisfying recursive relationships. Bellman equation (Bellman, 1957a) is used to update values of each state iteratively as seen in Eq. (1.7). This algorithm is also called iterative policy evaluation.

$$V_{k+1}(s) = \sum_a \pi(s, a) \sum_{s'} P_a(s, s') [R_a(s, s') + \gamma V_k(s')] \quad (1.7)$$

V_k converges to V^π as $k \rightarrow \infty$, and γ is discount factor as used in Eq. (1.1). Bellman equation is the most applied equation in DP to estimate value function.

Policy improvement is the other step in policy iteration. The main idea is to find if there is a better policy than the current one. In state s , if selecting an action a and afterwards following π has greater value than selecting $\pi(s)$ as seen in Eq. (1.8), then the

policy is updated for $\pi(s)$. The updated policy, π' , is exactly the same as the old policy, π , except $\pi'(s) = a \neq \pi(s)$. Eq. (1.9) expresses the idea for policy improvement.

$$Q^\pi(s, a) \geq V^\pi(s) \quad (1.8)$$

$$\pi'(s) = \underset{a}{\operatorname{argmax}} Q^\pi(s, a) \quad (1.9)$$

where argmax_a denotes the action a at which $Q^\pi(s, a)$ is maximized. In short, making the updated π' greedy for value function of the original policy π is called policy improvement.

One deficiency of policy iteration is that in each of its iterations, policy evaluation also involves iteration, Eq. (1.7), requiring lots of computation and memory. Value iteration combines evaluation and improvement steps of policy iteration in one step to improve efficiency. Eq. (1.10) expresses value iteration that is an update rule turned from Bellman optimality equation.

$$V_{k+1}(s) = \max_a \sum_{s'} P_a(s, s') [R_a(s, s') + \gamma V_k(s')] \quad (1.10)$$

Furthermore, during policy iteration, letting policy evaluation and policy improvement processes interact with each other to find a joint solution is termed generalized policy iteration (GPI), and policy evaluation and policy improvement are also known as prediction problem and control problem, respectively. The following subsections explain different approaches to solve those problems. Figure 1.3 illustrates schema for GPI.

As seen from Eqs. (1.7) and (1.10), estimation update of value for a state depends on value estimate of its successor states. This is called bootstrapping. Besides DP, there are some other solution methods that bootstrap even though they do not require a complete model of the environment.

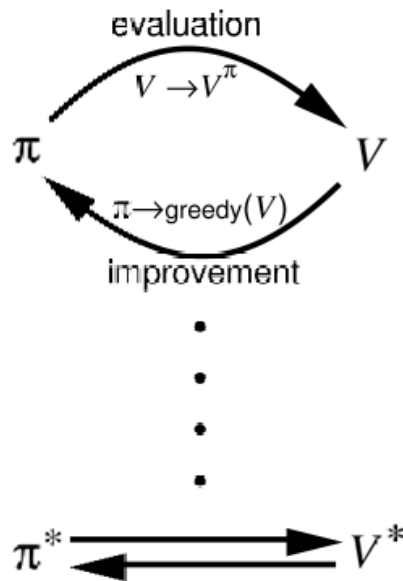


Figure 1.3 Generalized policy iteration (GPI) cycle: policy and value interact until they are optimal (Sutton and Barto, 2005).

1.3.4.2 Monte Carlo Methods

Monte Carlo methods use experience from interaction with the environment. Instead of assuming complete knowledge of the environment, Monte Carlo methods average sample values obtained from experience that consists of episodes. The only assumption is that each episode terminates eventually whichever action is selected. Estimation update is done after termination of each episode, not after each step.

The main idea introduced by DP, i.e., generalized policy iteration (GPI), is used by Monte Carlo as mentioned in previous subsection. The only difference is that update of value estimation depends on experience instead of estimation of another state. It means Monte Carlo does not bootstrap for prediction problem. Therefore, computational complexity of estimating value for a state does not depend on number of states.

During policy evaluation process, Monte Carlo (MC) uses sample values obtained after state visits in episodes to estimate expected value. The every-visit MC method averages values of all visits for each state in episodes while the first-visit MC method uses only first visits of states in episodes for average. Both MC methods converge to expected value as more episodes are executed.

When complete model of the environment is known, estimating state-value function, $V^\pi(s)$, is enough to define policy. It can be decided which action is selected for state s by using model and $V^\pi(s)$. However, there is no prior knowledge about environment in order to make selection, Monte Carlo methods must explicitly estimate how good each action is for state s . Therefore action-value function, $Q^\pi(s, a)$, is estimated during policy evaluation, and this makes the environment model unnecessary while greedy policy is being computed by using Eq. (1.9) in policy improvement process. Yet, it is important to ensure the all state-action pairs are visited and estimated to compare all possible action-values for each state. This is exploitation/exploration trade-off problem. To solve this problem, there are two approaches, *on-policy* methods and *off-policy* methods. *On-policy* methods evaluate and improve same policy used for decisions in experiences. To ensure that all actions are selected to learn better choices ϵ -greedy policies are used as mentioned in subsection 1.3.3. On the other hand, *off-policy* methods use two policies, behavior policy and estimation policy. Behavior policy is used to make decision, and it needs to ensure all actions have a probability to be selected to explore all possibilities. Estimation policy is the policy evaluated and improved, and since it does not affect decisions, this policy can be totally greedy.

1.3.4.3 Temporal-Difference Learning

Temporal-Difference (TD) learning divides the learning problem into a prediction problem and a control problem as described by GPI. As solution for these problems, TD has an approach combining DP and Monte Carlo methods explained in the previous subsections. TD methods can learn from experience without any need of model so TD is model-free like Monte Carlo. However, it is not necessary for TD to wait until the end of episodes for update of value functions since it bootstraps like DP meaning that TD updates value estimate by using another value estimate as basis. While Monte Carlo uses actual values to update estimates, TD uses immediate rewards and estimates of successor states for value estimation as seen in Eq. (1.11) for policy evaluation process of GPI cycle.

$$V(s) \leftarrow V(s) + \alpha [R_a(s, s') + \gamma V(s') - V(s)] \quad (1.11)$$

where α is step-size parameter, also known as learning rate and differs between 0 and 1. If α is constant and small enough, estimated value converges to $V(s)$ keeping varying in small amounts and if α is reduced properly, $V(s)$ can converge to $V^\pi(s)$ completely. Another advantage of TD over Monte Carlo is that if there is a limited amount of experience which is not sufficient to generalize whole problem, Monte Carlo can only find an optimal policy in limited way while TD can converge to an optimal policy representing the problem completely. Additionally, since TD does not use a model of the environment, it needs to handle action selection for control part of the GPI cycle, and state-value function, $V^\pi(s)$, is insufficient to make decisions thus TD needs to estimate action-value functions, $Q^\pi(s, a)$, like Monte Carlo. Besides, two different approaches are applied to deal with exploration/exploitation trade-off; *on-policy* and *off-policy*.

Although there are some other TD methods using *on-policy* approach, the most common *on-policy* TD method is Sarsa. As in all *on-policy* methods, Sarsa uses one policy for evaluation and decision making and uses ε -greedy method to ensure all actions possible. Eq. (1.12) defines update rule of Sarsa algorithm using quintuple parameters, (s, a, r, s', a') , from which Sarsa name comes. For state s , the agent decides and performs action a , then it observes an immediate reward r and new state s' , and decides another action a' for state s' .

$$Q(s, a) \leftarrow Q(s, a) + \alpha [R_a(s, s') + \gamma Q(s', a') - Q(s, a)] \quad (1.12)$$

The most popular TD algorithm is Q-learning which is an *off-policy* method and first introduced by (Watkins, 1989). Eq. (1.13) defines Q-learning algorithm that updates action-value function (Q-function) by greedy method, independent of behavior policy after selecting action a for state s , and receiving reward r and next state s' .

$$Q(s, a) \leftarrow Q(s, a) + \alpha [R_a(s, s') + \gamma \max_a Q(s', a) - Q(s, a)] \quad (1.13)$$

All classes of solution methods mentioned above are described as GPI. The point they differ from each other is that they have different approaches to the prediction

problem and the control problem. Table 1.2 compares the solution classes developed for decision making algorithms.

Table 1.2 Comparison for classes of methods to solve decision making algorithms.

	Dynamic Programming	Monte Carlo Methods	TD Learning	
			Sarsa	Q-Learning
Model-based Model-free	Model-based	Model-free	Model-free	Model-free
Require Experience	No	Yes	Yes	Yes
Bootstrap	Yes	No	Yes	Yes
Estimation Update Period	Step-by-step	Episode-by-episode	Step-by-step	Step-by-step
On-policy Off-policy	—	Both	On-policy	Off-policy

CHAPTER 2

LITERATURE REVIEW

The term reinforcement is firstly used in psychology which concerns the idea of learning by trial and error in 1920s; thereafter, this idea became popular in computer science. In 1957, Bellman introduced dynamic programming providing optimal control and then Markov decision processes (MDP) (Bellman, 1957b). Although dynamic programming is a solution for discrete stochastic MDPs, it requires expensive computations growing exponentially as number of state increases. Therefore temporal-difference brought a novel aspect to reinforcement learning, and in 1989, Q-learning explored by Watkins was an important breakthrough in AI field. Finally Sarsa algorithm is introduced by Rummery and Niranjan, as “modified Q-learning” in 1994.

While RL techniques are developed, they started to be preferred as learning algorithms since they are more feasible and applicable than other techniques by not needing a prior knowledge. Some base studies using reinforcement learning to provide an effective learning for control problem are explained through the following paragraphs. Section 2.1 mentions mobile robot applications using different RL techniques. Finally, Section 2.2 gives several examples of RL approaches for vision based applications which mostly consist of navigation problem.

Anderson, since 1986, have used different techniques like fuzzy logic, control system design and neural networks for solving optimal control and search problems with reinforcement learning such as the Towers of Hanoi puzzle (Anderson, 1987), the inverted pendulum problem (Anderson, 1989) and control of heating in buildings (Anderson et al., 1997). Figure 2.1 shows another example of Anderson’s studies. Neural network was used to learn value function in the Mountain Car problem

(Anderson and Kretchmar, 2012). The implementation has three main structures, environment, agent and simulation as basis, and related code is available in the referenced link.

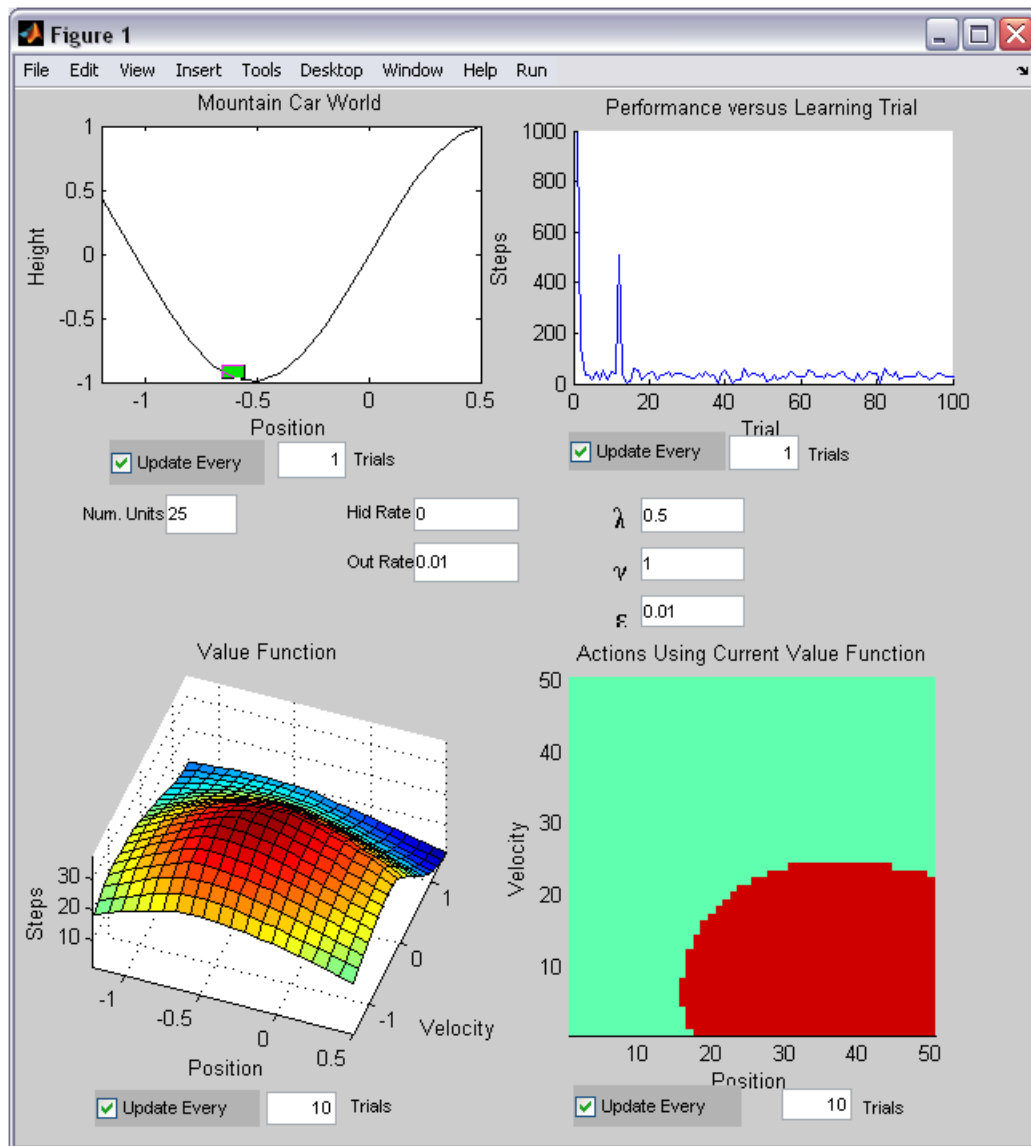


Figure 2.1 GUI for application of Mountain Car problem by Anderson (2012).

Martín and de Lope presented a new RL algorithm, $Ex(\alpha)(\lambda)$, developed to deal with problems of using continuous actions in 2009. This algorithm bases on k -nearest neighbors approach dealing with discrete actions and enhances the kNN -TD(λ) algorithm.

The classic k -NN technique classifies objects according to their distance to the classes in the feature space. In this algorithm, two parameters, k which is the number of neighbors to be considered and the distance function, are determinant. In this study, the value of k was 8 and Euclidian distance was used as distance function. The base k NN-TD(λ) algorithm works in three steps, perception, action selection and learning. In perception step, the agent finds k -nearest neighbors set, knn , of its current state, and each neighbor has its own Q-value. Then the agent obtains a probability distribution, $p(knn)$, over knn using their distance from the state s . At the end of this step, the state s is defined by two components knn and $p(knn)$. For rational action selection, it is necessary to find Q-value of current perception, and each neighbor in knn set affects the Q-value as much as their probabilities. After estimating action-value using Eq. (2.1) for each possible action, the agent chooses the greedy one to perform.

$$Q(knn, a) = \sum_{i=1}^{knn} Q(i, a)p(i) \quad (2.1)$$

For the learning step, the algorithm uses eligibility traces for TD(λ) technique. This technique and eligibility traces are explained in Section 3.2. The parameter λ refers decay rate for eligibility traces, and it is necessary to define a matrix e to store the traces. In this study, probability distribution $p(knn)$ is used to update the e matrix for selected actions.

While discrete actions list can include independent actions, continuous actions are related to each other and represented by an interval. Henceforth, only difference of the developed algorithm, $Ex\langle\alpha\rangle(\lambda)$, is in action selection step. Since this algorithm deals with continuous actions, calculation of best action value uses different mechanism. Firstly, the best Q-valued actions of each neighbor in knn set are selected. Then, expected optimal action is calculated using the selected actions and their probabilities. Eqs. (2.2), (2.3) and (2.4) expresses the action selection step for continuous actions.

$$I = \arg \max_a Q(i, a) \quad \forall i \in [knn] \text{ and } \forall a \in A \quad (2.2)$$

$$A^* = A[I] \quad (2.3)$$

$$\langle \alpha \rangle = \sum A^*(i)p(i) \quad \forall i \in knn \quad (2.4)$$

where I is an index list over the action list A , A^* is an action list containing optimal actions of knn set, $\langle \alpha \rangle$ is the expected value of the continuous optimal action to be executed and $p(i)$ is the probability of action $A^*(i)$ for given state s .

Finally, $Ex\langle \alpha \rangle(\lambda)$ algorithm was tested two different problems which are the Cart-Pole (see Figure 2.2) and the Helicopter Hovering. When performance of the developed algorithm was compared to $kNN\text{-}TD(\lambda)$; in Cart-Pole problem, the $Ex\langle \alpha \rangle(\lambda)$ algorithm using continuous actions could learn faster than the other, and in the Helicopter Hovering problem, although it was not possible to produce a convergence using $kNN\text{-}TD(\lambda)$ algorithm, $Ex\langle \alpha \rangle(\lambda)$ algorithm could converge in simulated environment.

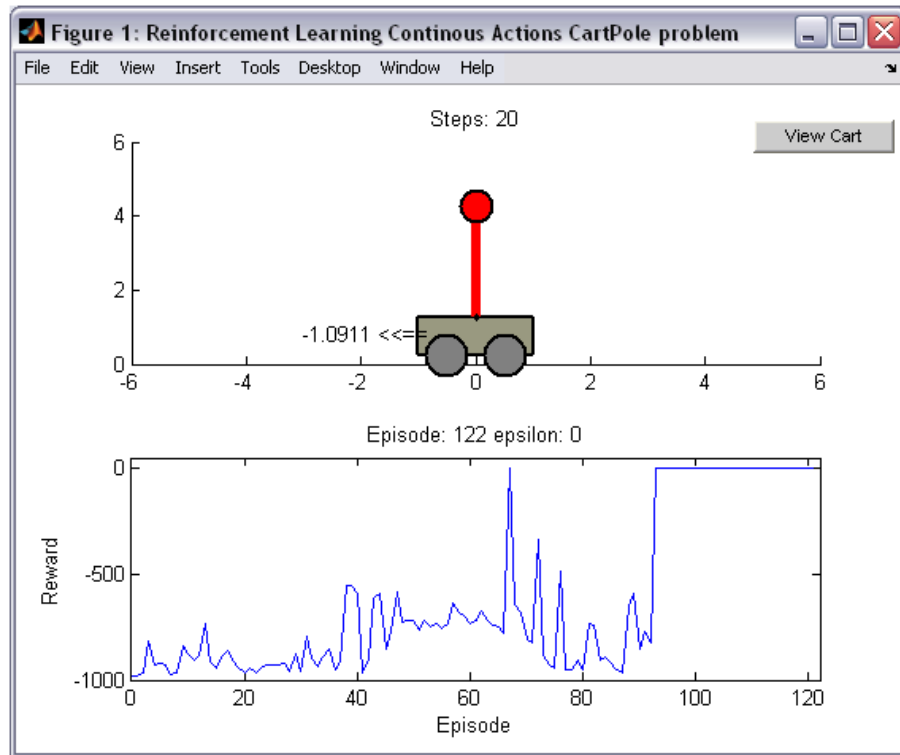


Figure 2.2 $Ex\langle \alpha \rangle(\lambda)$ algorithm results ($k = 8$; $\lambda = 0.9$; $\varepsilon = 0$; $\alpha = 1.2$; $\gamma = 1.0$) for the Cart-Pole problem.

2.1 MOBILE ROBOT APPLICATIONS

Because of complexity of the navigation problem, reinforcement learning is widely preferred to control mobile robots. This section examines a few examples of such navigation applications.

Maček et al. (2002) had a study using neural network approach in RL for obstacle avoidance of mobile robots. The main aim of this study was only to avoid from obstacles while roaming, there was not a specific goal to achieve. The reason of why they used neural network instead of look-up table was to minimize the required space and to maximize performance of the learning. In their design, they represented the environment by Kohonen neural network structure, and discretized their input space by using clustering technique to represent state set. In control step, greedy policy was applied to select actions and to update weight vectors, μ , representing Q-values. Figure 2.3 illustrates the applied structure for implementation of RL algorithm.

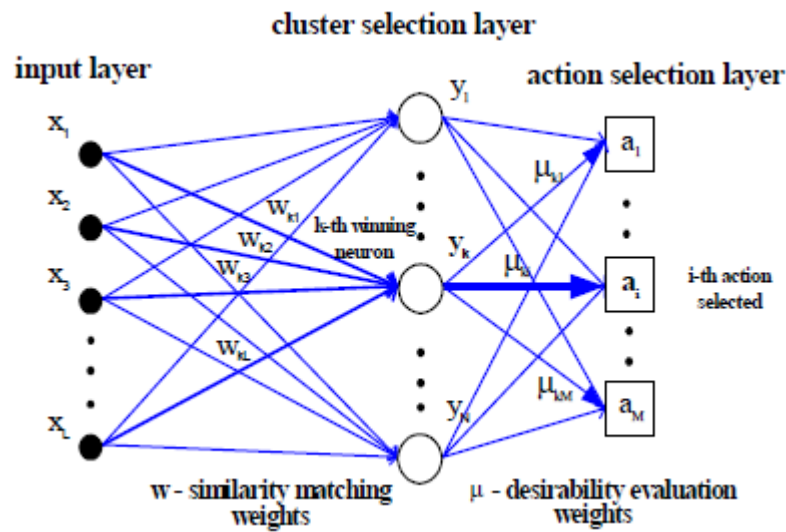


Figure 2.3 The neural network structure used by Maček et al. (2002).

The study is experimented only in a simulated platform. The simulated mobile robot had 8 sonar sensors located at $\pm 90^\circ, \pm 50^\circ, \pm 30^\circ, \pm 10^\circ$, and each sensor was assumed to return values of [15, 250] range and in cm unit. After these 8 values were

normalized, they clustered to find related state of the input as mentioned above. Then actions were selected from the set {"move 10cm forward", "turn 30° left and move 10cm forward", "turn 30° right and move 10cm forward"}. The robot was made on constant forward move to prevent it from being in stuck like turning on the same spot. Finally the reward function was selected as -1 for collision and entering the dangerous zone of 15 cm the closest object, and 0 otherwise. Figure 2.4 illustrates an example simulated result of the study.

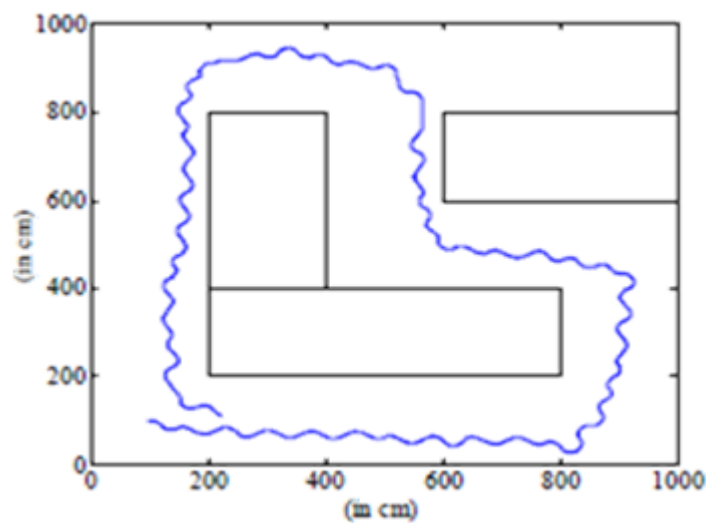


Figure 2.4 An example simulation for study of Maček et al. (2002).

Khriji et al. (2011) used prior knowledge within Q-learning in order to reduce memory requirement for look-up table and increase performance of the learning process. They integrated fuzzy logic into reinforcement learning for this purpose, and the action set was divided into four main behaviors, {"goal reaching", "obstacle avoidance", "wall following", "emergency situations"}. Q-learning algorithm was applied to provide coordination between those behaviors, and each behavior was designed using fuzzy logic.

Fuzzy behaviors had their own objectives; "goal reaching" guided the robot to reach a desired point, "obstacle avoidance" steered the robot to avoid collision with near obstacles, "wall following" kept the robot at a safe close distance to the wall in a

smooth line and “emergency situations” drive the robot away from U-traps and similar obstacles. Each behavior had a fuzzy rule table with two input and two output variables, all of which had degree of memberships in their membership functions. The input variables were “distance” that is the distance in meter between the robot and the goal, and “ θ -error” which is the difference in degree between the desired angle and the current angle of the robot. The output variables were “steering” that is the desired angle in degree for the next step, and “velocity” which is the speed of the robot in meter per second.

Khriji et al. implemented the system on a Peeke robot (see Figure 2.5a) which was equipped with 15 infrared sensors clustered into 6 groups as illustrated in Figure 2.5b. Those sensors measured the distance to obstacles as input for Q-learning algorithm. States used by Q-learning were determined by three different variables. These were “distance”, “ θ -error” and “D_roi” that is the difference between the robot and obstacle measured by *ith* cluster of sensors. Action set included four fuzzy behaviors mentioned above, and reward was 60000 when the goal was achieved, 1500 for another good action choice and -1500 otherwise.

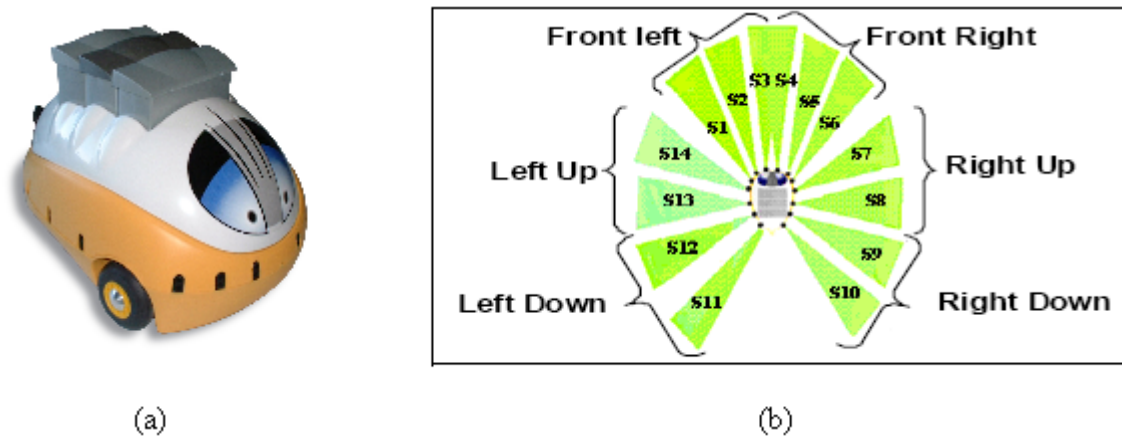


Figure 2.5 (a) Peeke robot and (b) its infrared sensors clustered by Khriji et al. (2011).

The system was experimented in two phases. The first one was learning phase which is used to obtain convergent Q-values. The Q-learning algorithm converged to optimal at around 60 episodes in this phase. Then, optimal Q-values stored in the

learning phase were used to measure the performance of the system in the test phase. This phase was applied in both simulated (see Figure 2.6a) and real platforms (see Figure 2.6b). Figure 2.6 shows that the robot can reach the goal with avoiding obstacles by using the optimal Q-values.

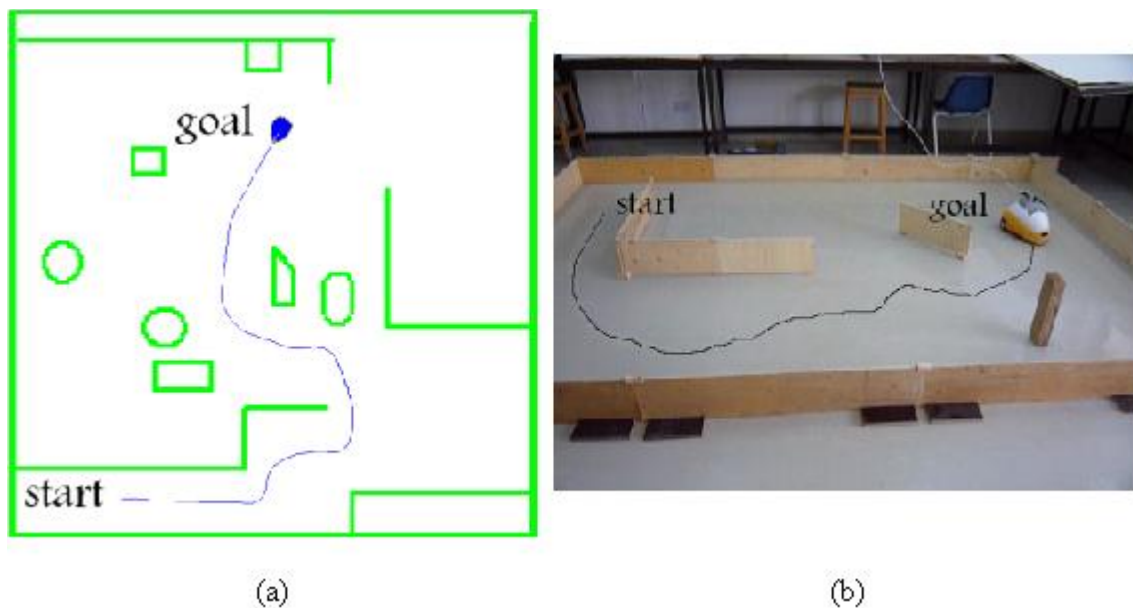


Figure 2.6 Example results in (a) simulated and (b) real platforms (Khriji et al., 2011).

2.2 VISION BASED APPLICATIONS

All vision based applications covered in this section preferred different reinforcement learning approaches to solve their problems – mostly the mobile robot navigation problem.

Zhumatiy et al. (2006) presented an algorithm called Piecewise Continuous Nearest Sequence Memory (PC-NSM) that extends the instance-based algorithm for discrete, partially observable state spaces, Nearest Sequence Memory (NSM) (McCallum, 1995). Like NSM, PC-NSM stores all the data it collects from the environment to deal with partially observability of vision-based systems and remembers locations of the target and obstacles if they move out of view; but unlike NSM, it uses a

continuous metric for state representation so that no prior input discretization of the perceptual space is necessary.

For experiments, a mobile robot equipped with two ultrasound distance sensors placed forward and backward, and a network camera located on the top was used in 3x4m walled area (see Figure 2.7). State set was defined by 5 features which are “x-y coordinates” of the target in the visual field with a predicate “ p ” indicating whether the target is visible and “forward-backward sonar readings”. Action set consists of 8 actions which are “turning left or right by either 22.5° or 45° ”, and “move forward or backward by either 5cm or 15cm”. The maximum velocity of the robot was limited to 10cm/s and k -nearest neighbor algorithm was used to select action by ε -greedy policy for the purpose of eliminating sensory noise. The received reward was based on distances of the robot from target and obstacles. After the learning algorithm was firstly executed in simulated platform, the resulting policy was transferred to the real robot to test the performance. Figure 2.7 illustrates an experimental result on the real robot using the learned policy.



Figure 2.7 A sample performance of the real robot used by Zhumatiy et al. (2006).

Busquets et al. (2002) divided their system into three subcomponents; Navigation system, Pilot system and Vision system. The Navigation system was responsible to find a path for guiding the robot to reach a specific goal through landmarks placed in the environment. The Vision system had a task to identify the tracking landmarks, the target and obstacles. The Pilot system controlled the motions of robot wheels to perform commands from the Navigation and Vision systems by avoiding obstacles. All systems cooperated to achieve a common task whereas each system also competed for the services of the other systems. For instance, both the Vision and the Navigation systems competed for the Pilot system. The Vision needs the Pilot to capture images after and before a suitable motion and the Navigation needs the Pilot to move the robot to the goal. Additionally, the Pilot and the Navigation both compete for the Vision in a similar manner (Busquets et al., 2002). Figure 2.8 illustrates the overall system that uses RL to coordinate the subcomponents for cooperation and competition mentioned above.

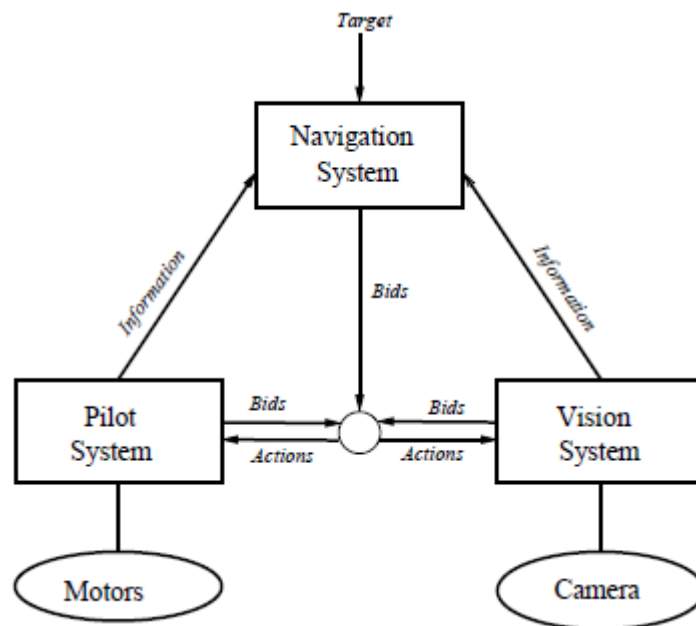


Figure 2.8 The system architecture used by Busquets et al. (2002).

The main aim of the learning agent was to achieve the goal by minimizing the use of the camera. To accomplish this aim, Prioritized Sweeping (Moore and Atkeson, 1993) algorithm was used. Prioritized Sweeping is a model-based algorithm and works

like DP except it updates values of k most important states instead of all states using Bellman equation (1.10). The state set to be used in the Prioritized Sweeping algorithm was defined by 4 variables; “target distance”, “target location imprecision”, “landmark count” and “landmark imprecision”. The action set consisted of 5 actions; “move blind”, “move and look for landmarks”, “move orthogonally to target”, “move and verify landmarks”, and “move and verify target”. Since the aim was to minimize camera usage, the reward function varied depending on required image processing for the selected action. Although the Pilot was said to avoid obstacles, the system was simulated in an environment without any obstacles.

Zhu and Levinson (2001) proposed a hybrid state-mapping model by simulating a vision based learning approach for robot navigation. The simulated robot was designed to use two stereo cameras as sensors integrated to its header. Negative rewards for losing sight of the target in either of views, and positive rewards for achieving the goal sensed by the sensors on its gripper were received. States were defined by two integers; d' and x_{left} where d' is the difference between x_{left} and x_{right} , x_{left} is the left-most point of the target in the left images and x_{right} is the right-most points in the right images. Since two integers were used for state representation, the input state space became infinitely large. Yet, neighbor states having same action policy could be grouped to reduce state space size. This time, not knowing which states to group became the remaining problem. To solve this problem, initially, the continuous feature space was represented by a small state set. For learning purpose Q-learning algorithm was used to find optimal policy and ϵ -greedy approach was used for action selection. But if a state is ambiguous, then the agent picks the action randomly and calls state splitting which is a recursive algorithm splitting the current state into smaller states until the ambiguity is eliminated.

The simulation was performed in a $4 \times 4 \text{m}^2$ environment using 5 actions that are “turning left/right for 3°” and “going forward for 6/12/24 inches”, and reward function defined as 1000 for achieving the target and -1000 for losing its sight in either of camera. No obstacle was inserted into the environment and the target object was defined by a simple rectangle of $6 \times 15 \text{cm}$ (Figures 2.9b and 2.9c). It was assumed that the distance between the stereo cameras was 8cm, and resolution of each camera was 320×240 pixels. During simulation, each episode started from a random position. Figure 2.9a displays an example episode of simulation.

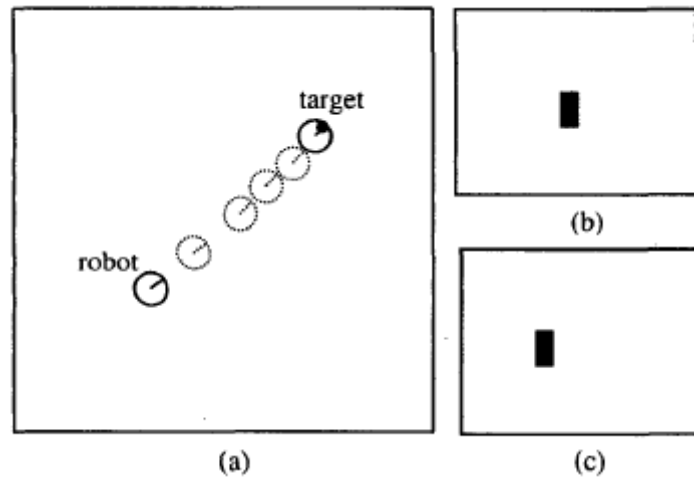


Figure 2.9 An example simulation of (a) environment and images perceived (b) right and (c) left stereo cameras (Zhu and Levinson, 2001).

Menegatti et al. (2004) used Sarsa(λ) to implement vision agents (VAs) in a network of cameras named distributed vision system. Each VA was an omni-directional vision sensor and communicated each other to share the knowledge acquired using Sarsa(λ) algorithm. All VAs had a unique purpose of learning to navigate a dummy robot with no sensors and to guide it through the goal rationally. ϵ -greedy policy and replacing eligibility traces (see Section 3.2) was used in the algorithm in order to achieve this purpose. VA extracted 5 features to define a state by using captured image with resolution of 640x480 pixels. These features were “distance between the robot and the target position”, “distance between the robot and the camera”, “distance between the target position and the camera”, “the orientation of the robot with respect to the target position” and “the orientation of the robot with respect to the camera”. Action set was defined as a pair of linear and angular velocities. The VA got a reward of -60 for guiding the robot outside of view and 10 for guiding it to the target successfully. In other cases, the reward function varied according to the selected velocity. Finally, the whole system was tested in a simulated platform.

Shaker et al. (2010) used an RL algorithm to simulate vision-based landing of an unmanned aerial vehicle (UAV). The system worked in a way that the UAV perceived its environment through a camera mounted on the central bottom, and then image processing techniques were applied to extract features representing current state of the agent. Canny edge detection operator and Hough transform algorithms in OpenCV

library was used to find landing target. All state set variables were calculated depending on the target detected from the captured images. Table 2.1 presents the parameters used in RL system. Finally, using the calculated variables, Least-Squares Policy Iteration (LSPI) that is an RL algorithm was applied for optimal control purpose (Lagoudakis and Parr, 2001).

Table 2.1 The parameters used by RL system (Shaker et al., 2010).

Parameter	Parameters Range
r	$[0, 10]$ meters
θ	$[0^\circ, 360^\circ]$
ϕ	$[-90^\circ, 90^\circ]$
Reward	(1) 120 if it gets to the goal, (2) -1500 if it finishes outside state space, (3) equal to a value, this value is decrease as ϕ or θ increase
Goal State	$r = [0, 0.1]$ m, $\theta = [0^\circ, 10^\circ]$, $\phi = [-5^\circ, 5^\circ]$
Control	Lateral Velocity: $[-5, 5]$ Linear Velocity: $[-5, 5]$

The first row in Table 2.1 shows parameters for representation of the state set; r is the radial distance of the agent from the center of the target, θ represents the angle between x -axis of the target and projection of the radius, r , on the xy -plane, and ϕ is the polar angle from the z -axis. The last row displays the continuous action set with two variables, lateral and linear velocities controlling right/left and forward/backward velocities, respectively. Although, the simulated UAV had capability of two more actions that are altitude and rotational velocities controlling up/down and rotation movement of the agent; only former two actions were desired to be learned and altitude velocity was set to decrease at a fixed rate. If the UAV descends so much that it loses track of the target, the RL controller is stopped to ensure safety of the agent.

LSPI is an RL algorithm that learns from experience by approximating Q-values with a parametric function that is a linear weighted combination of k basis functions.

Shaker et al. extended LSPI algorithm to deal with continuous action set defined in Table 2.1. Two different basis functions were used in the simulation, polynomial basis function (PBF) and radial basis function (RBF). The system was simulated for testing PBF using polynomials of order 2 and 4, and RBF using Gaussian function in a bounded environment with no physical obstacles and external effects like wind. The best results were obtained when RBF was used with learning rate of 2.1 iterations in average. The complexity of LSPI algorithm depends on the number of basis function used and the number of samples collected, and is not affected by state and action set dimensions. Figure 2.10 shows sequence of images for an example simulated landing process with learned controller.

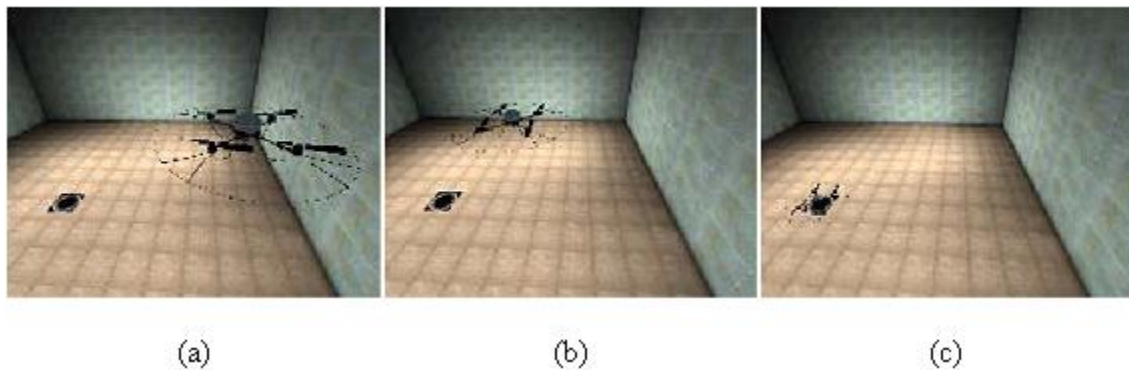


Figure 2.10 Image sequence of a simulated landing (Shaker et al., 2010).

CHAPTER 3

IMPLEMENTATION

This chapter explains implementation details of the system that aims basically to learn how to enable a visionary agent to reach a desired goal without any obstacle collisions by the help of reinforcement learning. Section 3.1 examines the proposed system developed for this purpose by considering the problem defined in Section 1.2. Section 3.2 expresses mechanism of eligibility traces which is used by the proposed system as a RL algorithm that combines Monte Carlo and TD methods. Finally, Section 3.3 introduces interface of the proposed system and gives some information about usage of the interface.

3.1 PROPOSED SYSTEM

Firstly, it is necessary to design the system properly in order to achieve the main purpose of autonomous navigation. The mobile agent is supposed to reach the goal by following an optimal path and avoiding obstacles in the environment after a set of trials. The proposed system uses on-line updating meaning that the agent improves its guess about the environment during the trials not after terminations of the trials. As the agent performs more trials and gains more information about its environment, it exhibits better performance for navigation purpose.

Q-values used by the reinforcement learning algorithms are in tabular form and represented as matrices by the system. This causes lots of matrix calculations necessary during the learning process, and since those calculations can be made in a concise way by the help of MATLAB language, the system has been implemented as a MATLAB

application. As mentioned above, experimentation of the system is performed on the Robotino[®] that is an educational robot by Festo company (see Appendix A). Robotino MATLAB API is integrated to the system in order to communicate with the Robotino[®].

Figure 3.1 shows a general block diagram indicating processing and argument modules of the proposed system. Arrows in the figure imply providing information for another module, and the modules in light blue are argument modules. It is important to define system arguments appropriately to have a good performance in learning. The following subsections explain the modules constructing the system briefly.

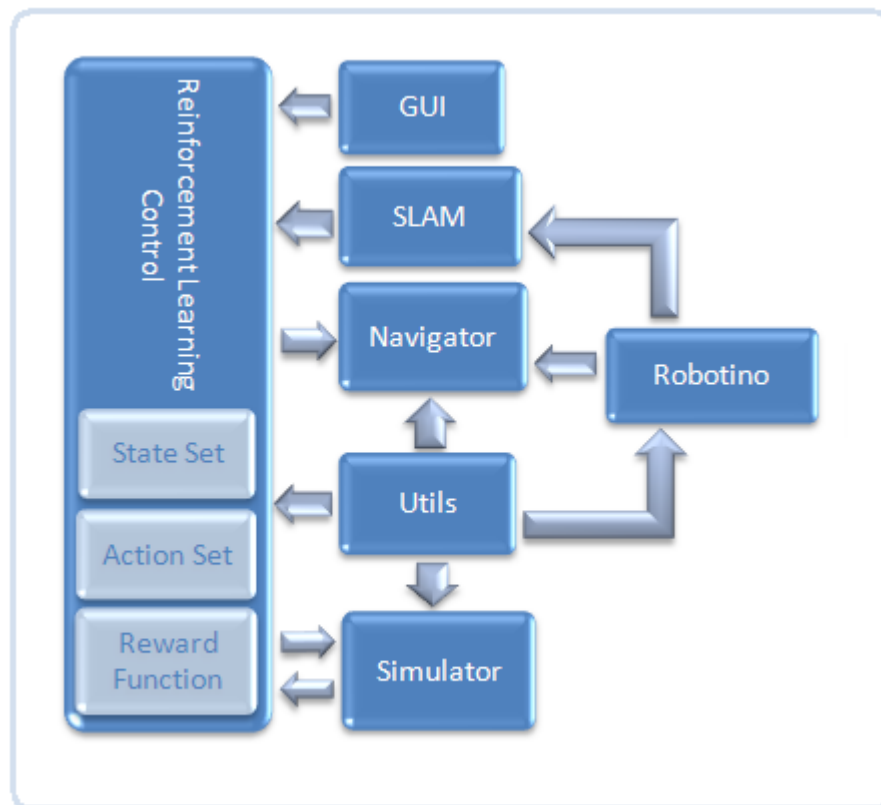


Figure 3.1 The block diagram of the proposed system.

3.1.1 Graphical User Interface (GUI)

This module provides an interface to the user to execute the system and test its algorithms. The user can tune initial parameters, like learning rate, and can select the

embedded algorithm executed in the RL control module to observe how they affect the performances in learning.

3.1.2 Navigator

The Navigator module is responsible to perform the actions selected by the reinforcement learning control module. In order to connect to the robot and send the action orders, this module uses functions of Robotino MATLAB API, which are provided by the Robotino module.

3.1.3 Reinforcement Learning Control

This is the central module running the selected algorithms in the system. The user can select either the Sarsa(λ) or Q(λ) algorithms through GUI module. The λ parameter that can also be changed by the user is trace-decay parameter. The usage of λ is explained in Section 3.2. Both of algorithms use the ε -greedy method for action selection. Additionally, Sarsa(λ) uses ε -greedy for also policy estimation while Q(λ) algorithm uses greedy method. Initial value of the ε variable can be changed by the user through the GUI module; yet, it is decayed by 0.99 after each episode as the agent learns more.

This module includes three main argument modules: the Action Set, the Reward Function and the State Set modules. Those are critical modules that affect the performance of the RL control. It is essential to define those modules well enough to make the agent learn as expected.

3.1.3.1 Action Set

Since the system is experimented in a continuous environment, it is important to decide the action set. The actions should be selected small enough to be executed in one step while sufficient enough to let the robot navigate to its goal. The action set used in the system consists of three actions that are “move forward with linear velocity v ”, “turn right with angular velocity w ” and “turn left with angular velocity w ”. Default values of the variables, v and w are 150mm/s and 15deg/s, respectively, though; the user can change those values through the GUI module.

In simulation platform, it is better to perform actions considering distance the agent moves instead of its velocity. Therefore the simulated action set is a little bit different from the real-platform action set. The three simulated actions are “move forward for distance d ”, “turn left for angle θ ” and “and turn right for angle θ ”. Whereas the d and θ variables can be changed by user from the GUI module, the default values are 100mm and 10° , respectively.

3.1.3.2 Reward Function

This module indicates the rewards the agent receives from the environment as a result of its actions, in other words; reward function determines response of the environment to the actions of the agent. There are four different rewards defined as +100 for achieving the goal, -100 for obstacle collision, -1 for the actions making the agent get closer to the target and -2 otherwise.

3.1.3.3 State Set

Great number of states defined in RL algorithms not only increases the complexity of the system but also makes harder to sample all states and so to find an optimal policy. However, if the states are not defined well enough, the system can suffer from not representing all situations sufficiently in the environment.

The proposed system discretizes continuous states of the environment depending on the position of the agent according to the target and the nearest obstacle on the way by using four variables, “ d_{tar} ” that is the distance in mm of unit between the robot and the target, “ θ_{tar} ” that is the orientation of the robot with respect to the target in degrees, “ d_{obs} ” that is the distance between the robot and the closest obstacle, “ θ_{obs} ” which is the orientation of the robot with respect to the closest obstacle. The distance variables range [0mm, 1000mm, 2000mm] and orientation variables range $[-30^\circ, 0^\circ, 30^\circ]$. This makes the system be represented by 81 different states. Figure 3.2 shows the variables defined in the state set. While determining the nearest obstacle, not all obstacles are considered, only the obstacles in the interval of $[-45^\circ, 45^\circ]$ (see Figure 3.2b) with respect to the robot direction are taken into account.

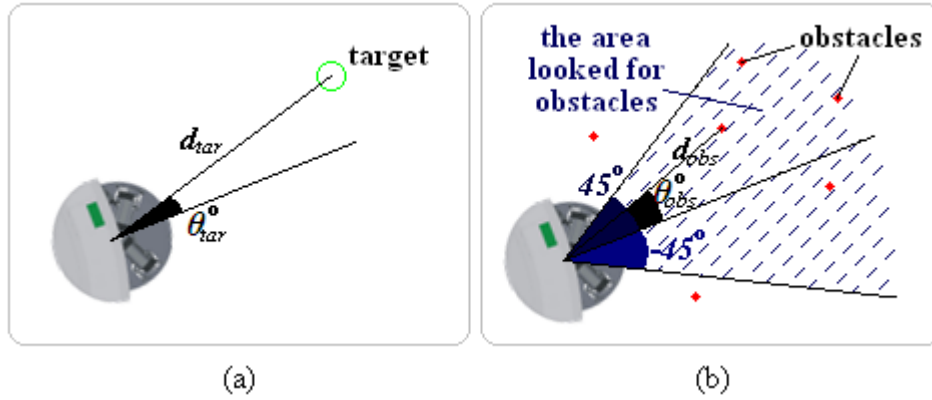


Figure 3.2 Variables of the State Set module are defined by robot distance and orientation according to (a) the target position represented as d_{tar} and θ_{tar} ; and (b) the nearest obstacle on the way represented as d_{obs} and θ_{obs} .

3.1.4 Simulator

The Simulator module, in a way, combines the Navigator, Robotino and SLAM modules which are used for the real platform. When the user selects the simulated platform, this module performs the actions selected by the RL Control module and returns the resulting position of the robot during the learning process. Since the system is designed to be stochastic, the Simulator module is developed to have some inaccuracies like the real platform.

3.1.5 Simultaneously Localization and Mapping (SLAM)

This module is responsible for visionary part of the system. It gets captured images of the environment, and returns estimated locations of the robot and of the obstacles with respect to the starting point in world reference frame. The uncertainty of locations reduces as the robot navigates in the environment. The technical details of this module are out of scope of this thesis. The system, VBAN, developed by Öztürk (2012) is design to be integrated to provide the vision functionality of this module.

VBAN extracts some robust features of natural landmarks through images perceived from the environment by using some popular algorithms which are scale invariant feature transform (SIFT) and speeded up robust feature (SURF). The choice of which one is used depends on the user. By integrating the extracted features and

odometry data, VBAN updates its estimate of locations of the robot and the landmarks in 3D environment. This estimation is made with some amount of uncertainty by using extended Kalman filter (EKF), and the uncertainty reduces as the system performs more actions.

3.1.6 Utils

The Utils module consists of some utility functions returning Euclidean distance between rows of two matrices and angle between two lines, enabling the system to sleep for a while and so on. The Controller module uses this module to find in which state the agent is and to decide the action according to the state. The Navigator and Simulator modules use functions of this module to calculate how much the agent moved during the selected action. Finally, the Robotino module needs the Utils module to sleep for a while in order to wait the system to connect to or disconnect from the Robotino[®] and to wait the Robotino[®] to perform the received orders.

3.2 ELIGIBILITY TRACES

Solution classes developed for decision making algorithms which are dynamic programming, Monte Carlo methods and temporal-difference were investigated in Chapter 1. All classes have their own strengths and weaknesses, and they are not actually alternatives for each other. Eligibility traces, one of the unified algorithms developed to embody key ideas of those methods, are combination of TD learning and Monte Carlo methods.

On the one hand, Monte Carlo methods average real values returned after a complete episode, so suffer from the requirement of waiting until the termination of episodes. On the other hand, TD methods use one-step return value that is equal to sum of immediate reward and discounted next Q-value (see Eqs. (1.12) and (1.13)), and updates value estimation of only one state-action pair which causes the problem of slow convergence for large state/action spaces. However, it is possible to update estimated values of recently visited state-action pairs since the environment is sequential meaning that the action selection is affected by all performed actions not by only previous action.

Eligibility traces use two different ways to update estimations of value functions similar to $TD(\lambda)$ learning methods; on-line and off-line updating. In the former updating, value estimates are updated during the episodes whereas updates are accumulated and used after termination of the episodes in the latter updating. When performance of on-line and off-line updating methods are compared, empirical results show that on-line methods converges to the optimal policy faster than off-line methods (Sutton and Barto, 2005). Thus, the proposed system uses on-line updating.

In order to implement $TD(\lambda)$ method, an additional variable $e_t(s, a)$ is defined as eligibility trace for all state-action pairs at time t . The trace-decay parameter, λ , in $TD(\lambda)$ algorithm refers to the use of eligibility traces in order to update values of recently visited states. At each step, eligibility traces for all states are decayed by $\gamma\lambda$ defining the recently visited traces. Two kinds of traces, accumulated traces and replacing traces, are used to record traces of state-action pairs. The accumulated traces increments the eligibility trace, $e_t(s, a)$, by 1 for each visit of state-action pairs while the replacing traces reset the eligibility trace to 1. The replacing traces can also reset the other actions available in the visited state to 0. Generally replacing traces have better performance than accumulating traces. Eq. (3.1) defines $e_t(s, a)$, the replacing traces used by the proposed system.

$$e_t(s, a) = \begin{cases} 1 & \text{if } s = s_t \text{ and } a = a_t \\ 0 & \text{if } s = s_t \text{ and } a \neq a_t \\ \gamma\lambda e_{t-1}(s, a) & \text{if } s \neq s_t \end{cases} \quad \text{for all } s, a \quad (3.1)$$

The one-step TD error calculated by Eq. (3.2) is used to update values of all recently visited state-action pairs by their nonzero traces using Eq. (3.3).

$$\delta_t = R_a(s, s') + \gamma Q_t(s', b) - Q_t(s, a) \quad (3.2)$$

$$\Delta Q_t(s, a) = \alpha \delta_t e_t(s, a), \quad \text{for all } s, a \quad (3.3)$$

where value of $Q_t(s', b)$ depends on preferred algorithm which can be Sarsa(λ) or Q(λ) explained in the following subsections.

3.2.1 Sarsa(λ)

This subsection explains how eligibility traces are integrated into Sarsa algorithm examined in Chapter 1. The eligibility trace version is known as Sarsa(λ), and Eq.(3.4) defines update step of the Sarsa(λ) algorithm.

$$Q_{t+1}(s, a) = Q_t(s, a) + \Delta Q_t(s, a), \quad \text{for all } s, a \quad (3.4)$$

where $\Delta Q_t(s, a)$ is defined by Eq. (3.3), and $Q_t(s', b) = Q_t(s', a')$ in Eq. (3.2) since Sarsa(λ) is an *on-policy* method as defined in Eq. (1.12). Figure 3.2 shows pseudocode for the complete Sarsa(λ) algorithm using ϵ -greedy policy, on-line updating and replacing traces.

```

Initialize  $Q(s, a) \leftarrow 0$  and  $e(s, a) \leftarrow 0$  for all  $s, a$ 
Repeat (for each episode):
  Initialize  $s, a$ 
  Repeat (for each step of episode):
    Take action  $a$ , observe  $r, s'$ 
    Choose  $a'$  from  $s'$  using policy derived from  $Q$  ( $\epsilon$ -greedy)
     $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$ 
    For  $b$  in Action Set:
       $e(s, b) \leftarrow 0$ 
     $e(s, a) \leftarrow 1$ 
    For all  $s$  in State Set,  $a$  in Action set:
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$ 
       $e(s, a) \leftarrow \gamma \lambda e(s, a)$ 
     $s \leftarrow s'$ 
     $a \leftarrow a'$ 
  until  $s$  is terminal

```

Figure 3.3 The pseudocode for Sarsa(λ) algorithm used by the proposed system.

3.2.2 $Q(\lambda)$

The proposed system uses Watkins's $Q(\lambda)$ that is a Q-learning algorithm into which the eligibility traces are integrated. Since $Q(\lambda)$ is an off-policy algorithm, only traces of greedy actions are used. Therefore, the eligibility traces are set to 0 when a non-greedy action is selected for exploration. Like Sarsa(λ), this algorithm uses Eq. (3.4) to update Q-values with $Q_t(s', b) = \max_a Q_t(s', a)$ in Eq. (3.2) because the $Q(\lambda)$ algorithm uses greedy method in policy improvement process as expressed in Eq. (1.13). Figure 3.3 shows pseudocode for the complete $Q(\lambda)$ algorithm using ϵ -greedy behavior policy, greedy estimation policy, on-line updating and replacing traces.

```

Initialize  $Q(s, a) \leftarrow 0$  and  $e(s, a) \leftarrow 0$  for all  $s, a$ 
Repeat (for each episode):
  Initialize  $s, a$ 
  Repeat (for each step of episode):
    Take action  $a$ , observe  $r, s'$ 
    Choose  $a'$  from  $s'$  using policy derived from  $Q$  ( $\epsilon$ -greedy)
     $a^* \leftarrow \operatorname{argmax}_b Q(s', b)$ 
     $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s, a)$ 
    For  $b$  in Action Set:
       $e(s, b) \leftarrow 0$ 
     $e(s, a) \leftarrow 1$ 
    For all  $s$  in State Set,  $a$  in Action set:
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$ 
      If  $a' = a^*$ , then  $e(s, a) \leftarrow \gamma \lambda e(s, a)$ 
      Else  $e(s, a) \leftarrow 0$ 
     $s \leftarrow s'$ 
     $a \leftarrow a'$ 
  until  $s$  is terminal

```

Figure 3.4 The pseudocode for $Q(\lambda)$ algorithm used by the proposed system.

3.3 SOFTWARE INTERFACE OF THE SYSTEM

The proposed system is a high-level application implemented in MATLAB as mentioned above. Figure 3.5 shows interface of the system named Reinforcement Learning Control (RLC). RLC interface is designed to consist of six main parts; Navigation Path, Reward Function, Initial Parameters, Info Box, Image Frames and Button groups.

The Navigation Path part plots the target as a green circle, the obstacles in the environment as red demands, the robot as a gray-filled circle and its path during the episode as a black line. This part can be made inactive to increase the learning speed by avoiding unnecessary graphical processes. After termination of episodes, the robot path is plotted to give a clue to the user about the learning process. The Reward Function part plots the cumulative reward gathered from the environment after each episode. When the agent learns optimal behavior well enough, the reward function plotted in this part is expected to become stable. The Initial Parameters part gives different options to the user to control the system, e.g, selecting the performed algorithm, the experiment phase and the experimental platform. Additionally, the Initial Parameters part let the user observe how different parameters values affect performance of the learning. The Image Frames part displays the images captured through the camera of the Robotino[®] with rate of 5Hz. This part is disabled for simulated platforms and can be made inactive to increase the speed of the learning process in real experiments.

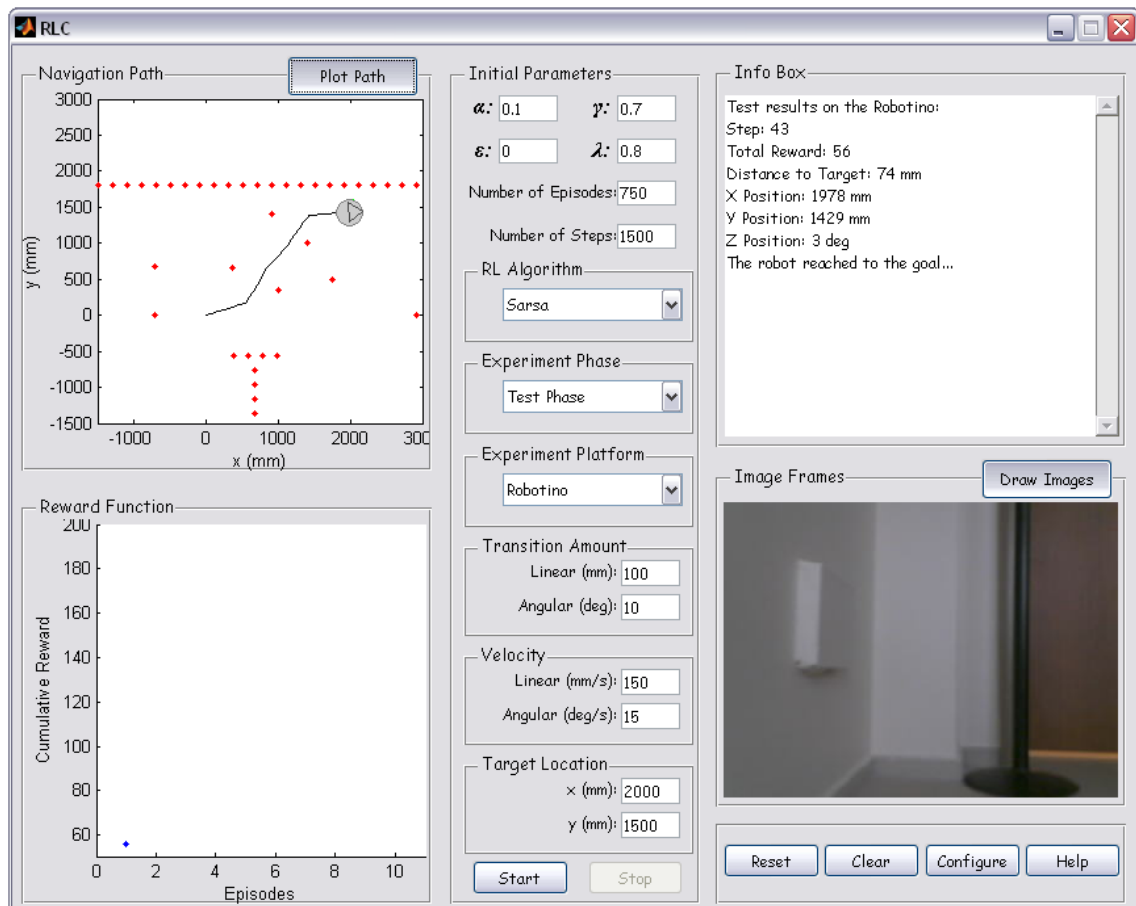


Figure 3.5 The GUI of the proposed system named RLC.

The Info Box part informs the user about each episode by giving how many steps are taken during the episode, the cumulative reward, distance to the target, final position of the robot, if the robot is able to reach to the goal and if it bumps against an obstacle. The Info Box also gives information about what the parameters are when the user clicks the 'Help' button at the right bottom corner. Finally, the Button Groups includes 4 buttons the 'Help' button gives information about initial parameters and their features. The 'Reset' button resets all parameters to their default values. The 'Clear' button clears the four parts informing the user; the Navigation Path, Reward Function, Info Box and Image Frames parts. Since the system is not capable of detecting obstacles for now, it is necessary to configure the environment before the learning process hence the 'Configure' button is added to the GUI of the system. Figure 3.6 shows the interface for configuration of the environment. The user can change the boundary of the environment, and there are three different obstacle options; 'No Obstacle', 'Random

Obstacles' and 'Set Obstacles'. 'No Obstacle' is the default option and the system executes in an environment without obstacles. When 'Random Obstacles' option is selected, the system generates N random obstacles for the environment where N can be set by the user. The last option is 'Set Obstacles' enabling the user to set obstacles manually.

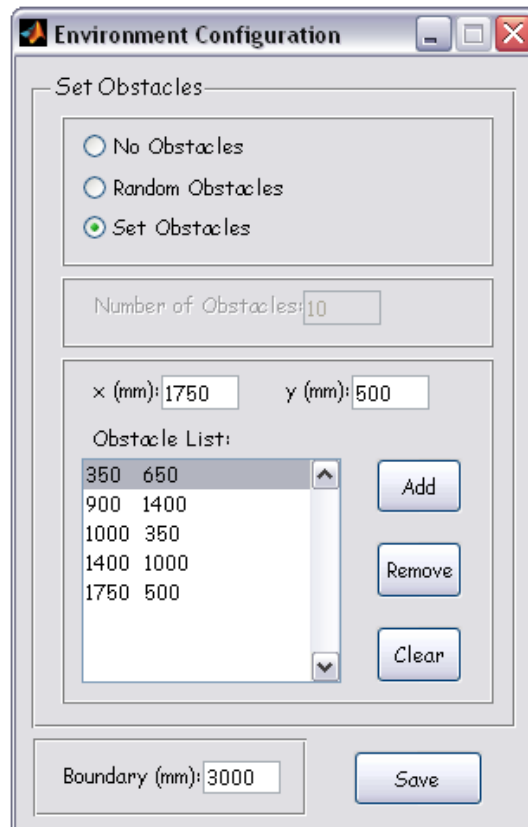


Figure 3.6 The interface for environment configuration of the system.

CHAPTER 4

EXPERIMENTS AND RESULTS

The proposed system is designed to execute in both simulated and real platforms, and experiments are done in two phases; learning phase and testing phase. The learning phase is preferred to be experienced only in simulated platform since the real robot can be damaged because of collisions with obstacles at the beginning of the phase. Besides, it is hard to relocate the robot between consecutive episodes, yet several episodes should execute for a successful learning.

The RLC system is tested in the environment both with and without obstacles. As might be expected the learning performance is better in the environments without obstacles since there is no risk for obstacle collisions which makes the environment much simpler to be learned. The following sections explain experimental results in simulated and real platforms, respectively.

4.1 SIMULATED EXPERIMENTS AND RESULTS

For the simulated platform, a virtual robot is implemented to perform the selected actions in each step of the episodes. In order to make a realistic simulation, some errors are added to those actions during execution. The main purpose of simulation is obtaining an optimal Q-value matrix to be used in the real robot. Figures 4.1 and 4.2 illustrate performances of Sarsa(λ) and Q(λ) algorithms in the environment with three different obstacle options (see Figure 3.6), respectively. All the other parameters are the same and have their default values.

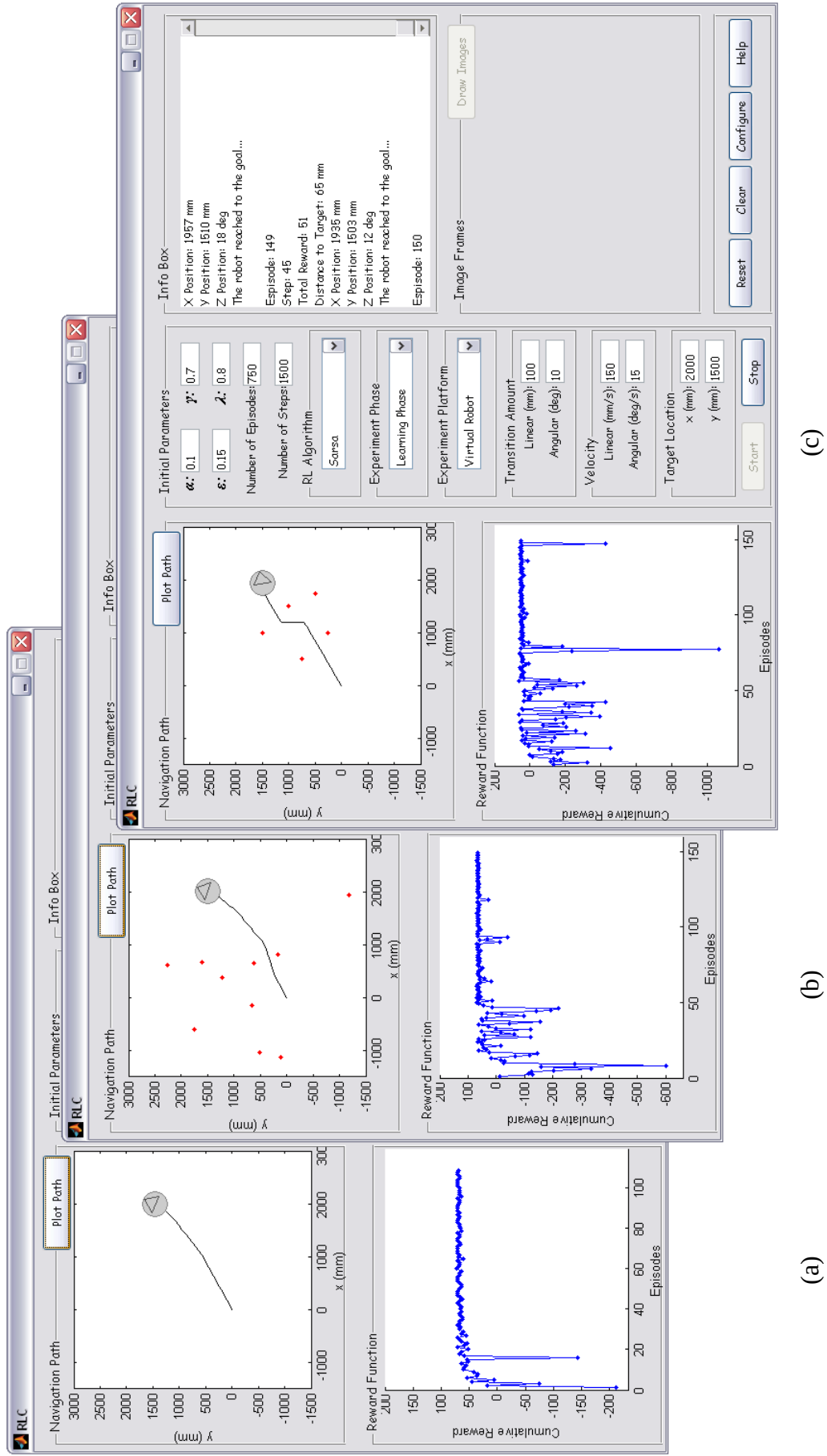


Figure 4.1 Simulated Sarsa(λ) results for three different obstacle options: (a) No Obstacles (b) 10 Random Obstacles (c) Set 5 Obstacles.

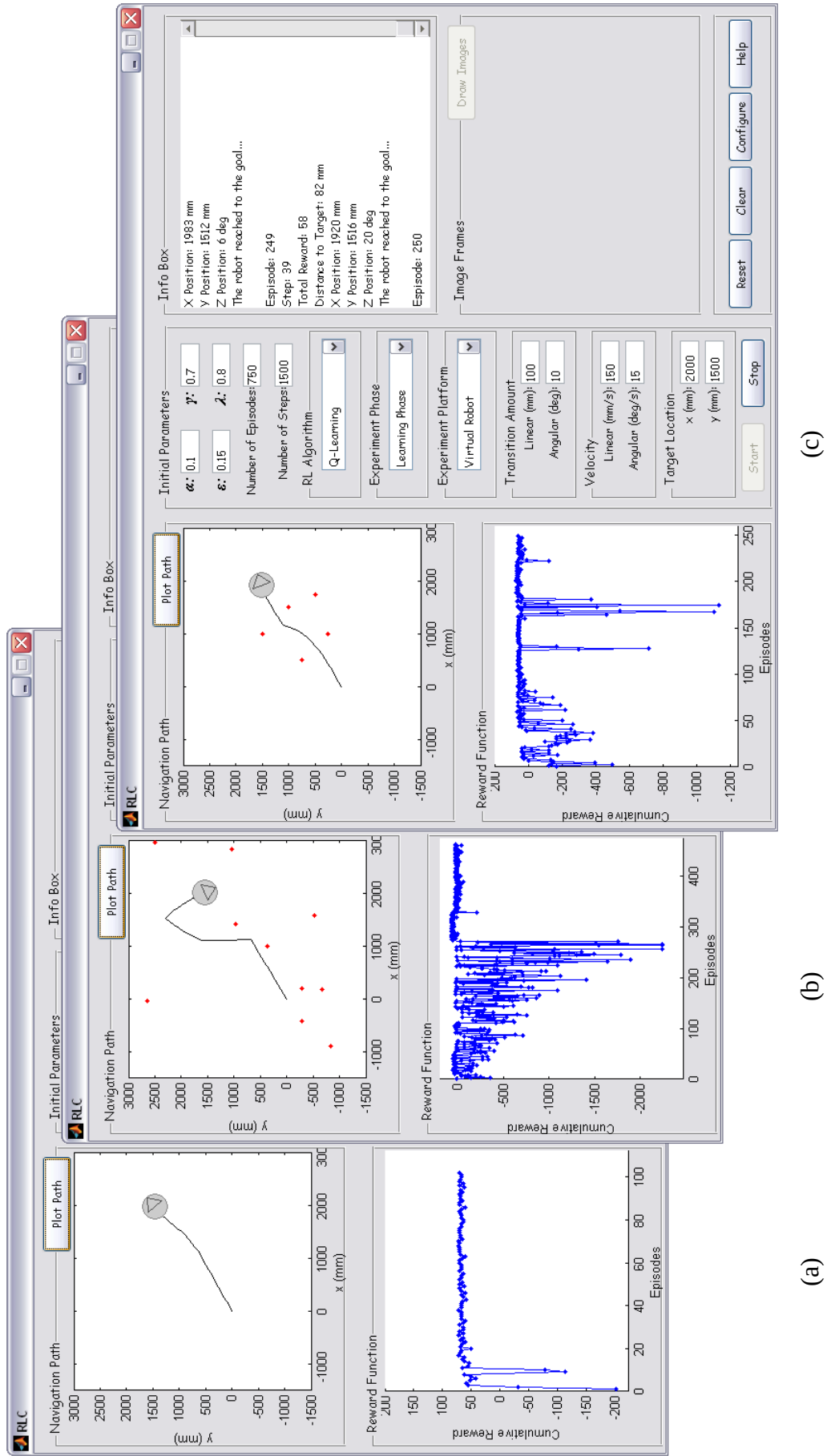


Figure 4.2 Simulated $Q(\lambda)$ results for three different obstacle options: (a) No Obstacles (b) 10 Random Obstacles (c) Set 5 Obstacles.

When there are no obstacles in the environment, performance of both algorithms is similar to each other and the cumulative reward stabilizes at around 20 episodes as seen in Figures 4.1a and 4.2a. However, adding obstacles to the environment makes the learning process more complicated. Despite the fact that performances of both algorithms decreases, Sarsa(λ) converges faster than Q(λ). In Figures 4.1c and 4.2c, the obstacle locations are the same but Sarsa(λ) algorithm finds the optimal way requiring less episodes than Q(λ) algorithm.

Even though Figures 4.1 and 4.2 shows examples of finding optimal paths for Sarsa(λ) and Q(λ) algorithms, respectively, not all experiments result by finding optimal paths, the agent may result in a vicious circle at the end of some learning processes. Sarsa(λ) algorithms has about 90% performance rate for finding optimal paths whereas Q(λ) has approximately 70% performance rate. Yet the performance of both algorithms depends also on initial parameter values, number and locations of the obstacles, so the user is also responsible for tuning the parameters to get the best performance.

Finally, since the virtual robot cannot capture images of the simulated platform, information about locations of the obstacles in the environment is gathered from the simulator module instead of the sensors.

4.2 EXPERIMENTAL PLATFORM – THE ROBOTINO[®]

Robotino[®] is an educational robot platform developed by Festo. It has a built-in PC working at 500MHz and using Linux based operating system. Details about Robotino[®] can be seen in Appendix A. Although the PC embedded in Robotino[®] is insufficient to be able to execute the proposed system, it provides W-LAN and other services to communicate with another computer by transferring sensor values and getting necessary orders. Therefore, the system is executed on an external computer, and uses Robotino MATLAB API for communication.

In order to make the Robotino[®] perform selected actions properly, it is necessary to send related orders for the actions according to Degree of Freedom (DoF) of the Robotino[®] (see Figure 4.3). For instance, moving forward and left mean moving

through x-axis and y-axis in positive direction, respectively so when the Robotino[®] is ordered to turn for θ° , it turns to left, and for $-\theta^\circ$, it turns right.



Figure 4.3 Degree of Freedom (DoF) of the Robotino[®].

Although Robotino[®] has some other sensors, only three of them are designed to be used in the system; the bumper, the odometer and the camera with resolution of 320x240 pixels. The bumper is used to detect collisions with the obstacles. The odometer is used to get location of the robot with respect to its first location. Despite the fact that RLC is designed to be a vision based system which uses the camera to gather information about locations of the obstacles in the environment, the details of how to do so is out of scope of this thesis. For now, the obstacle locations are given to the system as prior knowledge, but the next step is to integrate VBAN by Öztürk (2012) to provide vision capability.

Studying on a real platform causes some extra problems that are not encountered on the simulated platform. For instance, it is not a problem for the simulated agent to make decisions in rest and between actions selected from a set of transitions for specific amounts; yet, this situation causes the real robot to move jerkily. In order to avoid that

jerky transition, it is designed that the Robotino[®] keeps its motion of the previously selected action during decision making process for the next action.

Another problem for real experimental platform is about accelerating motors of the omnidirectional wheels on the robot to the specified velocity. Subsection 4.2.1 gives details about the PID control to solve this problem. Besides, it is not possible to estimate how much the robot moved during the acceleration process so it is necessary to get information about robot position using the odometer on the robot. Subsection 4.2.2 explains motion measurements and how reliable the Robotino[®] odometer is.

4.2.1 PID Control in Robotino[®]

When the system orders the robot to move with a certain velocity v , it takes time to accelerate the velocity from 0 to v . This acceleration should be accomplished as fast and smooth as possible to avoid big delays and robot oscillations. Robotino[®] uses closed-loop PID control to adjust speed of motors of its three omnidirectional wheels, and EA09 I/O board updates PID control loops as seen in Figure 4.4. Robotino MATLAB API provides functionality to tune PID parameters of those motors.

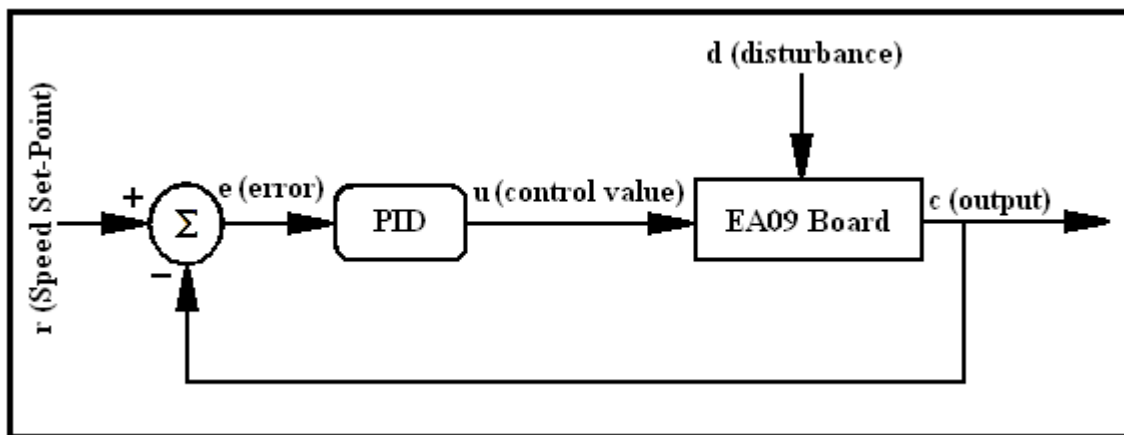


Figure 4.4 PID controller loops of motors of the Robotino[®] are updated by EA09 board.

Indeed, there are some methods developed to tune PID parameters to their optimal values in the closed-loop, but it is another research field and out of scope of this thesis.

Instead of using such methods, in this system, the parameters are set as $K_P = 60$, $K_I = 1.5$ and $K_D = 0$ after a set of trials and observations of motor behaviors.

4.2.2 Motion and Odometry Accuracy

As defining the problem, it is predicted that real motion amounts of the robot is different than the expected due to different factors like communication delays and positions of the three wheels that are placed by having 120° between each. Furthermore, the action set of the system consists of transitions with certain velocities, and PID control is used to accelerate the robot as mentioned in previous subsection. Since how long it takes for the robot to reach the required speed is not known, it is not feasible to assume how much the robot moved while performing the selected action. Hence, it is necessary to gather position information from odometer instead of using assumptions.

Odometer calculates the position of the robot by measuring wheel rotations. Though, using odometer to learn position of the robot brings some other questions like how accurate the odometer is or whether the velocity of the robot affects odometer accuracy. Tables 4.1, 4.2 and 4.3 list results for some single motions of the Robotino[®] with different linear and angular velocities for varied transition amounts. Since action set of the proposed system consists of motions through x -axis, θ and $-\theta$, motion through y -axis is not measured.

For all measurements in the following tables, initial position and velocities of Robotino[®] are set as $x = 0$, $y = 0$, $\theta = 0$, $v = 0$ and $w = 0$, and as mentioned above, PID parameters of the robot motors are set to $K_P = 60$, $K_I = 1.5$ and $K_D = 0$. Finally, the robot keeps capturing 5 image frames per second during the experiments to eliminate one of the causes of delays.

Table 4.1 Single robot motion through x-axis for distance d and linear velocity v .

Motion on x-axis (mm)	Linear Velocity (mm/s)	Final Positions					
		Real Values			Odometer Values		
		x (mm)	y (mm)	θ (deg)	x (mm)	y (mm)	θ (deg)
100	75	98.5	2.5	-2.0	100.1884	0.1099	-0.05
100	150	103.0	4.8	2.0	102.3579	0.5344	0.02
100	300	105.7	0.5	0.5	104.4813	0.5266	0.693
1000	75	1007.0	-7.2	-4.5	1014.3	-13.2668	-1.32
1000	150	1002.2	-22.0	-3.5	1004.0	-22.6210	-2.26
1000	300	1007.8	7.0	-6.5	998.108	-9.7256	-2.73
2000	75	1991.0	-48.5	-3.2	2000.6	-53.7815	-2.38
2000	150	2006.0	114.8	4.5	2004.6	38.6941	1.98
2000	300	2162.0	-30.0	-3.2	2005.3	-14.1646	-1.28

Table 4.2 Single robot rotation for amount of θ and angular velocity w .

Rotation (deg)	Angular Velocity (deg/s)	Final Positions					
		Real Values			Odometer Values		
		x (mm)	y (mm)	θ (deg)	x (mm)	y (mm)	θ (deg)
10	7.5	-1.0	0.5	11.0	-0.3744	0.6194	9.91
10	15	-0.5	0.5	11.5	1.142	-0.5786	10.22
10	30	-0.2	-2.0	12.5	-0.0479	-0.2393	10.51
90	7.5	2.8	5.9	90.0	1.3069	2.6053	89.9
90	15	0.0	3.0	89.5	4.7936	1.6707	89.71
90	30	0.2	4.0	90.8	0.5901	1.0602	89.69
180	7.5	-2.9	6.0	178.0	-3.7029	7.9776	179.81
180	15	-1.9	0.0	-177.5	-1.356	3.1025	179.94
180	30	3.0	0.2	-178.5	0.8837	2.6997	-178.79

Table 4.3 Single robot rotation for amount of $-\theta$ and angular velocity $-w$.

Rotation (deg)	Angular Velocity (deg/s)	Final Positions					
		Real Values			Odometer Values		
		x (mm)	y (mm)	θ (deg)	x (mm)	y (mm)	θ (deg)
-10	-7.5	-1.0	0.5	-9.9	1.5330	-1.0398	-9.92
-10	-15	0.0	-0.5	-9.5	0.0787	-0.5213	-9.76
-10	-30	7.0	-0.5	-12.5	-0.7226	-0.1306	-10.1
-90	-7.5	-0.5	-3.0	-88.8	-1.5384	0.3978	-89.96
-90	-15	-3.5	-4.5	-88.0	-1.9532	2.5471	-89.94
-90	-30	-0.5	0.0	-89.5	-0.5845	1.0286	-89.72
-180	-7.5	-6.2	0.0	179.5	-5.6863	7.8970	-179.89
-180	-15	-4.8	-0.5	179.5	-1.1337	1.0127	179.72
-180	-30	0.5	0.2	-179.0	0.8415	0.7057	179.98

Transition amount and linear and angular velocities can be changed by the user therefore it is important to be sure that the odometer uncertainty is in an acceptable interval for all velocities and transition amounts. Generally, it is approved that the error should be at most 0.02mm per 1mm of movement and 0.4° per 1° of turn. The average movement error between real position of the robot and the odometer values is calculated as 0.026mm per 1mm by using measurements in Table 4.1, and the error rates are 0.039° and 0.013° per 1° by values in Tables 4.2 and 4.3, respectively. Both tables show information about rotation of the robot thus the error can be averaged as 0.026° per 1° of turn.

The tables above show that odometry accuracy of the Robotino[®] is good enough for single motions yet the odometry uncertainty should not grow even if the Robotino[®] keeps moving. To check if there is a cumulative error rate, Robotino[®] is ordered to go for a distance and to move back to its original location and orientation as in Figure 4.5.

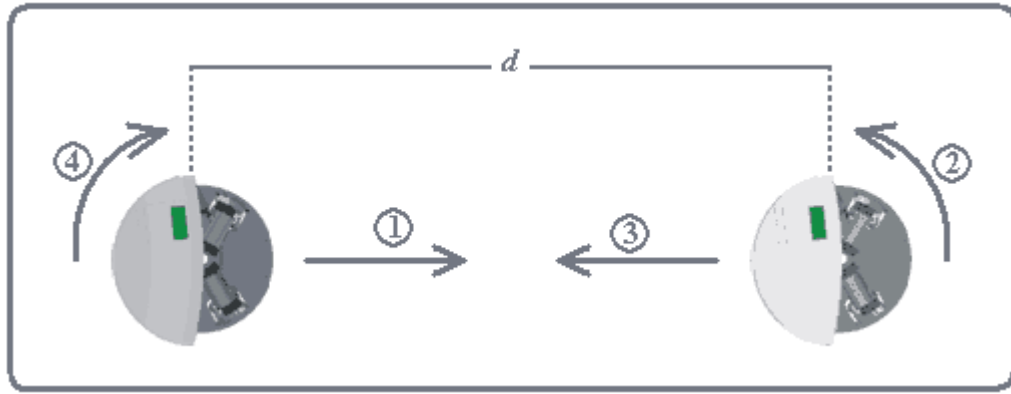


Figure 4.5 One empirical cycle: (1) go forward for distance d , (2) rotate for 180° , (3) go forward for distance d , (4) rotate for -180° .

Table 4.4 list results for different number of cycles of varied distances for fix linear and angular velocities of 150mm/s and $15^\circ/\text{s}$, respectively. Although the average errors are 0.011mm per 1mm and 0.005° per 1° and do not grow in long motions, the measured errors becomes larger. That is the reason of preferring to use the SLAM module instead of odometer to localize the robot, but for now the odometry accuracy can be accepted as sufficient to be used in the system.

Table 4.4 Results of empirical cycles shown in Figure 4.5.

Distance (mm)	Number of Cycles	Final Positions					
		Real Values			Odometer Values		
		x (mm)	y (mm)	θ (deg)	x (mm)	y (mm)	θ (deg)
1000	1	-0.5	0.2	-1.0	-1.0857	19.0449	-2.32
1000	3	-10.8	-10.0	-2.0	-7.3683	33.2082	-5.24
2000	1	11.0	-106.0	6.2	5.0226	-112.7069	7.0
2000	3	6.2	131.5	-2.5	33.6395	-166.6017	10.29

4.2.3 Experimental Results

After the learning phase of the system executed, the user can test how well the system learned what to do in both simulated and real platforms. In the test phase, the learned Q-Values are used to decide actions. If the Q-Value matrix converges to optimal sufficiently, it is expected for the robot to reach goal by avoiding obstacles. Figure 4.6 shows an example execution of the system on the Robotino[®]. The path of the robot is illustrated by a black line.

Figure 3.5 illustrates the resulting system interface at the termination of the system execution demonstrated in Figure 4.6. As well Figure 4.7 shows an example interface during the test execution in Figure 4.6. As can be seen in the figures, after a successful learning phase, the system can navigate the robot in the test phase regardless of executing on the simulated or real platform.

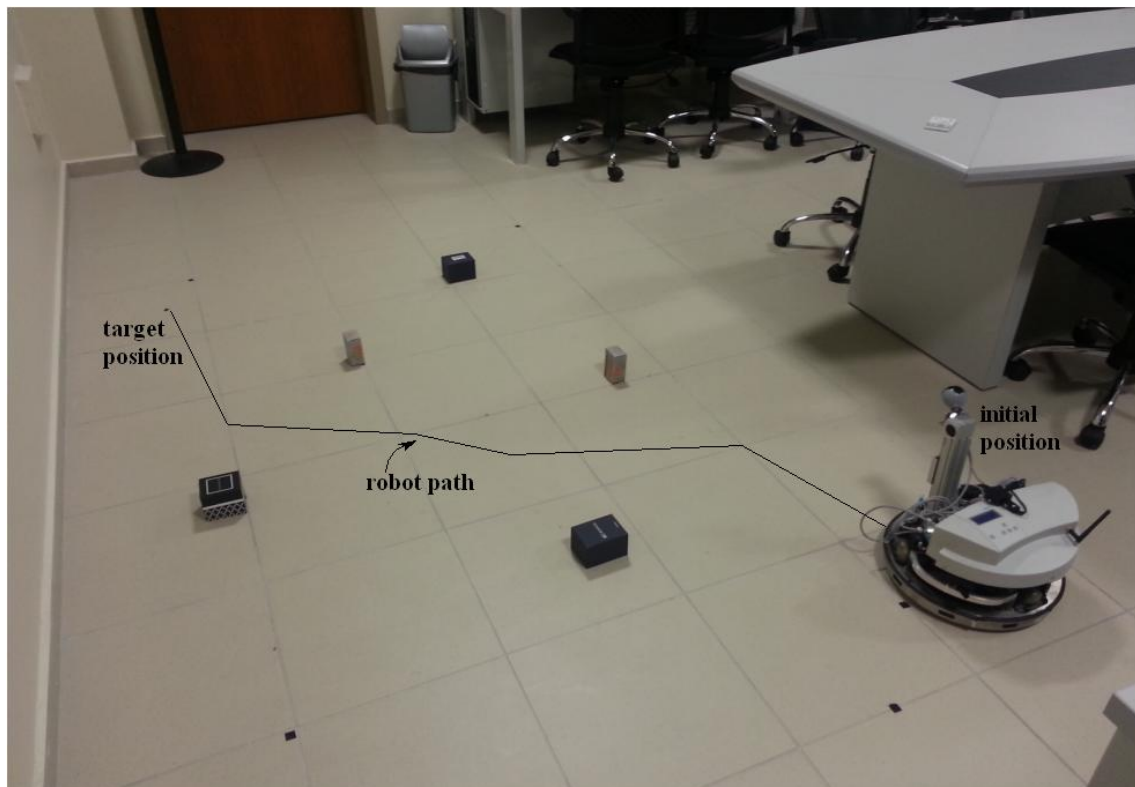


Figure 4.6 The Robotino[®] can reach the target by avoiding obstacles in the test phase.

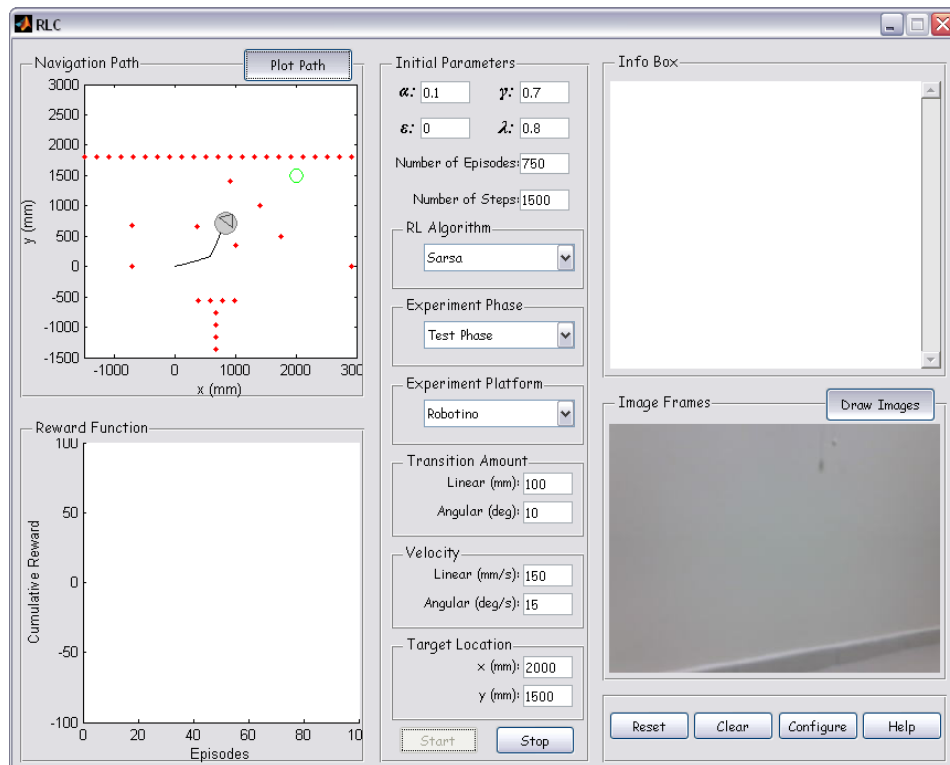


Figure 4.7 The interface during the test execution in Figure 4.6.

CHAPTER 5

CONCLUSION

The literature review performed within the scope of this study shows that reinforcement learning algorithms are widely used for optimal control problems, for instance, mobile robot navigation, even though it can be problematic to design and implement a reinforcement learning algorithm for continuous and stochastic environments. It is essential to define the state and action sets to perform a successful learning since continuous environments have infinite possible states and actions. Thus, discretizing the continuous space determines the performance of the implemented RL algorithm. Another decision criterion is how to describe the reward function that is the response of the environment to the actions of the intelligent agent.

After two $TD(\lambda)$ based RL algorithms, $Sarsa(\lambda)$ and $Q(\lambda)$, are implemented in the system, their learning performances are compared, and as stated in the literature, it is seen that $Sarsa(\lambda)$ which is an *on-policy* method converges to optimal values faster than $Q(\lambda)$ that is an *off-policy* algorithm. Besides, $Sarsa(\lambda)$ learns the environment on which the system executed with a higher rate of 90% than that of $Q(\lambda)$ of 70%. Nevertheless, no matter which algorithm is used, it is insufficient to find the ideal Q-Value matrix that is optimal for all environments since not all situations can be represented in continuous-state environments.

Consequently, this thesis may be beneficial to understand the main idea of reinforcement learning algorithms by emphasizing subtle implementation details that are effective on the performance of RL based control systems. Additionally, the presented demonstrations of two RL algorithms for mobile robot navigation problem can be a base for applications which are used to simplify human lives.

5.1 FUTURE WORK

The implemented system in this study can be enhanced to increase its speed and performance. First of all, Q-values in the algorithms used by the system are represented in tabular form which requires large space in the memory and complex mathematical calculations. Instead, it is possible to integrate supervised learning to the system to reduce memory requirement and provide faster convergence to optimal policy. Theoretically, all supervised learning algorithms can be integrated, yet not all of them are equally suitable for RL. Hence, it is necessary to search an ideal algorithm for the implemented system. Besides, the environment in the current system is represented by only 81 states which are actually not enough to discretize all continuous situations sufficiently, and this causes a decrease in learning rate. As a solution, the state and action sets can be redefined to improve learning of the system.

After all, one more property should be integrated in order to obtain a completely accomplished system: The vision. It is also desired to combine the proposed system in the thesis and VBAN developed by Öztürk (2012). In this manner, the visionary feature of the mobile robot, the Robotino[®], on which the system is performed can be integrated in order to provide an autonomous navigation.

REFERENCES

- Anderson, C.W. and Kretchmar, R.M., *Solving Optimal Control and Search Problems with Reinforcement Learning in MATLAB*, 2012
<http://www.cs.colostate.edu/~anderson/res/rl/matlabpaper/rl.html>
- Anderson, C.W., Hittle, D., Katz, A. and Kretchmar, R.M., "Synthesis of Reinforcement Learning, Neural Networks, and PI Control Applied to a Simulated Heating Coil", *Journal of Artificial Intelligence in Engineering*, Vol. 11, #4, pp. 423-431, 1997.
- Anderson, C.W., "Learning to Control an Inverted Pendulum with Neural Networks", *IEEE Control Systems Magazine*, April 1989, Vol. 9, No. 3, pp. 31-37, 1989.
- Anderson, C.W., "Strategy Learning with Multilayer Connectionist Representations", *Proceedings of the Fourth International Workshop on Machine Learning*, pp. 103-114, Irvine California, 1987.
- Argall, B.D., Chernova, S., Veloso, M. and Browning B., "A Survey of Robot Learning from Demonstration", *Robotics and Autonomous Systems*, Vol. 57, pp. 469-483, May 2009.
- Bellman, R.E., *Dynamic Programming*, Princeton University Press, Princeton, New Jersey, 1957a.
- Bellman, R.E., "A Markov Decision Process", *Journal of Mathematical Mechanics*, Vol. 6, No. 5, pp. 679-684, 1957b.
- Busquets, D., de M'antaras, R.L., Sierra, C. and Dietterich, T.G., "Reinforcement Learning for Landmark-based Robot Navigation", *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, Part 2, pp. 841-842, 2002.
- Cassandra, A.R., Littman, M., Kaelbling L., *POMDPs for Dummies*, 1999,
<http://www.cs.brown.edu/research/ai/pomdp/tutorial/index.html>
- Da Silva, B.C., *Reinforcement Learning Repository at University of Massachusetts, Amherst*, 2012, <http://www-anw.cs.umass.edu/rlr/>
- Khriji, L., Touati, F., Benhmed, K. and Al-Yahmedi, A., "Mobile Robot Navigation Based on Q-Learning Technique", *International Journal of Advanced Robotic Systems*, Vol. 8, No. 1, pp. 45-51, 2011.

- Lagoudakis, M.G. and Parr, R., “Model-free Least Squares Policy Iteration”, *Proceedings of Advances in Neural Information Processing Systems (NIPS-14)*, 2001.
- Maček, K., Petrović, I. and Perić, N., “A Reinforcement Learning Approach to Obstacle Avoidance of Mobile Robots”, *Proceedings of 7th International Workshop on Advanced Motion Control*, Habakuk - Maribor, 3 – 5 June 2002, pp. 462 - 466, Slovenia, 2002.
- Markov, A.A., *Theory of Algorithms*. [Translated by Jacques J. Schorr-Kon and PST staff] Imprint Moscow, Academy of Sciences of the USSR, Original title: *Teoriya Algorifmov*, 1954.
- Martín, J.A. and de Lope, J., “Ex(α): An Effective Algorithm for Continuous Actions Reinforcement Learning Problems”, *Proceedings of 35th Annual Conference of the IEEE Industrial Electronics Society (IECON 2009)*, Oporto-Portugal, November 2009, pp. 2063-2068, IEE, 1991.
- McCallum, R.A., “Instance-Based State Identification for Reinforcement Learning”, *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 7, pp. 377-384, The MIT Press, 1995.
- Menegatti, E., Cicirelli, G., Simionato, C., D’Orazio, T. and Ishiguro, H., “Explicit Knowledge Distribution in an Omnidirectional Distributed Vision System”, *Proceedings of 2004 IEEE WRSJ International Conference on Intelligent Robots and Systems*, Sendai, 28 September-2 October 2004, Vol. 3, pp. 2743-2749, Japan, 2004.
- Moore, A.W. and Atkeson, C.G., “Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time”, *Machine Learning*, Vol. 13, pp. 103-130, October 1993.
- Öztürk, C.N., *Vision-Based Autonomous Navigator*, M.S. Thesis, Fatih University, 2012.
- Rummery, G.A. and Niranjan, M., *On-line Q-learning Using Connectionist Systems*, Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, 1994.
- Russell, S.J. and Norvig, T., *Artificial Intelligence: A Modern Approach*, 2nd edition, Prentice Hall, New Jersey, 2003.
- Shaker, M., Smith, M.N.R., Yue, S. and Duckett, T., “Vision-Based Landing of a Simulated Unmanned Aerial Vehicle with Fast Reinforcement Learning”, *Proceedings of 2010 International Conference on Emerging Security Technologies*, Canterbury, 6-8 September 2010, pp. 183-188, United Kingdom, 2010.
- Sutton, R.S. and Barto, A.G., *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge, Massachusetts, London, England, 2005.

- Watkins, C.J.C.H., *Learning from Delayed Rewards*, Ph.D Thesis, Cambridge University, 1989.
- Zhu, W. and Levinson, S., “Vision-based Reinforcement Learning for Robot Navigation”, *Proceedings of the International Joint Conference on Neural Networks*, Washington DC, 15-19 July 2001, Vol. 2, pp. 1025-1030, 2001.
- Zhumatiy, V., Gomez, F., Hutter, M. and Schmidhuber J., *Metric State Space Reinforcement Learning for a Vision-Capable Mobile Robot*, IDSIA, Galleria 2, CH-6928 Manno-Lugano, Switzerland, 2006.

APPENDIX A

ROBOTINO

The mobile robot system Robotino[®] by Festo Didactic is a platform with an open mechanical interface for the integration of additional mechanical devices. Power is supplied via two 12V lead gel batteries which permit a running time of up to two hours. The scope of delivery likewise includes a charging device. Robotino[®] is driven by 3 independent, omnidirectional drive units. They are mounted at an angle of 120° to each other. The three omnidirectional drive units of Robotino[®], defines the robot as being holonomic, meaning that the controllable degrees of freedom equals the total degrees of freedom of the robot. The drive units are integrated in a sturdy, laser welded steel chassis. The chassis is protected by a rubber bumper with integrated switching sensor. Robot dimensions:

- Diameter: 370mm
- Height including housing: 210mm
- Overall weight: approx. 11kg
- Maximal payload of about 6kg

A.1 DRIVE UNIT

Each of the 3 drive units consists of the following components:

- DC Dunker motor with nominal speed of 3600rpm.
- Omnidirectional wheels of diameter of 80mm.
- A gear transmission ratio of 16:1 and allows speeds of up to 10km/h.

- Incremental encoder with a resolution of 2048 increments per motor rotation.

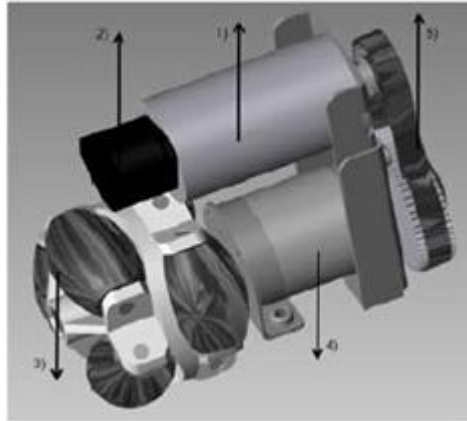


Figure A.1 Drive unit with (1) motor, (2) encoder, (3) omnidirectional wheel, (4) planetary gear, (5) toothed belt.

The motor speed is controlled via a PID controller implemented on EA09 control board. Omnidirectional wheels are all based on the same general principle: while the wheel proper provides traction in the direction normal to the motor axis, the wheel can slide frictionless in the motor axis direction. In order to achieve this, the wheel is built using smaller wheels attached along the periphery of the main wheel, see Figure A.1.

A.2 SENSORS

Robotino[®] is equipped with 9 infrared distance measuring sensors which are mounted in the chassis at an angle of 40° to one another. Robotino[®] can scrutinize all surrounding areas for objects with these sensors. Each of the sensors can be queried individually via the controller board. The sensors are capable of accurate or relative distance measurements to objects at distances of 4 to 30cm. Sensor connection is especially simple including just one analogue output signal and supply power. The sensors' evaluation electronics determine distance and read it out as an analogue signal.

The anti-collision sensor is comprised of a switching strip which is secured around the entire circumference of the chassis. A switching chamber is located inside a plastic profile. 2 conductive surfaces are situated inside the chamber, between which a given clearance is maintained. These surfaces are short circuited when even minimal pressure is applied to the strip. Collisions with objects at any point on the housing are detected and, for example, Robotino[®] is brought to a standstill.

The inductive proximity sensor is supplied as an additional component. It serves to detect metallic objects on the floor and is used for continuous-path control. It reads out signals of varying strength depending on whether it is located in the middle or at the edge of the metal strip. The output voltage is 0 to 10V. The sensing range is 0 to 6mm.

Robotino[®] also included two diffuse sensors. Flexible fibre-optic cables are connected to a fibre-optics unit which works with visible red light. Reflected light is detected. Different surfaces and colors produce different degrees of reflection.

Robotino[®] is equipped with a color webcam of resolution 320x240 pixels. The webcam is equipped with an USB interface and provides an integrated jpeg compression. It supports a color depth of 24 bit true color and a VGA resolution with 15fps. The reason for jpeg compression is that image processing can be done on an external PC via WLAN connection.

Additionally, Gyroscope CruzCore Sensor is mounted on Robotino[®]. Since mechanical gyroscopes are too large, heavy and susceptible, the CruzCore XG1000 is a digital gyroscope in a handy housing unit with an integrated USB interface. It can measure angular velocities of up to $\pm 100^\circ/\text{s}$ with a precision of $< 0.1^\circ/\text{s}$. It has a maximum drift of $10^\circ/\text{h}$.

A.3 CONTROLLER BOARD

The controller housing is connected to the wiring in the chassis via one plug-in. Thus, it is easy to take off the controller housing and to have direct access to the mechanical system. The controller system of Robotino[®] is divided into two parts – an embedded PC and a microcontroller interface card.

The interface card consists of EA09 control board which provides a PID controller for the three motors. Further, they can provide the control for another motor. The interface card communicates via a serial interface, RS232, with the embedded PC. Also the microprocessors send and read the integrated sensor signals. An FPGA (Xilinx Spartan3) is used to read the encoder values for the motors. Therefore, the control board which is LPC2377 32-bit microcontroller also provides an open I/O interface to connect additional sensors and actuators:

- 8 analogous inputs 0-10V, 50Hz
- 8 digital inputs and outputs (24V, short-circuit-proof)
- 2 relays for additional actuators

The main controller is the embedded PC 104 plus controller with the 500MHz processor AMD LX800. The PC has a SDRAM of 128MB and is provided with a 1GB flash card. There are numerous communication interfaces on board:

- Ethernet
- 2 x USB
- Parallel port and VGA port
- Wireless LAN Access Point following the standards 802.11.g and 802.11.b.
The access point can be switched into a client mode. As an option you may use WPA2-ssl coding.

A.4 SOFTWARE

There is a Linux operating system with real-time kernel running on the embedded PC 104 and provided on the 1GB CF card. The main part of the controller is Robotino[®] server, a real-time Linux application. It controls the drive units and provides interfaces to communicate with external PC applications via W-LAN. There is an API with libraries which allow creating applications for Robotino[®] in numerous programming languages which are C++, C, C#, .Net, JAVA, MATLAB, Simulink and Labview which is a platform developed by Festo.