University of Girona

Department of Electronics, Informatics and Automation

PhD Thesis

A Proposal of a Behavior-based Control Architecture with Reinforcement Learning for an Autonomous Underwater Robot

Thesis presented by

Marc Carreras Pérez,

to obtain the degree of:
PhD in Computer Engineering.

Supervisor: Dr. Pere Ridao Rodríguez

Girona, May 2003

To my family, and especially, to Pepa.

Acknowledgements

I would like to express in these lines my gratitude to those people who have helped me in the accomplishment of this PhD thesis.

I must start by acknowledging my first director, Joan Batlle, for pushing me some years ago into the exciting field of research. I thank him for transmitting to me so much enthusiasm and motivation. At the same time, I would like to thank my current director and friend Pere Ridao, for helping me with the ideas and experiments of this dissertation. I appreciate his patience in listening to all my questions and problems and his effort in revising all my work.

I want also to express my gratitude to the supervisors who have welcomed me at their research centers. To Prof. Geoff Roberts from the University of Wales College, Newport, for his good advice when I was there at the beginning of this research. Also, to Prof. Junku Yuh from the University of Hawaii, for guiding me through the achievements of this thesis and for allowing me to perform my experiments in his laboratory.

I would like also to mention other researchers for their collaboration and kindness; Prof. Antonio Tiano from the University of Pavia, and Dr. Biel Oliver and his team from the University of the Balearic Islands.

I must also thank all the people of the *Computer Vision and Robotics* research group and the people of the department of *Electronics, Informatics and Automation* for creating a pleasant work environment. I would like to extend my warmest thanks to JordiF, Rafa, Xevi, Quim, JoanM and PepF, I really appreciate the friendship you extended. My special gratitude to the PhD students sharing the laboratory with me: Jessica, Tudor, Zoran, Dino, David and Andres. Thanks for your daily help and humor. And my gratitude to the other colleagues of the group: XMangui, XMuñoz, XLlado, Ela, JordiP, Radu and Carles, for their support and friendship.

My very special gratitude to the invaluable support of Josep Tomas, Lluis Magi and Toni for technical aspects, to Isela and Sonia for administrative affairs, and to Christine for revising my English. Thanks to all for attending my requests in such a short time!

A general acknowledgement goes to all the undergraduate students who have worked in the laboratory on our research projects. Among them I must mention Ferran and OscarP who contributed directly to the development of the underwater robot URIS.

I cannot forget the friends I met during my two stays in Newport and Hawaii. I must thank Ioannis and Vivianne for their hospitality when I was in Newport. Also, the people of the *Mechatronics Research Centre* for teaching me how to prepare some cakes! From my stay in Hawaii, I must acknowledge my good friend Alberto, who helped me to discover the most amazing places on the island. Also, Ursula and Sergio for their great friendship. I'm sure the success of that stay was in part because of you. Finally, my gratitude to all the people of the *Autonomous Systems Laboratory* of the University of Hawaii, for taking me in so kindly and for helping me with the experiments. My special gratitude to Hyun Taek, Side, Scott, Kaikala, Giacomo, Mick, Wesley and Song.

And last, but not least, my most especial gratitude is to my parents, to all my family and friends for trusting in me and for their support during all these years. Finally, the most important acknowledgement goes to Pepa for her support and her patience. This thesis is especially dedicated to you.

Contents

List of Figures

List of Tables

Glossary

AFSM	Augmented Finite States Machines
AI	Artificial Intelligence
ASD	Action Selection Dynamics
AUV	Autonomous Underwater Vehicle
BBCA	Behavior-based Control Architectures
CMAC	Cerebellar Model Articulation Controller
DOF	Degree Of Freedom
DP	Dynamic Programming
FMDP	Finite MDP
FSM	Finite State Machine
GA	Genetic Algorithm
HHCN	Hierachical Hybrid Coordination Node
HSL	Hue Saturation Luminance
IDL	Interface Definition Language
MC	Monte Carlo
MDP	Markov Decision Process
NQL	Neural-Q_learning
PDL	Process Description Language
POMDP	Partially Observable MDP
QL	Q_Learning
RDF	Radial Basis Function
RGB	Red Green Blue
RL	Reinforcement Learning
RLP	Reinforcement Learning Problem
ROV	Remotely Operated Vehicle
SMDP	Semi MDP
SONQL	Semi-Online Neural-Q_learning
TD	Temporal Difference
UUV	Unmanned Underwater Vehicle

Chapter 1

Introduction

Technical advances in our society have demonstrated great achievements in many diverse fields like transportation, communication, manufacturing and computation. Machines help us in our daily activities, extending our natural acting capabilities. Also, a diverse set of perception systems allow us to extend our natural senses. However, there are few autonomous devices able to integrate the two concepts: *sensing* and *acting*. *Autonomous robots* is a research subject which gathers topics like sensing, actuation, powering, communication, control theory and artificial intelligence. The goal residing in the design of an autonomous robot is the development of a machine able to fulfill a task in the real world. Tasks which are similar to those humans commonly do without effort. An autonomous robot must be able to act in the world by changing its state or by moving itself through it. It must also be able to sense the state of the world, which requires the interpretation of the sensor information. Finally, it must be able to decide *what to do*, how to relate the state of the environment to its action possibilities in order to achieve a predefined goal.

The state/action relation, which is trivial for human intelligence, represents a very difficult task for a computerized system, specially in unstructured and unknown environments. To overcome this, two different problems have to be dealt with. The first is the *state interpretation* problem, that is, the understanding of what the sensors perceive. The second is the *action-decision* problem which consists of deciding the movements of the robot to accomplish the mission. To solve the action-decision problem it is assumed that the state has been correctly interpreted. The field of *Artificial Intelligence* is usually applied to autonomous robots for solving both of these problems. However, a feasible solution can be obtained in only a few set of cases.

This thesis is concerned with the field of autonomous robots and the problem of action-decision. The thesis analyzes some Artificial Intelligence

techniques which can be successfully applied to the control of an autonomous robot. In particular, the thesis is based on the topics of *Behavior-based Robotics* and the *Reinforcement Learning* theory. The experimental platform used in this research is an Autonomous Underwater Robot (AUV). Throughout this chapter the main aspects which have conditioned this research project will be overviewed. First, the research antecedents which were found will be presented in the beginning of this thesis. Then, the goal of the thesis will be pointed out. Finally, the chapter will finish with the outline of this dissertation.

1.1   Previous Research Work

The research presented in this thesis has been fulfilled in the *Computer Vision and Robotics* research group of the University of Girona. This group has been doing research in underwater robotics since 1992 (supported by several programs of the Spanish commission MCYT). The main contribution throughout the past years is the development of two *Unmanned Underwater Vehicles* (UUV). The first prototype, called GARBI, was developed in collaboration with the Polytechnical University of Catalonia. This vehicle was conceived as a *Remotely Operated Vehicle* (ROV). The second prototype, URIS, was fully developed in the University of Girona and was designed as an *Autonomous Underwater Vehicle* (AUV).

The design of an autonomous vehicle requires a solution for the action decision problem, which was introduced at the beginning of this chapter. A *Control Architecture* is the part of the robot control system in charge of making decisions. An autonomous robot is designed to accomplish a particular *mission*, and the control architecture must achieve this mission by generating the proper actions. The control architecture is also known as the *High-Level Controller* since it decides the movements to be followed by the robot. Another kind of controller, which is also present in any autonomous robot, is the *Low-Level Controller*. This one will make the movements proposed by the high-level controller and drive the required actuators. The main difference between a ROV and an AUV resides in the control architecture. In a ROV, the low-level controller will make the movements a human proposes by means of a joystick for example. In an AUV, the control system needs the presence of a control architecture or high-level controller to generate these actions.

The design of the autonomous robot URIS and previously, the adaptation of GARBI from a ROV to an AUV, led to the development of a new control architecture. The main feature of this architecture, called O^2CA^2, was the break down of the whole robot control system into a set of objects. The

parallel execution of these objects allowed a real-time execution of the control system. In addition, some of these objects represented primitive behaviors of the robot. The control architecture used these behaviors to propose the best movement at any given moment.

The development of the O^2CA^2 control architecture led to the PhD. thesis of Dr. Pere Ridao [Ridao, 2001]. The thesis presented in this document is a continuation of the work started with the O^2CA^2 architecture. In this thesis, the set of behaviors which constitute the action-decision methodology are studied. The thesis investigates some implementation aspects which influence the overall performance of the robot. To do this, the topic of *Behavior-based Robotics* was surveyed. Also, the thesis explored the use of learning algorithms to improve the efficiency of the behaviors. A learning theory, called *Reinforcement Learning*, was also overviewed and applied. Finally, the proposals of this thesis were tested with the underwater robot URIS.

1.2   Goal of the thesis

After the description of the research antecedents, the goal of this thesis is stated. The general purpose is summarized as:

"The development of a robot control architecture for an AUV able to achieve simple tasks and exhibit real-time learning capabilities"

This goal was selected in order to continue the research line started with the development of the O^2CA^2 control architecture. On the one hand, it had the intention of exploiting this architecture by refining some performance aspects. But on the other hand, it opened the new research field of robot learning, which is one of the most active topics in robotics. The application of learning algorithms in the control architecture was certainly the most important purpose of this dissertation. Finally, the fundamental goal on which this research project was based is the experimentation with real robots in real-time computation. The basic premise was to demonstrate the research advances with real experiments.

The general goal of the thesis can be divided into three more specific points:

Behavior-based control layer. Design of a behavior-based control system which will be contained in the overall control architecture with the purpose of accomplishing simple tasks. A task is intended as one of the phases in which a mission can be divided. It is assumed that the sequential achievement of a set of tasks entails the achievement of the

mission. The behavior-based control system must assure the safety of the robot while demonstrating a high control performance.

Reinforcement Learning-based behaviors. Integration of a reinforcement learning algorithm in the control architecture. This learning theory will be applied to the acquisition of the internal structure of a robot behavior. The purpose of using Reinforcement Learning is to reduce the required human work in the development of a new robot behavior. Instead of implementing the action-decision rules, the designer need only to define the goal of the behavior.

Experimentation with an AUV. Evaluation of the proposed control and learning systems with real experiments using an Autonomous Underwater Vehicle. The feasibility and limitations of these approaches must be experimentally tested with the available systems and resources.

1.3   Outline of the thesis

The contents of this thesis can be divided into three parts. The *first* part overviews the field of Behavior-based Robotics (Chapter 2) and proposes an architecture (Chapter 3). The *second* part overviews the field of Reinforcement Learning (Chapter 4) and proposes an algorithm for its application to robotics (Chapter 5). Finally, the *third* part of the thesis shows the experimental results by first describing the set-up (Chapter 6) and then the results (Chapter 7). A brief description of each chapter is next presented.

Chapter 2 *Behavior-based Control Architectures.* This presents the field of "Behavior-based Robotics" and overviews some classic architectures. A simulated task is used to implement and compare these architectures.

Chapter 3 *Hybrid Coordination of Behaviors.* This proposes a control architecture based on a behavior-based control layer. A Hybrid Coordination methodology is contained in this layer which attempts to guarantee the robustness while providing a high trajectory performance.

Chapter 4 *Reinforcement Learning.* This presents the theory of Reinforcement Learning and overviews the most important research issues among which, the generalization problem is specifically treated for its strong influence in robotics. Some methodologies to deal with this are presented.

Chapter 5 *Semi-Online Neural-Q_learning.* This proposes an algorithm to learn robot behaviors using Reinforcement Learning. The generalization problem is faced with a Neural Network and a database of the most representative learning samples.

Chapter 6 *URIS' Experimental Set-up.* This details the main features of URIS AUV, and all the systems designed to perform the experiments of the SONQL algorithm and the hybrid coordination methodology.

Chapter 7 *Experimental Results.* This shows the experimental results obtained with URIS. The SONQL is tested with a "target following" behavior. The hybrid coordination system is applied to coordinate this behavior with some manually implemented behaviors. Also, the SONQL is tested with a Reinforcement Learning benchmark to demonstrate its generalization capability.

Chapter 8 *Conclusions.* This concludes the thesis by summarizing the work and points out the contributions and future work. It also comments the research evolution and the publications accomplished during this research project.

Chapter 2

Behavior-based Control Architectures

This chapter overviews the field of Behavior-based Control Architectures also known as Behavior-based Robotics [Arkin, 1998]. It first reviews the history of control architectures for autonomous robots, starting with traditional methods of Artificial Intelligence. The most important facts are revised up to the inception of the field of Behavior-based Robotics. The chapter then describes the principles and main features to be found in a behavior-based system. After this general introduction to the field, four representative behavior-based architectures are described as they were originally designed. The architectures are *Subsumption, Action Selection Dynamics, Motor Schema* and *Process Description Language.* In addition to its description, the architectures are also tested with an experimental task, in which an AUV must be controlled. Finally, using the obtained results the chapter analyzes how the coordination methodology influences to the control performance.

2.1 Behavioral-based Robotics versus Traditional AI

The first attempt at building autonomous robots began around the mid-twentieth century with the emergence of Artificial Intelligence. The approach begun at that time is now referred to as "Traditional AI", "Classical AI", "Deliberative approach" or "Hierarchical approach". Traditional AI relies on a centralized world model for verifying sensory information and generating actions in the world. The design of the control architecture is based on a top-down philosophy. The robot control architecture is broken down into

Figure 2.1: Phases of a deliberative control architecture.

an orderly sequence of functional components [Brooks, 1991a] and the user formulates explicit tasks and goals for the system. The sequence of phases usually found in a deliberative control architecture are next described; see also Figure 2.1.

1. **Perception**. In the first component, a sensor interpreter resolves noise and conflicts in the sensory data. Perception algorithms are used to find characteristics and objects within the environment.

2. **Modelling**. With the data obtained from perception, the world modelling component builds a symbolic representation of the world. This representation contains geometric details of all objects in the robot's world with their positions and orientations.

3. **Planning**. The planner then operates on these symbolic descriptions of the world and produces a sequence of actions to achieve the goals given by the users.

4. **Task execution**. This function controls the execution of the planned tasks generating the set-points for each actuator.

5. **Motor control**. This control system is used to control the actuators in accordance with the set-points.

The principal characteristics which all AI approaches have in common are:

• **Hierarchical structure**. The main goals are divided into different tasks, sub-tasks, etc, in a hierarchical manner. Higher levels in the hierarchy provide sub-goals for lower subordinate levels. The tasks are accomplished using a top-down methodology.

• **Sequential processing**. These processes are executed in serial form starting with the sense activities, moving through the modelling and planning, ending with the actuation.

- **Symbolic planner**. The planner reasons, basing itself on a symbolic world model. The world must be generated by linking the physical perceptions to the corresponding symbols.

- **Functionally compartmentalized**. There is a clear subdivision of the different tasks which must be carried out. Each component in the control architecture will be in charge of only one of these functions.

In the 1970's one of the earliest autonomous robots was built using a deliberative control architecture [Fikes and Nilsson, 1971][Nilsson, 1984]. This robot was called Shakey and inhabited a set of specially prepared rooms. It navigated from room to room, trying to satisfy a given goal. While experimenting with this robot, new difficulties were found. The planning algorithms failed with non-trivial solutions, the integration of the world representations was extremely difficult and finally the planning did not work as well as was hoped. The algorithms were improved and better results were obtained. However, the environment was adapted totally to the robot's perceptive requirements. Many other robotic systems have been built with the traditional AI approach [Albus, 1991, Huang, 1996, Lefebvre and Saridis, 1992, Chatila and Laumond, 1985, Laird and Rosenbloom, 1990]. Nevertheless, traditional approaches still have problems when dealing with complex, non-structured and changing environments. Only in structured and highly predictable environments have they proved to be suitable. The principal problems found in Traditional AI can be listed as:

- **Computation**. Traditional AI requires large amounts of data storage and intense computation. For an autonomous mobile robot this can be a serious problem.

- **Real time processing**. The real world has its own dynamics and, for this reason, systems must react fast enough to perform their tasks. Most often, traditional AI is not fast enough because the information is processed centrally. Modelling and planning are long sequential processes, and the longer they take, the more changed the world will be when the robot decides to act. The agent needs simple, multiple and parallel processes instead of only a few long sequential processes.

- **Robustness and Generalization**. Traditional AI usually lacks in generalization capacity. If a novel situation arises, the system breaks down or stops all together. Also, it does not take into consideration problems of noise in the sensory data and actuators when giving its symbolic representation.

- **The accurate world model**. In order to plan correctly, the world model must be very accurate. This requires high-precision sensors and careful calibration, both of which are very difficult and expensive.

- **The Frame problem**. This problem arises when trying to maintain a model of a continuously changing world. If the autonomous robot inhabits a real world, the objects will move and the light will change. In any event, the planner needs a model with which to plan.

- **The Symbol-grounding problem**. The world model uses symbols, such as "door", "corridor", etc, which the planner can use. The Symbol-grounding problem refers to how the symbols are related to real world perceptions. The planner is closed in a symbolic world model while the robot acts in the open real world.

In the middle of the 1980s, due to dissatisfaction with the performance of robots in dealing with the real world, a number of scientists began rethinking the general problem of organizing intelligence. Among the most important opponents to the AI approach were Rodney Brooks [Brooks, 1986], Rosenschein and Kaelbling [Rosenschein and Kaelbling, 1986] and Agre and Chapman [Agre and Chapman, 1987]. They criticized the symbolic world which Traditional AI used and wanted a more reactive approach with a strong relation between the perceived world and the actions. They implemented these ideas using a network of simple computational elements, which connected sensors to actuators in a distributed manner. There were no central models of the world explicitly represented. The model of the world was the real one as perceived by the sensors at each moment. Leading the new paradigm, Brooks proposed the "Subsumption Architecture" which was the first approach to the new field of "Behavior-based Robotics".

Instead of the top-down approach of Traditional AI, Behavior-based systems use a bottom-up philosophy like that in Reactive Robotics. Reactive systems provide rapid real-time responses using a collection of preprogrammed rules. Reactive systems are characterized by a strong response, however, as they do not have any kind of internal states, they are unable to use internal representations to deliberate or learn new behaviors. On the other hand, Behavior-based systems can store states in a distributed representation, allowing a certain degree of high-level deliberation.

The Behavior-based approach uses a set of simple parallel behaviors which react to the perceived environment proposing the response the robot must take in order to accomplish the behavior (see Figure 2.2). There are no problems with world modelling or real-time processing. Nevertheless, another difficulty has to be solved; how to select the proper behaviors for robustness
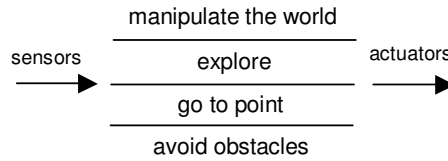
Figure 2.2: Structure of a Behavior-based control architecture.
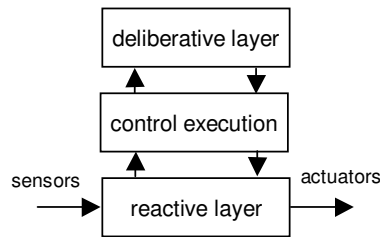
Figure 2.3: The hybrid control architecture structure.

and efficiency in accomplishing goals. New questions also appeared which Traditional AI was not taking into consideration; how to adapt the architecture in order to improve its goal-achievement, and how to adapt it when new situations appear. This powerful methodology demonstrated simplicity, parallelism, perception-action mapping and real implementations.

Behavior-based Robotics has been widely used and investigated since then. This new field has attracted researchers from many and divers disciplines such as biologists, neuroscientists, philosophers, linguists, psychologists and, of course, people working with computer science and artificial intelligence, all of whom have found practical uses for this approach in their various fields of endeavor. The field of Behavior-based Robotics has also been referred to as the "New AI" or, under a more appropriate denomination for the fields mentioned above, "Embodied Cognitive Science".

Finally, some researchers have adopted a hybrid approach between Traditional AI and Behavior-based Robotics. Hybrid systems attempt a compromise between bottom-up and top-down methodologies. Usually the control architecture is structured in three layers: the deliberative layer, the control execution layer and the functional reactive layer (see Figure 2.3). The deliberative layer transforms the mission into a set of tasks which perform a plan. The reactive behavior-based system takes care of the real time issues related to the interactions with the environment. The control execution layer interacts between the upper and lower layers, supervising the accomplishment of the tasks. Hybrid architectures take advantage of the hierarchical
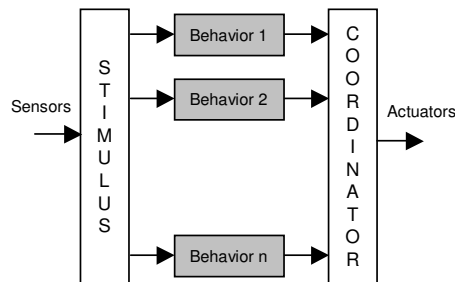
Figure 2.4: Structure of behavior-based CA.

planning aspects of Traditional AI and the reactive and real time aspects of behavioral approaches. Hybrid architectures have been widely used. Some of the best known are AuRA [Arkin, 1986], the Planner-Reactor Architecture [Lyons, 1992] and Atlantis [Gat, 1991] used in the Sojourner Mars explorer.

2.2   Fundamentals of Behavior-based Robotics

Behavior-based Robotics is a methodology for designing autonomous agents and robots. The behavior-based methodology is a bottom-up approach inspired by biology, in which several behaviors act in parallel achieving goals. Behaviors are implemented as a control law mapping inputs to outputs. They can also store states constituting a distributed representation system [Mataric, 1999]. The basic structure consists of all behaviors taking inputs from the robot's sensors and sending outputs to the robot's actuators, see Figure 2.4. A coordinator is needed in order to send only one command at a time to the motors.

The internal structure of a behavior can also be composed of different modules interconnected by sensors, various other modules and finally, the coordinator [Brooks, 1986]. However, behaviors must be completely independent of each other. The global structure is a network of interacting behaviors comprising low-level control and high-level deliberation abilities. The latter is performed by the distributed representation which can contain states and, consequently, change the behavior according to their information.

The parallel structure of simple behaviors allows a real-time response with low computational cost. Autonomous robots using this methodology can be built easily at low cost. Behavior-based robotics has demonstrated its reliable performance in standard robotic activities such as navigation, obstacle avoidance, terrain mapping, object manipulation, cooperation, learn-

ing maps and walking. For a more detailed review refer to [Arkin, 1998, Pfeifer and Scheier, 1999].

2.2.1 Principles

There are a few basic principles which have been used by all researchers in Behavior-based Robotics. These principles provide the keys to success of the methodology.

- **Parallelism**. Behaviors are executed concurrently. Each one can run on its own processor. Parallelism appears at all levels, from behavioral design to software and hardware implementation. This characteristic contributes to the speed of computation and consequently to the dynamics between the robot and the environment.

- **Modularity**. The system is organized into different modules (behaviors). The important fact is that each module must run independently. This important principle contributes to the robustness of the system. If, for example, one behavior fails due to the break down of a sensor, the others will continue running and the robot will always be controlled. Another important consequence of modularity is the possibility of building the system incrementally. In the design phase, the priority behaviors are first implemented and tested. Once they run correctly, more behaviors can be added to the system.

- **Situatedness/Embeddedness**. The concept of "Situatedness" means that a robot is situated in and surrounded by the real world. For this reason it must not operate using an abstract representation of reality, it must use the real perceived world. "Embeddednes" refers to the fact that the robot exists as a physical entity in the real world. This implies that the robot is subjected to physical forces, damages and, in general, to any influence from the environment. This means that the robot should not try to model these influences or plan with them. Instead it should use this system-environment interaction to act and react with the same dynamics as the world.

- **Emergence**. This is the most important principle of Behavior-based Robotics. It is based on the principles explained above and attempts to explain why the set of parallel and independent behaviors can arrive at a composite behavior for the robot to accomplish the expected goals. Emergence is the property which results from the interaction between the robotic behavioral system and the environment. Due to emergence,

the robot performs behaviors that were not pre-programmed. The interaction of behavior with the environment generates new characteristics in the robot's behavior which were not pre-designed. Numerous researchers have talked about emergence. Two examples are "Intelligence emerges from interaction of the components of the system" [Brooks, 1991b] and "Emergence is the appearance of novel properties in whole systems" [Moravec, 1988].

2.2.2   Behavior features

A behavioral response is a functional mapping from the stimulus plane to the motor plane. The motor plane usually has two parameters, the strength (magnitude of the response) and the orientation (direction of the action). According to Arkin [Arkin, 1998], a behavior can be expressed as (S, R, β) where:

- **S: Stimulus Domain**. S is the domain of all perceivable stimuli. Each behavior has its own stimulus domain.

- **R: Range of Responses**. For autonomous vehicles with six degrees of freedom, the response $r \in R$ of a behavior is a six-dimensional vector: $r = [x, y, z, \phi, \theta, \psi]$ composed of the three translation degrees of freedom and the three rotational degrees of freedom. Each parameter is composed of strength and orientation values. When there are different responses r_i, the final response is $r_i' = g_i \cdot r_i$, where g_i is a gain which specifies the strength of the behaviour relative to the others.

- **β: Behavioral Mapping**. The mapping function "β" relates the stimulus domain with the response range for each individual active behavior: $\beta(s) \rightarrow r$. Behavioral mappings β can be :

  - *Discrete*: numerable sets of responses.
  - *Continuous*: infinite space of potential reactions.

2.2.3   Coordination

When multiple behaviors are combined and coordinated, the emergent behavior appears. This is the product of the complexity between a robotic system and the real world. The two primary coordination mechanisms are:

- **Competitive methods**. The output is the selection of a single behavior, see Figure 2.5a. The coordinator chooses only one behavior to
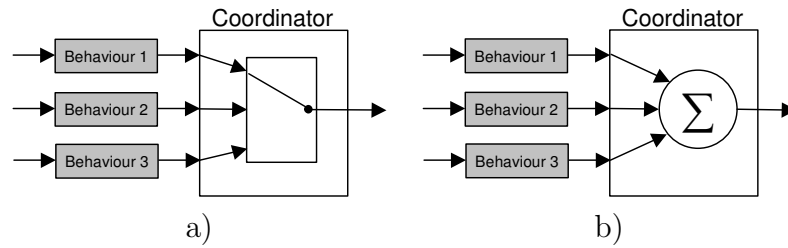
Figure 2.5: Coordination methodologies: a) competition b) cooperation.

control the robot. Depending on different criteria the coordinator determines which behavior is best for the control of the robot. Preferable methods are suppression networks such as Subsumption architecture [Brooks, 1986], action-selection [Maes, 1990] and voting-based coordination [Rosenblatt and Payton, 1989].

- **Cooperative methods**. The output is a combination function off all the active behaviors, see Figure 2.5b. The coordinator applies a method which takes all the behavioral responses and generates an output which will control the robot. Behaviors which generate a stronger output will impose a greater influence on the final behavior of the robot. Principal methods are vector summation such as potential fields [Khatib, 1985], and behavioral blending [Saffiotti et al., 1995].

Basic behavior-based structures use only a coordinator which operates using all the behaviors to generate the robot's response. However, there are more complex systems with different groups of behaviors coordinated by different coordinators. Each group generates an output and with these intermediate outputs, the final robot's response is generated through a final coordinator. These recursive structures are used in high level deliberation. By means of these structures a distributed representation can be made and the robot can behave differently depending on internal states, achieving multi-phase missions.

2.2.4 Adaptation

One of the fields associated with Behavior-based robotics is Adaptation. Intelligence cannot be understood without adaptation. If a robot requires autonomy and robustness it must adapt itself to the environment. The primary reasons for autonomous adaptivity are:

- The robot's programmer does not know all the parameters of the behavior-based system.

- The robot must be able to perform in different environments.

- The robot must be able to perform in changing environments.

And the properties which adaptive systems in robotics must contemplate are [Kaelbling, 1999]:

- Tolerance to sensor noise.

- Adjustability. A robot must learn continuously while performing its task in the environment.

- Suitability. Learning algorithms must be adequate for all kinds of environments.

- Strength. The adaptive system must posses the ability to influence the control of the robot in a desired place in order to obtain the desired data.

However, adaptation is a wide term. According to McFarland there are various levels of adaptation in a behavior-based system [McFarland, 1991]:

- **Sensory Adaptation**. Sensors become more attuned to the environment and changing conditions of light, temperature, etc.

- **Behavioral Adaptation**. Individual behaviors are adjusted relative to the others.

- **Evolutionary Adaptation**. This adaptation is done over a long period of time inducing change in individuals of one species, in this case robots. Descendants change their internal structure based on the success or failure of their ancestors in the environment.

- **Learning as Adaptation**. The robot learns different behaviors or different coordination methods which improve its performance.

Many parameters can be adapted in a behavior-based robotic system. At the moment there are only a few examples of real robotic systems which learn to behave and there is no established methodology to develop adaptive behavior-based systems. The two approaches most commonly used are Reinforcement learning and Evolutionary techniques [Ziemke, 1998, Arkin, 1998,

Dorigo and Colombetti, 1998, Pfeifer and Scheier, 1999]. Both have interesting characteristics but also disadvantages like convergence time or the difficulties in finding a reinforcement or fitness function respectively. In many cases they are implemented over control architectures based on neural networks. Using the adaptive methodologies, the weights of the network are modified until an optimal response is obtained. The two approaches have demonstrated the feasibility of the theories in real robots in all levels of adaptation. The basic ideas of the two methodologies are described next.

- **Reinforcement learning**
  Reinforcement learning (RL) is a class of learning algorithm where a scalar evaluation (reward) of the performance of the algorithm is available from the interaction with the environment. The goal of a RL algorithm is to maximize the expected reward by adjusting some value functions. This adjustment determines the control policy that is being applied. The evaluation is generated by a reinforcement function which is located in the environment. Chapter 4 gives a detailed description of Reinforcement Learning and its application to robotics. Main references about RL are [Kaelbling et al., 1996, Sutton and Barto, 1998].

- **Evolutionary Robotics**
  Evolutionary learning techniques are inspired by the mechanisms of natural selection. The principal method used is Genetic Algorithms (GAs). Evolutionary algorithms typically start from a randomly initialized population of individuals/genotypes encoded as strings of bits or real numbers. Each individual encodes the control system of a robot and is evaluated in the environment. From the evaluation, a score (fitness) is assigned which measures the ability to perform a desired task. Individuals obtaining higher fitness values are allowed to reproduce by generating copies of their genotypes with the addition of changes introduced by various genetic operators (mutation, crossover, duplication, etc.). By repeating this process over several generations, the fitness values of the population increase. Evolutionary robotics has shown good results in real robots. Usually they are used over neural networks modifying the weights of the nodes. Evolutionary algorithms have demonstrated more reliable solutions than reinforcement learning when the reinforcement frequency is low. However, evolutionary approaches also have problems due to the longer time necessary to converge into a reasonable solution. For a complete introduction to Evolutionary Robotics refer to [Nolfi, 1998, Harvey et al., 1997].
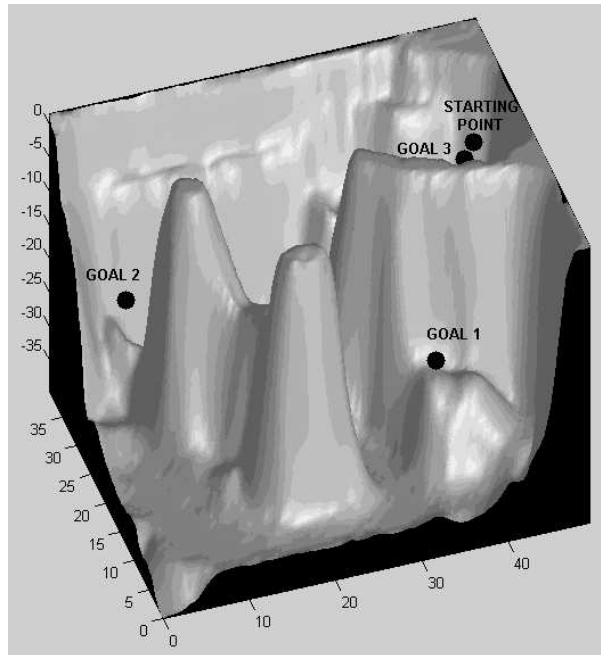
Figure 2.6: Simulated underwater environment in which the AUV must reach three goal points. The dimensions, in meters, of the environment can also be seen.

2.3   Experimental task: control of an AUV

In order to test some behavior-based control architectures, an experimental task was designed. The task consisted of achieving three "goal-points", one after the other, avoiding obstacles and avoiding becoming trapped. The starting point of the task and the three goal points can be seen in Figure 2.6.

A behavior-based control architecture with three behaviors was designed to fulfill the proposed task. All the architectures were implemented with these behaviors changing implementation aspects but maintaining their structure. These behaviors use different inputs but generate an established output. The output was a three-dimensional vector representing the speed the vehicle should follow. The vector $v_i = (v_{i,x}, v_{i,z}, v_{i,yaw})$ is composed of two linear velocities ($v_{i,x}$ with respect to the X axis and $v_{i,z}$ with respect to the Z axis) and an angular velocity ($v_{i,yaw}$ respect Z axis). The three behaviors are next described:

"Go to" behavior. The purpose of this behavior is to drive the vehicle towards the goal point. It proposes a speed vector with a constant
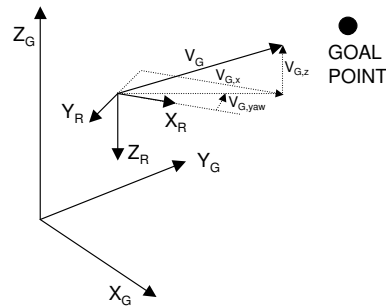
Figure 2.7: Response, V_G, of the "Go To" behavior.
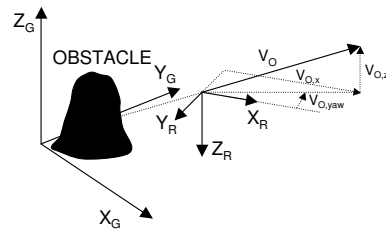
Figure 2.8: Response, V_O, of the "Obstacle Avoidance" behavior.

module. The direction of the vector is the one which joins the position of the vehicle with the goal point, see Figure 2.7. The input of the vector is the position and orientation of the vehicle. The goal points are stored by the behavior and changed when the robot gets close.

Obstacle avoidance behavior. This behavior is used to keep the robot from crashing into obstacles. The inputs of the behavior are various sonar measures. The vehicle is simulated with seven sonar transducers: three at the front, one on each side, one at the back and another on the bottom. Each sonar direction has a maximum range. If one of the distances is less than this range, the behavior will generate an opposite response. The behavior computes all distances and creates a 3D speed vector which indicates the direction the robot must take to avoid obstacles detected by the transducers, see Figure 2.8.

"Avoid trapping" behavior. The avoid trapping [Balch and Arkin, 1993] behavior is used to depart from situations in which the robot becomes trapped. The input of the behavior is the position of the robot which is used to save a *local map* of the recent path of the vehicle. The map is centered on the robot's position and has a finite number of cells.
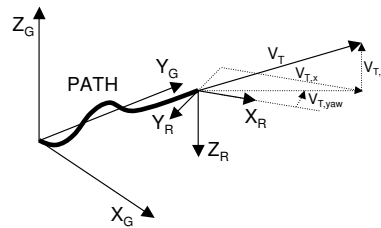
Figure 2.9: Response, V_T, of the "Avoid Trapping" behavior.

Each cell contains the number of times the robot has visited the cell. If the sum of all the values is higher than a threshold, the behavior becomes active and a speed vector is generated. The direction of the vector will be the one opposed to the gravity center of the local map. The module will be proportional to the sum of the cell values. The cells are incremented by a configurable sample time and saturated on a maximum value. However, if cells are not visited, the values are decreased allowing going back to a visited zone.

This behavior becomes active in two specific situations. The first is when the vehicle is trapped in front of obstacles. In this case the behavior will take control of the vehicle and drive it away to another zone. The second situation is when the vehicle is navigating very slowly due to the interaction of the other behaviors. In this case the cells will increase in value rapidly and the behavior will become active driving the vehicle away from the path, see Figure 2.9.

These behaviors were implemented with each tested architecture constituting the control architecture. In addition to the control architecture, other modules were required such as low-level controllers, a accurate dynamics model of the AUV and a graphical interface with an underwater environment, see Figure 2.10. All these components were implemented through a simulated environment. It is specially relevant the identification of the dynamics model of the underwater robot, which was previously performed with real experiments. The simulation of the experimental task was considered to be enough to evaluate the different aspects of the control architectures. It is important to note that typical problems of real robots (position estimation, noise in signals, faults in sonar distances, etc.) were not simulated. The non-consideration of these aspects breaks several principles of Behavior-based Robotics. However, these simulations intend only to test the performance of the control architectures not the principles. We assume that if the architectures were implemented in a real robot, the features of Behavior-based
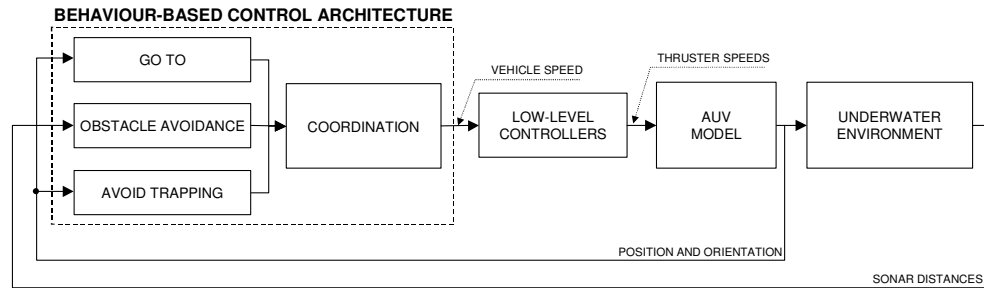
Figure 2.10: Overall control system used in the simulated task.

Control Architecture	Behavioral choice and design	Behavioral encoding	Coordination method	Programming method
Subsumption architecture	Experimentally	Discrete	Competitive, arbitration via inhibition and suppression	AFSM, Behavioral Language or behavioral libraries
Action Selection Dynamics	Experimentally	Discrete	Competitive, arbitration via levels of activation	Mathematical algorithms
Schema-based approach	Ethologically	Continuous, Potential fields	Cooperative via vector summation and normalization	Parameterized behavioral libraries
Process Description Language	Experimentally	Continuous	Cooperative via values integration and normalization	Process Description Language

Table 2.1: Four relevant Behavior-based approaches.

Robotics would assure robustness when faced with these problems.

2.4   Some Behavior-based approaches

Many proposals have appeared since the field of Behavior-based Robotics began in 1986 with Subsumption architecture. From among them, four architectures have been chosen which represent the main principles of Behavior-based Robotics. These architectures were pioneers in the field and their feasibility was demonstrated with real robots by their designers. The architectures and their basic characteristics can be seen in Table 2.1.

In the next subsections, each architecture is first described as it was originally designed and then tested with the experimental task described in the previous section. For each architecture, a discussion of the main advantages and disadvantages is given according to some qualitative properties. Refer to Table 2.2 for the list and description of these properties.

Property	Description
Performance	Quality of the trajectory generated by the vehicle. A control architecture with a good performance will have an optimized and smooth trajectory without big jumps on the vehicle's heading.
Modularity	Property of being able to add new behaviors without having to modify (gains or parameters) the current ones.
Robustness	Property of being able to preserve the integrity of the vehicle and the fulfilment of the task when some small changes in the parameters or in the environment occur.
Development time	Time needed to implement the control architecture, from the design until the programming phase.
Tuning time	Time needed to find the parameters which maximize the performance.
Simplicity	Simplicity of the methodology, including the design and programming phases.

Table 2.2: Qualitative properties used in the evaluation of the control architectures.

2.4.1   Subsumption architecture

Description

The Subsumption architecture was designed by Rodney Brooks in the 1980s at the Massachusetts Institute of Technology. His work opened the field of Behavior-based Robotics. To overcome the problems encountered in Traditional AI when designing real robotic systems, Brooks proposed a completely different methodology. He questioned the centralized symbolic world model and proposed a decentralized set of simple modules which reacted more rapidly to environmental changes. To accomplish this, he presented the Subsumption architecture in 1986 with the paper "A Robust Layered Control System for a Mobile Robot" [Brooks, 1986]. Later on, he modified a few aspects of the architecture as a result of suggestions from J.H. Connell. The final modifications on the Subsumption architecture can be found in [Brooks, 1989, Connell, 1990]. Subsumption Architecture has been widely applied to all kinds of robots since then. Some further modifications have also been proposed. However, in this report, the original Subsumption approach will be described.

Subsumption Architecture is a method of reducing a robot's control architecture into a set of task-achievement behaviors or competences represented as separate layers. Individual layers work on individual goals concurrently and asynchronically. All the layers have direct access to the sensory information. Layers are organized hierarchically allowing higher layers to inhibit or suppress signals from lower layers. Suppression eliminates the control signal from the lower layer and substitutes it with the one proceeding from the
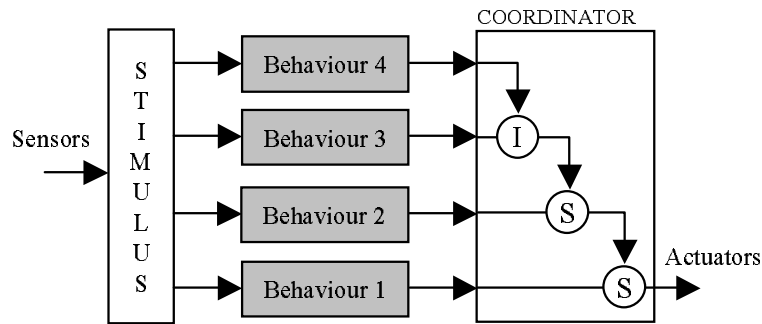
Figure 2.11: Subsumption control architecture. Coordination through suppression and inhibition nodes.

higher layer. When the output of the higher layer is not active, the suppression node does not affect the lower layer signal. On the other hand, only inhibition eliminates the signal from the lower layer without substitution. Through these mechanisms, higher-level layers can subsume lower-levels. The hierarchy of layers with the suppression and inhibition nodes constitute the competitive coordination method, see Figure 2.11.

All layers are constantly observing the sensory information. When the output of a layer becomes active, it suppresses or inhibits the outputs of the layers below, taking control of the vehicle. The layer has internal states and timers which allow it to generate an action which depends on the current state and input. Also, the timers allow the robot to maintain the activity for a period of time after the activation conditions finish.

This architecture can be built incrementally, adding layers at different phases. For example, the basic layers, such as "go to behavior" or "avoid obstacles behavior", can be implemented and tested in the first phase. Once they work properly, new layers can be added without the necessity of redesigning previous ones.

The layers of the subsumption architecture were originally designed as a set of different modules called Augmented Finite States Machines (AFSM). Each AFSM is composed of a Finite State Machine (FSM) connected to a set of registers and a set of timers or alarm clocks, see Figure 2.12. Registers store information from inside FSM as well as from the outside sensors and other AFSM. Timers enable state changes after a certain period of time while finite state machines change their internal state depending on the current state and inputs. An input message or the expiration of a timer can change the state of the machine. Inputs from the AFSM can be suppressed by other machines and outputs can be inhibited. AFSM behave like a single FSM but
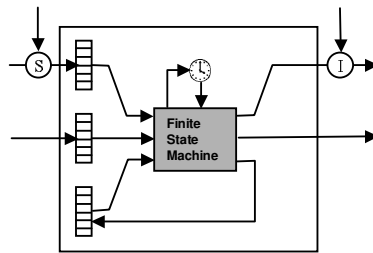
Figure 2.12: Augmented Finite State Machine (AFSM).

with the added characteristics of registers and timers.

Using a set of augmented finite state machines a layer can be implemented to act like a behavior. A layer is constructed as a network of AFSM joined by wires with suppression and inhibition nodes. Figure 2.13 shows the AFSM network designed by Brooks for a mobile robot with the layers "avoid objects", "wander" and "explore", [Brooks, 1986]. As the figure shows, designing the network so as to accomplish a desired behavior is not exactly clear. For this reason, Brooks developed a programming language, the Behavioral Language [Brooks, 1990], which generates the AFSM network using a single rule set for each behavior. The high-level language is compiled to the intermediate AFSM representation and then further compiled to run on a range of processors.

One of the principles of the Subsumption architecture is independence from the layers. The implementation methodology, as stated above, consists of building the layers incrementally once the previous layers have been tested. Nevertheless, in Figure 2.13 there are some wires which go from one layer to another breaking the independence. This fact was shown by Connell [Connell, 1990] who proposed a total independence of the layers until the coordination phase. This assures the possibility of implementing the layers incrementally without redesigning the previous ones. This is also useful in order to map each layer into a different processor in the robot. Connell and other researchers have also proposed other formalism instead of AFSM to implement the layers. Usually, computer programs are used for simplicity in programming rules without the use of FSM. AFSM must be considered as the formalism Brooks chose, for its simplicity and rapid processing, to implement the Subsumption architecture, not as part of it.
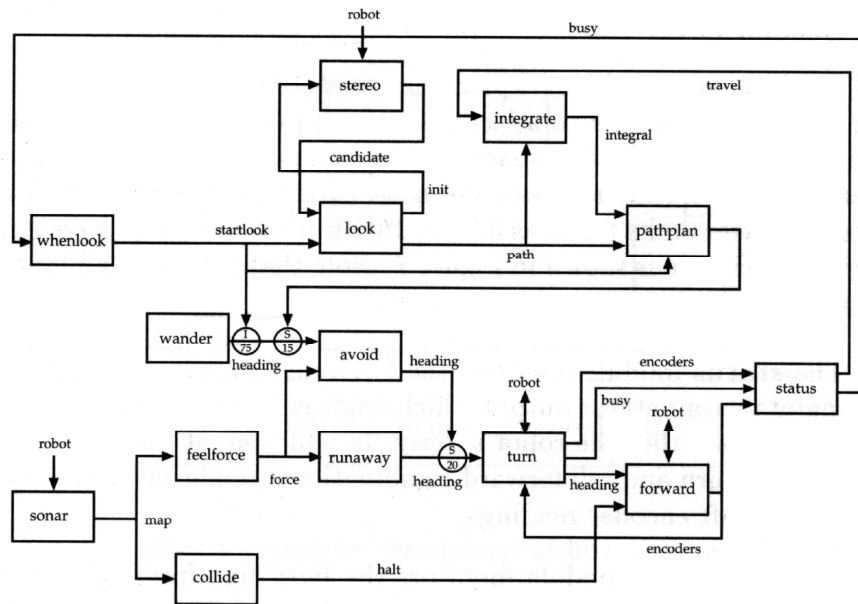
Figure 2.13: AFSM network designed by Brooks for a mobile robot.
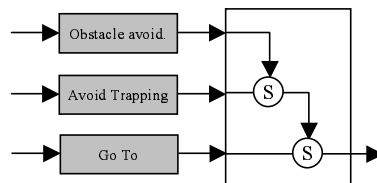
Figure 2.14: Task implementation with Subsumption architecture.

Implementation

Subsumption Architecture was implemented to accomplish the experimental task described in Section 2.3. The three behaviors were implemented in three different functions which use sensory information as inputs and, as outputs, the 3-dimensional velocity the vehicle should follow. Also, two suppression nodes were used to coordinate behaviors. AFSM was not used due to the simplicity of the functions. However, as mentioned before, the implementation method is not the most important fact in the subsumption architecture. The three behaviors were implemented as shown in Figure 2.14.

The hierarchy of behaviors was constructed as follows: at the top, the "Obstacle avoidance" behavior, followed by the "Avoid trapping" and "Go
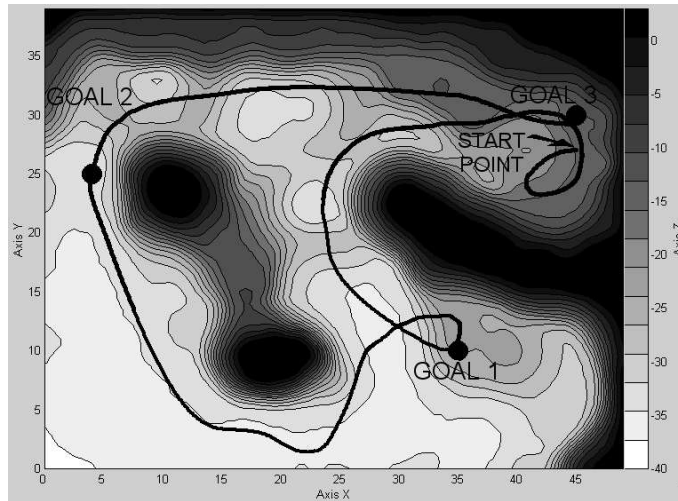
Figure 2.15: Top view of the trajectory carried out by the AUV with Subsumption architecture.

to" behaviors. This hierarchy primarily assures that the vehicle stays away from obstacles. As the task never requires proximity to obstacles, the "Obstacle avoidance" in the top level assures the safety of the vehicle. At the second level, the "Avoid trapping" behavior takes control over the "Go to" behavior if the vehicle becomes trapped. As in AFSM, a timer was used to maintain activity in the behaviors for a short time after the activation conditions were finished.

Some graphical results of the Subsumption approach can be seen in the next three figures. Figure 2.15 shows a top view of the simulation and Figure 2.16 shows a vertical view. Finally Figure 2.17 shows a three-dimensional representation of the simulation.

Given the results, it can be said that the principal advantages of the Subsumption approach are robustness, modularity and easy tuning of the behaviors. Behaviors can be tuned individually and, once they work properly, can be mixed. The design of the hierarchy is very easy once the priorities of the behaviors are known (a difficult task when working with a large architecture). This architecture is very modular, every behavior can be implemented with a different processor with the responses coordinated as a final step. A sequential algorithm is not necessary as all behaviors are completely independent. The principal disadvantage is the non-optimal trajectories, due to the competitive coordination method, with a lot of jumps in the vehicle's heading when the active behavior changes. Table 2.3 summarizes the prin-
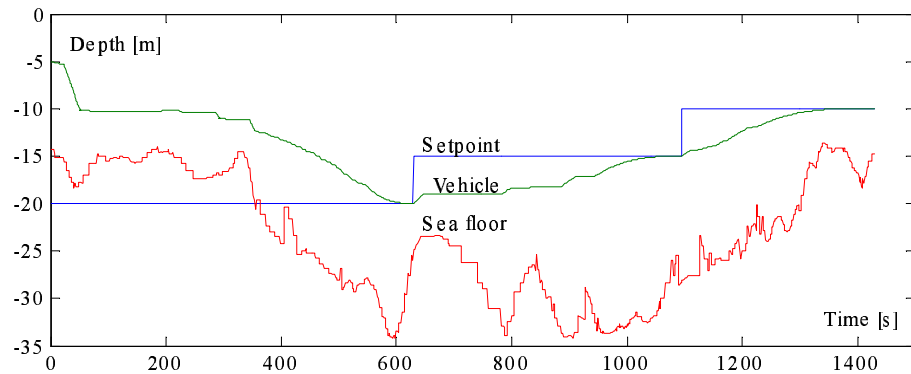
Figure 2.16: Vertical view of the trajectory carried out by the AUV with Subsumption architecture. The set-point graph is the depth of the goal to be achieved. The vehicle graph is the depth of the vehicle during the whole simulation. And the sea floor graph is the depth of the sea floor directly under the vehicle.
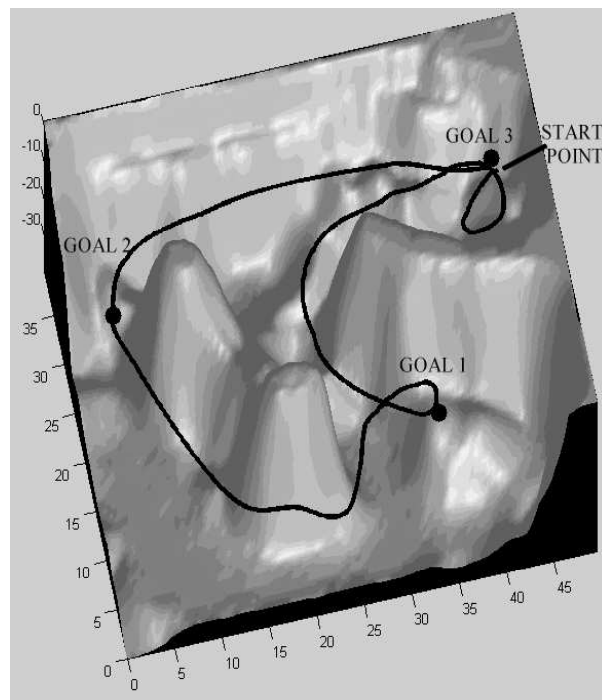
Figure 2.17: Three-dimensional view of the trajectory carried out by the AUV with Subsumption architecture.

SUBSUMPTION ARCHITECTURE	
Developer	Rodney Brooks, Massachusetts Institute of Technology
References	[Brooks, 1986, Brooks, 1989, Connell, 1990]
Behavioral encoding	Discrete
Coordination method	Competitive, arbitration via inhibition and suppression
Programming method	AFSM, Behavioral Language or behavioral libraries
Advantages	Modularity, Robustness and Tuning time
Disadvantages	Development time and performance

Table 2.3: Subsumption architecture features.

cipal characteristics of Subsumption Architecture. For a description of the advantage/disadvantage terms refer to Table 2.2.

2.4.2   Action Selection Dynamics

Description

Action Selection Dynamics (ASD) is an architectural approach which uses a dynamic mechanism for behavior (or action) selection. Pattie Maes from the AI-Lab at MIT developed it toward the end of the 1980s. Principal references are [Maes, 1989, Maes, 1990, Maes, 1991]. Behaviors have associated activation levels which are used to arbitrate competitively the activity which will take control of the robot. Other approaches for action selection have been proposed [Tyrell, 1993], however, ASD is the most well known and most commonly specified.

Action Selection Dynamics uses a network of nodes to implement the control architecture. Each node represents a behavior. The nodes are called *competence modules.* A network of modules is used to determine which competence module will be active and, therefore, control the robot. The coordination method is competitive, only one module can be active at any moment. To activate the competence modules some binary states are used. Each competence module has three lists of states which define its interaction within the network. The first list is the *precondition list* and contains all the states which should be true so that the module becomes executable. The second list is the *add list* and contains all the states which are expected to be true after the activation of the module. Finally, the third list is the *delete list* and contains the states which are expected to become false after the execution of the module.

The states are external perceptions of the environment gathered by the sensors. Usually some kind of processing will be necessary to transform the analogue outputs of the sensors to a binary state. For example, for the state "No_Obstacle", all the values provided by the sonar have to be processed

to determine if there are nearby obstacles. The states can also be internal assumptions or motivations of the robot. The state "Way-point-Reached" could be one example. The states are also used to determine the *goals* and *protected goals* of the robot. The goals would be the states which are desired to be true. The protected goals are the goals already achieved and therefore retained. The task of the robot is defined by the assignment of the goals to some states.

Once all the states and competence modules are defined, the decision network can be built. Different links appear between the nodes based on the precondition, add and delete lists: ·

- **Successor link**: For each state which appears in the add list of module A and in the precondition list of module B, a successor link joins A with B.

- **Predecessor link**: A predecessor link joints B with A if there is a successor link between A and B.

- **Conflicter link**: For each state which appears in the delete list of module B and in the precondition list of module A, a conflicter link joins A with B.

In Figure 2.18 successor links can be seen as solid line arrows and conflicter links as solid lines with a big dot instead of an arrow. Note that predecessor links are inverted successor links.

Activation of competence modules occurs depending on the quantity of energy they are given. The energy is spread in two phases. In the first phase three different mechanisms are used:

1. *Activation by the states*: if at least one state in the precondition list is true, activation energy is transferred to the competence module.

2. *Activation by the goals*: if at least one state in the add list belongs to a goal state, activation energy is transferred to the competence module.

3. *Activation by the protected goals*: if at least one state in the delete list belongs to a protected goal state, activation energy is removed from the competence module.

The spread of energy in phase one is shown in Figure 2.18 with dotted lines. On the other hand, phase two spreads energy from competence modules. Three mechanisms are also used:
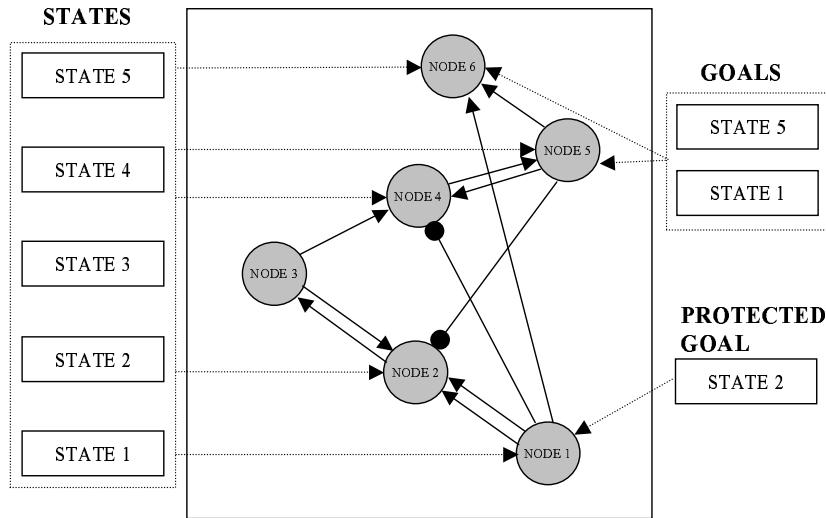
Figure 2.18: Example of an Action Selection Dynamics network.

1. *Activation of Successors*: Executable modules spread a fraction of their own energy to successors which aren't executable if the state of the link is false. The goal is to increase activation of modules which become executable after the execution of the predecessor module.

2. *Activation of Predecessor*: Non-executable modules spread a fraction of their own energy to the predecessor if the state of the link is false. The goal is to spread energy to the modules so that, through their execution, the successor module becomes executable.

3. *Inhibition of Conflicters*: Competence modules decrease the energy of conflicter modules if the state of the link is true. The goal is to decrease the energy of the conflicters which, by becoming active, make the preconditions of the module false.

In each cycle the competence modules increase or decrease their energy until a global maximum and minimum level are reached. The activated module has to fulfil three conditions:

1. It has to be executable (all preconditions have to be true).

2. Its level of energy has to surpass a threshold.

3. Its level of energy has to be higher than that of the modules accomplishing conditions 1 and 2.

State	Description
WAY-POINT	There is a way-point to go
NO_WAY-POINT	There is not any way-point
OBSTACLE	There is a nearby obstacle
NO_OBSTACLE	There is not any nearby obstacle
TRAPPED	The vehicle cannot depart from the same zone
NO_TRAPPED	The vehicle is moving through different zones

Table 2.4: ASD States to fulfill the task.

When a module becomes active, its level of energy is re-initialized to 0. If none of the modules fulfil condition 2, the threshold is decreased. Several parameters are used for the thresholds and the amount of energy to be spread. Also, normalization rules assure that all modules have the same opportunities to become active. Note that the energy of the modules is accumulated incrementally, then the sample time becomes very important because it determines the velocity of the accumulation. For a mathematical notation of this algorithm refer to [Maes, 1989].

The intuitive idea of Action Selection Dynamics is that by using the network and the spreading of energy, after some time, the active module is the best action to take for the current situation and current goals. Although Action Selection Dynamics is complex and difficult to design, it has been successfully tested in real robotic systems.

Implementation

Action Selection Dynamics Architecture was used to accomplish the experimental task described in Section 2.3. In the implementation of the ASD network, the three behaviors represent three competence modules. Each behavior was implemented in a function. The coordination module which contains the ASD network was also implemented in another function following the mathematical notations found in [Maes, 1989]. Each module used different binary states in its lists. In Table 2.4 all the states are shown. Note that from the six states, three are the negation of the other three. This is to simplify the ASD algorithm. The goal of the robot is the state "No_Way_Point". When this state is true the robot has passed through all the way-points and therefore the task is complete. After the state description, the preconditions list, add list and delete list were defined, see Table 2.5.

Following the lists of each competence module the network can be implemented, see Figure 2.19. Note that the design phase for the ASD architecture consists of specifying the states and the lists. After this, the entire network

Go To	STATES:		
PRECONDITION LIST	WAY-POINT	NO_OBSTACLE	NO_TRAPPED
ADD LIST	NO_WAY_POINT	OBSTACLE	TRAPPED
DELETE LIST	WAY-POINT		
Obstacle Avoidance	**STATES:**		
PRECONDITION LIST	OBSTACLE		
ADD LIST	NO_OBSTACLE		
DELETE LIST	OBSTACLE		
Avoid Trapping	**STATES:**		
PRECONDITION LIST	WAY-POINT	NO_OBSTACLE	TRAPPED
ADD LIST	NO_TRAPPED	OBSTACLE	
DELETE LIST	TRAPPED		

Table 2.5: Precondition, add and delete lists for the 3 competences.
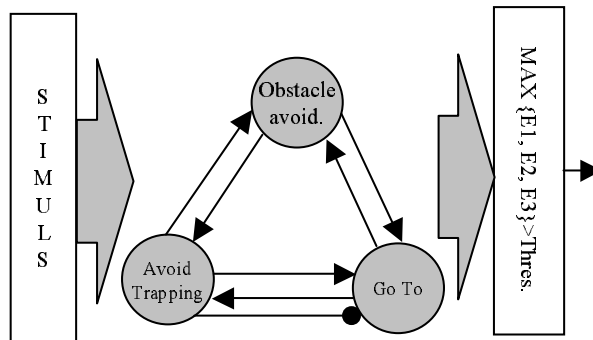
Figure 2.19: Task implementation with Action Selection Dynamics.

can be generated automatically and only some parameters must be tuned.

The ASD network spreads energy from the states, the goal and between the competence modules. The competence module, which is executable and has more energy than the others and the threshold, will activate and control the robot until the next iteration. The parameters of the decision network that have been used can be seen in Table 2.6. The spreading of energy between the modules is determined by the relationship between the different parameters.

Some graphical results of the ASD approach can be seen in the next three figures. Figure 2.20 shows a top view of the simulation and Figure 2.21 shows a vertical view. Finally, Figure 2.22 shows a three-dimensional representation of the simulation. As can be seen, the trajectory obtained with Action Selection Dynamics is quite optimal. The principal advantages of this method are robustness of the architecture and automatic coordination, once the network

Parameter	Description	Value
π	Maximum level of energy per module	40 units
ϕ	Amount of energy spread by a state	10 units
γ	Amount of energy spread by a goal	20 units
δ	Amount of energy spread by a protected goal	0 units
θ	Threshold to becoming active	15 units
T_s	Sample time	1 second

Table 2.6: Parameters of the ASD implementation found experimentally.
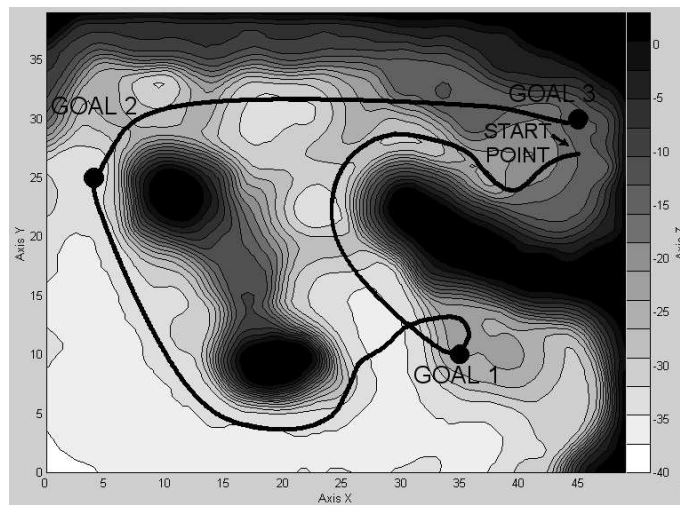
Figure 2.20: Top view of the trajectory carried out by the AUV with Action Selection Dynamics.

has been generated. However, the design and implementation phases are very complex and difficult. Table 2.7 summarizes the principal characteristics of Action Selection Dynamics. For a description of the advantage/disadvantage terms refer to Table 2.2.

2.4.3 Motor Schemas approach

Description

Schema-based theories appeared in the eighteenth century as a philosophical model for the explanation of behavior. Schemas were defined as the mechanism of understanding sensory perception in the process of storing knowledge. Later on, at the beginning of the twentieth century, the schema
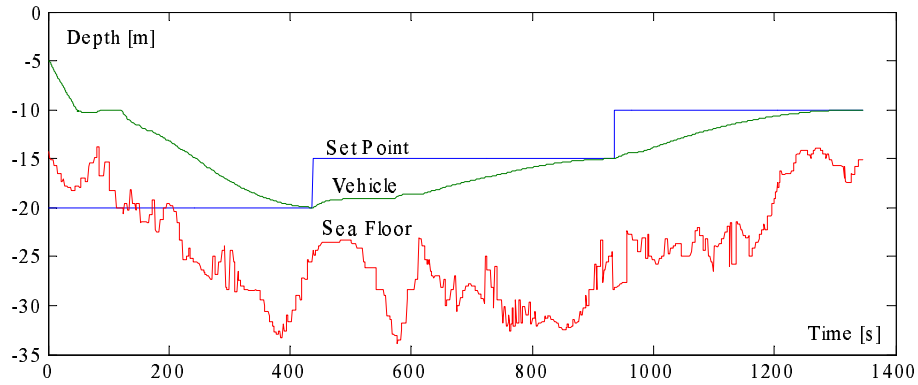
Figure 2.21: Vertical view of the trajectory carried out by the AUV with Action Selection Dynamics. The depths of the goal-point, the vehicle and the sea floor are shown.
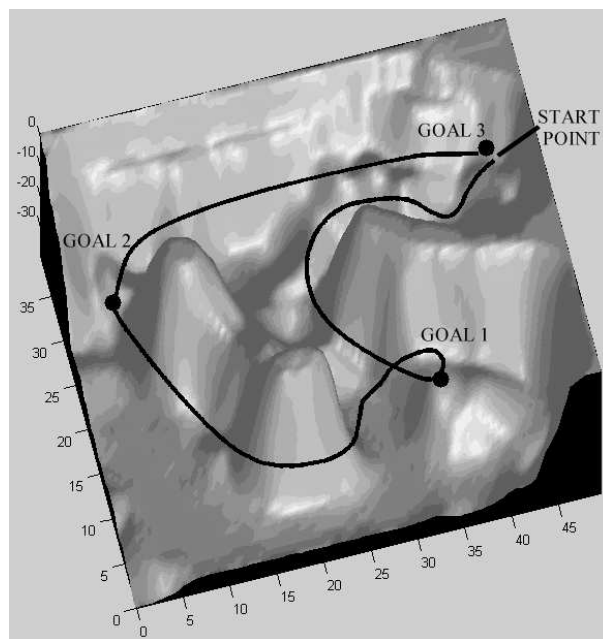
Figure 2.22: Three-dimensional view of the trajectory carried out by the AUV with Action Selection Dynamics.

ACTION SELECTION DYNAMICS	
Developer	Pattie Maes, Massachusetts Institute of Technology
References	[Maes, 1989, Maes, 1990, Maes, 1991]
Behavioral choice and design	Experimentally
Coordination method	Competitive, arbitration via levels of activation
Programming method	Mathematical algorithms
Advantages	Modularity and Robustness
Disadvantages	Development time and No Simplicity

Table 2.7: Action Selection Dynamics features.

theory was adapted in psychology and neuroscience as a mechanism for expressing models of memory and learning. Finally in 1981, Michael Arbib adapted the schema theory for a robotic system [Arbib, 1981]. He built a simple schema-based model inspired by the behavior of the frog to control robots. Since then, schema-based methodologies have been widely used in robotics. The principal proposal is Motor Schemas developed by Ronald Arkin at Georgia Institute of Technology, Atlanta. Arkin proposed Motor Schemas [Arkin, 1987] as a new methodology of Behavior-based Robotics.

From a robotic point of view "a motor schema is the basic unit of behavior from which complex actions can be constructed; it consists of the knowledge of how to act or perceive as well as the computational process by which it is enacted" [Arkin, 1993]. Each schema operates as a concurrent, asynchronous process initiating a behavioral intention. Motor schemas react proportionally to sensory information perceived from the environment. All schemas are always active producing outputs to accomplish their behavior. The output of a motor schema is an action vector which defines the way the robot should move. The vector is produced using the potential fields method [Khatib, 1985]. However, instead of producing an entire field, only the robot's instantaneous reaction to the environment is produced, allowing a simple and rapid computation.

The coordination method is cooperative and consists of vector summation of all motor schema output vectors and normalization. A single vector is obtained determining the instantaneous desired velocity for the robot. Each behavior contributes to the emergent global behavior of the system. The relative contribution of each schema is determined by a gain factor. Safety or dominant behaviors must have higher gain values. Normalization assures that the final vector is within the limits of the particular robot's velocities. Figure 2.23 shows the structure of motor schema architecture.

Implementation of each behavior can be done with parameterized behavioral libraries in which behaviors like "move ahead", "move-to-goal", "avoid-static-obstacle", "escape" or "avoid-past" can be found. Schemas have inter-
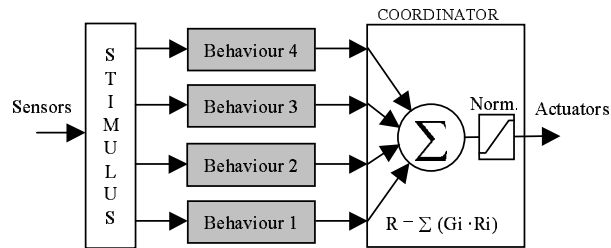
Figure 2.23: Example of a control architecture with Motor Schema approach.
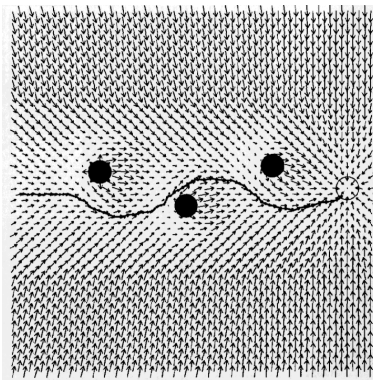
Figure 2.24: Potential field generated with Motor Schema approach.

nal parameters depending on the behavior and an external parameter, the gain value. Each schema can be executed into a different processor. Nevertheless, the outputs must have the same format in order to be summed by the coordinator. For two-dimensional vehicle control refer to [Arkin, 1989] and for three-dimensional control to [Arkin, 1992]. For a set of positions, each behavior generates a potential field which indicates the directions to be followed by the robot in order to accomplish the behavior. The merging of the behaviors in different robot positions, provides a global potential field, see Figure 2.24, which gives an intuitive view of the motor schema architecture performance.

Implementation

Motor Schema Architecture was applied to accomplish the experimental task described in Section 2.3. Each behavior was implemented in a different function. A simple coordination module was used to sum the signals and normalize the output. The structure of the control architecture can be seen in
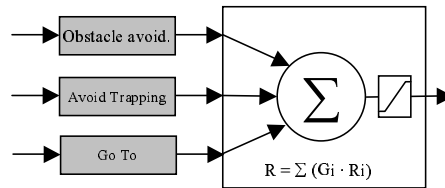
Figure 2.25: Task implementation with Motor Schema approach.
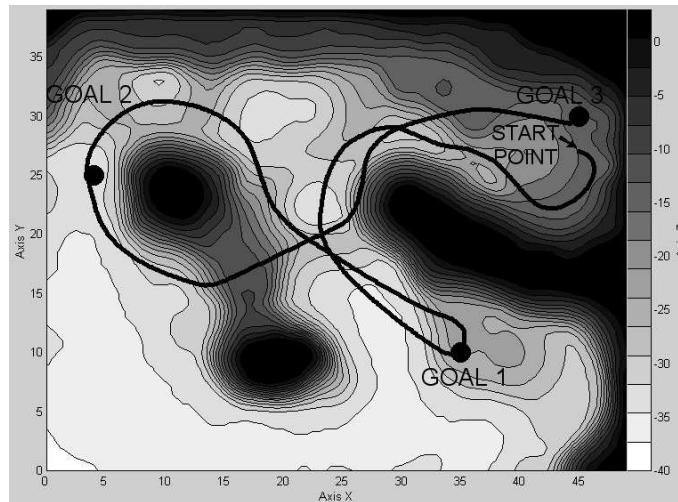
Figure 2.26: Top view of the trajectory carried out by the AUV with Motor Schema approach.

Figure 2.25. After tuning the system, "Obstacle avoidance" behavior had the highest gain value, followed by "Avoid trapping" and "Go to" behaviors. As in Subsumption architecture, higher priority was given to the safety behavior "Obstacle avoidance", followed by "Avoid trapping" to take control over "Go to" when necessary.

Some graphical results of the Motor Schema approach can be seen in the next three figures. Figure 2.26 shows a top view of the simulation and Figure 2.27 shows a vertical view. Finally, Figure 2.28 shows a three-dimensional representation of the simulation.

After reviewing the results, it can be said that the principal advantages are simplicity and easy implementation, as well as optimized trajectories. The architecture can be implemented in different processors because the algorithm is fully parallel. However, difficulties appeared in tuning the gain values. The values are very sensitive and have to be tuned together. When
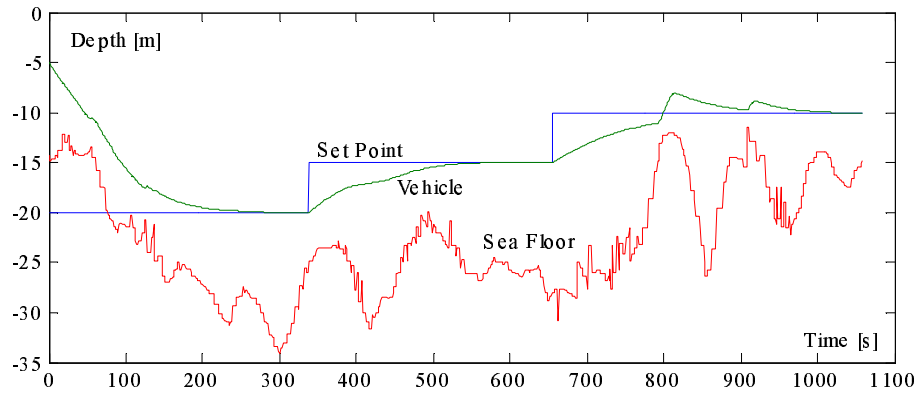
Figure 2.27: Vertical view of the trajectory carried out by the AUV with Motor Schema approach. The depths of the goal-point, the vehicle and the sea floor are shown.
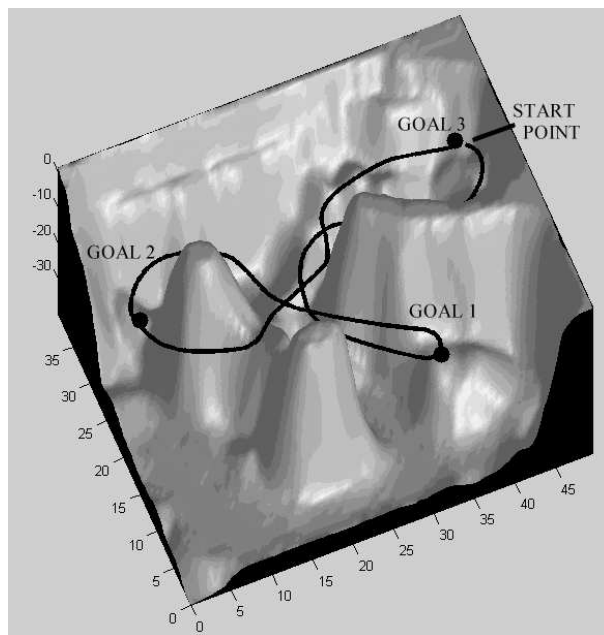
Figure 2.28: Three-dimensional view of the trajectory carried out by the AUV with Motor Schema approach.

MOTOR SCHEMA APPROACH	
Developer	Ronald Arkin, Georgia Institute of Technology
References	[Arkin, 1987, Arkin, 1989, Arkin, 1992]
Behavioral encoding	Continuous using potential fields
Coordination method	Cooperative via vector summation and normalization
Programming method	Parameterized behavioral libraries
Advantages	Development time and simplicity
Disadvantages	Tuning time, robustness and modularity

Table 2.8: Motor Schema approach features.

new behaviors are added, re-tuning is necessary because the sum of the responses of some behaviors can cancel the effect of others, such as "Obstacle avoidance". For this reason, robustness and modularity are very low. Table 2.8 summarizes the principal characteristics of Motor Schema approach. For a description of the advantage/disadvantage terms refer to Table 2.2.

2.4.4   Process Description Language

Description

Process Description Language (PDL) was introduced in 1992 by Luc Steels from the VUB Artificial Intelligence Laboratory, Belgium. PDL [Steels, 1992, Steels, 1993] is intended as a tool to implement process networks in real robots. PDL is a language which allows the description and interaction of different process constituting a cooperative dynamic architecture.

PDL architecture is organized with different active behavior systems, see Figure 2.29. Each one is intended as an external behavior of the robot like "explore", "go towards target" or "obstacle avoidance". Each behavior also contains many active processes operating in parallel. Processes represent simple movements which the behavior will use to reach its goal. Processes take information from sensors and generate a control action if needed. The control action is related to the set-points which must reach several actuators of the robot. A process output is an increment value which will be added to or subtracted from some set-points of the actuators. This means, for example, that the process "turn right if the left bumpers are touched" will add a value to the left motor set-point speed and subtract it from the right, if the necessary conditions are true (in this case, touching the left bumpers). The contribution of all the processes will be added to the current set-points and then a normalization will assure a bounded output. This simple methodology constitutes the cooperative coordination method.

Process description language proposes the language used to implement such processes. The functions are very simple allowing high speed processing.
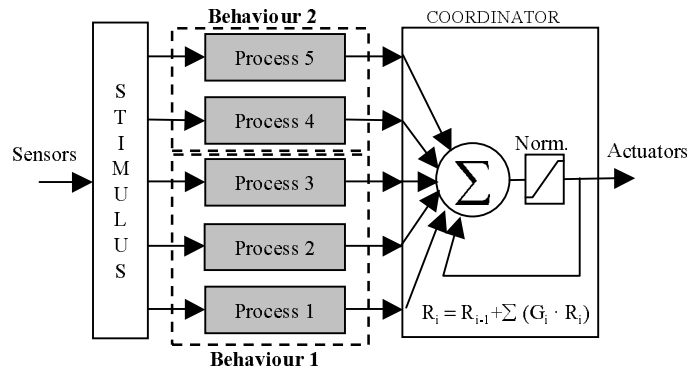
Figure 2.29: Example of a control architecture with Process Description Language.

For example, the process "turn right if the left bumpers are touched" would be implemented as:

```
void turn_right(void)
{
      if(bumper_mapping[3]>0{
            add_value(left_speed, 1);
            add_value(right_speed, -1);}
}
```

The relative contribution of each process is determined by the value added to or subtracted from the set-points. Processes with large values will exert a greater influence on the robot. The ultimate direction taken by the robot will depend on which process influences the overall behavior in the strongest way. It must be noted that these values are added each time step. For this reason it is very important that the ranges of these values be related to the sample time in which the architecture is working. It is possible that with small values and a big sample time, the architecture might not be able to control the robot. The dynamics of the architecture must be faster than the dynamics of the robot. This is due to the fact that PDL works by manipulating derivatives of the set-points implying a fast control loop to assure the system's stability. It's important to remember that although PDL is structured in simple and fast processes, the dynamics will always have to be faster than that of the robot or vehicle to be controlled.
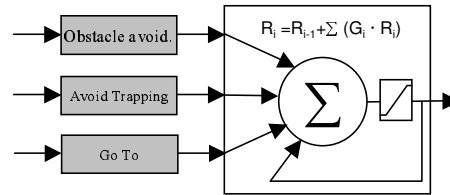
Figure 2.30: Task implementation with Process Description Language.

The overall execution algorithm is defined by the following recursive procedure:

1. All quantities are frozen (sensory information and set-points).

2. All processes are executed and their relative contribution are added or subtracted.

3. The set-points are changed based on the overall contribution of the processes.

4. The set-points are sent to the actuator controllers.

5. The latest sensory quantities are read in.

Implementation

Process Description Language Architecture was used to accomplish the experimental task described in Section 2.3. Each behavior was implemented in a different function. The low level processes of each behavior were assembled and the behavior only generated a response. The response is a vector which changed the current velocity of the vehicle in the direction desired by the behavior. The coordinator is a simple module which sums the current velocity with those generated by the behaviors. The final vector was normalized. Figure 2.30 shows the structure of this architecture.

The module of the vectors is used to give priority to some behaviors over others. In this case, the maximum speed of the robot and the modules of the behaviors can be seen in Table 2.9. If the response of the coordinator exceeded the maximum speed value, the response was saturated. This means that when the "Obstacle avoidance" behavior became active, it affected the overall behavior in the strongest way. And the "Avoid trapping" only dominated the "Go to" behavior to depart from possible entrapment situations. All these values are closely related to the sample time of the control architecture. PDL is a methodology which works with derivatives of the speed, in

Vector	Maximum Magnitude [m/s]
Robot speed	0.5
"Obstacle avoidance"	0.15
"Avoid trapping"	0.04
"Go to"	0.03

Table 2.9: Maximum values for the robot speed and behaviors. Found experimentally.
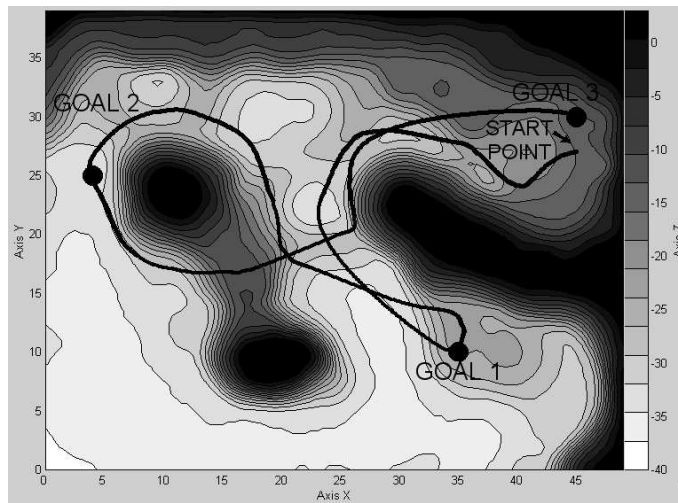
Figure 2.31: Top view of the trajectory carried out by the AUV with Process Description Language.

this case. This means that the dynamics of PDL should be faster than the dynamics of the robot. Experiments with PDL were initially carried out with the same sample time as with the other evaluated architectures. However, the control architecture was not fast enough and the final sample time had to be considerably reduced.

Some graphical results of Process Description Language approach can be seen in the next three figures. Figure 2.31 shows a top view of the simulation and Figure 2.32 shows a vertical view. Finally, Figure 2.33 shows a three-dimensional representation of the simulation.

After reviewing the simulated results it can be said that PDL provides an easy tool to implement a control architecture. Advantages are simplicity and optimized trajectories when the architecture is tuned. However, as
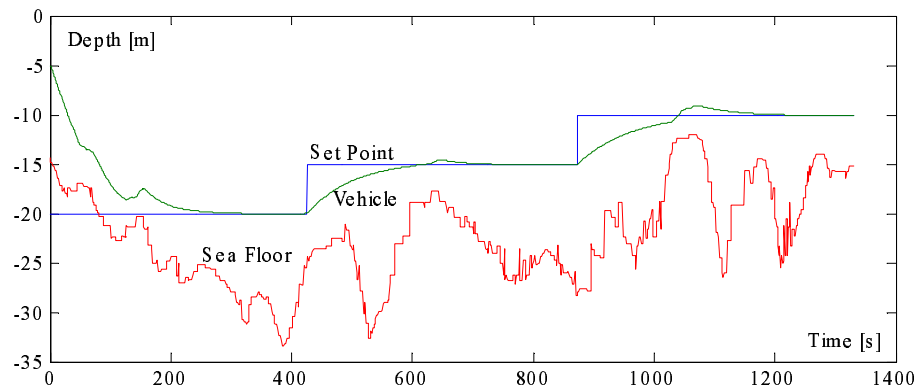
Figure 2.32: Vertical view of the trajectory carried out by the AUV with Process Description Language. The depths of the goal-point, the vehicle and the sea floor are shown.
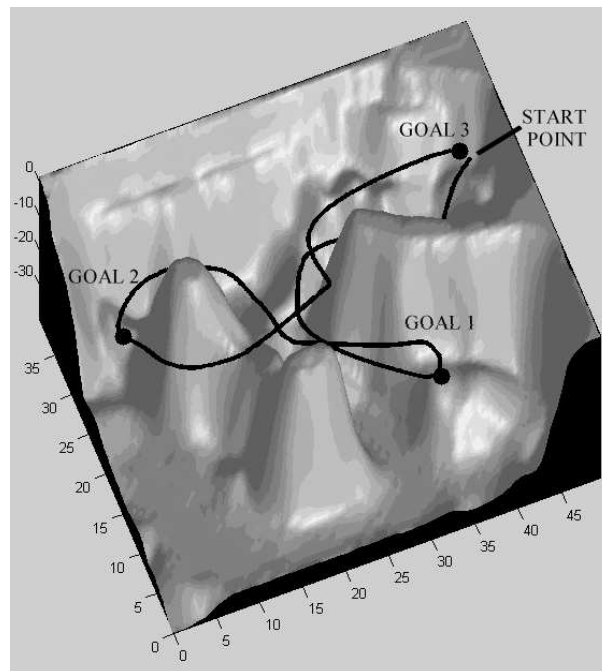
Figure 2.33: Three-dimensional view of the trajectory carried out by the AUV with Process Description Language.

PROCESS DESCRIPTION LANGUAGE	
Developer	Luc Steels, VUB Artificial Intelligence Laboratory
References	[Steels, 1992, Steels, 1993]
Behavioral encoding	Continuous
Coordination method	Cooperative via values integration and normalization
Programming method	Process Description Language
Advantages	Development time and simplicity
Disadvantages	Small sample time, Tuning time, Robustness and Modularity

Table 2.10: Process Description Language features.

Property\Architecture:	1st	2nd	3rd	4th
Performance	SCHE.	PDL	ASD	SUBS.
Modularity	SUBS.	ASD	PDL	SCHE.
Robustness	SUBS.	ASD	PDL	SCHE.
Development time	SCHE.	PDL	SUBS.	ASD
Tuning time	SUBS.	ASD	PDL	SCHE.
Simplicity	SCHE.	PDL	SUBS.	ASD.

Table 2.11: Behavior-based architectures rankings.

the coordinator method is cooperative, the tuning was very difficult. When new behaviors are added, re-tuning is necessary. Also there is the problem of the sample time, which must be faster than with the other approaches. Moreover, since the final velocity vector is obtained incrementally, the architecture acts sequentially and has a low modularity. Table 2.10 summarizes the principal characteristics of the Process Description Language approach. For a description of the advantage/disadvantage terms refer to Table 2.2.

2.5   Coordination Methodology Comparison

Once the four behavior-based architectures had been implemented and tested some conclusions were drawn. It should be noted that the architectures were implemented for a simple but representative task. However, this task was felt to be sufficient to discover the attractiveness and deficiencies of each architecture. As commented above, each architecture exhibited its own advantages and disadvantages and for this reason each would be well suited for a particular application. For each evaluated property, a ranking of architectures has been established, see Table 2.11. In this table, the architecture listed first is the one which best fits the corresponding property.

Looking at the rankings table, it can be seen that properties can be grouped in accordance with the coordination method, as summarizes Ta-

COMPETITIVE METHODS		COOPERATIVE METHODS	
Subsumption and ASD		Motor Schema and PDL	
Advantages	*Disadvantages*	*Advantages*	*Disadvantages*
Modularity	Performance	Performance	Modularity
Robustness	Development Time	Development Time	Robustness
Tuning time	No Simplicity	Simplicity	Tuning Time

Table 2.12: Properties according to the coordination methodology.

ble 2.12. *Competitive* methods (Subsumption and ASD) have robustness, modularity and easy tuning. This is due to the fact that they only have one active behavior at any given moment. Therefore, robustness is preferable because in dangerous situations only a safety behavior will act and the danger is avoided. Modularity should also be considered an important property of competitive methodologies because more behaviors can be added without influencing the old ones. Only the coordinator will have to be adapted to the new input. For this reason the tuning time is very short. Behaviors are tuned independently and once they work properly they never have to be re-tuned.

However, competitive methods have disadvantages as well, mainly in the coordinator design. In order to choose only one active behavior, a complex method must be used (ASD) or a clear understanding of the hierarchy of behaviors is necessary (Subsumption). Once this is done, the final tune-up is very easy. For this reason the development time is usually long and the coordinator can become very complex. Another negative property is slow performance due to the non-instant merging of behaviors. Big changes in the vehicle's heading occur when more than one behavior is acting consecutively.

As competitive approaches, the two methodologies studied, Subsumption and Action Selection Dynamics, possess all these properties. However, they have quite different philosophies. Subsumption is more a low-level approach. The hierarchy of behaviors has to be known and then the network has to be designed. Subsumption offers a series of tools, the suppression and inhibition nodes, to build the network. For this reason, the implementation will be simple, but perhaps the design will be quite difficult. On the other hand, Action Selection Dynamics is a high level approach to building an architecture. All the competence modules are completely described and the design consists of filling in all the module lists. Once all the behaviors are perfectly described, the network will automatically choose the best behavior. In ASD the design will be easier but the implementation is more difficult.

In contrast to competitive approaches, methods with a *cooperative* coordinator have other properties like simplicity, performance and development

time, see Table 2.12. Due to the fact that all behaviors are active, the response will always be a merging of these behaviors. This means that the trajectory described by the robot will be smoother than that described by a competitive method. For this reason performance will be a common property.

Another property is simplicity. The coordinator will be very simple, because the final output is usually a kind of sum of all the behaviors multiplied by the gain factors. In relation to simplicity, the development time will be small. However, this simplicity causes great difficulties in tuning the priority gains as the values are very sensitive and critical. In extreme situations, non-safe behaviors can cancel the safe ones. Unfortunately, the modularity will be very bad as a result, because each new behavior will cause the re-tuning of all the priority gains.

The differences between the two cooperative approaches studied, Motor Schema and Process Description Language, are in the level of abstraction and in the coordinator. In Motor Schema, behaviors are implemented individually with the behavioral library. It's more a high-level design. However, in PDL behaviors are implemented as low-level processes which change the set-points of the actuators a little. Implementation will be simple but the design will be more complex. Nevertheless, the principal difference between the two approaches is found in the coordinator. In Motor schema, the output is obtained every time step from the outputs of the behaviors. On the other hand, in PDL the output is an integration of all the outputs generated before. This implies that a small sample time is needed to assure the stability of the system, which can be problematic if there is a lot of computation to do.

Concluding this comparison, it can be said that, depending on the exigencies of the robot to be controlled, one method can prove to be more appropriate than another. Once the architecture has been implemented, a cooperative method can be more suitable if better performance is necessary. Or a competitive method if robustness is the basic premise. The control architecture could also depend on hardware availability, sensorial system and compatibility with adaptive algorithms.

Chapter 3

Hybrid Coordination of Behaviors

Following the analysis of Behavior-based control architectures in Chapter 2, this chapter contains the first research contribution of this thesis. This approach consists of a hybrid behavior coordination system which is able to act competitively and cooperatively. The purpose of this approach is to benefit from the advantages of both methodologies. The chapter first introduces the behavior-based control layer in which the hybrid coordinator is expected to act. Then, the hybrid coordination methodology is detailed. Finally, the approach is tested with the experimental task presented in Chapter 2. Real results of the hybrid coordinator with the underwater robot URIS are found in Chapter 7, and the experimental set-up which has been specifically developed is detailed in Chapter 6.

3.1 The Behavior-based Control Layer

The control architecture, or high-level controller, proposed in this thesis, is a *hybrid* control architecture. As explained in Chapter 2, the main advantages of hybrid architectures are reactive and fast responses due to a *behavior-based* layer, and mission planning due to a *deliberative* layer. In this case, the deliberative layer has the goal of breaking down the mission to be accomplished into a set of *tasks*. The behavior-based layer has the goal of carrying out each one of these tasks and is compounded of a set of behaviors and a coordinator. The deliberative layer acts over the behavior-based layer by configuring the particular set of behaviors and the priorities existing among them. It activates the best behavior configuration for the current task. To decide if the task is being accomplished properly, it supervises what the robot is per-
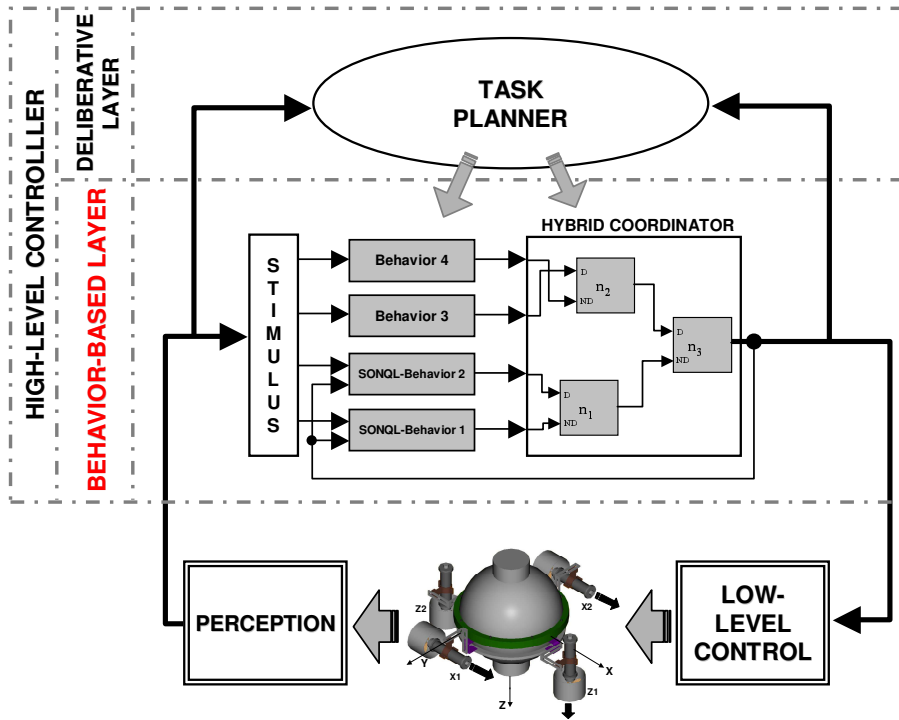
Figure 3.1: General schema of the control system conceived to control an autonomous robot. This thesis has been centered in the design of the behavior-based layer.

ceiving and also the actions that the behavior-based layer is proposing. The behavior-based layer acts over the *low-level controller* generating the actions to be followed. These actions depend directly on the current perception of the robot since behaviors are very reactive. Finally, the low-level controller acts over the actuators to accomplish these robot actions. The whole control system of the robot just described is depicted in Figure 3.1.

This thesis has centered only on the *behavior-based* layer. This layer has been designed using the principles which behavior-based control architectures propose. Therefore, the layer is compounded of a set of behaviors and a coordinator. The most distinctive aspect of the layer is the coordination methodology. In Chapter 2, the coordinators have been classified as competitive or cooperative. In competitive methodologies, only one behavior controlls the robot at each time step. And in cooperative methodologies, the merging of all active behaviors constitutes the action to be performed. As has been demonstrated with robot simulations, the main advantages and disadvantages of behavior-based architectures can be grouped according to

the coordination methodology. The coordination methodology proposed in this thesis, is a *hybrid* coordination methodology, between competition and cooperation, which tries to benefit from the advantages of both.

Another distinctive aspect can be found in the proposed behavior-based layer. Due to the difficulty of manually tuning or designing each behavior, some learning capabilities have been included in the behaviors. As will be presented in Chapter 4, Reinforcement Learning is a suitable technique to online learning when no information about the environment is available. This technique has been used to learn the internal state/action mapping which a reactive behavior contains. The proposed approach will be detailed in Chapter 5.

The next sections describe the general structure of the behavior-based layer and the hybrid coordination method.

3.2 Hybrid Coordination of Behaviors

Competition methods choose only one behavior at each time-step to control the robot. In such methods, the priorities among the behaviors have to be established so that the highest priority behavior becomes activated and will control the robot. When this kind of coordinator is applied in a navigation task, it is usual to obtain a very *robust* system, although the *performance* of the trajectory is sometimes clearly non-optimal. The robustness feature is strongly related to competitive coordinators, since the highest priority behavior will always control the robot. This property is illustrated in Figure 3.2, in which the competitive coordinator chooses the "Obstacle avoidance" behavior to prevent a collision with the obstacle. However, the same coordinator causes a very bad trajectory because two behaviors, the "Obstacle Avoidance" and the "Go To", are consecutively taking the control of the robot.

Cooperative methods offer other advantages. Usually these coordinators are very simple and the obtained trajectory can be optimal. The term *optimal* is here understood as a smooth, short and safe trajectory. The reason is that not only one behavior action is taken by the coordinator, but a merging of all behavior responses. Although high priority behaviors influence the trajectory to a large degree, the control actions of the non-priority behaviors also have an influence on the final trajectory. If the parameters required by the coordinator are properly given, the trajectory can be optimal. However, a small variation in these parameters, or a small change in the environment, can generate a wrong control action. These properties are also illustrated in Figure 3.2. It can be seen how the cooperative coordinator generates an

action against the obstacle in the "robustness" column. The cause of this wrong action is due a higher collision action, as a result of the addition of two behaviors, with respect to the "Obstacle Avoidance" action.

After having analyzed the advantages and disadvantages of the coordination methodology in the preceding paragraphs and in Section 2.5, the necessity of a *hybrid* methodology is fully justified. The main goal of this approach is to take advantage of both approaches. A competitive coordinator assures a good robustness as well as modularity and a short tuning time. On the other hand, a cooperative coordinator merges the knowledge of all the active behaviors, which, in normal situations, implies a rapid accomplishment of the task.

The hybrid coordinator proposed in this thesis is able to operate as both a competitive and a cooperative method. As will be described, the coordination is done through a hierarchy among all the behaviors and an activation level. If higher priority behaviors are fully activated, the coordinator will act as competitive. On the other hand, if higher priority behaviors are partially activated, a merged control action will be generated. These two different behaviors are illustrated in Figure 3.2.

It is necessary to note that the hybrid coordination system has been designed to improve competitive and cooperative methodologies, but will not effect other general problems found in behavior-based systems. A behavior-based control system does not assure the stability of the system. There is no methodology, to the author's best knowledge, to prevent unstable situations which may appear from behavior interactions. Moreover, it is assumed that any modification to assure this stability will go against the advantages of behavior-based robotics. The solution adopted in this thesis assumes that the deliberative layer will detect these unstable situations and actuate over the behavior-based layer.

The next section details the structure and equations of the proposed hybrid coordinator. Later on, the same experimental task used to evaluate the behavior-based control architectures in Chapter 2 is used again to evaluate the hybrid coordinator.

3.3   Coordination Methodology

The proposed hybrid coordination system was designed to coordinate a set of independent behaviors in a behavior-based control architecture or control layer. The methodology allows the coordination of a large number of behaviors without the need of a complex designing phase or tuning phase. The addition of a new behavior only implies the assignment of its priority with
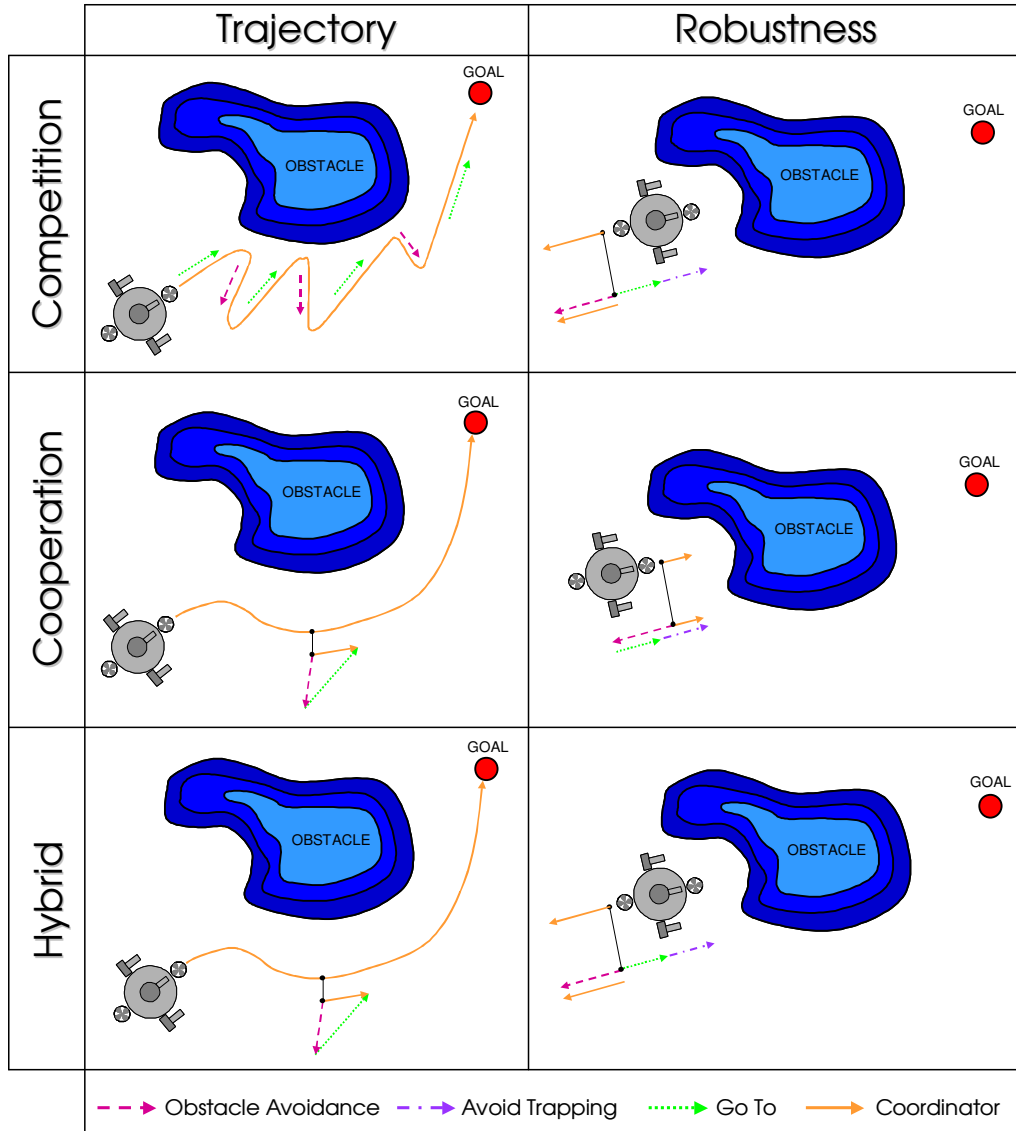
Figure 3.2: Typical behavior of the robot according to the coordination methodology. Competitive coordinators have good robustness when faced with critical situations but can have non-optimized trajectories. Cooperative coordinators have optimized trajectories but can fail in some situations evidencing poor robustness. Finally, a hybrid coordinator behaves cooperatively to achieve optimized trajectories but also competitively when critical situations occur.
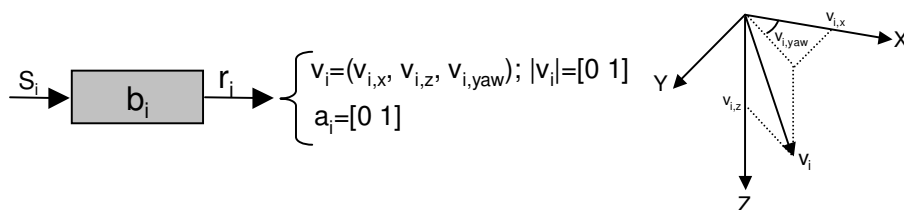
Figure 3.3: The normalized robot control action v_i and the behavior activation level a_i constitute the behavior response r_i.

reference to other behaviors. The hybrid coordinator uses this priority and a behavior activation level to calculate the resultant control action. Therefore, the response r_i of each behavior is composed of the activation level a_i and the desired robot control action v_i, as illustrated in Figure 3.3. The activation level indicates the degree to which the behavior wants to control the robot. This degree is expressed by a numerical value from 0 to 1.

The robot control action is the movement to be followed by the robot. There is a different movement for each degree of freedom (DOF). By movement, we mean the velocity the robot will achieve for a particular DOF. In the case of the underwater robot URIS, which has 3 controllable DOFs, the control action is a vector with three components. This vector is normalized and its magnitude cannot be greater than 1. Therefore, the units of the vector v_i do not correspond to any real units. After the coordination phase, this normalized vector will be re-escalated to the velocities of the vehicle.

The hybrid coordinator uses the behavior responses to compose a final control action. This process is executed at each sample time of the high-level controller. The coordination system is composed of a set of nodes n_i. Each node has two inputs and generates a response which also has an activation level and a control action. The response of a node cannot be discerned from one of a behavior. By using these nodes, the whole coordination process is accomplished. After connecting all the behavior and node responses with other nodes, a final response will be generated to control the robot.

Each node has a *dominant* and a *non-dominant* input. The response connected to the dominant input will have a higher priority than the one connected to the non-dominant. When the dominant behavior is completely activated, $a_d = 1$, the response of the node will be equal to the dominant behavior. Therefore, in this case, the coordination node will behave competitively. However, if the dominant behavior is partially activated, $0 < a_d < 1$, the two responses will be combined. The idea is that non-dominant behaviors can modify the responses of dominant behaviors slightly when these are
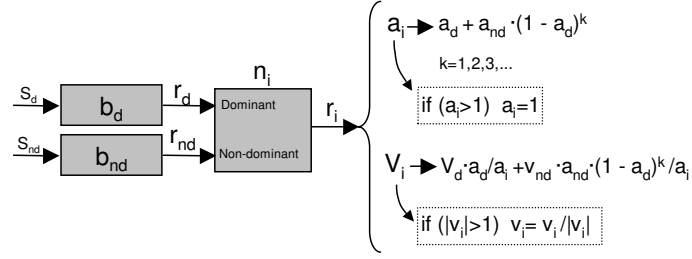
Figure 3.4: Hierarchical Hybrid Coordination Node. The equations used to calculate the response of the node are shown.

not completely activated. In this case, the node will behave cooperatively. Finally, if the dominant behavior is not activated, $a_d = 0$, the response of the node will be equal to the non-dominant behavior. These nodes are called *Hierarchical Hybrid Coordination Nodes* (HHCN) as its coordination methodology changes depending on the activation level of the behaviors and the hierarchy between them.

Figure 3.4 shows the equations used to calculate the response of an HHCN. The activation level will be the sum of the activation levels of the input responses, in which the non-dominant activation level has been multiplied by a reduction factor. This factor, $(1-a_d)^k$, depends on the activation of the dominant behavior and on the value of the integer parameter k. If $k = 1$, the activation level will linearly decrease as a_d increases. If more drastic reduction is desired, the value of k can be set at 2,3,4,... Figure 3.5 shows how the reduction factor changes depending on k. This parameter does not have to be tuned for each node. The same value, for example a quadratic reduction $k = 2$, can be applied to all the coordination nodes. Finally, if the new activation level is larger than 1, the level is saturated to 1.

The control action is calculated in the same way as the activation level. Vector v_i will be the sum of v_d and v_{nd}, applying the corresponding proportional factors. Therefore, each component of v_d will be taken in the proportion of the activation level, a_d with respect to a_i. And, each component of v_{nd} will be taken in the proportion of the reduced activation level, a_{nd} with respect to a_i. If the module of v_i is larger than 1, the vector will be resized to a magnitude equal to 1.

An example of the use of the hierarchical hybrid coordination node is seen in Figure 3.6. In this figure, two different situations are depicted. In the first situation, the node acts *cooperatively* generating an action which mainly follows the dominant response, but it is also affected by the non-dominant response. In the second situation, the dominant behavior is completely acti-
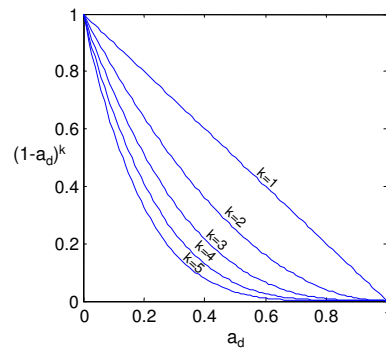
Figure 3.5: Reduction factor applied to the activation level of the non-dominant behavior depending on the activation of the dominant and the factor k.

vated and the node acts *competitively.*

As commented on above, the hybrid coordinator is composed of a set of HHCNs which connect each pair of behaviors or nodes until a final response is generated. In order to build up this network of nodes, it is necessary to set up the hierarchy among the behaviors. This hierarchy will depend on the task to be performed. Once the priorities have been set, usually by the mission designer, the hybrid coordinator will be ready to use. Figure 3.7 shows an example of a set of three nodes which coordinate four behaviors.

The advantage of the hybrid coordination system is that the coordinator has good modularity, see Table 2.2. Each time a new behavior is added, the priority of the new behavior with respect to the others is the only aspect which has to be chosen. The tuning time is also a very good property since after implementing a new behavior, only the k parameter has to be changed. Therefore, these advantages, together with the advantage of robustness from competition and optimized trajectories from cooperation, point out the suitability of the proposed hybrid coordinator. The next section shows an example in which the hybrid coordinator has been used in a simulated task. In Chapter 7, real results of the hybrid coordinator with the underwater robot URIS are presented.

3.4   Experimental task

The hybrid coordinator has been tested with the same experimental task used in Chapter 2 and described in Section 2.3. The task consisted of reaching a collection of way-points avoiding obstacles and entrapment. A total number
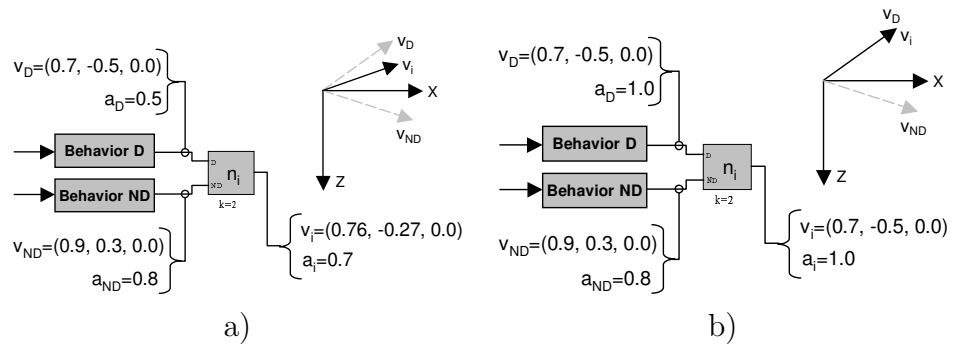
Figure 3.6: Response of the HHCN. Only two-dimensional actions are represented for a better understanding. The node acts cooperatively a), when the dominant behavior is not fully activated. In case the dominant behavior is completely activated, a competitive action is generated b).
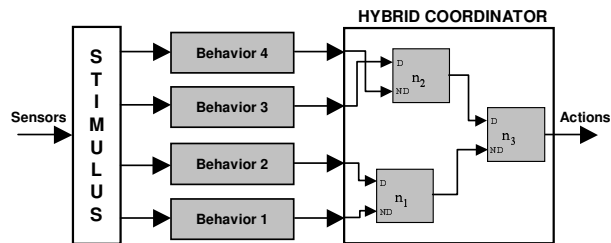
Figure 3.7: Example of a reactive layer with four behaviors and the hybrid coordinator. The priority among the behaviors depends on the input connections of the HHCNs.

of three behaviors were used. The first behavior, "Go to", is in charge of driving the vehicle toward the way-points. The second behavior, "Obstacle avoidance", has the goal of maintaining the vehicle away from obstacles. The third behavior, "Avoid trapping", is used to depart from zones in which the vehicle could be trapped.

The three behaviors were implemented in three different functions. The response of each behavior was composed by the control action and the activation level. As far as the activation level is concerned, each behavior generated a value corresponding to the farness in the achievement of the behavior goal. Therefore, the "Obstacle avoidance" behavior was completely activated when an obstacle was very close. At certain distance from the obstacles, the behavior started to become less activated until activation was 0. These distances or thresholds had to be tuned according to a minimum security distance, and to the vehicle's dynamics. It is important to note that the behavior can be activated by an obstacle detected in only one DOF. This means that even if there are not obstacles in the other DOFs, the behavior will become activated, slowing down the velocity in all the DOFs. Similarly, the "Avoid trapping" behavior will be activated according to the recent path of the vehicle. If the robot has been located in the same zone for a long time, the behavior will be completely activated. In the event the vehicle starts travelling, the activation will decrease. Therefore, in this case, the behavior will also require a tuning phase. Finally, the "Go to" behavior will always be completely activated since the task consists of achieving a set of positions.

The three behavior responses were connected to the coordination module using two HHCNs. The hierarchy of behaviors was as follows: at the top, the "Obstacle avoidance" behavior was followed by the "Avoid trapping" and "Go to" behaviors. This hierarchy primarily assures that the vehicle stays away from obstacles. As the task never requires proximity to obstacles, the "Obstacle avoidance" with the highest priority, assures the safety of the vehicle. At a second level, the "Avoid trapping" behavior takes control over the "Go to" behavior if the vehicle becomes trapped. Finally, the "Go to" behavior has the lowest priority and guides the robot to the target point. A quadratic reduction factor, $k = 2$, was used. The three behaviors were implemented as shown in Figure 3.8.

Some results with the Hybrid Coordination Methodology can be seen in the next three figures. Figure 3.9 shows a top view of the simulation and Figure 3.10 shows a vertical view. Finally, Figure 3.11 shows a three-dimensional representation of the simulation. In these figures, it can be observed how the hybrid coordination system was able to fulfill the simulated task. The performance of the hybrid coordination nodes can be seen in Figure 3.12. In this figure, the responses of the three behaviors and the response of the coordi-
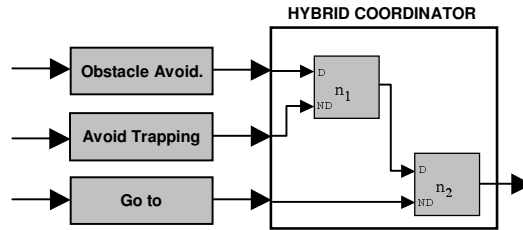
Figure 3.8: Task implementation with the hybrid coordinator.
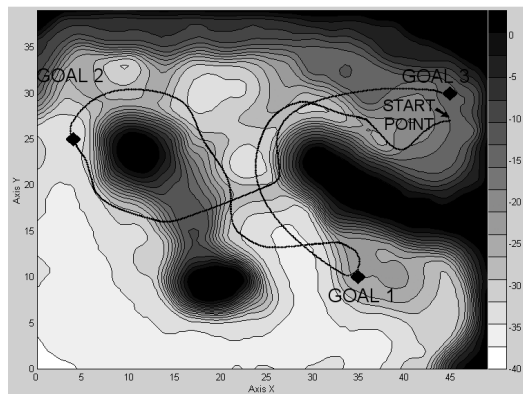
Figure 3.9: Top view of the trajectory carried out by the AUV.
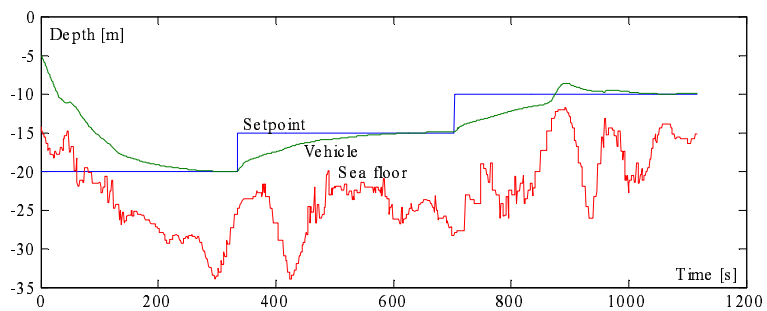
Figure 3.10: Vertical view of the trajectory carried out by the AUV.
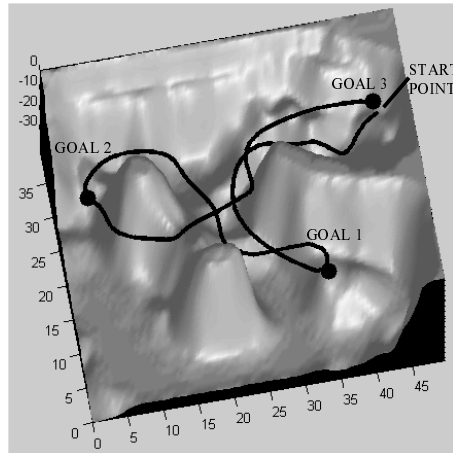
Figure 3.11: Three-dimensional view of the trajectory carried out by the AUV.

nator (from the final HHCN) can be seen during the first 90 seconds. When the activation of the coordinator coincides with the activation of one behavior, the coordinator is acting competitively or only one behavior is activated. On the other hand, if the activation of the coordinator has an intermediate value, the coordinator is acting cooperatively. In the same figure, the control actions of the three DOFs of the robot can be observed. The final control actions, generated by the coordinator, are generated cooperatively or competitively depending on the activation levels.

The results obtained in this simulated experiment with the hybrid coordinator fulfilled the desired goals. The coordinator demonstrated it could react cooperatively and competitively, achieving an optimized trajectory and maintain a good robustness. Therefore, the two advantages of competitive and cooperative coordinators have been mixed in this new hybrid coordination system. As a consequence, the methodology is also very modular, and the tuning time consists of the independent tuning of each behavior. This experimental task has shown the effectiveness of the hybrid coordinator in simulation.

3.5   Discussion

The hybrid coordination approach was designed to face some disadvantages which were experimentally found in other approaches. As described, the coordinator can act cooperatively and competitively depending on the ac-
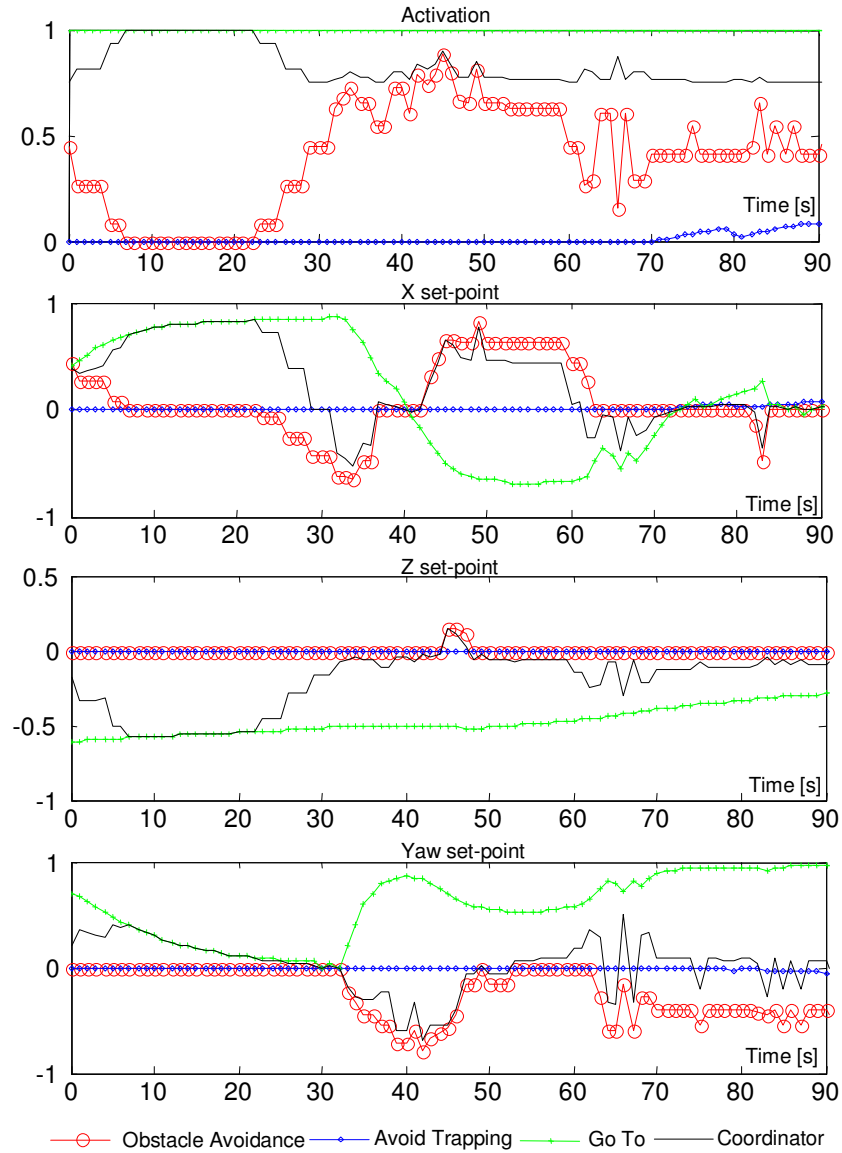
Figure 3.12: Response of the 3 behaviors and the one generated by the hybrid coordinator during the first 90 seconds of the task. The activation level of these four responses is shown in the first graph. Then, the control actions for the X, Z and Yaw degrees of freedom are also shown.

tivation levels. There are no parameters to tune and the hierarchy among the behaviors is the only knowledge that a designer must introduce. Evidently, if a good performance is desired, the designer's work will not be the implementation of the coordinator but in the tuning of each independent behavior. Hence, the advantage of this simple coordinator is that the robustness is always guaranteed and that a good performance is also feasible with the condition that behaviors are properly implemented.

In the reasoning of the hybrid coordinator, a set of advantages have been mentioned, such as robustness, optimal trajectories, simplicity and modularity. All these features have been qualitatively analyzed in relation with the architectures treated in Chapter 2. Due to the difficulty in numerically measuring the evaluated features, it has not been possible to obtain quantitative results. The effectiveness of the hybrid coordination system can also be verified in real systems. As will be discussed in Chapter 7, the approach demonstrated its advantages in a real task with the autonomous underwater robot URIS.

Finally, to the author's best knowledge, there are no other proposals using a hybrid behavior coordinator. It could be said that any cooperative coordinator can also act competitively, for example, if only one behavior is activated. However, the drastic methodology in which the hybrid coordinator acts, depending on the activation of the highest priority behaviors, is not present in any other approach. This feature is considered to be an important factor in controlling an autonomous robot.

Chapter 4

Reinforcement Learning

Reinforcement Learning (RL) is learning by interaction with the environment in order to accomplish a goal. This chapter overviews the field of RL and its application to robotics. The chapter first describes the problem addressed by RL from a general point of view. Then it introduces Finite Markov Decision Processes (FMDP) as a tool to express the RL problem (RLP). In order to solve the RLP, three methodologies are presented, among which Temporal Difference (TD) is the most suitable in a robotics domain. The chapter follows with the description of two representative TD algorithms; the Actor-Critic algorithm and the Q_learning algorithm. After the presentation of the field of Reinforcement Learning, several research issues are presented. Among them the generalization problem, which highly affects robotics, is emphasized. The most common approaches to confront this problem and its application to robotics tasks are overviewed. Most part of this chapter is an overview of RL theory. For a deeper understanding of this field, refer to the book "Reinforcement Learning. An introduction", written by Sutton and Barto [Sutton and Barto, 1998].

4.1   The Reinforcement Learning Problem

Reinforcement Learning (RL) is an approach to learning by trial and error in order to achieve a goal. Basic to the learning process is the interaction between the RL algorithm and an environment. An RL algorithm does not use a set of examples which show the desired input/output response, as do *supervised learning* techniques. Instead, a reward given by the environment is required. This reward evaluates the current state of the environment. The *Reinforcement Learning Problem* (RLP) consists of maximizing the sum of future rewards. The goal to be accomplished by RL is encoded in the received

reward. To solve the problem, an RL algorithm acts over the environment in order to yield maximum rewards. Any algorithm able to solve the RLP is considered an RL algorithm.

Several elements appear in all RLPs. The first is the RL algorithm or the *learner*. The learner interacts with an *environment*. The environment comprises everything outside the learner. The learner can usually observe the *state* of the environment and interacts with it by generating an *action*. The environment reacts to the action with a new state and a *reward*. The reward is a scalar value generated by the *reinforcement function* which evaluates the current state and/or the last executed action according to the RLP. This learner/environment interaction can be represented in a discrete time base as in the diagram shown in Figure 4.1. In each time iteration t, the learner observes the state s_t and receives the reward r_t. Following the rules of the RL algorithm, the learner generates an action a_t. The environment reacts to the action changing to state s_{t+1} and generates a new reward r_{t+1}. Both will be taken during the next iteration. The state space $\{S\}$ can contain different variables (n-dimensional) and the action space $\{A\}$ different actions (m-dimensional). The time notation used for the states, rewards and actions is interpreted as: being at state s_t and applying action a_t, the environment reacts with a reward r_{t+1} and a new state s_{t+1}. A sequence of states, actions and rewards is shown in Figure 4.1, in which the states are represented as circles, the actions as lines and the rewards as dots. The most important features of the learner and environment are enumerated hereafter:

Learner :

- Performs the learning and decides the actions.

- Input: the state s_t and reward r_t (numerical value).

- Output: an action a_t.

- Goal: to maximize the amount of rewards $\sum_{i=t+1}^{\infty} r_i$.

Environment :

- Everything outside the learner.

- It reacts to actions with a new state.

- Contains the Reinforcement Function which generates the rewards.
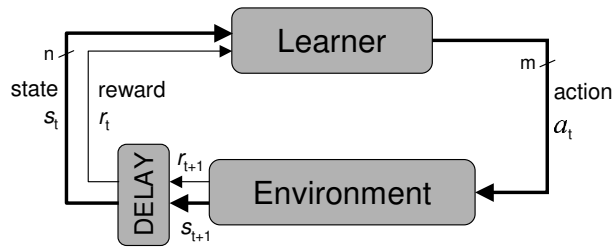
Figure 4.1: Diagram of the learner/environment interaction.
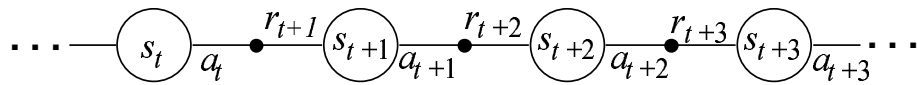
Figure 4.2: Sequence of states, actions and rewards.

The learner interacts with the environment in order to find the action which maximizes the sum of future rewards. The action to be applied will always depend on the current state. Therefore, the learner needs a *mapping function* which relates each state with an action. Two processes are commonly used to find the best actions: *exploration* and *exploitation*. Exploration means that the learner will have to select the actions which have not yet been used for a particular state. On the other hand, in order to solve the RLP, the best known actions must be exploited. The compromise between exploration and exploitation is a common feature in RL. It is important to note that the final goal of RL is to maximize the sum of future rewards. The best state/action mapping can propose actions which do not maximize immediate rewards, but actions which maximize the long term sum of rewards. The effect of an action can also be delayed in terms of the received reward. An action can be the best one to solve the RLP, but high rewards may not come until after some iterations.

The main keys of Reinforcement Learning which have been related are:

- Learning through rewards, not through examples.

- Trial-Error search to find the best state/action mapping.

- Exploration/exploitation dilemma.

- Delayed rewards, the effect of the actions is not immediately rewarded.

Before finalizing this brief introduction to Reinforcement Learning, the

basic RL functions will be described. These functions constitute the defini-
tion of the RLP, and the tools needed to find the solution.

Policy Function. A policy function indicates the action to be taken at each
moment. A policy is a mapping between the states and the actions
and is contained in the learner. When this mapping is stochastic, the
policy is usually represented with the $\pi(s, a)$ function, which contains
the probability of choosing action a from state s. When a policy solves
the RLP; that is, the action related to each state maximizes the sum
of future rewards, it is said that the policy is *optimal* π^*. A policy can
be *random*, in which case the action will be chosen randomly. A *greedy*
policy contains the best mapping known by the learner; that is, the
actions supposed to best solve the RLP. A *greedy action* is an action
taken from a greedy policy. Finally, a $\epsilon - greedy$ policy selects random
actions with ϵ probability and greedy actions with $(1 - \epsilon)$ probability.

Reinforcement Function. The reinforcement function is located in the en-
vironment and indicates the goal to be accomplished by the learner. A
reinforcement function $r(s, a)$ is a deterministic function which relates
a state and/or an action to a single numerical value or reward. The
learner uses the reward to learn the optimal policy. The reinforcement
function gives an immediate evaluation of the learner, which is not the
criterion to be improved in Reinforcement Learning.

Value Function. The value function contains the expected sum of future
rewards following a particular policy. This function is used and modi-
fied by the learner to learn the optimal policy. There are two kinds of
value functions. The *State-Value* function $V^\pi(s)$ contains the expected
sum of future rewards from state s followed by the policy π. The
Action-Value function $Q^\pi(s, a)$ contains the expected sum of future re-
wards from state s, applying the action a, and following afterwards the
policy π. The action-value function $Q^\pi(s, a)$ will be equal to the state-
value function $V^\pi(s)$ for all the actions which are greedy with respect to
π. Unlike the reinforcement function, the value functions give the long
term evaluation of the learner following a particular policy. These func-
tions directly show the performance of the learner in solving the RLP.
When following the optimal policy π^*, the value functions $V^*(s)$ and
$Q^*(s, a)$ show the maximum sum of expected rewards, and therefore
the functions are also optimal. In the same way, if the learner achieves
the optimal value functions, the optimal policy can be extracted from
it.

Dynamics Function. This function is located in the environment and is usually not known. A dynamics function is usually a stochastic function which generates the future state of the environment from its previous state and the applied action. Although this function is not known, the state transitions caused by the dynamics are contained in the value functions.

A RL algorithm uses these functions to solve the RLP. The learning process is performed over one of the value functions, the state or the action value functions. The algorithm proposes a learning update rule to modify the value function and also proposes a policy to be followed by the learner-environment interaction. If the RL algorithm converges after some iterations, the value function has changed to the optimal value function from which the optimal policy can be extracted. The solution of the RLP is accomplished by following the state-action mapping contained in this optimal policy π^*. Figure 4.3 shows a diagram of the phases found in a RL algorithm to solve the RLP.

4.2 RL with Finite Markov Decision Processes

Reinforcement Learning theory is usually based on *Finite Markov Decision Processes* (FMDP). The use of FMDP allows a mathematical formulation of the RLP, therefore, the suitability of RL algorithms can be mathematically demonstrated. In order to express an RLP with FMDP, the environment has to accomplish the *Markov Property*. An environment which accomplishes the Markov property shows in its state all the relevant information to predict the next state. This means that in order to predict the next state of the environment, the knowledge of the past states and actions are not necessary if the present state and action is known. At the same time, the future reward does not depend on past states and actions. Therefore, in a Markovian state the prediction of the next state and reward is accomplished by knowing only the present state and action. Equation 4.1 formally defines the Markov property equalling the probability of predicting the next state and reward with and without knowing the past states, actions and rewards.

$$Pr\{s_{t+1} = s', r_{t+1} = r'|s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., s_0, a_0\} =$$
$$Pr\{s_{t+1} = s', r_{t+1} = r'|s_t, a_t\} \tag{4.1}$$

If the environment accomplishes the Markov property it is considered a Markov Decision Process (MDP). Moreover, if the state and action spaces
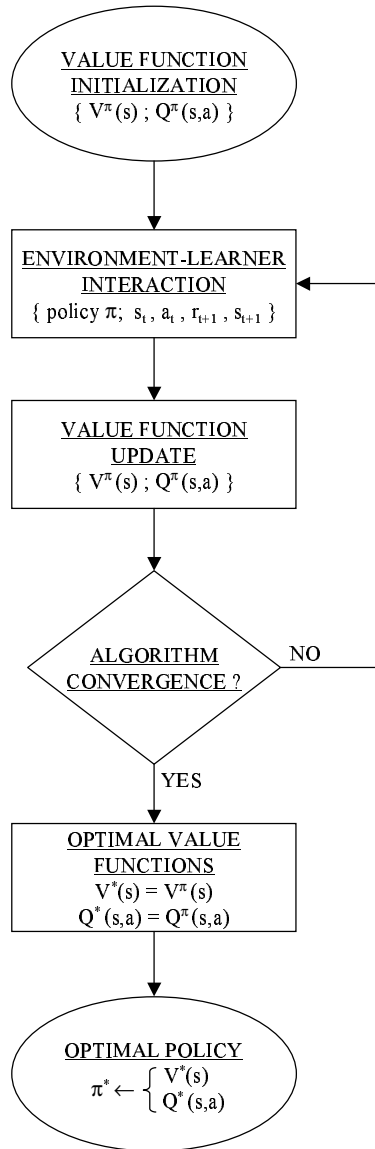
Figure 4.3: Typical phases of a Reinforcement Learning algorithm to solve the RLP. The learner interacts with the environment, updating the value function according to the RL algorithm. When the algorithm converges, the optimal value function has been found from which the optimal policy is extracted. The optimal policy solves the RLP, proposing the actions which maximize the sum of future rewards.

are finite, the environment is considered a Finite MDP (FMDP). Having discrete state and action spaces, the stochastic dynamics of the environment can be expressed by a transition probability function $P_{ss'}^a$. This function contains the probability of reaching the state s' if the current state is s and action a is executed. Equation 4.2 formally defines this function. Also, the reinforcement function can be expressed as a deterministic function $R_{ss'}^a$. In this case, the function contains the expected reward received when being at s, a is applied and s' is reached, see Equation 4.3

$$P_{ss'}^a = Pr\{s_{t+1} = s'|s_t = s, a_t = a\} \tag{4.2}$$

$$R_{ss'}^a = E\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\} \tag{4.3}$$

$P_{ss'}^a$ and $R_{ss'}^a$ specify the dynamics of the environment. One of the learner's functions, the value function, is highly related with this dynamic. For a particular learner policy π, the value function (V^π or Q^π) can be expressed in terms of $P_{ss'}^a$ and $R_{ss'}^a$, and as will be shown, the optimal value function (V^* or Q^*) can also be determined. Once this function is obtained, the optimal policy π^* can be extracted from it. Before getting to these expressions, a new function has to be defined.

As has been stated, the goal of RL is to maximize the sum of future rewards. A new function R_t is used in the FMDP framework to express this sum, as Equation 4.4 shows. This sum finishes at time T, when the task which RL is trying to solve finishes. The tasks having a finite number of steps are called *episodic tasks*. However, RL is also suitable for solving tasks which do not finish at a certain number of time steps. For example, in a robotics task, the learner may be continually activated. In this case, the tasks are called *continuing tasks* and can run to infinite. To avoid an infinite sum of rewards, the goal of RL is reformulated to the maximization of the *discounted sum* of future rewards. The future rewards are corrected by a discount factor γ as expressed in Equation 4.5.

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \tag{4.4}$$

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{4.5}$$

Setting the discount factor between a range of $0 \leq \gamma \leq 1$, the infinite sum of rewards does not achieve infinite values and therefore the RLP can be solved. In addition, the discount factor allows the selection of the number of future rewards to be maximized. For $\gamma = 0$ only the immediate reward is

maximized. For $\gamma = 1$ the maximization will take into account the infinite sum of rewards. Finally, for $0 < \gamma < 1$ only a reduced set of future rewards will be maximized.

The two value functions, V^π and Q^π, can be expressed in terms of the expected future reward R_t. In the case of the *state-value* function, the value of a state s under a policy π, denoted $V^\pi(s)$, is the expected discounted sum of rewards when starting in s and following π thereafter. For the *action-value* function, the value of taking action a in state s under policy π, denoted $Q^\pi(s, a)$, is the expected discounted sum of rewards when starting in s, the action a is applied, and following π thereafter. Equations 4.6 and 4.7 formally define these two functions.

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \tag{4.6}$$

$$Q^\pi(s, a) = E_\pi\{R_t | s_t = s, a_t = a\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\} \tag{4.7}$$

The last two equations define the value functions obtained when following a particular policy π. To solve the RLP, the optimal policy π^* which maximizes the discounted sum of future rewards has to be found. As the value functions indicate the expected sum of future reward for each state or state/action pair, an optimal value function will contain the maximum values. Therefore, from all the policies π, the one having a value function (V^π or Q^π) with maximum values in all the states or state/action pairs will be an optimal policy π^*. It is possible to have several policies $(\pi_1^*, \pi_2^*, ...)$ which accomplish this requirement, but only one optimal value function can be found (V^* or Q^*). Equations 4.8 and 4.9 reflect this statement.

$$V^*(s) = \max_\pi V^\pi(s) \tag{4.8}$$

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \tag{4.9}$$

In order to find these optimal value functions, the Bellman equation [Bellman, 1957] is applied. This equation relates the value of a particular state or state/action pair with the value of the next state or state/action pair. To relate the two environment states, the dynamics of the FMDP ($P_{ss'}^a$ and $R_{ss'}^a$) is used. The *Bellman optimality equations* for the state and action value functions are found in Equations 4.10 and 4.11.

$$V^*(s) = \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^*(s')] \tag{4.10}$$

$$Q^*(s,a) = \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma \max_{a'} Q^*(s',a')] \tag{4.11}$$

The Bellman optimality equations offer a solution to the RLP by finding the optimal value functions V^* and Q^*. If the dynamics of the environment is known, a system of equations with N equations and N unknowns can be written using Equation 4.10, being N the number of states. This nonlinear system can be solved getting the V^* function. Similarly, the Q^* function can be found.

Once V^* is known, the optimal policy can be easily extracted. For each state s, any action a which causes the environment to achieve a state s' with maximum state value with respect to the other achievable states can be considered as an *optimal action*. The set of all the states with its corresponding optimal actions constitutes an optimal policy π^*. It is important to note that to find each optimal action, it is only necessary to compare the state value of the next achievable states. This is due to the fact that the state-value function V^* already contains the expected discounted sum of rewards for these states. In the case where the Q^* is known, the extraction of the optimal policy π^* is even easier. For each state s, the optimal action will be the action a which has a maximum $Q^*(s,a)$ value, see Equation 4.12.

$$\pi^*(s) = arg \max_{a \in A(s)} Q^*(s,a) \tag{4.12}$$

This section has formulated the Reinforcement Learning problem using Finite Markov Decision Processes. It has also pointed out how to find the solution of the RLP when the dynamics of the environment is known. The next section will show how to find a solution to the RLP when the dynamics is not known, which is the most common case.

4.3   Methodologies to solve the RLP

There are several methodologies to solve the Reinforcement Learning Problem formulated as an FMDP. This section summarizes the main features of three methodologies: Dynamic Programming, Monte-Carlo Methods and Temporal Difference learning. The main purpose is to show the advantages and limitations in solving the RLP.

Dynamic Programming. This methodology is able to compute the optimal policy π^* by using the dynamics of the FMDP ($P_{ss'}^a$ and $R_{ss'}^a$). Dynamic Programming (DP) algorithms act iteratively to solve the Bellman optimality equations. DP algorithms are able to learn *online,*

that is, while the learner is interacting with the environment. At each iteration, they update the value of the current state based on the values of all possible successor states. The knowledge of the dynamics is used to predict the probability of the next state to occur. As the learning is performed online, the learner is able to learn the optimal policy while it is interacting with the environment. The general update equation for the state-value function is shown in Equation 4.13.

$$V_{k+1}(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')] \qquad (4.13)$$

In this equation, the state-value function at iteration $k + 1$ is updated with the state-value function at iteration k. Instead of using the real value $V^\pi(s')$, which is not known, an estimate of it is used $V_k(s')$. This feature, called bootstrapping, allows the algorithm to learn online. DP algorithms compute the value functions for a given policy. Once this value function is obtained, they improve the policy based on it. Iteratively, DP algorithms are able to find the optimal policy which solves the RLP.

Monte-Carlo Methods. Monte-Carlo (MC) methods are also able to solve the RLP. MC algorithms do not need the dynamics of the environment. Instead, they use the experience with the environment to learn the value functions. Another important feature of MC methods is that they do not learn online. MC algorithms interact with the environment following a particular policy π. When the episode finishes, they update the value of all the visited states based on the received rewards. Repeating the learning in several episodes, the value function for a particular policy is found. Equation 4.14 shows the general update rule to estimate the state-value function. It can be observed that the current prediction of the state-value $V^\pi_k(s)$ is modified according to the received sum of rewards R_t. There is also a learning rate α which averages the values obtained in different episodes.

$$V^\pi_{k+1}(s_t) \leftarrow V^\pi_k(s_t) + \alpha[R_t - V^\pi_k(s_t)] \qquad (4.14)$$

After the evaluation of a policy, MC methods improve this policy based on the learnt value function. By repeating the evaluation and improving phases, an optimal policy can be achieved.

MC methods are not suitable for continuing tasks as they can not update the value functions until a terminal state is found. The fact

that they learn using the real sum of received rewards does not obligate the algorithms to bootstrap, unlike DP algorithms. However, MC algorithms can not learn online and, therefore, they are not suitable for most problems which RL tries to solve. The main advantage of Monte-Carlo methods respecting Dynamic Programming is the use of the environment experience instead of its dynamics.

Temporal Difference learning. Temporal Difference learning (TD) is another methodology to solve the RLP. The main advantages of the previous methods are found in TD algorithms. First of all, TD learning is able to learn online as DP. It uses bootstrapping to estimate the unknown value functions. As a consequence, it can be applied in continuing tasks. The second advantage is that it does not need the dynamics of the environment, as do MC methods. Instead, it uses the experience with the environment. The general update rule for the state-value function can be seen in Equation 4.15. Similarly to MC methods, the updating of the state value $V_{k+1}^{\pi}(s_t)$ is accomplished by comparing its current value with the discounted sum of future rewards. However, in TD algorithms this sum is estimated with the immediate reward r_{t+1} plus the discounted value of the next state. This bootstrapping allows the learning to be online. Also, the update rule does not require the dynamics transition probabilities, which allow TD algorithms to learn in an unknown environment.

$$V_{k+1}^{\pi}(s_t) \leftarrow V_k^{\pi}(s_t) + \alpha[r_{t+1} + \gamma V_k^{\pi}(s_{t+1}) - V_k^{\pi}(s_t)] \qquad (4.15)$$

TD algorithms are suitable for solving the RLP when the dynamics is not known and when the learning has to be performed online, as in most robotics tasks.

From the three described methodologies, Temporal Difference learning is the one which best fits the features of the most common Reinforcement Learning problems. TD algorithms are the most used algorithms in RL. The next section describes some of the most popular and classic algorithms.

4.4   Temporal Difference Algorithms

The previous section presented Temporal Difference methods as the most suitable methodology to solve the Reinforcement Learning Problem. The

main advantages of this technique are the fact that the learning can be performed online, and the non-necessity of a dynamics model of the environment. This section shows the learning process of TD algorithms by describing two well-known techniques. The first one is the *actor-critic* algorithm which learns an optimal policy by using the state-value function. The second technique is the popular *Q_learning* algorithm which uses the action-value function in the learning process. These two algorithms are good examples of the performance of TD algorithms, although other algorithms can be found. Finally, this section reviews the concept of *Eligibility Traces* which is used to link Temporal Difference methods with Monte-Carlo methods. Eligibility Traces modify basic TD algorithms accelerating the learning process in most cases.

A very important feature of RL algorithms is the policy they must follow in order to guarantee the convergence. There are algorithms which to ensure the learning of the optimal policy, have to follow a particular policy. That is, at each time step, the action to be executed by the learner is determined by a fixed rule. If the learner does not follow this action, the algorithm cannot converge the RLP solution. These algorithms are called *on-policy* algorithms. On the other hand, there are algorithms which do not impose a particular policy to ensure the convergence. Instead, these algorithms usually need to visit all the states and actions regularly. The second kind of algorithm is called an *off-policy* algorithm. It is usually desirable to use an off-policy algorithm in a real system, since the actions can be externally modified by a supervisor without interrupting the learning process. From the two analyzed algorithms, the actor-critic is an on-policy algorithm and the Q_learning is an off-policy algorithm.

4.4.1 Actor-Critic methods

Actor-Critic [Witten, 1977] is not a particular TD algorithm, but a methodology to solve the RLP. Actor-Critic methods have two distinctive parts, the *actor* and the *critic*. There are algorithms which implement only one of the parts and, therefore, they have to be combined with other algorithms. The actor part contains the policy which will be followed. It observes the state of the environment and generates an action according to this policy. On the other hand, the critic observes the evolution of the states and criticizes the actions made by the actor. The critic contains a value function which tries to learn according to the actor policy. Figure 4.4 shows the general diagram of an Actor-Critic method.

In Actor-Critic methods, the critic typically uses the state-value function $V^\pi(s)$. This function contains the value of being in one state s when following
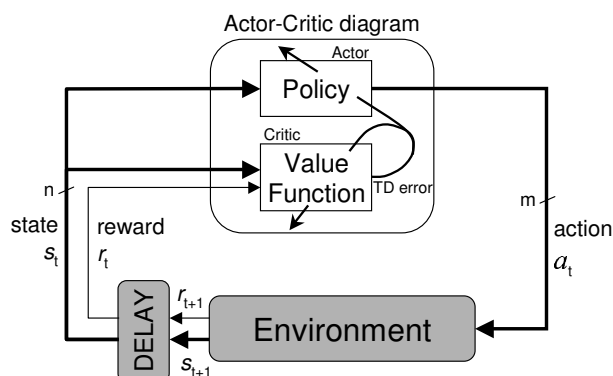
Figure 4.4: General diagram of the Actor-Critic methods.

a particular policy π. Since this policy cannot be extracted from $V^\pi(s)$ without knowing the dynamics of the environment, the Actor-Critic schema is used. Hence, the actor is initialized to a policy which relates all the states with an action, and the critic learns the state-value function of this policy. According to the values of the visited states, the critic calculates the value difference or *TD error*, and informs the actor. Finally, the actor modifies its policy according to this value difference. If the value of two consecutive states increases, the probability of taking the applied action increases. On the contrary, if the value decreases, the probability of taking that action decreases. Actor-critic methods refine the initial policy until the optimal state-value function $V^*(s)$ and an optimal policy π^* are found.

A common property of Actor-Critic methods is the fact that the learning is always on-policy. The action to be followed is the one indicated by the actor policy. The critic will always learn the state-value function for this policy. A policy $\pi_t(s, a)$ is defined as a set of probabilities between all the states and all the actions:

$$\pi_t(s, a) = Pr\{a_t = a | s_t = s\} \tag{4.16}$$

According to the TD error, these probabilities will be decreased or increased. The critic estimates the state-values using the basic TD update rule shown in Equation 4.15. Finally, a general Actor-Critic algorithm can be found in Algorithm 1. As can be seen, in addition to the learning rate α and the discount factor γ, a new parameter β is used to determine the change of $\pi(s_t, a_t)$.

Algorithm 1: General Actor-Critic algorithm

---

1. Initialize $V(s)$ arbitrarily, for all $s \in S$
2. Repeat until π is optimal:
   - (a) $s_t \leftarrow$ the current state
   - (b) $a_t \leftarrow$ action given by π_t for s_t
   - (c) Take action a_t, observe reward r_{t+1} and the next state s_{t+1}

   Critic update:
   - (d) $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$
   - (e) $V(s_t) \leftarrow V(s_t) + \alpha \delta_t$

   Actor update:
   - (f) $\pi(s_t, a_t) \leftarrow \pi(s_t, a_t) + \beta \delta_t$

4.4.2 Q_learning

The Q_learning algorithm [Watkins and Dayan, 1992] is another Temporal Difference algorithm. As distinguished from Actor-Critic methods, Q_learning uses the action-value function Q^π to find an optimal policy π^*. The $Q^\pi(s,a)$ function has an advantage with respect to the state-value function V^π. Once the optimal function Q^* has been learnt, the extraction of an optimal policy π^* can be directly performed without the requirement of the environment dynamics or the use of an actor. The optimal policy will be composed of a mapping relating each state s with any action a which maximizes the $Q^*(s,a)$ function.

Another important feature of Q_learning is the off-policy learning capability. That is, in order to learn the optimal function Q^*, any policy can be followed. The only condition is that all the state/action pairs must be regularly visited and updated. This feature, together with the simplicity of the algorithm, makes Q_learning very attractive for a lot of applications. In real systems, as in robotics, there are many situations in which not all the actions can be executed. For example, to mantain the safety of a robot, an action cannot be applied if there is any risk of colliding with an obstacle. Therefore, if a supervisor module modifies the actions proposed by the Q_learning algorithm, the algorithm will still converge to the optimal policy.

The Q_learning algorithm can be shown in Algorithm 2. As can be seen, the policy applied is the $\epsilon - greedy$ policy, although any other policy could have been used as well. Recovering the definition of the action-value function, the $Q^*(s,a)$ value is the discounted sum of future rewards when action a is executed from state s, and the optimal policy π^* is followed afterwards. To estimate this sum, Q_learning uses the received reward r_{t+1} plus the discounted maximum value of the future state s_{t+1}.

The first step of the algorithm is to initialize the values of the Q function for all the states s and actions a randomly . After that, the algorithm starts

interacting with the environment in order to learn the Q^* function. In each iteration, the update function needs an initial state s_t, the executed action a_t, the new state s_{t+1} which has been achieved, and the received reward r_{t+1}. The algorithm updates the value of $Q(s_t, a_t)$, comparing its current value with the sum of r_{t+1} and the discounted maximum Q value in s_{t+1}. The error is reduced with a learning rate and added to $Q(s_t, a_t)$. When the algorithm has converged to the optimal Q^* function, the learning process can be stopped. The parameters of the algorithm are the discount factor γ, the learning rate α and the ϵ parameter for the random actions.

Algorithm 2: Q_learning algorithm

---

    1. Initialize $Q(s, a)$ arbitrarily
    2. Repeat until Q is optimal:
      (a) $s_t \leftarrow$ the current state
      (b) choose action a_{max} that maximizes $Q(s_t, a)$ over all a
      (c) $a_t \leftarrow (\epsilon - greedy)$ action, carry out action a_{max} in the world with probability
      $(1 - \epsilon)$, otherwise apply a random action (exploration)
      (d) Observe the reward r_{t+1} and the new state s_{t+1}
      (e) $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \cdot max_{a_{max}} Q(s_{t+1}, a_{max}) - Q(s_t, a_t)]$

Another representation of Q_learning can be seen in Figure 4.5. At each iteration, the algorithm perceives the state from the environment and receives the reward. After updating the Q value, the algorithm generates the action to be undertaken. The Q function is represented as a table with a different state in each row and a different action in each column. The state space can contain different variables (n-dimensional) with different values which are finally encoded as p different states. The same occurs with the actions, different actions (m-dimensional) are encoded to a final set of q actions. As can be observed, the algorithm has to store the past state s_t and the past action a_t.

4.4.3 Eligibility Traces

The concept of Eligibility Traces is used to link Temporal Difference methods with Monte-Carlo methods. As described in Section 4.3, Monte-Carlo methods do not require the dynamics of the environment and wait until the discounted sum of future rewards, R_t, has been received. Similarly, TD methods do not need the dynamics of the environment and estimate R_t using $r_{t+1} + \gamma V_k^\pi(s_{t+1})$. This feature, called bootstrapping, allows TD algorithms to learn online. A TD algorithm which uses eligibility traces estimates R_t using the set of future rewards $\{r_{t+1}, r_{t+2}, ..., r_{t+n}\}$ and the estimated value of state s_{t+n}, as shown in next equation:
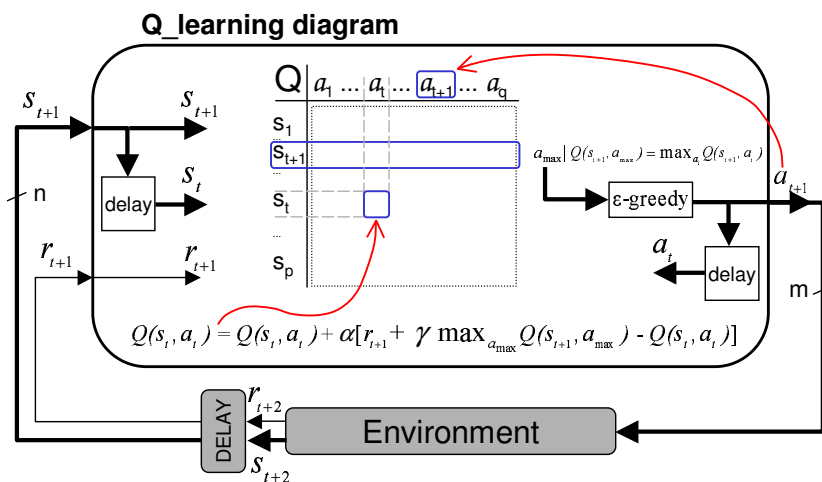
Figure 4.5: Diagram of the Q_learning algorithm.

$$R_t^n = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V_k^\pi(s_{t+n}) \qquad (4.17)$$

Therefore, for $n = 1$, a TD algorithm with eligibility traces is equivalent to the basic TD algorithm which is called *one-step TD method*. For $n = \infty$, the algorithm will be equivalent to a Monte-Carlo algorithm, in which case, the sum of rewards will be stopped at the end of the episode, since Monte-Carlo methods cannot be applied to continuing tasks.

The basic idea of eligibility traces consists of using more rewards than the immediate one to estimate the value function. However, the implementation of a TD algorithm with Equation 4.17 would imply the learning offline, since the equation cannot be computed until r_{t+n} is obtained. To avoid this disadvantage, the *eligibility trace function*, e_t, is introduced. This function contains, for each state or state/action pair, the number of visits which have recently been done. Therefore, analyzing the e_t function, the states which have recently been visited, can be found. For example, for the state-value function, $V(s)$, the eligibility trace $e_t(s)$ is defined as:

$$e_t(s) = \begin{cases} \gamma \lambda e_{t-1}(s) & \text{if } s \neq s_t; \\ \gamma \lambda e_{t-1}(s) + 1 & \text{if } s = s_t; \end{cases} \qquad (4.18)$$

The eligibility trace of state s will increase by 1 each time the state is visited, otherwise it will slowly decrease. To control the rate of decrease, a new parameter, λ, is introduced. The range of this parameter is $0 \leq \lambda \leq 1$. For $\lambda = 0$, a pure one-step TD algorithm will be obtained, and for $\lambda = 1$, the

algorithm will be equivalent to a Monte-Carlo method, in which the eligibility trace will only decrease due to the discount factor. Intermediate values will affect only a set of future time-steps, which is usually the desired effect.

After defining the eligibility trace function, e_t, the update rule for $V^\pi(s)$, considering eligibility traces, must be redefined. The update rule for the one-step TD method was presented in Equation 4.15. In that case, only the value of the current state was updated at each iteration, whereas when using an eligibility trace function, all the states which have recently been visited will be updated. This process is accomplished in two different phases. In the first phase, the one-step error δ is computed and the visits of the current state are increased by 1. Equations 4.19 and 4.20 show these two operations.

$$\delta = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{4.19}$$

$$e(s_t) \leftarrow e(s_t) + 1 \tag{4.20}$$

After the first phase, the state-values of all the states are updated and the eligibility traces are decreased. Therefore, for each state s, these two equations are applied:

$$V(s) \leftarrow V(s) + \alpha \, \delta \, e(s) \tag{4.21}$$

$$e(s) \leftarrow \gamma \, \lambda \, e(s) \tag{4.22}$$

The methodology just described for estimating the state-value function $V^\pi(s)$ under eligibility traces, is called the $TD(\lambda)$ algorithm [Sutton, 1988] . Similarly, an eligibility trace function for each state/action pair, $e(s, a)$, can be applied to estimate the action-value function $Q^\pi(s, a)$.

The use of eligibility traces has several advantages and disadvantages. The main disadvantage is that the computational cost is much higher than the one-step TD algorithms, as all the states or state/action pairs have to be updated at each time-step. However, the main advantage of using eligibility traces is faster learning, particularly when rewards are delayed. When a RLP has a reinforcement function which, for example, only gives non-zero values when the goal is achieved, it is said that rewards are delayed. In these systems, the non-zero rewards will only be given at some states. Using one-step TD algorithms, the propagation of these values to preceding states will take several iterations. Eligibility traces accelerate the propagation of these values, achieving an optimal policy in fewer iterations. A second advantage of eligibility traces resides in their ability to learn in partially non-Markov states. The main reason for this feature is the fact that the learning is not completely based on bootstrapping, but also on the observation of a set of future rewards.

Eligibility traces can be applied in almost all one-step TD algorithms. The Q_learning algorithm was modified to the Q(λ) algorithm [Watkins, 1989, Peng and Williams, 1994], and also Actor-Critic methods were implemented using eligibility traces [Barto et al., 1983, Sutton, 1984]. Eligibility traces is not a research topic, but an important aspect to be considered in RL. Although the theory of eligibility traces has been sufficiently developed, there are not many applications using it in real systems.

4.5   Issues in RL

After the description of the Reinforcement Learning Problem and its solution using two classic Temporal Difference algorithms, this section presents some advanced issues. These new aspects reflect the main research topics currently being studied by the RL community. The purpose of this section is to describe the basic concepts needed to understand each topic. The next section will focus on one of them; the generalization problem, which is the research topic treated in this thesis.

The Generalization Problem. As described in Section 4.2, Reinforcement Learning is usually formulated using Finite Markov Decision Processes (FMDP). This formulation implies a discrete representation of the state and action spaces. However, in some tasks the states and/or the actions are continuous variables. A first solution can be to maintain the same RL algorithms and to discretize the continuous variables. If a coarse discretization is applied, the number of states and actions will not be too high and the algorithms will be able to learn. However, in many applications the discretization must be fine in order to assure a good performance. In these cases, the number of states will grow exponentially, making the use of RL impractical. The reason is the high number of iterations necessary to update all the states or state/action pairs until an optimal policy is obtained. This problem is known as the *curse of dimensionality*. To make RL feasible, *generalization* must be applied among the state and actions. To solve the generalization problem, RL algorithms modify the classical algorithms replacing the discrete spaces with a *function approximator*. The generalization problem is a very common problem in robotics and in control tasks in general. However, there are plenty of successful approaches in which function approximation techniques have been applied. Section 4.6 overviews the basic techniques and their application to robotics.

Non-Markov Environments. When RL is applied to real systems, it is

common to have difficulties in observing the state. As explained in Section 4.2, the Markov property implies the observability of the complete state of the environment. Real systems are observed with sensors which always have noise. Also, it is sometimes not possible to measure some magnitudes, in which case the state is *hidden*. If this corrupted data is used as the state of the environment, TD algorithms cannot assure the learning of an optimal policy. This kind of environment is called a *Partially Observable Markov Decision Process* (POMDP). A solution to POMDP is to learn a model of the environment which includes the non-observed states. This model estimates the Markov states using the past estimated states, the last action and the corrupted observation of the environment. The model gives a probability for each state to be effectively the current state, and a specific RL algorithm uses this information to solve the RLP. Some approaches can be found which use this *state-estimator* methodology [Cassandra et al., 1994, Whitehead and Lin, 1995]. However, for large state spaces, this method becomes computationally unfeasible and alternative solutions have been proposed [Singh et al., 1994a]. POMDP is an active research topic in RL with a high relevance to robotics, although few real examples can be found. In [Bakker et al., 2002] a simulated application with a mobile robot is presented.

Other kind of non-Markovian environments to be found in RL are Semi-Markov Decision Processes (SMDP). The difference between FMDP and SMDP resides in the amount of time which passes between the algorithm iterations. In FMDP, a constant time-step is assumed between two sequential iterations. However, in SMDP the amount of time between two iterations is a random variable. In the simplest case, this amount of time is a multiple of an underlying time-step. SMDP are usually used in hierarchical RL, which will be described below. For more detailed information on SMDP, refer to [Sutton et al., 1999].

Policy Methods. This chapter has described RL algorithms as any algorithm able to solve the RLP. After formally defining an RLP with FMDP, Temporal Difference methods have been presented as the most suitable methods to solve the problem. The technique consists of estimating an optimal value function (state or action function) and extracting an optimal policy from it. There is another technique to solve the RLP. Instead of first searching the optimal value function, the optimal policy is directly searched. RL algorithms using this technique are called *Policy* methods. Policy methods use a policy function $\pi(s, a, \theta)$, which gives, for each state s and each action a, a probability value.

This function is implemented with a function approximator which has a set of parameters θ. Policy methods use a function ρ to evaluate the performance of the current policy. According to the policy gradient with respect to ρ function, the parameters of the policy are updated. A first policy algorithm was the REINFORCE algorithm [Williams, 1992]. Recently, several works have demonstrated the convergence of policy methods in optimal policies [Jaakkola et al., 1995, Sutton et al., 2000, Baxter and Barlett, 2000]. There are some advantages of policy methods with respect to value-algorithms. Policy methods already use a function approximator to implement the policy function, which solves the generalization problem. They also face POMDP environments satisfactorily, since the observation of a corrupted state damages the value function to a greater degree than the policy function. The main disadvantage of policy methods is the increase of the convergence time with respect to value methods. There are some successful applications of policy methods in robotics. In [Bagnell and Schneider, 2001] an autonomous helicopter was controlled and in [Rosenstein and Barto, 2001] a simulated robotic arm learnt to lift a weight.

Hierarchical Learning. Hierarchical RL appeared as a result of the generalization problem. Instead of solving the problem with a function approximator, hierarchical learning is based on temporal abstractions. The basic assumption is to break down the whole task into subtasks. Each subtask has its own subgoal and can be solved using a policy. While a subtask is being executed, another subtask can be called, as if it was an action. The higher subtask will wait until the finalization of the lower subtask and then it will continue learning as if only one iteration had happened. For this reason, hierarchical methods use SMDP, in which the elapsed time between two sequential iterations can change. Different task-division architectures can be found in hierarchical learning [Sutton et al., 1999, Parr, 1998, Dietterich, 2000]. A common feature in these algorithms is that the breaking down of the whole task in subtasks must be known, and the algorithm learns the policy of each subtask. This fact restricts the use of these algorithms in problems where task-division is clear, as for example, in problems which can be divided in sequential phases. However, in control problems, where this division is not clear, these algorithms cannot be used. Hierarchical learning is also an active research topic with a high applicability in robotics and in large generalization problems. In [Takahashi and Asada, 2000] hierarchical learning was applied to acquire a vision-guided robot behavior, in [Kawano and Ura, 2002] the

application was the motion planning of an autonomous underwater vehicle, and in [Ryan and Reid, 2000] the problem was to control an aircraft in a flight simulator. Finally, hierarchical learning for POMDP in a robot navigation task was treated in [Theocharous, 2002]. A recent review of hierarchical RL can be found in [Barto and Mahadevan, 2003].

4.6   Generalization methods

This section overviews the main methodologies used to solve the generalization problem. As defined above, the generalization problem appears when the environment has continuous states and/or actions. In order to adapt these continuous variables to the finite TD algorithms, the first solution is to discretize the continuous spaces to finite spaces. However, if a fine discretization must be applied, the high number of states or state/action pairs makes RL impractical for real applications. This section shows the most common function approximation techniques combined with RL algorithms to deal with the generalization problem.

The reason why there are several techniques to solve the generalization problem is because any one of them offers a perfect solution. Some techniques have a higher generalization capability while others are computationally faster. However, the most important feature is their capability to converge into an optimal policy. Convergence proofs have only been obtained for algorithms which use a linear function approximator [Sutton, 1988, Dayan, 1992, Tsitsiklis and Roy, 1997, Singh et al., 1994b], that is, a function which is linear with respect to a set of parameters. In addition, in order to maintain the stability, the learning must be done on-policy. This means that the Q-learning algorithm, for example, cannot profit from this convergence proof [Precup et al., 2001]. TD algorithms estimate the value function based on the immediate rewards and on the same value function (bootstrapping feature). When using a function approximator with TD algorithms, the value function will always be an approximation of the discrete function. It has been verified that off-policy learning algorithms with linear function approximators can cause divergence. Some classical examples of this divergence can be found in [Baird, 1995, Tsitsiklis and Roy, 1996]. Several methods were developed to deal with this divergence problem; for example, residual algorithms [Baird, 1995], averagers [Gordon, 1999] and interpolative representations [Tsitsiklis and Roy, 1996]. These methods use special update rules and/or function approximators to ensure the convergence. However, their use is very limited in practice since their convergence is much slower and they have a lower generalization capability.

Despite convergence proofs, there are many successful examples including linear and non-linear approximators with on-policy and off-policy algorithms. Some references are [Lin, 1992, Zhang and Dietterich, 1995, Sutton, 1996] [Crites and Barto, 1996]. The most important breakthrough was the use of a Neural Network to learn the game of backgammon [Tesauro, 1992]. Using this non-linear function approximator the algorithm was able to play at the same level as the best human players in the world. These successful applications have motivated the use of a diverse set of function approximation techniques for real systems. Although it is not intended as a survey, the most common methodologies and their use in robotics will be next described.

4.6.1   Decision trees

One of the most intuitive approaches to solving the generalization problem is to discretize the state or state/action space with a different resolution. This variable resolution substantially reduces the number of finite states or state/action pairs. *Decision trees* allow the space to be divided with varying levels of resolution. Usually, only the state space is represented. In a decision tree, see Figure 4.6, the *root* of the tree represents the entire space. Each branch of the tree divides the space in two space zones. After several branches a terminal space zone is found which is called the *leaf*. The leaves contain the values of the approximated function for the space zone they represent. RL algorithms use their update rules to modify the value of the leaf containing the evaluated state s_t or state/action pair (s_t, a_t), depending if the approximated function is the state-value function, $V(s)$, or the action-value function, $Q(s, a)$, respectively.

Algorithms using decision trees refine the space resolution by splitting the leaves. This refinement is not performed in the whole space but only in the zones where a higher resolution is required. There are some criteria which determine this refinement. If two adjacent leaves propose different greedy actions, the leaves are split. If the values (state-value or action-value) of two adjacent leaves are substantially different, even if the greedy action is the same, the leaves are also split.

Several proposals using decision trees to solve the generalization problem can be found. The *G-Learning* algorithm [Chapman and Kaelbling, 1991] uses a decision tree to represent the Q function over a discrete space. That is, the state is compounded of a set of binary variables. According to the binary values, a leaf is reached. This leaf contains the values for a discrete set of actions which represent the Q values. Also, the leaf contains the set of state variables which have not been used to achieve the leaf. If more refinement is required, the leaf will be split according to one of the non-used binary vari-
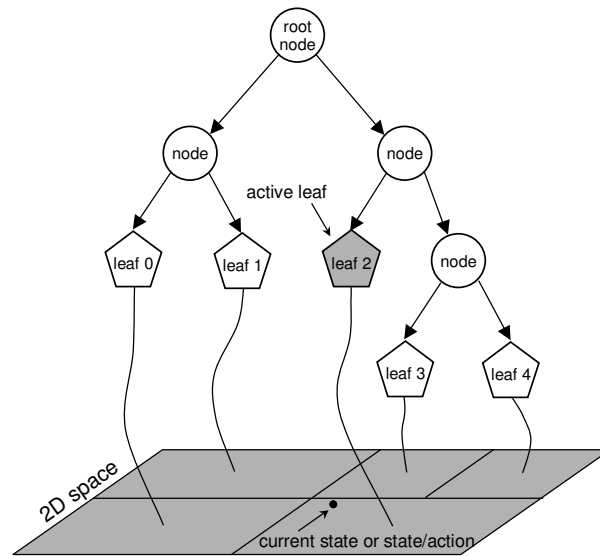
Figure 4.6: Approximating the value space with a Decision Tree.

ables. Another approach, the *Continuous U-tree* [Uther and Veloso, 1998], applies ideas similar to G-learning but with a continuous representation of the state. This approach was applied to a simulated robot. The *Variable Resolution Dynamic Programming* [Moore, 1991] is another successful approach to generalization with decision trees. However, its main drawback is the requirement of the environment model.

Other approaches use a *multigrid* which is a set of layers with different resolutions. Each layer has a uniform resolution for the entire state-space. Top layers use a coarse resolution and low layers a fine resolution. Multigrid algorithms update the values for all the layers. Evidently, the percentage of updated cells in low layers will be less than in top layers. Therefore, multigrid algorithms will decide at which layer the confidence of the value function is higher and will use this value in the RL update rule. An example of a multigrid algorithm can be found in [Vollbrecht, 1999]. The drawback of most multigrid methods is the assumption that the model of the environment is known, while this cannot be assumed in a robotics domain.

Decision trees highly improve the generalization problem with respect to classical RL algorithms which use a uniform discretization. In addition, the convergence of the algorithms is sometimes proved and the generalization capability has been shown to be very high [Munos and Moore, 2002]. However, the learning convergence has not demonstrated to be fast enough for its application in real systems where the convergence time must be short.
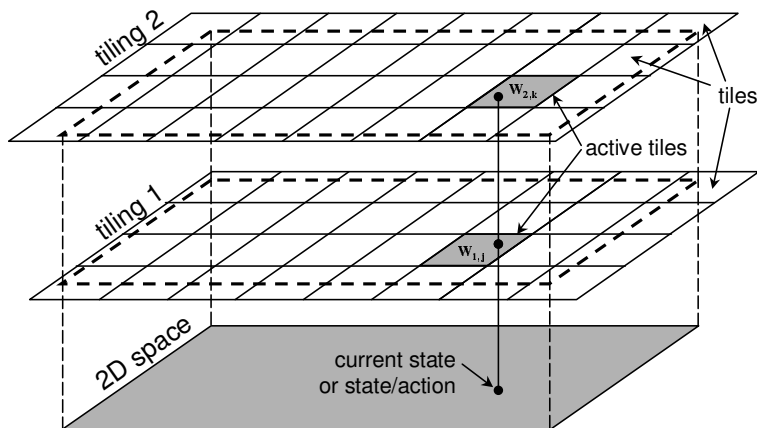
Figure 4.7: Approximating the value space with the CMAC function approximator.

Most of the recent proposals show their results only in simulated tasks [Reynolds, 2002, Munos and Moore, 2002]. Moreover, decision trees always use a finite set of actions and, therefore, the generalization is only carried out in the state space. Finally, in order to select the greedy actions, the whole set of actions must be tested, which slows the learning process when several continuous actions are present.

4.6.2   CMAC

One of the most popular techniques to generalize in RL is the *Cerebellar Model Articulation Controller* or *CMAC* [Albus, 1971, Albus, 1981]. CMAC is a simple linear function approximator based on a set of features or *tiles*. A tile is a representation of a small zone of the whole value space (state space or state/action space). Each tile i has a weight, w_i, associated with it. Tiles can be superimposed, that is, a position in the space can be comprised of different tiles. All tiles comprising the current input value are considered to be *active* tiles. CMAC uses a very simple procedure, the value of an input state or state/action pair will be the sum of all w_i belonging to an active tile. RL algorithms use their update rules to modify the weights, w_i, which influenced the evaluation of the state-value function, $V(s)$, or the action-value function, $Q(s, a)$.

Usually, tiles in CMAC are uniformly distributed, which makes the computation of the algorithm faster. In this case, tiles are associated to *tilings*. Each tiling comprises the whole space and has a uniform set of tiles, see Figure 4.7. Tiles from the same tiling are not superimposed, which means that

only one tile is active in each tiling. Therefore, the number of active tiles will be equal to the number of tilings. The evaluation of the action-value function with CMAC can be performed with the next equation:

$$Q(s,a) = \sum_{i,j} w_{i,j} F(s,a,i,j) \qquad (4.23)$$

in which, w_{ij} is the weight of tile j from tiling i, and $F(s,a,i,j)$ is a binary activation function. This function is equal to 1 if tile j from tiling i is activated for state s and action a, otherwise it is equal to 0. Similarly, the state-value function $V(s)$ can be represented with CMAC.

The generalization capability of CMAC depends on the number of tilings and the number of tiles within each tiling. The higher the number of tiles the better the resolution and the higher the number of tilings the better the generalization. Since CMAC is a linear approximation function, its convergence can be guaranteed when an on-policy algorithm is used, as stated in Section 4.6. However, many examples of both on-policy [Sutton, 1996, Santamaria et al., 1998] and also off-policy methods [Saito and Fukuda, 1994, Watkins, 1989] demonstrate the feasibility of CMAC. In most of these examples, CMAC approximates the action-value function $Q(s,a)$. The disadvantages of CMAC are similar to those of decision trees. Although a high generalization capability has been proved, the convergence time is still too long to be applied in real systems. Also, the search of the greedy action requires an exhaustive search and is not scalable when several continuous actions are present. This makes CMAC difficult to apply in robotics.

A similar CMAC technique is the use of *Radial Basis Functions* (RBF) [Powell, 1987, Poggio and Girosi, 1990]. The idea is to substitute the binary activation function of each tile by a continuous function between 0 and 1. Radial Basis functions usually have a Gaussian response and are defined by their center position and width. A network of RBFs distributed over the value space constitutes a linear function approximator. A more interesting use of RBFs is achieved when the center positions and widths can be changed according to the learning process. In this case, the method is nonlinear and is able to generalize with more precision than CMAC, as stated in [Kretchmar and Anderson, 1997]. The main disadvantage of RBFs is its higher computational cost. However, some applications using RBFs in robotics can be found [Santos, 1999, Kondo and Ito, 2002]

4.6.3 Memory-based methods

Another methodology to approximate the value functions is the use of *Memory-based* methods, also called *Instance-based* methods. In a memory-based func-
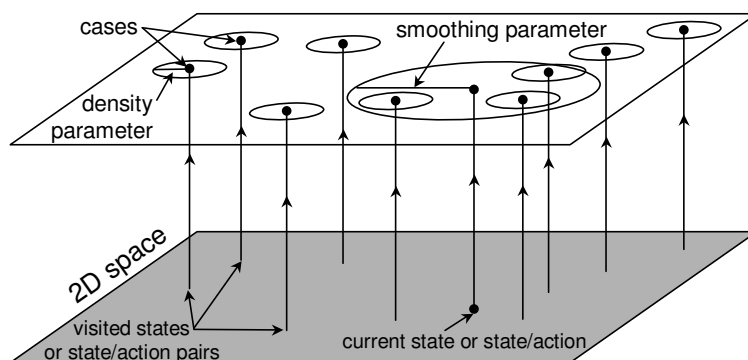
Figure 4.8: Approximating the value space with a Memory-Based function approximator.

tion approximator, each element of the memory represents a visited state or state/action pair, which is also called a *case*. Elements contain the value of the approximated function for the cases they represent. The memory is initialized with zero elements and is dynamically filled according to the visited cases. To obtain the value for a particular case, which is not contained in the memory, a set of neighbor cases are used. There are different techniques to approximate this new value. The simplest technique is to use the value of the *nearest neighbor* case. A more elaborate technique is to average the values of the cases which are "close", called the neighbor cases. This average is done according to the inverse of the distance between the new case and the neighbor case. This means that the closest cases will affect more than the ones farther away. This technique is called *weighted average*. Finally, another technique is to fit a surface using all neighbor cases. Once the surface is computed, the value for the new case is calculated. This technique is called *locally weighted regression*.

Memory-based algorithms use some parameters or thresholds to define how the memory is filled and how the approximation of new cases is performed. The *density* parameter defines the maximum number of cases which can be stored in the memory, see Figure 4.8. Every time a new case is presented, it will be stored if the euclidian distance to its nearest neighbor is larger than the density threshold. A small density threshold produces a function approximator with high resolution but with large amounts of cases in memory. The *smoothing* threshold defines the set of neighbors to be used to approximate the value of a new case. When using a weighted average or a locally weighted regression technique, cases will be considered as neighbors if the distance to the new case is smaller than the smoothing threshold. A

large smoothing threshold with respect to the density threshold, produces a high number of neighbor cases which can slow down the approximation of new cases. These two parameters determine the performance of memory-based approximators. In addition, several weighting functions, regression approaches, distance functions, and other aspects have to be taken into account. For a complete survey on memory-based function approximators refer to [Atkenson et al., 1997].

RL algorithms use their update rules to modify the values of the neighbor cases which influenced in the approximation of the state-value function, $V(s)$, or the action-value function, $Q(s, a)$. Most of the approaches use the action-value function. The convergence of memory-based approaches is not guaranteed, although the generalization capability is very high. The main disadvantage of memory-based approaches is the high computation required. Each time a new estimation of the value function is required, the whole approximation process, with all the neighbor cases must first be performed. Memory-based methods are also called *lazy learning*, since all the computation is not performed until a query has to be answered. In addition, as it happened with the previous methodologies, in order to find a greedy action, a finite set of actions must be evaluated, which again, implies an extra computational cost. Despite these inconveniences, Memory-based function approximation has been widely applied in RL. Some examples of its performance on simulated tasks can be found in [Peng, 1995, McCallum, 1995, Ormoneit and Sen, 2002]. Also, some approaches have demonstrated its application on real robots [Smart, 2002, Millan et al., 2002]. Finally, the higher suitability of a memory-based function approximator compared with CMAC is stated in [Santamaria et al., 1998].

4.6.4 Artificial Neural Networks

The last described function approximators are *Artificial Neural Networks* (NN). An NN is a parameterized function which can approximate the value-function or the action-function. NN are basically compounded of a set of *neurons* which become activated depending on some *inputs* values. The activation level of a neuron generates an *output* according to an activation function. Neuron outputs can be used as the inputs of other neurons. By combining a set of neurons, and using some non-linear activation functions, an NN is able to approximate any non-linear function. The overall NN also has a set of inputs and a set of outputs. The inputs are connected to the first layer of neurons and the outputs of the NN correspond to the outputs of the last layer of neurons. A formal definition of NN can be found in Section 5.3, and for an overview of NN refer to [Haykin, 1999].
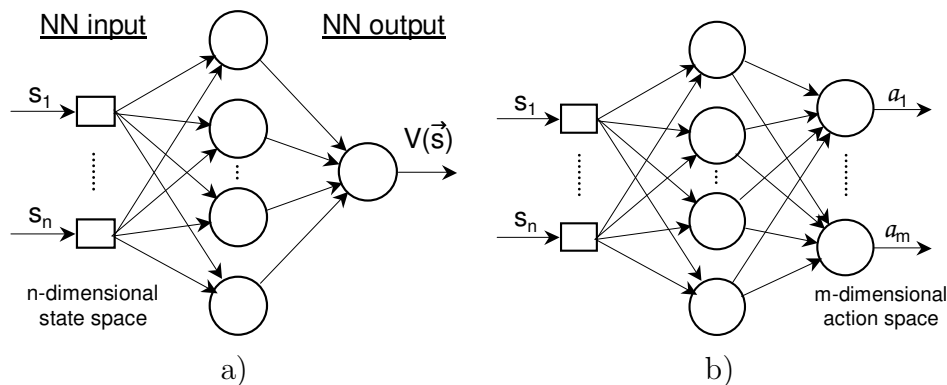
Figure 4.9: a) Approximating the state-value function with a Neural Net-work. The state is composed of n variables. b) Implementation of the *actor* function with a NN. The action is composed of m variables.

NN can be used to approximate the state-value function or the action-value function. In the case of the state-value function, $V(s)$, the network has as inputs the number of continuous variables which compose the state and as output the value of the state. In Figure 4.9a, the general implementation of $V(s)$ is shown. As described in Section 4.4.1, RL algorithms which work with the value-function (the critic) usually require an actor containing the optimal policy. According to the error committed by the critic, the policy is updated. An NN can also be used to implement the actor. In this case, the input of the NN is also the state, and the outputs are the number of continuous variables which compose the action, Figure 4.9b.

One of the most important successes of Reinforcement Learning was achieved by Tesauro, who used an NN to learn the state-value of the backgammon game [Tesauro, 1992]. In this case, the actor was not implemented with an NN since the rules of the game, which relate a state/action pair with the next state, were known. The greedy action was extracted analyzing the values of the next achievable states. The implementation of the actor-critic method with an NN in a robotics application can be found in [Lin, 1993], and also in [Gachet et al., 1994]. In the latter case, a similar actor-critic schema was used to coordinate a set of robot behaviors.

A second use of NN is the approximation of the action-value function, $Q(s,a)$. One of the first implementations consisted of approximating only the state space and using one NN for each discrete action [Lin, 1993]. The inputs of the NN were the variables composing the state, and the output was the Q value for the current state and for the action which the NN represented. This implementation, known as the $QCON$ architecture, was also
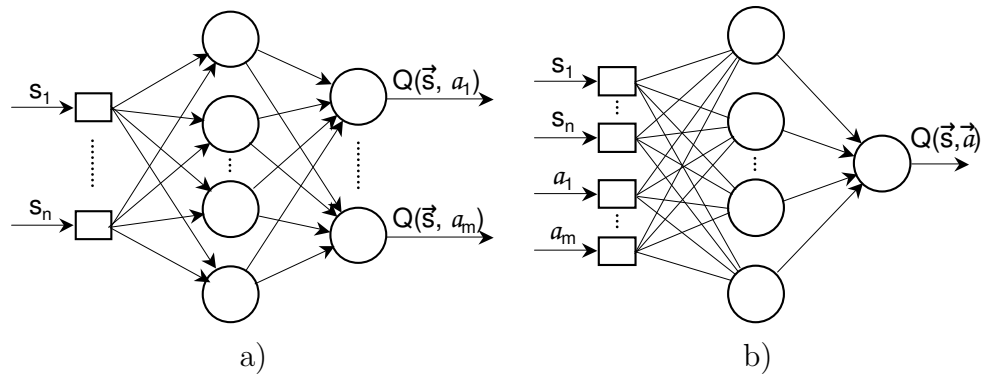
Figure 4.10: Implementation of the action-value function, $Q(s, a)$ with a NN. In (a) the *QCON* architecture is shown, and in (b) the *direct Q_learning* architecture is shown.

compacted in one NN which had as many outputs as the number of discrete actions, see Figure 4.10a. The reason why the Q_function was not directly implemented was to reduce the space in which the NN had to generalize and, therefore, to simplify the approximation. However, approaches using a direct approximation of the Q_function with an NN can also be found. In this case, the inputs of the NN are the continuous variables of the state and the action vectors, and the output is the $Q(s, a)$ value, see Figure 4.10b. This NN implementation combined with the Q_learning algorithm is also known as *direct Q_learning* [Baird, 1995]. An application of this in a robotics task can be found in [Hagen and Krose, 2000].

Despite the implementation aspects, an NN requires a learning algorithm to adapt the approximated outputs to the desired ones. This adaptation or learning is accomplished by modifying some weights which multiply the inputs of each neuron. The update rules of RL algorithms are used to find the error of the NN. After that, an NN learning algorithm distributes this error to all the neurons and modifies the NN weights. The most popular algorithm is the *back-propagation* algorithm, refer to Section 5.3.3. The combination of a multi-layer NN and back-propagation results in a fast and powerful algorithm which allows the approximation of any non-linear function. Other NN schemas and learning algorithms can also be used. In [Touzet, 1997] a neural implementation with a *Self-Organizing Map* for robotics tasks is proposed. Moreover, a network of Radial Basis Functions is also considered as an NN, which was briefly touched on in the previous section.

The convergence of an RL algorithm using a multi-layer Neural Network cannot be guaranteed due to its non-linear approximation capability,

refer to Section 4.6. In addition, multi-layer NN suffer from an important problem which is known as the *interference* problem [Weaver et al., 1998, Atkenson et al., 1997]. This problem is caused by the impossibility of generalizing in a local zone of the entire space. The previously analyzed methodologies, decision trees, CMAC and memory-based approaches update only the values of the active leaves, tiles or cases. These methods work locally. However, when an NN learning algorithm updates the weights to change the value of one state, the modification affects the entire space. It is said that, "interference occurs when learning in one area of the input space causes unlearning in another area" [Weaver et al., 1998]. In practice, a first solution to avoid interference is to use the NN in a local area. This is the solution proposed in the QCON architecture described above. Also, different NN schema can be used. Radial Basis Functions and Self-Organizing Maps are local NN, however, they have a smaller generalization capability than multi-layer NN. A second solution is to learn from a well-distributed data set, which assures the generalization over the entire space. This solution was adopted by Tesauro in the backgammon application. However, in a robotics task, this uniform data distribution cannot be obtained since the updated space area depends on the robot trajectory.

Finally, NN suffer also from the computational cost of finding the greedy actions. As happened with the three analyzed methodologies, the search of a greedy action implies the evaluation of a set of discrete actions. In an NN, this process requires less computation than memory-based approaches, but more than in decision trees and CMAC. There is an approach called *Wire Fitting* [Baird and Klopf, 1993] which allows a fast search of greedy actions. An implementation of Wire Fitting and NN was proposed in [Gaskett, 2002] to be used in robotics.

Neural Networks is one of the most used methodologies to approximate the value functions in robotics [Gross et al., 1998, Hagen and Krose, 2000, Maire, 2000, Gaskett, 2002, Buck et al., 2002]. In order to deal with the interference problem, RL algorithms use NN locally, or uniformly update the space. NN [Zhang and Dietterich, 1995, Bertsekas and Tsitsiklis, 1996, Tesauro, 1992] has demonstrated a higher generalization capability than the other methodologies. The approach presented in this thesis also uses an NN to generalize the Q_function. The Neural Q_learning algorithm is complemented with a database which contains a representative set of samples to avoid the interference problem. The next chapter describes the proposed RL algorithm.

Chapter 5

Semi-Online Neural-Q_learning

This chapter contains the main research contribution of this thesis. It proposes the Semi-Online Neural-Q_learning algorithm (SONQL), an RL algorithm designed to learn with continuous states and actions. The purpose of the SONQL algorithm is to learn the state/action mapping of a reactive robot behavior. The chapter concentrates on theoretically analyzing the points taken into account in the design of this approach. The main features of the algorithm are the use of a Neural Network and a database of learning samples which stabilize and accelerate the learning process. The implementation of the algorithm in a reactive behavior is described. Results of the SONQL algorithm will be shown in Chapter 7.

5.1 Reinforcement Learning based behaviors

Reinforcement Learning (RL) is a very suitable technique to learn in unknown environments. Unlike supervised learning methods, RL does not need any database of examples. Instead, it learns from interaction with the environment and according to a scalar value. As has been described in Chapter 4, this scalar value or *reward*, evaluates the environment state and the last taken action with reference to a given task. The final goal of RL is to find an *optimal state/action mapping* which maximizes the sum of future rewards whatever the initial state is. The learning of this optimal mapping or *policy* is also known as the *Reinforcement Learning Problem* (RLP).

The features of RL make this learning theory useful for robotics. There are parts of a robot control system which cannot be implemented without experiments. For example, when implementing a reactive robot behavior, the main strategies can be designed without any real test. However, for the final tuning of the behavior, there will always be parameters which have to be

set with real experiments. A dynamics model of the robot and environment could avoid this phase, but it is usually difficult to achieve this model with reliability. RL offers the possibility of learning the behavior in real-time and avoid the tuning of the behaviors with experiments. RL automatically interacts with the environment and finds the best mapping for the proposed task, which in this example would be the robot behavior. The only necessary information which has to be set is the *reinforcement function* which gives the rewards according to the current state and the past action. It can be said that by using RL the robot designer reduces the effort required to implement the whole behavior, to the effort of designing the reinforcement function. This is a great improvement since the reinforcement function is much simpler and does not contain any dynamics. There is another advantage in that an RL algorithm can be continuously learning and, therefore, the state/action mapping will always correspond to the current environment. This is an important feature in changing environments.

RL theory is usually based on *Finite Markov Decision Processes* (FMDP). The dynamics of the environment is formulated as a FMDP and the RL algorithms use the properties of these systems to find a solution to the RLP. *Temporal Difference* (TD) techniques are able to solve the RLP incrementally and without knowing the transition probabilities between the states of the FMDP. In a robotics context, this means that the dynamics existing between the robot and the environment do not have to be known. As far as incremental learning is concerned, TD techniques are able to learn each time a new state is achieved. This property allows the learning to be performed *online*, which in a real system context, like a robot, can be translated to a real-time execution of the learning process. The term "online" is here understood as the property of learning with the data that is currently extracted from the environment and not with historical data.

Of all TD techniques, the best known and most used technique is the *Q learning* algorithm proposed by Watkins in 1992 [Watkins and Dayan, 1992]. The advantages of this algorithm are its simplicity and the fact that it is an off-policy method. For these reasons, which will be detailed in the next section, Q learning has been applied to a huge number of applications. The RL algorithm proposed in this dissertation was also based on Q learning.

The main problem of RL when applied to a real system is the *generalization* problem, treated in Section 4.5. In a real system, the variables (states or actions) are usually continuous. However, RL theory is based on FMDP, which uses discrete variables. Classic RL algorithms must be modified to allow continuous states or actions, see Section 4.6. Another important problem of RL when applied to real systems is the correct observation of the environment state. In a robotic system, it is usual to measure signals with noise or

delays. If these signals are related to the state of the environment the learning process will be damaged. In these cases, it would be better to consider the environment as a *Partially Observable MDP*, refer also to Section 4.5.

The approach presented in this thesis attempts to solve only the generalization problem. As commented on above, the approach is based on the Q_learning algorithm and includes a Neural Network (NN) to generalize. NNs are able to approximate very complex value functions, but they are affected by the *interference* problem, as described in Section 4.6.4. To overcome this problem, the presented approach uses a database of *learning samples* which contains a representative set of visited states/action pairs which stabilizes and also accelerates the learning process. The approach has been named *Semi-Online Neural-Q_learning* (SONQL). The next subsections will detail each part of the SONQL algorithm and the phases to be found on its execution.

The combination of Reinforcement Learning with a behavior-based system has already been used in many approaches. In some cases, the RL algorithm was used to adapt the coordination system [Maes and Brooks, 1990, Gachet et al., 1994, Kalmar et al., 1997, Martinson et al., 2002]. Moreover, some researches have used RL to learn the internal structure of the behaviors [Ryan and Pendrith, 1998, Mahadevan and Connell, 1992, Touzet, 1997, Takahashi and Asada, 2000, Shackleton and Gini, 1997] by mapping the perceived states to control actions. The work presented by Mahadevan demonstrated that the breaking down of the robot control policy in a set of behaviors simplified and increased the learning speed. In this thesis, the SONQL algorithm was designed to learn the internal mapping of a reactive behavior. As stated in Chapter 3, the coordinator must be simple and robust. These features cannot be achieved with an RL algorithm since, if the data becomes corrupted, the optimal policy can be unlearnt. Instead, RL can satisfactorily learn a behavior mapping, which simplifies the implementation and tuning of the algorithm. The chapter concludes with the implementation of the SONQL algorithm in a reactive robot behavior.

5.2   Q_learning in robotics

The theoretical aspects of the Q_learning [Watkins and Dayan, 1992] algorithm have been presented in Section 4.4.2. This section analyzes the application of the algorithm in a real system such as a robot. Q_learning is a Temporal Difference algorithm and, like all TD algorithms, the dynamics of the environment do not have to be known. Another important feature of TD algorithms is that the learning process can be performed online. How-

ever, the main advantage of Q_learning with respect to other TD algorithms is that it is an *off-policy* algorithm, which means that in order to converge to an optimal state/action mapping, any policy can be followed. The only condition is that all state/action pairs must be regularly visited.

The *policy* in an RL algorithm, indicates the action which has to be executed depending on the current state. A *greedy* policy chooses the best action according to the current state/action mapping; that is, the action which will maximize the sum of future rewards. A *random* policy generates an aleatory action independently of the state. An ϵ-*greedy* policy chooses the greedy action with probability $(1 - \epsilon)$, otherwise it generates a random action. The importance of the policy relapses in the *explotation/exploration* dilemma.

Q_learning can theoretically use any policy to converge to the optimal state/action mapping. The most common policy is the ϵ-greedy policy which uses random actions to explore and greedy actions to exploit. The off-policy feature is a very important feature in a robotic domain, since, on occasion, the actions proposed by the learning algorithm can not be carried out. For example, if the algorithm proposes an action which would cause a collision, another behavior with a higher priority will prevent it with the generation of another action. In this case, Q_learning will continue learning using the action which has actually been executed.

Q_learning uses the *action-value* function, Q, in its algorithm. The $Q(s, a)$ function contains the discounted sum of future rewards which will be obtained from the current state s, executing action a and following the greedy policy afterwards. The advantage of using the action-value function resides in the facility of extracting the greedy action from it. For the current state s, the greedy action a_{max} will be the one which maximizes the $Q(s, a)$ values over all the actions a. Consequently, when the Q_learning algorithm converges to the optimal action-value function Q^*, the optimal action will be extracted in the same way. The simplicity of Q_learning is another important advantage in its implementation on a complex system such as a robot.

5.3   Generalization with Neural Networks

When working with continuous states and actions, as is costumary in robotics, the continuous values have to be discretized in a finite set of values. If an accurate control is desired, a small resolution will be used and, therefore, the number of discrete states will be very large. Consequently, the Q function table will also become very large and the Q-learning algorithm will require a long learning time to update all the Q values. This fact makes the im-

plementation of the algorithm in a real-time control architecture impractical and is known as the *generalization* problem. There are several techniques to combat this problem, as overviewed in Section 4.6.

In this thesis, a Neural Network (NN) has been used to solve the generalization problem. The main reason for using an NN was for its excellent ability to approximate any nonlinear function, in comparison with the other function approximators. Also, an NN is easy to compute and the required number of parameters or values is very small. The strategy consists of using the same Q_learning algorithm, but with an NN which approximates the tabular Q function. The number of parameters required by the NN does not depend on the resolution desired for the continuous states and actions. It will depend only on the complexity of the Q function to be approximated. As stated in Section 4.6.4, when generalizing with an NN, the interference problem destabilizes the learning process. This problem was taken into account in the SONQL algorithm and will be treated in Section 5.4.

5.3.1 Neural Networks overview

A Neural Network is a function able to approximate a mathematical function which has a set of inputs and outputs. The input and output variables are real numbers and the approximated functions can be non-linear, according to the features of the NN. Artificial Neural-Networks were inspired by the real neurons found in the human brain, although a simpler model of them is used. The basic theory of NN was widely studied during 1980s and there still are many active research topics. For an overview of NN, refer to [Haykin, 1999].

One model frequently used in the implementation of an artificial neuron is depicted in Figure 5.1. A neuron j located in layer l, has a set of inputs $\{y_1^{l-1}, y_2^{l-1}, ..., y_p^{l-1}\}$ and one output y_j^l. The value of this output depends on these inputs, on a set of *weights* $\{w_{j1}^l, w_{j2}^l, ..., w_{jp}^l\}$ and on an *activation function* $\varphi^{(l)}$. In the first computation, the induced *local field* v_j^l of the neuron j, is calculated by adding the products of each input y_i^{l-1} by its corresponding weight w_{ji}^l. An extra input y_0^{l-1} is added to v_j^l. This input is called the *bias* term and has a constant value equal to 1. By adjusting the weight w_{j0}^l, the neuron can be activated even if the inputs are equal to 0. The local field v_j^l is then used to calculate the output of the neuron $y_j^l = \varphi^{(l)}(v_j^l)$. The activation function has a very important role in learning efficiency and capability.

As mentioned above, neurons are grouped in different *layers*. The first layer uses as neuron inputs $\{y_0^0, y_1^0, ..., y_{in}^0\}$ the input variables of the NN. This set of inputs is also called the *input layer*, although it is not a layer of neurons. The second and consecutive layers use as neuron inputs the neuron
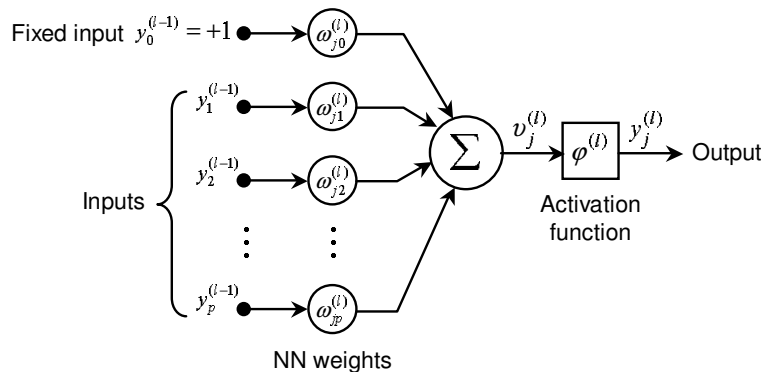
Figure 5.1: Diagram of an artificial neuron j located at layer l.

outputs of the preceding layer. Finally, the last layer of the NN is the *output layer*, in which each neuron generates an output of the network. All the neuron layers preceding the output layer are also called the *hidden layers*, since the neuron output values are not seen from the outside nor are they significant. The learning process in an NN consists of adapting the weights of the network until the output is equal to a desired response. An NN algorithm has the goal of indicating the procedure to modify the values of these weights.

Different network architectures can be found according to the connections among the neurons. *Feed-forward* networks have the same structure as described previously. Neuron inputs always proceed from a preceding layer, and signals are always transmitted forward ending at the output layer. Other kinds of architectures are *recurrent* networks. In this case, a feedback loop connects neuron outputs to the inputs of neurons located in a preceding layer. It is also possible to connect the output of one neuron to its own input, in which case it would be a self-feedback. Recurrent networks have a higher learning capability and performance, although they show a nonlinear dynamical behavior. Besides the signal transmission, NNs are also classified according to the number of layers. *Single-layer* networks have only one layer of neurons, the output layer, and are very suitable for pattern classification. On the other hand, *multilayer* networks have usually one or two hidden layers plus the output layer. Multilayer networks are able to learn complex tasks by progressively extracting more meaningful features from the NN inputs.

5.3.2   Neural Q learning

In order to approximate the Q function, a feed-forward multilayer neural network has been used. This architecture allows the approximation of any
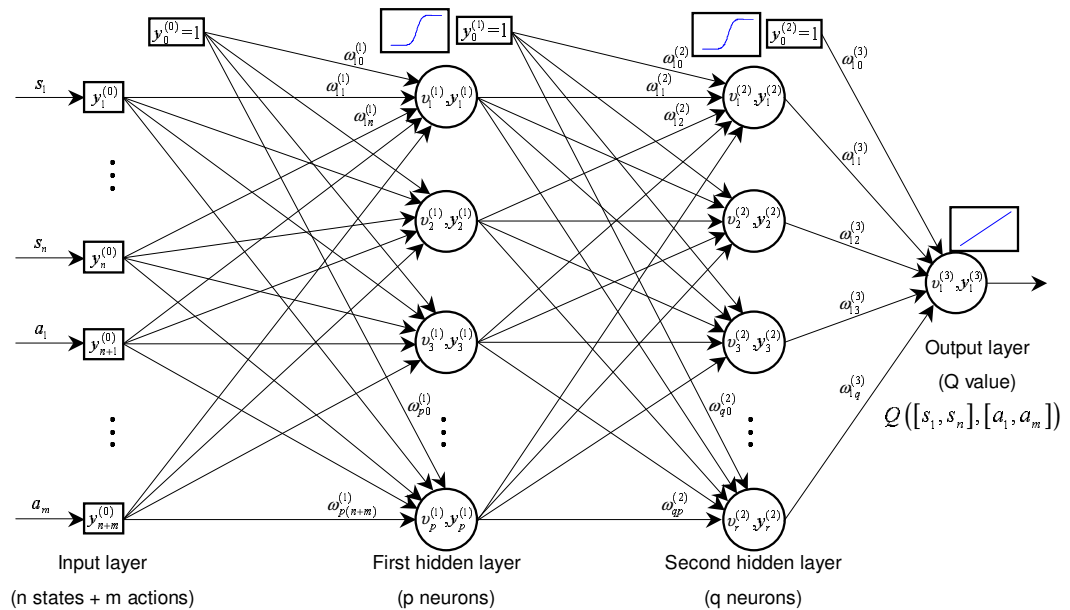
Figure 5.2: Graph of the multilayer NN which approximates the Q function. In this case, two hidden layers are used. The input layer is composed of the states and the actions. The output layer has only one neuron which contains the Q value for the current input values.

nonlinear function assuming that the number of layers and neuron, and the activation functions are appropriated. The input variables of the NN are the environment states and the actions, which have n and m dimensions respectively. The output of the network has only one dimension and corresponds to the Q value for the current states and actions. The number of hidden layers will depend on the complexity of the Q function. Figure 5.2 shows a schema of the network.

The use of Neural Networks in the Q learning algorithm is known as Neural-Q learning (NQL). There are several approaches in which NN can be applied, as commented on Section 4.6.4. In particular, the approximation of the Q function using a feed-forward NN is known as *direct* Q learning [Baird, 1995]. This is the most straight-forward approach since the whole function is approximated in only one NN. This implementation is affected by the interference problem, which will be treated in the next section. The technique used to learn the Q function is the *back-propagation* algorithm. This algorithm uses the error between the output neuron and the desired response to adapt the weights of the network.

To compute the desired response of the NN, the update equation of

Q learning algorithm is used. As detailed in the previous section, the Q value for a given state and action is equal to Equation 5.1. This means that the desired response of the NN has to be equal to Equation 5.1 and the committed error will be used to update the weights of the network.

$$Q(s_t, a_t) = r_{t+1} + \gamma \cdot max_{a_{max}} Q(s_{t+1}, a_{max}) \qquad (5.1)$$

As can be seen, the computation of Equation 5.1 requires the use of the same NN to calculate $Q(s_{t+1}, a_{max})$. In order to find the action a_{max}, which maximizes the Q value for state s_{t+1}, a simple strategy is used. The action space is discretized in a set of actions according to the smallest resolution distinguished in the environment . For each action, the Q value is computed and the maximum Q value is used. If the SONQL algorithm is applied in a task in which there is only one continuous action, the computational cost associated with the searching of a_{max} does not represent a problem. However, if more continuous actions are present, the required time can increase appreciably and the feasibility of the algorithm decreases. As will be described in Section 5.7, the SONQL algorithm is used to learn a reactive behavior which can have multiple continuous states and one continuous action for each DOF of the robot. The behavior uses one SONQL algorithm for each DOF and therefore, the computational cost of searching the a_{max} action will not represent any problem.

It is very important to note that two different learning processes are simultaneously in execution and with direct interaction. First of all, the Q learning algorithm updates the Q values in order to converge to the optimal Q function. On the other hand, the NN algorithm updates its weights to approximate the Q values. While the Q function is not optimal, both processes are updating the weights of the network to fulfill its learning purposes. It is clear that the stability and convergence of the NQL algorithm can be seriously affected by this dual learning. The next section will focus on this issue.

Finally, Equation 5.1 shows that the necessary variables to update the Neural-Q function are the initial state s_t, the taken action a_t, the received reward r_{t+1} and the new state s_{t+1}. These four variables (the states and actions can be multidimensional) constitute a *learning sample*. This term will be used in the following subsections.

5.3.3   Back-propagation algorithm

The learning algorithm applied to the NQL is the popular back-propagation algorithm, refer to [Haykin, 1999]. This algorithm has two important phases.

In the *forward* phase, an input vector is applied to the input layer and its effect is propagated through the network layer by layer. At the output layer, the error of the network is computed. In the *backward* phase, the error is used in an *error-correction* rule to update the weights of the network starting in the output layer and ending in the first hidden layer. Therefore, the error is propagated backwards.

The correction rule used to update a weight $w_{ji}^{(l)}$ is based on different aspects. A first term is the *local gradient* $\delta_j^{(l)}$ which is influenced by the propagated error and the derivative of the activation function. The derivative is calculated for the induced local field $v_j^{(l)}$ computed in the forward phase. The derivative represents a sensitivity factor which determines the direction of search in the weight space. The second term is the output signal $y_i^{(l-1)}$ transmitted through the weight. The final term is the learning rate α which determines the learning speed. If a small rate is used, the convergence of the NN to the desired function will require many iterations. However, if the rate is too large, the network may become unstable and may not converge. For a more detailed comprehension of back-propagation refer to Algorithm 3. The algorithm has been adapted to the NQL network and uses as input a learning sample k.

The activation function determines the capability of learning nonlinear functions and guarantees the stability of the learning process. The activation function that in the hidden layers is a *sigmoidal* function, in particular the *hyperbolic tangent* function. This function is antisymmetric and accelerates the learning process. The equation of this function can be seen in Algorithm 3 and Figure 5.3 shows its graph. The reason why sigmoidal functions are generally used in neural networks is for its derivative. The maximum of a sigmoidal derivative is reached when the local field is equal to 0. Since the weight change depends on this derivative, its maximum change will be performed when the function signals are in their midrange. According to [Rumelhart et al., 1986] this feature contributes to stability. The activation function of the output neuron is a *linear* function. This permits the Q function to reach any real value, since sigmoidal functions become saturated to a maximum or minimum value.

A final aspect taken into account is the *weight initialization*. This operation is done randomly but the range of values of this random function is very important. For a fast convergence, it is preferred that the activation function operate in the non-saturated medium zone of its graph, see Figure 5.3. To operate in this range, the number of inputs of each neuron and the maximum and minimum values of these inputs has to be known. Therefore, according to these parameters, the maximum and minimum values in which a weight

---

Algorithm 3: Back-propagation algorithm adapted to NQL.

---

1. Initialize the weights $w_{ji}^{(l)}$ randomly
2. **For** each *learning sample* k, composed by:
$$\{s_t(k), a_t(k), s_{t+1}(k), r_{t+1}(k)\}$$

Repeat:

(I) Forward computation

　　For each neuron j of each layer l, **compute:**
　　　a) the induced local field

$$v_j^{(l)}(k) = \sum_{i=0}^{n_n(l-1)} w_{ji}^{(l)} y_i^{(l-1)}(k)$$

　　　　where, $n_n(l-1)$ is the number of neurons of layer $l-1$
　　　b) the output signal $y_j^{(l)}(k)$

$$y_j^{(l)}(k) = \varphi^{(l)}(v_j^{(l)}(k))$$

　　　　where, $\varphi^{(l)}(x)$ is the activation function of layer l, and corresponds to,
　　　　- hyperbolic tangent in hidden layers: $\varphi(x) = 1.7159tanh(0.6667x)$
　　　　- linear function in the output layer: $\varphi(x) = x$

(II) Error computation

　　a) the output of the NQL is found in the last layer L :
$$NQL(s_t(k), a_t(k)) = y_1^{(L)}(k)$$
　　b) the desired NQL response $d(k)$ is:
$$d(k) = r_{t+1}(k) + \gamma \cdot max_{a_{max}} NQL(s_{t+1}(k), a_{max})$$
　　c) and the error is:
$$e(k) = d(k) - NQL(s_t(k), a_t(k))$$

(III) Backward computation

　　a) compute the local gradient of the output neuron:

$$\delta_1^{(L)}(k) = e(k)\varphi_L'(v_1^{(L)}(k))$$

　　　where, φ_L' is the derivative of φ and it is equal to 1 (as $\varphi(x) = x$)
　　b) **For** the rest of the neurons, starting from the last hidden layer,
　　compute the local gradient:

$$\delta_j^{(l)}(k) = \varphi_{l<L}'(v_j^{(l)}(k)) \sum_i \delta_i^{(l+1)}(k)w_{ij}^{(l+1)}$$

　　　where, $\varphi_{l<L}'$ is equal to: $\varphi(x)' = 1.1439(1 - tanh^2(0.6667x))$
　　c) **For** all the weights of the NQL, **update** its value according to:

$$w_{ji}^{(l)} = w_{ji}^{(l)} + \alpha\delta_j^{(l)}(k)y_i^{(l-1)}(k)$$
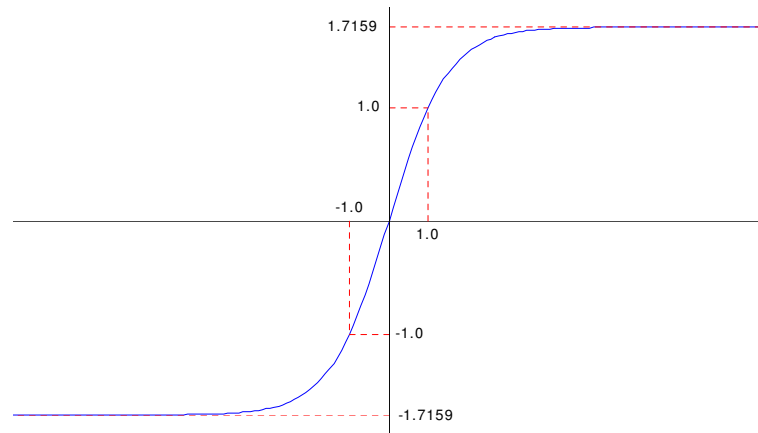
Figure 5.3: Sigmoidal function used as the activation function of the hidden layers. In particular, the function is antisymmetric with the form of a hyperbolic tangent.

can be initialized is calculated. The weight initialization implies that the maximum and minimum values of the input network signals must be known. As these signals are the state and action, which depend on the problem to be solved by reinforcement learning, a normalization, from -1 to 1, must be applied beforehand. Using this normalization, the weight initialization process will not change if the NQL algorithm is applied in different problems.

5.4   Semi-Online Learning

The previous section presented the Neural-Q_learning approach, also known as direct Q-learning. This approach has already been analyzed [Baird, 1995], and demonstrated as being unstable in simple tasks. Therefore, the convergence of Neural-Q_learning is not guaranteed and will depend on the application. This instability has also been verified in a well-known generalization problem, refer to Section 7.2. As will be described, the algorithm was not able to converge in a considerable percentage of the experiments.

The problem Neural Networks have when used to generalize with an RL algorithm is known as the *interference problem*, see Section 4.6.4. Interference in NN occurs when learning in one zone of the input space causes loss of learning in other zones. It is specially prevalent in online applications where the learning process is done according to the states and actions visited rather than with some optimal representation of all the training data [Weaver et al., 1998]. The cause of this problem is that two learning pro-

cesses are actuating at the same time and each process is based on the other. Q learning uses the NN to update the Q values and the NN computes the error of the network according to Q learning the algorithm. This dual learning makes the NQL algorithm very unstable, as has been shown. An important problem is that each time the NN updates the weights, the whole function approximated by the network is slightly modified. If the NQL algorithm updates the network using learning samples, which are all located in the same state/action zone, the non-updated state/action space will be also be affected. The result is the state/action zones which have been visited and learnt are no longer remembered. If the NQL algorithm is updating different state/action zones but with no homogeneity, the interaction between Q learning and the NN can cause instability.

The solution to the interference problem is the use of a Network which acts locally and assures that learning in one zone does not affect other zones. Approaches with Radial Basis Functions have been proposed to this end [Weaver et al., 1998], however, this implies abandoning the high generalization capability of a multilayer NN with back-propagation. The solution used in this thesis proposes the use of a *database* of *learning samples*. This solution was suggested in [Pyeatt and Howe, 1998], although to the author's best knowledge, there are no proposals which use it. The main goal of the database is to include a representative set of visited learning samples, which is repeatedly used to update the NQL algorithm. The immediate advantage of the database, is the stability of the learning process and its convergence even in difficult problems. Due to the representative set of learning samples, the Q function is regularly updated with samples of the whole visited state/action space, which is one of the conditions of the original Q learning algorithm. A consequence of the database is the acceleration of the learning. This second advantage is most important when using the algorithm in a real system. The updating of the NQL is done with all the samples of the database and, therefore, the convergence is achieved with less iterations.

It is important to note that the learning samples contained in the database are samples which have already been visited. Also the current sample is always included in the database. The use of the database changes the concept of online learning which Q learning has. In this case, the algorithm can be considered as *semi-online*, since the learning process is based on current as well as past samples. For this reason the proposed reinforcement learning algorithm has been named *Semi-Online Neural-Q learning* algorithm (SONQL).

Each learning sample, as defined before, is composed of the initial state s_t, the action a_t, the new state s_{t+1} and the reward r_{t+1}. During the learning evolution, the learning samples are added to the database. Each new sam-
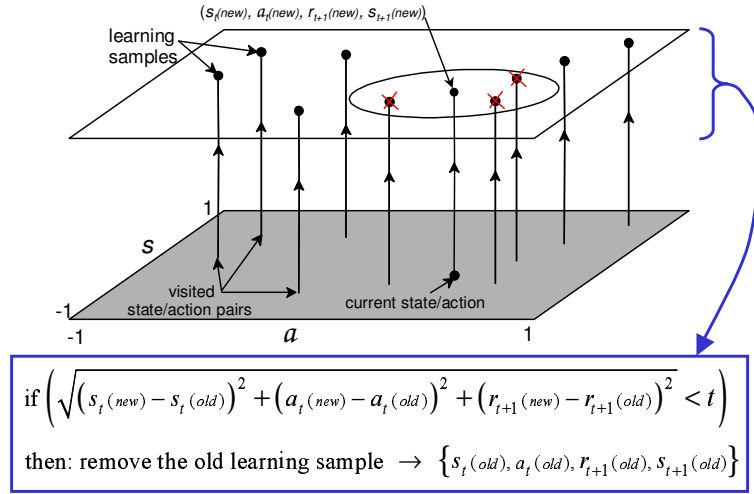
Figure 5.4: Representation of the learning sample database. The state and action have only one dimension. The replacement rule for all the old samples is also shown.

ple replaces older samples previously introduced. The replacement is based on the geometrical distance between vectors (s_t, a_t, r_{t+1}) of the new and old samples. If this distance is less than a *density parameter t* for any old sample, the sample is removed from the database. The size of the database is, therefore, controlled by this parameter which has to be set by the designer. Once the algorithm has explored the reachable state/action space, a homogeneous, and therefore, representative set of learning sample is contained in the database. Figure 5.4 shows a representation of the learning samples database in a simple case in which the state and action are only one-dimensional.

The selection of s_t, a_t and r_{t+1} to determine if an old learning sample has to be removed has several reasons. The use of only s_t and a_t, and not s_{t+1} is due to the assumption that the dynamic of the environment is highly deterministic. Therefore, it is assumed that if two samples have the same s_t and a_t values, but different s_{t+1}, it means that the environment may have changed. It is preferable then to remove the old sample and retain the new sample which should be more representative of the environment. Since the application of the SONQL algorithm is for robot learning, it is assumed that the stochastic transition which may occur is not significative. Finally, the use of r_{t+1} is only to acquire more samples in the space zones in which the reward changes. If s_t and a_t of the old and new samples are very close but the reward is different, it is important to retain both samples. This will allow to the algorithm to concentrate on these samples and learn the cause which

make the reward different.

After including the database of learning samples, the difference between a NQL iteration and a SONQL iteration must be distinguished. In each iteration of the SONQL algorithm there will be as many NQL iterations as the number of samples. Moreover, the number of SONQL iterations is equivalent to the number of interactions with the environment. This justifies the learning acceleration, since for each environment interaction, the SONQL algorithm updates the NQL function several times. Finally, the improvements caused by the database have not been theoretically demonstrated, although this section has attempted to justify them. Only empirical results, see Chapter 7, validate the proposal.

5.5 Action Selection

After updating the Q function with the learning samples, the SONQL algorithm must propose an action. As the algorithm is based on Q_learning, which is an off-policy algorithm, any policy can be followed. In practice, the policy followed is the $\epsilon - greedy$ policy, which was described in Section 5.2. With probability $(1 - \epsilon)$, the action will be the one which maximizes the Q_function in the current state s_{t+1}. Otherwise, an aleatory action is generated. Due to the continuous action space in the Neural-Q_function, the maximization is accomplished by evaluating a finite set of actions. The action space is discretized with the smallest resolution that the environment is able to detect. In the case of a robot, the actions would be discretized in a finite set of velocity values considered to have enough resolution for the desired robot performance. As commented in Section 5.3.2, if more than one action is present, the search of the optimal action can require a lot of computation. In that case, the search of the greedy action is necessary to obtain the $Q(s, a_{max})$ value. The search of the greedy action is one of the drawbacks of continuous functions, as pointed out in [Baird and Klopf, 1993]. However, the SONQL algorithm was designed to learn only one DOF of the robot and, therefore, this problem is avoided in the results presented in this thesis. Section 5.7 details the application of the SONQL algorithm.

5.6 Phases of the SONQL Algorithm

In this section, the Semi-Online Neural-Q learning algorithm is broken down in a set of phases. This break down allows a clearer comprehension of the algorithm. Each phase is used to fulfill a simple task of the algorithm. The

algorithm is structured sequentially starting with the observation of the environment state and finishing with the proposal of a new action. The SONQL algorithm is divided into four different phases which are graphically shown in Figure 5.5.

Phase 1. LS Assembly. In the first phase, the current Learning Sample (LS) is assembled. The LS is composed of the state of the system s_t, the action a_t taken from this state, the new state s_{t+1} reached after executing the action, and the reward r_{t+1} received in the new state. The action will usually be the one generated in the fourth phase of this algorithm. However, in some applications an external system can modify the executed action. For instance, in the control architecture proposed in Section 3.2, the action proposed by a higher priority behavior can be selected instead of the one proposed by the SONQL algorithm. Therefore, the real executed action must be observed. The last term to complete the learning sample is the reward r_{t+1}. Although in the original RL problem the reward is perceived from the environment, in this approach it has been included as a part of the algorithm. However, it is computed according to a preprogrammed function which usually uses the state s_{t+1}. This function has to be set by the programmer, and will determine the goal to be achieved.

Phase 2. Database Update. In the second phase, the database is updated with the new learning sample. As has been commented on, old samples similar to the new one will be removed. Therefore, all the samples contained in the database will be compared with the new one.

Phase 3. NQL Update. The third phase consists of updating the weights of the NN according to Algorithm 3. For each sample of the database, an iteration of the NQL algorithm with the back-propagation algorithm is performed.

Phase 4. Action Selection. The fourth and final phase consists of proposing a new action a_{t+1}. The policy followed is the $\epsilon - greedy$, which was described in Section 5.5.

5.7   SONQL-based behaviors

After having analyzed the SONQL algorithm, this section shows the application for which it was designed. The goal of the SONQL algorithm is to learn the state/action mapping of a reactive behavior, as introduced in Section 3.1. An example of a reactive behavior is the *target following* behavior.
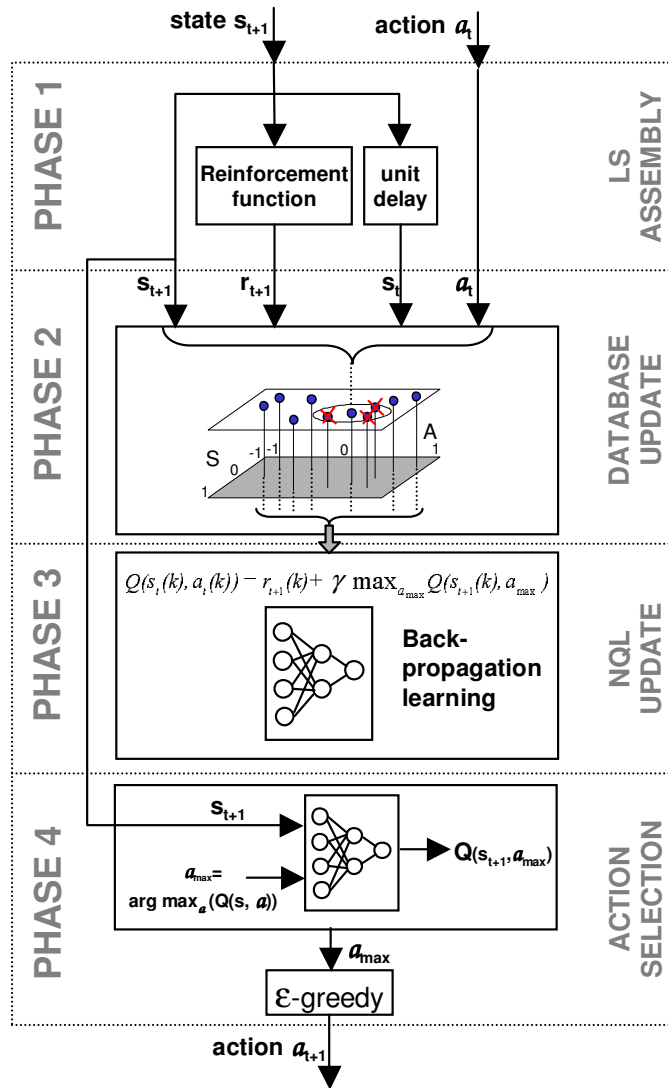
Figure 5.5: Phases of the Semi-Online Neural-Q learning algorithm.

In this case, the goal of the behavior is to generate the control actions which, according to the position of the target, make the robot follow the target. In this kind of application, the states and actions are usually continuous, which causes the generalization problem. The use of the SONQL algorithm permits the learning of the state/action mapping solving the generalization problem. The SONQL algorithm makes Reinforcement Learning feasible for this application.

The implementation of the reactive behavior with the SONQL algorithm implies the accomplishment of an important condition. The behavior has to generate a control action at the frequency of the high-level controller. This means that the SONQL algorithm cannot stop the periodic execution of the robot control system. To guarantee this constraint, the SONQL algorithm is implemented with two different *execution threads*. The *control* thread has a high priority and is executed at the frequency of the high-level controller. This thread computes phases 1, 2 and 4 of the SONQL algorithm, that is the "LS assembly", the "database update" and the "action selection". These phases do not require a great deal of computation and are used to memorize the new learning sample and to generate a new control action. The second thread is the *learning* thread, which contains phase 3 and requires more computation. This thread has a lower priority and will be executed during all the available time until all the samples of the database have been updated. Therefore, the available computational resources will be used to update the NQL function. Figure 5.6 shows the SOQNL algorithm arranged with these two threads.

As in all Reinforcement Learning problems, the application of the SONQL algorithm in a behavior implies the identification of the environment. Everything external to the algorithm having an influence over the observed state is considered as the environment. The observed state will depend on the current state and the executed control action. This action is generated by the hybrid coordinator which can or cannot use the action proposed by the SONQL algorithm. For this reason, and taking advantage of the *off-policy* feature of Q_learning, this final action is feedback to the SONQL algorithm, see Figure 5.7. Once the final action has been generated, the low-level controller will actuate over the robot which will move accordingly. These systems are considered as the environment of the RL problem. Also, included in the environment, a perception module will be responsible for observing the environment state which will, in turn, be sent to the SONQL algorithm.

It is very important to note that the Q_learning theory is based on the assumption that the environment can be modelled with a FMDP. The most important constraint of this assumption is that the state contains a complete observation of the environment. Therefore, the state must contain all
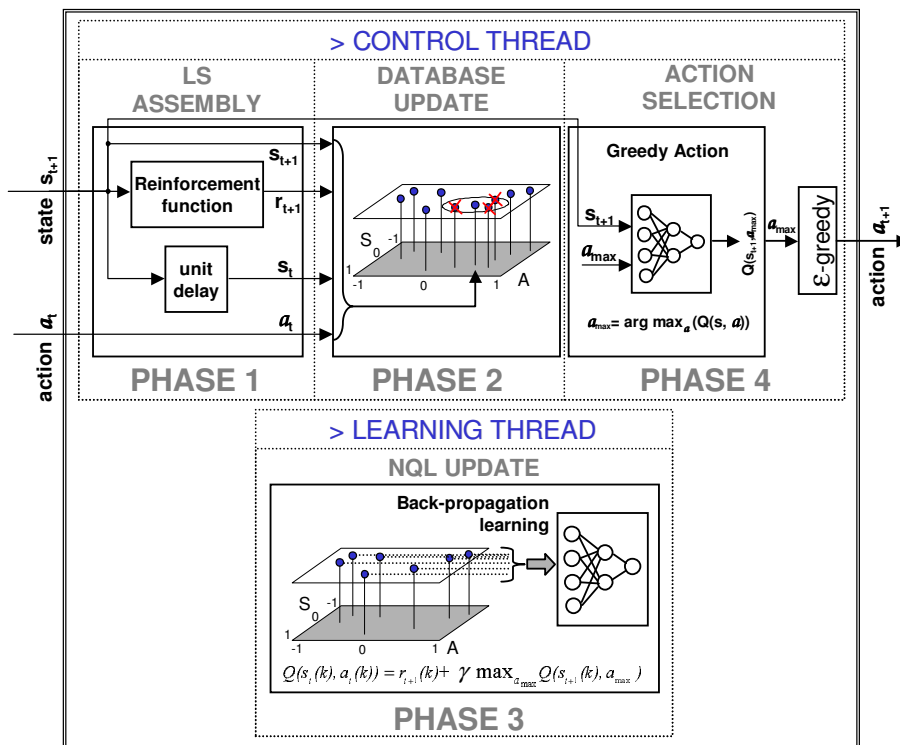
Figure 5.6: Diagram of the SONQL algorithm in its implementation for a robot behavior. The control thread is used to acquire new learning samples and generate the control actions. The learning thread, with less priority, is used to update the NQL function.
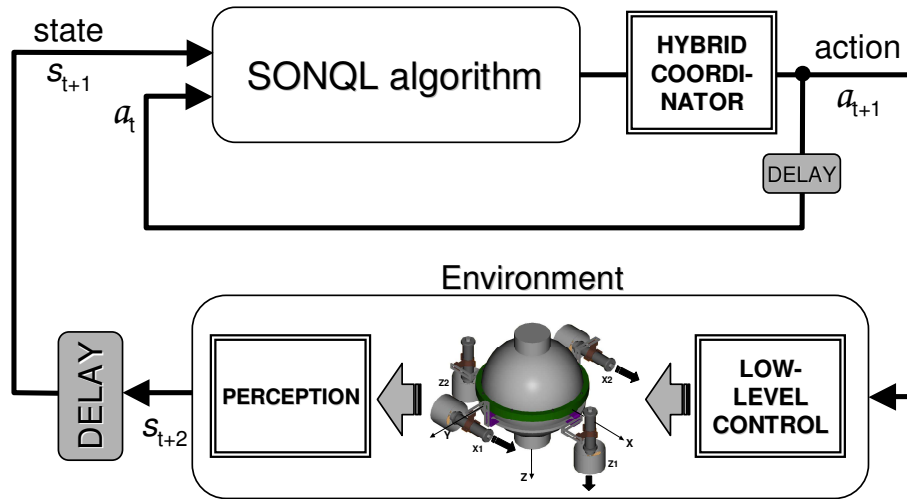
Figure 5.7: Diagram of the SONQL algorithm acting as a behavior, respect to the robot control system.

the variables needed to predict the new state without using past information. An easy mistake when applying RL to a real system is to not provide the complete state, which breaks the theoretical assumptions and makes the learning impossible. It is also very common to not observe the state correctly and this also impedes the learning. The importance of the correct observation of the Markovian state is one of the biggest problems of RL together with the generalization problem. This issue will be covered in the experimental results shown in Chapter 7.

As has been described, the implementation of the reactive behaviors can be accomplished using the SONQL algorithm, which contains the state/action mapping. However, a behavior response, as defined is Section 3.2, is also composed by an *activation level*. The SONQL algorithm does not include this activation level, and has to be manually implemented. For example, in the case of the *target following* behavior, the activation level will be $a_i = 1$ if the target is detected, otherwise it will be $a_i = 0$. Therefore, the implementation of a SONQL-based behavior requires the definition of the reinforcement function and the activation level function.

Another implementation aspect is the uncoupling of the degrees of freedom (DOF) of the robot. For each DOF of the robot, an independent SONQL algorithm is used. This greatly improves the real-time execution of the SONQL algorithm. The searching of the a_{max} action is accomplished by discretizing the action space. If more than one action is present, the num-
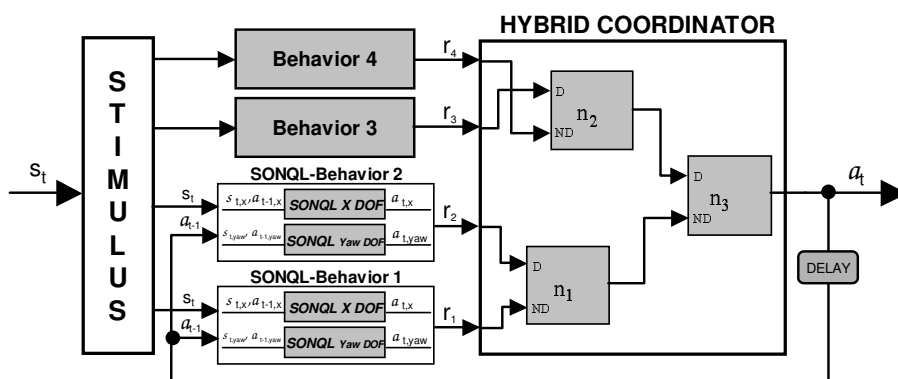
Figure 5.8: Example of the behavior-based control layer with two SONQL-based behaviors.

ber of combinations will be very high and the algorithm will need a number of computations to find the greedy action. It must be noted that, this greedy action has to be found in the fourth phase of the algorithm and for each learning sample learnt in phase two. The use of more than one action with the SONQL algorithm is an important problem to ensure the real-time execution of the algorithm. The adopted solution consists of having a different SONQL for each DOF with each one is independently learnt. This could be a disadvantage. However, the simultaneous learning of two DOFs in practice would make the correct observation of the Markovian state very difficult, as will be shown in Section 7.1. This means that the limitation of the SONQL algorithm with multiple actions is not a basic necessity in most robotics tasks. Figure 5.8 shows an example of the behavior-based control layer in which there are two SONQL-based behaviors and two manually tuned behaviors. Two different learning algorithms have been used for each SONQL behavior.

As already commented on, the state has to contain all the necessary variables to predict the new state. It is very important to simplify the state information as much as possible. For example, the target following behavior will need the position of the target with respect to the robot. If the target is detected through a video camera, it is necessary to reduce the pixel information to a simple value. For each DOF, this value is calculated according to the position of the target in the image or the size of the target. Figure 5.9 shows how the target position has been translated in three variables $\{f_x, f_y, f_z\}$.

Finally, the reinforcement function must be designed in order to point out the goal of the behavior. Although any reward value can be assigned to a state, only a finite set of values are used in the SONQL-based behaviors: $\{-1, 0, 1\}$. Figure 5.9 shows the reinforcement functions $\{r_x, r_y, r_z\}$ for each
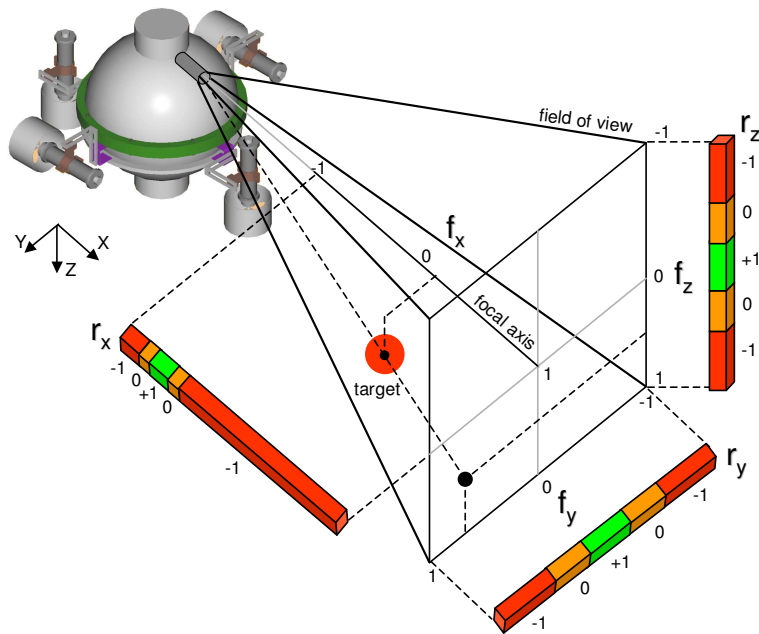
Figure 5.9: State $\{f_x, f_y, f_z\}$ and reward $\{r_x, r_y, r_z\}$ variables in a *target following* behavior.

DOF. The simplicity of the function reduces the necessary knowledge of the designer to implement a new behavior. Section 7 demonstrates the facility and feasibility of this reinforcement function definition.

5.8   Discussion

The proposed SONQL algorithm has been demonstrated to be a feasible approach to learn each DOF of a robot behavior. The experimental results will be analyzed in Chapter 7. However, it is important to summarize the theoretical aspects in which this approach is based. The SONQL is based on Q_learning and uses a Neural Network and a database of learning samples to solve the generalization problem. The use of Q_learning is because its learning procedure is off-policy. Another feature is that it uses the action-value function, Q, from which the optimal policy is easy to extract. The reason a Neural Network was used is for its high generalization capability. However, these two components, Q_learning and NN, contradict some of the convergence proofs explained in Section 4.6. First, the convergence of linear approximators combined with off-policy methods is not guaranteed. Indeed, there are counterexamples which show the divergence. In addition,

NN is a non-linear function approximator for which the convergence is also not guaranteed. The large number of successful examples which have been demonstrated in practice are the only reasons to justify the selection of these techniques.

Besides convergence proofs, Neural Networks suffer from the interference problem when they are used as function approximators with RL algorithms as explained in Section 4.6.4. To solve this problem, the SONQL algorithm uses the database of learning samples. The goal of this database is to acquire a representative set of samples,which continually update the NN. The homogeneity which the database provides is able to solve the interference problem. In addition, this database accelerates the learning process since several updates can be performed at each environment iteration.

The main drawback of the SONQL algorithm is the searching of greedy actions. As has been described in Section 5.5, greedy actions are found by discretizing the action space. From a control point of view, this is not a problem if the discretization is fine enough. However, the necessary computation increases if several continuous actions are present. This problem does not appear in the implementation of the SONQL algorithm in a reactive behavior. For each DOF of the robot, an independent SONQL algorithm is used. Since the main goal of this approach was to design an RL algorithm able to learn robot behaviors, this drawback was not considered, although it represents one of the future works of this dissertation.

Finally, the use of the SONQL in a behavior requires the definition of a set of parameters: the NN configuration (number of layers and neurons), the learning rate α, the discount factor γ, the exploration probability ϵ, the database density parameter t, the reinforcement function $r_i(s)$ and the activation level function a_i. Also, the goal to be accomplished by the behavior has to be analyzed, assuring that the state is completely observed. A first conclusion of the SONQL-based behaviors could induce a very complex technique for solving a much simpler problem. However, it has to be noted that most of these parameters will depend on the robot's dynamics and, therefore, they will be equal for other behaviors. These invariant parameters are: the NN configuration (number of layers and neurons), the learning rate, the discount factor, the exploration probability and the database density parameter. Consequently, the implementation of a new behavior will require only the design of the reinforcement function, the activation level function and the analysis of the behavior task. These two functions, as has been shown, are very simple and intuitive.

Chapter 6

URIS' Experimental Set-up

The experimental set-up designed to work with the Autonomous Underwater Vehicle URIS is compounded of a water tank, two sensory systems, a distributed software application and the robot itself. The overall set-up is shown in Figure 6.1. The correct operation of all these systems in real-time computation allows the experimentation and, therefore, the evaluation of the proposed SONQL behaviors and hybrid coordination system. The purpose of this chapter is to report the characteristics of these systems and their interactions. First, the main features of the robot are given. These include the design principles, the actuators and the on board sensors. The two sensory systems, specially designed for these experiments, are then presented. The first of these systems is the target detection and tracking system. The second is the localization system which is used to estimate the three-dimensional position, orientation and velocity of the vehicle inside the water tank. Finally, the software architecture, based on distributed objects, is described.

6.1 Robot Overview

*U*nderwater *R*obotic *I*ntelligent *S*ystem is the meaning of the acronym URIS. This Unmanned Underwater Vehicle (UUV) is the result of a project started in 2000 at the University of Girona. The main purpose of this project was to develop a small-sized underwater robot with which to easily experiment in different research areas like control architectures, dynamics modelling and underwater computer vision. Another goal of the project was to develop an Autonomous Underwater Vehicle (AUV) with the required systems, hardware and software as the word *autonomous* implies. Other principles are flexibility in the tasks to be accomplished and generalization in the developed systems.

Figure 6.1: URIS' experimental environment.

6.1.1   Design

The design of this vehicle was clearly influenced by its predecessor Garbi UUV
[Amat et al., 1996], although some mechanical features were redesigned. The
shape of the vehicle is compounded of a spherical hull surrounded by various
external elements (the thrusters and camera sensors). The hull is made of
stainless steel with a diameter of 350mm, designed to withstand pressures
of 3 atmospheres (30 meters depth). On the outside of the sphere there
are two video cameras (forward and down looking) and 4 thrusters (2 in X
direction and 2 in Z direction). All these components were designed to be
water-proof, with all electrical connections made with protected cables and
hermetic systems. Figure 6.2 shows a picture of URIS and its body fixed
coordinate frame. Referred to this frame, the 6 degrees of freedom (DOFs)
in which a UUV can be moved are: *surge*, *sway* and *heave* for the motions in
X, Y and Z directions respectively; and *roll*, *pitch* and *yaw* for the rotations
about X, Y and Z axes respectively.

URIS weighs 30 Kg., which is approximately equal to the mass of the
water displaced and, therefore, the buoyancy of the vehicle is almost neutral.
Its gravity center is in the Z axis, at some distance from below the geometrical
center. The reason for this is the distribution of the weight inside the sphere.
The heavier components are placed at the bottom. This difference between

a)   b)
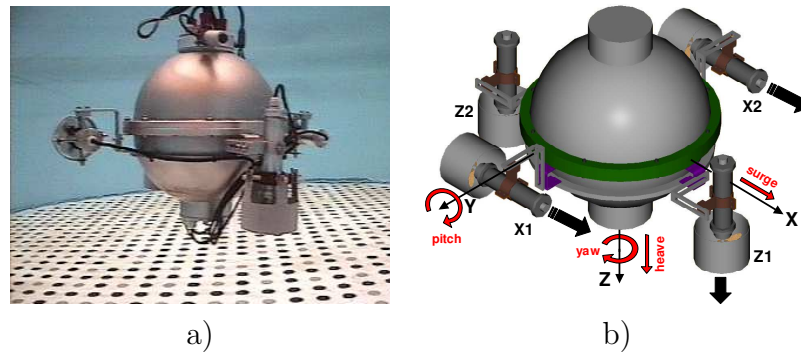
Figure 6.2: URIS' AUV, a) picture b) schema.

the two centers entails a stability in both *pitch* and *roll* DOFs. The further down the gravity center is, the higher the torque which has to be applied in the X or Y axes to incline the robot a certain value in *roll* or *pitch*, respectively.

The movement of the robot is accomplished by its 4 thrusters. Two of them, labelled *X1* and *X2* in Figure 6.2b, exert a force in X axis and a torque in Z axis. The resultant force of both trusters is responsible for the *surge* movement of the vehicle, and the resultant torque is responsible for the *yaw* movement. Analogously, the other two thrusters, Z1 and Z2, exert a force in Z axis and a torque in Y axis. The resultant force is responsible for the *heave* movement of the vehicle, and the resultant torque is responsible for the *pitch* movement. In this case, the *pitch* movement is limited to only a few degrees around the stable position, since the gravity and buoyancy forces cause a high stabilization torque compared to that of the thruster. Therefore, only 4 DOFs can be actuated leaving the *sway* and *roll* movements without control. Like the *pitch* DOF, the *roll* DOF is stabilized by the gravity and buoyancy forces. The *sway* movement is neither controlled nor stabilized by any force, which makes it sensitive to perturbations like water currents or the force exerted by the umbilical cable. Hence, URIS is a nonholonomic vehicle.

The inside of the hull was arranged to contain all the necessary equipment for an autonomous system. First of all, the lower part of the sphere contains various battery packages conceived to power the vehicle for a period of one hour. A second level, above the batteries, contains the drivers of the 4 thrusters. Some electronic boards, mainly sensor interfaces, are also included in this level. In the third level, all the hardware components and electrical connections among all systems is found. The hardware architecture is com-

pounded of two embedded computers. One computer is mainly in charge of the control of the robot and the other is used for image processing and other special sensors. The communication between computers is done through an ethernet network and the communication between these computers and sensors/actuators is done through other interfaces: serial lines, analog and digital inputs and outputs, and video frame grabbers. All these devices, except the thruster drivers, are powered by a DC-DC converter which supplies different voltages. The thruster drivers are directly powered by some battery packages specifically used for that purpose.

Besides the systems located in the robot, URIS' experimental set-up is also compounded of external systems, making some kind of connection indispensable. For this purpose, an underwater umbilical cable is used. Three different types of signals are sent through this cable. First, two power signals are sent to the robot to supply the power for the thrusters and the power for all the electronics independently. The second type of signal is an ethernet connection, connecting the on-board and off board computers. Finally, two video signals from the two on board cameras are also transmitted. Different reasons justify the use of this umbilical cable. First, the use of external batteries increases the operation time of the robot up to the whole journey. The second reason is to help in the understanding of the experiments, allowing a real-time supervision of them through data and video monitoring. The third reason is to allow the computation of a part of the software architecture out board, such as the target tracking (Section 6.2) and the localization system (Section 6.3). The first and second reasons are aids in the development of any new experiment. The third reason allows us to confront some technological problems using external hardware and computational resources.

6.1.2   Actuators

As commented on above, URIS has four actuators to move the vehicle in four DOFs. Each actuator or thruster is equipped with a DC motor, encapsulated in a waterproof hull and connected, through a gear, to an external propeller. Around the propeller, a cylinder is used to improve the efficiency of the water jet. The power of each thruster is 20 Watts carrying out a maximum thrust of 10 Newtons at 1500 rpms. The control of the DC motor is accomplished by a servoamplifier unit. This unit measures the motor speed with a tacho-dynamo and executes the speed control according to an external set-point. The unit also monitors the values of the motor speed and electric current. Communication between the onboard computer and each motor control unit is done through analog signals.

6.1.3   Sensors

The sensory system is one of the most important parts in an autonomous robot. The correct detection of the environment and the knowledge of the robot state, are very important factors in deciding how to act. URIS has a diverse set of sensors. Some of them are used to measure the state of the robot and others to detect the environment. Hereafter the main characteristics and utility of each sensor are commented upon.

- **Water Leakage Detection**. In order to detect any water leakage, there are several sensors which use the electric conductivity of the water to detect its presence. These sensors are located inside each thruster case as well as inside the lower part of the hull. Any presence of water is immediately sited before valuable systems can be damaged. The interface of the sensors is through digital signals.

- **Thruster monitors**. As commented on in Section 6.1.2, each thruster is controlled by a control unit which monitors the thruster's rotational speed and its electric current. These two analog measures can be used to detect faults. For instance, if the current is much higher or much lower than in normal conditions, it may mean that the helix has been blocked or has been lost. In addition, knowledge of the thruster speeds can be used to calculate the thrust and, using the dynamics model of the vehicle , to estimate the acceleration, velocity and position of the vehicle. Obviously, the inaccuracies of the model, the external perturbations and drift of the estimations would entail to a rough prediction, but combining it with another navigation sensor, a more realistic estimation can be obtained.

- **Inertial Navigation System**. An inertial unit (model MT9 from Xsens) is also placed inside the robot. This small unit contains a 3D accelerometer, a 3D rate-of-turn sensor and a 3D earth-magnetic field sensor. The main use of this sensor is to provide accurate real-time orientation data taken from the rate-of-turn sensor. The accelerometers and the earth-magnetic field sensors provide an absolute orientation reference and are used to completely eliminate the drift from the integration of rate-of-turn data. From this sensor then, the *roll*, *pitch* and *yaw* angles can be obtained. The interface with the sensor is through the serial line.

- **Pressure sensor**. This sensor measures the state of the robot. In this case, the pressure detected by the sensor provides an accurate

measurement of the depth of the robot. Due to the electromagnetic
noise, the sensor signal needs hardware and software filtering and also
data calibration.

- **Forward and Downward looking video cameras**. Unlike previ-
  ous sensors, the video cameras provide detection of the environment.
  URIS has two water-proof cameras outside the hull. One of them is
  a color camera looking along the positive X axis, see Figure 6.2. The
  use of this camera is to detect targets, as will be shown in Section 6.2.
  Another use is for teleoperation tasks. The second camera is a black-
  and-white camera looking along the positive Z axis. The main utility of
  this camera is the estimation of the position and velocity of the vehicle.
  For this purpose, two different approaches have been considered. In the
  first approach, the motion estimation is performed from images of the
  real underwater bottom using visual mosaicking techniques. This lo-
  calization system is one of the research lines of the underwater robotics
  group [Garcia et al., 2001]. The second approach was inspired by the
  first and was developed to work specifically in the URIS experimental
  set-up. It is a localization system for structured environments based
  on an external coded pattern. For further information on both systems
  refer to Section 6.3.

- **Sonar transducer**. This sensor is used to detect the environment.
  The sonar transducer (Smart sensor from AIRMAR) calculates the dis-
  tance to the nearest object located in the sonar beam. The transducer
  is placed outside the hull looking in the direction in which objects have
  to be detected. A typical application is to point the beam at the bot-
  tom to detect the altitude of the vehicle. The interface of this sensor
  is through a serial line.

The sensors used in the experiments presented in this dissertation are
the water leakage sensors and the two video cameras. The following sections
detail the computer vision systems which were developed to extract useful
information from the camera images. The forward looking camera was used
to detect a moving target in the environment, and the downward looking
camera to estimate the state (position and velocity) of the robot inside the
water tank.

6.2   Target Tracking

One of the sensory systems developed for the experimental set-up of URIS
is the *target detection and tracking* system. This vision-based application
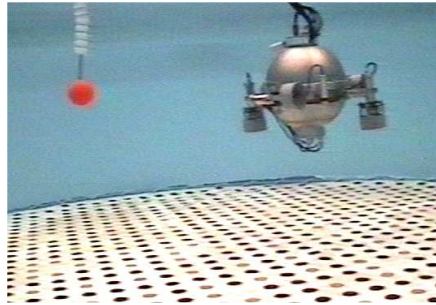
Figure 6.3: URIS in front of the artificial target.

has the goal of detecting an artificial target by means of the forward looking camera. This camera provides a large underwater field of view (about 57° in width by 43° in height). This system was designed to provide the control architecture with a measurement of the position of an object to be tracked autonomously. Since the goal of this dissertation is to test control and learning systems, a very simple target was used. The shape selected for the target was a sphere because it has the same shape from whatever angle it is viewed. The color of the target was red to contrast with the blue color of the water tank. These simplifications allowed us to use simple and fast computer vision algorithms to achieve real-time (12.5 Hz) performance. Figure 6.3 shows a picture of the target being observed by URIS.

The procedure of detecting and tracking the target is based on image segmentation. Using this simple approach, the relative position between the target and the robot is found. Also, the detection of the target in subsequent images is used to estimate its relative velocity. The following subsections detail the image segmentation algorithm, the coordinate frame in which the position is expressed and the velocity estimation.

6.2.1   Image Segmentation

The detection of the target is accomplished by color segmentation. This technique is very common in computer vision and consists of classifying each pixel of the image according to its color attributes. The pixels which satisfy the characteristics of the target are classified as part of it. In order to express the color of an object, the *HSL* (Hue, Saturation and Luminance) color space is usually preferred over the standard *RGB* (Red, Green and Blue). Within the HSL color space, the hue and saturation values, which are extracted from the RGB values, define a particular color. Therefore, the color of an object is defined by a range of values in hue and saturation, and the segmented image
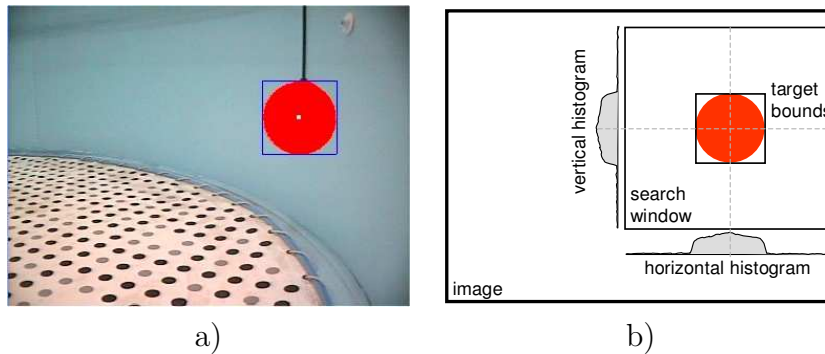
Figure 6.4: Image segmentation, a) real image with the detected target b) scheme used in the segmentation process.

is the one containing the pixels within the hue and saturation ranges.

In this application, the segmentation is carried out in two phases. The first phase consists of finding just a portion of the target within the whole searched image. It is important to ignore pixels which, due to noise or reflections, have a similar color to the target. Therefore, restrictive hue and saturation ranges are applied. The consequence is an image in which only the most saturated portions of the target appear. After the first segmentation, two histograms of the segmented pixels in the horizontal and in the vertical axes of the searched image are calculated. From the two histograms, the maximum values are found. These two coordinates constitute the *starting point* which is considered to belong to the target. In Figure 6.4, the segmentation process in a real image and a scheme of the image, in which the histograms are represented, is shown.

After the computation of the starting point, the second segmentation is carried out. In this case, the segmentation is less severe than the first. What that means is, besides the pixels belonging to the target, other pixels of the image can also be segmented. This segmentation is used to find the portions of the target not detected by the previous one. Using the second segmentation, a region growing process is applied. The growing process is begun at the starting point and will expand the target area until no more segmented pixels are connected between them. At this point, the target has been completely detected, see Figure 6.4a. The position of the target is considered to be the center of the rectangle which contains the target. The size of the target is calculated according to the mean value of the two sides of the rectangle.

The effect of the first segmentation guarantees the correct location of the target even in the presence of noise. On the other hand, the second

segmentation guarantees the correct segmentation of portions of the target which may be affected by shadows which reduce the saturation of the color. Finally, instead of searching for the target in the whole image, a smaller window is used, see Figure 6.4b. This window is centered on the position found in the previous image. In case the target is not found inside the window, the whole image is explored.

6.2.2 Target Normalized Position

Once the target has been detected, its relative position with respect to the robot has to be expressed. The coordinate frame which has been used for the camera has the same orientation as the URIS coordinate frame, but is located in the focal point of the camera, see Figure 6.5. Therefore, the transformation between the two frames can be modelled as a pure translation.

The X coordinate of the target is related to the target size detected by the segmentation algorithm. A normalized value between *-1* and *1* is linearly assigned to the range comprised between a maximum and minimum target size respectively. It is important to note that this measure is linear with respect to the size, but non-linear with respect to the distance between the robot and the target measured in X axis. In Figure 6.5 the f_x variable is used to represent the X coordinate of the target.

Similarly, the Y and Z coordinates of the target are related to the horizontal and vertical positions of the target in the image, respectively. However, in this case, the values represented by the f_y and f_z variables do not measure a distance, but an angle. The f_y variable measures the angle from the center of the image to the target around the Z axis. The f_z variable measures the angle from the center of the image to the target around the Y axis. In this case, the angles are also normalized from *-1* to *1* as it can be seen in the Figure 6.5.

As has been described, the coordinates of the target are directly extracted from the position and size found in the segmentation process. This means that the calibration of the camera has not been taken into account. Therefore, the non-linear distortions of the camera will affect the detected position. Moreover, the measures of the f_x variable are non-linear with the distance to the target in X axis. These non-linear effects have consciously not been corrected to increase the complexity with which the SONQL-based behavior will have to deal.
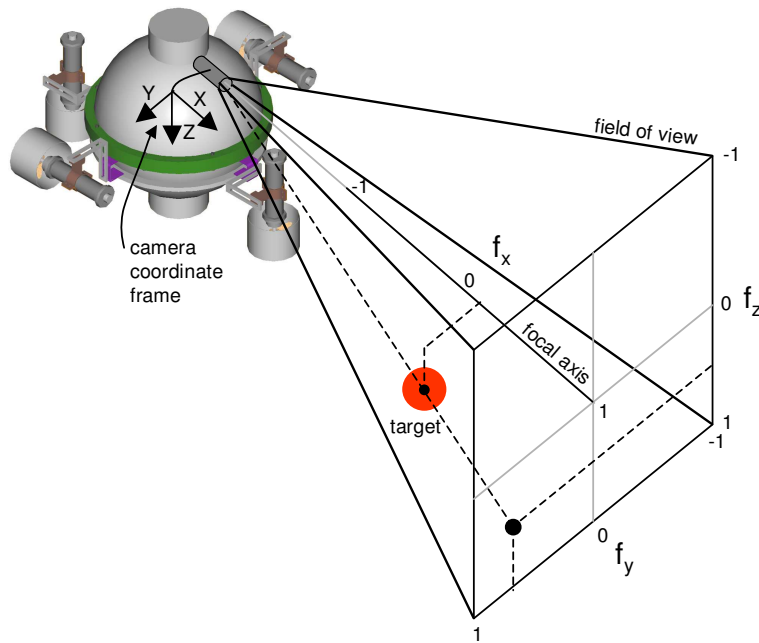
Figure 6.5: Coordinates of the target in respect with URIS.

6.2.3   Velocity Estimation

In order to properly follow the target, the measure of its relative position is not enough. An estimation of its relative velocity is also necessary. To calculate this velocity, the f_x, f_y and f_z variables are differentiated from the sequence of images. In particular, a first order Savitzky-Golay [Savitzky and Golay, 1964] filter, with a first order derivative included, is applied to these signals. The result of this operation is the estimation of $\frac{df_x}{dt}$, $\frac{df_y}{dt}$ and $\frac{df_z}{dt}$. Due to the filtering process, a small delay is added to these signals with respect to the ideal derivatives. However, these delays do not drastically affect the performance of the experiments, as will be shown in Chapter 7. Figure 6.6 shows the movement of the target in Y axis. The target was first moved to the right and then twice to the left. The estimated velocity is also shown.

6.3   Localization System

Localization is the estimation of the vehicle's position and orientation with respect to a global coordinate frame. A localization system is needed when tasks involving positioning have to be carried out. Moreover, an estima-
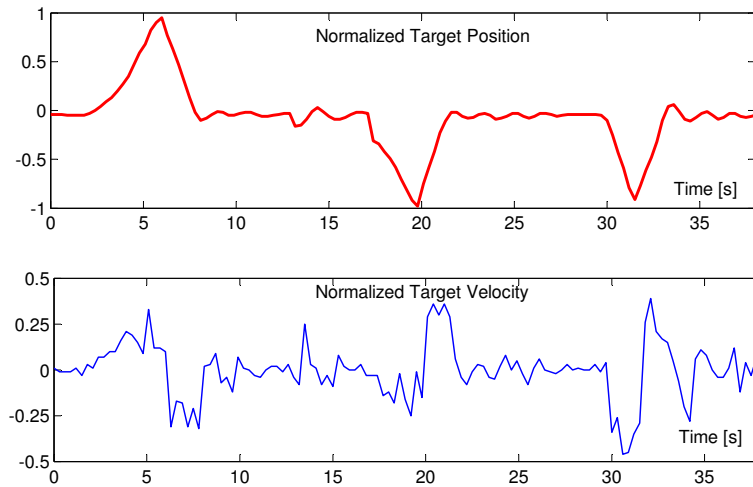
Figure 6.6: Normalized angular position and velocity of the target in Y axis.

tion of the vehicle's velocity is usually required by the low-level controller. The localization of an underwater vehicle is a big challenge. The detection of the vehicle's speed with respect to the water is very inaccurate and not reliable due to water currents. Electromagnetic waves are strongly attenuated when travelling through water, which also dismisses the use of a GPS. Main techniques used for underwater vehicle localization are inertial navigation systems, and acoustic and optical sensors. Among these techniques, visual mosaics have greatly advanced over the last few years offering, besides position, a map of the environment [Negahdaripour et al., 1999, Gracias and Santos-Victor, 2000]. The general idea of visual mosaicking is to estimate the movement of the vehicle by recognizing the movement of some features on the ocean floor. An onboard downward-looking camera is used to perceive these features. Main advantages of mosaicking with respect to inertial and acoustic sensors are lower cost and smaller sensor size. Another advantage is that the environment does not require any preparation, in contrast with some technologies which use a network of acoustic transponders distributed in the environment. However, position estimation based on mosaics can only be used when the vehicle is performing tasks near the ocean floor and requires reasonable visibility in the working area. There are also unresolved problems like motion estimation in presence of shading effects, presence of "marine snow" or non-uniform illumination. Moreover, as the mosaic evolves, a systematic bias is introduced in the motion estimated by the mosaicking algorithm, producing a drift in the localization of the robot [Garcia et al., 2002].

In the experimental set-up used for URIS' AUV, a vision-based localization system was developed. The system was inspired by visual mosaicking techniques [Garcia et al., 2001]. However, simplifications were made in order to have a more accurate and drift free system. Instead of looking at the unstructured ocean floor of a real environment, a coded pattern was used. This pattern has the same size as the water tank and was placed on its bottom. The pattern contains landmarks which can be easily tracked and, by detecting its global position, the localization of the vehicle is accomplished.

The localization system provides an estimation of the three-dimensional position and orientation of URIS referred to in the tank coordinate frame. In addition, an estimation of the vehicle's velocities, including *surge, sway, heave, roll, pitch* and *yaw*, is computed. The algorithm is executed in real-time (12.5 Hz) and is entirely integrated in the controllers of the vehicle.

In the next subsections, detailed information about the localization system is given. First, the projective model of the downward-looking camera is detailed. Then, the design and main features of the coded pattern are described. After describing the main components of the system, the different phases found in the localization algorithm are sequentially explained. Finally, some of the results and experiments concerning the accuracy of the system are presented.

6.3.1   Downward-Looking Camera Model

The camera used by the localization system is an analog B/W camera. It provides a large underwater field of view (about 57° in width by 43° in height). We have considered a pinhole camera model, in which a first order radial distortion has been considered. This model is based on the projective geometry and relates a three-dimensional position in the space with a two-dimensional position in the image plane, see Figure 6.7. The equations of the model are the following:

$$\frac{^C X}{^C Z} = \frac{(x_p - u_0)(1 + k_1 r^2)}{f k_u} \tag{6.1}$$

$$\frac{^C Y}{^C Z} = \frac{(y_p - v_0)(1 + k_1 r^2)}{f k_v} \tag{6.2}$$

$$r = \sqrt{\left(\frac{x_p - u_0}{k_u}\right)^2 + \left(\frac{y_p - v_0}{k_v}\right)^2} \tag{6.3}$$

where, $(^C X, ^C Y, ^C Z)$ are the coordinates of a point in the space with respect to the camera coordinate frame $\{C\}$ and $(^I x_p, ^I y_p)$ are the coordinates, measured in pixels, of this point projected in the image plane. And, as to intrinsic
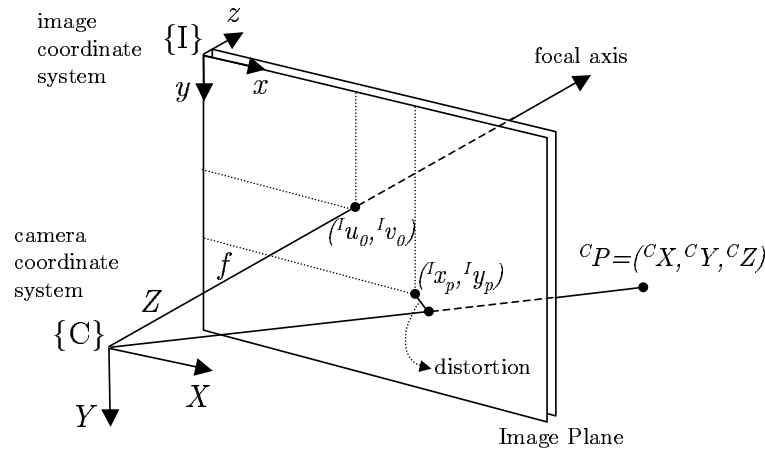
Figure 6.7: Camera projective geometry.

parameters of the camera, $(^{I}u_0,^{I}v_0)$ are the coordinates of the center of the image, (k_u,k_v) are the scaling factors, f is the focal distance and k_1 is the first order term of the radial distortion. Finally, r is the distance, in length units, between the projection of the point and the center of the image.

The calibration of the intrinsic parameters of the camera was done off-line using several representative images. In each of these images, a set of points were detected and its correspondent global position was found. Applying the Levenberg-Marquardt optimization algorithm [Gill et al., 1981], which is an iterative non-linear fitting method, the intrinsic parameters were estimated. Using these parameters, the radial distortion can be corrected, as can be seen in Figure 6.8. It can be appreciated how radial distortion is smaller for the pixels which are closer to the center of the image $(^{I}u_0,^{I}v_0)$.

6.3.2 Coded Pattern

The shape of the tank is a cylinder 4.5 meters in diameter and 1.2 meters in height. This environment allows the perfect movement of the vehicle along the horizontal plane and a restricted vertical movement of only 30 centimeters.

The main goal of the pattern is to provide a set of known global positions to estimate, by solving the projective geometry, the position and orientation of the underwater robot. The pattern is based on grey level colors and only round shapes appear on it to simplify the landmark detection, see Figure 6.9. Each one of these rounds or dots will become a global position used in the position estimation. Only three colors appear on the pattern, white as back-
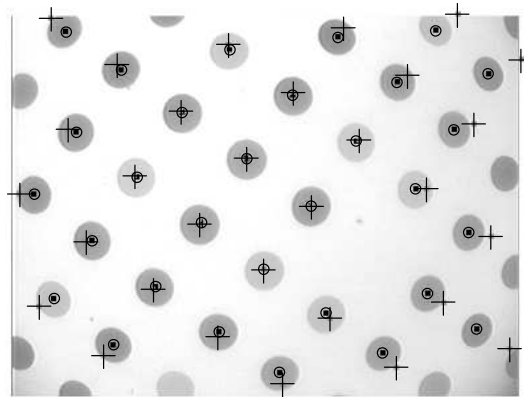
Figure 6.8: Acquired image in which the center of the dots has been marked with a round. After correcting the radial distortion the center of the dots has changed to the one marked with a cross.

ground, and grey or black in the dots. Again, the reduction of the color space was done to simplify the dot detection and to improve the robustness. The dots have been distributed throughout the pattern following the X and Y directions. All lines parallel to the X and Y axis are called the *main lines of the pattern*, see Figure 6.10. This term will be useful in the description of the algorithm used for localization, refer to Section 6.3.3.

The pattern contains some global marks which encode a unique global position. These marks are recognized by the absence of a dot surrounded by 8 dots, see Figures 6.9 and 6.10a. From the 8 dots surrounding the missing dot, 3 are used to find the orientation of the pattern and 5 to encode the global position. The 3 dots marking the orientation appear in all the global marks in the same position and with the same colors. In Figure 6.10a, these 3 dots are marked with the letter "o". In Figure 6.10b it can be seen how, depending on the position of these 3 dots, the direction of the X and Y axis can be detected.

The global position is encoded in the binary color (grey or black) of the 5 remaining dots. Figure 6.10a shows the position of these 5 dots and the methodology in which the global position is encoded. The maximum number of positions is 32. These global marks have been uniformly distributed throughout the pattern. A total number of 37 global marks have been used, repeating 5 codes in opposite positions on the pattern. The zones of the pattern that do not contain a global mark, have been filled with alternately black and grey dots, which help the tracking algorithm, as will be explained in Section 6.3.3.
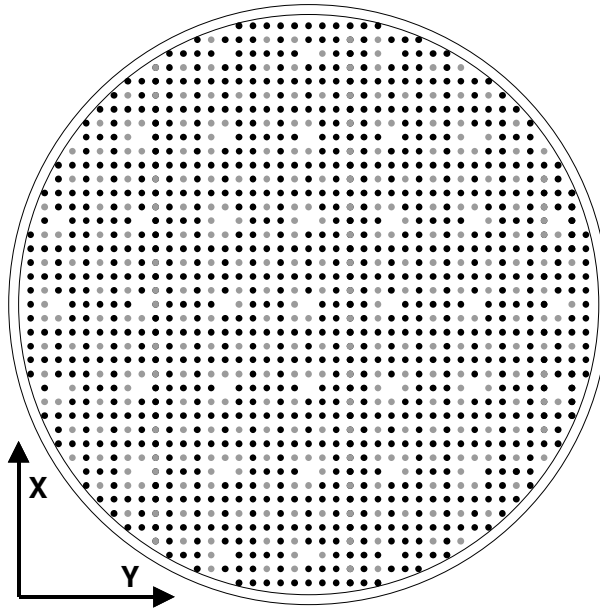
Figure 6.9: Coded pattern which covers the bottom of the water tank. The absence of a dot identifies a global mark.
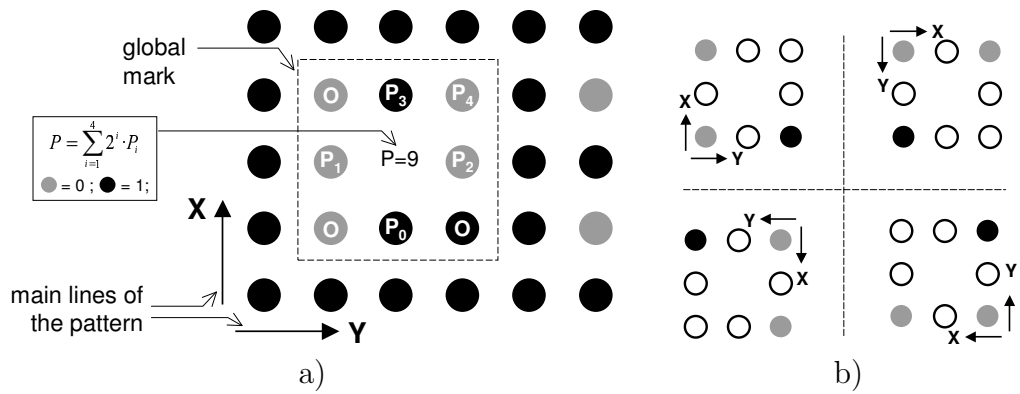
Figure 6.10: Features of the pattern, a) the main lines of the target and details about the absolute marks are shown, b) the three orientation dots of a global mark indicate the direction of the X and Y axis.

In order to decide the distance between two neighboring dots, several aspects were taken into account. A short distance would represent a higher number of dots appearing in the image and, therefore, a more accurate estimation of the vehicle's position. But, if a lot of dots appeared in the image while the vehicle was moving fast, dot tracking would be very hard or even impossible. On the other hand, a long distance between two neighboring dots would produce the opposite effect. Therefore, an intermediate distance was chosen for this particular application. The aspects which influenced the decision were the velocities and oscillations of the vehicle, the camera's field of view and the range of depths in which the vehicle can navigate. The distance between each neighboring dot finally chosen was 10 cm. The range of distances between the center of the robot and the pattern, used in the design are from 50 cm to 80 cm and the minimum number of dots which must be seen is 6, as will be described in the next subsection.

6.3.3   Localization Procedure

The vision-based localization algorithm was designed to work at 12.5 frames per second, half of the video frequency. Each iteration requires a set of sequential tasks starting from image acquisition to velocity estimation. The next subsections describe the phases which constitute the whole procedure.

Pattern Detection

The first phase of the localization algorithm consists of detecting the dots in the pattern. To accomplish this phase, a binarization is first applied to the acquired image, see Figure 6.11a and 6.11b. Due to the non-uniform sensitivity of the camera in its field of view, a correction of the pixel grey level values is performed before binarization. This correction is based on the illumination-reflectance model [Gonzalez and Woods, 1992] and provides a robust binarization of the pattern also under non-uniform lighting conditions.

Once the image is binarized, the algorithm finds the objects in the image. This task is accomplished by an algorithm which scans the entire image and for each pixel that has *white* color, it applies a region growing process. This process expands the region until the boundaries of the objects are found. Some features of the object, like the surface, the center, the boundaries and the aspect ratio, are calculated. Finally, the color of the pixels belonging to the object is changed to *black* and the scanning process is continued until no more *white* pixels are found. Some of the objects which do not fulfill a minimum and maximum surface, or do not have a correct aspect ratio, are
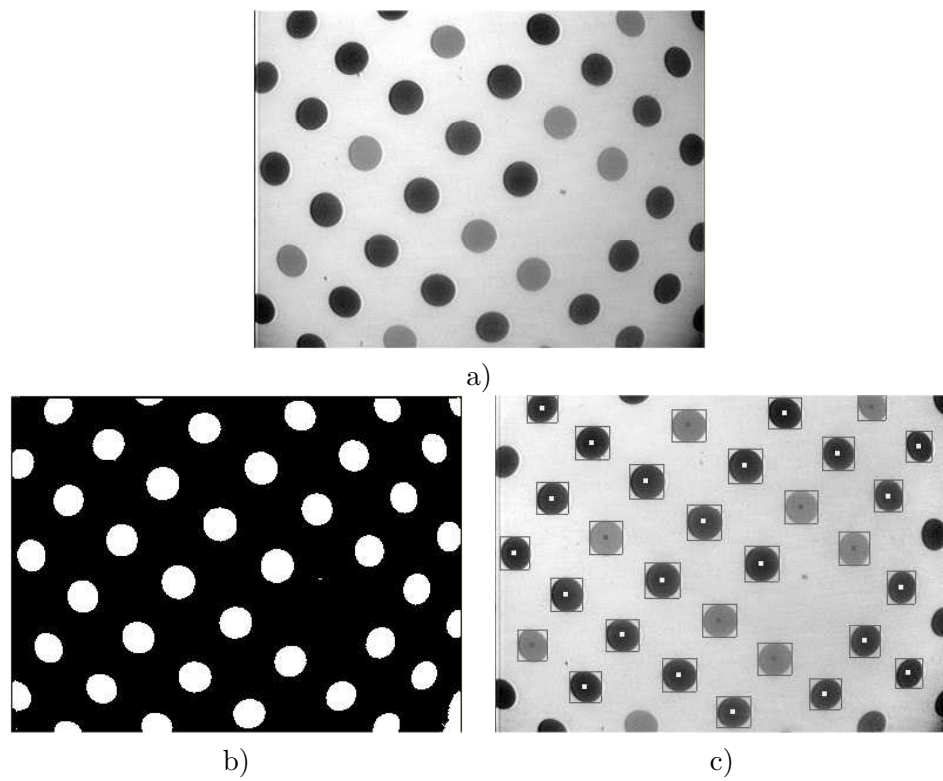
Figure 6.11: Detection of the pattern: a) acquired image, b) binarization, c) detection of the position, size and color of the dots.

dismissed. The other objects are considered to be one of the dots in the pattern.

Finally, for each detected dot and using the original image, the algorithm classifies its grey level, labelling them in three groups: *grey*, *black* or *unknown*. In the case of the label being unknown, the dot will be partially used in following phases, as Section 6.3.3 details. Figure 6.11c shows the original image with some marks on the detected dots. The rectangle containing each dot shows its boundaries. The color of the small point centered in each dot indicates the color which has been detected. If the color is white, the dot has been classified as *black*, if the color is dark grey the dot is *grey*, and if the color is light grey the dot is *unknown*. In the image shown in Figure 6.11c, only *black* and *grey* dots were found.

Dots Neighborhood

The next phase in the localization system consists of finding the neighborhood relation among the detected dots. The goal is to know which dot is next to which other dot. This will allow the calculation of the global position of all of them, starting from the position of only one. The next phase will consider how to find this initial position.

The first step in this phase is to compensate the radial distortion which affects the position of the detected dots in the image plane. Figure 6.8 has already shown the effect of the correction of the radial distortion. Another representation of the same image is shown in Figure 6.12a. In this Figure, the dots before the distortion compensation are marked in black and after the compensation in grey. The new position of the dots in the image is based on the ideal projective geometry. This means that lines in the real world appear as lines in the image. Using this property, and also by looking at relative distances and angles, the two *main lines* of the pattern are found. These two lines can also be seen in the figure. The two *main lines* of the pattern indicate the directions of the X and Y axis, although the correspondence between each main line and each axis it is not known. To detect the main lines, at least 6 dots must appear in the image.

The next step consists of finding the neighborhood of each dot. The algorithm starts from a central dot and goes over the others according to the direction of the main lines. To assign the neighborhood of all the dots, a recursive algorithm was developed which also uses distances and angles between dots. After assigning all the dots, a network joining all neighboring dots can be drawn (see Figure 6.12b).

a)   b)
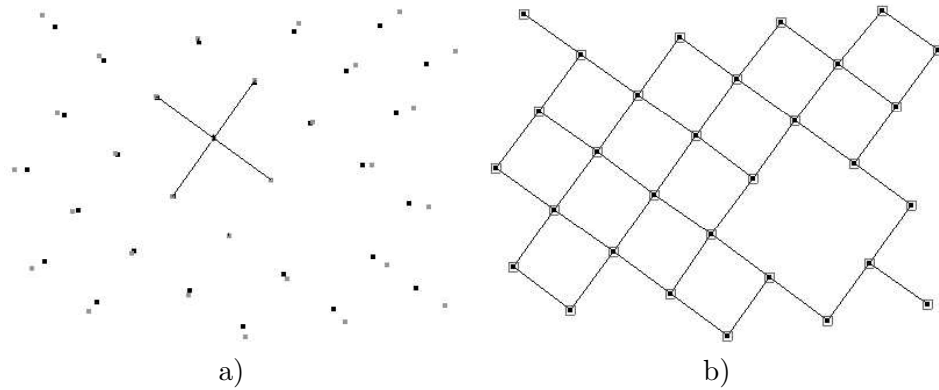
Figure 6.12: Finding the dots neighborhood: a) main lines of the pattern, b) extracted neighborhood.

Dots Global Position

Once the neighborhood of all the dots has been found, the global position of these points is required. Two methodologies are used to identify the global position of a subset of them. After these initial positions are known, the neighborhood network is used to calculate the position of all of these points.

The first methodology is used when a global mark is detected, see Figure 6.12b. The conditions for using a global mark are that a missing dot surrounded by 8 dots appears on the network and the color of these 8 dots is recognizable. In this case, the algorithm checks first the three orientation dots to find how the pattern is oriented. As showed in Figure 6.10b, the algorithm has to check how the pattern is oriented and, therefore, what are the directions of the X and Y axis. From the four possible orientations, only one matches the three colors. After that, the algorithm checks the five dots which encode a memorized global position, refer also to Figure 6.10a. Once the orientation and position of the global mark has been recognized, the algorithm calculates the position of all the detected dots.

The second methodology is used when no global marks appear on the image, or when there are dots of the global mark which have the color label *unknown*. It consists of *tracking* the dots from one image to the next. The dots which appear in a very small zone in two consecutive images are considered to be the same and, therefore, the global position of the dot is transferred. Refer to Figure 6.13 to see a graphical explanation of the tracking process. The high speed of the localization system, compared with the slow dynamics of the underwater vehicle, assures the tracking performance. The algorithm distinguishes between grey and black dots, improving the ro-

a)   b)
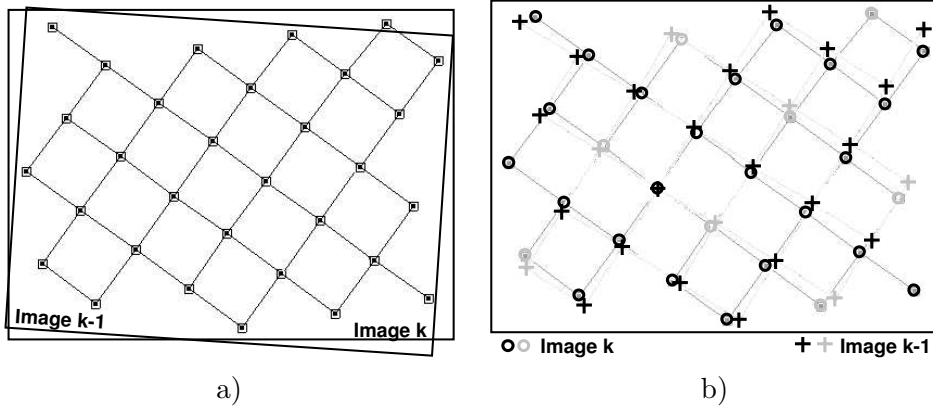
Figure 6.13: Tracking of dots: a) field of view of images k and $k-1$, b) superposition of the dots detected in images k and $k-1$. Dots with the same color which appear very close in two sequential images are considered to be the same dot.

bustness of the tracking. Moreover, since different dots are tracked at the same time, the transferred positions of these dots are compared, using the dot neighborhood, preventing possible mistakes.

Position and Orientation Estimation

Once the global positions of all the detected dots are known, the localization of the robot can be carried out. Equation 6.4 contains the homogeneous matrix which relates the position of one point $(^C X_i, ^C Y_i, ^C Z_i)$ with respect to the camera coordinate frame $\{C\}$, with the position of the same point with respect to the water tank coordinate frame $\{T\}$. The parameters of this matrix are the position $(^T X_C, ^T Y_C, ^T Z_C)$ and the rotation matrix of the camera with respect to $\{T\}$. The nine parameters of the orientation depend only on the values of *roll* (ϕ), *pitch* (θ) and *yaw* (ψ) angles. For abbreviation, the *cosine* and *sinus* operations have been substituted with "c" and "s" respectively.

$$
\begin{pmatrix} ^T X_i \\ ^T Y_i \\ ^T Z_i \\ 1 \end{pmatrix} = \begin{pmatrix} c\psi c\theta & -s\psi c\phi + c\psi s\theta s\phi & s\psi s\phi + c\psi s\theta c\phi & ^T X_C \\ s\psi c\theta & c\psi c\phi + s\psi s\theta s\phi & -c\psi s\phi + s\psi s\theta c\phi & ^T Y_C \\ -s\theta & c\theta s\phi & c\theta c\phi & ^T Z_C \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} ^C X_i \\ ^C Y_i \\ ^C Z_i \\ 1 \end{pmatrix}
$$

$$(6.4)$$

For each dot i, the position $(^TX_i, ^TY_i, ^TZ_i)$ is known, as well as the ratios:

$$\frac{^CX_i}{^CZ_i} = V_{ix} \text{ and } \frac{^CY_i}{^CZ_i} = V_{iy} \qquad (6.5)$$

which are extracted from Equations 6.1 and 6.2, and have been named V_{ix} and V_{iy}. The estimation of the state of the robot is accomplished in two phases. In the first phase, ^TZ_C, *roll* (ϕ) and *pitch* (θ) are estimated using the non-linear fitting method proposed by Levenberg-Marquardt [Gill et al., 1981]. This recursive method estimates the parameters which best fit in a non-linear equation from which a set of samples are known.

Given two dots i and j, the square of their distance is the same when computed with respect to $\{C\}$ or to $\{T\}$, see Equation 6.6. From this equation, ^CX_i and ^CY_i can be substituted applying Equation 6.5. After the substitution, Equation 6.7 is obtained, in which D_{ij} is the square of the distance between the two dots and is calculated with the right part of Equation 6.6.

$$(^CX_i - {}^CX_j)^2 + (^CY_i - {}^CY_j)^2 + (^CZ_i - {}^CZ_j)^2 =$$
$$(^TX_i - {}^TX_j)^2 + (^TY_i - {}^TY_j)^2 + (^TZ_i - {}^TZ_j)^2 \qquad (6.6)$$

$$((^CZ_iV_{ix} - {}^CZ_jV_{jx})^2 + (^CZ_iV_{iy} - {}^CZ_jV_{jy})^2 + (^CZ_i - {}^CZ_j)^2 = D_{ij} \qquad (6.7)$$

By rearranging the terms of Equation 6.7, Equation 6.8 can be written. This equation is the one to be optimized by the Levenberg-Marquardt algorithm. The unknowns of the equation are ^CZ_i and ^CZ_j. These values can be calculated using Equations 6.9 and 6.10, which have been extracted from Equation 6.4. The unknowns of Equations 6.9 and 6.10 are ^TZ_C, *roll* (ϕ) and *pitch* (θ). Therefore, from each pair of detected dots, one Equation 6.8 can be written depending only on the three unknowns ^TZ_C, *roll* (ϕ) and *pitch* (θ). The algorithm finds the values for these three unknowns which best fit with a certain number of equations. In order to compute a solution, the minimum number of equations must be the number of unknowns. Therefore, three equations, from three different dots, are required in order to have an estimation. Evidently, the higher the number of dots, the more accurate the estimations are.

$$^CZ_i^2(V_{ix}^2 + V_{iy}^2 + 1) + {}^CZ_j^2(V_{jx}^2 + V_{jy}^2 + 1)$$
$$-2\,{}^CZ_i\,{}^CZ_j(V_{ix}V_{jx} + V_{iy}V_{jy} + 1) = D_{ij} \qquad (6.8)$$

$$^C Z_i = \frac{^T Z_i - {}^T Z_C}{-s\theta V_{ix} + c\theta s\phi V_{iy} + c\theta c\phi} \tag{6.9}$$

$$^C Z_j = \frac{^T Z_j - {}^T Z_C}{-s\theta V_{jx} + c\theta s\phi V_{jy} + c\theta c\phi} \tag{6.10}$$

The second phase consists of estimating the $^T X_C$ and $^T Y_C$ positions and the *yaw* (ψ) angle. In this case, a linear least square technique is applied. This technique uses a set of linear equations to estimate a set of unknowns. The general form of the linear system can be seen in Equation 6.11. The $y(t)$ term is a vector which contains the independent terms of the linear equation. The $H(x(t), t)$ matrix contains the known values which multiply the unknown parameters contained in θ. The solution of the linear system can be easily computed applying Equation 6.12.

$$y(t) = H(x(t), t)\theta \tag{6.11}$$

$$\theta = (H^T H)^{-1} H^T y \tag{6.12}$$

The equations which contain the three unknowns ($^T X_C$, $^T Y_C$ and ψ) are included in Equation 6.4, and can be rewritten as Equations 6.13 and 6.14. Both equations can be applied to each dot i. The equations are non linear due to the *cosine* ($c\psi$) and *sinus* ($s\psi$) of the *yaw* angle. However, instead of considering the *yaw* as one unknown, each operation ($c\psi$ and $s\psi$) has been treated as an independent unknown. Equations 6.13 and 6.14 can be rewritten to Equations 6.15 and 6.16, which have the same structure as Equation 6.11. The number of files n of the terms $y(t)$ and $H(x(t), t)$ is the double of the number of detected dots, since for each dot there are two available equations. To solve the linear system, Equation 6.12 is applied obtaining the four unknowns, $c\psi, s\psi, {}^T X_C$ and $^T Y_C$. The *yaw* (ψ) angle is calculated applying the *atan2* operation.

$$\begin{aligned}
^T X_i = {} & (c\psi c\theta)\,^C X_i + \\
& (-s\psi c\phi + c\psi s\theta s\phi)\,^C Y_i + \\
& (s\psi s\phi + c\psi s\theta c\phi)\,^C Z_i + {}^T X_C
\end{aligned} \tag{6.13}$$

$$\begin{aligned}
^T Y_i = {} & (s\psi c\theta)\,^C X_i + \\
& (c\psi c\phi + s\psi s\theta s\phi)\,^C Y_i + \\
& (-c\psi s\phi + s\psi s\theta c\phi)\,^C Z_i + {}^T Y_C
\end{aligned} \tag{6.14}$$

$$
\begin{pmatrix}
{}^T X_1 \\
{}^T Y_1 \\
... \\
... \\
{}^T X_i \\
{}^T Y_i \\
... \\
... \\
{}^T X_n \\
{}^T Y_n
\end{pmatrix}
= H
\begin{pmatrix}
c\psi \\
s\psi \\
{}^T X_C \\
{}^T Y_C
\end{pmatrix}
\tag{6.15}
$$

$$
H =
\begin{pmatrix}
c\theta\, {}^C X_1 + (s\theta s\phi)\, {}^C Y_1 + (s\theta c\phi)\, {}^C Z_1 & -c\phi\, {}^C Y_1 + s\phi\, {}^C Z_1 & 1 & 0 \\
c\phi\, {}^C Y_1 - s\phi\, {}^C Z_1 & c\theta\, {}^C X_1 + (s\theta s\phi)\, {}^C Y_1 + (s\theta c\phi)\, {}^C Z_1 & 0 & 1 \\
... & ... & ... & ... \\
... & ... & ... & ... \\
c\theta\, {}^C X_i + (s\theta s\phi)\, {}^C Y_i + (s\theta c\phi)\, {}^C Z_i & -c\phi\, {}^C Y_i + s\phi\, {}^C Z_i & 1 & 0 \\
c\phi\, {}^C Y_i - s\phi\, {}^C Z_i & c\theta\, {}^C X_i + (s\theta s\phi)\, {}^C Y_i + (s\theta c\phi)\, {}^C Z_i & 0 & 1 \\
... & ... & ... & ... \\
... & ... & ... & ... \\
c\theta\, {}^C X_n + (s\theta s\phi)\, {}^C Y_n + (s\theta c\phi)\, {}^C Z_n & -c\phi\, {}^C Y_n + s\phi\, {}^C Z_n & 1 & 0 \\
c\phi\, {}^C Y_n - s\phi\, {}^C Z_n & c\theta\, {}^C X_n + (s\theta s\phi)\, {}^C Y_n + (s\theta c\phi)\, {}^C Z_j & 0 & 1
\end{pmatrix}
\tag{6.16}
$$

Once the three-dimensional position and orientation of the camera has been found, a simple translation is applied to find the position of the center of the robot. Figure 6.14 shows a representation of the robot position in the water tank. Also, the detected dots are marked on the pattern.

Filtering and Velocity Estimation

In the estimation of the position and the orientation, there is an inherent error. The main sources of this error are the simplifications, the quality of the systems and the uncertainty of some physical parameters. Refer to the next section for more detailed information about the accuracy of the system. Due to this error, small uncertainties about the vehicle position and orientation exist and cause some oscillations even if the robot is static. To eliminate these oscillations, a first order Savitzky-Golay [Savitzky and Golay, 1964] filter has been applied. This online filter uses a set of past non-filtered values to estimate the current filtered position or orientation.

Finally, the velocity of the robot with respect to the onboard coordinate frame $\{R\}$ is also estimated. A first order Savitzky-Golay filter with a first order derivative included is applied to the position and orientation values. This filter is also applied online and uses a window of past non-filtered samples. The output of the filtering process is directly the filtered velocity.
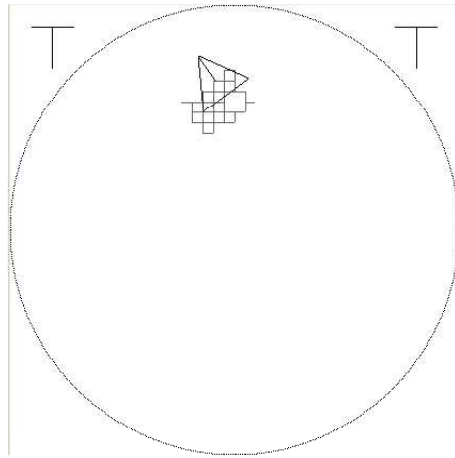
Figure 6.14: Estimated position and orientation of the robot in the water tank. The triangle indicates the *X, Y* and *Yaw* state of the robot. *Roll* and *Pitch* angles are indicated by two vertical lines at the top left and right corners respectively. Finally, the dots seen by the robot are also drawn.

After calculating the velocities with the Savitzky-Golay filter, a transformation from $\{T\}$ to $\{R\}$ coordinate frames is applied. Therefore, the position and orientation are referred to the water tank coordinate frame $\{T\}$, while the velocities are referred to the onboard coordinate frame $\{R\}$.

As usual when filtering a signal, an inherent delay will be added to the velocity or position. However, it has been verified that this small delay does not affect the low-level controller of the vehicle, as will be shown in Section 6.4.2. Figure 6.15 shows the estimated three-dimensional position and orientation with and without filtering, and also the velocities for the same trajectory.

6.3.4   Results and Accuracy of the System

The localization system offers a very accurate estimation of the position, orientation and velocity. The system is fully integrated on the vehicle's controller, providing measures at a frequency of 12.5 times per second. Because of the high accuracy of the system, other measures like the heading from a compass sensor or the depth from a pressure sensor are not needed. In addition, the localization system can be used to calibrate sensors, to validate other localization systems or to identify the dynamics of the vehicle. An example of a trajectory measured by the localization system can be seen in Figure 6.16.
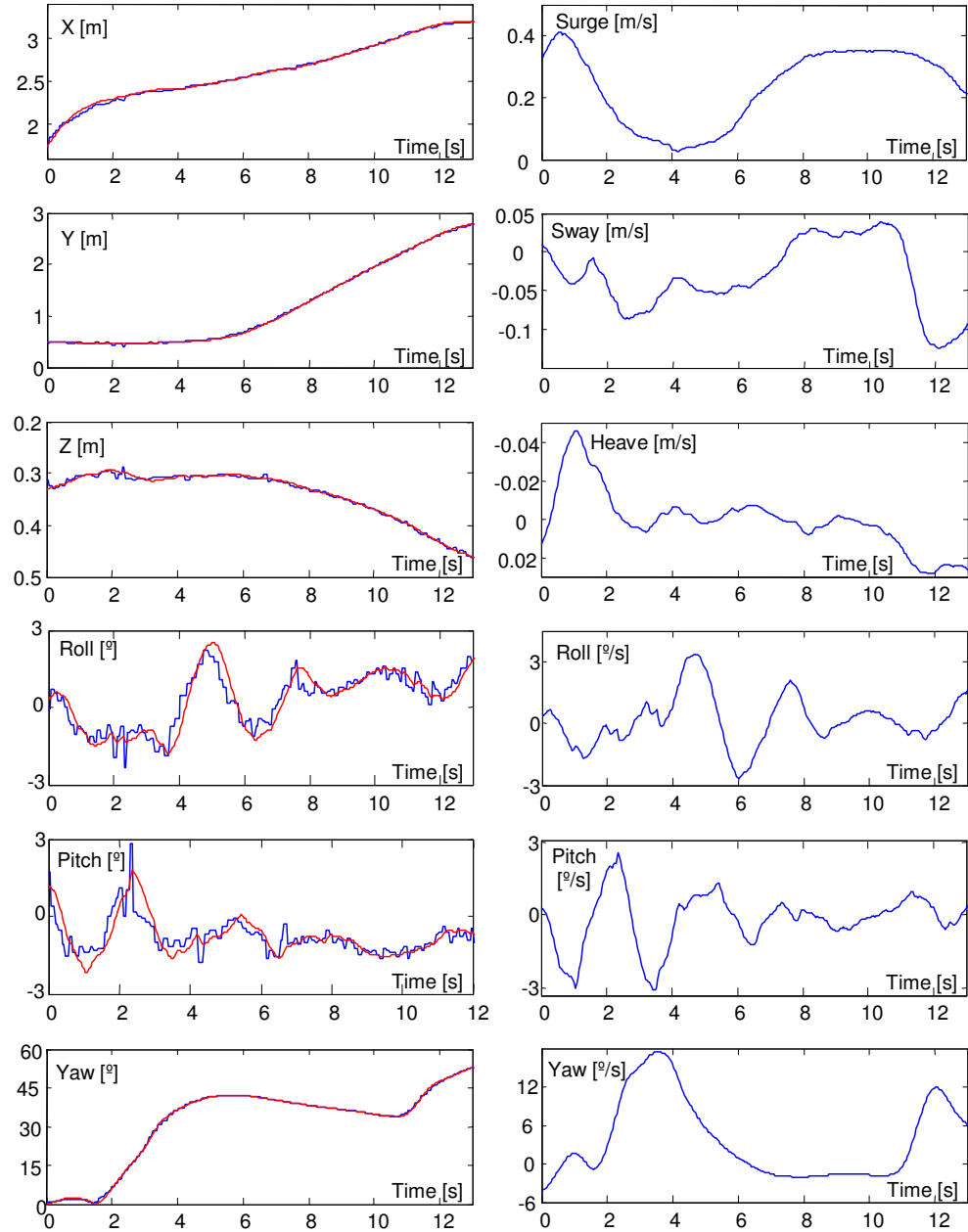
Figure 6.15: The left column shows the position and orientation before and after filtering during a trajectory. The right column shows the velocity for the 6 DOFs with respect to the on board coordinate frame $\{R\}$.
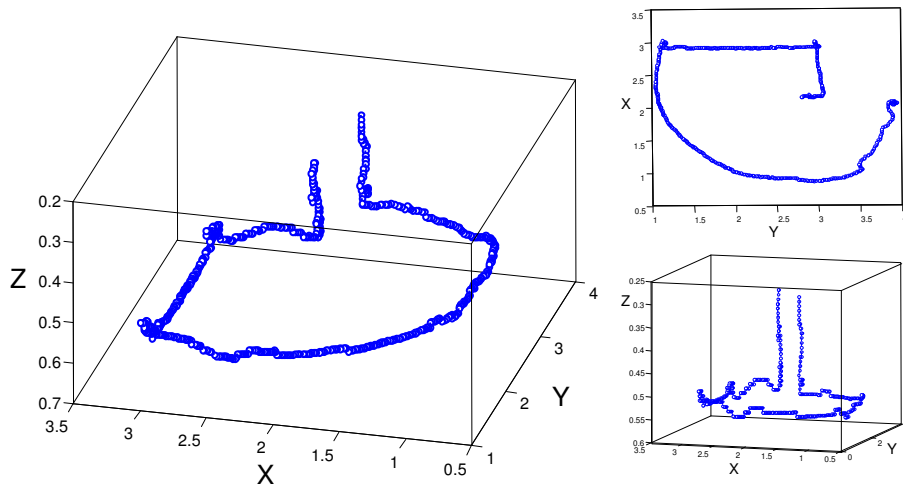
Figure 6.16: Three-dimensional trajectory measured by the localization system. Three views are shown.

In order to determine the accuracy of the system, the errors affecting the estimations have been studied. Main sources of error are the imperfections of the pattern, the simplification on the camera model, the intrinsic parameters of the camera, the accuracy in detecting the centers of the dots and the error of least-square and Levenberg-Marquardt algorithms on its estimations. It has been assumed that the localization system behaves as an aleatory process in which the mean of the estimates coincides with the real position of the robot. It is important to note that the system estimates the position knowing the global position of the dots seen by the camera. In normal conditions, the tracking of dots and the detection of global marks never fails, which means that there is no drift in the estimates. By normal conditions we mean, when the water and bottom of the pool are clean, and there is indirect light from the sun.

To find out the standard deviation of the estimates, the robot was placed in 5 different locations. In each location, the robot was completely static and a set of 2000 samples was taken. By normalizing the mean of each set to zero and grouping all the samples, a histogram can be plotted, see Fig. 6.17. From this data set, the standard deviation was calculated obtaining these values: $0.006[m]$ in X and Y, $0.003[m]$ in Z, $0.2[°]$ in *roll*, $0.5[°]$ in *pitch* and $0.2[°]$ in *yaw*.

The accuracy of the velocity estimations is also very high. These measurements are used by the low level controller of the vehicle which controls the *surge*, *heave*, *pitch* and *yaw* velocities.
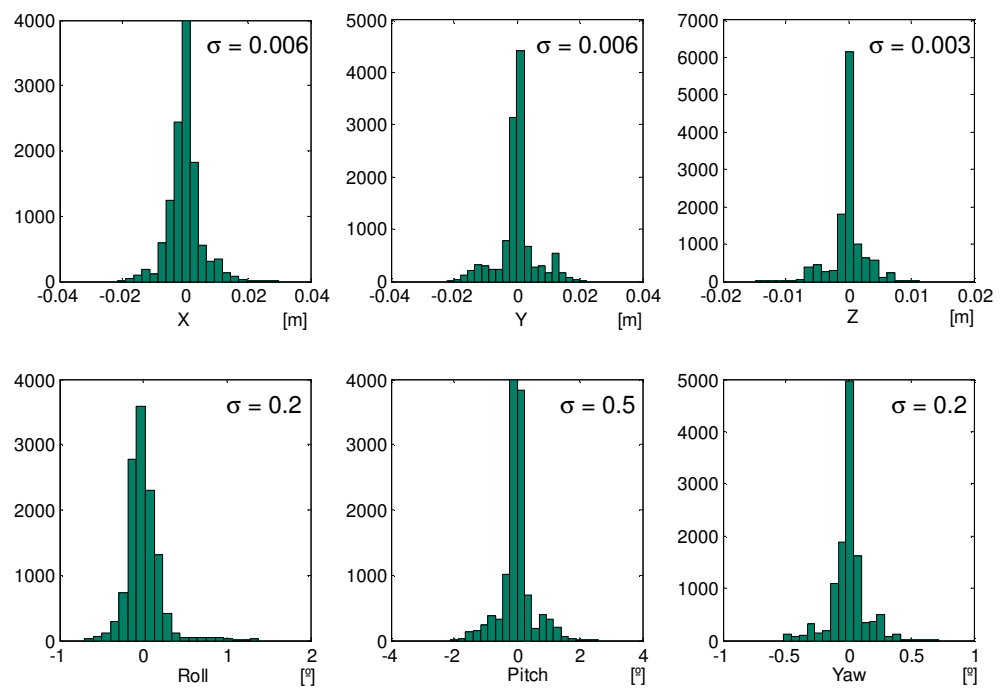
Figure 6.17: Histogram of the estimated position and orientation.

The only drawback of the system is the pattern detection when direct light from the sun causes shadows to appear in the image. In this case, the algorithm fails in detecting the dots. Any software improvement to make a more robust system in presence of shadows would increase the computational time and the time cycle of the algorithm would be too slow. However, the algorithm is able to detect these situations and the vehicle is stopped.

6.4   Software Architecture

In this section, the software architecture used to control URIS AUV is detailed. As will be described, a software framework was developed as a tool to easily implement the architecture needed to carry out a mission. After the description of this framework, the particular architecture used in the experiments of this thesis is detailed.

6.4.1   Distributed Object Oriented Framework

When working with physical systems such as an underwater robot, a real-time Operating System (OS) is usually required. The main advantage is better control of the CPU work. In a real-time OS, the scheduling of the processes to be executed by the CPU is done according to preemptive priorities. More priority processes will be first executed and will also advance processes which are already in execution. Using a correct priority policy it is possible to guarantee the frequency in which the control architecture has to be executed, which is very important to assure the controllability of the robot.

A software framework, based on a real-time operating system was specially designed for URIS AUV. In particular, QNX OS was used. This framework is intended to assist the architecture designers to build the software architecture required to carry out a particular mission with URIS AUV. The framework proposes the use of a set of distributed objects which represent the architecture. Each object represents a component of the robot (sensors or actuators), or a component of the control system (low-level or high-level controllers). An Interface Definition Language (IDL) is used to define the services which the object supports. From the information contained in the IDL, the object skeleton is automatically generated. Each object has usually two threads of execution. One of them, the *periodic thread*, is executed at a fixed sample time and is used to perform internal calculations. The other thread, the *requests thread*, is used to answer requests from clients.

The priority of each object thread is set independently and, depending on that, the objects will be executed. If a sample time is not accomplished,

a notification is produced. These notifications are used to redesign the architecture in order to accomplish the desired times.

The software framework allows the execution of the objects in different computers without any additional work for the architecture designer. A server name is used in order to find the location of all the objects. Evidently, objects that are referred to as physical devices, such as sensors or actuators, have to be executed in the computer which has the interfaces for them. Communication between objects is performed in different ways depending on whether they are executed sharing the same logical space and if they are executed in the same computer. However, these variations are hidden by the framework, which only shows a single communication system to the architecture designer.

Although this software framework was developed to work under a real-time operating system, the execution of objects under other conventional OS is also supported. The main reason for that is the lack of software drivers of some devices for the QNX OS.

6.4.2 Architecture Description

The software architecture used in the experiments presented in this thesis can be represented as a set of components or objects which interact among them. The objects which appear in the architecture can be grouped in three categories: actuators, sensors and controllers. A scheme of all the objects, with the connections between them, can be seen in Figure 6.18. The actuators' category contains the four *actuators* objects. The sensor category contains the *water detection*, *target tracking* and *localization* objects and, the controllers category contains the *low-level*, *high-level* and the two *SONQL behaviors*. It is important to remark on the difference between *low* and *high* level controllers. The low-level controller is in charge of directly actuating on the actuators of the robot to follow the movement set-points which the high-level controller generates. On the other hand, the high-level controller is responsible for the mission accomplishment and generates the set-points which the low-level controller has to reach.

All these objects are mainly executed in the on board embedded computer, but two external computers have also been used. The *on board computer* contains the actuators, controllers and sensory objects. As can be seen, not all the sensors presented in Section 6.1.3 have been used. One of the external computers, which has been called the *vision computer*, contains the two vision-based sensory systems, which are the target tracking (Section 6.2) and the localization system (Section 6.3). The other external computer, which has been called the *supervision computer*, is only used as
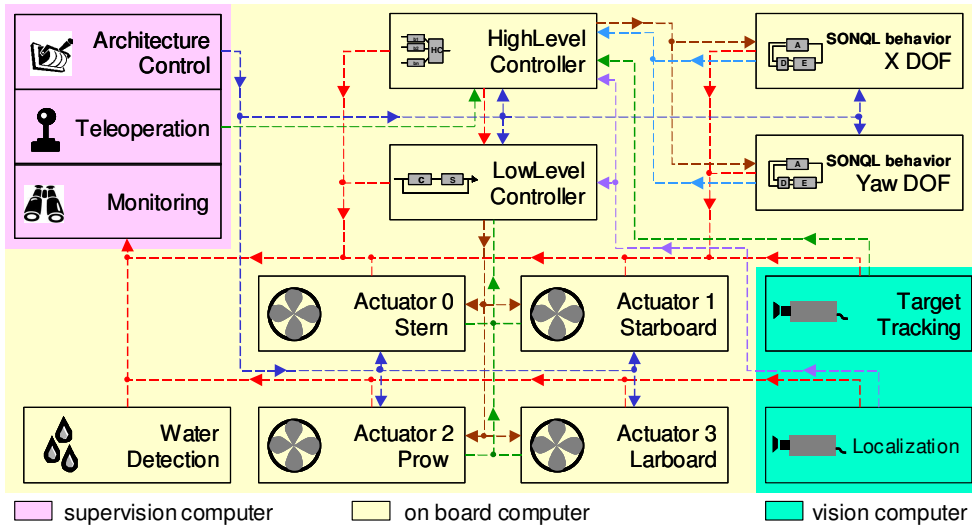
Figure 6.18: Objects which appear in the software architecture used in the experiments. The connections between the objects show the direction in which the information is transferred. The execution of the objects is performed in the on board computer or in the two external computers as indicated.
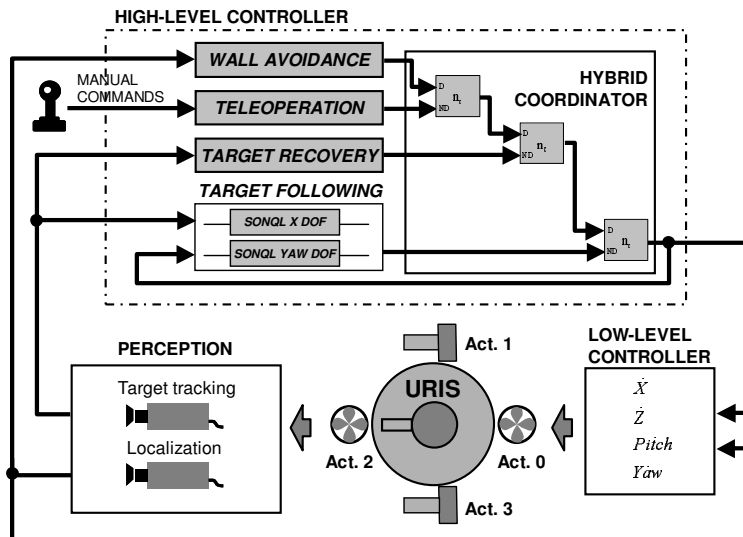
Figure 6.19: Components of the control system architecture used in the target following task.

a client of the previous objects. Its main goal is to control the operation of the objects and to monitor them. It is also used to send commands for robot teleoperation.

Another representation of the same architecture can be seen in Figure 6.19. In this case, the objects or components are seen from the point of view of the control system. In this representation, the sensing components, high-level controller, low-level controller and actuators are more clearly differentiated.

Hereafter, a more detailed description of each object is given. First, the objects belonging to the *onboard* embedded computer are reviewed. This computer is responsible for the control of the robot and runs the QNX real-time operating system.

- **Actuators**. The four thrusters of the robot are controlled through these four objects. These objects have only one thread for the requests from the clients. There is no internal calculation to perform. The services which these objects accept are used to modify the set-point velocity of the motor, enable/disable the thruster, and to read the real velocity and electric current of the motor.

- **Water Leakage Detection**. This object is used to measure the five water leakage sensors contained in the vehicle. The object has only one thread to attend to the requests from the clients. The only service which this object supports is to give the state of the five water sensors.

- **Low-Level Controller**. This object is in charge of the computation of the low-level controller of the vehicle. Its goal is to send control actions to the actuators to follow the set-points given by the high-level controller. Unlike the previous objects, a *periodic thread* is used to regularly recalculate and send these control actions. In particular, a sample time of 0.1 seconds is used. The DOFs to be controlled are four: *surge*, *heave*, *pitch* and *yaw*. The control system implemented for each DOF is a simple PID controller. The state which is controlled is the velocity of each DOF. The feedback measure of this velocity is obtained from the *localization* object. The four PID controllers are executed in parallel at each iteration and the control actions are superposed and sent to the actuators. In Figure 6.20 the performance of the *surge* and *yaw* controllers is shown.

  The low-level controller offers different services. The most important service is to receive the set-points. Three set-points are requested: the surge, the heave and the yaw velocities. The pitch velocity is not
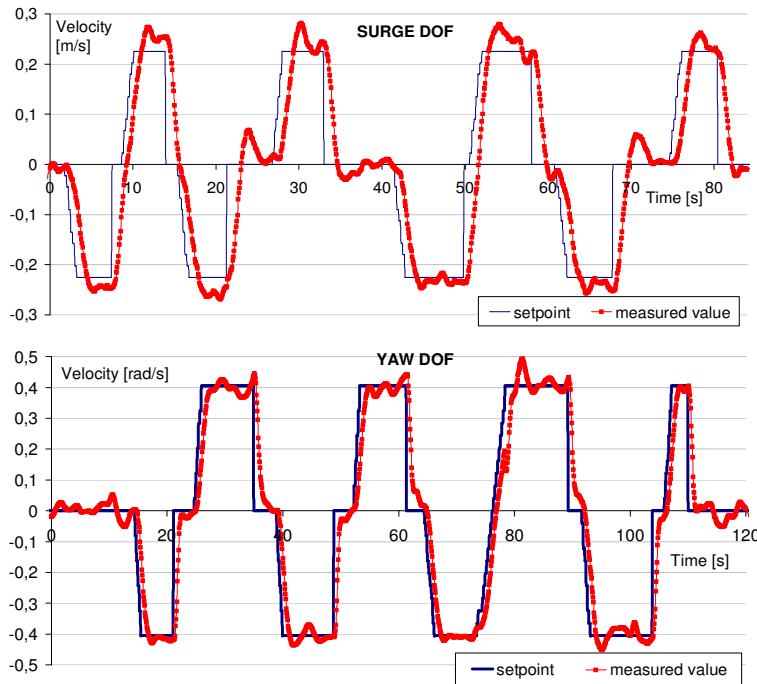
Figure 6.20: Performance of the surge and yaw velocity-based controllers.

requested as it is always considered to be zero. Other services are also available which are mainly used for the tuning of the controllers. By calling these services, the parameters of the controllers can be modified online, and internal variables can be checked. There is also a general service to enable/disable the controller.

- **High-Level Controller**. This object is the one which contains the behavior-based control architecture and, therefore, the part which is evaluated in this thesis. It contains the set of behaviors and the hybrid coordination system which has been designed to accomplish a particular task.

  Each behavior requests information from other objects, such as the target tracking object or the localization object, in order to generate its response. After the behaviors have been executed, the coordinator computes the final response and sends it to the low-level controller.

  As with the low-level controller, a *periodic thread* is used, but in this case with a sample time of 0.3 seconds. The services offered by this object allow us to enable/disable the controller and each behavior. Another service is in charge of receiving the response which a teleoperation

behavior will have. Using this service, an external user can command the robot.

- **SONQL-based Behaviors**. Although the behaviors are executed in the high-level controller, two auxiliary objects are used to implement the SONQL-based behaviors. Each one is used to learn a different DOF of the behavior. These objects contain the learning algorithm proposed in this thesis. The structure of the algorithm is divided in two threads. The *periodic thread* is in charge of updating the NN weights with the *learning samples database*, refer to section 5.7. The rest of the SONQL algorithm is executed every time a request is received.

  Each time the high-level controller has to calculate the response of a SONQL-based behavior, it sends to one of the SONQL-based behaviors the current and past states of the environment and the last taken action. For example, in the case of the X DOF of a target following behavior, it sends the current and past positions and velocities in X axis of the target, and the last taken action in *surge*. The SONQL-based behavior receives the states and action through the *requests thread*. This thread has a higher priority than the *periodic thread* and, therefore, the execution of the last one is stopped. Using the received state, the reinforcement is calculated completing a new *learning sample*. This sample is added to the *learning sample database*. Finally, an action is calculated and is returned to the high-level controller. The *requests thread* is finished, and the *periodic thread* starts again to update the NN weights but with the recently added *learning sample*.

  The services which these objects offer are basically to receive new *learning samples* and to configure all the parameters of the SONQL algorithm. It is also possible to reset the NN weights and all the variables.

The objects which will next be reviewed belong to the *vision computer*. This computer is one of the external computers and is run by the Windows operating system. The reason why a non real-time operating system has been used is because the frame grabbers, which are used to acquire images, do not provide software drivers for QNX OS. This computer was not located inside the robot because, at the time of the experiments, the embedded vision computer was not available.

- **Target Detection and Tracking**. This object implements the vision algorithm to detect and track an artificial target located in the water tank, as has been described in Section 6.2. The algorithm is executed in the *periodic thread*. In this case, the sample time of the thread is 0.08

seconds (12.5 Hz), which is double the sample time in which a video camera acquires a complete image. This means that the algorithm is applied once every images. The *requests thread* is used to send the last calculated position and velocity of the target.

- **Localization**. Similar to the previous object, the localization object implements the algorithm which estimates the position, orientation and velocity of the robot inside the water tank, refer to Section 6.3. The localization algorithm is also executed in the *periodic thread* at a sample time of 0.08 seconds. The *requests thread* is used to send the last calculated position and velocity of the robot.

The last components of the software architecture are the object clients which are contained in the *supervision computer*. The main uses of these clients are to control and monitor the architecture.

- **Architecture Control**. This component is used to enable and disable some of the objects of the architecture. Concretely, these objects are those belonging to the actuator and controller categories, refer to Figure 6.18.

- **Teleoperation**. As has been commented above, the high-level controller contains a teleoperation behavior which is commanded from the exterior. The teleoperation component then has the goal of getting the control command from a human and sending it to the high-level controller. This human-machine interface is accomplished by a joystick.

- **Monitoring**. Finally a monitoring component is in charge of consulting the state of all the objects through their services. This component will aid in the comprehension of the experiments.

Chapter 7

Experimental Results

This chapter presents the experimental results of this thesis. The chapter is organized in two parts. The first part describes the application of the behavior-based control layer to fulfill a robot task. This task consisted of following a target with the underwater robot URIS using the experimental set-up described in Chapter 6. The main goal of these experiments was to test the feasibility of the hybrid coordination system as well as the SONQL based behaviors. To accomplish this, a set of behaviors were designed, from which one was learnt with the SONQL algorithm while the other behaviors were manually implemented. A detailed description of the whole control system will be given and the results presented. The results will first focus on the SONQL algorithm, showing the learning of the X DOF, the Yaw DOF and some tests concerning the convergence of the algorithm. The hybrid coordination system will then be tested with the manual and learnt behaviors. The second part of this chapter shows the results of the SONQL algorithm in an RL benchmark. In this case, the task was used to test the generalization capability of the algorithm. The problem is called the "mountain-car" task and was executed in simulation. The suitability of the SONQL algorithm for solving the generalization problem will be stated.

7.1   Target-following task

The task consisted of following a target with the underwater robot URIS. Three basic aspects were considered. The first was to avoid obstacles in order to ensure the safety of the vehicle. In this case, the wall of the pool was the only obstacle. The second aspect was to ensure the presence of the target within the camera's field of view. The third aspect was to follow the target at a certain distance. Each one of these aspects was translated to a

robot behavior, hence, the behaviors were: the "wall avoidance" behavior, the "target recovery" behavior and the "target following" behavior. It is important to note the simplicity which behavior-based controllers offer. It is much simpler to design each behavior independently rather than developing a single control strategy to handle all of the considerations.

An additional behavior was included, the "teleoperation" behavior, which allowed the robot to move according to the commands given by a human. This behavior did not influence the outcome of the target following task, but was used to test the performance of the system, for example, by moving the vehicle away from the target.

Due to the shallow water in the tank in which the experiments were performed, only the motions on the horizontal plane were considered. Therefore, to accomplish the target following task, only the *surge* and *yaw* control actions were generated by the behaviors. The other two controllable DOFs (*heave* and *pitch*) were not used. In the case of the heave movement, the low-level controller maintained the robot at an intermediate depth. Regarding the pitch orientation, a zero degree set-point (normal position) was used.

Following the behavior-based control layer proposed in Chapter 3, each behavior b_i generated a response r_i, which was composed of an activation level a_i and a control action v_i. Since the heave movement was not present, the control action vector was $v_i = (v_{i,x}, 0, v_{i,yaw})$. The range used for the actions was $v_{i,j} = [-1, 1]$, corresponding to the maximum backward and forward velocity set-points in surge and yaw.

After defining the behaviors present in this task, the next step was to set their priorities. To determine these priorities, the importance of each behavior goal was ranked. This is the hierarchy used:

1. *"Wall Avoidance" behavior.* This was the highest priority behavior in order to ensure the safety of the robot even if the task was not accomplished.

2. *"Teleoperation" behavior.* This was given a higher priority than the next two behaviors in order to be able to drive the robot away from the target when desired.

3. *"Target Recovery" behavior.* This was given a higher priority than the target following behavior so as to be able to find the target when it was not detected.

4. *"Target Following" behavior.* This was the lowest priority behavior since it should only control the robot when the other behavior goals had been accomplished.
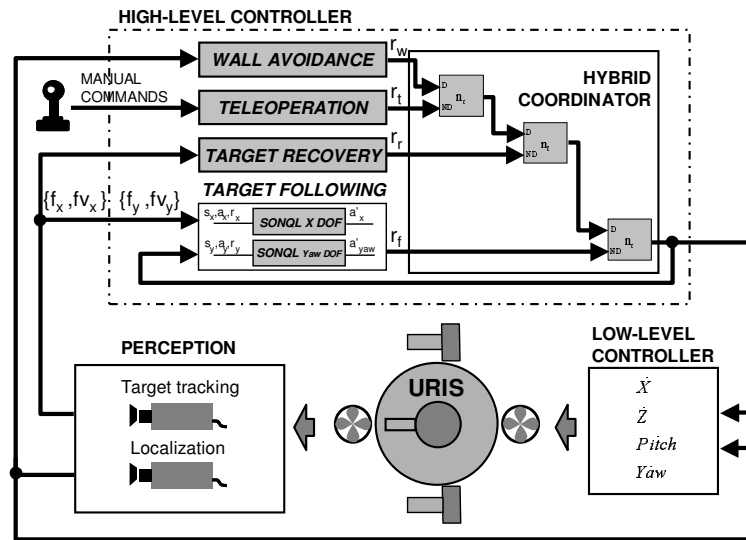
Figure 7.1: Implementation of the target following task with the proposed behavior-based control layer.

The establishment of the priorities allowed the composition of the behavior-based control layer. Figure 7.1 shows the implementation of the four behaviors using three hybrid coordination nodes. Finally, the last step was the definition of the internal state/action mapping and the activation level function for each behavior. The "target following" behavior was implemented with the SONQL algorithm. Its implementation and learning results will be described in the next section. As far as the other behaviors go, the main features were:

"Wall Avoidance" behavior This behavior was manually implemented. It used the absolute position of the robot as input, see Figure 7.1. The robot position was used to calculate the minimum distance to the circular wall of the water tank called d_w, see Figure 7.2. The behavior response r_w was computed according to this distance. The idea was to activate the behavior linearly with close proximity to the wall. To accomplish this, two threshold distances were set, $t_{w,min}$ and $t_{w,max}$, where $t_{w,min} < t_{w,max}$. If $d_w > t_{w,max}$, this meant the robot was too far from the wall and the behavior was not active ($a_w = 0$). On the other hand, when $d_w < t_{w,min}$, the robot was too close to the wall. Therefore, the activation was set at $a_w = 1$ and the coordinator actuated competitively in order to restore the safety of the robot. Finally, if $t_{w,min} < d_w < t_{w,max}$, this meant the robot was close to the wall
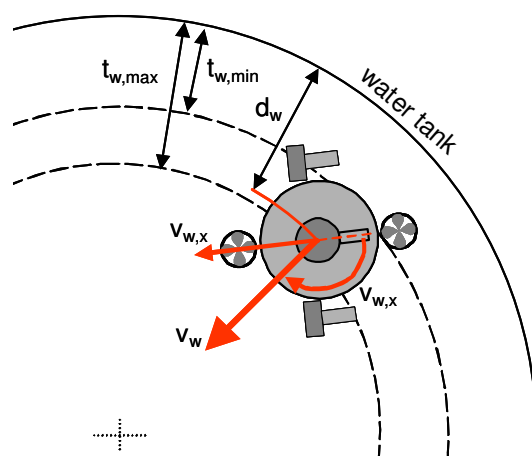
Figure 7.2: Schema of the "wall avoidance" behavior. The control action and the zones where the behavior is active are shown.

but not in a dangerous position and the activation was linearly calculated between 0 and 1. The control action v_w was only calculated when the activation was larger than 0. In that case, in order to drive the robot away from the wall, the control action was set to point towards the center of the circumference. According to the vehicle's yaw angle, this two-dimensional control action was split in the surge and yaw movements. The implementation of the "wall avoidance" behavior was designed experimentally with the thresholds and other parameters also obtained experimentally.

"Teleoperation" behavior The "teleoperation behavior" had to reflect the commands given by a joystick module. The teleoperation response r_t was composed of the activation level a_t and the control action vector v_i. The activation level of this behavior was 1 when the joystick was activated to send commands, otherwise it was set to 0. Moreover, the control actions were taken directly from the joystick. The surge movement $v_{j,x}$ corresponded to the forward/backward joystick command and the yaw movement $v_{j,yaw}$ corresponded to the left/right command.

"Target Recovery" behavior This behavior was also manually implemented but was much simpler than the "wall avoidance" behavior. The input of this behavior was the target position in respect to the robot, see Figure 7.1. This behavior was active $a_r = 1$ only when the target was not detected. In that case, it generated a constant rotational movement $v_{r,yaw} = \pm 0.8$ which spun the robot in the direction in which the target

had last been detected.

To conclude, the hybrid coordinator was implemented with a quadratic parameter $k = 2$. This parameter assured the prevalence of higher priority behaviors, and was also experimentally set.

7.1.1 SONQL behavior

The "target following" behavior was learnt using two SONQL algorithms, one for each DOF. The goal of this algorithm was to learn the state/action mapping of this behavior. This mapping determined the movements the robot had to perform in order to locate itself at a certain distance from the target and pointing directly at it. In order to generate the robot response r_f, the activation level was $a_f = 1$ whenever the target was detected and $a_f = 0$ otherwise. Moreover, the control action v_f was generated by the SONQL algorithms. In particular, the SONQL algorithm of the X DOF generated the $v_{f,x}$ control action and the algorithm of the Yaw DOF generated the $v_{f,yaw}$. The next paragraphs will describe the state variables, the reinforcement function and the parameters of the SONQL algorithm used in each DOF.

Environment State

A reinforcement learning algorithm requires the observation of the complete environment state. If this observation is not complete or the signals are too corrupted with noise or delays, the convergence will not be possible. The state of the environment must therefore contain all the required measurements relating the target to the robot. The first measurement was the relative position between the target and the robot: f_x in X DOF and f_y in Yaw DOF. The procedure to compute these continuous values was described in Section 6.2. A second necessary measurement was the relative velocity between the target and the robot. For instance, let's consider the case when the target is in front of the robot but there is a relative angular velocity between them. If the algorithm decides to execute the action $v_{f,yaw} = 0$, in the next step, the target will not be directly in front of the robot. On the contrary, if the relative velocity between them is zero and the executed action is again zero, in the next step, the target will still be located in front of the robot. Hence, we have two situations in which the target was located in the same place and the executed action was also the same, yet the system was brought to a different state. Since the environment is considered to be deterministic, this pointed out that the relative velocity was also required to

differentiate both states. These measurement were: fv_x for the X DOF and fv_y for the Yaw DOF.

Finally, if the target and the robot are both rotating at a velocity which makes the relative position and relative velocity equal to zero, a new measurement would be required to differentiate this state from the state in which everything is stopped and the target is directly in front of the robot. In this case, the absolute velocity of the robot or the absolute velocity of the target would also be required. This new measurement would be necessary to learn the behavior in case the target was moving. However, this case was not tested due to the complexity it would have represented in the exploration of a three-dimensional space. Instead, the behavior was learnt using a static target.

Besides the number of variables which composed the environment state, the quality of these variables was also very important. The target tracking system was designed to accurately estimate the relative positions and velocities. Indeed, the estimation of the relative velocity was especially difficult, as in the filtering process in which a delay was unavoidably added to the signal. Initially, this delay did not allow the system to learn, since the estimated velocities did not match the real movement. The filtering had to be accurately improved in order to remove part of this delay. Finally, it must be remembered that the state variables have been extracted from a vision-based system in which the non-linear distortions were not corrected. Moreover, the f_x measure is non-linear with respect to the relative distance to the target. All these non-linear effects, which were consciously not corrected, should not pose a problem for the learning algorithm since a Markovian environment does not imply linearity.

In summary, the state for the X DOF of the SONQL algorithm was composed of f_x and fv_x; and the YAW DOF of the SONQL algorithm was composed of f_y and fv_y. Figure 7.1 shows these measurements as inputs for the "target following" behavior and Figure 7.3 shows a two-dimensional representation of the f_x and f_y axes.

Reinforcement Function

The reinforcement function is the only information a designer must introduce in order to determine the goal of the behavior. The reinforcement function must be a deterministic function which relates the state and action spaces to a reward value. In the case of the "target following" behavior, the reinforcement function took only the target position as input. Therefore, according to these variables, f_x in the X DOF and f_y in the Yaw DOF, a reward value was given. Only three reward values were used: -1, 0 and 1, simplifying the
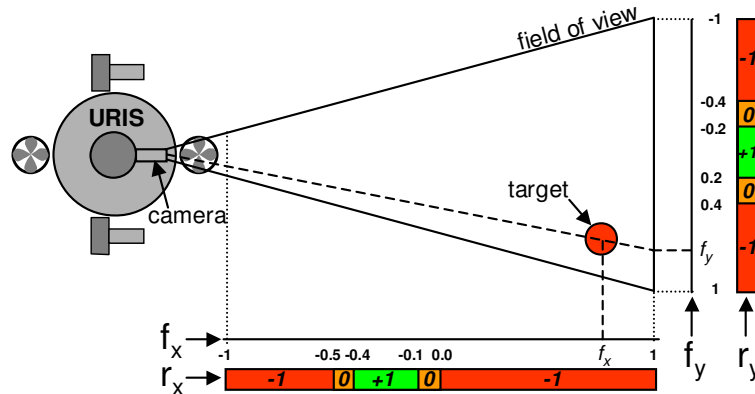
Figure 7.3: Relative positions f_x and f_y and the reinforcement values r_x and r_y, for the "target following" behavior.

implementation of this function. Figure 7.3 shows the reward values according to the target relative position. Basically, in order to maintain the target in front of the robot, the positive rewards $r = 1$ were given when the position of the target was around $f_y = 0$ and at a certain distance from the robot in X axis, around $f_x = -0.25$. The other reward values were progressively given if the target was farther. The r_x variable was used for the X DOF and the r_y variable for the Yaw DOF. As can be observed, the reward functions change the values at some thresholds, which were found experimentally.

SONQL parameters

After defining the states, actions and reinforcement function, the final step to learn the "target following" behavior with the SONQL algorithm consisted of setting the parameters of the algorithm. The same parameter values were used for both DOFs. The values of the parameter and some considerations are described, as follows:

NN configuration. The Neural Network had 3 continuous variables as inputs: $\{f_x, fv_x, v_{f,x}\}$ for the X DOF and $\{f_y, fv_y, v_{f,yaw}\}$ for the Yaw DOF. The output was the estimated $Q(s, a)$ value. Only one hidden layer was used with 6 neurons. This configuration was found experimentally and used for both DOFs. Different configurations were also tested, although this one provided the best compromise between generalization capability and convergence time. The more neurons, the higher the generalization capability, but also, the higher number of learning iterations needed to converge.

Learning rate α. A diverse set of values were tested. The final learning rate was set to $\alpha = 0.1$, which demonstrated a fast and stable learning process.

Discount factor γ. The discount factor was set to $\gamma = 0.9$. Since the learning of a robot behavior is a continuous task without a final state, a discount factor smaller than 1 was required. In the case of using $\gamma = 1.0$, the Q function values would increase or decrease, depending on the state/action zone, until they reach $-\infty$ or ∞. This value was chosen experimentally without exploring many values.

Exploration probability ϵ. The learning was performed with a $\epsilon - greedy$ policy. The exploration probability was $\epsilon = 0.3$. A smaller exploration probability increased the time required to converge, and a higher probability caused the robot to act too randomly. The value was also set experimentally.

Database Density Parameter t. The density parameter was set to $t = 0.5$. However, this parameter does not provide intuitive information about the number of learning samples. It indicates the minimum distance between two learning samples. This distance is calculated according to the vectors (s, a, r) of each sample, which, in this case was a four-dimensional vector since the state has two dimensions. In practice, the number of learning samples resulted in being less than 30, although it depended on the state/action space exploration. Several parameters were tested, concluding that with a larger value (less samples) the convergence was not always achieved due to the interference problem. Similar results were obtained for both DOFs.

7.1.2   Learning Results

An exhaustive set of experiments were carried out in order to test the SONQL algorithm. In these experiments, optimal parameters were found and a fast and effective learning of the robot behavior was achieved.

X DOF

The experiments in the X DOF were carried out by placing the target in front of the robot and starting the learning algorithm. The target was placed next to the wall of the water tank and the robot was placed in front of it at the center of the circular tank. This positioning gave the robot the maximum space in which to explore the state. In addition, as the target was placed
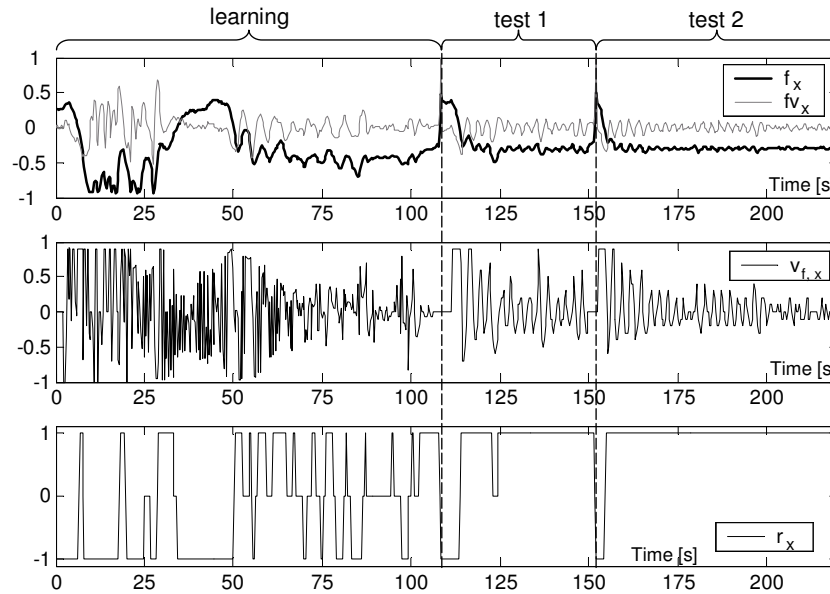
Figure 7.4: Real-time learning evolution and behavior testing in the X DOF. In the same experiment, the behavior was first learnt and then tested. In the learning phase, the state/action space exploration can be appreciated. The testing consisted of moving the robot away from the target first, and then allowing the behavior to reach it again. The evolution of the states, actions and rewards is shown.

close to the wall of the tank, the vehicle was stopped by the "wall avoidance" behavior when it came too close to the wall, preventing a collision with the target.

The learning of the X DOF took about 110 seconds, which represents about 370 learning iterations (sample time = 0.3 seconds). Figure 7.4 shows a typical real-time learning evolution. It can be seen how the robot explored the state in the learning phase. Immediately after the learning phase the behavior was tested by applying, with the "teleoperation" behavior, an action which moved the vehicle away from the target. It can be seen how the target was again reached and the maximum rewards achieved.

The policy learnt after this learning can be seen in Figure 7.5. The optimal action $v_{f,x}$ according to the state $\{f_x, fv_x\}$ can be appreciated. For example for $f_x = 1$ and $fv_x = 0$, which means that the target is at the farthest distance and the velocity is zero, the optimal action is $v_{f,x} = 1$, that is, "go forward". This is a trivial case, however, but the policy also shows situations in which intermediate action values are given.
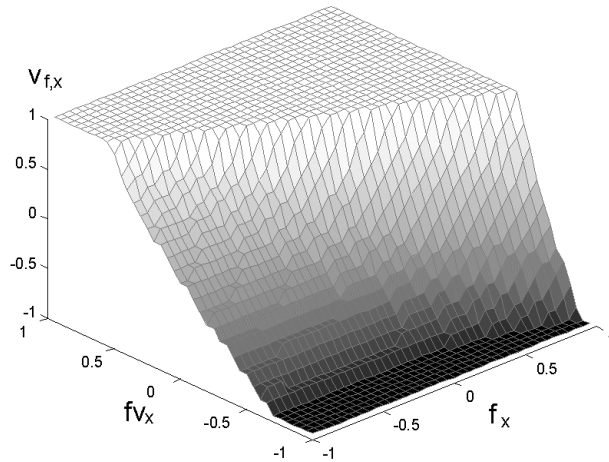
Figure 7.5: State/action policy learnt for the X DOF.

The Q-function learnt in one of the experiments can also be seen. Figure 7.6 shows the maximum Q value for each state $s = (f_x, fv_x)$ which is also the state value $V(s)$. It can be appreciated that the maximum values are in the target positions (f_x) where the reinforcement function was maximum. However, according to the velocity variable, fv_x, the value of zones with the same position f_x change. This is due to the prediction provided by the state value function. If the target is located at the position where the reward is maximum but the velocity causes it to move away, the value of this state is lower than the states in which the future rewards will also be maximum. Finally, it must be noted that the function approximated by the NN is not only the one shown in this figure. Another dimension, corresponding to the continuous action $v_{f,x}$, is also contained in the function. This demonstrates the high function approximation capability of NN.

YAW DOF

Similar experiments were performed for the Yaw DOF. The target was located in front of the robot when the SONQL was started. During the exploration of the space, the robot frequently lost the target but the "target recovery" behavior became active and the target was detected again.

The learning of the Yaw DOF took about 60 seconds, which represents about 200 learning iterations (sample time = 0.3 seconds). The learning of this DOF was faster than the learning of the X DOF since the state/action space to be explored was smaller. Figure 7.7 shows a typical real-time learning evolution. As happened with the X DOF, learning and testing phases
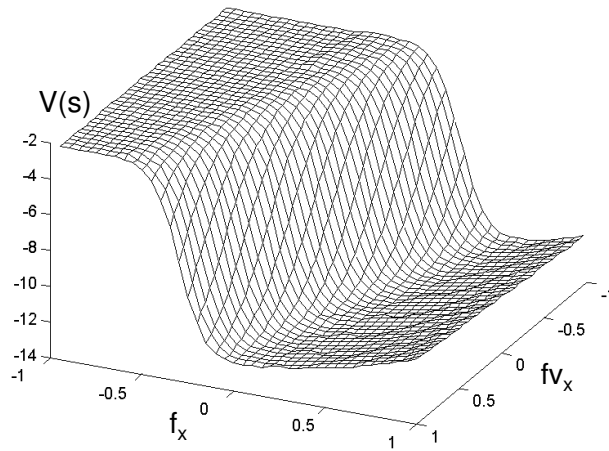
Figure 7.6: State value function, $V(s)$, after the learning for the X DOF.

were consecutively performed. During the testing phase, the "teleoperation" behavior was used to interfere with the "target following", causing the robot to loose the target. It can be seen how the target was always reached again and the maximum rewards were achieved.

The policy and state value functions for the Yaw DOF can be seen in Figure 7.8 and Figure 7.9 respectively. The policy relates the optimal action $v_{f,yaw}$ according to the environment state $\{f_y, fv_y\}$. As far as the state value function $V(s)$ is concerned, it can be clearly appreciated how the maximum values are in the target positions near the center ($f_y = 0$).

Convergence Test

The SONQL algorithm demonstrated that it converges to the optimal policy in a relatively small time, as commented above. The only condition to assure the convergence was to guarantee the reliability of the observed state. This means that perturbations like the influence of the umbilical cable had to be avoided. Figure 7.10 shows six consecutive learning attempts of the Yaw DOF. This figure also shows that the averaged reward increased, demonstrating that the behavior was being learnt. It can be seen that the algorithm started exploring the state in order to find the maximum reward. Once the whole state had been explored, the algorithm exploited the learnt Q_function and obtained the maximum reward.
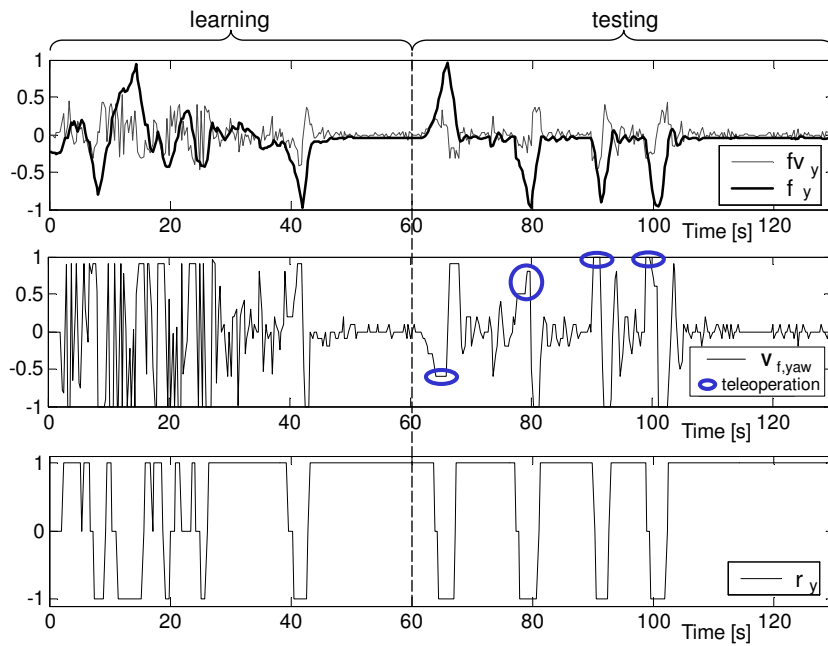
Figure 7.7: Real-time learning evolution and behavior testing of the Yaw
DOF. In the same experiment, the behavior was learnt first and then tested.
The testing consisted of moving the robot away from the target with the "tele-
operation" behavior and allowing the "target following" behavior to reach it
again. The evolution of the states, rewards and actions is shown.
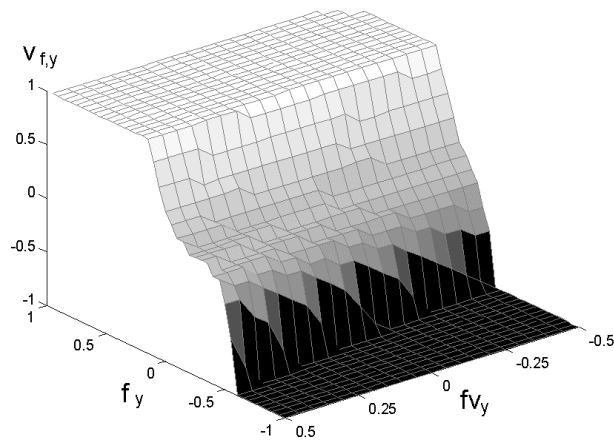
Figure 7.8: State/action policy learnt for the Yaw DOF.
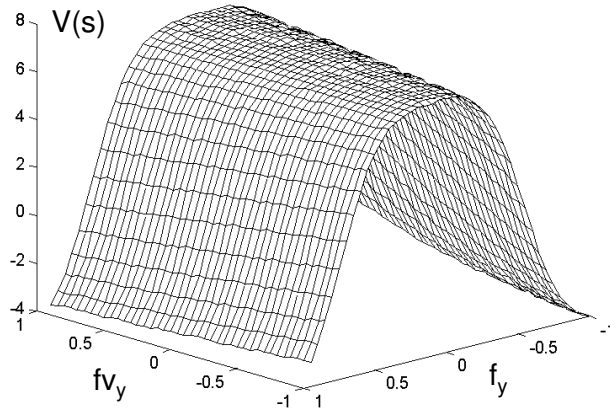
Figure 7.9: State value function $V(s)$, after the learning of the Yaw DOF.
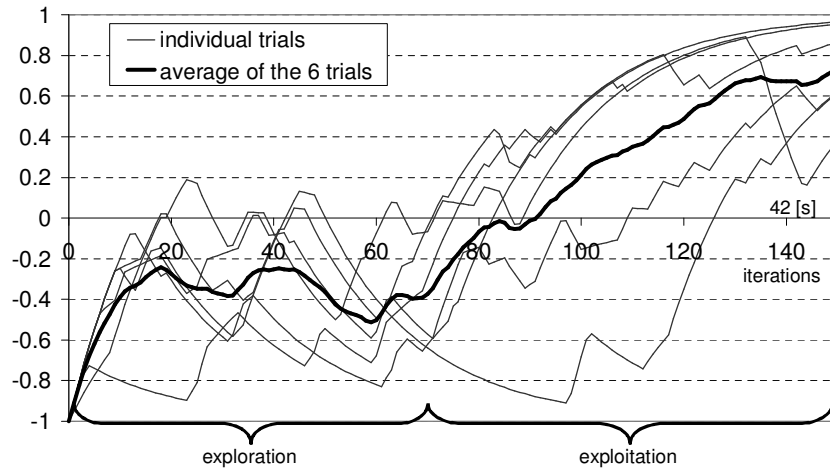
Figure 7.10: Behavior convergence for different attempts. The results of six different learning trials in the Yaw DOF are shown. Each graph represents the mean of the last 20 rewards. This representation is useful to verify that the algorithm is receiving the maximum rewards. The minimum and maximum values correspond to the minimum ($r_y = -1$) and maximum rewards ($r_y = 1$). Also, the average of the six experiments can be seen, pointing out that the accumulated rewards increased while the behavior was learnt.
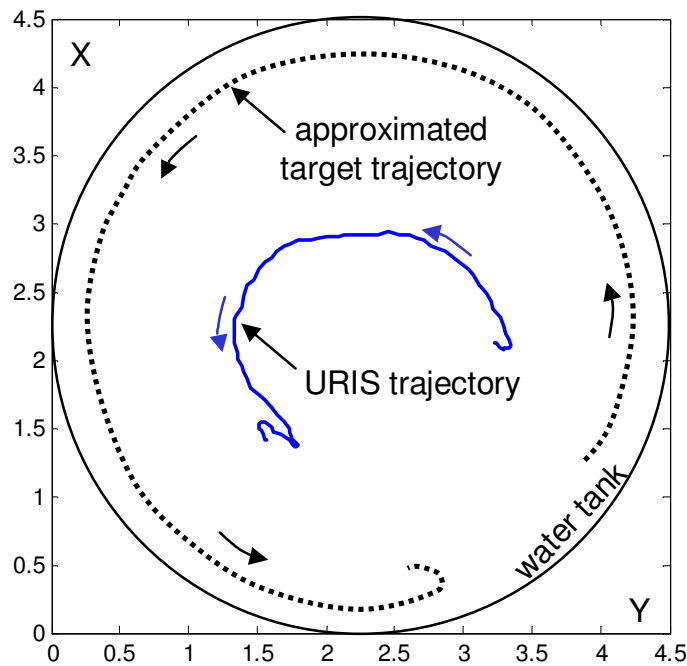
Figure 7.11: Trajectory of URIS while following the target in the water tank.

7.1.3   Task achievement

This section shows the performance of the behavior-based control layer in two situations. The first situation shows the accomplishment of the target following task, see Figure 7.11. In this figure, it can be seen how the robot followed the target at a certain distance. The target was moved manually around the water tank and, therefore, the robot trajectory was also a circumference. Note that the position of the target is approximate since there is no system to measure it. The behavior responses for this experiment are shown in Figure 7.12. It can be seen that at the beginning of the trajectory the target was not detected since the "target recovery" behavior was active. This behavior generated a constant yaw movement until the target was detected (see the Yaw DOF graph). Then, the "target following" behavior became active and generated several surge and yaw movements generating the circular trajectory (see X and Yaw graphs). Finally, the target was moved very fast, causing the activation of the "target recovery" behavior again.

The second situation of interest is the coordination of the "wall avoidance" and "target following" behaviors, see Figure 7.13. In this case, the target was used to push the robot backwards to the wall of the tank. The signals in this figure show how the "wall avoidance" behavior became active and stopped the
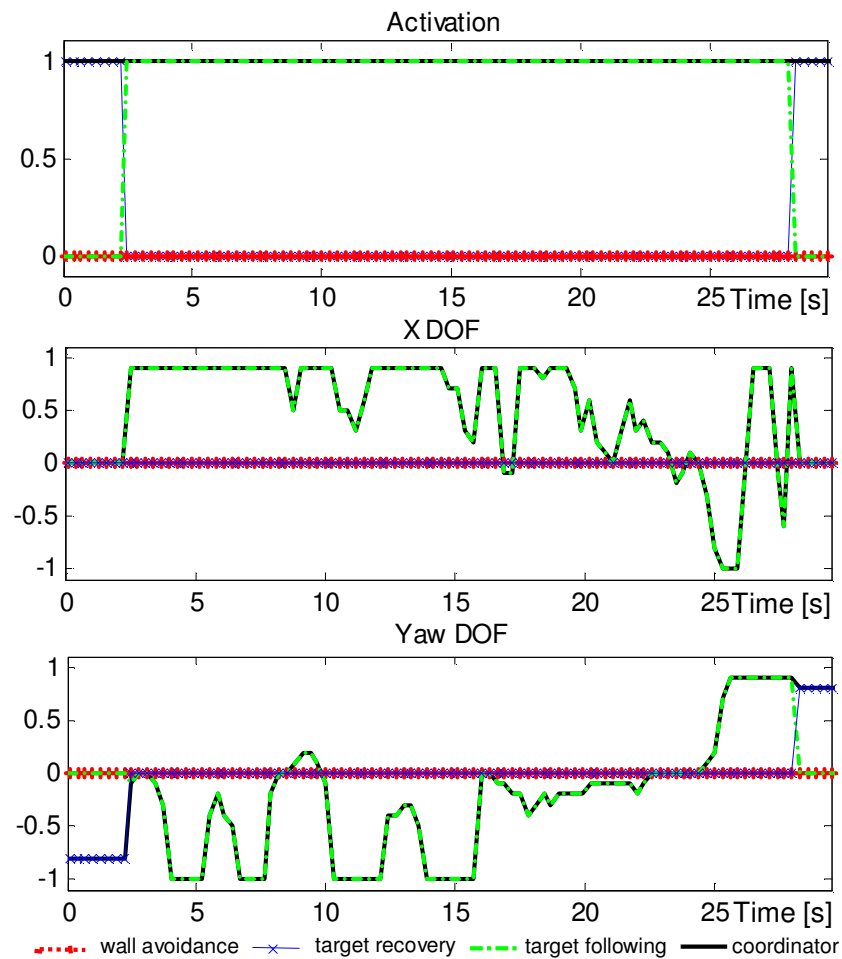
Figure 7.12: Activation levels and control actions for the "wall avoidance", "target recovery" and "target following" behaviors. The response of the coordinator is also shown. These signals correspond to the trajectory shown in Figure 7.11.

control action of the "target following" behavior. The coordinated response in the surge movement (see the X DOF graph) was nearly zero, although the "target following" behavior was generating a backward movement. The result was that the vehicle stopped between the target and the wall of the tank, thus producing the desired effect of the hybrid coordinator.

7.1.4 Conclusions and Discussion

After the presentation of the real experiments with URIS, some conclusions can be extracted:

- The behavior-based control layer and, in particular, the hybrid coordinator system is a suitable methodology to solve a robot task. The task must be previously analyzed and a set of behaviors with their priorities must be designed. The proposal is simple and actuates with robustness and good performance. The parameter k, used by the coordinator nodes, is not a parameter to tune and does not greatly affect the final performance. It simply determines the degree to which dominant behaviors will subsume the non-dominant ones.

- The design and implementation of some behaviors is sometimes very simple, like the "target recovery" behavior. However, other behaviors, like the "target following" behavior, require a deeper analysis. For this kind of behavior it is interesting to use the SONQL algorithm.

- The SONQL algorithm simplifies the designing of robot behaviors. The definition of the reinforcement function and the analysis of the observed state are the most important tasks. The parameters of the algorithm can first be taken from other behaviors and then refined if necessary. The SONQL will learn the optimal mapping between the state and the action, whatever the relation is.

- Another advantage of the SONQL algorithm is that it is an off-policy algorithm. This feature allows the interaction of the other behaviors while performing the learning. This is especially important in behavioral architectures where more than one behavior is simultaneously enabled.

- An RL learning algorithm should not be considered as a classic controller. The problem solved by RL is the maximization of the sum of future rewards. Therefore, an optimal mapping is the one which solves this problem. The optimal control action, according to control theory,
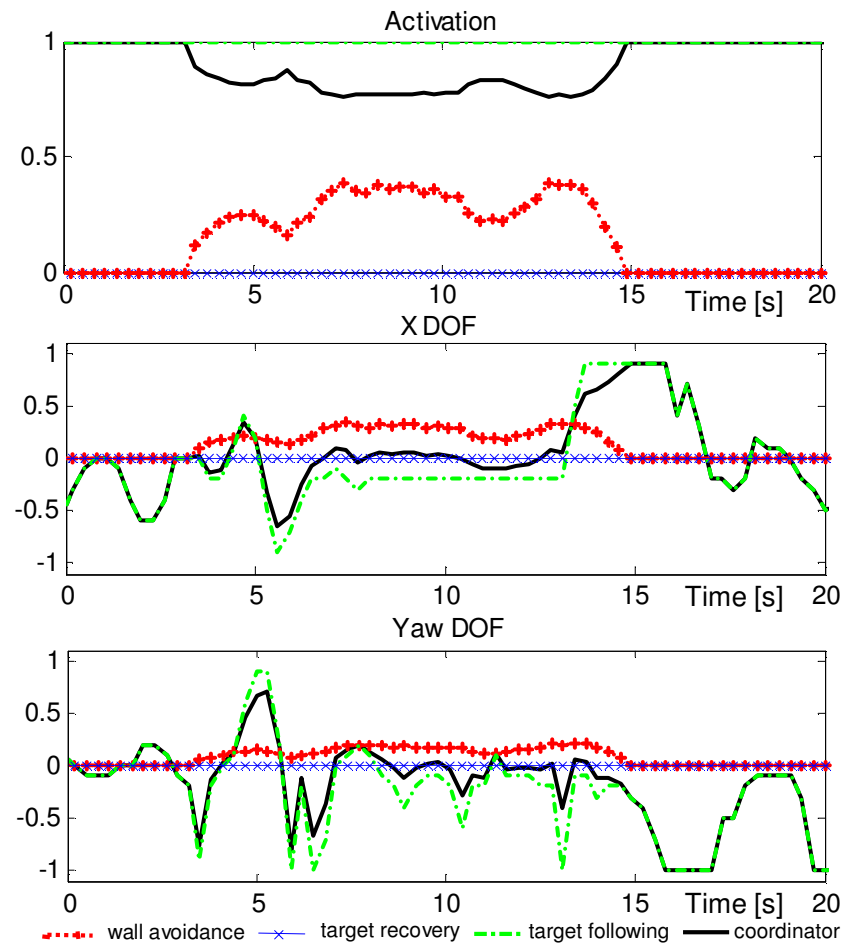
Figure 7.13: Performance of the hybrid coordination system. It can be seen how the "wall avoidance" behavior stopped the movement of the "target following" behavior. This experiment was achieved by bring the target closer to the robot, making the robot go backwards until the wall was found.

can only be achieved if the terms defining this optimality are considered in the design of the reinforcement function.

- One of the most important drawbacks is the accuracy required in the observation of the environment state. If the state is not fully observed, or has noise or delays, the learning cannot be accomplished. When selecting the components of the state, it is important to choose the variables which can be measured with more accuracy. Also, the measurement of the state will depend on the sample time of the algorithm. If the sample time is very small (fast execution), the state measurement must have a higher precision. The use of eligibility traces (see Section 4.4.3) could probably minimize this problem. Algorithms using eligibility traces update the values of a set of past states and this causes an average effect less sensitive to the poor accuracy of the state measurement. However, the implementation of eligibility in dealing with the generalization problem is much more complicated.

- Finally, an implementation issue concerning the action used in the learning process must be commented on. In the RL update rule, action a_t is the one proposed at time t and contributes to the achievement of state s_{t+1}. Surprisingly, when this action was used, the learning had many problems in converging. After analyzing the state transitions and the executed actions, it was found that there was a more logical state transition when considering action a_{t-1} for the change from s_t to s_{t+1}. By applying action a_{t-1}, the learning became much more stable due to the fast execution of the SONQL algorithm. As commented in Section 6.4.2, the high level controller is executed every 0.3 seconds, while the low-level controller id executed at 0.1 seconds. The low-level controller did not have enough time to achieve the set-points, and the actions were not effective until the next iteration. Again, this problem could also be solved with eligibility traces.

7.2   SONQL in the "Mountain-Car" task

This section presents the application of the SONQL algorithm to the "mountain-car" benchmark. This problem is widely accepted by the RL research community as a convenient benchmark to test the convergence and generalization capabilities of an RL algorithm. Although the convergence of the SONQL algorithm cannot be formally probed, it is assumed that if it is able to converge on this complex task, it will also be able to converge in simpler tasks, such as the reactive robot behavior at hand. The "mountain-car"

benchmark is not a continuous task like a robot behavior, but an episodic task. Moreover, contrary to a robot behavior, the environment is completely observable without or noise. Hence, this task is highly suitable to test the generalization capability of the SONQL only.

This section first describes the "mountain-car" task. Then, in order to have a performance baseline, the Q_learning algorithm is applied to the problem. After showing the performance of the Q_learning, the SONQL algorithm is applied. The results of the algorithm using different configurations will be analyzed. Finally, a comparison of the SONQL performance with respect to other RL algorithms is done.

7.2.1   The "Mountain-Car" task definition

The "mountain-car" task [Moore, 1991, Singh and Sutton, 1996] was designed to evaluate the generalization capability of RL algorithms. In this problem, a car has to reach the top of a hill. However, the car is not powerful enough to drive straight to the goal. Instead, it must first reverse up the opposite slope in order to accelerate, acquiring enough momentum to reach the goal. The states of the environment are two continuous variables, the position p and the velocity v of the car. The bounds of these variables are $-1.2 \leq p \leq 0.5$; and $-0.07 \leq v \leq 0.07$. Action a is a discrete variable with three values $\{-1, 0, +1\}$, which correspond to reverse thrust, no thrust and forward thrust respectively. The mountain geography is described by the equation: $altitude = sin(3p)$. Figure 7.14 shows the "mountain-car" scenario. The dynamics of the environment, which determines the state evolution, is defined by these two equations:

$$v_{t+1} = bound[v_t + 0.001\, a_t - 0.0025\, cos(3\, p_t)] \tag{7.1}$$

$$p_{t+1} = bound[p_t + v_{t+1}] \tag{7.2}$$

in which the *bound* operation maintains each variable within its allowed range. If p_{t+1} is smaller than its lower bound, then v_{t+1} is reset to zero. On the other hand, if p_{t+1} achieves its higher bound, the episode finishes since the task is accomplished. The reward is -1 everywhere except at the top of the hill, where it is 1. New episodes start at random positions and velocities and run until the goal has been reached or a maximum of 200 iterations have elapsed. The optimal state/action mapping to solve the "mountain-car" task is not trivial since, depending on the position and the velocity, a forward or reverse action must be applied.
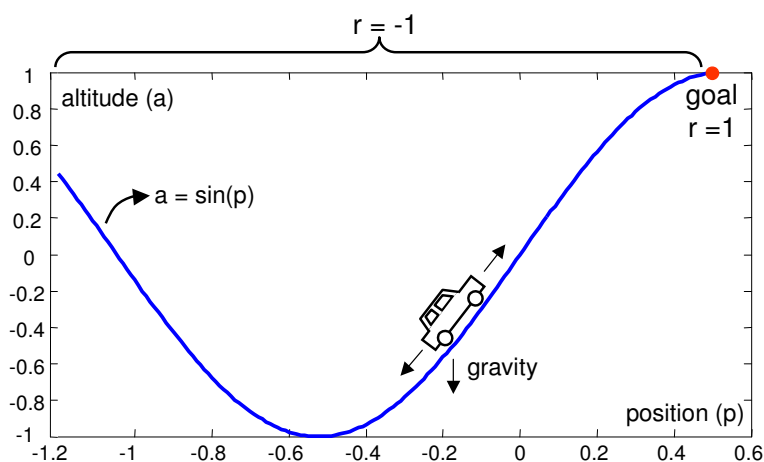
Figure 7.14: The "mountain-car" task domain.

7.2.2   Results with the Q_learning algorithm

To provide a performance baseline, the classic Q_learning algorithm was applied. The state space was finely discretized, using 180 states for the position and 150 for the velocity. The action space contained only three values, -1, 0 and 1. Therefore, the size of the Q table was 81000 positions. The exploration strategy was an $\epsilon - greedy$ policy with ϵ set at 30%. The discount factor was $\gamma = 0.95$ and the learning rate $\alpha = 0.5$, which were found experimentally. The Q table was randomly generated at the beginning of each experiment. In each experiment, a learning phase and an evaluation phase were repeatedly executed. In the learning phase, a certain number of iterations were executed, starting new episodes when it was necessary.

In the evaluation phase, 500 episodes were executed. The policy followed in this phase was the *greedy* policy, since only exploitation was desired. In order to numerically quantify the *effectiveness* of the learning, the average time spent in each episode is used. This time is measured as the number of iterations needed by the current policy to achieve the goal. After running 100 experiments with Q_learning, the average episode length in number of iterations once the optimal policy had been learnt was 50 iterations with 1.3 standard deviation. The number of learning iterations to learn this optimal policy was approximately 10^7. Figure 7.15 shows the effectiveness evolution of the Q_learning algorithm after different learning iterations.

It is interesting to compare this mark with other state/action policies. If a forward action ($a = 1$) is always applied, the average episode length is 86. If a random action is used, the average is 110, see Figure 7.15. These
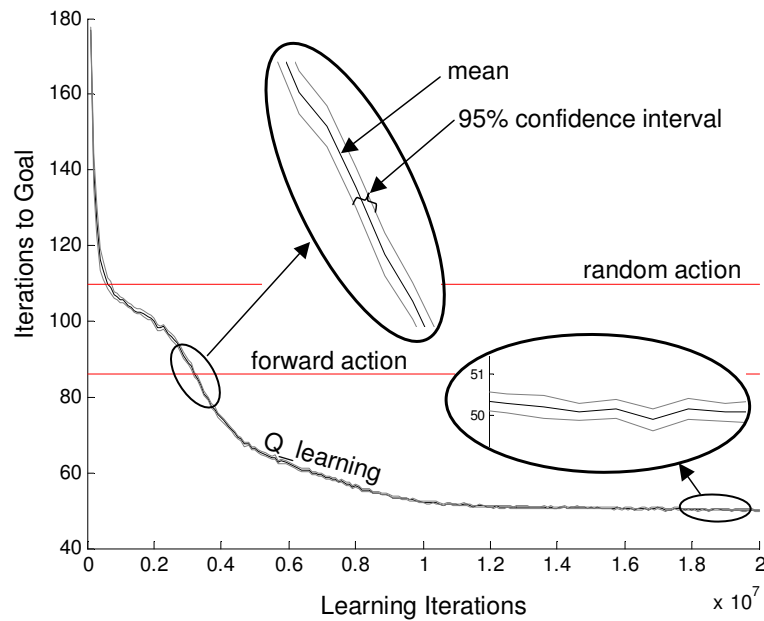
Figure 7.15: Effectiveness of the Q_learning algorithm with respect to the learning iterations. During the first iterations the efficiency was very low, requiring many iterations to reach the goal. The graph was obtained by averaging 100 trials. In each trial, the effectiveness was calculated by averaging the number of iterations to goal in 500 episodes. After converging, the effectiveness was maximum, requiring only 50 iterations to accomplish the goal. The 95% confidence intervals are also shown. Finally, the effectiveness levels of random and forward policies can be observed.

averages depend highly on the fact that the maximum number of iterations in an episode is 200, since in many episodes these policies do not fulfill the goal.

The optimal policy learnt by Q_learning is shown in Figure 7.16. This mapping relates the state of the environment (car position and velocity) with the discrete actions, $a = \{-1, 0, 1\}$. In this figure four different mappings are presented which correspond to the same trial but at different learning iterations. It can be observed how the optimal policy becomes more defined as a function of the learning iterations. It is important to note the non-linear relation between the mapping and the optimal action. The disadvantage of a discrete state space is that each state/action pair must be updated several times until a homogeneous policy is obtained. This causes a high number of learning iterations.
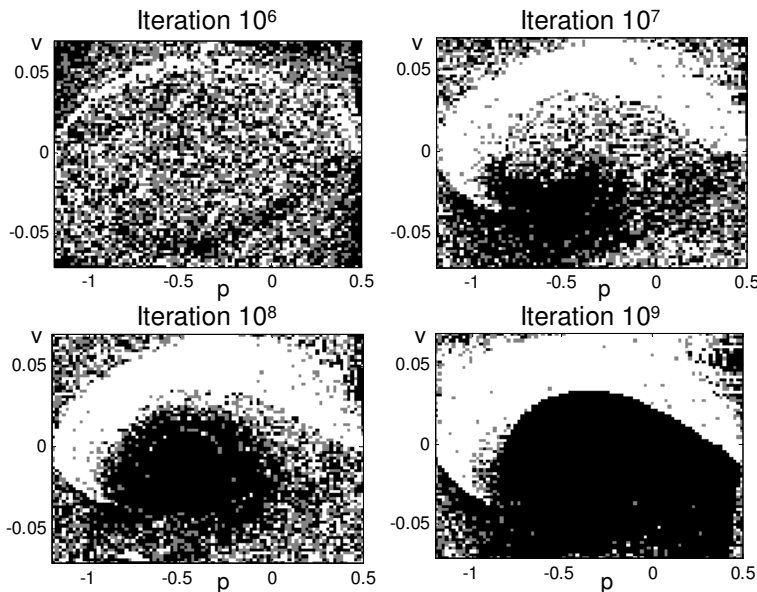
Figure 7.16: State/action policy after several number of learning iterations for the Q_learning algorithm. The actions are represented in different colors: white for forward thrust, gray for no-thrust, and black for reverse thrust.

Similar to the experiments with the URIS robot, the Q function can also be represented. In Figure 7.17, the maximum $Q(s, a)$ value for each state value is represented which is also the $V(s)$ function. For a clearer visualization, the $V(s)$ axis has been inverted. Hence, the states with a higher state-value are the ones which correspond to the lower parts of the three-dimensional surface. It can be observed how the shape of the state-value function evolves according to the learning iterations. Also, it is interesting to compare the evolution of the $V(s)$ function with respect to the evolution of the optimal policy. The $V(s)$ function evolves much faster to its definitive shape, while the policy is learnt slowly.

7.2.3   Results with the SONQL algorithm

The SONQL was also applied to the "mountain-car" task. Since the state space had been finely discretized with the Q_learning algorithm, it was assumed that with only three actions, the minimum number of iterations to fulfill the goal is 50. The SONQL algorithm cannot improve this mark as it is based on the Q_learning algorithm. However, it is expected that it can reduce the number of iterations required to learn the optimal policy.
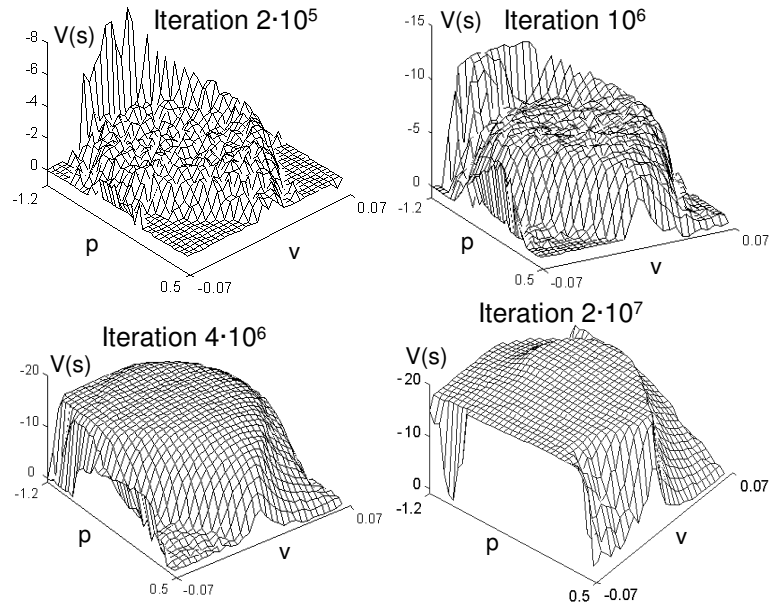
Figure 7.17: State value function $V(s)$ after different learning iterations for the Q_learning algorithm.

An extensive number of experiments were executed with the SONQL algorithm in order to find the best configuration. The NN had three layers with 15 neurons in the two hidden layers. Only three actions were used, as with the Q_learning experiments. The optimal learning rate and discount factor were $\alpha = 0.0001$ and $\gamma = 0.95$. And the ϵ parameter was set at 30%. Note the difference between the optimal learning rate of the SONQL algorithm and the Q_learning algorithm. The Q_learning has a higher rate but only one value of the discrete function is updated. On the other hand, the SONQL algorithm has a much smaller learning rate but each NN update affects the whole continuous space. The density parameter of the database was set to $t = 0.09$, which entailed approximately 470 learning samples at the end of each trial.

As with the Q_learning algorithm, each trial had a learning phase and an evaluation phase. In the evaluation phase 500 episodes were tested. For each experiment with a SONQL configuration, a total number of 100 trials were run. The average episode length in number of iterations for the parameters detailed above was 53 with 2.1 standard deviation. The number of learning iterations were only 20000, approximately. This result demonstrates that the SONQL algorithm is able to converge much faster than the Q_learning algorithm (from 10^7 to 20000 learning iterations), although it is not able to
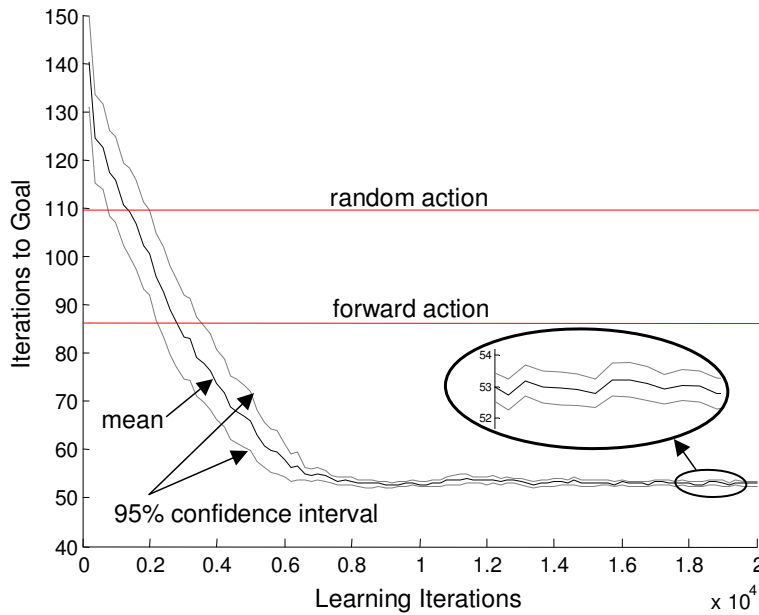
Figure 7.18: Effectiveness of the SONQL algorithm with respect to the learning iterations. The graph is the average of 100 trials. The 95% confidence intervals are also shown.

learn exactly the same optimal policy. The policy learnt by the Q_learning algorithm was able to solve the "mountain-car" task in 50 iterations, while the SONQL required 53 iterations. This difference is very small and, therefore, the feasibility of the SONQL algorithm in solving the generalization is affirmed. The convergence of the algorithm also proved to be very high and in all the experiments the optimal policy was learnt.

Figure 7.18 shows the performance evolution with respect to the number of learning iterations. After 20000 learning iterations, the number of iterations required to accomplish the goal are 53. It is also observed how the SONQL algorithm maintains effectiveness until the end of the experiment without diverging. Figure 7.19 shows another representation of the same experiment. In this case, the graph is drawn with respect to the total number of NN updates, considering each sample of the database as a learning iteration. The number of iterations is higher than before since, for each SONQL iteration, all the samples of the database are learnt. However, the number of NN updates are also significantly fewer than with the Q_learning algorithm, which is also represented. The total number of NN updates was approximately $8 \cdot 10^6$ iterations.

The optimal policy learnt by the SONQL algorithm is shown in Fig-
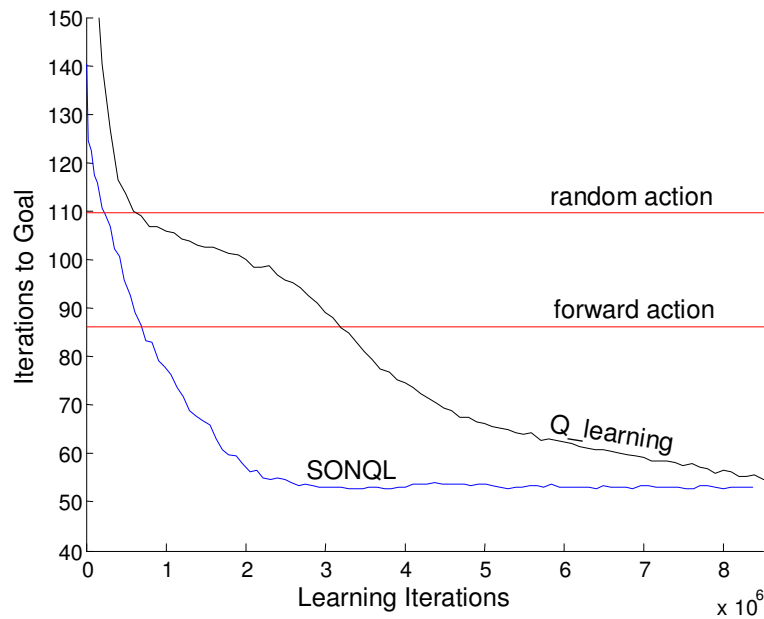
Figure 7.19: Effectiveness of the SONQL algorithm with respect to the number of NN updates.

ure 7.20. As previously done with the Q_learning algorithm, four different mappings are presented, which correspond to different learning iterations. It can be observed how the optimal policy becomes more defined as the learning is performed. It is interesting to compare the policies learnt by the two algorithms, see Figure 7.16 and Figure 7.20. In the policy learnt by the SONQL, the optimal actions are more clearly defined, although both have similar shapes. The state zones of the Q_learning policy which are less defined, indicate that they were not visited enough and, consequently, a higher divergence between the Q_learning and SONQL policies can be found.

The Q function at different learning iterations is represented in Figure 7.21. As in previous representations, the maximum value for each state is shown, which is the state-value function $V(s)$. It can be observed how the shape of the function evolves according to the learning iterations. The $V(s)$ axis has been inverted for a clearer visualization. It is interesting to compare the shape of the $V(s)$ function learnt by the SONQL algorithm with the one of the Q_learning algorithm, see Figure 7.17. The function learnt by the SONQL is much smoother, although the shape is similar. The ranges of the two functions are not exactly the same. The maximum and minimum values of the SONQL function are higher, which is caused by the interference of the NN when updating the samples. However, the values of different actions at
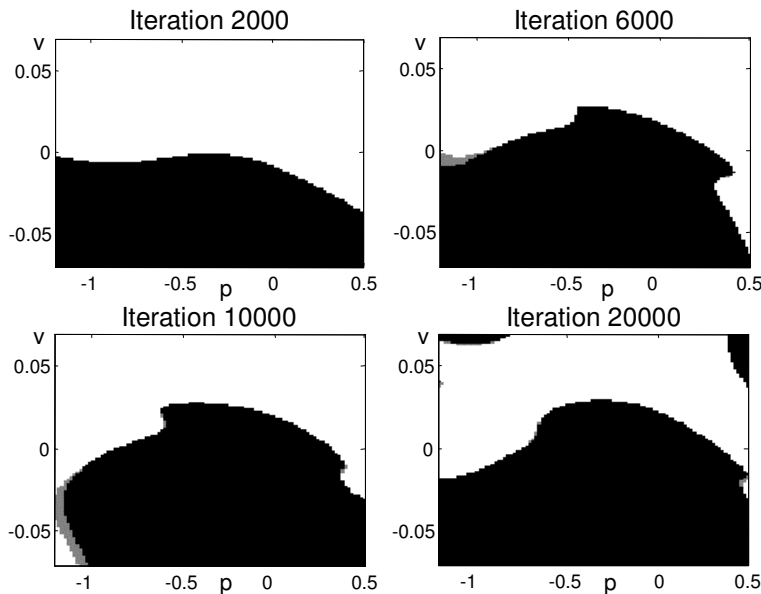
Figure 7.20: State/action policy after different learning iterations for the SONQL algorithm. The actions are represented in different colors: white for forward thrust, gray for no-thrust, and black for reverse thrust.

the same states maintain the same relation which finally defines the policy. The $V(s)$ function learnt by the SONQL algorithm after 20000 iterations is shown in Figure 7.22. In this figure two different views of the approximated state-value function can be observed.

The influence of the database was analyzed using the same SONQL configuration and changing the database size. Instead of referring to the density parameter t, the number of samples stored in the database will be used. The total number of NN updates was fixed at $8 \cdot 10^6$ iterations approximately, and in each experiment 100 trials were simulated. The first experiment consisted of learning with only the current learning sample, that is, without using the database. Figure 7.23 shows the obtained result. It can be appreciated that the algorithm is not able to learn an optimal mapping. The effectiveness occurs in 110 iterations, which is the same effectiveness as with a random policy. The confidence interval is very large, since each of the 100 trials had a very different result. The first conclusion is that, although the NN is able to approximate the optimal Q-function, the interference problem does not allow the stability of the learning process. The database is therefore necessary to ensure the convergence.

Three more experiments were executed in which the number of learn-
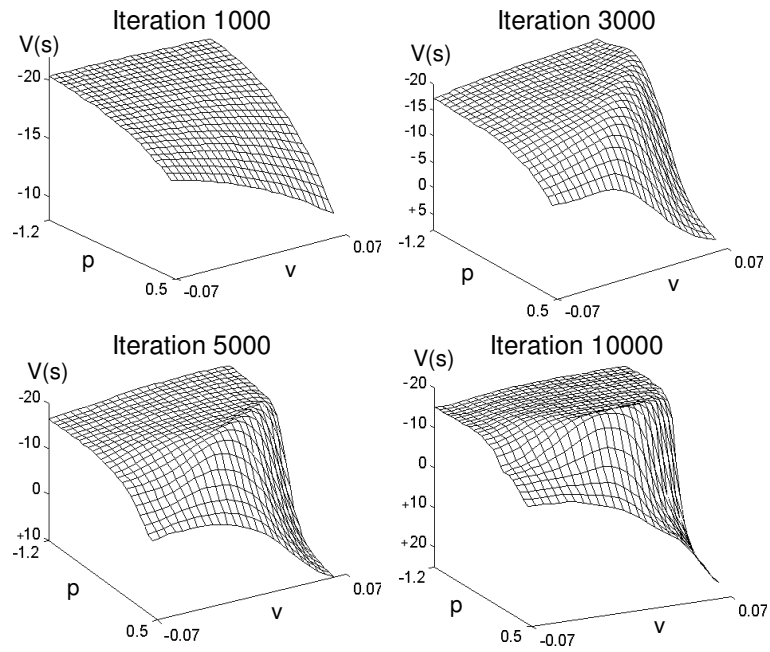
Figure 7.21: State value function $V(s)$ after several learning iterations for the SONQL algorithm.
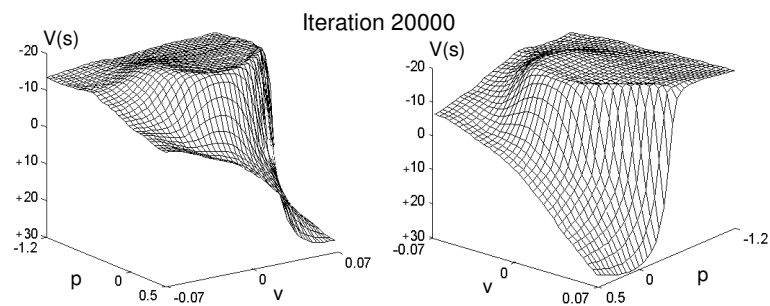
Figure 7.22: Two views of the state value function $V(s)$ after 20000 learning iterations with the SONQL algorithm.

ing samples were 280, 150 and 85 samples. In Figure 7.24, the effectiveness of these experiments, together with the previous ones, are shown. The graphs represent the number of NN updates. It can be observed that all the SONQL algorithms finished at the same number of iterations ($8 \cdot 10^6$). The convergence time is not drastically different, although the experiments with a smaller database converged sooner. However, the experiments with a larger database obtained a better effectiveness. The averages of the "D=470", "D=280", "D=150" and "D=85" experiments are 53, 54, 56 and 58 respectively. Figure 7.25 shows the same graphs but with respect to the number of learning iterations of the SONQL algorithm. The number of iterations is smaller since the number of NN updates is equal and the database is larger. A second conclusion can be extracted from these results; with a larger database, a better learning is achieved. A large database implies a large set of learning samples uniformly distributed throughout the visited space. This representative set of samples improves the learning capacity of the direct Q_learning. Finally, besides the improvement of the effectiveness, a larger database also implies a higher computation of the SONQL algorithm for the same number of iterations, which must be taken into account in a real-time application.

The results obtained in this section empirically demonstrate the benefit of using the SONQL algorithm, and especially the learning samples database, for solving the generalization problem.

7.2.4   Discussion

The comparison between the SONQL algorithm and the Q_learning algorithm must only be considered as an evaluation of the SONQL policy with respect to the optimal one, which was supposed to be the one learnt by Q_learning. The Q_learning exhibited a long convergence time since it was affected by the generalization problem. It is interesting to compare the performance of the SONQL algorithm with respect to other RL algorithms that have also dealt with the "moutain-car" task. The results of some of them are next commented. These algorithms have been classified according to the generalization methodology.

Decision Trees A decision tree used to generalize the Q_learning algorithm was proposed by Vollbrecht [Vollbrecht, 1999]. The effectiveness of this algorithm was 58 iterations to goal, and the convergence time was 20000 learning episodes. The number of learning iterations was not detailed. Another interesting work can be found in [Munos and Moore, 2002]. In this work, a detailed study of the value function and policy function was presented, although the effectiveness was not mentioned.
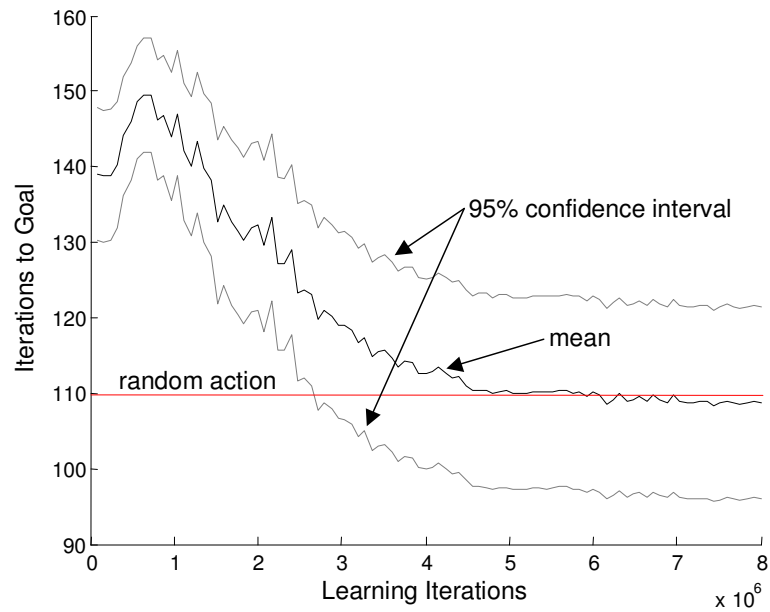
Figure 7.23: Effectiveness of the SONQL algorithm with respect to the learning iterations without using the database. In this case, the number of learning iterations is the same as the number of NN updates.
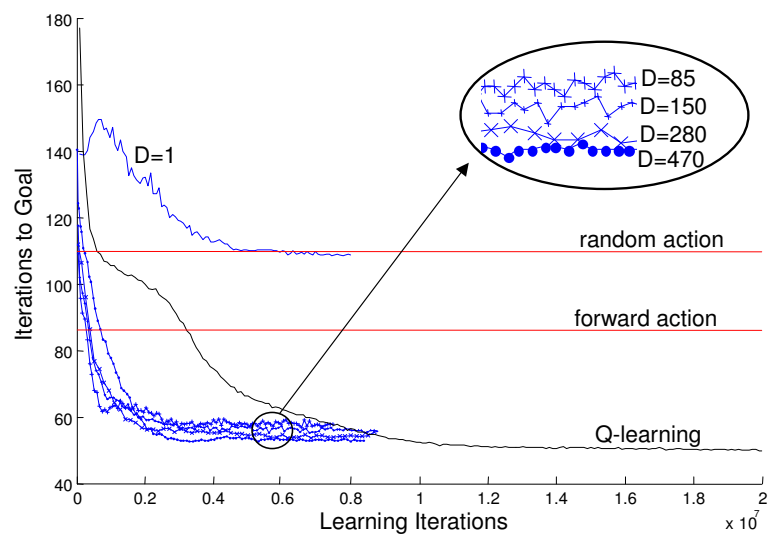
Figure 7.24: Effectiveness of the SONQL algorithm with respect to the number of NN updates. The performance of the database is shown with five different sizes (D=1,85,150,280 and 470).
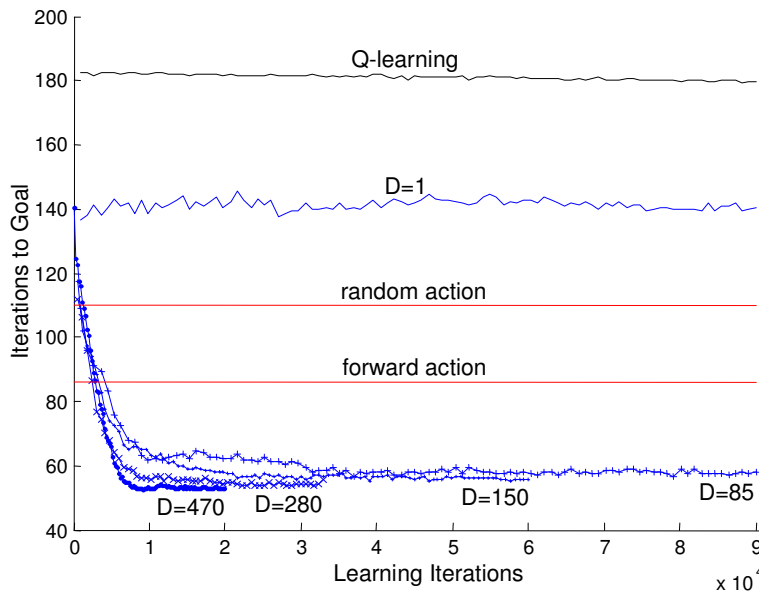
Figure 7.25: Effectiveness of the SONQL algorithm with respect to the learning iterations. The performance of the database is shown with five different sizes (D=1,85,150,280 and 470).

CMAC The use of the CMAC function approximator in the "mountain-car" task is also common. The most known application was done by Sutton [Sutton, 1996], who centered his work in the value function. He also analyzed the use of eligibility traces. CMAC was implemented with 10 tilings, having 9x9 tiles each. However, the effectiveness was not mentioned. Another implementation can be found in [Kretchmar and Anderson, 1997], in which they applied a CMAC function with 11 tilings and 3x3 tiles. The effectiveness was 68 iterations to goal and the convergence time was 1500 episodes. In the same work, the use of a radial basis function approximator was also reported. The results showed a better effectiveness with the same convergence time, the number of iterations to goal was 56.

Memory-based In [Smart, 2002], a memory-based method using locally weighted regression was evaluated with the "mountain-car" task. The effectiveness of the algorithm was 70 iterations to goal and the convergence time was not clearly stated.

Neural Networks To the authors best knowledge there are no successful examples that apply Neural Networks to the "mountain-car" task. In

[Boyan and Moore, 1995], the divergence of Neural Networks in this problem was stated using a 2 layer network with 80 hidden neurons and using back-propagation updating. This result is the same obtained with the SONQL algorithm when the database was not used. Therefore, the SONQL algorithm, making use of the database, can be considered as one successful case in which a NN was able to solve the generalization problem proposed in the "mountain-car" task.

The results of these algorithms does not improve the mark obtained with the SONQL algorithm. Any of them was able to reach an effectiveness of 53 iterations to goal. As far as the number of learning iterations, it was not clearly stated in these works and, therefore, it cannot be directly compared. However, from the comments about this feature, it is extracted that the SONQL algorithm requires less iterations to converge. They talk about the number of episodes to learn, which even if it is multiplied by the final effectiveness, gives a higher number of iterations. Another important aspect is the computational cost. Although it cannot be quantitatively compared, the SONQL algorithm may require more computation than these algorithms, due to the high number of learning samples to update at each SONQL iteration. Therefore, the conclusion of this comparison is, the SONQL has a higher generalization capability and a short convergence time, but these features must be balanced with respect to the available computational power and real-time requirements.

Finally, after presenting the results of the SONQL algorithm in the "mountain-car" benchmark, it can be concluded that:

- The SONQL algorithm was able to solve the generalization problem found in the "mountain-car" task. The generalization capability of the NN allowed the approximation of the Q-function. The database of learning samples was also a requirement to guarantee the convergence in 100% of the cases.

- The number of learning samples clearly influenced the performance of the SONQL algorithm. The higher the number of samples, the higher the performance. However, a high number of samples also implies a high computational cost, which must be taken into account in a real-time application.

- The effectiveness of the SONQL algorithm was not as good as the effectiveness of the Q_learning algorithm. However, a drastic reduction of the number of iterations needed to converge was demonstrated by the SONQL algorithm. Even the number of NN updates was significantly

smaller than the number of iterations needed for the convergence of the Q_learning algorithm.

- The results of the SONQL algorithm with respect to other RL algorithms has shown a higher effectiveness and a smaller convergence time.

- It has been demonstrated that the SONQL algorithm is a suitable approach for reinforcement learning problems in which a generalization and fast learning are required.

Chapter 8

Conclusions

This chapter concludes the work presented throughout this dissertation. It first summarizes the thesis by reviewing the contents described in each chapter. It then points out the research contributions extracted from the proposals and the experiments. In addition, all aspects which have not been accomplished as well as some interesting future research issues are commented on in the future work section. Then, the research framework in which this thesis was achieved is described. Finally, the publications related to this work are listed.

8.1   Summary

In order to develop an autonomous robot, a control architecture must be included in the robot control system. The control architecture has the goal of accomplishing a mission which can be divided into a set of sequential tasks. Chapter 2 presented Behavior-based control architectures as a methodology to implement this kind of controllers. Its high interaction with the environment, as well as its fast execution and reactivity, are the keys to its success in controlling autonomous robots. This chapter also compared some classic approaches by testing their performance in a simulated task with an Autonomous Underwater Vehicle. The main attention was given to the coordination methodology. Competitive coordinators assured the robustness of the controller, whereas cooperative coordinators determined the performance of the final robot trajectory. Chapter 3 proposed the structure of a control architecture for an autonomous robot. Two main layers are found in this schema; the deliberative layer which divides the robot mission into a set of tasks, and the behavior-based layer which is in charge of accomplishing these tasks. This chapter and the thesis centered only on the behavior-based

layer. A behavior coordination approach was proposed. The main feature is its hybrid coordination of behaviors, between competitive and cooperative approaches. The approach was tested with the simulated task as well.

The second part of the thesis centered on the implementation of the robot behaviors. It proposed the use of a learning algorithm to learn the internal mapping between the environment state and the robot actions. Chapter 4 presented Reinforcement Learning as a suitable learning theory for robot learning. Its online applicability and the non-requirement of any previous information about the environment are the most important advantages. In addition, the Q_learning algorithm was presented, which is specially adequate for its capability in off-policy learning. The most important drawback is the generalization problem. Reinforcement Learning algorithms are based on discrete representations of the state and action spaces. When these algorithms are applied to continuous variables, as most robotics applications require, the discretization of the variables causes an enormous number of states and a long learning time. The generalization makes the application of Reinforcement Learning in robotics impractical. However, several techniques were presented which attempt to solve this problem. Chapter 5 proposed a Reinforcement Learning algorithm designed to be applied to robotics. The goal of the SONQL algorithm is to learn robot behaviors. It is based on the Q_learning algorithm and solves the generalization problem by using a Neural Network and a database of the most representative learning samples.

The thesis has shown some experiments with the Autonomous Underwater Vehicle URIS. Chapter 6 detailed the experimental set-up specifically designed for these experiments. A description of the vehicle was done first, followed by the presentation of two sensory systems. The target detection and tracking system was in charge of detecting an artificial target by using computer vision. Its purpose was to provide the detection of the environment state for the SONQL algorithm. The second sensorial system was the localization system, which also uses computer vision to estimate the position and velocity of the vehicle. The system was responsible for the fine controllability of the robot. Finally, Chapter 7 showed some results of the SONQL algorithm. The feasibility of the approach was demonstrated by learning a target following behavior in real-time computation. The hybrid coordination system demonstrated to be a suitable methodology, by coordinating the SONQL-based behavior and other manually implemented behaviors. The SONQL algorithm was also tested in a simulated benchmark. This task demonstrated empirically the feasibility of this algorithm in a complex generalization problem.

8.2   Contributions

This thesis has accomplished the proposed goal which is the development of a robot control architecture for an AUV able to achieve simple tasks and exhibit real-time learning capabilities. In the development of this goal, some research contributions were achieved. Hereafter these contributions are listed:

Online learning of robot behaviors . The most important contribution has been the online learning of robot behaviors. The use of Reinforcement Learning in robotics is very common nowadays. However, there are not many approaches which perform an online learning. It is, therefore, an important contribution to demonstrate the feasibility of the SONQL in a real-time task, specially in a complex domain such as underwater robotics. The algorithm proved able to learn the state/action mapping of one DOF in less than 400 iterations, which was less than two minutes. Although the best parameters were used and the experiments were designed in detail, these results point out the important role learning algorithms will have in future robotics applications.

SONQL as a continuous state RL algorithm . The second contribution is also related to the SONQL algorithm. This algorithm demonstrated a high generalization capability in the "mountain-car" benchmark. The combination of the Neural Network and the learning samples database resulted in an algorithm able to face the generalization problem. The Neural Network offered a high function approximation capability, and the database guaranteed its stability by avoiding the interference problem. To the best of the author's knowledge, similar approaches have not been found in the literature and, therefore, the SONQL represents a contribution in the Reinforcement Learning field. However, it must be noted that, although the action space is continuous in the Neural Network, the search of greedy actions requires a discretization of this space. Therefore, the SONQL must be considered only as an algorithm to solve the generalization problem in the state space.

Methodologies for Generalizing . The generalization problem in Reinforcement Learning was treated in detail. The most important methodologies currently being applied were described. This study was not considered as an exhaustive survey but a general overview of the most used techniques.

Development of a behavior-based control system . Another contribution was the development of the behavior-based control layer, and in

particular, the hybrid coordination methodology. The main features of the coordination system are its simplicity and robustness which assure the safety of the vehicle. In addition, the cooperation between behaviors improves the final robot trajectory. The behavior-based control layer demonstrated as being an efficient tool in the implementation of a set of behaviors and the obtained results were highly satisfactorily.

Behavior-based control architectures . Four classic Behavior-based control architectures were presented, tested and compared. These architectures represent the most important principles in this field. Therefore, this study offers an exemplified introduction to Behavior-based Robotics. The testing of the architectures in a simulated environment also led to the identification of the dynamics model of GARBI and URIS underwater robots.

Development of a localization system . A localization system for underwater robots in structured environments was proposed. The system is able to estimate the position and orientation in three degrees of freedom and also the velocity. The localization is based on a coded pattern and a computer vision system. The high accuracy of the estimations and the real-time execution of the algorithm are the main features. The localization system has been one of the most important factors for the success of the presented experiments.

8.3   Future Work

The development of a research project always provokes the discovery of new problems as well as new interesting research topics. Future work of the sort contained in this thesis has elements of both kinds. Five different points were considered to be the most logical lines to continue this research. The order in which they are presented corresponds to its hypothetic chronological execution, and also to its specification level.

Exploration/Exploitation policy . The policy which was followed while the SONQL was learning is the $\epsilon - greedy$ policy. The advantage of this is the exploration of new state/action pairs which could have a higher Q value. The effectiveness of random actions in exploring is, at the same time, a problem when working with real systems. Random actions cause very abrupt changes of the robot's movement, which puts at risk the safety and controllability of the robot. A future improvement could be the design of a exploration/exploitation policy which is more

appropriate for robotics. This policy could make use of the learning sample database which already contains the non-explored space. However, the convergence of the algorithm and its necessary time should be studied and compared with the $\epsilon - greedy$ policy.

Further SONQL testing . The experimental results have shown the learning of the "target tracking" behavior. This behavior was chosen for the ease in detecting an artificial target with a computer vision system. It would be interesting to add a new state dimension to allow the learning of moving targets. It would also be interesting to test the feasibility of the SONQL with other behaviors. A "trajectory following" behavior, for instance, has a difficult solution for non-holonomic vehicles. The solution adopted by the SONQL algorithm would certainly be interesting. Another important test would be the execution of the algorithms in a natural environment, which usually has much more perturbations, and would also allow the learning of the heave DOF.

Action space generalization . As was treated throughout this dissertation, the SONQL algorithm cannot work effectively if several continuous actions are present. In the learning of the robot behaviors, only one continuous action was used since each DOF was independently learnt. However, the extension of the algorithm to more than one continuous action cannot be easily accomplished. The main reason for that resides in the difficulty in finding the maximum value of the Q-learning function when it is implemented with a Neural Network. This problem, which is also found in other generalization techniques, also constitutes a future work.

Other RL issues . Besides the generalization problem, the correct observation of the Markovian state is also a very important point in robotics. Partially Observable Markov Decision Processes were presented in Chapter 4 to deal with environments in which the state is corrupted. The use of a POMDP algorithm should be considered in future research since the difficulty in having a correct observation is very high. Also the use of eligibility traces has been pointed in the experimental results for their higher suitability in Non-Markovian environments. In addition, some other research issues about Reinforcement Learning were pointed out. Policy methods and hierarchical learning are two interesting topics which have recently received a lot of attention and are very suitable to robotics.

Deliberative layer This thesis has concentrated on the behavior-based layer

of a complete control architecture only. It is logical to note as a future work the development of the upper layer, which is the deliberative layer. This layer would allow the execution of real missions instead of simple tasks. The deliberative layer would configure a set of behaviors by setting the priorities among them to execute a particular task. After the fulfillment of this task, a new one would be started. This would be repeated until the mission was completed. However, in order to test the deliberative layer, an assorted set of sensors and behaviors must first be fully working. This future work also involves a new research line since behavior-based robotics is not a suitable approach for mission planning.

8.4   Research Framework

The results and conclusions presented in this thesis are based on a set of experiments. During the period in which this thesis was completed, several robot platforms were used. This section summarizes the research facilities and evolution of this thesis. The most relevant research publications will be referred and can be checked in the next section.

The first experiments consisted of testing some Behavior-based Robotics control architectures. At that time, the robot GARBI was available for tele-operation tasks but it was still not ready to test a control architecture. As has been described throughout this dissertation, in order to perform a test with a control architecture, the subordinate components, such as sensors, actuators and the low-level controller, must be properly working. Therefore, the experimentation was performed with a simulated dynamics model of GARBI and a simulated underwater environment. This led to the identification of the dynamics model of this vehicle [CAMS'01a]. Moreover, further work on dynamics identification was conducted since then [IIA'01,MED'02,GCUV'03b]. This realistic model allowed the execution of a simulated task and the comparison of different control architectures [MCMC'00,QAR'00]. This work was carried out during a research stage at the University of Wales College, Newport, under the supervision of Prof. Geoff Roberts. Also, the hybrid coordinator methodology [IROS'01a,IIA'00] was designed using this simulator.

The second part of this thesis discussed the use of a Reinforcement Learning algorithm to learn the internal structure of a behavior. The first steps in this field were carried out with simulation and published in [IROS'01a]. At that time, as a result of a research stay at the University of Hawaii, under the supervision of Prof. Junku Yuh, the learning algorithms were applied to two different robots. The first one was a commercial mobile robot (Magellan

Pro mobile robot). The advantages of using this platform first, instead of an AUV, were certainly great. The ease in controlling the vehicle and the environment allowed the execution of a high number of experiments [IIA'03]. The utility of these experiments was the detection of the interference problem in the Neural Network. This problem was solved by designing a first version of the SONQL algorithm which was tested with a second robot. In this case, an underwater robot called ODIN, developed in the University of Hawaii. The experiments with ODIN [IFAC'02,OCEANS'01] demonstrated the feasibility of Reinforcement Learning with an autonomous underwater robot.

The experiments were then improved and reproduced with the underwater robot URIS [IROS'02], which are the experiments presented in this thesis. In this case, several sensory systems were specifically developed. Among them, the accurate localization system permitted a fine control of the robot [ICRA'03a,GCUV'03a]. The experiments with URIS are the most complete, although they could not have been achieved without the previous experience. Finally, the generalization capability of the SONQL with the "mountain-car" task was evidently performed in simulation [IROS'03].

8.5   Related Publications

Reinforcement Learning and Robotics

[IROS'03] Marc Carreras, Pere Ridao and Andres El-Fakdi, "Semi-Online Neural-Q-_learning for Real-time Robot Learning", IEEE-RSJ International Conference on Intelligent Robots and Systems, Las Vegas, USA, October 27 - 31, 2003.

[IROS'02] Marc Carreras, Pere Ridao, Joan Batlle and Tudor Nicosevici, "Efficient Learning of Reactive Robot Behaviors with a Neural-Q_Learning Approach", IEEE/RSJ International Conference on Intelligent Robots and Systems, Lausanne, Switzerland, September 30 - October 4, 2002.

[IFAC'02] Marc Carreras, Junku Yuh and Joan Batlle, "High-level Control of Autonomous Robots using a Behavior-based Scheme and Reinforcement Learning", 15th IFAC World Congress on Automatic Control, Barcelona, Spain, July 21-26, 2002.

[IIA'03] Marc Carreras, Junku Yuh, Pere Ridao and Joan Batlle, "A Neural-Q_learning Approach for Online Robot Behavior Learning". (research

work performed in 2001 at the Autonomous Systems Laboratory, University of Hawaii). Research report IIA 03-06-RR. Institute of Informatics and Applications. University of Girona. May 2003.

[OCEANS'01] Marc Carreras, Junku Yuh and Joan Batlle. "An hybrid methodology for RL-based behavior coordination in a target following mission with an AUV". OCEANS 2001 MTS/IEEE Conference. Honolulu, Hawaii, USA, 5-8 November, 2001.

[IROS'01a] Marc Carreras, Joan Batlle and Pere Ridao. "Hybrid Coordination of Reinforcement Learning-based Behaviors for AUV Control". IROS 2001, International Conference on Intelligent Robots and Systems. Maui, Hawaii, USA, October, 2001.

Behavior-based control architectures

[MCMC'00] Marc Carreras, Joan Batlle, Pere Ridao and Geoff Roberts. "An overview on behaviour-based methods for AUV control". MCMC'2000, 5th IFAC Conference on Manoeuvring and Control of Marine Crafts. Aalborg, Denmark. 23-25 August, 2000.

[QAR'00] Marc Carreras, Joan Batlle and Pere Ridao. "Reactive control of an AUV using Motor Schemas". QAR'2000, International conference on Quality control, Automation and Robotics. Cluj Napoca, Rumania. 19-20 May, 2000.

[IIA'00] Marc Carreras. "An Overview of Behaviour-based Robotics with simulated implementations on an Underwater Vehicle". Research report IIA 00-14-RR. Institute of Informatics and Applications. University of Girona. October 2000.

Vision-based Localization System for an AUV

[ICRA'03a] Marc Carreras, Pere Ridao, Rafael Garcia and Tudor Nicosevici, "Vision-based Localization of an Underwater Robot in a Structured Environment", IEEE International Conference on Robotics and Automation ICRA'03, Taipei, Taiwan, 2003.

[GCUV'03a] Marc Carreras, Pere Ridao, Joan Batlle and Xavier Cufí, "AUV navigation in a structured environment using computer vision", 1st IFAC Workshop on Guidance and Control of Underwater Vehicles GCUV '03, Wales, UK, April 9-11, 2003.

UUV Dynamics Identification

[GCUV'03b] Marc Carreras, Antonio Tiano, Andres El-Fakdi, Antonio Zirilli and Pere Ridao, "On the identification of non linear models of unmanned underwater vehicles" (part II), 1st IFAC Workshop on Guidance and Control of Underwater Vehicles GCUV '03, Wales, UK, April 9-11, 2003.

[MED'02] Antonio Tiano, Marc Carreras, Pere Ridao and Antonio Zirilli, "On the identification of non linear models of unmanned underwater vehicles" (part I), 10th Mediterranean Conference on Control and Automation, Lisbon, Portugal, July 9-12, 2002.

[CAMS'01a] Pere Ridao, Joan Batlle and Marc Carreras. "Model identification of a low-speed UUV with on-board sensors". IFAC conference CAMS'2001, Control Applications in Marine Systems. Glasgow, Scotland, U.K., 18-20 July, 2001.

[IIA'01] Pere Ridao, Joan Batlle and Marc Carreras. "Dynamics Model of an Underwater Robotic Vehicle". Research report IIiA 01-05-RR. Institute of Informatics and Applications. University of Girona. April 2001.

Related work in underwater robotics

[ICRA'03b] Rafael Garcia, Xavier Cufí, Marc Carreras and Pere Ridao, "Correction of Shading Effects in Vision-Based UUV Localization", IEEE International Conference on Robotics and Automation ICRA'03, Taipei, Taiwan, 2003.

[CEP'02] Pere Ridao, Joan Batlle and Marc Carreras, "O^2CA^2, a new object oriented control architecture for autonomy: the reactive layer", Control Engineering Practice, Volume 10, Issue 8, Pages 857-873, August 2002.

[ICRA'01] Pere Ridao, Joan Batlle, Josep Amat and Marc Carreras. "A distributed environment for virtual and/or real experiments for underwater robots". ICRA'01, 2001 IEEE International Conference on Robotics and Automation, Seoul, Korea, 21-26 May, 2001.

[CAMS'01b] Pere Ridao, Marc Carreras, Joan Batlle and Josep Amat. "O^2CA^2: A new hybrid control architecture for a low cost AUV". IFAC conference CAMS'2001, Control Applications in Marine Systems. Glasgow, Scotland, U.K., 18-20 July, 2001.

[IROS'01b] Rafael Garcia, Xevi Cufí and Marc Carreras. "Estimating the motion of an Underwater Robot from a Monocular Image Sequence".

IROS 2001 International Conference on Intelligent Robots and Systems. Maui, Hawaii, USA, October, 2001.

[MED'01] Joan Batlle, Pere Ridao and Marc Carreras. "An Underwater Autonomous Agent. From Simulation To Experimentation". 9th Mediterranean Conference on Control and Automation. Dubrovnik, Croatia, 27-29, 2001.

Bibliography

[Agre and Chapman, 1987] Agre, P. and Chapman, D. (1987). Pengi: an implementation of a theory of activity. In *Proceeding of the Sixth Annual Meeting of the American Association for Artificial Intelligence*, pages 268–272, Seattle, Washington.

[Albus, 1971] Albus, J. S. (1971). A theory of cerebellar function. *Mathematical Biosciences*, 10:25–61.

[Albus, 1981] Albus, J. S. (1981). *Brain, behavior and robotics*. Byte Books, Peterborough, NH.

[Albus, 1991] Albus, J. S. (1991). Outline for a theory of intelligence. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):473–509.

[Amat et al., 1996] Amat, J., Batlle, J., Casals, A., and Forest, J. (1996). GARBI: a low cost ROV, constrains and solutions. In *6ème Seminaire IARP en robotique sous-marine*, pages 1–22, Toulon-La Seyne, France.

[Arbib, 1981] Arbib, M. A. (1981). Perceptual structures and distributed motor control. In *Handbook of physiology-The nervous system II: Motor Control*, chapter 33, pages 1449–1480. Oxford University Press, Oxford, UK.

[Arkin, 1986] Arkin, R. C. (1986). Path planning for a vision-based autonomous robot. In *Proceedings of the SPIE Conference on Mobile Robots*, pages 240–249, Cambridge, MA.

[Arkin, 1987] Arkin, R. C. (1987). Motor schema based navigation for a mobile robot: an approach to programming by behaviour. In *Proceedings of the IEEE Int. Conference on Robotics and Automation*, pages 264–271, Raleigh, NC.

[Arkin, 1989] Arkin, R. C. (1989). Motor schema-based mobile robot navigation. *International Journal of Robotics Research, August 1989*, 8(4):92–112.

[Arkin, 1992] Arkin, R. C. (1992). Behaviour-based robot navigation for extended domains. *Adaptive Behaviour, Fall 1992*, 1(2):201–225.

[Arkin, 1993] Arkin, R. C. (1993). Modeling neural function at the schema level: Implications and results for robotic control. In *Biological neural networks in invertebrate neuroethology and robotics*, page 17. Boston: Academic Press.

[Arkin, 1998] Arkin, R. C. (1998). *Behavior-based Robotics*. MIT Press.

[Atkenson et al., 1997] Atkenson, C., Moore, A., and Schaal, S. (1997). Locally weighted learning. *Artificial Intelligence Review*, 11:11–73.

[Bagnell and Schneider, 2001] Bagnell, J. and Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Korea.

[Baird, 1995] Baird, K. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning: Twelfth International Conference*, San Francisco, USA.

[Baird and Klopf, 1993] Baird, L. and Klopf, A. (1993). Reinforcement Learning with High-Dimensional, Continuous Action. Technical Report WL-TR-93-1147, Wright Laboratory.

[Bakker et al., 2002] Bakker, B., Linaker, F., and Schmidhuber, J. (2002). Reinforcement learning in partially observable mobile robot domains using unsupervised event extraction. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Lausanne, Switzerland.

[Balch and Arkin, 1993] Balch, T. and Arkin, R. (1993). Avoiding the past: A simple but effective strategy for reactive navigation. In Werner, Robert; O'Conner, L., editor, *Proceedings of the 1993 IEEE International Conference on Robotics and Automation: Volume 1*, pages 678–685, Atlanta, GE. IEEE Computer Society Press.

[Barto and Mahadevan, 2003] Barto, A. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete-Event Systems journal*, in press.

[Barto et al., 1983] Barto, A., Sutton, R., and Anderson, C. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13:835–846.

[Baxter and Barlett, 2000] Baxter, J. and Barlett, P. (2000). Reinforcement learning in POMDPs via direct gradient ascent. In *Proceedings of the Seventeenth International Conference on Machine Learning.*

[Bellman, 1957] Bellman, R. (1957). *Dynamic Programming.* Princenton University Press.

[Bertsekas and Tsitsiklis, 1996] Bertsekas, D. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA.

[Boyan and Moore, 1995] Boyan, J. and Moore, A. (1995). Generalization in reinforcement learning: Safely approximating the value function. In *NIPS-7*, San Mateo, CA, USA.

[Brooks, 1986] Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23.

[Brooks, 1989] Brooks, R. A. (1989). A robot that walks; emergent behaviours from a carefully evolved network. *Neural Computation, 1989*, 1(2):253–262.

[Brooks, 1990] Brooks, R. A. (1990). The behavior language: User's guide. Technical Report AIM-1227, Massachusetts Institute of Technology.

[Brooks, 1991a] Brooks, R. A. (1991a). Intelligence without reason. In Myopoulos, J. and Reiter, R., editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[Brooks, 1991b] Brooks, R. A. (1991b). New approaches to robotics. *Science*, 235:1227–1232.

[Buck et al., 2002] Buck, S., Beetz, M., and Schmitt, T. (2002). Approximating the value function for continuous space reinforcement learning in robot control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Lausanne, Switzerland.

[Cassandra et al., 1994] Cassandra, A., Kaelbling, L., and Littman, M. (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA.

[Chapman and Kaelbling, 1991] Chapman, D. and Kaelbling, L. (1991). Input generalization in delayed reinforcement learning: an algorithm and preformance comparisons. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 726–731, San Mateo, CA.

[Chatila and Laumond, 1985] Chatila, R. and Laumond, J. (1985). Position referencing and consistent world modelling for mobile robots. In *IEEE International Conference on Robotics and Automation*, pages 138–170.

[Connell, 1990] Connell, J. H. (1990). *Minimalist Mobile Robots*. Academic Press, Perspectives in Artificial Intelligence Series San Diego, 1990, 175 pages. Introduction by Rodney Brooks.

[Crites and Barto, 1996] Crites, R. and Barto, A. (1996). Improving elevator performance using reinforcement learning. In *NIPS-8*, Cambridge, MA. MIT Press.

[Dayan, 1992] Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning*, 8:341–362.

[Dietterich, 2000] Dietterich, T. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

[Dorigo and Colombetti, 1998] Dorigo, M. and Colombetti, M. (1998). *Robot Shaping: An Experiment in Behavior Engineering*. MIT Press/Bradford Books.

[Fikes and Nilsson, 1971] Fikes, R. and Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

[Gachet et al., 1994] Gachet, D., Salichs, M., Moreno, L., and Pimental, J. (1994). Learning emergent tasks for an autonomous mobile robot. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS'94)*, pages 290–97, Munich, Germany.

[Garcia et al., 2001] Garcia, R., Batlle, J., Cufi, X., and Amat, J. (2001). Positioning an underwater vehicle through image mosaicking. In *IEEE International Conference on Robotics and Automation*, pages 2779–2784, Rep.of Korea.

[Garcia et al., 2002] Garcia, R., Puig, J., Ridao, P., and Cufi, X. (2002). Augmented state kalman filtering for AUV navigation. In *IEEE International Conference on Robotics and Automation*, pages 4010–4015, Washington.

[Gaskett, 2002] Gaskett, C. (2002). *Q_learning for Robot Control*. PhD thesis, Australian National University.

[Gat, 1991] Gat, E. (1991). *Reliable Goal-directed Reactive Control for Real-World Autonomous Mobile Robots*. PhD thesis, Virginia Polytechnic and State University, Blacksburg, Virginia.

[Gill et al., 1981] Gill, P., Murray, W., and Wright, M. (1981). The levenberg-marquardt method. In *Practical Optimization*, pages 136–137. London: Academic Press.

[Gonzalez and Woods, 1992] Gonzalez, R. and Woods, R. (1992). *Digital Image Processing*. Addison-Wesley, Reading, MA.

[Gordon, 1999] Gordon, G. (1999). *Approximate Solutions to Markov Decision Processes*. PhD thesis, Carnegie-Mellon University.

[Gracias and Santos-Victor, 2000] Gracias, N. and Santos-Victor, J. (2000). Underwater video mosaics as visual navigation maps. *Computer Vision and Image Understanding*, 79(1):66–91.

[Gross et al., 1998] Gross, H., Stephan, V., and Krabbes, M. (1998). A neural field approach to topological reinforcement learning in continuous action spaces. In *Proceedings of the IEEE World Congress on Computational Intelligence, WCCI'98 and International Joint Conference on Neural Networks, IJCNN'98*, Anchorage, Alaska.

[Hagen and Krose, 2000] Hagen, S. and Krose, B. (2000). Neural q_learning. Technical Report IAS-UVA-00-09, Computer Science Institute, University of Amsterdam, The Netherlands.

[Harvey et al., 1997] Harvey, I., Husbands, P., Cliff, D., Thomson, A., and Jakobi, A. (1997). Evolutionary robotics: the sussex approach. *Robotics and Autonomous systems*, 20(2-4):205–224.

[Haykin, 1999] Haykin, S. (1999). *Neural Networks, a comprehensive foundation*. Prentice Hall, 2nd ed. edition.

[Huang, 1996] Huang, H. M. (1996). An architecture and a methodology for intelligent control. *IEEE Expert: Intelligent Systems and their applications*, 11(2):46–55.

[Jaakkola et al., 1995] Jaakkola, T., Singh, S., and Jordan, M. (1995). *Reinforcement Learning algorithms for partially observable Markov decission problems*, volume 7, pages 345–352. Morgan Kaufman.

[Kaelbling, 1999] Kaelbling, L. P. (1999). Robotics and learning. In *The MIT encyclopedia of the cognitive sciences*. MIT Press.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

[Kalmar et al., 1997] Kalmar, Z., Szepesvari, C., and Lorincz, A. (1997). Module-Based Reinforcement Learning: Experiments with a Real Robot. In *Proceedings of the 6th European Workshop on Learning Robots*.

[Kawano and Ura, 2002] Kawano, H. and Ura, T. (2002). Fast reinforcement learning algorithm for motion planning of non-holonomic autonomous underwarter vehicle in disturbance. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Lausanne, Switzerland.

[Khatib, 1985] Khatib, O. (1985). Real-time obstacle avoidance for manipulators and mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 500–505, St. Louis, MO.

[Kondo and Ito, 2002] Kondo, T. and Ito, K. (2002). A Reinforcement Learning with Adaptive State Space Recruitment Strategy for Real Autonomous Mobile Robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Lausanne, Switzerland.

[Kretchmar and Anderson, 1997] Kretchmar, R. and Anderson, C. (1997). Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning. In *Proceedings os the IEEE International Conference on Neural Networks*, pages 834–837, Houston, TX.

[Laird and Rosenbloom, 1990] Laird, J. E. and Rosenbloom, P. S. (1990). Integrating, execution, planning, and learning in Soar for external environments. In Dietterich, T. S. W., editor, *Proceedings of the 8th Annual Meeting of the American Association for Artificial Intelligence*, pages 1022–1029, Hynes Convention Centre. MIT Press.

[Lefebvre and Saridis, 1992] Lefebvre, D. and Saridis, G. (1992). A computer architecture for intelligent machines. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 245–250, Nice, France.

[Lin, 1992] Lin, L. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3/4):293–321.

[Lin, 1993] Lin, L. (1993). *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie-Mellon University.

[Lyons, 1992] Lyons, D. (1992). Planning, reactive. In *Encyclopedia of Artificial Intelligence*. John Wiley and Sons, New York.

[Maes, 1989] Maes, P. (1989). How to do the right thing. *Connection Science*, 1(3):291–323.

[Maes, 1990] Maes, P. (1990). Situated agents can have goals. *Robotics and Automation Systems*, 6:49–70.

[Maes, 1991] Maes, P. (1991). A bottom-up mechanism for behaviour selection in an artificial creature. In Meyer, J.-A. and Wilson, S. W., editors, *From animals to animats*, pages 238–246. First International Conference on Simulation of Adaptive Behavior.

[Maes and Brooks, 1990] Maes, P. and Brooks, R. (1990). Learning to coordinate behaviors. In *Proceedings of Eighth AAAI*, pages 796–802. Morgan Kaufmann.

[Mahadevan and Connell, 1992] Mahadevan, S. and Connell, J. (1992). Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365.

[Maire, 2000] Maire, F. (2000). Bicephal reinforcement learning. In *Proceedings of the 7th International Conference on Neural Information Processing*, Taejon, Korea.

[Martinson et al., 2002] Martinson, E., Stoytchev, A., and Arkin, R. (2002). Robot Behavioral Selection Using Q_learning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Lausanne, Switzerland.

[Mataric, 1999] Mataric, M. (1999). Behavior-based robotics. In *The MIT encyclopedia of the cognitive sciences*. MIT Press.

[McCallum, 1995] McCallum, R. (1995). Instance-based state identification for reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS 7)*.

[McFarland, 1991] McFarland, D. (1991). What it means for robot behaviour to be adaptive. In Meyer, J.-A. and Wilson, S. W., editors, *From animals to animats*, pages 22–28. First International Conference on Simulation of Adaptive Behavior.

[Millan et al., 2002] Millan, J., Posenato, D., and Dedieu, E. (2002). Continuous-action q_learning. *Machine Learning*, 49:247–265.

[Moore, 1991] Moore, A. (1991). Variable resolution dynamic programming: Efficiently learning action maps on multivariate real-value state-spaces. In *Proceedings of the Eighth International Conference on Machine Learning*.

[Moravec, 1988] Moravec, H. (1988). *Mind Children: the future of robot and the human interaction*. Harvard University Press, 1988, 214 pages.

[Munos and Moore, 2002] Munos, R. and Moore, A. (2002). Variable resolution discretization in optimal control. *Machine Learning*, 49:291–323.

[Negahdaripour et al., 1999] Negahdaripour, S., Xu, X., and Jin, L. (1999). Direct estimation of motion from sea floor images for automatic station-keeping of submersible platforms. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 24(3):370–382.

[Nilsson, 1984] Nilsson, N. J. (1984). Shakey the robot. Technical Report 323, SRI International, Menlo Park, California.

[Nolfi, 1998] Nolfi, S. (1998). Evolutionary robotics: Exploiting the full power of self-organization. *Connection Science*, 10(3-4):167–183.

[Ormoneit and Sen, 2002] Ormoneit, D. and Sen, S. (2002). Kernel-based reinforcement learning. *Machine Learning*, 49:161–178.

[Parr, 1998] Parr, R. (1998). *Hierarchical Control and Learning for Markov Decision Processes*. PhD thesis, University of California, Berkeley, CA.

[Peng, 1995] Peng, J. (1995). Efficient Memory-based Dynamic Programming. *International Conference on Machine Learning*.

[Peng and Williams, 1994] Peng, J. and Williams, R. (1994). Incremental multi-step Q_learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, San Francisco, USA.

[Pfeifer and Scheier, 1999] Pfeifer, R. and Scheier, C. (1999). *Understanding Intelligence*. MIT Press.

[Poggio and Girosi, 1990] Poggio, T. and Girosi, F. (1990). Regularization algorithms for learning that are equivalent to multilayer networks. *Science*, 247:978–982.

[Powell, 1987] Powell, M. (1987). Radial basis functions for multivariate interpolation: A review. In *Algorithms for Approximation*, pages 143–167. Clarendon Press, Oxford.

[Precup et al., 2001] Precup, D., Sutton, R., and Dasgupta, S. (2001). Off-policy temporal-difference learning with function approximation. In *Proceedings of the 18th International Conference on Machine Learning*.

[Pyeatt and Howe, 1998] Pyeatt, L. and Howe, A. (1998). Decision tree function approximation in reinforcement learning. Technical Report TR CS-98-112, Colorado State University.

[Reynolds, 2002] Reynolds, S. (2002). *Reinforcement Learning with Exploration*. PhD thesis, University of Birmingham, United Kingdom.

[Ridao, 2001] Ridao, P. (2001). *A hybrid control architecture for an AUV*. PhD thesis, University of Girona.

[Rosenblatt and Payton, 1989] Rosenblatt, J. and Payton, D. (1989). A fine-grained alternative to the subsumption architecture for mobile robot control. In *Proceedings of the International Joint Conference on Neural Networks*, pages 317–323.

[Rosenschein and Kaelbling, 1986] Rosenschein and Kaelbling (1986). The synthesis of digital machines with provable epistemic properties. *TARK: Theoretical Aspects of Reasoning about Knowledge*, pages 83–98.

[Rosenstein and Barto, 2001] Rosenstein, M. and Barto, A. (2001). Robot weightlifting by direct policy search. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

[Rumelhart et al., 1986] Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning representations of back-propagation errors. *Nature*, 323:533–536.

[Ryan and Pendrith, 1998] Ryan, M. and Pendrith, M. (1998). RL-TOPs: An Architecture for Modularity and Re-Use in Reinforcement Learning. In

Fifteenth International Conference on Machine Learning, Madison, Wisconsin.

[Ryan and Reid, 2000] Ryan, M. and Reid, M. (2000). Learning to fly: An application of hierarchical reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 807–814, San Francisco, USA.

[Saffiotti et al., 1995] Saffiotti, A., Konolige, K., and Ruspini, E. H. (1995). A multivalued-logic approach to integrating planning and control. *Artificial Intelligence*, 76(1-2):481–526.

[Saito and Fukuda, 1994] Saito, F. and Fukuda, T. (1994). Learning architecture for real robot systems- extension of connectionist q_learning for continuous robot control domain. In *Proceedings of the International Conference on Robotics and Automation*, pages 27–32.

[Santamaria et al., 1998] Santamaria, J., Sutton, R., and Ram, A. (1998). Experiments with reinofrcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6:163–218.

[Santos, 1999] Santos, J. (1999). *Contribution to the study and design of reinforcement functions*. PhD thesis, Universidad de Buenos Aires, Universite d'Aix-Marseille III.

[Savitzky and Golay, 1964] Savitzky, A. and Golay, M. (1964). *Analytical Chemistry*, volume 36, pages 1627–1639.

[Shackleton and Gini, 1997] Shackleton, J. and Gini, M. (1997). Measuring the Effectiveness of Reinforcement Learning for Behavior-based Robotics. *Adaptive Behavior*, 5 (3/4):365–390.

[Singh et al., 1994a] Singh, S., Jaakkola, T., and Jordan, M. (1994a). Learning without state-estimation in partially observable markovian decision processes. In *Proceedings of the Eleventh International Conference on Machine Learning*, New Jersey, USA.

[Singh et al., 1994b] Singh, S., Jaakkola, T., and Jordan, M. (1994b). Reinforcement learning with soft state aggregation. In *Proceedings of the 1994 Conference on Advances in Neural Information Processing Systems*.

[Singh and Sutton, 1996] Singh, S. and Sutton, R. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158.

[Smart, 2002] Smart, W. (2002). *Making Reinforcement Learning Work on Real Robots*. PhD thesis, Department of Computer Science at Brown University, Rhode Island.

[Steels, 1992] Steels, L. (1992). The PDL reference manual. Technical Report 92-5, VUB AI Lab, Brussels, Belgium.

[Steels, 1993] Steels, L. (1993). Building agents with autonomous behaviour systems. In *The artificial route to artificial intelligence. Building situated embodied agents*. Lawrence Erlbaum Associates, New Haven.

[Sutton, 1984] Sutton, R. (1984). *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst.

[Sutton, 1988] Sutton, R. (1988). Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44.

[Sutton, 1996] Sutton, R. (1996). Generalization in reinforcement learning: Successful examples using sparce coarse coding. *Advances in Neural Information Processing Sustems*, 9:1038–1044.

[Sutton and Barto, 1998] Sutton, R. and Barto, A. (1998). *Reinforcement Learning, an introduction*. MIT Press.

[Sutton et al., 2000] Sutton, R., McAllester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems*, 12:1057–1063.

[Sutton et al., 1999] Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and Semi-MDPs: A framework for temporal abstraction in Reinforcement Learning. *Artificial Intelligence*, 112:181–211.

[Takahashi and Asada, 2000] Takahashi, Y. and Asada, M. (2000). Vision-guided behavior acquisition of a mobile robot by multi-layered reinforcement learning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

[Tesauro, 1992] Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8(3/4):257–277.

[Theocharous, 2002] Theocharous, G. (2002). *Hierarchical Learning and Planning in Partially Observable Markov Decision Processes*. PhD thesis, Michigan State University.

[Touzet, 1997] Touzet, C. (1997). Neural reinforcement learning for behavior synthesis. *Robotics and Autonomous Systems*, 22:251–281.

[Tsitsiklis and Roy, 1996] Tsitsiklis, J. and Roy, B. V. (1996). Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94.

[Tsitsiklis and Roy, 1997] Tsitsiklis, J. and Roy, B. V. (1997). Average cost temporal-difference learning. *IEEE Transactions on Automatic Control*, 42:674–690.

[Tyrell, 1993] Tyrell, T. (1993). *Computational mechanisms for action selection*. PhD thesis, University of Edinburgh, Scotland.

[Uther and Veloso, 1998] Uther, W. and Veloso, M. (1998). Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*.

[Vollbrecht, 1999] Vollbrecht, H. (1999). kd-Q learning with hierarchic generalization in state space. Technical Report SFB 527, Department of Neural Information Processing, University of Ulm, Germany.

[Watkins, 1989] Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, Cambridge University.

[Watkins and Dayan, 1992] Watkins, C. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.

[Weaver et al., 1998] Weaver, S., Baird, L., and Polycarpou, M. (1998). An Analytical Framework for Local Feedforward Networks. *IEEE Transactions on Neural Networks*, 9(3).

[Whitehead and Lin, 1995] Whitehead, S. and Lin, L. (1995). Reinforcement learning in non-markov environments. *Artificial Intelligence*, 73(1-2):271–306.

[Williams, 1992] Williams, R. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

[Witten, 1977] Witten, I. (1977). An adaptive optimal controller for discrete-time markov environments. *Information and Control*, 34:286–295.

[Zhang and Dieterich, 1995] Zhang, W. and Dietterich, T. (1995). A reinforcement learning approach to job-shop scheduling. In *IJCAI95*.

[Ziemke, 1998] Ziemke, T. (1998). Adaptive behavior in autonomous agents. *Presence*, 7(6):564–587.