

Relational Reinforcement Learning with Continuous Actions by Combining Behavioural Cloning and Locally Weighted Regression

Julio H. Zaragoza, Eduardo F. Morales

National Institute of Astrophysics, Optics and Electronics, Computer Science Department, Tonantzintla, México.
Email: {jzaragoza, emorales}@inaoep.mx

Received April 14th, 2010; revised April 25th, 2010; accepted April 30th, 2010.

ABSTRACT

Reinforcement Learning is a commonly used technique for learning tasks in robotics, however, traditional algorithms are unable to handle large amounts of data coming from the robot's sensors, require long training times, and use discrete actions. This work introduces TS-RRLCA, a two stage method to tackle these problems. In the first stage, low-level data coming from the robot's sensors is transformed into a more natural, relational representation based on rooms, walls, corners, doors and obstacles, significantly reducing the state space. We use this representation along with Behavioural Cloning, i.e., traces provided by the user; to learn, in few iterations, a relational control policy with discrete actions which can be re-used in different environments. In the second stage, we use Locally Weighted Regression to transform the initial policy into a continuous actions policy. We tested our approach in simulation and with a real service robot on different environments for different navigation and following tasks. Results show how the policies can be used on different domains and perform smoother, faster and shorter paths than the original discrete actions policies.

Keywords: Relational Reinforcement Learning, Behavioural Cloning, Continuous Actions, Robotics

1. Introduction

Nowadays it is possible to find service robots for many different tasks like entertainment, assistance, maintenance, cleanse, transport, guidance, etc. Due to the wide range of services that they provide, the incorporation of service robots in places like houses and offices has increased in recent years. Their complete incorporation and acceptance, however, will depend on their capability to learn new tasks. Unfortunately, programming service robots for learning new tasks is a complex, specialized and time consuming process.

An alternative and more attractive approach is to show the robot how to perform a task, rather than trying to program it, and let the robot to learn the fine details of how to perform the task. This is the approach that we follow on this paper.

Reinforcement Learning (RL) [1] has been widely used and suggested as a good candidate for learning tasks in robotics, e.g., [2,3,4,5-8,9]. This is mainly because it allows an agent, i.e., the robot, to “autonomously” develop a control policy for performing a new task while interacting with its environment. The robot only needs to

know the goal of the task, i.e., the final state, and a set of possible actions associated with each state.

The use and application of traditional RL techniques however, has been hampered by four main aspects: 1) vast amount of data produced by the robot's sensors, 2) large search spaces, 3) the use of discrete actions, and 4) the inability to re-use previously learned policies in new, although related, tasks.

Robots are normally equipped with laser range sensors, rings of sonars, cameras, etc., all of which produce a large number of readings at high sample rates creating problems to many machine learning algorithms.

Large search spaces, on the other hand, produce very long training times which is a problem for service robots where the state space is continuous and a description of a state may involve several variables. Researchers have proposed different strategies to deal with continuous state and action spaces, normally based on a discretization of the state space with discrete actions or with function approximation techniques. However, discrete actions produce unnatural movements and slow paths for a robot and function approximation techniques tend to be com-

putationally expensive. Also, in many approaches, once a policy has been learned to solve a particular task, it cannot be re-used on similar tasks.

In this paper, *TS-RRLCA (Two-Stage Relational Reinforcement Learning with Continuous Actions)*, a two stage method that tackles these problems, is presented. In the first stage, low-level information from the robot's sensors is transformed into a relational representation to characterize a set of states describing the robot's environment. With these relational states we applied a variant of the *Q-learning* algorithm to develop a relational policy with discrete actions. It is shown how the policies learned with this representation framework are transferable to other similar domains without further learning. We also use Behavioural Cloning [10], *i.e.*, human traces of the task, to consider only a subset of the possible actions per state, accelerating the policy learning process and obtaining a relational control policy with discrete actions in a few iterations. In the second stage, the learned policy is transformed into a relational policy with continuous actions through a fast Locally Weighted Regression (*LWR*) process.

The learned policies were successfully applied to a simulated and a real service robot for navigation and following tasks with different scenarios and goals. Results show that the continuous actions policies are able to produce smoother, shorter, faster and more similar paths to those produced by humans than the original relational discrete actions policies.

This paper is organized as follows. Section 2 describes related work. Section 3 introduces a process to reduce the data coming from the robot's sensors. Section 4 describes our relational representation to characterize states and actions. Sections 5 and 6 describe, respectively, the first and second stages of the proposed method. Section 7 shows experiments and results, Section 8 presents some discussion about our method and the experimental results, and Section 9 concludes and suggests future research directions.

2. Related Work

There is a vast amount of literature describing RL techniques in robotics. In this section we only review the most closely related work to our proposal.

In [8] a method to build relational macros for transfer learning in robot's navigation tasks is introduced. A macro consists of a finite state machine, *i.e.*, a set of nodes along with rulesets for transitions and action choices. In [11], a proposal to learn relational decision trees as abstract navigation strategies from example paths is presented. These two approaches use relational representations to transfer learned knowledge and use training examples to speed up learning, however, they only consider discrete actions.

In [9], the authors introduced a method that temporarily drives a robot which follows certain initial policy while some user commands play the role of training input to the learning component, which optimizes the autonomous control policy for the current task. In [2], a robot is tele-operated to learn sequences of state-action pairs that show how to perform a task. These methods reduce the computational costs and times for developing its control scheme, but they use discrete actions and are unable to transfer learned knowledge.

An alternative to represent continuous actions is to approximate a continuous function over the state space. The work developed in [12] is a Neural Network coupled with an interpolation technique that approximates *Q-values* to find a continuous function over all the search space. In [13], the authors use *Gaussian Processes* for learning a probabilistic distribution for a robot navigation problem. The main drawback of these methods is the computational costs and the long training times as they try to generate a continuous function over all of the search space.

Our method learns, through a relational representation, relational discrete actions policies able to transfer knowledge between similar domains. We also speed up and simplify the learning process by using traces provided by the user. Finally we use a fast *LWR* to transform the original discrete actions policy into a continuous actions policy. In the following sections we describe in detail the proposed method.

3. Natural Landmarks Representation

A robot senses and returns large amounts of data readings coming from its sensors while performing a task. In order to produce a smaller set of meaningful information *TS-RRLCA* uses a process based on [14,15] In [14] the authors described a process able to identify three kinds of natural landmarks through laser sensor readings: 1) discontinuities, defined as an abrupt variation in the measured distance of two consecutive laser readings (**Figure 1(a)**), 2) walls, identified using the Hough transform (**Figure 1(c)**), and 3) corners, defined as the location where two walls intersect and form an angle (**Figure 1(d)**). We also add obstacles identified through sonars and defined as any detected object within certain range (**Figure 1(e)**).

A natural landmark is represented by a tuple of four attributes: $(DL, \theta L, A, T)$. DL and θL are, respectively, the relative distance and orientation from the landmark to the robot. T is the type of landmark: l for left discontinuity, r for right discontinuity (see **Figure 1(b)**), c for corner, w for wall and o for obstacle. A is a distinctive attribute and its value depends on the type of landmark; for discontinuities A is depth (*dep*) and for walls A is its

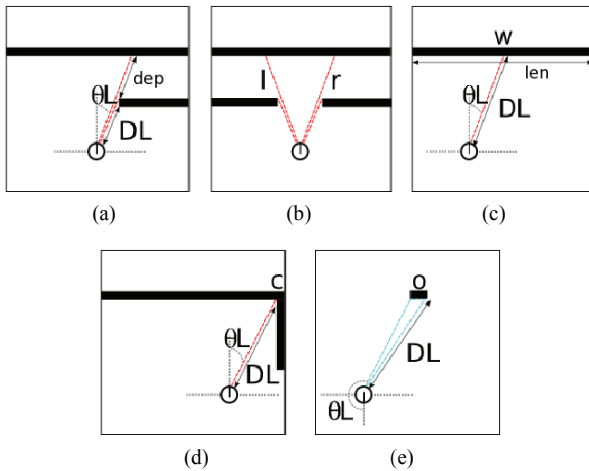


Figure 1. Natural landmarks types and associated attributes, (a) discontinuity detection; (b) discontinuity Types; (c) wall detection; (d) corner detection; (e) wall detection

length (*len*), for all of the other landmarks the *A* attribute is not used.

In [15] the data from laser readings is used to feed a clustering-based process which is able to identify the robot's actual location such as room, corridor and/or intersection (the location where rooms and corridors meet).

Figure 2 shows examples of the resulting location classification process.

Table 1 shows an example of the data after applying these processes to the laser and sonar readings from **Figure 3**. The robot's actual location in this case is *in-room*.

The natural landmarks along with the robot's actual location are used to characterize the relational states that describe the environment.

4. Relational Representations for States and Actions

A relational representation for states and actions has the advantage that it can produce relational policies that can be re-used in other, although similar, domains without any further learning. The idea is to represent states as sets of properties that can be used to characterize a particular situation which may be common to other states. For example, suppose the robot has some predicates that are able to recognize a room from its sensors' readings. If the robot has learned a policy to exit a room, then it can apply it to exit any recognizable room regardless of the current environment.

A relational state (r-state) is a conjunction of first order predicates. Our states are characterized by the following predicates which receive as parameters a set of values such as those shown in **Table 1**.

1) *place*: This predicate returns the robot's location, which can be in-room, in-door, in-corridor and in-intersection.

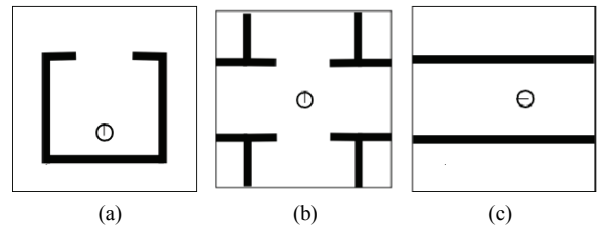


Figure 2. Locations detected through a clustering processes, (a) room; (b) intersection; (c) corridor

Table 1. Identified natural landmarks from the sensor's readings from Figure 3

N	<i>DL</i>	θ_L	<i>A</i>	<i>T</i>
1	0.92	-17.60	4.80	<i>r</i>
2	1.62	-7.54	3.00	<i>l</i>
3	1.78	17.60	2.39	<i>l</i>
4	0.87	-35.70	1.51	<i>w</i>
5	4.62	-8.55	1.06	<i>w</i>
6	2.91	-6.54	1.88	<i>w</i>
7	1.73	23.63	0.53	<i>w</i>
8	2.13	53.80	2.38	<i>w</i>
9	5.79	-14.58	0.00	<i>c</i>
10	2.30	31.68	0.00	<i>c</i>
11	1.68	22.33	0.00	<i>c</i>
12	1.87	-170.00	0.00	<i>o</i>
13	1.63	-150.00	0.00	<i>o</i>
14	1.22	170.00	0.00	<i>o</i>
15	1.43	150.00	0.00	<i>o</i>

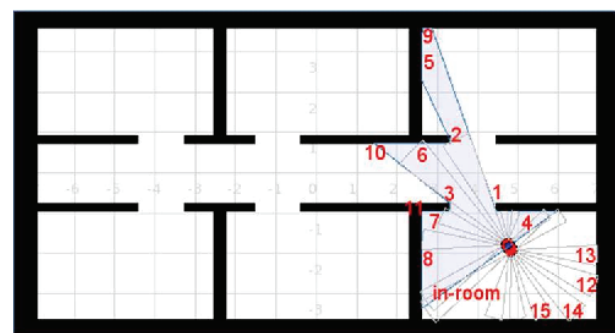


Figure 3. Robot sensing its environment through laser and sonar sensors and corresponding natural landmarks

2) *doors-detected*: This predicate returns the orientation and distance to doors. A door is characterized by identifying a right discontinuity (r) followed by a left discontinuity (l) from the natural landmarks. The door's orientation angle and distance values are calculated by averaging the values of the right and left discontinuities angles and distances. The discretized values used for door orientation are: *right* (door's angle between -67.5° and -112.5°), *left* (67.5° to 112.5°), *front* (22.5° to -22.5°), *back* (157.5° to -157.5°), *right-back* (-112.5° to -157.5°), *right-front* (-22.5° to -67.5°), *left-back* (112.5° to 157.5°) and *left-front* (22.5° to 67.5°). The discretized values used for distance are: *hit* (door's distance between 0 m and 0.3 m), *close* (0.3 m to 1.5 m), *near* (1.5 m to 4.0 m) and *far* (door's distance > 4.0 m).

For example, if the following discontinuities are obtained from the robot's sensors (shown in **Table 1**: [0.92, -17.60 , 4.80, r], [1.62, -7.54 , 3.00, l]), the following predicate is produced:

doors-detected ([*front*, *close*, -12.57 , 1.27])

This predicate corresponds to the orientation and distance descriptions of a detected door (shown in **Figure 3**), and for every pair of right and left discontinuities a list with these orientation and distance descriptions is generated.

3) *walls-detected*: This predicate returns the length, orientation and distance to walls (type w landmarks). Possible values for wall's length are: *small* (length between 0.15 m and 1.5 m), *medium* (1.5 m to 4.0 m) and *large* (wall's size or length > 4.0 m). The discrete values used for orientation and distance are the same as with doors and the same goes for predicates *corners-detected* and *obstacles-detected* described below.

4) *corners-detected*: This predicate returns the orientation and distance to corners (type c landmarks).

5) *obstacles-detected*: This predicate returns the orientation and distance to obstacles (type o landmarks).

6) *goal-position*: This predicate returns the relative orientation and distance between the robot and the current goal. Receives as parameter the robot's current position and the goal's current position, though a trigonometry process, the orientation and distance values are calculated and then discretized as same as with doors.

7) *goal-reached*: This predicate indicates if the robot is in its goal position. Possible values are *true* or *false*.

The previous predicates tell the robot if it is in a room, a corridor or an intersection, detect walls, corners, doors, obstacles and corridors and give a rough estimate of the direction and distance to the goal. Analogous to r-states, r-actions are conjunctions of the following first order logic predicates that receive as parameters the odometer's speed and angle readings.

8) *go*: This predicate returns the robot's actual moving

action. Its possible values are *front* (speed > 0.1 m/s), *nil* (-0.1 m/s $<$ speed < 0.1 m/s) and *back* (speed < -0.1 m/s).

9) *turn*: This predicate returns the robot's actual turning angle. Its possible values are *slight-right* ($-45^\circ <$ angle $< 0^\circ$), *right* ($-135^\circ <$ angle $\leq -45^\circ$), *far-right* (angle $\leq -135^\circ$), *slight-left* ($45^\circ >$ angle $> 0^\circ$), *left* ($135^\circ >$ angle $\geq 45^\circ$), *far-left* (angle $\geq 135^\circ$) and *nil* (angle = 0°).

Table 2 shows an r-state-r-action pair generated with the previous predicates which corresponds to the values from **Table 1**. As can be seen, some of the r-state predicates (doors, walls, corners and obstacles detection) besides returning the nominal descriptions; they also return the numerical values of every detected element. The r-action predicates also return the odometer's speed and the robot's turning angle. These numerical values are used in the second stage of the method as described in Section 6. The discretized or nominal values, *i.e.*, the r-states and r-actions descriptions, are used to learn a relational policy through *rQ-learning* as described below.

5. TS-RRLCA First Stage

TS-RRLCA starts with a set of human traces of the task that we want the robot to learn. A trace $T_k = \{f_{k1}, f_{k2}, \dots, f_{kn}\}$ is a log of all the odometer, laser and sonar sensor's readings of the robot while it is performing a particular task. A trace-log is divided in frames; every frame is a register with all the low-level values of the robot's sensors ($f_{kj} = \{\text{laser1} = 2.25, \text{laser2} = 2.27, \text{laser3} = 2.29, \dots, \text{sonar1} = 3.02, \text{sonar2} = 3.12, \text{sonar3} = 3.46, \dots, \text{speed} = 0.48, \text{angle} = 87.5\}$) at a particular time.

Once a set of traces (T_1, T_2, \dots, T_m) has been given to TS-RRLCA, every frame in the traces, is transformed

Table 2. Resulting r-state-r-action pair from the values in Table 1

<i>r-state</i>	<i>r-action</i>
<i>Place (in-room)</i> ,	<i>go (nil, 0.0)</i> ,
<i>doors-detected</i> ([[<i>front</i> , <i>close</i> , -12.57 , 1.27]]),	<i>turn (right, 92)</i> .
<i>walls-detected</i> ([[<i>right-front</i> , <i>close</i> , <i>medium</i> , -35.7 , 0.87], [<i>front</i> , <i>far</i> , <i>small</i> , -8.55 , 4.62], [<i>front</i> , <i>near</i> , <i>medium</i> , -6.54 , 2.91], [<i>left-front</i> , <i>near</i> , <i>small</i> , 23.63, 1.73], [<i>left-front</i> , <i>near</i> , <i>medium</i> , 53.80, 2.13]]),	
<i>corners-detected</i> ([[<i>front</i> , <i>far</i> , -14.58 , 5.79], [<i>front</i> , <i>near</i> , 31.68, 2.30], [<i>left-front</i> , <i>near</i> , 22.33, 1.68]]),	
<i>obstacles-detected</i> ([[<i>back</i> , <i>near</i> , -170.00 , 1.87], [<i>right-back</i> , <i>near</i> , -150.00 , 1.63], [<i>back</i> , <i>close</i> , 170.00, 1.22], [<i>left-back</i> , <i>close</i> , 150.00, 1.43]]),	
<i>goal-position</i> ([<i>right-front</i> , <i>far</i>]),	
<i>goal-reached</i> (<i>false</i>).	

into natural landmarks along with the robot's location. This transformed frames are given to the first order predicates to evaluate the set of relations, *i.e.*, generate the corresponding r-state and r-action (as the one shown in **Table 2**). By doing this, every frame from the traces corresponds to an r-state-r-action pair and every one of these pairs is stored in a database (*DB*).

Algorithm 1 gives the pseudo-code for this Behavioural Cloning (*BC*) approach. At the end of this *BC* approach, the *DB* contains r-state-r-action pairs corresponding to all the frames in the set of traces.

As the traces correspond to different examples of the same task and as they might have been generated by different users, there can be several r-actions associated to the same r-state. *RL* is used to develop a control policy that selects the best r-action in each r-state.

5.1 Relational Reinforcement Learning

The *RL* algorithm selects the r-action that produces the greatest expected accumulated reward among the possible r-actions in each r-state. Since we only used information from traces only a subset of all the possible r-actions, for every r-state, are considered which significantly reduces the search space. In a classical reinforcement learning framework a set of actions (*A*) is predefined for all of the possible states (*S*). Every time the agent reaches a new state, it must select one action from all of the possible actions in *S* to reach a new state. In our *RL* approach when the robot reaches a new r-state, it chooses one action from a subset of r-actions performed in that r-state in the traces.

In order to execute actions, each time the robot reaches an r-state, it retrieves from the *DB* the associated r-actions. It chooses one according to its policy and the associated nominal value of the selected r-action is transformed into one of the following values:

1) For the predicate *go*, if the description of the r-action is *front* the corresponding value is 0.5 m/s, for *back* the corresponding value is -0.5 m/s, and for *nil* the value is 0.0 m/s.

2) For the predicate *turn* the values are: *slight-right* = -45° , *right* = -90° , *far-right* = -135° , *slight-left* = 45° , *left* = 90° , *far-left* = 135° and *nil* = 0° .

Once the r-action has been chosen and executed the robot gets into a new r-state and the previous process is repeated until reaching a final r-state.

Algorithm 2 gives the pseudo-code for this *rQ-learning* approach. This is very similar to the *Q-learning* algorithm, except that the states and actions are characterized by relations.

By using only the r-state-r-action pairs from the traces (stored in the *DB*) our policy generation process is very fast and thanks to our relational representation, policies can be transferred to different, although similar office or

Algorithm 1. Behavioural cloning algorithm

Require: T_1, T_2, \dots, T_n : Set of n traces with examples of the task the robot has to learn.

Ensure: *DB*: r-state-r-action pairs database.

for $i = 1$ to n **do**

$k \leftarrow$ number of frames in the trace i

for $j = 1$ to k **do**

Transform $frame_j$ (frame j from trace i) into their corresponding natural landmarks and into the corresponding robot's location.

Use the natural landmarks and the robot's location to get the corresponding r-state (through the first order predicates).

Use the robot's speed and angle to get the corresponding r-action.

$DB \leftarrow DB \cup \{r\text{-state, r-action}\}$. % Each register in *DB* contains an r-state with its corresponding r-action

End for

End for

Algorithm 2. rQ-learning algorithm

Require: *DB*, r-state-r-action pairs database.

Ensure: function *Q*: discrete actions relational control policy.

Initialize $Q(S_i, A_i)$ arbitrarily

Repeat

$s_t \leftarrow$ robot's sensors readings values.

Transform s_t into its corresponding natural landmarks and into the corresponding robot's location.

$S_t \leftarrow$ r-state (s_t) % Use those natural landmarks and the robot's location to get the corresponding r-state (through the first order predicates).

for each step of the episode **do**

Search the r-state (S_t) description in *DB*.

for each register in *DB* which contains the r-state (S_t) description **do**

Get its corresponding r-actions

End for

Select an r-action A_t to be executed in S_t through an action selection policy (e.g., ϵ -greedy).

Execute action A_t , observe r_{t+1} and s_{t+1}

Transform s_{t+1} into its corresponding natural landmarks and into the corresponding robot's location.

$S_{t+1} \leftarrow$ r-state (s_{t+1}) % Use those natural landmarks and the robot's location to get the corresponding r-state (through the first order predicates).

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(r_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$

$S_t \leftarrow S_{t+1}$

End for

until S_t is terminal

house-like environments. In the second stage, this discrete actions policy is transformed into a continuous actions policy.

6. TS-RRLCA Second Stage

This second stage refines the coarse actions from the previously generated discrete actions policy. This is achieved using Locally Weighted Regression (*LWR*).

The idea is to combine discrete actions' values given by the policy obtained in the first stage with the action's values previously observed in the traces. This way the robot follows the policy, developed in the first stage, but the actions are tuned through a *LWR* process. What we do is to detect the robot's actual r-state, then, for this r-state the previously generated discrete actions policy determines the action to be executed (**Figure 4(a)**). Before performing the action, the robot searches in the *DB* for all the registers that share this same r-state description (**Figure 4(b)**). Once found, the robot gets all of the numeric orientation and distance values from these registers. This orientation and distance values are used to perform a triangulation process. This process allows us to estimate the relative position of the robot from previous traces with respect to the robot's actual position. Once this position has been estimated, a weight is assigned to the distance of the robot from the traces with respect to the actual robot's position (**Figure 4(c)**). These weights are used to perform the *LWR* that produces continuous r-actions (**Figure 4(d)**).

The triangulation process is performed as follows. The robot *R* in the actual r-state (**Figure 5(a)**), senses and detects elements *E* and *E'* (which can be a door, a corner, a wall, etc.). Each element has a relative distance (*a* and

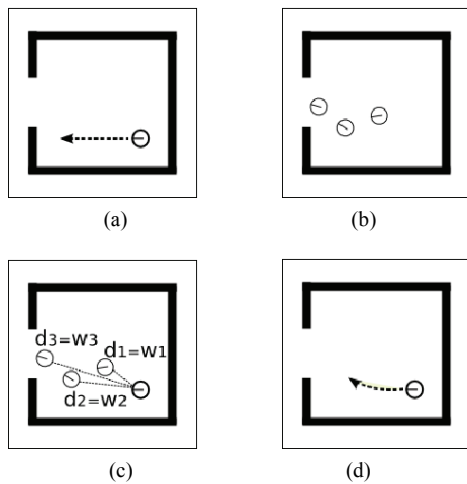


Figure 4. Continuous actions developing process, (a) r-state and corresponding r-action; (b) a trace segment; (c) distances and weights; (d) resulting continuous action

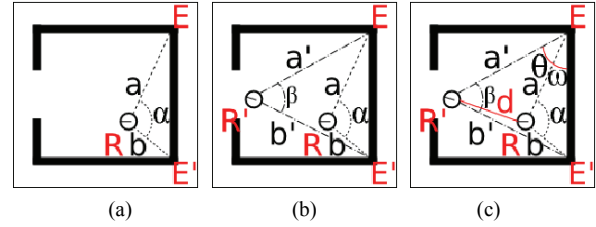


Figure 5. Triangulation process, (a) R robot's r-state and identified elements; (b) R' robot from traces; (c) elements to be calculated

b) and a relative angle with respect to *R*. The angles are not directly used in this triangulation process, what we use is the absolute difference between these angles (α). The robot reads from the *DB* all the registers that share the same r-state description, *i.e.*, that have the same r-state discretized values. The numerical angle and distance values associated with these *DB* registers correspond to the relative distances (a' and b') from the robot *R'* in a trace relative to the same elements *E* and *E'*, and the corresponding angle β (**Figure 5(b)**). In order to know the distance between *R* and *R'* (d) through this triangulation process, Equations 1, 2, 3, and 4 are applied.

$$\overline{EE'} = \sqrt{a^2 + b^2 - 2ab \cos(\alpha)} : \text{Distance between } E \text{ and } E'. \quad (1)$$

$$\theta + \omega = \arcsin(a' / \overline{EE'}) : \text{Angle between } a' \text{ and } \overline{EE'}. \quad (2)$$

$$\omega = \arcsin(a / \overline{EE'}) : \text{Angle between } a \text{ and } \overline{EE'}. \quad (3)$$

$$d = \sqrt{a^2 + a'^2 - 2aa' \cos(\theta)} : \text{Distance between } R \text{ and } R'. \quad (4)$$

These four equations give the relative distance (d) between *R* and *R'*. Once this value is calculated, a kernel is used to assign a weight (w). This weight is multiplied by the speed and angle values of the *R'* robot's r-action. The resulting weighted speed and angle values are then added to the *R* robot's speed and angle values. This process is applied to every register read from the *DB* whose r-state description is the same as *R* and is repeated every time the robot reaches a new r-state.

To summarize this process, each time the robot reaches an r-state and chooses an r-action according to the learned policy; it retrieves from the *DB* all the registers that share the same r-state. It uses the numerical values of the retrieved r-states to evaluate the relative distance of the position of the robot in a trace to the position of the robot in the actual r-state. Once all the distance values (d_i) are calculated we apply a Gaussian kernel (Equation 5) to obtain a weight w_i . We tried different kernels, e.g., Tricubic kernel, and results were better with

Gaussian kernel but further tests are needed.

$$w_i(d_i) = \exp(-d_i^2): \text{Gaussian kernel.} \quad (5)$$

Then, every weight w_i is multiplied by the corresponding speed and angle values ($w_i \times speed_{DBi}$ and $w_i \times angle_{DBi}$) of the r-state-r-action pairs retrieved from the *DB*. The resulting values are added to the discrete r-action ($rA_i = \{disc_speed, disc_angle\}$) values of the policy obtained in the first stage in order to transform this discrete r-action into a continuous action (Equations 6 and 7) that is finally executed by the robot. This process is performed in real-time every time the robot reaches a new r-state.

$$continuous_speed = disc_speed + \{w_1 \times speed_{DB1}\} + \{w_2 \times speed_{DB2}\} + \dots + \{w_n \times speed_{DBn}\}: \text{LWR for developing the continuous speed.} \quad (6)$$

$$continuous_angle = disc_angle + \{w_1 \times angle_{DB1}\} + \{w_2 \times angle_{DB2}\} + \dots + \{w_n \times angle_{DBn}\}: \text{LWR for developing the continuous angle.} \quad (7)$$

The weights are directly related to the distances between the robot in the actual r-state to the r-states to the robot in the human traces stored in the *DB*. The closer the human traces registers are to the robot's actual position, the higher the influence they have in transforming the discrete action into a continuous action.

The main advantage of our approach is the simple and fast strategy to produce continuous actions policies that, as will be seen in the following section, are able to produce smoother and shorter paths in different environments.

7. Experiments

For testing purposes, two types of experiments were performed:

1) Learning Curves: In these experiments we compared the number of iterations it takes our method *TS-RRLCA* to learn a policy against classical Reinforcement Learning (*RL*) and against the *rQ-learning* algorithm (shown in Algorithm 2) without using Behavioural Cloning approach, which we will refer to as Relational Reinforcement Learning (*RRL*).

2) Performance: In these experiments we compared the performance of the policies learned through *TS-RRLCA* with discrete actions against the policies learned through *TS-RRLCA* with continuous actions. Particularly we tested: How close the tasks are to the tasks performed by the user and how close the tasks are from obstacles in the environment.

3) Execution times.

These experiments were carried out in simulation (*Player/Stage* [16]) and with a real robot which is an *ActivMedia GuiaBot* (www.activrobots.com).

Both robots (simulated and real) are equipped with a 180° front laser sensor and an array of four back sonars (located at -170°, -150°, 150° and 170°).

The laser range is 8.0 m and for the sonars is 6.0 m. The tasks in these experiments are “navigating through the environment” and “following an object”.

The policy generation process was carried out in the map shown in **Figure 6** (Map 1 with size 15.0 m × 9.0 m). For each of the two tasks a set of 20 traces was generated by the user. For the navigation tasks, the robot and the goal's global position (for the *goal-position* predicate) were calculated using the work developed in [14]. For the following tasks we used a second robot which orientation and angle were calculated through the laser sensor. **Figure 6** shows an example of navigation and a following trace.

To every set of traces, we applied our approach to abstract the r-states and induce the subsets of relevant r-actions. Then, *rQ-learning* was applied to learn the policies. For generating the policies, *Q-values* were initialized to -1, $\epsilon = 0.1$, $\gamma = 0.9$ and $\alpha = 0.1$. Positive reinforcement, r (+100) was given when reaching a goal (within 0.5 m), negative reinforcement (-20) was given when the robot hits an element and no reward value was given otherwise (0).

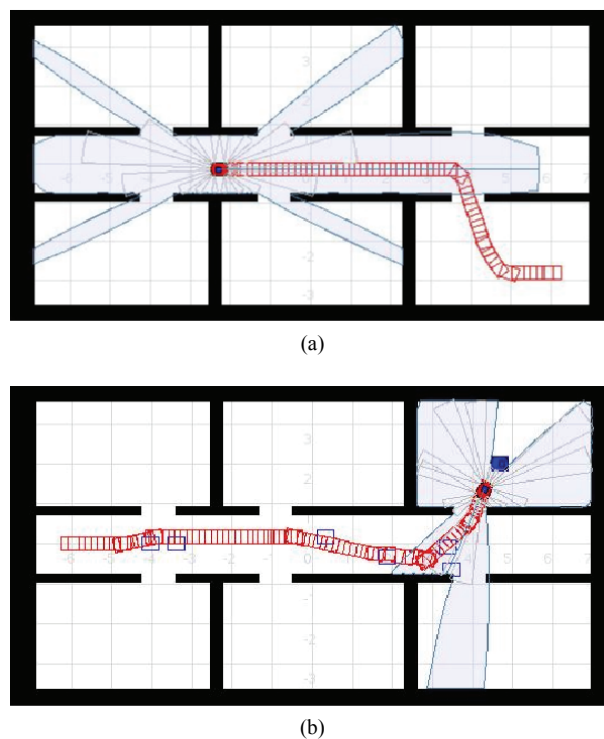


Figure 6. Traces examples, (a) navigation trace; (b) following trace

7.1 Learning Curves

Our method (*TS-RRLCA*) was compared in the number of iterations it takes to develop a control policy, against classical reinforcement learning (*RL*) and against the *rQ-learning* algorithm described in **Algorithm 2**, considering all the possible r-actions (the 8 r-actions, shown in Section 4) per r-state (*RRL*).

For developing the “navigating through the environment” policy with *RL* we discretized the state and action space as follows: the training Map 1, depicted in **Figure 6**, was divided in states of 25 cm². Since this map’s size is 15 m × 9 m, the number of states is 2,160. In every state, one of the next 8 actions can be chosen to get into a new state which gives a total of 17,280 state-action pairs (This set of 8 actions correspond to the set of 8 r-actions we used in our *rQ-learning* algorithm).

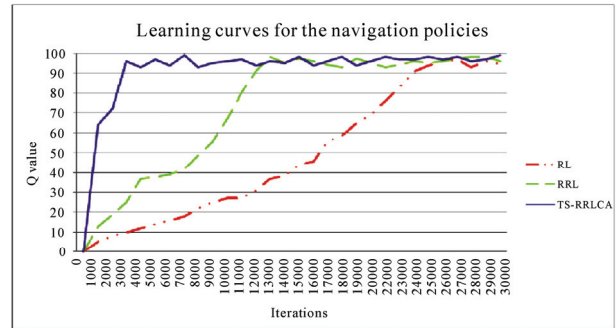
- 1) *front*: robot goes forward 25 cm.
- 2) *back*: robot goes back 25 cm.
- 3) *slight-right*: robot turns -45° .
- 4) *right*: robot turns -90° .
- 5) *far-right*: robot turns -135° .
- 6) *slight-left*: robot turns 45° .
- 7) *left*: robot turns 90° .
- 8) *far-left*: robot turns 135° .

For developing the navigation policy with *RRL* we have 655 r-states with 8 possible r-actions for each r-state, this gives a total of 5,240 possible r-state-r-action pairs. The number of r-states corresponds to the total number of r-states in which the training map can be divided.

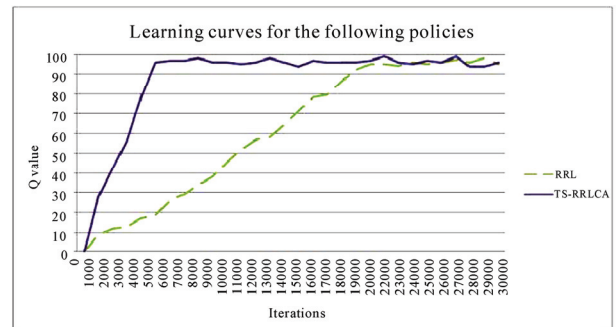
For developing the navigation policy with *TS-RRLCA* we used 20 navigation traces from which 934 r-state-r-action pairs were obtained. As can be seen, by using our Behavioural Cloning approach we significantly reduced the number of state-action pairs to consider in the learning process.

In each trace, every time our program performed a robot’s sensors reading, which includes laser, sonars and odometer, we first transformed the laser and sonar readings into natural landmarks (as described in Section 3). These natural landmarks are sent to the predicates to generate the corresponding r-state, the corresponding r-action is generated by using the odometer’s readings (as described in Section 4). This gives an r-state-r-action pair such as the one shown in **Table 2**.

Figure 7(a) shows the learning curves of *RL*, *RRL* and *TS-RRLCA* for a navigation policy. They show the accumulated Q-values every 1,000 iterations. As can be seen from this figure, the number of iterations for developing an acceptable navigation policy with *TS-RRLCA* is very low when compared to *RRL* and is significantly lower when compared to *RL*. It should be noted that the navigation policy learned with *RL* only works for going to a single destination state while the policies learned with our relational representation can be used to reach



(a)



(b)

Figure 7. Learning curves comparison, (a) learning curves for the navigation policies; (b) learning curves for the following policies

several destination places in different environments.

For developing the “following an object” policy, the number of r-state-r-action pairs using our relational representation (*RRL*) is 3,149, while the number of r-state-r-action pairs using the same representation but with behavioural cloning (*TS-RRLCA*) is 1,406, obtained from 20 traces. For the following policy we only compared our approach against *RRL*.

Figure 7(b) shows the learning curves of these two methods. As can be seen the number of iterations that our method needs to generate an acceptable following policy is much lower than *RRL*.

To generate the continuous actions policies, *LWR* was applied using the Gaussian kernel for estimating weights. In the next section we compare the traces performed with the discrete actions policy with those using continuous actions.

7.2 Performance Tests

Once the policies were learned, experiments were executed in the training map with different goal positions and in two new and unknown environments for the robot (Map 2 shown in **Figure 8** with size 20.0 m × 15.0 m and Map 3, shown **Figure 9**, which corresponds to the real robot’s environment whose size is 8.0 m × 8.0 m). A total of 120

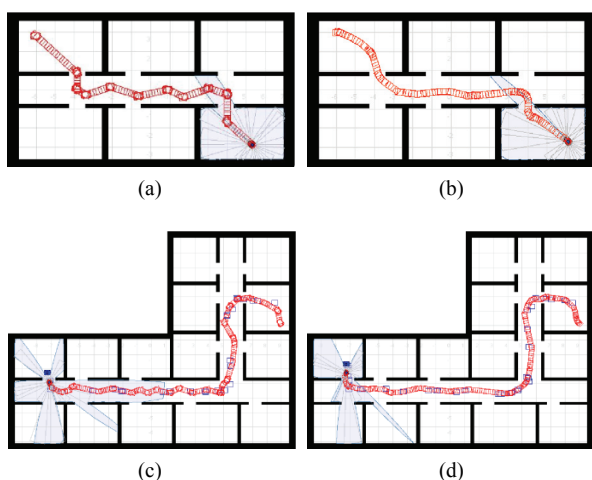


Figure 8. Navigation and following tasks performed with the policies learned with *TS-RRLCA*, (a) navigation task with discrete actions, Map 1; (b) navigation task with continuous actions, Map 1; (c) following task with discrete actions, Map 2; (d) following task with continuous actions, Map 2

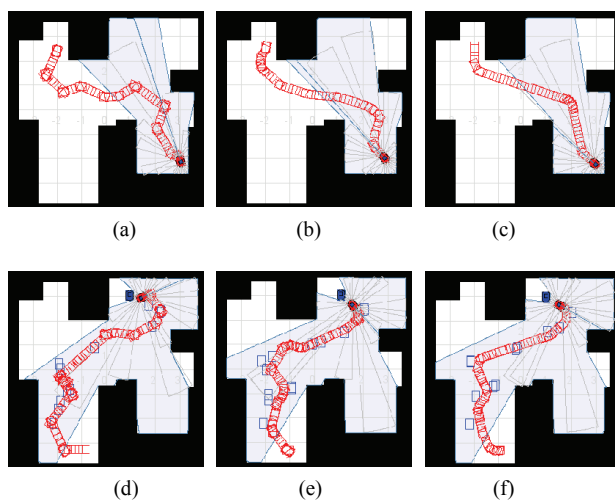


Figure 9. Navigation and following tasks examples from Map 3, (a) navigation task with discrete actions; (b) navigation task with continuous actions; (c) navigation task performed by user; (d) following task with discrete actions; (e) following task with continuous actions; (f) following task performed by user

experiments were performed: 10 different navigation and 10 following tasks in each map, each of these tasks were executed first with the discrete actions policy from the first stage and then with the continuous actions policy from the second stage. Each task has a different distance to cover and required the robot to traverse through different places. The minimum distance was 2 m. (Manhattan distance), and it was gradually increased up to 18 m.

Figure 8 shows navigation (on the top) and a following task (on the bottom) performed with discrete and con-

tinuous actions policies respectively.

Figure 9 shows navigation and a following task performed with the real robot, with the discrete and with the continuous actions policy.

As we only use the r-state-r-action pairs from the traces developed by the user in Map 1 (as the ones shown in **Figure 6**), when moving the robot to the new environments (Map 2 and Map 3), sometimes, it was not able to match the new map's r-state with one of the previously visited states by the user in the traces examples. So when the robot reached an unseen r-state, it asked the user for guidance. Through a joystick, the user indicates the robot which r-action to execute in the unseen r-state and the robot saves this new r-state-r-action pair in the *DB*. Once the robot reaches a known r-state, it continues its task. As the number of experiments increased in these new maps, the number of unseen r-states was reduced. **Table 3** shows the number of times the robot asked for guidance in each map and with each policy.

Figure 10(a) shows results in terms of the quality of the performed tasks with the real robot. This comparison is made against tasks performed by humans (For **Figures 10(a)**, **10(b)** and **11**, the following acronyms are used, NPDA: Navigation Policy with Discrete Actions, NPCA: Navigation Policy with Continuous Actions, FPDA: Following Policy with Discrete Actions and FPCA: Following Policy with Continuous Actions).

All of the tasks performed in the experiments with the real robot, were also performed by a human using a joystick (**Figures 9(c)** and **9(f)**), and logs of the paths were saved. The graphic shows the normalized quadratic error between these logs and the trajectories followed by the robot with the learned policy.

Figure 10(b) shows results in terms of how closer the robot gets to obstacles. This comparison is made using the work developed in [17]. In that work, values were given to the robot accordingly to its proximity to objects or walls. The closer the robot is to an object or wall the higher cost it is given. Values were given as follows: if the robot is very close to an object (between 0 m and 0.3 m) a value of -100 is given, if the robot is close to an object (between 0.3 m and 1.0 m) a value of -3 is given, if the robot is near an object (between 1.0 m and 2.0 m) a value of -1 is given, otherwise a value of 0 is given. As can be seen in the figure, quadratic error and penalty values for continuous actions policies are lower than those with discrete actions.

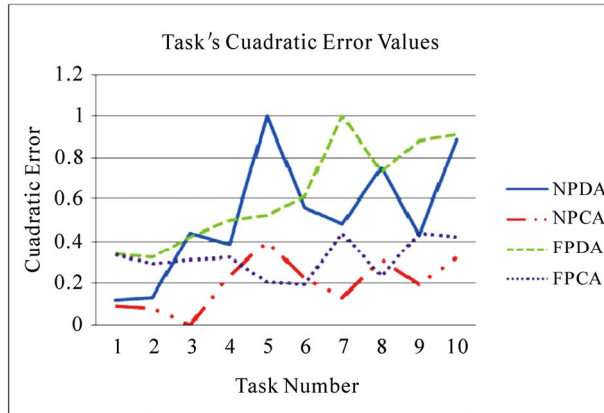
Policies developed with this method allow a close-to-human execution of the tasks and tend to use the available free space in the environment.

7.3 Execution Times

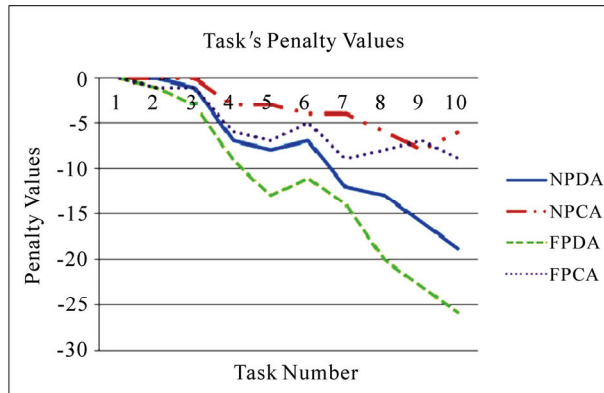
Execution times with the real robot were also registered. We compared the time that takes to the robot to perform a tasks with discrete actions against tasks performed with

Table 3. Number of times the robot asked for guidance in the experiments

Policy type	Map 1	Map 2	Map 3	Total
Navigation	2	6	14	22
Following	7	15	27	49



(a)



(b)

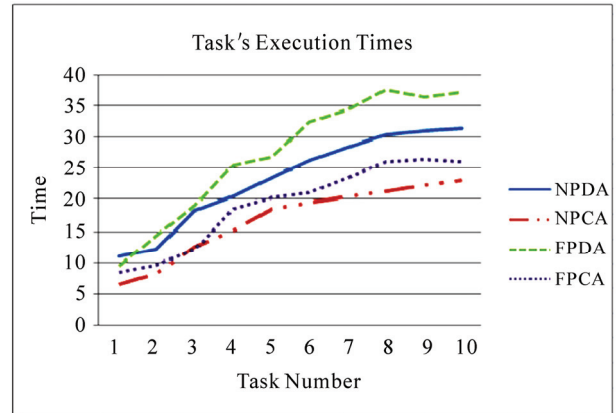
Figure 10. Navigation and following results of the tasks performed by the real robot, (a) quadratic error values; (b) penalty values

continuous actions. Every navigating or following experiment, that we carried out, was performed first with discrete actions and then with continuous actions.

As can be seen in **Figure 11**, continuous actions policies execute faster paths than the discrete actions policy despite our triangulation and *LWR* processes.

8. Discussion

In this work, we introduced a method for teaching a robot how to perform a new task from human examples. Experimentally we showed that tasks learned with this method and performed by the robot are very similar to

**Figure 11. Execution times results**

those tasks when performed by humans. Our two-stage method learns, in the first stage, a rough control policy which, in the second stage, is refined, by means of Locally Weighted Regression (*LWR*), to perform continuous actions. Given the nature of our method we can not guaranteed to generate optimal policies. There are two reasons why this can happen: 1) the actions performed by the user in the traces may not part of the optimal policy. In this case, the algorithm will follow the best policy given the known actions but will not be able to generate an optimal policy. 2) The *LWR* approach can take the robot to states that are not part of the optimal policy, even if they are smoother and closer to the user's paths. This has not represented a problem in the experiments that we performed.

With the Behavioural Cloning approach we observed around a 75% reduction in the state-action space. This reduction depends on the traces given by the user and on the training environment. In a hypothetical optimal case, where a user always performs the same action in the same state, the system only requires to store one action per state. This, however, is very unlikely to happen due to the continuous state and action space and the uncertainty in the outcomes of the actions perform with a robot.

9. Conclusions and Future Work

In this paper we described an approach that automatically transformed in real-time low-level sensor information into a relational representation. We used traces provided by a user to constraint the number of possible actions per state and use a reinforcement learning algorithm over this relational representation and restricted state-action space to learn in a few iterations a policy. Once a policy is learned we used *LWR* to produce a continuous actions policy in real time. It is shown that the learned policies with continuous actions are more similar to those performed by users (smoother), and are safer and faster than the policies obtained with discrete actions. Our relational policies are expressed in terms of more natural descriptions, such as

rooms, corridors, doors, walls, etc., and can be re-used for different tasks and on different house or office-like environments. The policies were learned on a simulated environment and later tested on a different simulated environment and on an environment with a real robot with very promising results.

There are several future research directions that we are considering. In particular, we would like to include an exploration strategy to identify non-visited states to complete the traces provided by the user. We are also exploring the use of voice commands to indicate the robot which action to take when it reaches an unseen state.

10. Acknowledgements

We thank our anonymous referees for their thoughtful and constructive suggestions. The authors acknowledge to CONACyT the support provided through the grant for MSc. studies number 212418 and in part by CONACyT project 84162.

REFERENCES

- [1] C. Watkins, "Learning from Delayed Rewards," PhD Thesis, University of Cambridge, England, 1989.
- [2] K. Conn and R. A. Peters, "Reinforcement Learning with a Supervisor for a Mobile Robot in a Real-World Environment," *International Symposium on Computational Intelligence in Robotics and Automation*, Jacksonville, FL, USA, June 20-23, 2007, pp. 73-78.
- [3] E. F. Morales and C. Sammut, "Learning to Fly by Combining Reinforcement Learning with Behavioural Cloning," *Proceedings of the Twenty-First International Conference on Machine Learning*, Vol. 69, 2004, p.76.
- [4] J. Peters, S. Vijayakumar and S. Schaal, "Reinforcement Learning for Humanoid Robotics," *Proceedings of the Third IEEE-RAS International Conference on Humanoid Robots*, Karlsruhe, Germany, September 2003, pp. 29-30.
- [5] W. D. Smart, "Making Reinforcement Learning Work on Real Robots," Department of Computer Science at Brown University Providence, Rhode Island, USA, 2002.
- [6] W. D. Smart and L. P. Kaelbling, "Effective Reinforcement Learning for Mobile Robots," *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, Washington, DC, USA, 2002, pp.3404-3410.
- [7] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An introduction," MIT Press, Cambridge, MA, 1998.
- [8] L. Torrey, J. Shavlik, T. Walker and R. Maclin, "Relational Macros for Transfer in Reinforcement Learning," *Lecture Notes in Computer Science*, Vol. 4894, 2008, pp. 254-268.
- [9] Y. Wang, M. Huber, V. N. Papudesi and D. J. Cook, "User-guided Reinforcement Learning of Robot Assistive Tasks for an Intelligent Environment," *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 1, 2003, pp. 27-31.
- [10] I. Bratko, T. Urbancic and C. Sammut, "Behavioural Cloning of Control Skill," In: R. S. Michalski, I. Bratko and M. Kubat, Ed., *Machine Learning and Data Mining*, John Wiley & Sons Ltd., Chichester, 1998, pp. 335-351.
- [11] A. Cocora, K. Kersting, C. Plagemann, W. Burgard and L. De Raedt, "Learning Relational Navigation Policies," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Beijing, China, October 9-15, 2006, pp. 2792-2797.
- [12] C. Gaskett, D. Wettergreen and A. Zelinsky, "Q-learning in Continuous State and Action Spaces," *In Australian Joint Conference on Artificial Intelligence*, Australia, 1999, pp. 417-428.
- [13] F. Aznar, F. A. Pujol, M. Pujol and R. Rizo, "Using Gaussian Processes in Bayesian Robot Programming," *Lecture notes in Computer Science*, Vol. 5518, 2009, pp. 547-553.
- [14] S. F. Hernández and E. F. Morales, "Global Localization of Mobile Robots for Indoor Environments Using Natural Landmarks," *IEEE Conference on Robotics, Automation and Mechatronics*, Bangkok, September 2006, pp. 29-30.
- [15] J. Herrera-Vega, "Mobile Robot Localization in Topological Maps Using Visual Information," Master's thesis (to be published), 2010.
- [16] R. T. Vaughan, B. P. Gerkey and A. Howard, "On Device Abstractions for Portable, Reusable Robot Code," *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 3, 2003, pp. 11-15.
- [17] L. Romero, E. F. Morales and L. E. Sucar, "An Exploration and Navigation Approach for Indoor Mobile Robots Considering Sensor's Perceptual Limitations," *Proceedings of the IEEE International Conference on Robotics and Automation*, Seoul, Korea, May 21-26, 2001, pp. 3092-3097.