Find Kick Play
An Innate Behavior for the Aibo Robot

Ioana Butoi '05

Advisors: Prof. Douglas Blank and Prof. Geoffrey Towell

Bryn Mawr College, Computer Science Department

Senior Thesis

Spring 2005

Abstract

This thesis presents an innate behavior for a robot that is supposed to find an object and take it to a goal location. The specific behavior is designed for an Aibo robot to find a ball and kick it towards a goal until it scores. The behavior is implemented using a Finite State Machine and explores several aspects of robotics such as: object detection and recognition, navigation in an unknown environment, goal detection as well as goal achievement. Part of this thesis examines the design and implementation of a Python based Aibo controller that allows the user to easily write behaviors avoiding the low-level details.

1 Introduction

The purpose of this thesis is to create an innate behavior that can lead to complex actions. This innate behavior can be used in Developmental Robotics [4] as a starting model for a robot that wants to explore the world. The goal of Developmental Robotics is to design a self-motivated robot that is able to develop, learn, and adapt in somewhat the same way as a baby. The idea of Developmental Robotics is like giving a kid a coloring book, where the innate behavior is the outline. The developmental part starts when the kid colors in the figure and continues when the kid draws other figures inspired by the original. Just like the kid whose imagination is triggered by the pictures and starts to draw other pictures, the robot's development would be triggered by the innate behavior and start to independently engage the world around it and interact with the environment in a novel way. Without the innate behavior, the developmental process cannot be triggered, thus having such a behavior is a key aspect of Developmental Robotics. To help research towards the greater goal of Developmental Robotics, this thesis focuses on the design of an innate behavior.

This paper presents an innate behavior that is designed to allow a robot to explore the world by accomplishing a task. The robot has to find a specified object and move it to a goal location. This problem has several sub-problems including object detection and recognition, navigation in an unknown environment, goal detection and localization, and goal achievement.

The specific experiment to explore this issue is to have a robot find a ball and kick it to a goal. At every step the robot must track the ball, know where the goal is, and kick the ball towards the goal. At first glance this problem might not seem very hard since it is a simple matter of kicking the ball straight towards the goal. Actually it is rather complex because the robot has to continuously reorient itself towards the goal because it cannot accurately kick in a straight line due to inconsistency in the kick, unevenness of the surface, and asymmetries of the ball.

In this project I used Sony Aibo robots [1]. I chose to use these robots because I wanted a robot that is able to move the ball with a foot instead of just pushing it like a wheeled robot. Due to the imperfect control of the robot's mechanics, the kick cannot be exactly reproduced each time, leading to inconsistency in the kick. This inconsistency factor is an important part of the experiment since it adds to the complexity of the behavior. The main tool I used in this project was Pyro [11, 2, 3], a programming environment designed to ease the development of robot behaviors. In order to use Pyro, I first had to write the Aibo controller to integrate it with Pyro. Therefore, in addition to this report, the result of my thesis is a package for controlling the Aibo robots through Pyro.

This project is comprised of two heterogeneous components: the controller and the behavior. Since the behavior is built using the controller, I will first talk about the controller and then about the actual experiments. To accomplish this project I worked with Professor Douglas Blank, who helped me write the controller and provided guidance for the innate behavior design.

2 Background Work

The background of this work falls into two general sections: first, the work that has been done on a controller for Aibo and the tools I used to write the controller; and second, the algorithms and methods used for object detection and motion tracking.

2.1 The Controller

The decision to write the controller came from the fact that we wanted to have flexibility and to focus on the behaviors without having to worry about the low-level components. To ease the development of the controller we based it on an open source controller for the Aibo Robots called Tekkotsu [16]. Using some parts of the Tekkotsu project we wrote a Python-based controller that we integrated with Pyro.

2.1.1 Sony Aibo Robot

The Sony Aibo robots [1] are of recent vintage. Although the idea for Aibo originated in the early 90's it took almost a decade for them to materialize as the first entertainment robot: the Aibo ERS110, which became available in 1999. Since then, Sony has developed four new generations, each with more capabilities and improved usability. The current model, ERS-7, released in 2003, comes with entertainment

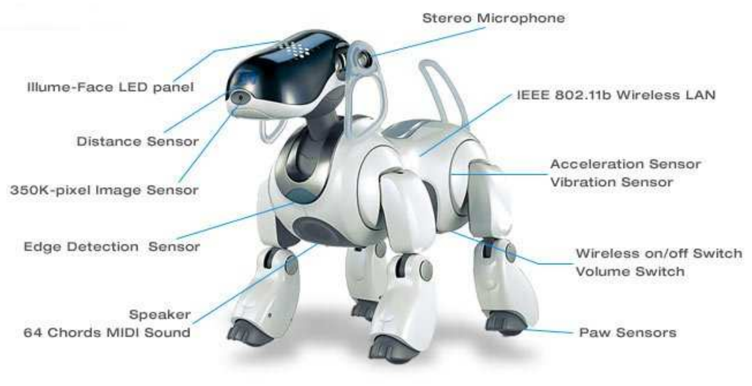


Figure 1: Picture of an ERS-7 Sony Aibo Robot. Image from the Aibo robot homepage [1].

software that allows the user to "train" the robot from a puppy state to a full grownup dog state. This software is designed purely from entertainment purposes and it cannot be altered in any way.

The ERS-7 model (see Figure 1) comes equipped with a program that makes the robot seem "alive". The dog can learn, play, appear to have emotions, and even satisfy its curiosity by exploring the environment. The robot's various sensors(a digital camera, microphones, speakers, IR distance sensors, touch sensors) allow it to perform all these tasks and even more.

Sony provides an Aibo Software Development Kit (SDK) called OPEN-R that allows people to write software for the entertainment robot. The SDK is meant to be a low-level interface that gives the user access to the robot functionality. Unfortunately, it is very cumbersome to write a behavior using OPEN-R. Even writing a move sequence, which involves specifying the beginning, intermediate, and end joint positions, requires a significant amount of work. Because we wanted to have a high-level abstraction of the robot controls we decided to implement our own controller.

2.1.2 Pyro

Pyro is a development environment that allows the exploration of topics in AI and especially in robotics. Pyro is a Python-based tool that hides all the low-level details from the user, shifting the focus to the development of behaviors. Pyro is a client-server application that controls the robots by communicating through the wireless network (see Figure 2).

The robots currently supported by Pyro are from the Pioneer [10] and Khepera [6] families as well as the newly added Sony Aibo. Pyro is also designed to support simulators and comes with a series of libraries that allow a better exploration of AI. Some of these libraries are more relevant to robotics, such as vision processing or finite state machines, while others are more general, such as artificial neural networks or genetic algorithms.

The main idea behind Pyro is to keep the user away from the low-level details and the specific design of a robot. Therefore, Pyro's main feature is its platform independence. A behavior designed for a Pioneer can work for an Aibo even though they are two very different robots. The move command is a

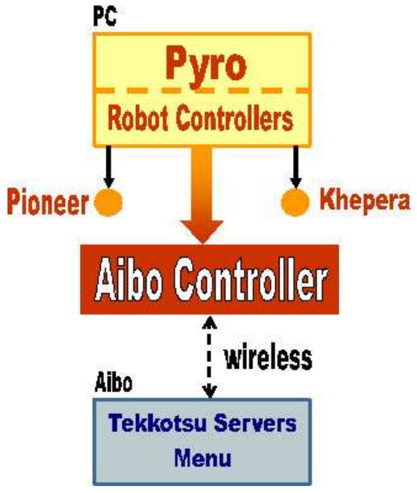


Figure 2: Integration of the Aibo Controller with Pyro.

good example to illustrate Pyro's portability. Even though the Pioneer is equipped with wheels and the Aibo with legs, the same command gets both robots to move. Hence, Pyro is a tool that allows the user to focus only on the behavior while allowing a great degree of freedom and flexibility.

2.1.3 Tekkotsu

Tekkotsu [16] is an open source project developed and maintained at Carnegie Mellon University. Tekkotsu, which in Japanese means "iron bones", is an application framework for Aibo robots. Tekkotsu is written in C/C++ and uses OPEN-R to control the robot. The project provides a first layer of abstraction that allows the user to control the Aibo robots more easily. We were interested in two aspects of this project: the servers running on the robot and Tekkotsu-Mon, an interface program that runs on the computer.

The Tekkotsu servers are designed to provide a way to control the robot's hardware. They reside in the robot's memory and act as a gateway to communicate with the robot's joints, sensors, buttons, camera, speakers and microphones. Through these servers the robot's functionality can be explored and used.

Tekkotsu-Mon is a Java-based interface that controls the Aibo robot by communicating with the servers via a wireless network. Tekkotsu-Mon is, in fact, a remote control designed for the user to explore the robot's functionality. Tekkotsu-Mon communicates with the servers through a dynamically built menu that resides on the robot. The purpose of the menu is to provide a way of turning on/off the servers. After a server is turned on, the program communicates with the server through sockets.

2.2 The Innate Behavior

One of the largest issues in this work is object recognition and tracking. Although many solutions have been provided the problem is still open for research because of its generality. Most known solutions are very specific and they depend greatly on some assumptions of the environment. Most of the algorithms presented below have been developed as a result of the robot soccer competition, RoboCup [13]. Given the nature of this work, one of its applications could be RoboCup.

2.2.1 The Robot Soccer Competition

The robot soccer competition started around 1997 and the goal is to develop a team of autonomous humanoid robots that can win against the human world champion team by the year 2050 [13]. RoboCup is divided into several leagues and one of these is dedicated to Aibo Robots. RoboCup incorporates various technologies and concepts such as multi-agent collaboration, vision and other sensor fusion,and real-time reasoning [8].

The Legged League of RoboCup is dedicated to the Sony Aibo robots. In the 2004 tournament [12] there were twenty-three teams split into four groups. Teams in each group played against each other and the top two teams moved onto the quarterfinals. The 2004 Champion is the German Team, which is comprised of researchers from several German universities: Humboldt Universitat Berlin, Universitat Bremen, Technische Universitat Darmstadt, and University of Dortmund, Germany [14]. In order to participate in the competition a team has to submit an application describing the research underlying the robot team.

In addition to the soccer competition, in 2003, a technical challenge competition was added. In 2004 there were three technical challenges: the open challenge, the variable lighting challenge, and the almost SLAM challenge. The objective of the open challenge is to encourage creativity in the Legged Division. The variable lighting challenge is intended to help the development of more robust vision modules that are less sensitive to lighting conditions. The almost SLAM challenge is intended to have robots able to localize using various landmarks and not the standard RoboCup markers.

As a result of this competition research in the field of robotics is on the rise. Vision is one of the main research areas that benefits from RoboCup, since it is a vital component of the success of a team.

2.2.2 Object Recognition

Object recognition is an important topic of Developmental Robotics. The goal is to have a robot that can recognize an object based on some general model. Humans can easily recognize a cup no matter its shape or its orientation. For a robot the same problem is very difficult, since the process of object recognition goes well beyond just image processing.

A simplified version of the problem, when general object recognition is not the desired goal, is to recognize an object based on its color. Under the assumption that the object has a color that is distinct from the rest of the environment, color-based object recognition is a viable solution [17].

Lighting is one of the biggest issues when dealing with object recognition based on color. Shades of color are very sensitive to the amount of light that falls on an object, and a small change in the shade can affect the recognition algorithm. A solution is to perform some pre-processing to the image in order to balance the light by using auto-additive adjusting [9]. The total luminance of the environment is calculated and then used to adjust the luminance of each pixel.

2.2.3 Object Tracking

Tracking a moving object falls into the greater category of motion detection. The main idea behind motion detection is to investigate the correlation between two consecutive frames. Usually the camera has to be still because otherwise the problem becomes too complicated. The main algorithms for motion detection are Edge Detection, Block Matching Algorithm, and Representative Block Model. The Edge Detection algorithm is very slow. The Block Matching Algorithm splits the image into blocks of fixed size and tries to correlate blocks from two different frames. The motion vector is computed based on how much a block has moved. This algorithm has a poor performance when the object moves within the same block. A better algorithm is Representative Block Model that uses gray block sampling to detect the object and then looks for the representative block that contains the object [7].

For an Aibo robot the motion detection problem is rather complicated since its camera is not fixed. A solution is to use camera motion compensation in order to obtain a stabilized background and then use Kalman filtering for motion tracking [5]. Another solution is to use inverse kinematics to compute the location of the ball relative to the robot. Based on the position of the body, the position of the camera, and the size of the ball, the robot can compute the location of the ball relative to its body [17, 15].

An important part of tracking a moving object is the ability to predict where it is going, in order to reduce the search space. Prediction can also be useful for determining a good strategy in a given situation. Several types of prediction can be implemented: position based, velocity based and acceleration based [9]. The position based prediction is the most useful when the tracked object follows irregular movement. The other two are more efficient in reducing the search space. Given the irregularities in the ball movement, for this experiment position based prediction was the best choice. This issue will be discussed at length in section 4.

3 The Controller

A controller is a tool used in the development of behaviors designed for robots. Its main purpose is to switch the focus from the low-level details to the high-level details in conceiving a robot experiment. A controller helps the researcher specify some robot actions in a very basic way at the software level instead of having to interact with the robot at the hardware level. Basically, a controller is an interface between the user and the robot's hardware.

This controller not only eases the development of behaviors but also provides a tool that can be used by other researchers to develop their own experiments. Given that we already had Pyro, it seemed natural to us to integrate the Aibo controller with Pyro in order to take advantage of the tools provided by Pyro.

Before we started working on the controller we assessed our options. The most obvious option was to build a controller from scratch using OPEN-R, the SDK provided by Sony. However, we did not want to code at such a low level and time was a constraint as well. Therefore we decided to base our work on Tekkotsu.

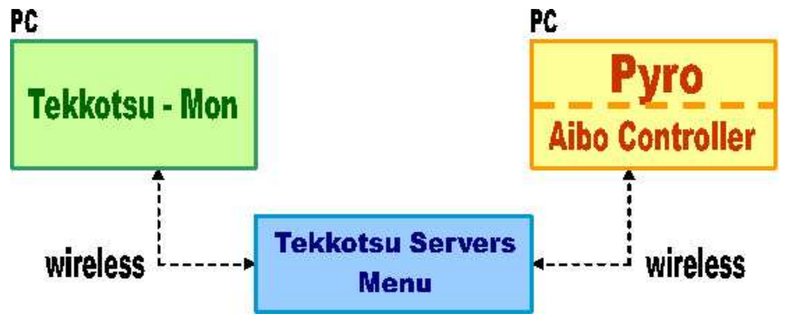


Figure 3: The Tekkotsu servers and the menu.

3.1 The Tekkotsu Servers

Tekkotsu looked like a great basis for the controller because it had almost all the required functionality and provided a good level of control of the robot. However, writing the Python controller was more complicated than just "translating" from Java to Python because the new controller had to be consistent with Pyro while interacting with the Tekkotsu architecture. Controlling the robot requires controlling the servers (see Figure 3), which, in turn, involves solving two major problems. The first problem is simply knowing whether a server is on or off. The second problem is packing and unpacking the information that is exchanged with the servers.

3.1.1 Turning On/Off the Servers

The Tekkotsu servers running on the robot can be considered to be the main component of the Tekkotsu project because they provide the gateway to control the robot. Before establishing communication with the servers, the controller has to make sure that they are turned on. This is not a trivial problem given the design of Tekkotsu.

The only way to turn a server on/off is through a dynamically built menu that resides on the robot. The menu blindly changes the state of a server without first checking its current state. This can be a serious problem because instead of turning on a server the system might turn it off and cause the failure of the program. On top of this, restarting the robot does not ensure that all the servers are off. For Tekkotsu-Mon changing the state of a server is not an issue because this application is actually a graphical interface designed to control the menu that resides on the robot.

On the other hand, these are important issues for our controller since it requires an automated way to control the state of a server. The first problem we had to resolve was the dynamic structure of the menu that controls the servers. The menu is built every time the robot is restarted and the only way to access it is through a pointer at the root. In order to change the state of a desired server, Prof. Blank wrote a function that searches the menu for the name of that server.

After we were able to turn on/off a server we had to make sure that the server was in the desired state (on or off). To overcome this difficulty we observed that whenever the state of a server changes, its name changes as well in the dynamically built menu. Whenever a server is turned on, the "#" character precedes its name. Therefore, we could determine the state of all the servers by searching for the "#" character preceding their name. Perhaps inelegant, this idea was successful in changing the state of a server as desired.

3.1.2 Packing and Unpacking Data

Having a reliable way to control and use the Tekkotsu servers allowed us to move to the next phase of the project, which involved communicating with the servers. In order to control the robot the program has to send and receive information from the servers. This is done through a communication channel established via a socket. Unfortunately the problem is more complicated than it appears since the servers can only "understand" a certain language. We had to figure out how to pack and unpack the data so that both the client and the server can communicate.

Packing and unpacking the data entails translating between the server language and the client language. The servers expect data to come in a specific format and the packing function makes sure data is formatted that way. Unpacking the data coming from the server makes it accessible to the client application. It is just the reverse action of packing.

Unpacking the joint data was a challenge because this feature was not well documented. The only guideline that ensured the correctness of the data was the timestamp that was packed together with the joint information. Since the timestamp did not follow an increasing pattern, we decided something was wrong with our unpacking algorithm. Eventually we had to contact the developers of Tekkotsu and get the pattern from them. Once we had the unpacking pattern we were able to read the joint information, including joint positions, sensor values, and button states. I wrote functions to make the data accessible through the Pyro interface. For example, *getSensor ("ir chest")* returns the value of the IR sensor placed in the robot's chest.

3.2 The Movement Module

Moving the robot involves three main components. The first component focuses on the ability of the robot to walk since it has four legs instead of a set of wheels. The second controls the robot's head movement. And finally, the third component deals with the ability of the robot to move individual joints. Prof. Blank wrote the functions needed for the first two components, while I wrote the ones needed for the individual joint movement. To comply with Pyro we ensured that all the movement ranges were between -1.0 and 1.0, with 0.0 being a central position or stop.

3.2.1 Moving the Robot Around

Tekkotsu Mon provides predefined sequences that give the robot three different walk modes. The basic walk mode, which is also the most stable and the fastest, is crawl, in which the robot walks on the knees of its front legs. A more natural walk for a real dog is what the program calls a tiger walk. The third option for a walk sequence is pace. Integrated in the motion module is also the robot's ability to turn as well as to move sideways (strafe). While move is a common function in Pyro, strafe is a new ability that takes advantage of the legged design of Aibo robots. To move the robot we only had to connect to

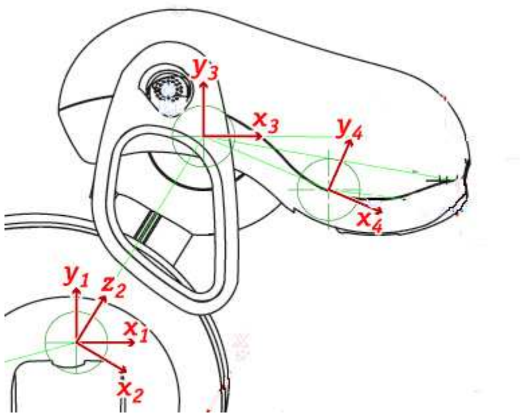


Figure 4: The movement of the Aibo's head. Image from the Tekkotsu homepage [16].

the server that controlled the robot movement and to send the necessary commands. The Pyro function sends the type of movement (translation, rotation or strafe) as well as the speed through the socket. All the move functions were written to comply with Pyro in order to preserve its portability. For example, to move a Pioneer which has wheels or to move an Aibo we can use the same Pyro function *move(translate, rotate)* .

3.2.2 Moving the Head

To move the robot's head we had to turn on the head movement server and send the move command through the socket. We treated Aibo's head as a camera device since the camera is placed in the robot's nose. The other use of moving the head around would be to get IR readings since there are two IR sensors placed in the Aibo's nose. Again, we had to make sure that moving the camera around was compatible with Pyro. We treated the head as a Pan Tilt Zoom (PTZ) camera even though the Aibo's camera has no zoom feature. Beside its ability to move the head up and down, left and right, the Aibo can also "nod", which is an extension of the vertical movement (see Figure 4). In the end, the head simulates a PTZ camera with no zoom, but with an extra degree of movement.

3.2.3 Moving a Specific Joint

Moving the robot as a whole does not take advantage of all the capabilities of an Aibo. For example, being able to move just a leg in order to kick a ball is an important feature that will prove to be very useful in the development of the behavior. The ability to move a specific joint gives the user more flexibility as well as a tool to write movement sequences. To move a specific joint, Pyro has to get all the joint positions from the Tekkotsu servers and then send them back with the modified values. This architecture is not the best approach to solve the problem as the movement of one joint should be independent from the movement of other joints. Unfortunately, since we were using the Tekkotsu servers we had to comply with this architecture. The following Pyro function bends the front knee: *setPose("front leg right knee", 0.7)*.

3.3 Other Features

3.3.1 Image Processing

The Aibo robot is able to process images on its own. However, Pyro is equipped with an image processing module. We preferred to save computational time on the robot end and perform it on the computer end. This way we took advantage of Pyro's capabilities and we made the controller consistent with Pyro's architecture. We needed to transfer the image from the robot to the computer in order to use Pyro's vision modules. Thus, Prof. Blank wrote the necessary functions to compress the image on the robot, send it over, and decompress it on the computer.

3.3.2 Playing Sounds

Tekkotsu Mon provides the ability to play standard audio files in the .WAV format. There are already a number of files residing on the robot and other files can be added. To play any of the .WAV files I wrote a function that turns the sound server on and then sends the file name through a socket. An example is *playSound("mew")*.

3.3.3 Processing Speed

The robot is set up to send information over the sockets as fast as it can. The usual speed for image transmission is about 30 fps. On the other hand Pyro is used to receiving information at a rate of at most 10fps. Also, Pyro needs information only when it is requested by a client. Since there is a lot of extra information that Pyro has to ignore, we had to slow down the rate of transmission. Prof. Blank wrote the code that allows Pyro to set the rate of transmission of the Tekkotsu servers.

3.4 Conclusions

With the controller complete, Pyro is now able to control the Aibo robot the same way it controls the Pioneers and the Kheperas. All the basic Pyro functionalities have been implemented and behaviors can now be easily written for Aibo robots. In addition, the behaviors already existing for Pioneers and Kheperas can be used on the Aibo robot. For a detailed description of the controller's API see Appendix A.

4 The Innate Behavior

The robot innate behavior I designed was to find an object and then move it to a desired location. The purpose of my experiment is to give the robot a starting point for further exploration and development. Even if this innate behavior is going to be just an initial state for the robot that it can build on, it is not a trivial task to accomplish. In designing and implementing the innate behavior, I came across various challenges that ranged from problems with the lighting, to problems with balance when the Aibo tries to kick.

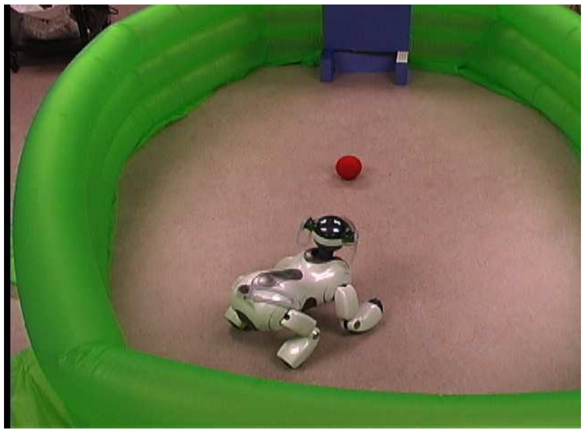


Figure 5: The robot is trying to kick the ball in the goal

4.1 The Experimental Setup

In this experiment an Aibo robot searches for a ball, approaches it, tries to kick it towards the goal, and checks to see if it scored (see Figure 5). This experimental setup assumes the following situations. The ball and the goal have distinct colors from the rest of the environment. Also, the environment has no boundaries so that the robot can rotate freely around the ball at any time.

One important aspect of the experiment is the kick. The robot cannot just push the ball, it has to kick it. The kick is an inconsistent action that makes the task harder for the robot. Given that this is an innate behavior we want to give the robot room to explore the world when this behavior is used in a Developmental Robotics setup.

4.2 Finite State Machine

The architecture of the behavior is based on a Finite State Machine (FSM). This modular approach helps keep the implementation clean in addition to breaking the behavioral design paradigm into several sub-problems. Each sub-problem can exist by itself and therefore can be of use at the developmental stage. A Finite State Machine allows an easy transition from one state to another, as well as a sharp definition of each state. This modular approach is easy to expand and change. Also, the robot can improve each of these modules individually instead of the whole behavior. A breakdown of the states can be seen in Figure 6). Integrating the Aibo controller with Pyro proved to be useful because I took advantage of the Finite State Machine built into Pyro. Each of the following sections describes a state in the FSM.

4.2.1 Search for the Ball

The first module I built for this behavior was to search for the ball. The approach I took was to find the ball based on its color. This is a common method used in robotics because it is one of the easiest ways to deal with object recognition [17]. To make sure it was going to work I assumed that the ball is the

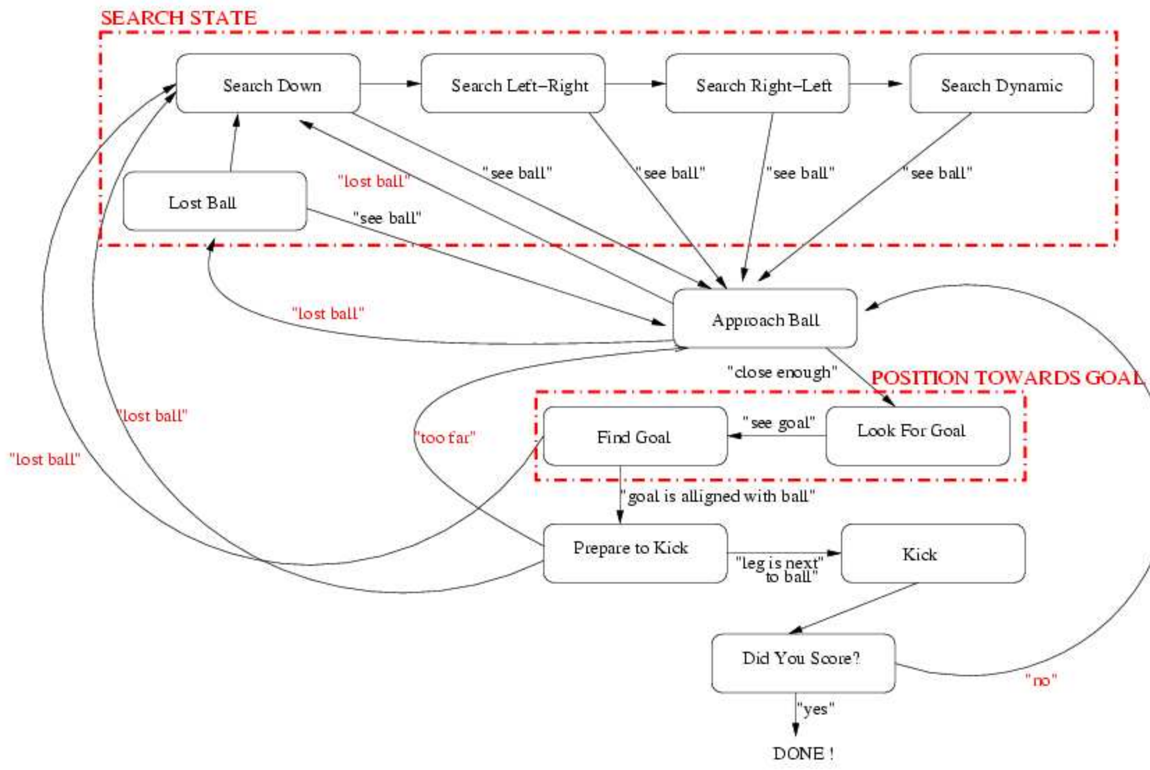
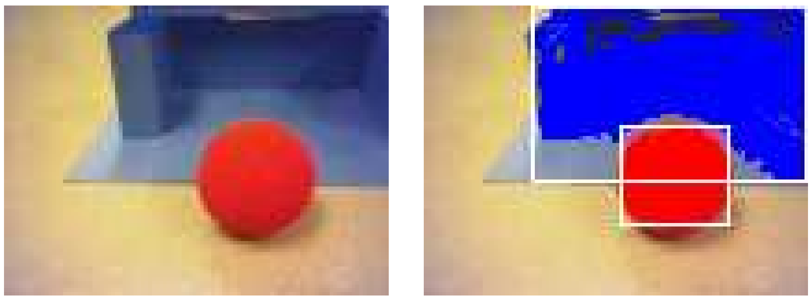


Figure 6: The states of the Finite State Machine.



(a) Before applying any filters

(b) After applying match and blobify to the ball and the goal

Figure 7: The robot's "eyes".

only pink object in the environment. Using the Pyro Vision Module, I applied a filter to the image that is sensitive to a specific color. Then I used the Pyro blobify function to find connected regions with the same color as the filter color(see Figure 7). The program returns the connected region with the biggest area that matches the filter color. This region is called a blob. The blobify function returns the area of the blob and the coordinates of its top-left and bottom-right corners. Given that the ball is the only pink object in the environment, the robot can assume it has identified the ball when it sees a pink blob.

Unfortunately finding the ball is not as easy as it sounds. In reality, finding a pink blob can be very difficult, especially if the lighting conditions change. The biggest problem is that the camera is very sensitive to lighting changes. The filter tries to match to a specific shade of color. If the lighting changes slightly, causing that shade of color to be different, then the filter can no longer recognize the color. The human eye is also sensitive to light, but the brain compensates for this sensitivity. Even though the human eye sees different shades of color under different lighting conditions, the human brain has a general color model where it can correctly place that specific shade so that the color can still be recognized. Building a color model for a robot is nearly as hard as building an object model. Therefore, an easier solution is to broaden the filter's color definition. This means giving the filter a wider range to match by increasing the threshold level. This proved to be a good solution since the ball is the only pink or even red object in the environment. However, in a different situation increasing the threshold might be problematic since more objects than intended would be matched to a particular color, even though they may not be the object of interest. A solution to this problem is auto-additive adjustment [9], which tries to balance the color. However, for the purposes of this experiment I assumed the lighting is constant throughout the environment.

To locate the ball the robot goes through several steps that are included in the more general Search Ball state as shown in Figure 6. First the robot looks down to check whether the ball is right in front of it. If the search fails, then it scans the environment from left to right and then from right to left, each time at a different angle. If the ball is still not seen, then it must be out of the robot's field of view. If that is the case, the robot starts turning in place. If the ball is still not discovered after a 360 degree turn, which is determined by keeping track of the robot's speed and the time elapsed, the Aibo will then turn around again, but with the camera at a different angle. Within at most two complete turns (720 degrees), the robot should be able to find the ball.

Besides the uninformed search states there is also a "semi-informed" search state called "Lost ball". The robot enters this state when it loses sight of the ball. The robot has an idea where the ball might have gone based on the information of the last ball-blob seen. Using the coordinates of the blob corners, the robot decides which direction it should look first to find the ball. This state is designed to reduce the search space when some information is available [9]. If the robot is unable to spot the ball, then it goes back to the general search module and starts the search process again.

4.2.2 Approach the Ball

Once the ball is located, the robot tries to approach it and get close enough to be able to kick it. When moving closer to the ball, the robot tries to center its camera on the ball so it is harder to lose. This also helps the robot to keep its "eyes" on the ball when moving.

The robot uses the crawl walking mode to move to the ball. Although this is not the natural way a dog walks, it is more stable than the tiger or pace walking modes. Not only does crawl allow the robot to move faster and avoid falling over, but also it keeps the camera more stable. Having a stable image is a very important aspect of the behavior since the robot should not lose the ball once it finds it. Therefore having a stabilized image will help prevent this issue. Camera stability is frequently an issue with legged robots while it is typically not a problem with wheeled robots when moving on a flat surface.

The robot tries to constantly adjust its head position so that the ball-blob is in the center of the camera. When the head turns past a threshold the robot stops moving forward and turns its body to try to look straight at the ball. As the robot turns its body, it also turns its head in the opposite direction to compensate for the body movement. In the end, the goal is to come close to the ball while looking straight at it.

Keeping the ball in sight is an important aspect of the behavior. Otherwise the robot has to go back to the search states. In this situation the robot would switch to the Lost Ball state. There are multiple reasons why the robot might lose track of the ball. The lighting might change so that the filter is no longer able to match the specific shade of color. If the ball moves too fast, then the robot might not be able to keep up with the ball. Another factor that can contribute to this issue is the update speed of the image. A slow update rate (at most 10fps) can affect the processing speed and slow down the ability of the robot to determine changes in the location of the ball. Unfortunately, this is a limitation of Pyro's Vision Module and we have to take it as it is.

While approaching the ball, the robot has to decide when it is close enough to stop. This is an important aspect of this state since the robot should not walk past the ball or even bump into it. Determining what is close enough was not straightforward. The approach I took in solving this problem was to have the robot stop when the ball-blob reaches a certain size. I had to determine this number experimentally by running several trials until I got the desired result. Even then, success was not guaranteed because of changes in lighting. The ball might be right in front of the robot at the desired position, yet the filter might only match half of the ball. In this case, the blob would only be half of the regular size and the robot would just continue to approach the ball.

To avoid this situation I tried to use the Aibo's IR sensors (two in the nose and one in the chest, see Figure 1). This did not help because the sensors do not seem to be sensitive enough to sense the distance to the ball. One possible explanation could be the ball itself. Given the round and shiny surface of the object, the IR rays might reflect off of it in various directions, misleading the sensors. IR sensors are not very accurate and the curved surface of the ball adds to this inaccuracy. Another problem with the IR sensors is that two of them are placed in the robot's nose. As the robot moves, the nose also moves, which gives fluctuations in the sensor readings.

Given that the robot could not base its decision on the IR sensors, the only solution was to broaden the color definition. While inelegant, this solution works and the robot consistently approaches the ball and stops at the right time.

4.2.3 Position Towards the Goal

The task of the robot is to kick the ball in the goal. Therefore the robot has to know where the goal is in order to position itself in the right direction. This positioning process involves two steps as shown in Figure 6. First the robot needs to locate the goal and second it has to rotate around the ball so that the ball is aligned with the goal.

Once the robot gets close enough to the ball it stops moving and looks around for the goal. Similar to ball recognition, the robot recognizes the goal based on its particular color. A match filter and a blobify function are applied to the image in order to obtain the goal-blob. Since Pyro allows matching and blobifying on three different color channels, there is no interference between the ball and the goal blobs. The goal is designed such that the bottom part is colored as well. This feature will be used in determining whether the robot scored. I had to deal with the same color matching issues that I encountered when I searched for the ball. Thus I applied the same solution and broadened the color definition to compensate for the lighting problems.

The robot searches the environment by moving its camera from one side to the other. If it tracks the goal-blob, then it stops the search and starts to rotate to align the ball with the goal. The robot determines the direction it needs to rotate by the position of its head when it sees the blob. For example, if the goal is to the robot's left, then the robot starts rotating to the right. If the robot is unable to detect the goal, then it starts to rotate around the ball.

The robot keeps the ball at the bottom of its image while it rotates around it trying to align the ball with the goal. When the alignment is complete, the robot stops and kicks the ball. Rotating around the goal while searching for the ball is the result of two actions. First the robot performs a lateral movement using strafe. Second, when the camera is no longer centered on the ball, the robot stops and rotates until the ball is re-centered in the image. These two actions combine, resulting in circular motion of the robot around the ball. Another characteristic of this motion is that the robot moves slowly away from the ball. This proves to be useful when the robot is too close to the ball and cannot see the goal because of the camera angle. For example, the goal might be right in front of the robot, but the robot cannot raise its head high enough to notice that since it needs to keep track of the ball. In this situation a rotation around the ball will allow the robot to slowly move away from the ball and broaden its field of view.

4.2.4 Prepare to Kick

Before the robot is able to actually kick the ball, it has to get into the right position. This involves getting one of its front legs close enough to the ball. One solution is to use inverse kinematics [17, 15]. Through this process, the robot can figure out the angle that it has to move its joints so that the leg is right next to the ball. I decided not to use this approach because it is very unnatural. When I play soccer I prepare to kick the ball through a visual process instead of thinking about the exact positioning of my leg. The visual feedback in this action is the key to a successful kick. I applied this logic to the robot, hence, it uses visual feedback for positioning prior to a kick.

Constraints on the mobility of the Aibo's head made it difficult for the robot to see that its leg is close to the ball. One idea I had was to place colored markers on its feet and use a match filter and the blobify function to recognize them. Although this seems like a natural approach to solve the problem I decided to go with another idea. I wanted to avoid the issues that would have come up in color recognition. Also, this approach would have restricted the robot's field of view while preparing to kick the ball, which might have increased the probability of losing the ball. Instead I decided to utilize the angle at which the robot looks at the ball for proper leg placement.

The robot keeps a constant eye on the ball while it tries to get into position to kick. The robot rotates around the ball while turning its head and keeping the ball centered in the image. When the camera angle is within a desired range and the blob size is appropriate for the given ball size, then the robot is in position and ready to kick. The robot decides which leg it is going to use based on the camera angle. The positioning of the robot depends heavily on the performance of the match filter. Therefore, instead of using the area of the blob as a target I used the size of the bordering square. Sometimes the bordering square of the blob is be bigger than the actual matched area. This solution proved to give a consistent result.

4.2.5 Kick the Ball

Once the robot is close enough to the ball and properly aligned with the goal it kicks the ball. Although this may seem like a very easy action it actually obscures two main issues. The first is maintaining balance and the second is the kick itself. To kick a ball the robot has to stand on three legs and maintain its balance while the fourth leg swings and actually kicks the ball.

To perform any of these steps the system has to control the robot's joints. The leg of an Aibo has three joints that can be controlled independently: rotator, elevator, and knee (see Figure 8). The rotator and elevator both act as the shoulder. The rotator enables the robot to swing the leg front and back. The elevator permits the robot to raise the leg away from the body. The knee, as expected, allows the robot to bend the leg.

Maintaining balance on three legs while allowing the fourth leg to swing freely is a rather complicated problem. For a dog that stands on three legs the body position is constantly adjusted to maintain

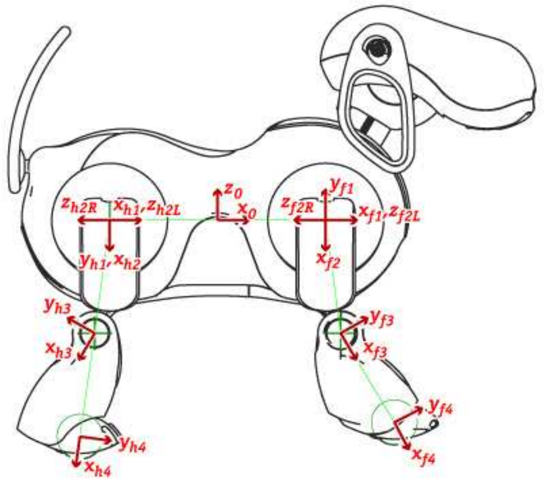


Figure 8: The Aibo joints. Image from the Aibo robot homepage [1].

balance. This is not a viable approach for a robot since it does not have the balancing system of a dog to performs the necessary small adjustments. Therefore, for a robot the balancing problem turns into finding a special joint configuration that best maintains balance. Once the robot is able to stand on three legs it has to maintain balance while it swings the other leg. This adds another degree of difficulty to the problem. The solution I designed uses the back leg on the same side as the kick leg to maintain balance. For example, if the robot kicks with its left leg, then it maintains balance using its left back leg. I ran many trials to determine the proper position for the back leg.

After the robot is balanced on three legs it is ready to kick the ball. To swing the leg I designed a sequence that uses the Pyro *setJoint* function, which allows independent control of each joint. To kick the robot uses the rotator and the knee joints so that it swings and extends its leg at the same time. After the kick the robot moves its front leg back to a position suitable for crawl. Even though the kick is fast, sometimes the robot becomes unbalanced and falls forward. To prevent this, the head moves into a central position to help with the balance. Unfortunately this makes the robot lose track of the ball when it kicks. However, this is a necessary step to help the robot maintain its balance.

Most of the time the kick is consistent; however there are some situations when the robot cannot get into the right kicking position. Mainly this has to do with the update speed of the joints. The function that controls each individual joint needs information from all the other joints before it can move that joint. (See Section 3.2.3)

4.2.6 Completing the Task

Immediately after the robot kicks the ball it checks to see if the task has been accomplished. In the particular case of this experiment, completing the task means scoring a goal marked as the "Did you score" state in Figure 6. The robot determines if it scored by looking at the ball and its surroundings. If the ball is surrounded by the color of the goal, then it means the robot scored. The robot knows it scored when the position of the goal-blob is lower than the position of the ball-blob. To determine this situation the robot lowers its camera until the ball is in the upper portion of the image. This allows the robot to see also part of the floor and determine if the ball is in the goal or not.

As mentioned earlier, the robot does not keep its eyes on the ball when it kicks since it uses the head to maintain balance. Therefore, the robot has to perform a mini-search for the ball. Once it finds the ball it checks to see if it has scored. If the robot determines that it has scored, then it wags its tail and barks three times.

4.3 Another Experiment

Based on this behavior I created another one in which two robots play with each other. The design of the innate behavior based on a FSM allowed for an easy expansion. To create this new behavior, the only step needed was to add a new state to the FSM that addressed the specificity of the situation (see "Wait for ball" in Figure 9). In this experiment the field is divided into two sections that are colored differently. Each robot sits on its own color and tries to kick the ball onto the other color. Each time a robot is successful it wags its tail and barks. After it scores, the robot enters a wait state that prevents it from moving until the ball moves back onto its side. If the ball is on the opponent's side, the robot keeps its eyes on it by moving its head around. Once the ball is kicked onto its half of the field, the robot is allowed to move and go through the states from the initial FSM. This experiment illustrates how this innate behavior can be used to build other behaviors and create more complex situations.

4.4 Analysis

I used two different balls to test the behavior. One is a foam ball that does not roll very much and that goes in a straight line. The other one is a hollow plastic ball that has some asymmetries in its construction, making it unlikely to roll in a straight line. I used the first ball to determine how consistent the robot is in finding the ball. I used the second ball to see if the robot can still accomplish the task even though the ball has an unpredictable trajectory. In the latter case the robot is still able to kick the ball in the goal, but it takes more time to accomplish the task. Using a slightly asymmetric ball ensures that the behavior is robust enough to handle various cases. For example, while we have not tested this, I am confident that the behavior described in this thesis would have no problems in handling a non-leveled playing field.

5 Future Work

An immediate step that follows from this behavior design is to have a robot that is able to learn how to kick. Starting from this innate behavior the robot should be able to improve and become more consistent at kicking. It would also be interesting to observe if the robot can learn how to score using fewer kicks. Several experiments that explore learning and adaptation can be based on this innate behavior. This would be the next step towards the greater goal of Developmental Robotics.

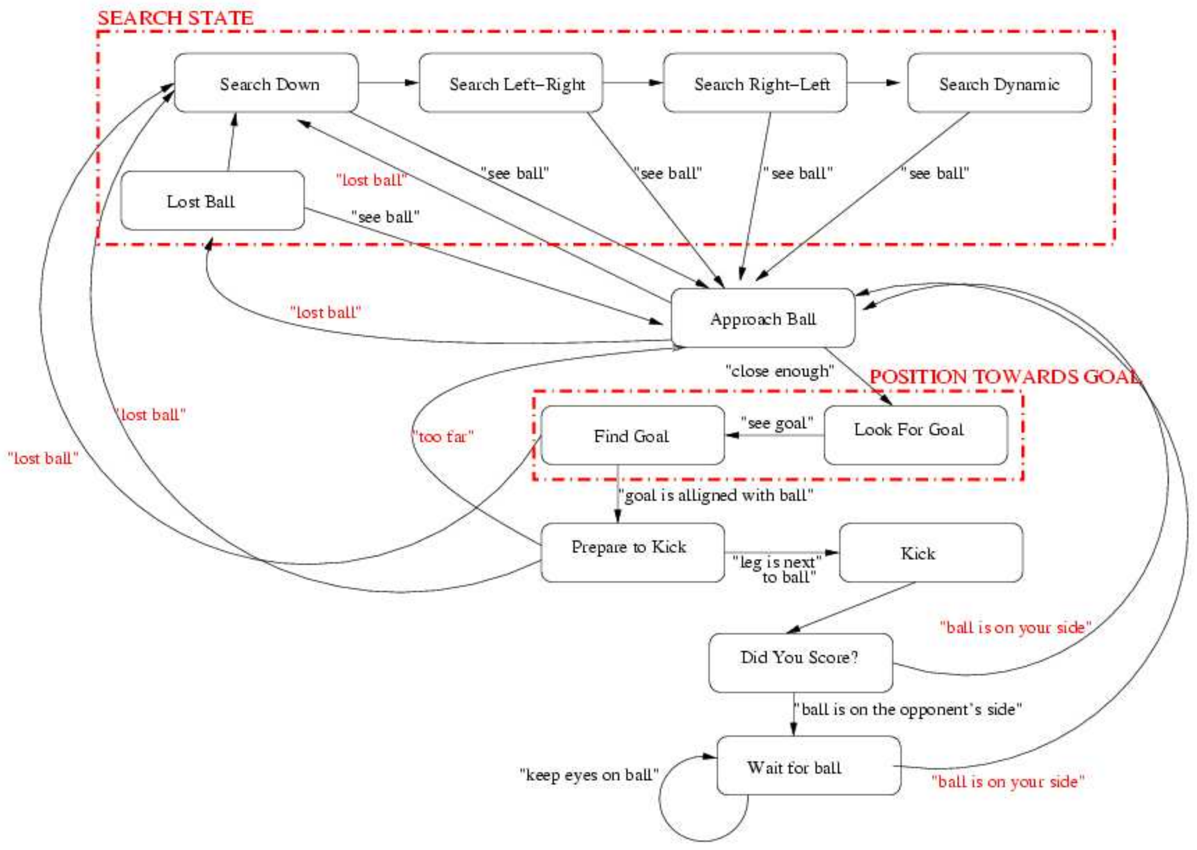


Figure 9: The FSM of the “Robot vs Robot” behavior.

One feature that would be helpful is obstacle avoidance. This can be done using the IR sensors that the robot comes equipped with. Also a stall detection feature would be useful to determine if the robot is no longer able to move because of obstructions. These features would make it possible to place the robot in a closed environment where it can learn to avoid barriers. Obstacle avoidance would also allow interaction between several robots sharing the same space.

Object recognition is an issue that would need improvement. The goal is to recognize an object based on its shape and color instead of just relying on color. As seen in this thesis even color recognition is a problematic approach. It would be beneficial to enhance the color recognition algorithm so that lighting is no longer an issue. The matching algorithm can be made more robust to allow the robot to distinguish between several objects of similar color in the same environment. The goal would be to have a robot that can recognize an object based on both its shape and color, but until then it is a very long road.

6 Conclusions

The innate behavior I designed is able to complete the task I proposed. The robot is successful in kicking the ball to the goal. As a result of the many factors that can influence each trial, experiments with robots do not always work as expected. For example, lighting problems were a major issue. Unfortunately, small changes in lighting conditions can trigger big changes in the behavior. The approach taken in this work to solve problems resulting from lighting changes was to broaden the color definition. Still, there are situations where the match filter cannot detect the ball or the goal. However, the broadened color definition gives a wide range of lighting under which the robot is consistent in kicking the ball in the goal.

The modular design of this innate behavior allows the use of certain parts of the behavior in other behaviors. This is advantageous because the states can be easily modified and enhanced without changing the behavior itself. Finite State Machines are ideal for this type of situation where the behavior is very modular.

The idea of integrating an Aibo controller with Pyro proved successful. We were able to use all of the controller's functionality. There are some parts that can be smoother and easier to use, but, I was still able to perform all the actions I wanted. One area of improvement would be to have a method that helps to design sequences, making things like designing a kick easier.

This innate behavior is ready to be used in a developmental robotics setup. The modularity of the behavior makes it easy to change and build new behaviors based on this one. The generality of the find and fetch problem gives the robot a lot of freedom in the exploration of the environment. Ideally the robot would start looking for other objects and new goals as it explores the environment. This innate behavior is meant to give the robot an idea of what it can do. From there on, the robot can design and accomplish its own tasks.

References

[1] Sony Aibo Robot. http://www.sony.net/Products/aibo/. Homepage of the Sony Aibo robot.

[2] D. Blank, D. Kumar, L. Meeden, and H. Yanco. Pyro: A Python-based Versatile Pragramming Environment for Teaching Robotics, 2004.

[3] D. Blank, L. Meeden, and D. Kumar. Python robotics: An Environment for Exploring Robotics Beyond LEGOs. *SIGCSE 2003*, 2002.

[4] D.S. Blank and Meeden L.A. Developmental Robotics. In *American Association of Artificial Intelligence Technical Report, Spring Symposium Series. AAAI Press.*, 2005.

[5] J. Ruiz del Sola and P. Vallejos. Motion Detection and Tracking for an AIBO Robot using Camera Motion Compensation and Kalman Filtering.

[6] Khepera Robots. http://www.k-team.com/robots/khepera/. Homepage of the Khepera robot.

[7] W. Kim, C. Hwang, and J. Lee. Efficient Tracking of a Moving Object using Optimal Representative Blocks. *International Journal of Control, Automation, and Systems*, 1(4), dec 2003.

[8] H. Kitan, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. RoboCup: The Robot World Cup Initiative. In *IJCAI-95 Workshop on Entertainment and AI Alife, Montreal*, 1995.

[9] B. Li, E. Smith, H. Hu, and L. Spacek. A Real-time Visual Tracking System in the Robot Soccer Domain. In *EUREL Robotics 2000, Salford, England*, 2000.

[10] Pioneer Robots. http://www.mobilerobots.com/. Homepage of the Pioneer robot.

[11] Pyro Robotics. http://www.pyrorobotics.org. Homepage of the Pyro project.

[12] RoboCup, The Legged Division. http://www.tzi.de/4legged/bin/view/Website/Teams2004. Teams that participated in the 2004 competition.

[13] RoboCup. http://www.robocup.org. Homepage of the RoboCup.

[14] Th. Rofer, H.-D. Burkhard, U. Duffert, J. Hoffmann, D. Gohring, M. Jungel, M. Lotzch, O.v. Stryk, R. Brunn, M. Kallnik, M. Kunz, S. Petters, M. Risler, M. Stelzer, I. Dahm, M. Wachter, K. Engel, A. Osterhues, C. Schumann, and J. Ziegler. German Tream RoboCup2004, Technical Report, 2004.

[15] R. Sargent, B. Bailey, C. Witty, and A. Wright. Dynamic Object Capture using Fast Vision Tracking. *AI Magazine*, 18(1), 1997.

[16] Tekkotsu Project. http://www.tekkotsu.org. Homepage of the Tekkotsu project.

[17] M. Veloso, S. Lenser, D. Vail, M. Roth, A. Stroupe, and S. Chernova. CMPack-2: CMU's Legged Robot Soccer Team, 2002.

Appendix A

The Aibo Controller API documents only the functions that are specific to the Aibo robot. The other Pyro functions are not presented here, since they can be found in the Pyro documentation.

Head Movement

The head has three degrees of freedom: pan, tilt, and roll. All the values range between -1 and 1.

robot.ptz.center()

This function re-centers the head (sets all the values to 0).

```
robot.ptz.center()
```

robot.ptz.setPose(pan, tilt,[roll])

This function can be used to set the head into a specific position. The roll argument is optional.

```
robot.ptz.setPose(amtPan, amtTilt)
robot.ptz.setPose(amtPan, amtTilt, amtRoll)
```

robot.ptz.pan(amount)

The pan function controls the horizontal movement of the head. The range of motion is between -1 and 1, where -1 is right and 1 is left, while 0 is the central position.

```
robot.ptz.pan(0.3)
```

robot.ptz.tilt(amount)

The tilt function controls the vertical movement of the head. The range of motion if between 0 and -1, where 0 is the central position and -1 is down.

```
robot.ptz.tilt(-1)
```

robot.ptz.roll(amount)

The roll function is an extension of the vertical movement that allows the robot to raise its head. The range of motion is between 0 and 1.

```
robot.ptz.roll(1)
```

Accessors

There are a series of functions that allow the user to obtain information about the robot's joints and sensors or to play a file.

robot.getJoint(jointName)

This function return the position of the joint together with its duty-cycle. The position of the joint is between -1 and 1 to be consistent with Pyro. Zero is usually a central location. Any of the following joints can be accessed:

leg “front/back left/right leg rotator/elevator/knee”

head “head tilt/pan/roll/nod”

tail “tail tit/pan”

mouth “mouth”

```
robot.getJoint(‘‘mouth’’)
robot.getJoint(‘‘left leg front rotator’’)
robot.getJoint(‘‘head tilt’’)
```

robot.getButton(buttonName)

The state of a button can be accessed using this function. The paws are on/off buttons while the rest are pressure buttons. The robot has the following buttons:

chin “chin”

head “head”

body “body front/middle/rear”

wireless “wireless”

paw “paw front/back left/right”

```
robot.getButton(‘‘chin’’)
robot.getButton(‘‘body front’’)
robot.getButton(‘‘paw front left’’)
```

robot.getSensor(sensorName)

The robot has 11 sensors whose values can be accessed using this function.

- The ir sensors: “ir near/far/chest”

  ir near Ranges from 50 - 500mm

  ir far Ranges from 200 - 1500mm

  ir chest Ranges from 100 - 900mm

- The acceleration sensors: “accel front-back/right-left/up-down”

  front-back Positive if the robot is inclined forward and negative if it is sitting.

  right-left Positive if the robot is lying on its right side and negative if the robot is lying on its left side.

  up-down Negative if the robot is standing up, positive otherwise.

- The power sensors: “power remaining/thermo/capacity/voltage/current”

  remaining The remaining power measured in percentage 0-1.

  thermo Measured in degrees Celsius.

  capacity Measured in milli-amp hours.

  voltage Measured in volts.

  current Measured in milli-amp; negative values.

```
robot.getSensor(‘‘ir near’’)
robot.getSensor(‘‘accel right-left’’)
robot.getSensor(‘‘power remaining’’)
```

robot.playSound(fileName)

This function plays a .WAV file that sits on the robot. The following files can be played: 3BARKS.WAV, 3YIPS.WAV, BARKHIGH.WAV, BARKLOW.WAV, BARKMED.WAV, BARKREAL.WAV, CAMERA.WAV, CATCRY.WAV, CATYOWL.WAV, CRASH.WAV, CUTEY.WAV, DONKEY.WAV, FART.WAV, GLASS.WAV, GROWL.WAV, GROWL2.WAV, GRRR.WAV, HOWL.WAV, MEW.WAV, PING.WAV, ROAR.WAV, SKID.WAV, SNIFF.WAV, TICK.WAV, TOC.WAV, WHIIP.WAV, WHIMPER.WAV, WHOOP.WAV, YAP.WAV, YIPPER.WAV

```
robot.playSound(‘‘3barks’’)
```

Body Movement

The robot can move around or it can move only a joint. Once a move command is issued, except for moving a joint, the robot moves until it is told to stop. The ranges of the moves are between -1 and 1.

robot.translate(amount)

The robot translates with the given speed. The range is between -1 and 1, where 1 is forward, 0 is stop, and -1 in backward.

```
robot.translate(0.9)
```

robot.rotate(amount)

The robot rotate with the given speed. The range is between -1 and 1, where -1 is right, 0 is stop, and 1 is left.

```
robot.rotate(-0.6)
```

robot.strafe(amount)

The robot performs a lateral movement. The range is between -1 and 1, where -1 is right, 0 is stop, and 1 is left.

```
robot.strafe(0.4)
```

robot.move(translate, rotate)

This function combines the translate and rotate functions.

```
robot.move(0.5,0.3)
```

robot.setWalk(type)

The robot can be in three different walk modes. The default walk mode is crawl. The other two walk modes are tiger and pace.

```
robot.setWalk(``tiger'')
```

robot.setPose(joint, amount)

This function sets a joint to a specific position. The following joints can be controlled individually:

mouth 0 is closed and -1 is open.

tail The tail has two degrees of freedom: pan and tilt. They can be controlled individually "tail pan/tilt" or together "tail".

leg Each leg has three joints that can be controller individually. The rotator and elevator act as the robot's shoulder.

- **rotator** Moves the leg front (1) and back (-1). Zero is straight down.
- **elevator** Moves the leg away or towards the body. 1 is a full extension while -1 is close to the body and 0 is straight down.
- **knee** The knee bends the leg. 1 means the leg is bent, -1 the leg is extended and at 0 the leg is straight.

```
robot.setPose(``leg front left knee'', 0.3)
robot.setPose(``right back leg'', amtRotator, amtElevator, amtKnee)
robot.setPose(``tail'', amtPan, amtTilt)
robot.setPose(``tail pan'', 0.3)
robot.setPose(``mouth'', -1)
```