

Soccer Robots

Implementing soccer behaviors in the Webots simulator for NAO robots

Kristjan Arumae

Sarah Myhre

Chris Cassian Olschewski

Section 1: Overview	1
1.1 Team Introduction	1
1.2 Motivation	2
1.3 Project Origin	2
1.4 Goals and Objectives	4
1.5 Project Evaluation	5
1.6 Technical Approach	6
1.7 Hardware	8
1.8 Research	10
1.8.1 CMUnited-98	11
1.8.2 ARAIBO-07	13
1.8.3 Machine Learning for Arm Leg Coordination	15
Section 2: Development Environment	17
2.1 Team Organization	17
2.1.1 Google Wiki	17
2.1.2 Asana with Instagantt	20
2.1.3 Github	24
2.1.4 Evaluation	26
2.2 Simulation Software	28
2.2.1 Choreographe	28
2.2.2 Webots for NAO	29
2.2.3 Evaluation	31
2.4 Libraries	31
2.4.1 Naoqi	32
2.4.2 OpenCV	34
2.4.3 Evaluation	35
2.3 IDEs	37
2.3.1 Eclipse	37
2.3.2 Sublime	39
2.3.3 Evaluation	40
Section 3: Build	42
3.1 Webots Field	42
3.2 Framework	43

3.3 Vision	47
3.4 Movement	51
3.5 Logic Implementation	60
Section 4: Behavior Creation	63
4.1 Chase	63
4.2 Goalie	66
4.3 Defender	71
4.4 Midfielder	71
4.5 Striker	73
Section 5: Design Challenges and Solutions	76
Section 6: Online Documentation	80
6.1 Simulation Setup	80
6.2 Webots for NAO Environment	81
6.3 First Code	83
6.4 Useful Methods	84
6.5 Soccer Behaviors	84
6.6 Team Templates	85
6.7 Videos	85
6.8 Evaluation	86
Section 7: Administrative Content	88
6.1 Finance	
6.2 Consultants	
6.2.1 Dr. Gita Sukthankar	
6.2.2 Astrid Jackson	
6.2.3 Dr. Mark Heinrich	
Section 8: Conclusion	
Appendix A	
A.1 Document Citations	
A.2 References	

Section 1: Overview

Robotics is an ever-increasing division in Engineering and Computer Science. As we begin the 21st Century, technology is growing by leaps and bounds, and such exponential rates are often overwhelming. As students in the field of Computer Science, we realize that in order to keep up with the times, constant learning and involvement is necessary. Learning should therefore never be defined to a school's curriculum or timeframe. Rather, the curriculum should drive individuals to explore outside of it. Unique projects and challenging new ideas are what Senior Design classes stand for. Thus Group 3 has chosen Humanoid Soccer Robots as our project, with code to govern robots individually in a group setting as an end goal. This project utilized software and hardware not yet used by The University of Central Florida, and thus the scope of this project coincides with the requirements of the class.

This project has individual, group and University merits. Individually each person learns a great deal of non-curriculum material. As a group we acquire teamwork skills for Software Development, and the University receives a vastly open ended project for many others to get involved with. Group 3 hopes to lay the groundwork with this project and spark interests in general Robotics. This document will illustrate all of the components of our Humanoid Soccer Robots. Hardware, software, and design ideas are explained in great detail as well as our goals and predicted results.

1.1 Team Introduction

As part of the first Computer Science Senior Design class at the University of Central Florida, a team of like-minded students joined together to create a project tentatively referred to as "Soccer Robots". Under the guidance of Dr. Mark Heinrich and over the course of twenty six weeks, our team will work with a simulator to program viable Artificial Intelligence behaviors for robots playing soccer, applying professional project management strategies.

The team consists of Kristjan Arumae, Sarah Myhre, and Chris Cassian Olschewski. Individually speaking, we had previous experiences or desires to learn about robotics, Artificial Intelligence design, and dynamic problem solving; the

success of this project will come from these strengths and interests that we bring to the table.

1.2 Motivation

As a team we have many shared motivations for choosing this project. Above all, this project allows for contact with information outside of the UCF-CS curriculum. Most, if not all, of the software and tools we use are new to the group members. Having to reach out of that comfort zone is a leap towards crafting future Computer Science skills, which is arguably a large reason for the existence of the class. These learned skills will set apart our group members from our competition when searching for a job or furthering our academic career. More specifically, the group learns about Artificial Intelligence design and implementation. This topic is only covered as an elective in most Computer Science programs and often not explored.

The motivation to select this project has other unforeseen consequences since the project has such potential for growth after we are finished with it. Beyond using this for Dr. Gita Sukthankars robotics class, it may be the base for a graduate level competition team. The next phase of the project can also be completed by a future Senior Design team. Even our own team members may use it as a research topic for post-graduate studies.

It should be noted that the open ended possibilities are intriguing and may spark interest for other students to study Artificial Intelligence or one of its branches. Thus it would bring general attention to the area of study, benefiting both the party entering the study and the University.

1.3 Project Origin

Our efforts are aligned with those of the official RoboCup Federation, “an international organization, registered in Switzerland, to organize international effort to promote science and technology using soccer games by robots and software agents.”[1]

The specific goal set as a vehicle for their efforts is to have a team of humanoid robots capable of defeating the World Cup champions of the human FIFA league by

2050.[2] As algorithms evolve and tackle new problems, many involved in RoboCup see this as a replacement of the computer chess challenge, which reached its climax with the defeat of Garry Kasparov by IBM's Deep Blue in 1996.

There are different leagues in which participants can test their mettle. The smallest robots compete in the Small Size league, also known as F180 league because 180mm is the diameter each robot must fit into. The field is 6.05 m long and 4.05 m wide. In this league the focus is solving the problem of "intelligent multi-robot/agent cooperation and control in a highly dynamic environment with a hybrid centralized/distributed system." [3] The latter part refers to the camera that is set up 4 m above the playing field to which all robots have access in conjunction with wireless communication from off-field computers that control all the robots.

In the Middle Size league robots with a maximum diameter of 0.5 m play with a regular soccer ball on a regular, reduced-size soccer field with up to 6 players on the team. Communication between them is done wirelessly over a network. [4]

The Humanoid league is designing their robots to be human-like in most every respect including body plan and senses. It differs from other leagues where chest cameras and unrealistic leg stretches are common. This league contains three sub-leagues: Teen Size, Kid Size, and Adult Size. [5]

For the Standard Platform League the success in competitions lies in the teams' software design. A standard robot manufactured by the same company is used by all teams. This is the Nao robot by Aldebaran Robotics, which replaced Sony's four-legged AIBO robot dog in 2008. Requiring a uniform hardware design is intended to spur the improvement of the code the robot soccer community has at their disposal.

Analogous to the SPL there exists the Simulation league. The focus remains as before, however the challenges of operating the robot hardware are eliminated. The Nao robots are running in a simulation with code that can also be used on the actual robots. On their website Aldebaran makes available the Webots simulator for their NAO robots. Our team will be programming the controllers to run robot soccer agents in this simulator.

In general for robot soccer, most rules from actual soccer stay intact with penalties for pushing, handling the ball with arms or hands, and scoring goals directly from kick-off. The extra constraint put on robotic soccer are penalties for walking off the

field, being immobile for more than ten seconds, and using non-human locomotion such as crawling.

1.4 Goals and Objectives

The scope of work involved in programming a successful soccer robots team has shown to exceed a development phase of two semesters. In fact, the RoboCup 2011: Robot Soccer World Cup XV champions, Virginia Tech, state that they developed the team 'over the past few years'. Thus, we do not expect to build a competitive robot team in the scope of this project.

Our focus, instead, is to establish a working robot team which successfully plays 'soccer' including, but not limited to, following functionality: locate the ball, line up with the ball, move to the ball, change walking speed as approaching objects, align and aim for kick, and kick the ball. We are also considering implementing defensive functionality such a dive move for the goalie to keep the ball out of the goal. Lining up with the ball could serve as a defensive move also should the opponent try to kick the ball in the direction of our player.

Through building this team, we hope to obtain thorough knowledge of the software involved with programming the robots so that we may write detailed documentation for Dr. Gita Sukthankars robotics students to use as they become familiar with the software. Working with the software, namely Eclipse with PyDev, Naoqi, and Webots, gives us hands on experience that allows us to write accurate and descriptive documentation for the robotics course.

We expect that the resulting product of this project is easy to follow documentation (and potentially curriculum) that will be implemented in Dr. Sukthankars Introduction to Robotics course as an option for their final project. The documentation will include:

- A wiki page for students to reference during set-up and development
- Instructions on how to properly setup the software for a successful development and simulation environments
- A list of problems we encounter at any time during set-up, development, and/or testing and solutions to those problems
- A list of frequently asked questions that will start with questions we had along the way, with answers we found for those questions; this is expected to expand as teams who work on this project contribute their questions and potentially the solutions they found

- A set of class instructions or curriculum intended as a guideline for the students' code development progress
- Scripts and demos that add clarification where needed

With this, we hope to begin a new tradition at the University of Central Florida of developing soccer robot teams that could eventually be used in competition after many modifications made by the generations of robotics students and possibly even future senior design teams. In any case, we expect that the teams who work on soccer robot development at UCF in the future will compete with teams we develop as well as teams that other students build after us.

1.5 Project Evolution

Throughout the year the project evolved and refined its approach in several ways. These changes aided in streamlining development, removing redundancies, and giving the project a more uniform, professional quality.

The initial approach was to utilize the versatility of Webots and its ability to run on several operating systems. The project was being developed on Windows as well as OSX with some minimal development on Linux. With testing of initial code and simple behaviors it became abundantly clear that one operating system must be used amongst all group members. The trackers would often not work on the Windows distribution of Webots and other smaller bugs were also present. The only feasible fix for this was to use a flavor of Linux. The group decided to use the newest stable release of Ubuntu, 12.04 Precise Pangolin. Thereafter all setup guides were written with Ubuntu in mind. This is also a smarter approach for future developers, since Ubuntu is free and can be booted from any machine. Meaning it is easier to conform to using Linux than buying a new machine or OS.

Choosing a development language to use was not as difficult of a choice as the OS selection. NAOqi is available in over 10 different languages, one of which is Java. Java was the first choice the group had since the UCF curriculum includes several classes teaching Java, or implementing projects in Java. Upon investigation into the API provided by Aldebaran, only two languages receive the maximum amount of support. Those languages are C++ and Python. Java and the rest of the languages are rather neglected. The development from there on out was all done in Python.

Choreographe was an excellent tool for modeling behaviors. However, behaviors created via Choreographe were templated and not easily manipulated. The

software is too pedestrian for use in a project of this scale. All of the code was instead developed with access to the NAOqi API. These same methods are used in the Choreographe modeling-since source is created when modeling behaviors. There was therefore not much use for this software.

The last big step was to fix the vision problems that the trackers gave the robots. Only two trackers are available using the given modules; the red ball tracker, and the face tracker. The face tracker was not at all useful for the project since the robots would never come into contact with humans, and the red ball tracker had several issues of its own. When tracking the ball it would track via color and shape, thus if another round object entered the frame the tracker would also track that. There was often a conflict of interest when another player came too close during ball tracking. The figure below shows a close up of the robots chest where the tracker would latch onto as the ball because of the shape created by the contrast.

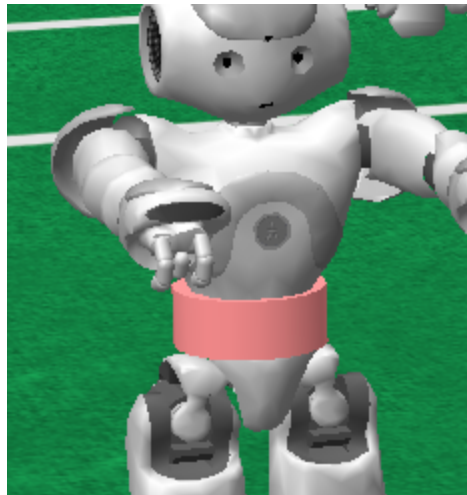


Figure 1.5.1: Robot chest shape

Another option would have been using sonar which was a capability of the NAO robots. The drawback of sonar would be not knowing what object the robot is detecting, whether it be a goal or another player. Also the spacing between the objects detected needs to be wide, thus there would be no differentiation between two enemy players standing close together or just one, by itself.

Since that OpenCV has been utilized as a source of vision management for the project. This is a very robust tool that allowed for the development of several trackers. Using OpenCV there was a tracker built for each of the objects the robot needed, the goal, ball, and other players-of either team.

1.6 Technical Approach

To develop a team of soccer robots we have many tools at our disposal. Additionally there is already a large amount of open source code for the Nao robot available, of which much pertains to the Simulation League in RoboCup and its agents ability to play soccer.

The question is how to approach the project to get the most out of everyone's efforts, and what specifically is really involved with programming and improving a soccer simulation from a technical aspect.

First and foremost there is the simulator itself, Webots. One of its purposes, aside from displaying the simulated world, is to control the robots placed within it. Through this we have access to functions like BallTracker and MoveToPoint, that can be called by the agents themselves.

Next there is the robot agent itself which is being programmed directly in Python. For its logic it uses a state machine, which makes decisions mostly relating to the agent's movement.

Tying together the behaviour of all the soccer playing robot agents, some of which will exhibit unique behaviors i.e. the goalie, is a governing program. This program will make decisions based on the game state and its perception of the world.

We will be involved with all three of these aspects to creating a simulated robot soccer team. Having already noticed issues with the BallTracker we are considering rewriting that function. Most of our time however will be spent on programming the agent behaviour and the governing program.

A lot of the programming done for the project will be for movements of the robots. Here we have the ability to use the powerful Choregraphe tool, which lets us model animations and then export the source code.

As seen in Figure 1.6.1 showing the RoboCup 2011 winning team's code functionality, there are even more detailed adjustments one can make to their team. Most of these we will have to rely on the default implementation, e.g. with the vision system, because thorough consideration lead us to realize the scope of that is beyond that of a 2 semester class.

RoboCup 2011 Humanoid League Winners

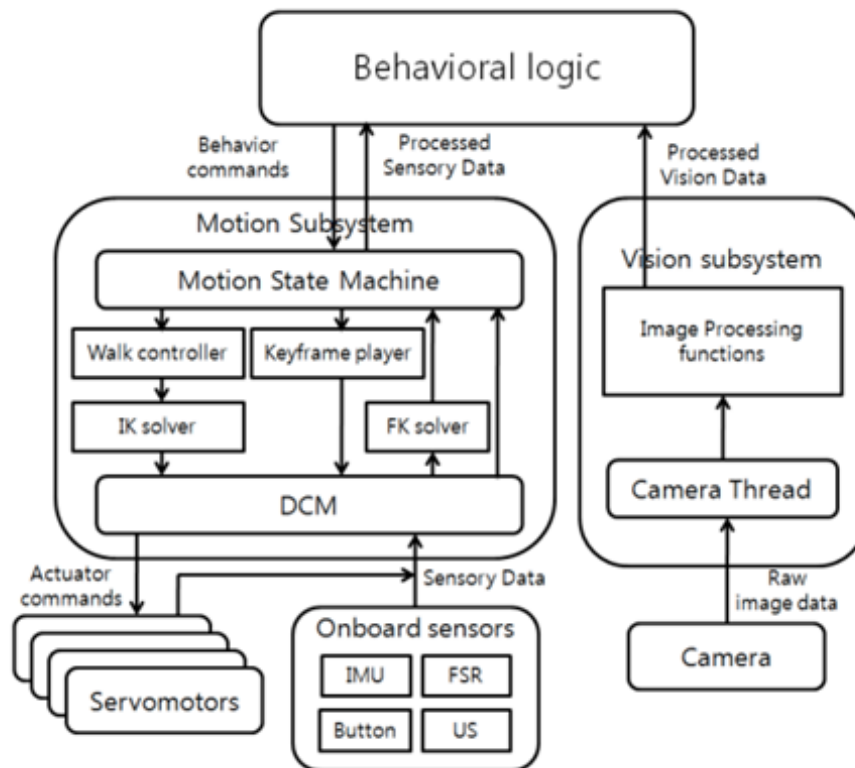


Figure 1.6.1

In summary the biggest amount of our efforts will be spent on the robot agents and their communication through the governing program. The project itself, given good documentation on our part, holds lots of potential for future students to dive deeper into the realm of intelligent agents and robot simulation.

1.7 Hardware

The hardware requirements for developers depend mostly on the ability to run Webots. Figure 1.7.1 shows the preferred Operating System versions. This is listed under hardware to relate to machines capable of running these OSs. Figure 1.7.2 lists preferred technical specifications.

	Supported OS
Windows	Windows 7 or newer preferred. Will not run on versions older than XP.
Macintosh	10.7 or 10.8 preferred. "Lion" and "Mountain Lion"
Linux	Does not run on Ubuntu 10.04(Lucid Lynx) or earlier versions. Preferred 12.04 (Precise Pangolin) or newer. Other flavors of Linux include RedHat, Mandrake, Debian, Gentoo, SuSE, and Slackware.

Figure 1.7.1: OS Support for Webots*

RAM	Minimum 2GB
CPU Specifications	Minimum: Dual Core @ 2GHz Preferred: Any Quad core.
Graphics Card	ATI or Nvidia card with at least 512MB Video RAM. For Linux only Nvidia cards recommended. Must be a card capable of OpenGL. Newer built in Macintosh cards also meet requirements

Figure 1.7.2: Hardware requirements.[10]

The Nao robot stands 23 inches tall and weighs 9.5 lbs. Under active use it runs 60 minutes and is powered by a 1.6 Ghz Intel Atom chip. For its vision two HD cameras

with 1280x960 resolution are used. The movement of the Nao robot is limited by the 25 degrees of freedom, same in the simulation as the actual robots.



Figure 1.7.3: Nao Robot, Details

For stability and its positioning the Nao robot relies on an inertial measurement unit containing an accelerometer, gyrometer and four ultrasonic sensors. The legs are build with eight force-sensing resistors and two bumpers. In the simulation this feedback is handled by the Nao robot templates that come with Webots for Nao.

Another important physical aspect of the robot is its ability to speak, which is allowed by the robo cups rules. Communication by voice may be used strategically if it does not interfere with the opposing teams robots. This would be the case where one team relies on voice communication and the other plays continuous white noise from its robots' speakers.

As can be seen be from this example, most teams prefer to rely on the inter-robot communication afforded by the controller program.

1.8 Research

Several publications were used by the group to get an initial understanding of Robotics and Artificial Intelligence. This was done with emphasis on robot soccer. This research was even more helpful because the team members have no

prior experience in this subject area. Included are implications and benefits of 3 documents.

1.8.1 CMUnited-98

Preliminary research for the team came from documentation referred by Dr. Sukthankar. The paper discussed the strategy and outcome of the 1998 Robocup championship for Carnegie Mellon's robotics team. In that year CMUnited-98 (the team) won all of its rounds thus winning the championship. A large portion of the document was, of course, very technical and not entirely in our scope of understanding. Though the following information was very useful and gave much insight into the design of our project.

Foremost, the governing algorithm for CMUnited-98: each robot must go through a perception/action cycle. In this case each cycle was 100ms, as it is for our simulation. Figure 1.8.1 shows the steps of cycle as documented by their team. Most importantly each robot must have as current of a "world model" as possible. Upon further reading we discovered that this world model includes all objects on the field of play, moving and stationary. See Figure 1.8.2. Each of these objects must be updated for each robot for every cycle. However for some cycles the robot may not have visual information on all objects but the world model updates nonetheless. Based on previous information such as velocity, location, and direction of movement.

- Assume the agent has consistent information about the state of the world at the end of cycle $t - 2$ and has sent an action during cycle $t - 1$.
- While the server is still in cycle $t - 1$, upon receipt of a sensation (see, hear, or sense_body), store the new information in temporary structures. Do not update the current state.
- When the server enters cycle t (determined either by a running clock or by the receipt of a sensation with time stamp t), use all of the information available (temporary information from sensations and predicted effects of past actions) to **update the world model** to match the server's world state (the "real world state") at the end of cycle $t - 1$. Then **choose and send an action** to the server for cycle t .
- Repeat for cycle $t + 1$.

Figure 1.8.1: Cycle algorithm from CMUniteds documentation.

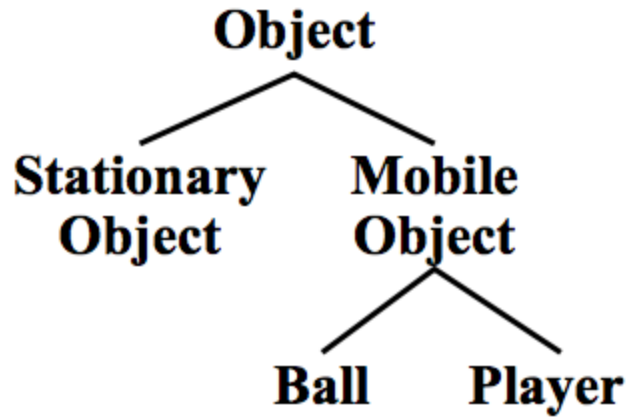


Figure 1.8.2: World Objects

This type of world model will be used in some sense in this project. Although there will not be as many objects to track, this is still taxing to design. In the CMUnited-98 case each robot had 22 objects to track: 10 teammates, 11 opponents, and 1 ball. The implementation of the governing program of each robot will have much of its foundation from the CMUnited-98 ideas. The complexity will most likely be less however.

The Carnegie Mellon team also describes individual action designs. Kicking and “Smart Dribbling” were covered in some depth. The more interesting aspects of the individual actions were how the team measured predictable success rate. (Figures 1.8.3 and 1.8.4) The individual modules would have a prediction that was handled by the decision making process. Each of these decisions were then weighted by the player’s location to other objects and to probability of the success of the action. As seen in additional videos and documents some robots may have the best decision to not move at all.

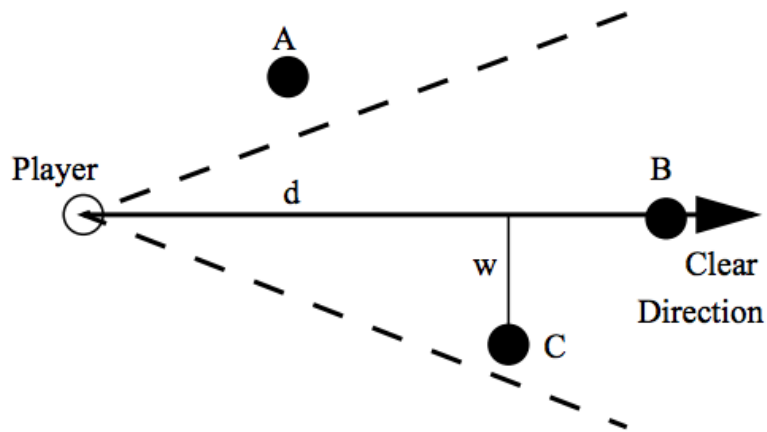


Figure 1.8.3: Success Determination of robot.

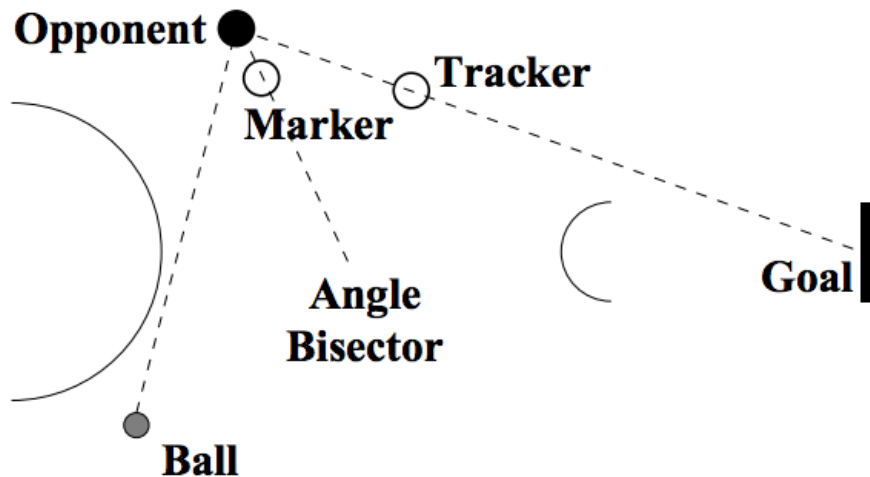


Figure 1.8.4: Success Determination II.

1.8.2 ARAIBO-07

ARAIBO, or Advanced Robotics with Artificial Intelligence for Ball Operation, is a soccer robots team with members from The University of Tokyo and Chuo University. They won the RoboCup 2007: Robot Soccer World Cup XI. After researching for soccer robot game theory, our team found the *Team Description of Team AARIBO 2007*.

The team description document expands on a theory of using dynamic programming to implement cooperative behavior. This theory involves the use of a decision tree, which is a feature we are likely to implement in our robots' decision making and gameplay. The approach uses a 8D state-action map for decision making on the playing field. Figure 1.8.5 below demonstrates 2 examples of potential gameplay within this theory.

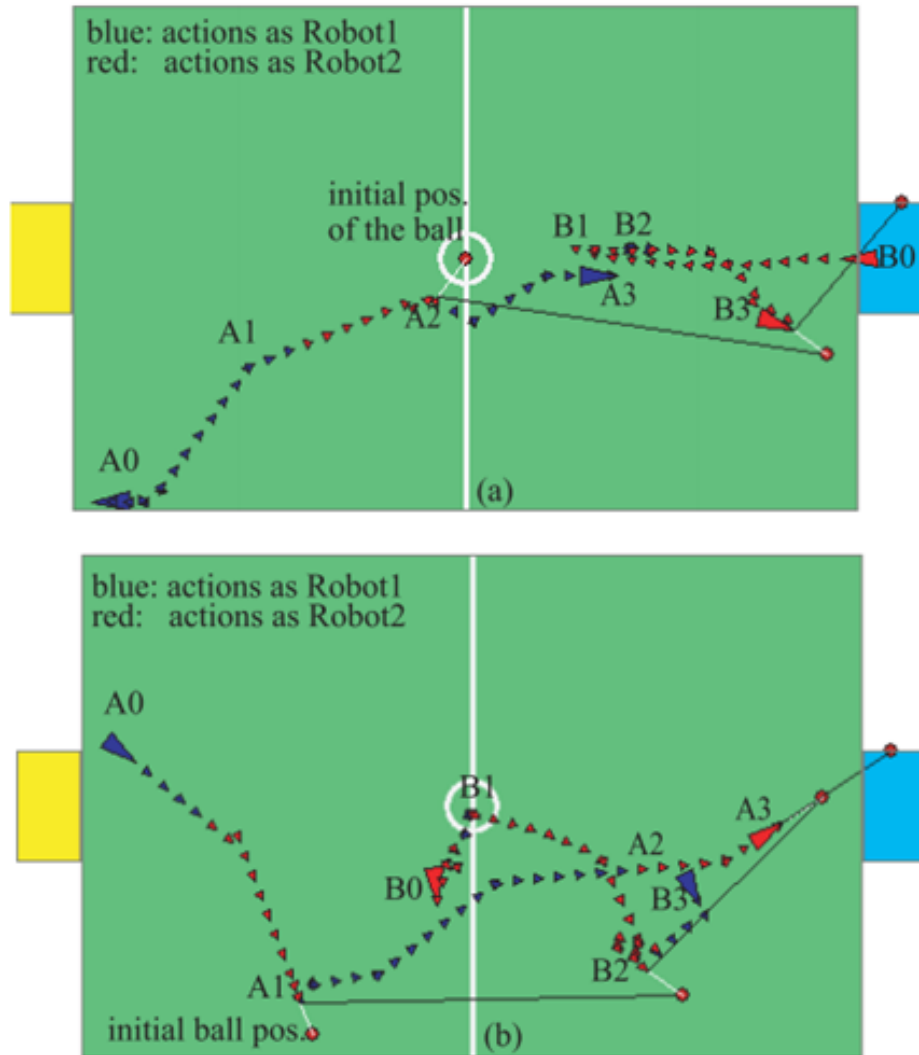


Figure 1.8.5

Over the years of competition, cooperative behavior and predictive decision making have proven essential to a successfully competitive RoboCup team. Inclusion of these components contributes to game playing strategy by allowing the soccer robots to play more similarly to the way humans do real life soccer games. The article does not go into explicit details about how these theories are implemented, rather is merely introduces the concepts of using them. The document also

introduces the concept of a fully DP goalie decision making process that involves working with the goalie's position and orientation and the position of the ball.

Overall, the document does not directly contribute to implementation, but may benefit us by sparking thoughts for further research that may lead to successful cooperative behavior and decision making.

1.8.3 Machine Learning for Arm Leg Coordination

The paper on "Policy Gradient Reinforcement Learning for Optimizing Arm to Leg Coordination and Walking Speed in the NAO Humanoid Robot" has been useful as inspiration for the improvements of our own set of movements. Machine learning was applied to train a robot to improve its own walking speed simply by the frequency in which the arms of the robot swing in regards to the leg movement.

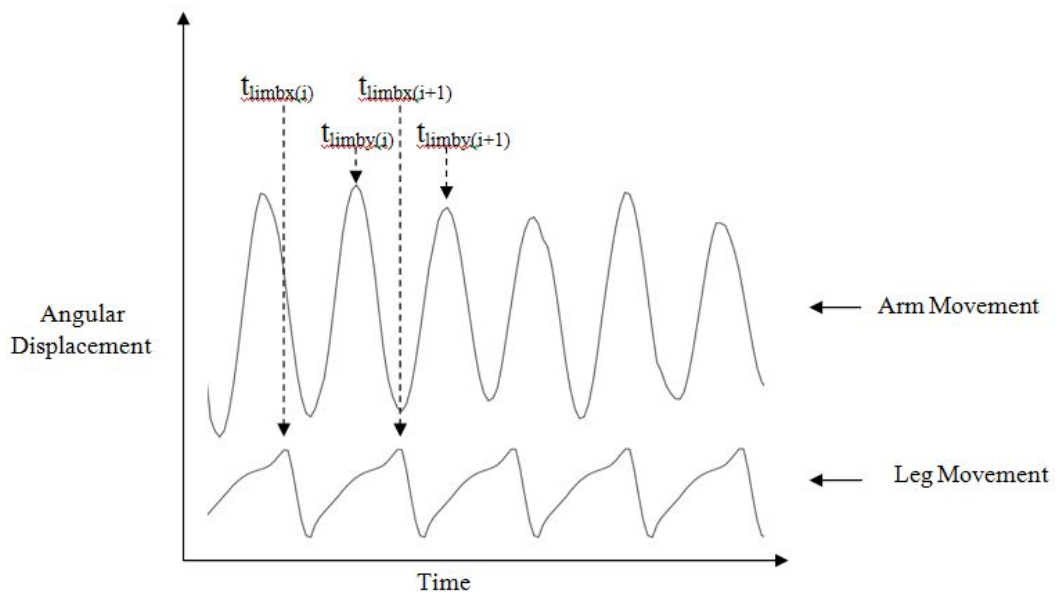


Figure 1.8.6

The results of the paper showed that the maximum speed could be achieved when each leg and opposite arm are in sync. Recognizing this fact; our improved kicking motion is based on positioning the limbs in a the same way for the moving kick. The increased speed allowing for a greater shot distance.

This process showed highly promising results and encourages pursuing the potential of employing machine learning techniques when refining our own

movements. Specifically in this case “the robot was able to increase its original speed of 33.22 cm/s to a speed of 37.85 cm/s”.

Section 2: Development Environment

During the course of this project there were a large amount of different software applications in use, as would be expected for Computer Science Senior Design. This software environment spans the range of tools for team organization, the simulation software for the NAO robot, libraries to extend functionality, and integrated development environments. Not all of them contributed equally in their usefulness; in retrospect some were used rarely as others dominated the attention of our productivity.

2.1 Team Organization

A software project cannot succeed without an effective team behind it. To aid in the creation of software, developers have always relied on tools to invest their resources for the greatest return, especially time and money. The following tools were implemented to aid Team Soccer Robots in the allocation of their time.

2.1.1 Google Sites Wiki

Much of the organization of Group 3's project relies on the infrastructure we have set up with our wiki. This Group Wiki was created from a template on Google Sites and then edited to fit the specific needs of our group. The site now serves as discussion board, task reminder, and a place for our documentation.



Figure 2.1.1

The first link on the website menu is an introduction to the project, including a description of it as well as a listing of the deliverable items at the end of two semesters and the objectives of the team.

Next is the team section which is a directory of contact information for all the people involved with the project, ranging from our Senior Design class instructor to our Ph.D. student adviser and our sponsor. The team is using mainly e-mail to communicate, with text messaging being used in urgent situation or for short questions. In summary it contains phone numbers, email addresses, and Github user names.

Another category is our guides for setting up the workspace. Given the amount of software being used in the project it was good to have installation guides set up in a location for everyone.

One of the major parts of a successful advancement of our results, long-term and short-term, are our weekly meetings. For an hour every week we meet in the Intelligent Agents Lab with our student adviser Astrid Jackson. We discuss problems, ask questions, relate difficulties, and are able to converse about anything relating to our project. These meetings evolved from help with the set-up of programs, to the discussion of simple implementations of robot behavior to the troubleshooting in specific functions and controller implementations in Webots.

Sitemap

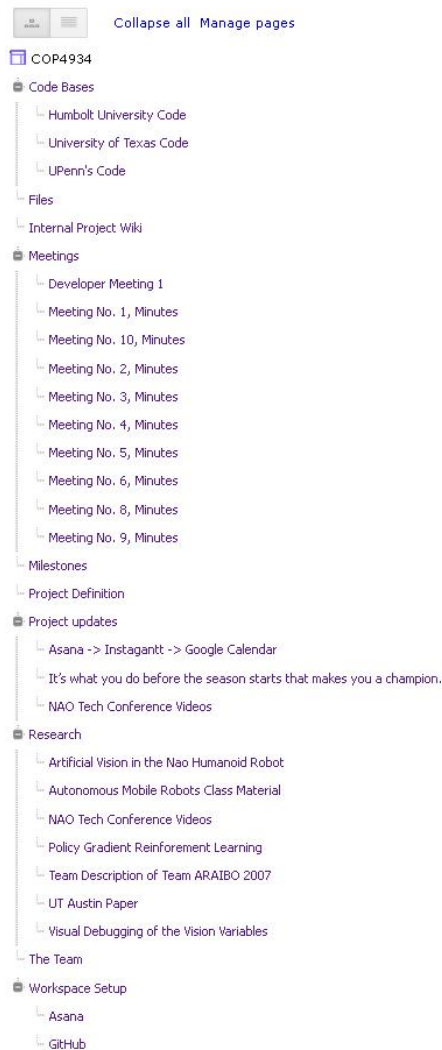


Figure 2.1.2

After “Meetings” the next menu item is “Milestones”. This gives us another overview of our tasks in the format of a google calendar. This calendar is synced up with Asana which will be discussed in detail later.

The Wiki also has a section for research. Here every team member posts their findings about anything relating to robot soccer, including general soccer strategies, Artificial Intelligence behavior, machine learning, specific code snippets or strategies used by other teams alike. Google sites includes an easily set up comment system for their websites, which is used to discuss the implementation, versatility, and feasibility of ideas brought up here.

When the code of other teams is discussed and it's available as open source, we also like to add it to another section on the wiki. Both the "Research" mentioned previously and the "Code Bases" section aid in availability of information should team members require to look up certain aspects in the future.

The main page of the website is designed as a motivator for the team members. It features a progress report that was started halfway through the semester. Every week the members post what they have achieved to keep Dr. Sukthankar and Astrid Jackson informed of our progress and the team itself on track.

2.1.2 Asana with Instagantt

Asana is an online project management tool. It has all the functionality that one would come to expect of such a product. It integrates with a convincing amount of additional services, including but not limited to Google Calendar, Instagantt, and Github. Log-in happens smoothly with your google account, something that we already basing a lot of our organization on, i.e. Google Docs and the Google Sites wiki.

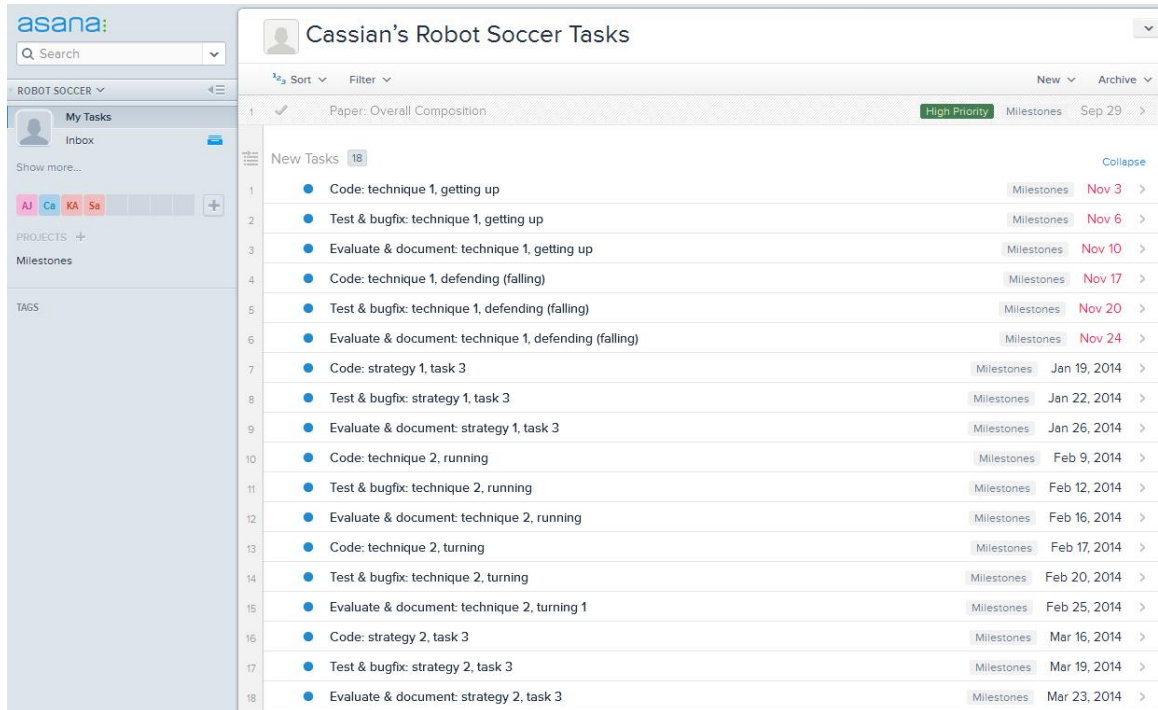


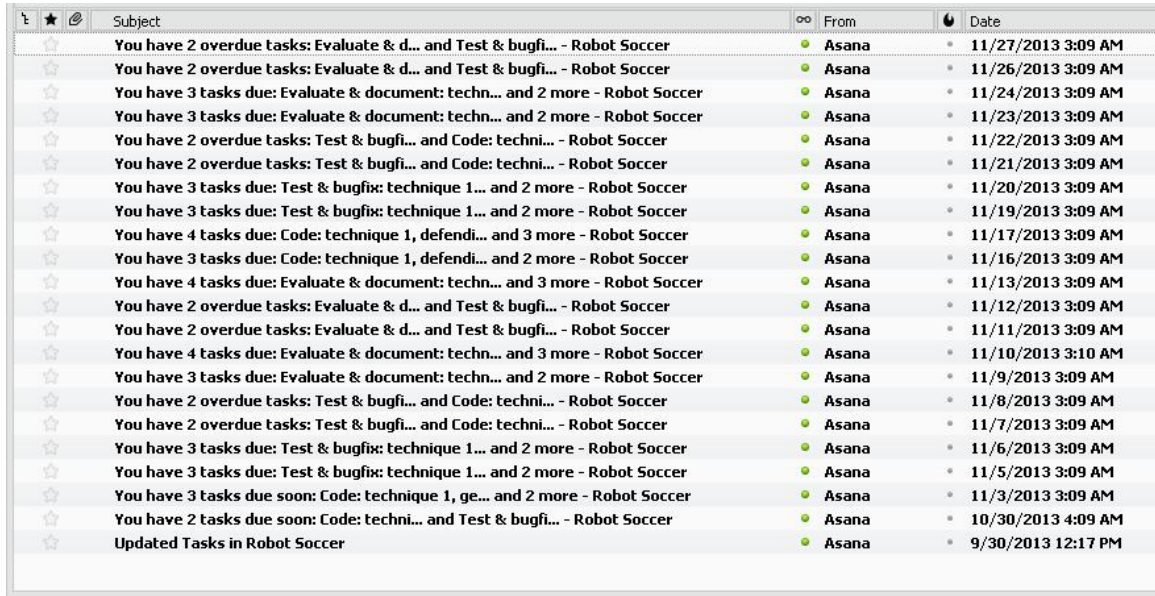
Figure 2.1.3

The way it functions is by setting up a project to whose group members are invited, which can be done by Google account names or e-mail address. Once the team is assembled task can be easily created and assigned. It even lets you quick add events into the currently selected column representing a day; you simply type the title into their quick add bar and hit enter. When it appears in the column of the calendar, it can then be edited to adjust the duration, owner, and dependencies. This is very useful when a large amount off tasks have to be added, as they can all be typed in rapidly and then have their duration and owners adjusted later on.

Any team member can become an owner of a task, or multiple people can own it. This is also visually represented by the task having a marker with the color of your group member icon when its opened. Notifications, are automatically turned on for the creator of a task and the owner of it. Group members can additionally subscribe to notifications for any task. They can be disabled manually in the properties menu of the task.

Notifications include reminders as well as status changes. If a group member is subscribed to a task's notifications, he will receive emails about task completion, partial completion, or edits to its properties. Additionally there are also three notifications about task in general: One about a task's upcoming due date sent a

day ahead, one to inform about a task being due, and one to inform about task being overdue.



Subject	From	Date
You have 2 overdue tasks: Evaluate & d... and Test & bugfi... - Robot Soccer	Asana	11/27/2013 3:09 AM
You have 2 overdue tasks: Evaluate & d... and Test & bugfi... - Robot Soccer	Asana	11/26/2013 3:09 AM
You have 3 tasks due: Evaluate & document: techn... and 2 more - Robot Soccer	Asana	11/24/2013 3:09 AM
You have 3 tasks due: Evaluate & document: techn... and 2 more - Robot Soccer	Asana	11/23/2013 3:09 AM
You have 2 overdue tasks: Test & bugfi... and Code: techni... - Robot Soccer	Asana	11/22/2013 3:09 AM
You have 2 overdue tasks: Test & bugfi... and Code: techni... - Robot Soccer	Asana	11/21/2013 3:09 AM
You have 3 tasks due: Test & bugfi: technique 1... and 2 more - Robot Soccer	Asana	11/20/2013 3:09 AM
You have 3 tasks due: Test & bugfi: technique 1... and 2 more - Robot Soccer	Asana	11/19/2013 3:09 AM
You have 4 tasks due: Code: technique 1, defendi... and 3 more - Robot Soccer	Asana	11/17/2013 3:09 AM
You have 3 tasks due: Code: technique 1, defendi... and 2 more - Robot Soccer	Asana	11/16/2013 3:09 AM
You have 4 tasks due: Evaluate & document: techn... and 3 more - Robot Soccer	Asana	11/13/2013 3:09 AM
You have 2 overdue tasks: Evaluate & d... and Test & bugfi... - Robot Soccer	Asana	11/12/2013 3:09 AM
You have 2 overdue tasks: Evaluate & d... and Test & bugfi... - Robot Soccer	Asana	11/11/2013 3:09 AM
You have 4 tasks due: Evaluate & document: techn... and 3 more - Robot Soccer	Asana	11/10/2013 3:10 AM
You have 3 tasks due: Evaluate & document: techn... and 2 more - Robot Soccer	Asana	11/9/2013 3:09 AM
You have 2 overdue tasks: Test & bugfi... and Code: techni... - Robot Soccer	Asana	11/8/2013 3:09 AM
You have 2 overdue tasks: Test & bugfi... and Code: techni... - Robot Soccer	Asana	11/7/2013 3:09 AM
You have 3 tasks due: Test & bugfi: technique 1... and 2 more - Robot Soccer	Asana	11/6/2013 3:09 AM
You have 3 tasks due: Test & bugfi: technique 1... and 2 more - Robot Soccer	Asana	11/5/2013 3:09 AM
You have 3 tasks due soon: Code: technique 1, ge... and 2 more - Robot Soccer	Asana	11/3/2013 3:09 AM
You have 2 tasks due soon: Code: techni... and Test & bugfi... - Robot Soccer	Asana	10/30/2013 4:09 AM
Updated Tasks in Robot Soccer	Asana	9/30/2013 12:17 PM

Figure 2.1.4

We integrated this software with the initially mentioned products. Exporting the Asana project to a Google calendar let us display our tasks on the wiki with the Google calendar plug-in, displaying the tasks in Instagantt made the information easier to visually parse, and linking Github commits helps everyone keep track of progress.

One of the plug-ins for Asana is Instagantt. It displays the tasks in Asana as a Gantt chart. Logging into Asana simultaneous logs you in here as well since both use Google accounts. Projects can then be imported one at a time to be shown as separate charts.

With the tools available here setting up projects under the umbrella of an agile development structure now becomes possible. Dependencies can be set between task by dragging a little protrusion at the end of a task to the beginning of another one. The dependency is now visible as a connecting line between the two lines.

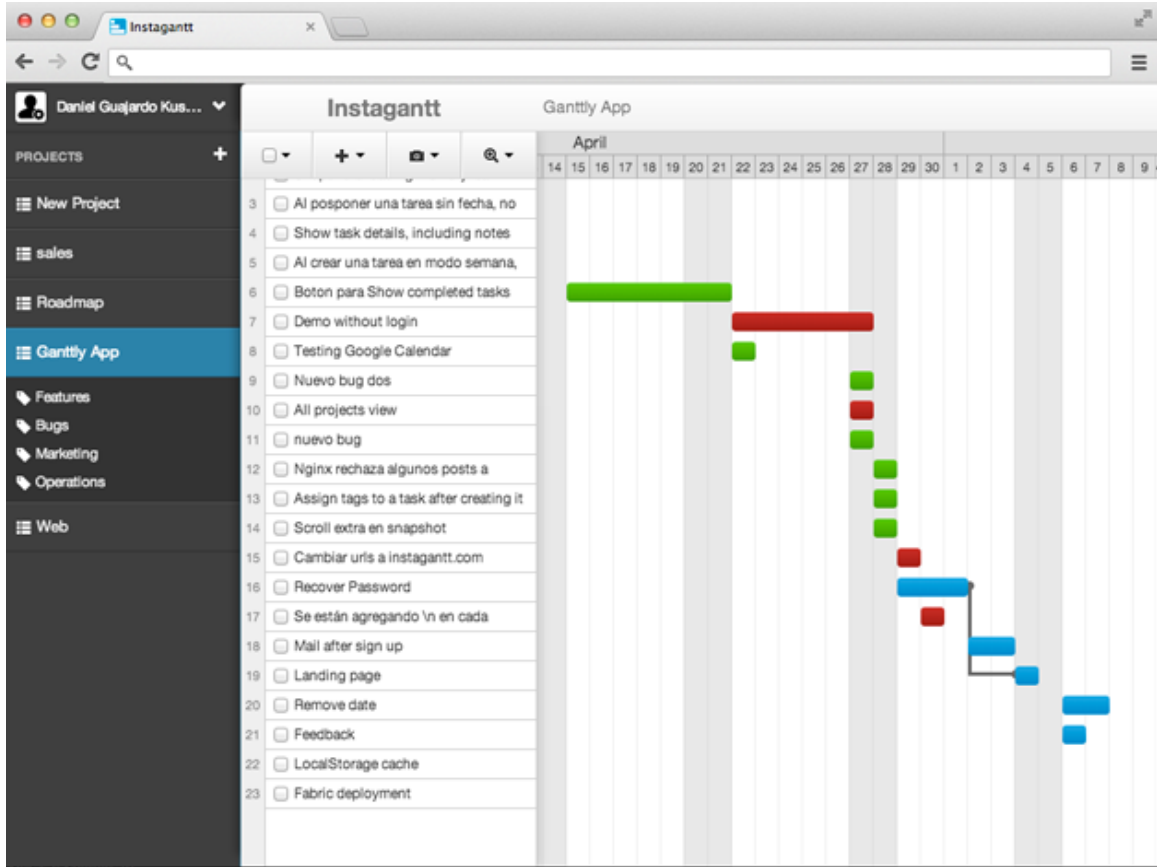


Figure 2.1.5

Once a project is imported Instagantt becomes the go-to application for planning out your task. Rather than setting start and end times for each task in a calendar field, one can move task by dragging and dropping them along the timeline and their duration can be adjusted by dragging the ends of the bar.

However we noticed some shortcomings with Instagantt as well. Even though you are able to set a task as a prerequisite for multiple follow tasks it is not possible to have more than one prerequisite. With many tasks there are multiple tasks that need to be finished as a prerequisite which cannot be shown adequately here.

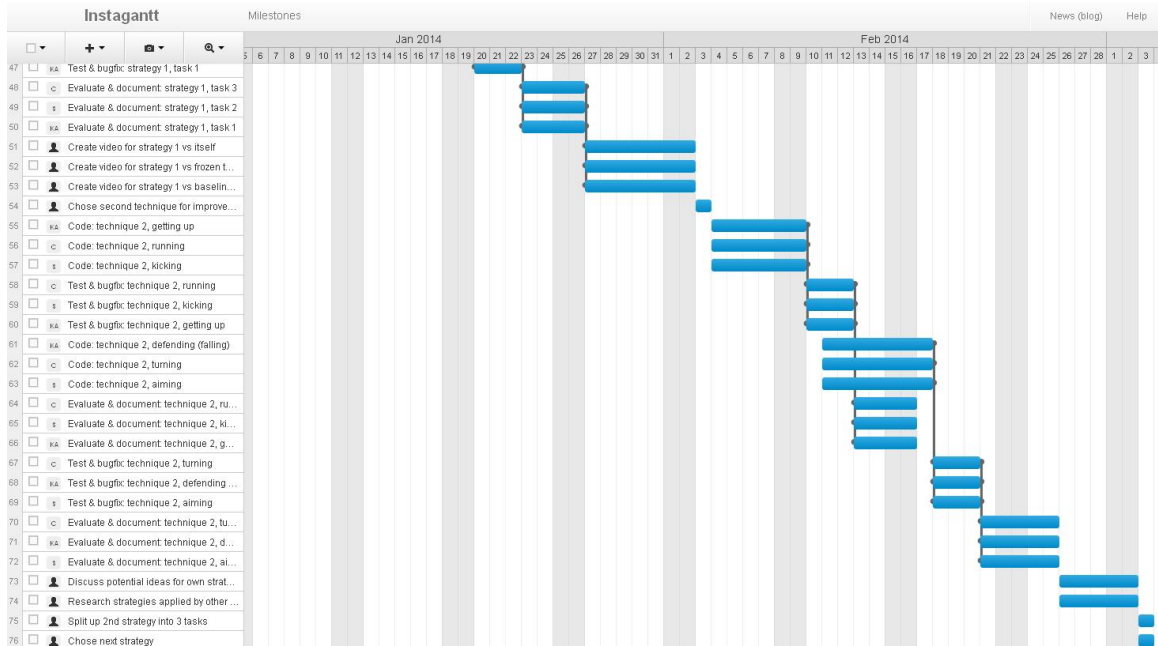


Figure 2.1.6

The website for Instagantt is not without glitches. Multiple times when moving tasks their duration was cut short one day. Other time when moving multiple tasks at once, the dependencies got moved by a day and looked different by having another line extending them vertically before connecting to the next task element.

A feature that was missing which would have been useful is that there is no coloring according to task owner. The only coloring is between unfinished and finished tasks. Instagantt also has no way to change the owner of a task, this has to be accomplished in Asana.

There were certainly some headaches to be had with planning all the tasks, but once everything is set Instagantt becomes solid concerning tracking tasks in the overall project timeline.

2.1.3 Github

Github is an online repository for software development projects. It uses the Git revision control system and is free to use for open-source projects. Github is the most popular online service for software collaboration. It was readily apparent that Github would be the best choice to jointly work on the code we are writing. The main platform being used by our team are Linux with occasional usage of Windows. All those are supported by Github letting us work on our OS of choice.



Figure 2.1.7: Nao-Ucf Repository on Github

Getting code updated on Github is a simple process. Once a member is added as a collaborator under the specifically for the project created repository they can send a pull request from their machine. This will bring the local copy of the code up to date.

Any change they make to the code locally gets back to the repository by a two stage process. When changes are ready to be uploaded, the state of the code can be committed locally. Once the code is committed it can then be pushed back onto Github. There it will be merged with the code on their, thus being available to all the team members.

Git Data Transport Commands

<http://osteele.com>

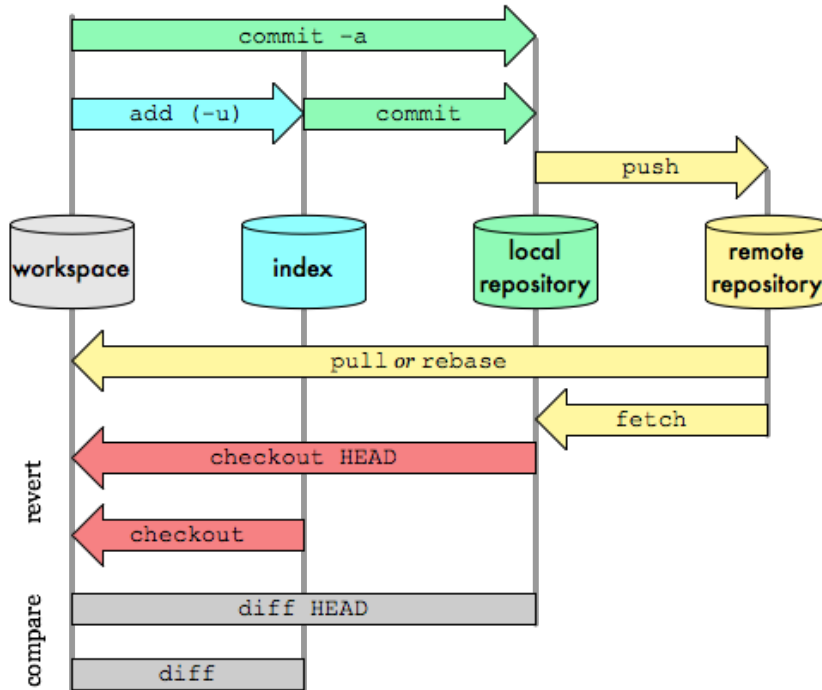


Figure 2.1.8: Git Workspace [9]

The most important part of this workflow is that any commits that are pushed back to the code online keep the code intact. If a partially unfinished code is added so that the controller as a whole does not run, then other team members are put behind on running the code to test their changes that came after.

Version control in Github is as simple as adding a tag to your commit naming the new version.

Another advantage of using Github is the plug-in that is available for Eclipse which is the IDE we are using exclusively for this project. With this configuration of software for our work-flow, everything we need to do is neatly available at our fingertips in Eclipse.

2.1.4 Team Organization Evaluation

Over the course of two semesters the standard workflow for the project changed tremendously. Initially there was a lot of individual work but a lot of communication over the wiki, eventually the wiki had served its purpose of getting

everyone up to speed and communication happened directly with a focus on collaborative work over github.

In the beginning of the project we created a wiki page on Google Sites for internal use. This proved very useful as a resource to share research on robotics and soccer simulations in general and specifically in regard to organizing the documentation provided by Aldebaran on Webots for NAO.

Useful was also the summary of meetings that were had with Astrid Jackson. It kept a record of the commitments to work to be done each week and cumulatively an overview of the progress on the project. In the second half of the semester the wiki was not utilized at all. The major factor was that discussion was focused on specific problems in the code that were usually addressed at that time in front of the computer.

Asana with Instagantt proved to be not useful at all. The complexity of setting up the initial timeline should have been an indication. There were at least 5 hours spend on setting it up. With this kind of opportunity cost to organizing the team, it was quickly abandoned to more traditional organization methods, namely the commitment to work on wiki as well as direct communication. In the end the actual progress of the team diverged so far from the Asana timeline that the weekly task due reminder emails became little more than “reminders to do work”.

What Asana showed nicely is the difficulty of projecting target goals into the future and the guaranteed problems that will arise and delay software, as our final deadlines, product, and results varied greatly from what was expected during the first week when the Asana timeline was created.

Getting Github to be adopted by the team proved challenging. In the first semester it was used almost exclusively for occasional backups of the software created up to that point. The collaborative aspect did not factor into the workflow. During the end of the first semester, with a codebase steadily increasing, Github started seeing use as a means to sync progress between the computer in the Senior Design cubicle and home machines. Only during the second semester of senior design and mostly to the end of it the team started realizing the potential of Github. Code was shared daily in the final phase and cooperation online in different locations started taking shape. By the end of Senior Design, Github had become an integral part of our team’s toolset.

We conclude that different tools showed their strength in different phases of the project, and specifically for Github, learning about it improved our value as programmers immensely.

2.2 Simulation Software

Simulation for the project was done using Webots for NAO v7.1.1. This is a commercially available software used for modeling robot behaviors in the real world. Alongside Webots Coreographe also played a small role in behavior design. Coreographe is not a simulator but rather a high level modeling software with three dimensional results of the robots.

2.2.1 Choreographe

Choreographe is an advanced programming tool for combining robot actions. The software allows users to highly modify how the behavior of the robot changes depending on inputs. On the surface it presents a very clean state-diagram of the actions that can be taken. (as seen in Figure 2.2.1.)

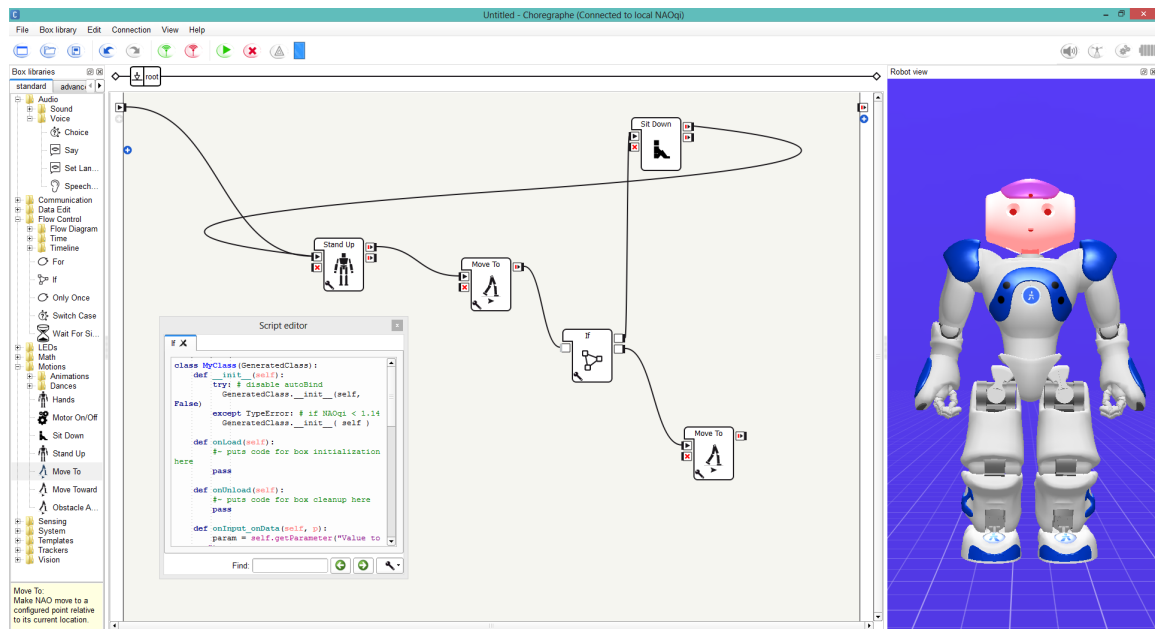


Figure 2.2.1: The application allows the developer to test in real time. Visible is the state diagram, the code for a single action and the robot simulation.

Users can edit the behavior of their robot down to the smallest detail by having the ability to edit the source code of actions. Beyond that, the application allows for users to upload their own libraries and run it on the simulator-Webots.

The developers for Choregraphe have a large document library online to help guide the user through the most advanced programming tasks. This is a very useful tool, and very well put together by the team at NAO. There is also a large amount of source code available in the soccer robot community, this will help kick start development for anyone not quite used to programming, and for doing simpler tasks.

2.2.2 Webots for NAO

The simulator specific to the NAO robots is Webots 7.2.4, also known as Webots for NAO.

The Webots simulator is an educational purpose simulation software for various robotics testing. The complexity of the modeling depends on the user and the general version of Webots (currently 7.3.0) can apply to a wide range of implementation. Webots is currently has a stable release available for Linux, OSX and Windows.

The software was originally developed by Dr. Olivier Michel, the founder of Cyberbotics-the company manufacturing the simulator. Dr. Michel started this project as a spin-off of the Microcomputing and Interface Lab at the Swiss federal Institute of Technology in 1998. He and his team still lead the company today.[6]

Webots uses the Open Dynamics Engine(ODE), which is a free software built by Russell Smith. ODE operates under the Lesser GNU Public License and has been used in video game design as well as other simulators. The physics engine in ODE is very well maintained and has extensive documentation via their wiki.[7]

Webots for NAO is, however, a lite version of the usual Webots. It is designed for the NAO robot only, and may not work if used for other simulations.[8] Webots 7.2.4 does include worlds built to simulate the soccer stadium(see Figure 2.2.2) that the robots compete in, and has the ability to import other simple objects.



Figure 2.2.2: Soccer stadium with player tracking ball object.

Webots can connect to Choregraphe or receive instructions from any other valid script from the developer. The simulation is linked to a network with each robot assigned a port. The network is by default the home network (127.0.0.1) and code should be changed if this is altered. The simulation browser shows the field(fig 2.2.3.a), console(2.2.3.b) and object settings (2.2.3.c). The simulation can be controlled above the viewing screen of the field. It can be paused or reset to original state. Users can save the state of the simulation to keep certain preferences or initial setup. As seen in Figure e.c objects parameters can be changed to reflect any real time condition being tested for. The Field also shows the camera views of the robots, both head and torso. Useful for testing ball-tracking or face-tracking.

Webots console will show as module methods accessed by the robots. If exceptions are thrown for the methods the content is displayed. Each cycle in the simulation is 100ms.



Figure 2.2.3: Webots browser, labeled

For the purpose of having the most optimal version of the simulator the simulations are run in version 7.1.1. This version is slightly outdated, however it performs better than the newest NAO version. It is explicitly mentioned by Aldebaran that results from this version are more accurate and that vision bugs are not present.

2.2.3 Simulation Software Evaluation

As mentioned previously Choreographe was not a useful tool in this project, although it can be used for modeling behaviors for less complicated work. What was ultimately needed was the environment that Webots created to run the code that was developed. Another thing Webots was able to deliver was a video rendering of the simulation that was running. The simulation, post-rendering, would play back in real time as opposed to the fractional speed it ran during actual simulation.

2.3 Libraries

The project was built upon two libraries, one inherently part of the simulator, the other added on after that half-way mark of the project. First there is the Naoqi API which allows for communication with Webots for NAO through the Python

programming language. This is used instead of Choregraphe and is necessary to have access to the more advanced functionality of the simulator.

The other library which soon became necessary is the OpenCV computer vision library for python. The limitations of NAOqi's vision could now be overcome by relying on the much more powerful OpenCV.

2.3.1 Naoqi

NAOqi is a set of core methods that set up the basic movement, memory, tracking, and connection proxies on the NAO robot. Aldebaran Robotics has set up a working framework available for download to aid anyone starting development with the robots. This framework is available in 8 programming languages overall with some having more support than others. In general the preferred languages for builds are C++ and Python. These languages have the most support on the Aldebaran forums and the most complete framework. Other languages include Java, .NET, Matlab and F#. Originally the group developed using Java since it best matches the UCF curriculum. After further investigation of the framework the decision was made to switch to Python.

Python is also well designed to work as a programming language for Artificial Intelligence. Easy threading and parallelism paired with how simple it is for a moderate programmer to pick up.

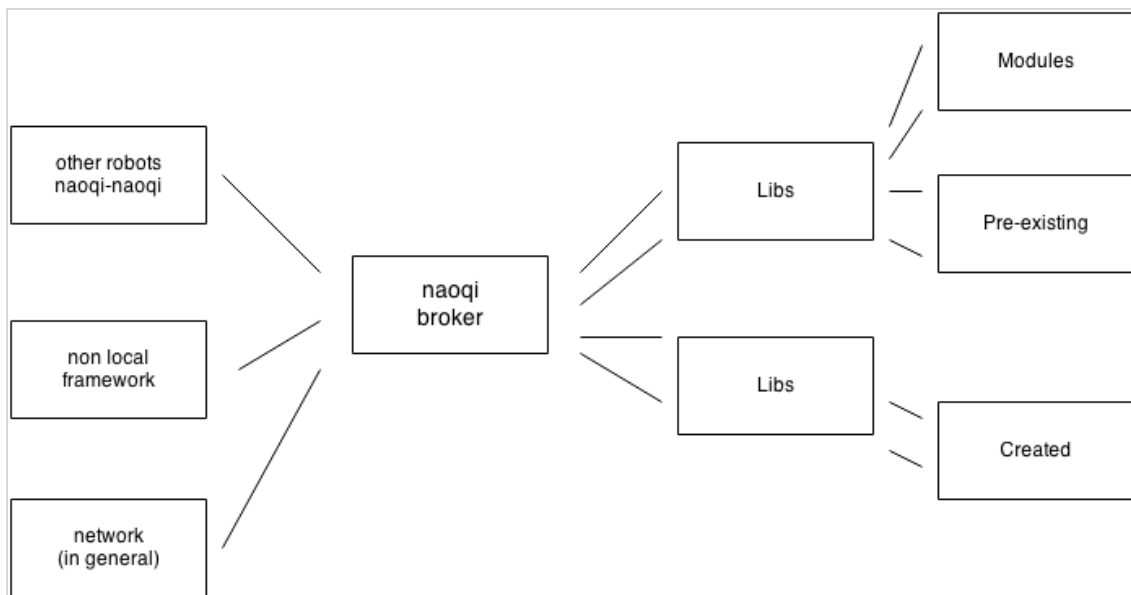


Figure 2.3.1: NAOqi linking.

The framework is called the NAOqi framework, and as mentioned the API is nearly equal for the top two languages. The NAOqi broker is the software that is running on the robot and loading and linking modules to implement behaviors. It is called the broker for obvious reasons, since it is the middle-man, so to speak, that connects all the code together. The libraries that the robot has access to are user specified and can be as complex or simple as preferred. Figure 2.3.1 illustrates the tree-like setup that the NAOqi software navigates. The main program imports any modules it needs for the run and those libraries are available to the robot. The available network can either be local or remote. But with remote frameworks the access time will dependent on network latency.

Most standard for simple coding is to make one module that makes proxies of already existing modules, as seen in Figure 2.3.2. In this script the module makes 3 proxies on lines 28, 35 and 41, while catching any errors. The object that is this module then has access to any and all methods belonging to that proxy. This is mostly just good coding practices and the programmers ability to keep the modules neat and organized.

```
21 def __init__(self, robotIp, robotPort):
22     self.ip = robotIp;
23     self.port = robotPort;
24 #     self.playerId = playerId;
25 #     self.teamId = teamId;
26
27     try:
28         self.motion = ALProxy("ALMotion", self.ip, self.port);
29     except Exception,e:
30         print "Could not create proxy to ALMotion";
31         print "Error was: ",e;
32         sys.exit(1);
33
34     try:
35         self.posture = ALProxy("ALRobotPosture", self.ip, self.port);
36     except Exception, e:
37         print "Could not create proxy to ALRobotPosture";
38         print "Error was: ", e;
39
40     try:
41         self.redBallTracker = ALProxy("ALRedBallTracker", self.ip, self.port);
42     except Exception,e:
43         print "Could not create proxy to ALRedBallTracker";
44         print "Error was: ",e;
45
46     self.naoBody = nao_body.NaoBody(self.motion, self.posture, self.redBallTracker);
47     self.naoHead = nao_head.NaoHead(self.motion, self.redBallTracker);
```

Figure 2.3.2: Python script for initializing modules into proxies.

2.3.2 OpenCV

OpenCV is an open source library for real-time computer vision. It has a long-standing reputation in the field of robotics for its quality and versatility. The main language of OpenCV is C++ but it also has a full interface for Python, allowing it to be integrated easily with the code base that already existed when the decision was made to switch gears with the computer vision.

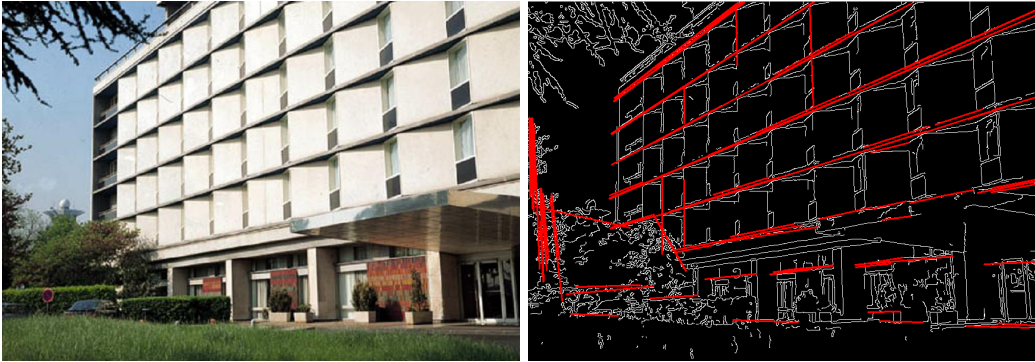
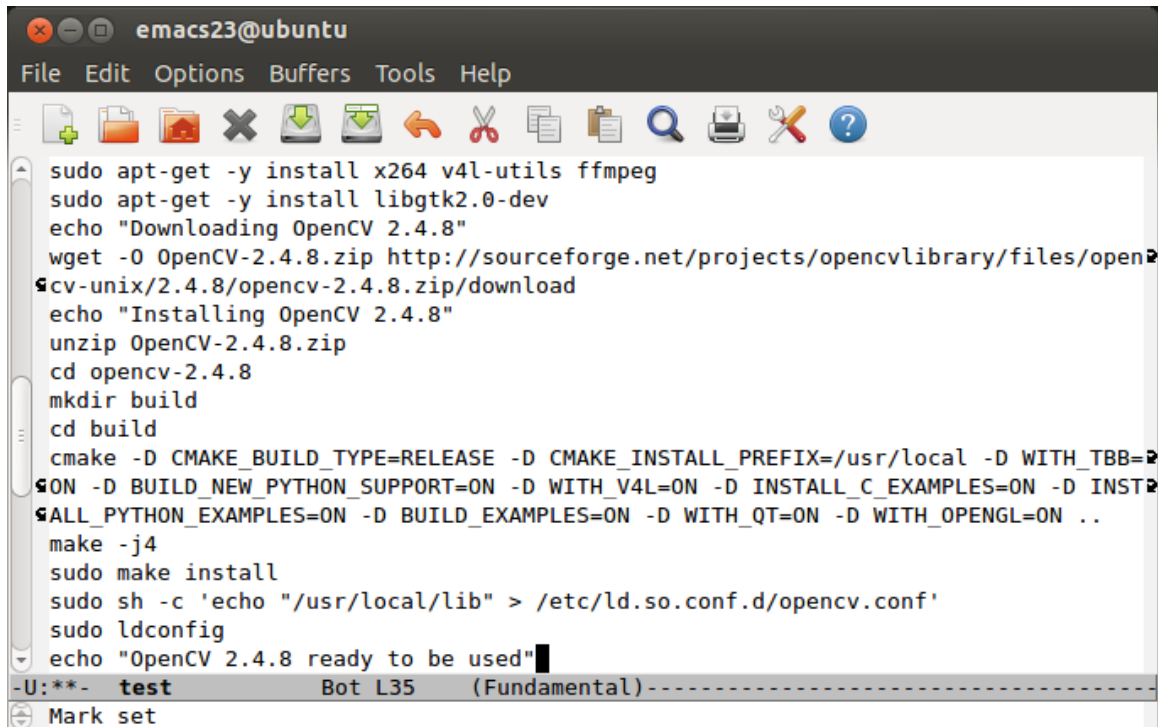


Figure 2.3.3

The power of OpenCV is undeniable. The library allows for the transformation of images based on color thresholds or other more complex algorithms such as Sobel Derivatives, Laplace Operator, or Hough Line Transforms. Also included is object detection, such as for balls and squares specifically or the detection of outlines, corners, and features generally. Another useful tool is the Canny edge detector in situation where shapes need to be found, but the expected colors in the image may change greatly, making color thresholding unfeasible.



```
emacs23@ubuntu
File Edit Options Buffers Tools Help
sudo apt-get -y install x264 v4l-utils ffmpeg
sudo apt-get -y install libgtk2.0-dev
echo "Downloading OpenCV 2.4.8"
wget -O OpenCV-2.4.8.zip http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.8/opencv-2.4.8.zip/download
echo "Installing OpenCV 2.4.8"
unzip OpenCV-2.4.8.zip
cd opencv-2.4.8
mkdir build
cd build
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D WITH_TBB=ON -D BUILD_NEW_PYTHON_SUPPORT=ON -D WITH_V4L=ON -D INSTALL_C_EXAMPLES=ON -D INSTALL_PYTHON_EXAMPLES=ON -D BUILD_EXAMPLES=ON -D WITH_QT=ON -D WITH_OPENGL=ON ..
make -j4
sudo make install
sudo sh -c 'echo "/usr/local/lib" > /etc/ld.so.conf.d/opencv.conf'
sudo ldconfig
echo "OpenCV 2.4.8 ready to be used"
-U:**- test Bot L35 (Fundamental)-----
Mark set
```

Figure 2.3.4: OpenCV-Python Installation Script

A concern when adding another dependency to the software was the complexity it would add for our target audience. Since a requirement for the project was also to create documentation that an introductory robotics class could easily absorb and use quickly, specifically within the timespan of a semester or less, considerations had to be given to the ease of setting up OpenCV.

The process of getting going with OpenCV in Python on Linux turns out to be a straightforward process. Available online is a simple script doing all the leg work and in all three cases of setting up machines to work with Webots and OpenCV we found that it worked flawlessly without exception.

2.3.3 Library Evaluation

Standing out most about the Naoqi API is probably its documentation. The arrangement of information was startling in its disorganization. A specifically glaring example is where in the documentation the Naoqi API main section has a Python subsection and a separate Python coding main section has a subsection with API methods, both with different examples some of which are the only explanation of further methods that are not mentioned elsewhere. Needless to say, finding specific information often became a long and arduous process.

Lots of initial effort was spent in organizing this information. Additionally it included sorting out deprecated functions. Also there were functions that are still supported but with the callee function having been changed, i.e. calling `move()` in turn calls a newer `walk()`-based function but passes in a parameter that is not in range of the newer function. The simulator makes this apparent by showing a change of the value to the maximum allowed value in the console.

When the modules for the movement started working there were many problems with the tracking of the red ball. Often it would return bad values for the red ball's coordinates. The solution was not to chase the red ball by moving to the coordinates of the red ball tracker, but instead moving in the direction of the head angle to its body once the red ball was tracked.

This was possible, because one effect of the red ball tracker call in the Naoqi API is that the head automatically starts following the red ball with its top camera. Given that the head will be pointing at the ball, we solved the issue of the improper coordinates returned by the tracker by instead querying the memory of the robot for the sensor values of the horizontal joint in the robot's neck. This value was given as an angle which we could plug directly into the *theta* value of the move function we were using, resulting in the robot very effectively moving towards the red ball when it was moved around in front of it.

Robots that can move and chase a red ball soon turned out to be insufficient at playing soccer according to our specifications. As the main focus lay on ball tracking, only after the solution to it did the shortcomings of the vision provided by Naoqi surface. Aside from the ball there are 4 more objects or multiple of objects to track. To be a competent soccer player every robot needs to be aware of his team mates, the opposing team's players, his own goal and the goal where he wants to score. This simply is not possible with the Naoqi API by itself, and a new solution became necessary.

After doing diligent research about computer vision and looking for a solution modular to our code, OpenCV quickly presented itself as the best solution available. An additional benefit of its long-standing tenure as a go to staple of open source computer vision, there is abundant documentation on the many features, with oftentimes surprisingly different applications of one and the same function, making the learning curve smooth and allowing for quick adoption.

One side effect of installing OpenCV on the system running Webots for NAO were updates to some media libraries. Specifically it updated ffmpeg to the newest version, a requirement for OpenCV. Since a previous version of Webots for NAO was chosen intentionally for its superior camera code, the new media library broke the internal recording functionality. The workaround was to use external recording tools instead.

In summary, the Naoqi API has a steep learning curve due to its horrendous documentation whereas along the same line OpenCV has excellent documentation making picking it up a breeze. Naoqi API and OpenCV in itself complement each other nicely and allow for effective work with NAO robots within the simulation environment

2.4 IDEs

The coding in the project was done using two main developing environments. These were picked for their ease of use, versatility or language support. The environments are Eclipse with PyDev, and Sublime Text. Both have documentation for setup available and both are available for use for further development of the code provided.

2.4.1 Eclipse

Code for the project was developed using Eclipse 4.3.x as an Integrated Development Environment. Benefits to using Eclipse in this project stem from the way Eclipse is built. Since the Software Development Kit for Eclipse is open sourced, any third party can develop a plugin. Thus making Eclipse into an ecosystem that can sustain a spectrum of projects. Installing and working with plugins is also very user friendly as shown in Figure 2.4.1. Eclipse requires the user to know the developers' download web address but will install and configure any available software with ease.

There is a general lack of well built Python developing tools for large projects. Originally looking to use simpler IDEs to escape the bloated nature of Eclipse, we discovered the lack of debugging support for other common environments. For a larger project such as this it was required to find software that caters for Python.

Pydev is a third-party plugin available from Appcelerator. It was originally developed as a project by Alexandar Totic but has since been acquired by several companies. Pydev centralizes all control to path variables in the workbench as well-see Figure 2.4.2. This allows us to extract and use the NAOqi framework provided by Aldebaran. Prior code testing without PyDev and Eclipse yielded different results. Path variables were often deleted or reset by the system after a reboot.

Pydev also allows for breakpoints in code, refactoring, and execution in a debugger as seen in Figure 2.4.3.

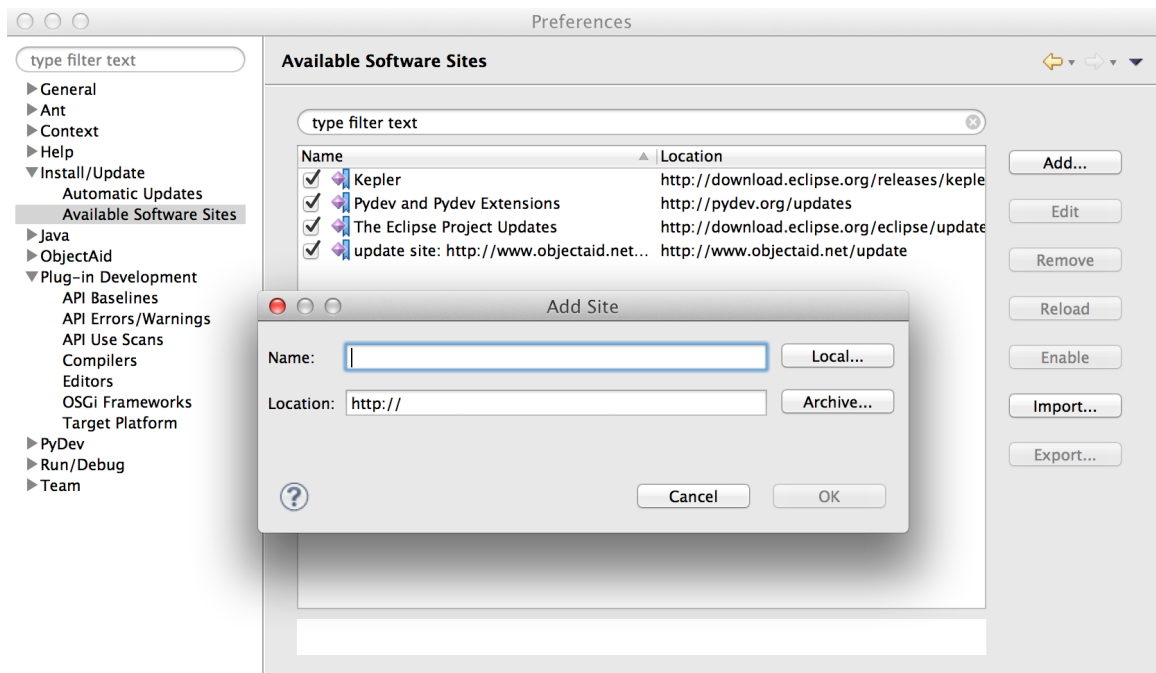


Figure 2.4.1: Plugin installation on Eclipse.

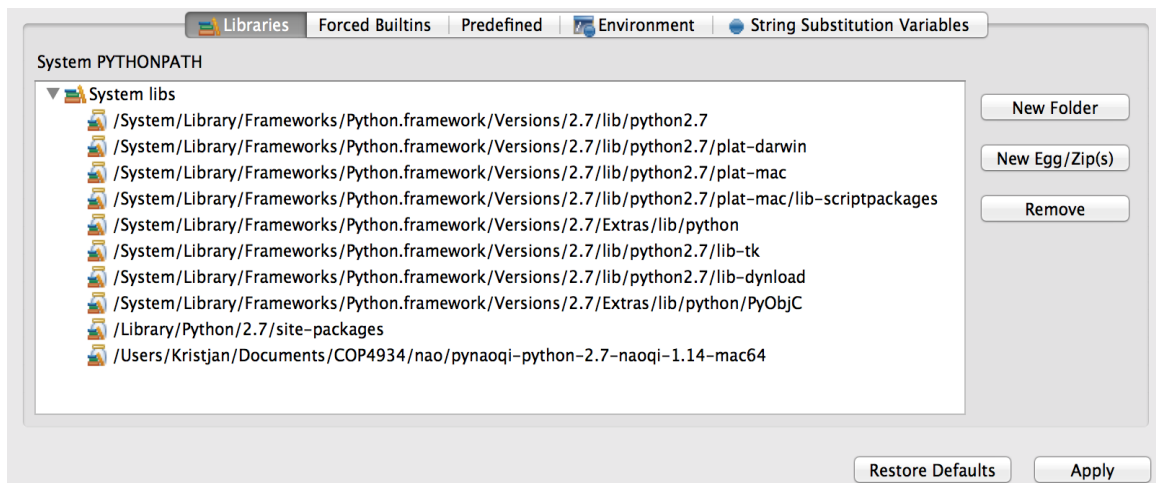


Figure 2.4.2: Management of Python Frameworks via PyDev.

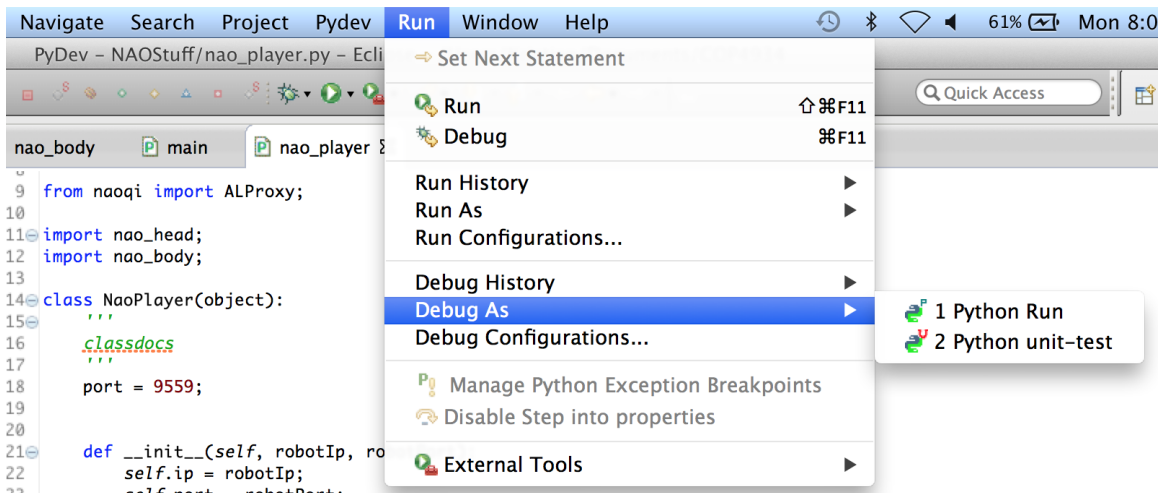


Figure 2.4.3: Debug execution.

2.4.2 Sublime

Sublime is a cross-platform source code editor with support for over 20 different languages. It comes with many editing features to streamline the coding process. This includes but is not limited to multi-line editing, go-to anywhere, and saving on loss of focus.

After switching to tracking with OpenCV it became the exclusive IDE for the project. Sublime allows for the creating of custom build configurations, where environment variables and other parameters can be defined.

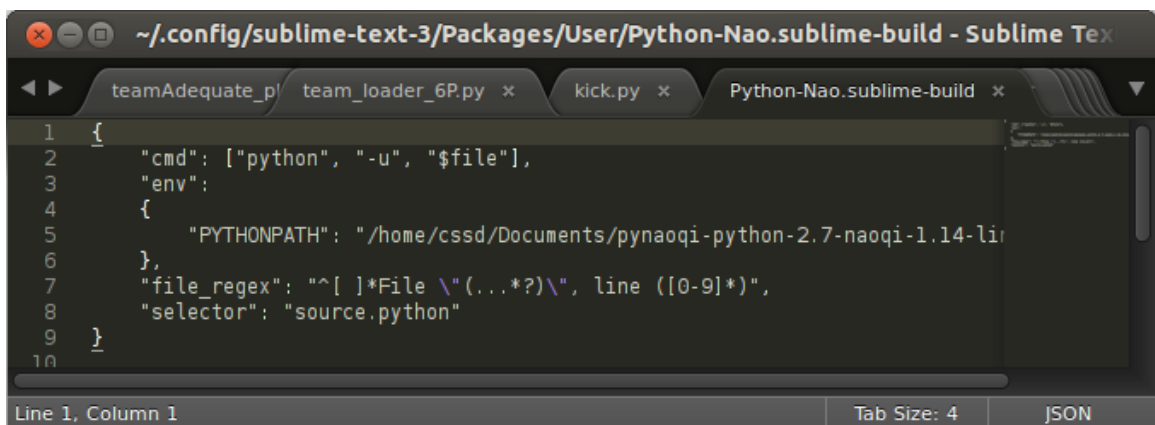


Figure 2.4.4: Sublime Custom Build Configuration

Another functionality that proved useful, especially since the code is written in Python, is the Tab definition and refactoring option. The number of how many

spaces a line should be indented upon hitting tab can be set here. The refactoring turns tabs in the code into the specified amount of spaces. It fixed the code many times after a line of code had been copied and pasted from the forums.

Sublime is proprietary software, but it has a non-intrusive trial period that outlasted the project's duration. For longer use the purchase of a copy would have been required, as necessitated by the EULA.

2.4.3 IDE Evaluation

Eclipse was the obvious choice for the project in its early stages. At that time Java was still the language that was used for coding and Eclipse is the staple IDE for many Java Developers. Since one computer was shared among all the team members it was necessary to choose a tool that all team members could use. Everyone on the team had used Eclipse before and was familiar with it.

The decision to switch programming languages came a few weeks into the project. This relates to the support by Aldebaran for the different programming languages in Webots for NAO. C++ and Python are the most integrated, with Java, .NET, Matlab, and Urbi enjoying very little support.

The documentation barely existed for Java and contained less than 10 examples of code. Examples ended up being very relevant in gaining an understanding of programming in Webots. Python on the hand had about 25 examples of code, and a very thorough documentation.

The Naoqi API itself is written in C++ so naturally documentation for it was the most complete, however by a small enough margin to not become a major factor in the decision for the project's new programming language. Python was favored by the team and chosen as the new programming language. Factoring into this decision was the preconceived preference towards it of the team, who were all new to the language, combined with the desire to learn a new skill.

The environment now had to be altered to accommodate the change in language. Since Eclipse had already been implemented in the workflow pipeline, it felt natural to search for tangential solutions. When looking into this issue it was quickly discovered that there exists a plug-in for Eclipse that allows coding in Python,

namely PyDev. It was an easy set-up, the plug-in installs right through Eclipse, that was used for the next 4 months for all of the coding.

OpenCV was the next big change to the project. As initially nobody was familiar with it or had ever used it before, getting it running was based on a lot of trial and error. Testing the installation with the Python command line soon verified its availability on the machine, however 2 complete days were invested in making OpenCV work in PyDev which never succeeded. For a week the coding was done in Eclipse, but the actual .py files were run on the command line where OpenCV worked.

Starting up the bulk of Eclipse simply to change a line of code seemed like a waste of time, especially since it was now just like a simple text editor. A new IDE was needed in which OpenCV would register so Python code could be run in it, to avoid further tedious use of the command line.

Upon recommendation from peers, Sublime was tested and subsequently integrated into our workflow. Importing the OpenCV libraries worked out of the box and once the Naoqi API library was appended to the PYTHONPATH, code could now be run directly from Sublime. This process was streamlined later, by adding the Naoqi API path in a custom Sublime build file.

In summary, Eclipse outlived the usage of Java during the project but eventually the sleeker Sublime, which loads up instantly, became the main IDE due to a better recognition of libraries.

Section 3: Build

Initially the robot behaviors and underlying system were tightly coupled. A differentiation could not be made easily between them. As understanding of the problem space increased by the team, a better framework began to crystalize. The structure of our code was also tailored to allow easy modification of behaviors by non-specialized users according to one of the project's requirements. In this section we will discuss all the assets that have been created to allow for the top-down integration of soccer behaviors for the robots.

3.1 Webots Field

The Field in the official Robocup live competition requires special attention for the robots to function properly. The green carpet has a length of 10.4 m and a width of 7.4 m. Marked upon this are the outlines of the field, which is 9 m long and 6 m wide. The goal post are 1.5 m apart. The modeling of the soccer field in the Webots simulator adheres to those measurements.



Figure 3.1.1: Field colors and layout.

A default Robocup soccer field is provided as a template within Webots for NAO. The template has two yellow goals and one orange ball. Both of these characteristics caused difficulties during the design phase of our framework.

Having two yellow goals was a change in specification going from 2011 to 2012 in the official Robocup simulation league. For our project design we decided to adopt the specifications from pre-2012, changing the color of one goal to a distinct blue (#2781BB). This makes tracking the target goal significantly easier for each team of robots. The decision was greatly informed by the fact that keeping a memory of the world state was part of our optional goals, which due to time constraints was not implemented. Without it, keeping track of the right goal for each robot would either have greatly reduced accuracy or limited movement in a significant way. It was decided this would be the better solution.

The color of the ball and its size was also edited. Our initial testing has shown that the default ball size in the simulation is sometimes insufficient to be recognized by the ball tracker module that's part of the API. For this and other modules relying on recognizing the ball its size has to be increased to 0.6 cm. The color was changed to red (#FF0000), making it easier for our vision module to differentiate the pink belts of one team from the orange default color of the ball.

3.2 Framework

The framework designed implements the objects and interactions between those objects using a set of Python modules described by the diagram below.

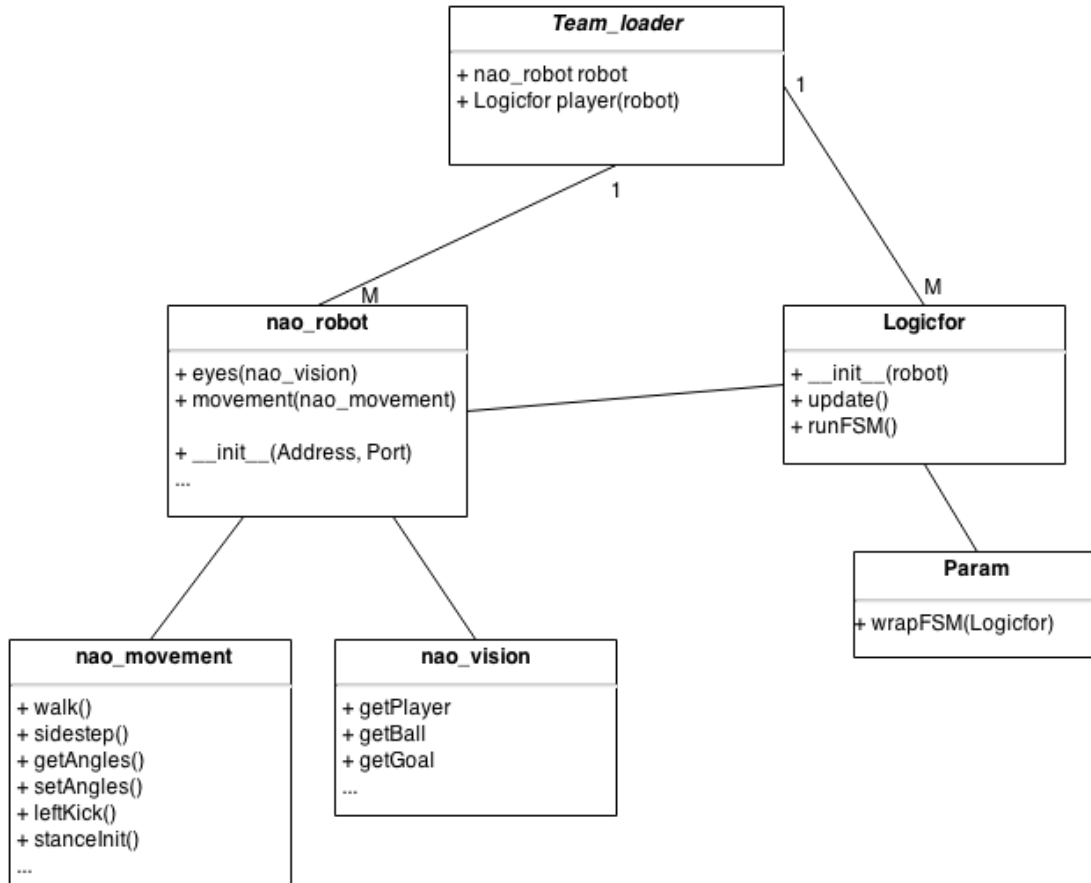


Figure 3.2.1

The Python interpreted scripting language was our language of choice due to the natural structure of the language. Python has various mechanisms that facilitate quick implementation of functionality. Dynamic typing in Python allows developers to create and then maintain short, readable code. This makes the developer's job easier by placing the tasks of declaring, tracking, and, checking data types into the hands of the interpreter. Just this mechanism alone makes Python code shorter and more readable than other static typed languages such as Java and C++. In addition, the built in Python libraries are written with short symbol names to couple with the idea of shorter code bases without creating unreasonably obfuscated naming conventions.

Another mechanism, forced indentation, follows the idea of short but readable code. The indentation of nested expressions is important in producing readable Python scripts. The indentation requirement that Python enforces effectively reduces the room for error when a developer is working with any given code base. Further, the

requirement makes code produced generally more readable and thus more maintainable.

Taking advantage of Python's clean-code mechanisms, the framework itself was designed with maintainability in mind. Due to the nature of this project, the requirements for soccer player behaviors are vast, and as with any artificial intelligence, functionality is key. Knowing this, our design choices consisted of those that would facilitate the addition of new functionality in future times.

We began the framework design process by analyzing the problem domain. Our scripts needed to be able to instruct a humanoid robot to play a game of soccer against other robots of the same type. Thus, the creation of the framework was driven by the existence of two main subproblems in the overall problem domain: One, how can the project be divided amongst the members of the team? And two, how can we logically implement the desired design functionality?

To answer the first problem we must look at what objects are present in the system and what responsibilities those objects have. A good place to begin is with the main agents in our soccer game, the players. A robot competitor is an obvious choice for encapsulation. The robot itself is a rather complex and general object. With the current subproblem in mind, is there a way to logically break down the robot into programmable modules? Now we can see that the subproblem reduces to one of design modularity when viewed in a problem-specific light.

The next step was to look at what the robots were logically composed of so that they could be more easily constructed in code. The robots are humanoid, and do have functioning vision and limb components. Coincidentally, those two components were crucial for carrying out a successful soccer game, virtual or not. We encapsulated the functionality of the legs and eyes of the robot and created modules to represent those function sets. Lastly, we ask, what drives these robot components? In the robot players, the scripted soccer behaviors had to be able to delegate commands to the robot's hardware components, or in our case the simulated components. Upon encapsulating the logic models we noticed that the soccer behaviors themselves create has-a relationships with the robots. This discovery was critical in the furtherment of our design. The has-a relationship that each subcomponent makes with the robot guided our design when modelling the relationships between the objects identified for encapsulation and implementation during the initial design process.

Ultimately, the need for a modular design due to the nature of team based projects caused us to devise a domain model that reflected, and facilitated the development of programs that were to be created in a modular manner. Now that we had a solution to the first subproblem, and incidentally a feasible concept for an extensible, modular, and maintainable framework, we could begin to construct a solution to the second subproblem, how can we logically implement the desired design functionality?

Taking the natural relationship of the objects identified earlier into consideration, the most logical design pattern to use was the composite pattern. As the name suggests, it is largely based upon the concept of composition. The Composition Design Principle places the general concept of composition into the light of object-oriented design in software. Conceptualized by the authors of “Design Patterns: Elements of Reusable Object-Oriented Software”, the composition design principle attempts to address the question of when to model is-a relationships versus has-a relationships in code.

To summarize, the composition principle states that one should favor object composition over inheritance. What this means is that rather than extending some class, call it A, and thereby creating a subclass, call that B, one should let class B simply contain a member of type A. By doing this the two classes are decoupled and thus support maintainability. For instance, if the code of class A needs to be modified or refactored due to additional functionality or changing requirements, then since class B is composed of A, class B is much less likely to need to be refactored or modified than if it was a direct subclass of A. What’s more, class B need not be recompiled after every change to class A, but instead, only when the external interface of class A changes, in the case of a method signature change for example. The type A member of class B in the composite pattern is effectively encapsulated and can be written, tested, and maintained independently of the classes of which it is a member. Due to the way that composite objects are constructed and used in typical projects, the external interfaces are generally more accessible when compared to those in inheritance trees. This allows the domain model and associated objects to be designed more simply, which lowers confusion and ambiguity, and minimizes the learning curve for future clients that may eventually use the code. Lastly, the composition pattern facilitate maintainability, and modularity, but it also allows for more complex combinations of objects to be implemented quicker.

In our design, the `nao_robot` class is composed of the `RobotEyes`, and `RobotLegs` classes. These objects contain methods for interfacing with the open computer vision libraries, and `naoqi` libraries respectively. The `nao_robot` itself exposes functionality of its `RobotEyes` and `RobotLegs` classes via its external interface methods which invoke calls on the appropriate member object instance. The complexities of the robot players are decoupled from the robot class itself which allows for multiple team members to implement the robot scripts at once.

The robot behaviors were encapsulated into their own classes and each one extended the `LogicFor` class to implement specific soccer behaviors. To a client developer using the framework, all of the `nao_robot` functions are visible and accessible from the `LogicFor` class, where they would be implementing their code. The `LogicFor` accepts an instance of `nao_robot` in its constructor function, so that no developer can instantiate useless rouge logic controllers. The `LogicFor` class then keeps a reference to the `nao_robot` that it is controlling so the developer can create a behavior by composing pieces of soccer behavior. Said pieces of behavior consist of methods that represent small encapsulated actions that a human soccer player would feasibly take in a real soccer game. These actions include, but are not limited to, sidestepping, kicking, seeking the ball, players or goal, rotating, and of course walking. The `LogicFor` functionality was uniformly controlled by the `FSMWrapper` class which handles the lifetime state of the `Threads` used to execute the instructions of the behaviors.

All of the components were unified under one centralized class called the `TeamBuilder`. This class provides the entry point to the program for the host computer and delegates the commands to start behavior threads on the physical or simulated robots via network communication methods implemented by the `NAOqi` framework.

3.3 Vision

The vision for all the robots is handled in a file called `nao_vision.py`. There the `RobotLegs` class is located that can be instantiated by the `Robot` class, giving each robot the access to all the methods necessary for vision. This section will give a brief overview of how vision recognition is achieved in the code.

To retrieve an image from the camera of the NAO a camera proxy has to be created of the type “`ALVideoDevice`”. Now that access to the robot is established through

the proxy the method `subscribeCamera()` is called to create a `videoClient` object which can then be queried to retrieve images.

```
# Parameters for Camera Proxy
self.camProxy = ALProxy("ALVideoDevice", self.IP, self.PORT)
self.resolution = 2 # VGA
self.colorSpace = 11 # RGB
self.bottomCam = 1
self.topCam = 0

# Subscribe to Camera Proxy
self.videoClient = self.camProxy.subscribeCamera("python_client", self.topCam, self.resolution, self)
self.videoClient2 = self.camProxy.subscribeCamera("python_client", self.bottomCam, self.resolution, self)
```

Figure 3.3.1

Parameters for the method are the resolution, the colorSpace, and the camera being subscribed to. Good results were obtained with the default values of the documentation, which were left unchanged. With a `videoClient` created, images can now be retrieved from the robots camera.

```
self.naoImageTop = self.camProxy.getImageRemote(self.videoClient)
self.naoImageBottom = self.camProxy.getImageRemote(self.videoClient2)
```

Figure 3.3.2

When the `RobotEyes` class is instantiated the first image is retrieved and stored in a class variable. To reduce the demands on the communication bus, functions that rely on the image of the camera are throttled meaning that if a stored image is less than second old it will not be renewed by an current camera image.

The first step in transforming the image for the purpose of retrieving information about environment is creating a PIL image from the pixel array that is returned by the camera. This is done by the `Image` library which is part of Python.

```
# Get the image size and pixel array.
imageWidth = self.naoImageBottom[0]
imageHeight = self.naoImageBottom[1]
array = self.naoImageBottom[6]

# Create a PIL Image from our pixel array.
im = Image.fromstring("RGB", (imageWidth, imageHeight), array)
```

Figure 3.3.3

The next transformation happens through `NumPy`, a dependency of `OpenCV`, which comes with its own set of arrays and is a library for manipulating them. The `NumPy`

array by default transforms the image from the RGB color space to BGR color space.

```
# Create a BGR Numpy pixel array from camera image
img = np.array(im)

# Convert BGR to HSV
img = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
```

Figure 3.3.4

Afterwards the first OpenCV transformation takes place. The color space is changed again to HSV which will allow us to use the OpenCV functions for applying a color threshold to an image.

```
# Red Ball
lower = np.array([110,150,150])
upper = np.array([130,255,255])
```

Figure 3.3.5

Thresholds are specified as two NumPy arrays constituting an upper and a lower bound. This specific example is applying a threshold to the color red in order to track the ball.

```
# Threshold the HSV image to get only blue color areas
img = cv2.inRange(img, lower, upper)

# Finding contours in the grayscale image
ret,thresh = cv2.threshold(img,127,255,0)
contours,hierarchy = cv2.findContours(thresh, 1, 2)
```

Figure 3.3.6

By passing the range to OpenCV the HSV image can now be transformed according to the thresholding parameters. The color in the image that is not within the range will be subtracted from the image. The next function, *threshold*, finds a level under which the image is converted into grayscale. From a grayscale image OpenCV can then grab the contours of the remaining object, which in this case is the red ball. The complete transformation can be seen below.

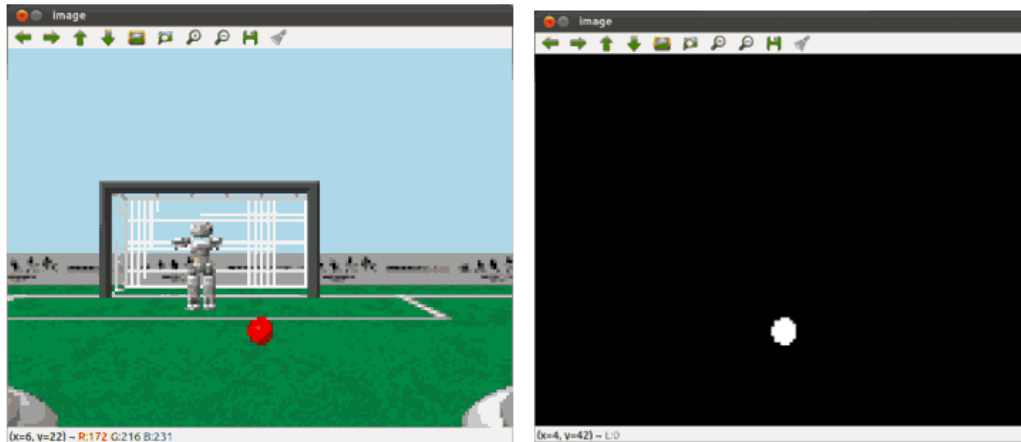


Figure 3.3.6: Red Ball Thresholding Transformation

One more step is required to make the vision information useful to the robot. From the contours of the shape of the ball OpenCV can create an array of moments of the object. These values allow the coordinates of the centroid to be calculated from a coordinate system originating in the top left of the screen. Passing the x and y value to the robot makes it possible in what direction the ball lies and how close it is to the robot.

```
# The function cv2.moments() gives a dictionary of all moment values calculated
M = cv2.moments(contours)

# m00 - contour area
# m10 - sum of all points distance to x-axis
# m01 - sum of all points distance to y-axis
# ==> centroid_x = M10/M00 and centroid_y = M01/M00
cx = int(M['m10']/M['m00'])
cy = int(M['m01']/M['m00'])

coords = [cx, cy]
```

Figure 3.3.7

The basic principle shown here is implemented in many functions in the RobotEyes class which can be called by the Robot class. There are functions for finding the position of both the goals, as well as all the teammates and opposing players on the field. Additionally there are functions to check the height of a goal to predict its distance. Lastly it should be mentioned that every single function has a counterpart sourcing the bottom camera instead of the head camera, otherwise fulfilling the same purpose, to increase the viewfield of each robot.

This toolkit of functions that were created gives the robots a complete sense of their surroundings, a necessary requirement for robot soccer.

3.4 Movement

We decided to implement a movement module that would house all of our robot abilities to move around the field while keeping our code as simple as possible. Thus, all locomotion for the robot is available in one class title robotLegs which is located in the nao_movement.py file. The name 'robotLegs' is relevant since nearly all locomotion for the robot has to do from the waist down. The movement methods are all specific to the object of which they belong. Furthermore, The proxies for motion, posture, and memory are all specific to a robot, thus they are not static. Ultimately, we constructed the basic movement methods with the hopes that it would contribute to an easy transition into building behavior modules, such as the chasing module, based off of the functions enclosed.

We began with the creation of a walking action within movement to allow for repositioning a robot as needed.

In addition, we decided to integrate the ability for the robot to sidestep by simply updating parameters within our module.

Stopping the movement of the robot was equally as important as starting it. Should we have utilized the available killWalk() method within the NAOqi API as it were, the robots may have terminated their walk in such a manner that left them in an unstable position. Therefore, in order for us to successfully stop the movement of the robot, we decided to make a killWalk() method that calls the stopWalk() in the NAOqi API, which allowed us to not only stop our agent from moving, but also assured that it did not stop too abruptly or end its movement in unsteady stance.

Also included among the movement methods is the kick. After many efforts of researching an optimal kick movement to run on our agents in simulation during gameplay, we found that the available open source python code for kicking was quite limited.

Thus purpose to develop the robot's kicking abilities was established. The goal was two-fold, first to develop a kicking strategy that improved the kick distance from a simple collision of a moving robot with the ball and then to improve it such that it can span at least half the field.

The challenge began with maximizing the distance of the ball traveled while keeping the robot from falling over as much as possible.

In order to maximize the efficiency of the shot we looked at the physical attributes of the best kicks found in human soccer. The best scenario for studying this sort of shot is the penalty kick in professional soccer. Not only does this provide an excellent analogy to our robots in-simulation moving behaviour, i.e. lining up the shot by sideways steps followed by starting the movement for the kick, but it also represents the technique for the highest accuracy and power combined. This lead us to believe that designing the kicking animation for our robots according to the optimal kicks in human soccer would yield the best overall shot.

To analyze a high quality shot, we found a document giving a detailed description of David Beckham's penalty kicks (arguably one of the finest penalty kickers during the peak of his game).[11]



Figure 3.4.1: David Beckham Free Kick

The motion is a double pendulum motion akin to that of the baseball swing. The lower leg is one pendulum and the upper leg is the other. Beckham cocks his lower leg all the way against his back to maximize the distance the foot will travel and therefore the acceleration of it. This is how he gets the most of the leg swing.

At the beginning of the kick he is seen jumping into the ball with his last step to maximize the kinetic energy his body holds. He moves his raised hand past his body to maintain balance and stop his body so that all the energy is transferred from the now resting body to the moving ball.

Our approach seeded from the idea of mimicking an optimal human approach to kicking the ball in a real soccer match, which can be reduced into four major steps:

1. Transferring kinetic energy from running into the swing of the action leg
2. Fixing the support leg such that it will maintain balance when the action leg begins its motion
3. Retracting the action leg, pulling it back far enough to put a fair amount of effort into the kick once it strikes the ball
4. And, releasing the action leg, swinging it forward toward the ball with maximal force

When implementing such actions on the robot agent, each action was broken down further into specific step by step motions compensating for limitations of the robot's abilities.

Before we discuss the contributing factors to our kicking animation, it must be recognized that the following limitations prevent certain implementation in robot soccer until the Standard Platform League deploys humanoid robots with equivalent balance and coordination as human players:

- First, the robots lose their balance quickly when they reach speeds remotely close to running. This restricts the use of the kinetic energy transfer from the motion of their bodies to the kick itself.
- In addition, even if the Beckham kick would be programmed into the robots, they would tumble over because they lack the momentary ability to adjust balance that we humans are endowed with.
- Last but not least is the limiting factor of the robot joints. The contractions and extensions of the limbs that are necessary for this kick are simply not possible with the Nao's range of motion.

Interesting, however, is what inspiration for our own modeling we drew from this analysis. First, we noticed that his extended arm plays a role in his balance. Naturally extending a limb far away from the center of gravity of its body requires and equal stress into the opposite direction to maintain the upright position.

This brought us to our first attempt of modeling kicking behaviour for the Nao robots. Our hypothesis was that if we can use one arm as counterweight for the kicking leg, we would be able to extend the latter farther and draw more strength out of our kick while improving the robots balance when entering the kicking motion.

Setting out to model this in Choregraphe, we used the timeline template that allowed us to set certain body positions at certain times which, when Choregraphe runs, it extrapolates the movement in between those positions.

Right off the bat, a major problem became apparent with using this method to create such behaviors. Modeling in Choregraphe is limited because it lacks the physics engine provided by Webots. In Figure 3.4.2 we exemplify the kicking animation that was modeled in the timeline template within Choregraphe; however, no matter how imbalanced the robot is in Choregraphe, it never falls over.

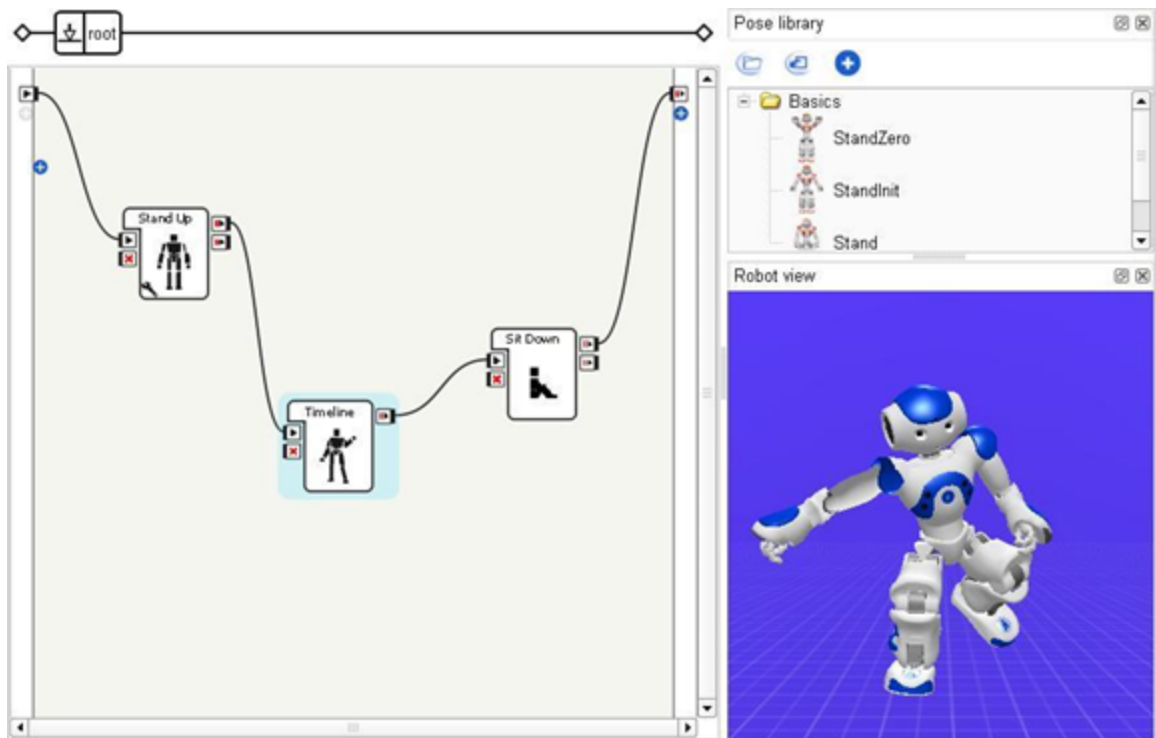


Figure 3.4.2: Choregraphe Animation

When the positioning was transferred into Webots, the Nao robots became very fickle; animations that seemed solid in Choregraphe turned out to yield undesired results. The animation that was crafted after the Beckham penalty kick turned out to be far more challenging to implement than we initially perceived. In its first iterations the robots fell over a lot, as can be seen in Figure 3.4.3.

Through a series of trial and error testing, by running the behavior in the simulator, we were able to make improvements by tweaking positioning of the joints, resulting in stabilizing the Nao more during his movements.

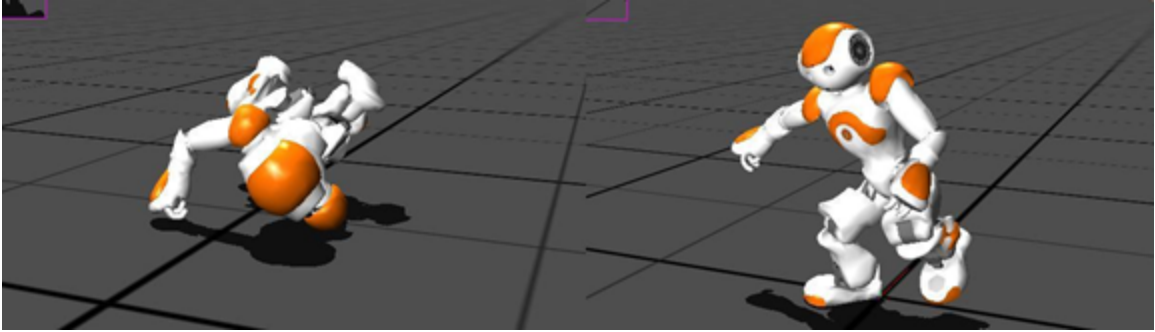


Figure 3.4.3: Animation Simulated in Webots

Still though, there was an undesired level of instability namely due to the issue of running through the positions quicker than the robot could handle.

Thus, we followed this implementation with a modification to the kicking movement based on a paper we researched that discusses the “biomechanical characteristics and determinants of indoor soccer kick”. The paper analyzes the properties of soccer kicks by specifying the linear velocity of the kicking person’s limbs, which proved to contribute to a useful advancement in our kicking motion.

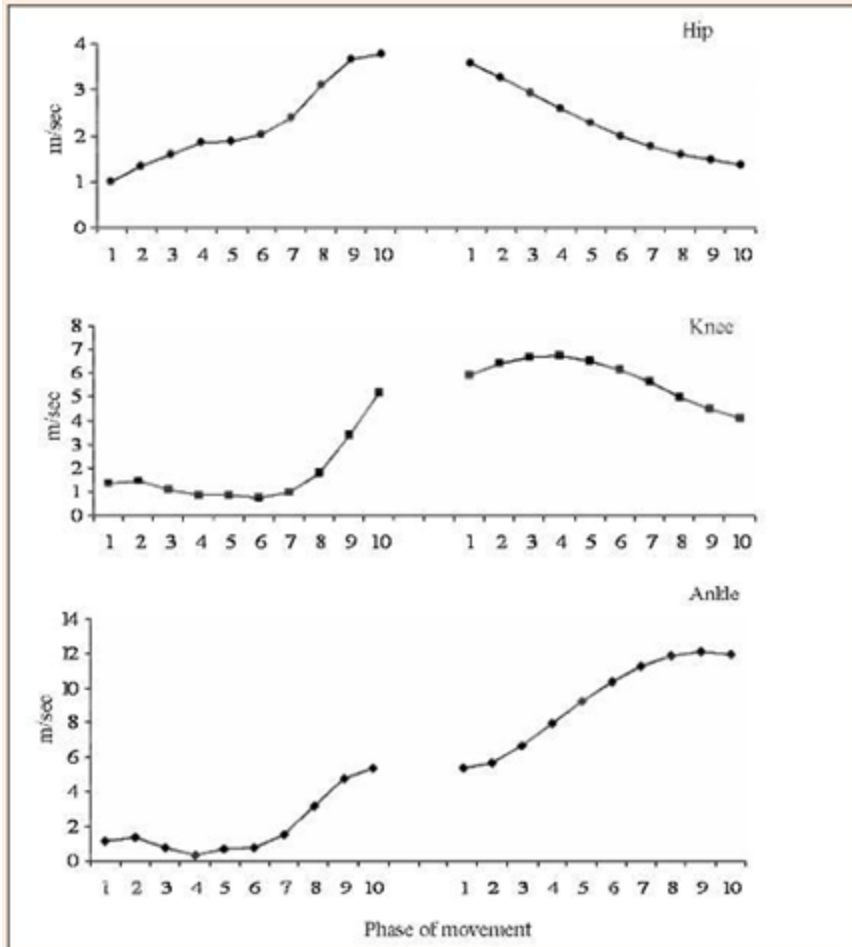


Figure 3.4.4: Ankle, knee and hip linear velocity during a soccer kick from the beginning of the motion to ground contact (left diagrams) and from ground contact to ball impact (right diagrams) separated to 10% for each phase.[12]

We established that the first step in modeling a decent kicking animation for the Nao robots was that the robot remains stable, on his feet during its whole duration. Following that, the second step was to make the animation as powerful as possible.

At first glance, Figure 3.4.4 seemed like another characteristic of the human soccer kick that could not be imparted onto a robot. However, we were able to make use of this chart in our robot soccer design by addressing one of the biggest challenges when creating motions in the timeline: dealing with restrictions on the duration between separate body positions due to the physical limitations of the NAO robots. As we found through testing the kick on robots in simulation, a few of our animations that eventually worked still caused the robots to tip over simply because it was going through the positions too quickly, causing instability.

We were able to utilize the linear velocity chart by syncing the duration between limb positions with the given linear velocity. The relationship is as follows: movement with a high linear velocity can be modeled to have a short duration between unique positions, causing the velocity of the moving limb increase, and vice versa.

At the conclusion of development for a stable kick, we recognized the main components of implementation and found that they were not far off from the four major steps of a human's approach to kicking the ball from which our initial approach seeded.

Although we eliminated the ability to utilize an initial energy transfer, we improved upon our understanding for what was needed for a successful kick on a humanoid robot, which includes the following:

- Activate balance on the entirety of the robot
- Fix leg constraints
- Activate balance on the support leg
- Run through a list of 3 positions at 3 specified time steps to simulate the leg swing action (as demonstrated in the figures below), which includes:
 - It starts with a subtle forward motion of the action leg, to allow for more efficient energy transfer into the next motion and ultimately leads to a more viable swing
 - Next we have a backward motion of the action leg, to increase the distance between the foot and the ball which increases the power behind the kick
 - Finally, we end with the action leg moving forward and striking the ball

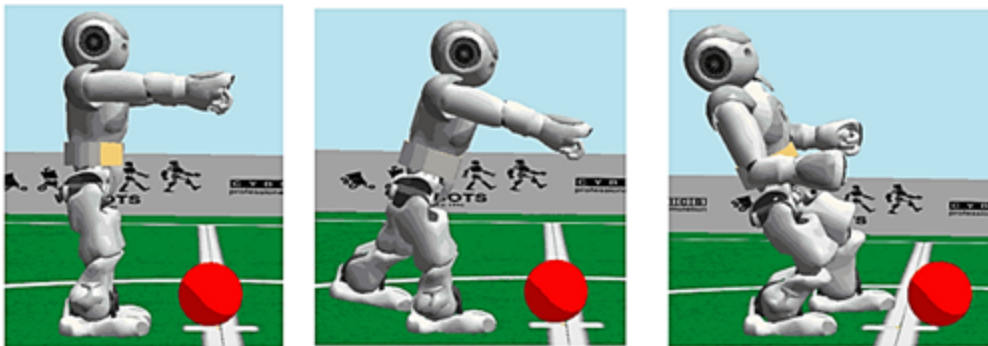


Figure 3.4.5

As we worked further with the kick behavior, making subtle modifications with each test run in the simulator, we ended our journey of developing the kick with achieving

the initial goals of creating a modified stable kick as well as implementing a power shot. We found that in order to kick the ball approximately halfway across the soccer field, i.e. the optimal power shot, we had to disregard the aftermath of the robot falling over, which was handled within behavior trees, as we discuss later, by simply standing back up if the agent had fallen. To us, this was an acceptable compromise since it not only had the potential to send the ball far enough away from the robot to allow him time to get up before needing to act again, but it ultimately results in a higher chance of the team scoring a goal by allowing accurate goal shots to be taken from halfway across the field.

What follows is a sample of our code, included so that the reader may recognize the steps within each action involved in creating the overall movement (as mentioned above) that results in the most optimal power shot.

```
def rightKick(self, prime, execute, cool_down):  
    # Activate Whole Body Balancer  
    isEnabled = True  
    self.motionProxy.wbEnable(isEnabled)  
  
    # Legs are constrained fixed  
    stateName = "Fixed"  
    supportLeg = "Legs"  
    self.motionProxy.wbFootState(stateName, supportLeg)  
  
    # Constraint Balance Motion  
    isEnabled = True  
    supportLeg = "Legs"  
    self.motionProxy.wbEnableBalanceConstraint(isEnabled, supportLeg)  
  
    # Com go to LLeg  
    supportLeg = "LLeg"  
    duration = 2.0  
    self.motionProxy.wbGoToBalance(supportLeg, duration)  
  
    # RLeg is free  
    stateName = "Free"  
    supportLeg = "RLeg"  
    self.motionProxy.wbFootState(stateName, supportLeg)
```

Figure 3.4.6

```

# RLeg is optimized
effectorName = "RLeg"
axisMask     = 63
space        = motion.FRAME_ROBOT

# Motion of the RLeg
dx          = 0.05           # translation axis X (meters)
dz          = 0.05           # translation axis Z (meters)
dwy         = 5.0*math.pi/180.0 # rotation axis Y (radian)

times       = [prime, execute, cool_down]
isAbsolute  = False

targetList  = [
    [-dx, 0.0, dz, 0.0, +dwy, 0.0],
    [+dx, 0.0, dz, 0.0, 0.0, 0.0],
    [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]

# Example showing how to Enable Effector Control as an Optimization
isActive    = False
self.motionProxy.wbEnableEffectorOptimization(effectorName, isActive)

# Com go to LLeg
supportLeg   = "RLeg"
duration     = 2.0
self.motionProxy.wbGoToBalance(supportLeg, duration)

# RLeg is free
stateName    = "Free"
supportLeg    = "LLeg"
self.motionProxy.wbFootState(stateName, supportLeg)

effectorName = "RLeg"
self.motionProxy.positionInterpolation(effectorName, space, targetList,
                                       axisMask, times, isAbsolute)

time.sleep(1.0)

# Deactivate Head tracking
isEnabled    = False
self.motionProxy.wbEnable(isEnabled)

```

Figure 3.4.7

With that, we conclude our current progress of the kicking animation. Following a human's kick as close as possible within the implementation of the humanoid robot's kick proved challenging and required quite a bit of trial and error before being able to refine such movement into our end result.

The movement section also controls the set and get angles of body parts. These are more necessary for tilting the head or resetting specific joints after a fall has occurred.

Lastly the file includes the method the robot calls to check if it has fallen during any cycle. The figure below illustrates foot sensor values to be tested when checking fall status.

```
314         key = "Device/SubDeviceList/LFoot/FSR/FrontLeft/Sensor/Value"
315         value = self.memoryProxy.getData(key)
316         self.LFFL = round(value, 2)
317
318         key = "Device/SubDeviceList/LFoot/FSR/FrontRight/Sensor/Value"
319         value = self.memoryProxy.getData(key)
320         self.LFFR = round(value, 2)
321
322         key = "Device/SubDeviceList/LFoot/FSR/RearLeft/Sensor/Value"
323         value = self.memoryProxy.getData(key)
324         self.LFRL = round(value, 2)
325
326         key = "Device/SubDeviceList/LFoot/FSR/RearRight/Sensor/Value"
327         value = self.memoryProxy.getData(key)
328         self.LFRR = round(value, 2)
```

Figure 3.4.8

These values constantly change during movement since the sensors are being triggered differently. In the case of a fall they hold their last given value, thus if there is no change for a period of 3 seconds we can safely conclude that the robot has fallen, as shown below.

```
352         if (self.oldLFFL == self.LFFL == self.oldLFFR == self.LFFR == self.oldLFRL == self.LFRL == self.oldLFRR == self.
353             return True
354         else:
355             return False
---
```

Figure 3.4.9

It should be noted for this section that in the simulation there was one sensor- specifically one on the rear of the left foot- that would constantly return 0 no matter what.

3.5 Logic Implementation

To create a more dynamic structure for loading and running players, each behavior to be created runs off of the same template. The figure illustrates the basic layout of every complex behavior created.

```
1  import sys
2  import time
3  from multiprocessing import Process
4
5  from util import nao_robot as robot
6  from util import parameters as param
7
8  class LogicFor:
9      def __init__(self, RobotGiven):
10         self.player = RobotGiven
11         self.isRunning = True
12
13         def __del__(self):
14             self.isRunning = False
15
16         def update(self):
17             while (self.isRunning):
18                 param.wrapFSM(self)
19
20         def runFSM(self):
21
```

Figure 3.5.1

Each behavior created will have its code inside the runFSM() method of the class LogicFor. The code structure is entirely up to the developer, it can be a finite state machine, a behavior tree, or some sort of hybrid structure the programmer designs. The runFSM code should be implemented in such a way that the code in it runs in cycles, thus no loops or thread-sleeps should be implemented in the code. An alternate implementation will not cause crashes but will run quite differently from the rest of the robots which all have the same framework. The update function loops through the code giving each cycle at least one second to finish.

To terminate the code the delete function can be used to stop the updates. The behaviors in the next section are implemented using this framework and are all designed in such a way that this template is utilized rather than looked at as a restraint.

Section 4: Behavior Creation

Using the NAOqi API, most of the basic movements were already defined with existing methods. However, the source-code that defined these methods was not available to us, thus there was no way to optimize and adjust. This resulted in manipulating the parameters as much as possible to achieve the most beneficial result. The movements we were most concerned about were the basic building blocks of our robot behaviors. These included: walking, kicking and sidestepping. From these simple movements the following behaviors were created. Combined these encompass a working team of soccer behaviors.

4.1 Chase

From the very beginning it was clear that giving a robot the same method call with no adjustment would result in unsatisfactory results. The API includes more than 10 methods just for walking. These methods take in parameters as simple as the desired X and Y coordinate, as well as more complex lists of parameters to define all joint control. After testing these methods we noticed that making several calls to `moveTo()`-a method we used to walk forward-resulted in the robot veering to a side from time to time. The robot would also often lose balance and tip over. The method did include a theta value which should control the curvature of the walk in the direction desired, as seen in the figure below.

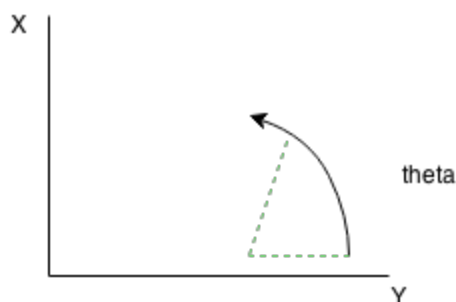


Figure 4.1.1

The robot views X as the forward axis and any movement sideways as on the Y axis. This angular change should be accounted for during every cycle of movement since slight differences will occur.

Another element added to every walk was arm-swing. The robots were naturally very stiff unless otherwise instructed. This caused the torso to not move when the lower body did thus causing the robot to quite often tip to the side. Although the behavior trees do account for the occurrence of falls it does slow down performance significantly to have to get up often.

The code below shows the fully parameterized version of the method call to walk along with a chart to illustrate what each value corresponds to.

Name		Default	Minimum	Maximum	Settable
MaxStepX	maximum forward translation along X (meters)	0.040	0.001	0.080 [3]	yes
MinStepX	maximum backward translation along X (meters)	-0.040			no
MaxStepY	absolute maximum lateral translation along Y (meters)	0.140	0.101	0.160	yes
MaxStepTheta	absolute maximum rotation around Z (radians)	0.349	0.001	0.524	yes
MaxStepFrequency	maximum step frequency (normalized, unit-less)	1.	0.	1.	yes
MinStepPeriod	minimum step duration (seconds)	0.42			no
MaxStepPeriod	maximum step duration (seconds)	0.6			no
StepHeight	peak foot elevation along Z (meters)	0.020	0.005	0.040	yes
TorsoWx	peak torso rotation around X (radians)	0.000	-0.122	0.122	yes
TorsoWy	peak torso rotation around Y (radians)	0.000	-0.122	0.122	yes
FootSeparation	alter distance between both feet along Y (meters)	0.1			no
MinFootSeparation	minimum distance between both feet along Y (meters)	0.088			no

Figure 4.1.2

```

36     def walk(self, value):
37         self.motionProxy.setWalkTargetVelocity(1.0,0.0,value,1.0,
38             [#LEFT
39              ["MaxStepX", 0.07],
40              #maxStepYRight,
41              ["MaxStepTheta", 0.349],
42              ["MaxStepFrequency", 1],
43              ["StepHeight", 0.015],
44              ["TorsoWx", 0.0],
45              ["TorsoWy", 0.0]],
46             [#RIGHT
47              ["MaxStepX", 0.7],
48              #maxStepYLeft,
49              ["MaxStepTheta", 0.349],
50              ["MaxStepFrequency", 1],
51              ["StepHeight", 0.015],
52              ["TorsoWx", 0.0],
53              ["TorsoWy", 0.0]])
54

```

Figure 4.1.3

If any parameter is left blank it takes the default value. The values set for this walk almost eliminated any tipping over and resulted in a smoother walk while tracking. The walk was also slowed down to avoid any difficulties.

Having a method call that provides better results was not enough however. To make the walk more optimal changes must be made every cycle to slightly adjust trajectory. To do this the theta must have a relationship to whatever object is being tracked in the field of vision. To illustrate this we will use the ball. The view below shows the robots head camera with the ball in view



Figure 4.1.4

The resolution of the image is 640 by 480 thus the middle line of the image is 320. If the returned coordinates of the ball are below 320 then the value that will be theta is calculated from the bottom of the frame clockwise to the center, otherwise theta will correspond to the angle created from the bottom of the frame counter-clockwise to the center line. Wherever the ball is results in where the angle stops. This value is then negated if the robot must move clockwise and positive otherwise. The angle is converted to radians. This value is then scaled down to twenty percent and passed in as the theta value. The reason for the scaling is to not cause the robot to move too severely resulting in a smoother recalculation of the trajectory.

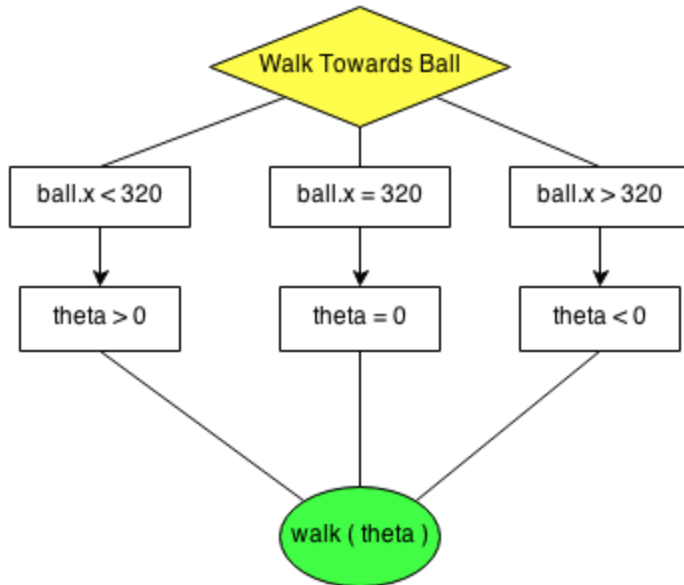


Figure 4.1.5

This algorithm produced the most desired results. The cycling of the walk trajectory must also take into account the wobble of the frame. At any point the returned image might be tilted one way or another. This is also a reason for scaling down the theta to a fraction of its true value. In earlier implementations when the parameter was the true theta- with a ceiling of 1rad and a floor of -1 rad-the robot would start doing very sharp turns to correct the path, this would cause the ball to be lost as quickly as it was located.

4.2 Goalie

As the team began theorizing what constitutes a decent goalie, we made the decision to implement the simplest behavior possible first and build from there. Thus, the logic behind the simplest goalie design includes the bare minimum functionality required by a moving goalie, which includes tracking the ball and attempting to block it from entering the goal. With that, we evaluated our options within movement and vision and concluded that following the logic of the behavior tree included below would satisfy these desired attributes such that we could use the goalie in a team-like environment and it would result in more goals blocked than the innate stationary goalie that we were working with in testing some of the other areas of development on this project.

The approach to developing the goalie behavior was established by means of implementing the Goalie Behavior Tree included below. The behavior begins by evaluating the leftmost branch to determine if the player has fallen, i.e. the player is not in the upright position it should be in to attentively defend the goal. If the goalie detects that it has fallen, it proceeds to take the action of getting up before continuing to evaluate other branches within the behavior tree.

Once the robot is no longer in a 'has fallen' state, it makes a decision based on its current field of vision. If the ball falls within range of the goalies vision it begins the action of side-stepping to align with the ball. This action requires further decision as to which side, if any, the goalie should step such that it results in him being lined up with the ball. Implementing this portion of the behavior allows the goalie to block the ball should it be kicked in the forward direction in an opponent's attempt to score a goal.

If the ball cannot be seen by the robot, i.e. it does not fall within the robot's field of vision; we have him take no action. This decision assures that he always stay near the goal such that he is always, with some respect, in front of the goal.

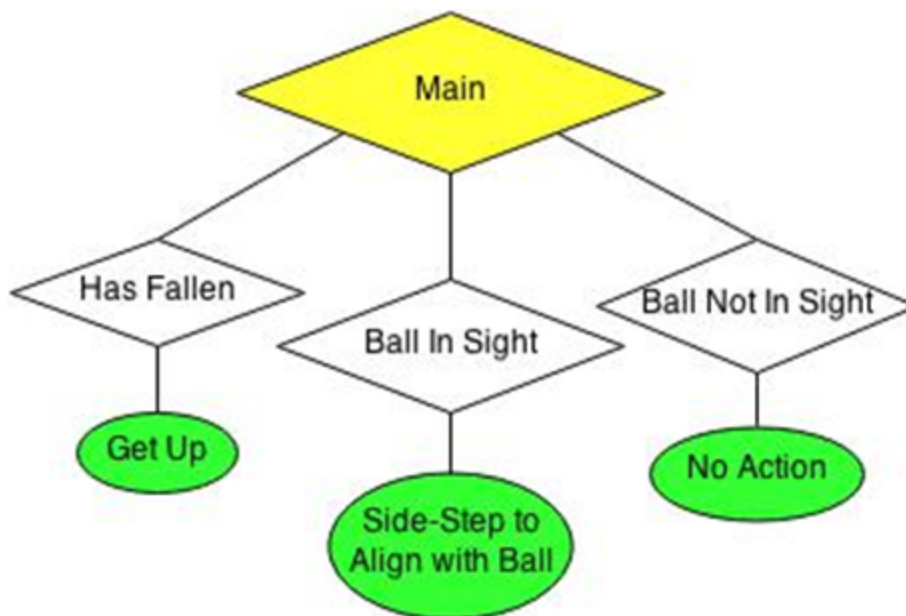


Figure 4.2.1

To assure that the goalie's behavior functioned as intended, we tested the implementation on an agent in Webots. We started by testing the simulation of only the goalie and the red ball in the field such that the goalie saw the red ball in its frame of vision. As we moved the red ball left and right with respect to the goalie's

position, we noticed that this attribute was successfully implemented. We also positioned the ball such that it left the goalie's line of vision to validate that the robot hinders any movement in such case, which also proved working as intended.

Once we completed the basic goalie behavior to our liking, we looked to do another subtle improvement that would increase the goalie's participation within gameplay. To us, this meant keeping precedence with the decision of blocking the ball from entering the goal but adding the functionality of getting the ball as far away from the goal as possible without leaving the goal vulnerable to another shot attempt.

Ultimately this improvement involved implementing the kick functionality which was already an element within the movement section. Thus, we improved upon the goalie's behavior by introducing the added decision to kick the ball should it be within range of kicking it. We made the decision to use the stable kick rather than the powershot since we wanted the goalie to always be standing after kicking for immediate return to his goal-blocking purpose thereby eliminating the opportunity for the opponent to shoot within the time it would take the goalie to get up.

Since this was merely an adaptation of the code for the simple goalie, the behavior trees look quite similar in manor, that is, the decision making process follows the same logic with one major modification. As the reader may identify in the behavior tree below, once the modified goalie has the ball within its field of vision, it makes a decision of action based on whether it sees the ball out of the upper camera or the lower camera. If the goalie sees the ball in the upper frame, it follows the same behavior as the simple goalie, that is, side-stepping to align with the ball. If, however, the ball falls in the view of the lower frame, the goalie is close enough to attempt kicking the ball and positions itself for the kick. It then proceeds with calling the stable kick method discussed within the movement section.

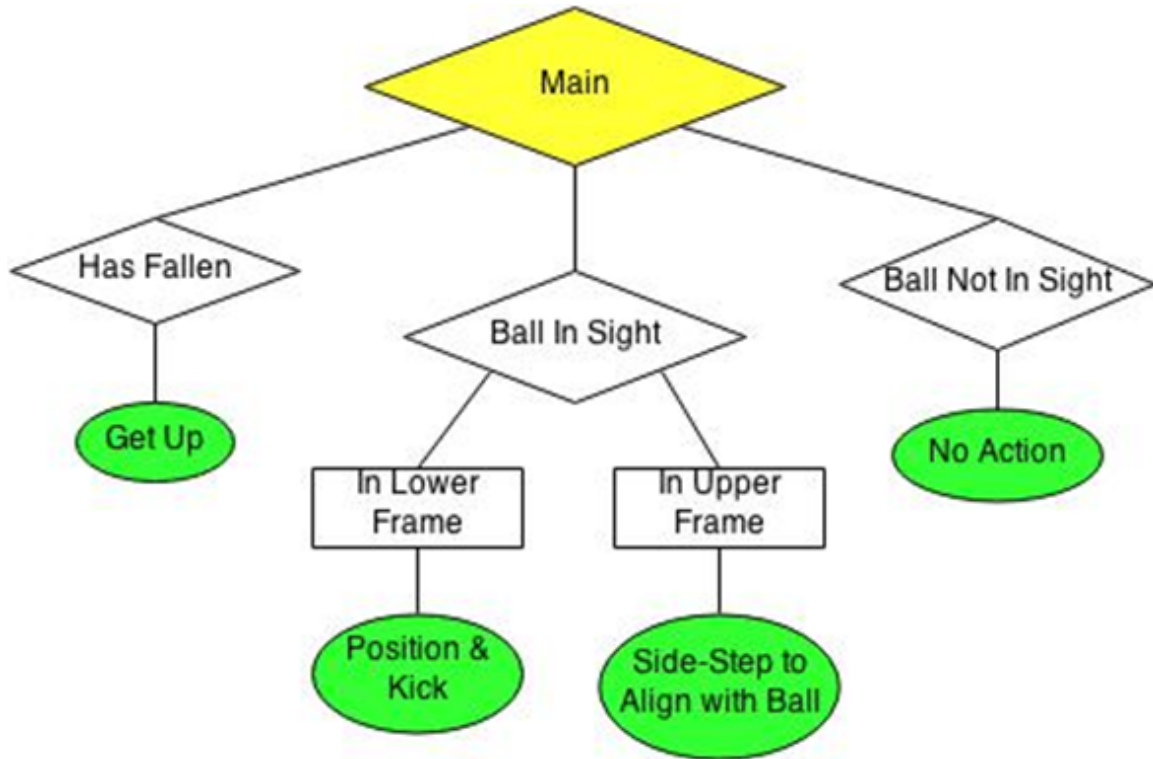


Figure 4.2.2

All other logic within this modified goalie did not change from that of which it was based, the simple goalie. This means that it still has the ability to detect if it has fallen and get up as well as the implementation of taking no action should the ball not fall into the frame of either camera.

Below we have included a segment of code that supports our implementation of the modified goalie behavior with hopes that reader may follow the logic of the behavior tree with respect to how such behavior was implemented.


```

def runFSM(self):
    #if has fallen, get up
    if (self.player.hasFallen()):
        self.player.initStance()
        self.player.setHeadAngle(0, "HeadPitch")

    # get value from camera 1
    RedBall = self.player.getRedBall()
    # if ball was found in camera 1, side-step to align with ball
    if (len(RedBall)>1):
        if (RedBall[0]<300):
            self.player.sidestep(0.2)
        if (RedBall[0]>400):
            self.player.sidestep(-0.2)
        if (RedBall[0]<=400 and RedBall[0]>=300):
            self.player.stop()
    # ball not found in camera 1
    else:
        # get value from camera 2
        RedBall = self.player.getRedBall2()
        #if ball was found in camera 2
        if (len(RedBall)>1):
            #if ball not aligned for kick, adjust i.e. re-position
            if (RedBall[0]<300):
                self.player.sidestep(0.2)
            if (RedBall[0]>400):
                self.player.sidestep(-0.2)
            # if ball is aligned as needed for kick, then proceed to kick
            if (RedBall[1]>470 and RedBall[0]<=400 and RedBall[0]>=300):
                self.player.rightKick(1.0, 1.4, 2.5)
            if (RedBall[0]<=400 and RedBall[0]>=300):
                self.player.stop()
        else:
            self.player.stop()

```

Figure 4.2.3

Alike our approach for testing the simple goalie, we began testing the modified goalie behavior by running the code on a robot in Webots and played with different ball positions such that we tested the new features added in the behavior tree. That is, we tested that the goalie differentiated between actions when the ball was in the lower frame versus the upper frame of vision. The test proved that the implementation was a success in that it followed what we desired to achieve for the modified goalie behavior.

We followed this test with many instances of running the goalie with an actor on the opposing team attempting to kick the ball into the goal. While we found that our behavior was satisfactory in blocking many of the attempted goal kicks, we discovered that there were some interesting flaws in our approach that might be

room for improvement for future developers working with this project. One flaw, that was a minimal concern to us, was that the goalie did eventually end up extending the bounds of the goalie box. Improving upon this might require further development in the vision such that the robot could detect the goalie box lines and make a decision to stay within those bounds.

Another flaw that might be more beneficial to improve upon in the more immediate future occurs with the decision to take no action when the ball is not in sight. This causes the goalie to take no attempt at blocking a kick that happens when the ball enters and leaves the goalie's line of sight too quick for the goalie to be able to react, which leaves a level of vulnerability for a speedy kick followed by an attack from either side such that the ball is still out of the range of vision of the goalie. We decided to proceed with development elsewhere as we saw the simple modified goalie was acceptable for our purposes.

4.3 Defender

When developing the defender, we started off by using the functionality of the goalie as a base. This included tracking the ball and sidestepping, as described in the behavior explained above. The main improvement we wanted to add to this was the ability for the defender to follow the ball in any direction in order for it defend, even if the offense has passed his position.

During developing this, we realized that some of the initial techniques we considered implementing had to be left for future development beyond the bounds of this project. One major ability that could contribute to a more competitive team is that of a defender having the ability to detect the bounds of the field and recognizing its whereabouts ie which side of the field it is on. Via testing of having a team with two strikers play as a team, as described in the next section, we found that if a defender is constantly following the ball on the field it did not create an optimal defending solution. A modified defender could anticipate the moves of the offensive player and position themselves in a way in which they can block the kick more effectively. We found that implementing of other behaviors over this one proved more practical for the purposes of our development. Thus, we proceeded with using a the simple sidestep approach defender.

4.4 Midfielder

Due to the defender already having a semi-optimal solution for blocking the ball from the goal, we decided to implement a chase feature to the midfielder. In order to do this without throwing off the vision of the robot as discussed in 4.3, we tracked the degrees in which the midfielder has turned in order to make sure it is always facing the proper direction. Due to the midfielder facing forward at the beginning of the game, all we have to do is make sure that they do not turn more than 90 degrees in either direction at any time.

The basic actions of the midfielder are to track the ball, turn in its direction and chase the ball down. As it approaches the ball, the robot slows down in order to avoid collision and attempts to kick the ball. Sense the midfielder is always facing forward, he will kick the ball forward in most cases. However, in rare cases, the offensive player could attempt to kick the ball as the midfielder did the same, sometimes causing the ball to go in unpredictable directions should both players succeed. Thus, we refrained from proceeding with this approach.

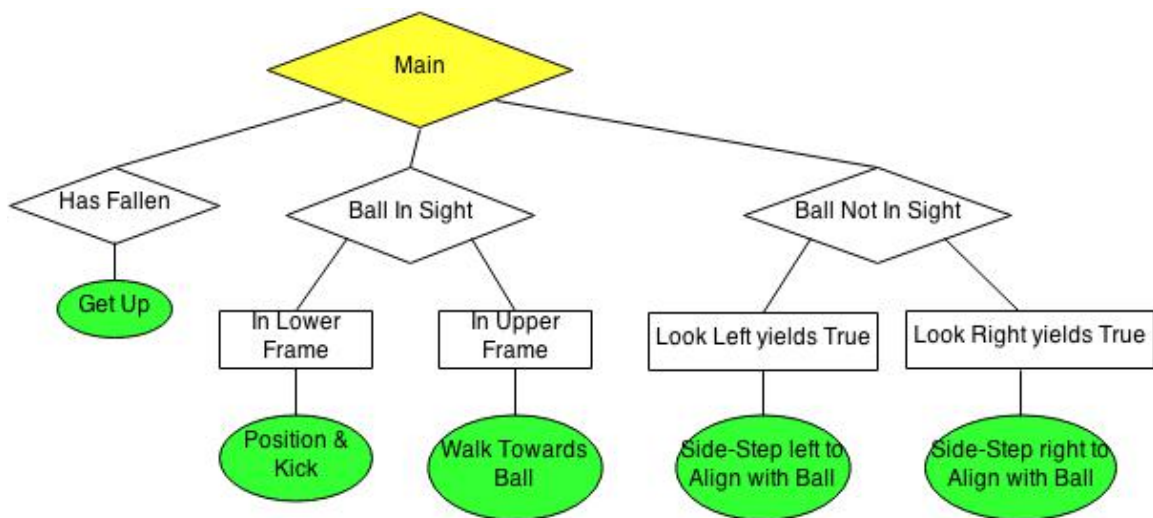


Figure 4.4.1

One area for potential future development should anyone decide to continue to work on this project would be a midfielder that would not only face and chase the ball in a forward motion, but one that could turn around in all directions in order to chase the offensive player. We believe, if done effectively, this could create a great team effort if mixed in with our current implementation of a defender. Our basic idea is for the

player to track the offensive player first instead of the ball. From there, it would chase the player down instead of the ball. This would work well in our case because there is only one offensive player on the field at the time to follow, thus opening up many prospective for implementing further behaviors such as passing. Once the midfielder got within a certain range of the ball, it would attempt to run in front of the ball in order to first block the balls path to the goal. The last thing the midfielder would do is try to kick the ball once it has reached a proper blocking position. This would also include turning to face the defensive player before kicking the ball.

4.5 Striker

The goals for having an all-around working offender was to have a behavior that would not interfere with other players' behavior and also not need assistance to carry out its own tasks. The logic behind the offensive player is based on the principle of having a way to get out of any un-ideal situation, in other words not getting stuck. The complexity of the offender should also be mature but simple. There should be no point in the code where the robot cannot get back out to an earlier level due to contradicting decision parameters. Therefore every level must operate smoothly to avoid any sort of cyclical dependencies. That being said the maturity should not only resolve trivial cases thrown at the robot but rather be dynamic and operate in a multitude of scenarios.

The figure below illustrates the logic given to the offender. Above all else the player is concerned with having the ball in view, and then having access to it. It is important to note this applies to the robot when it is in a standing position. The tree does account for falls every cycle.

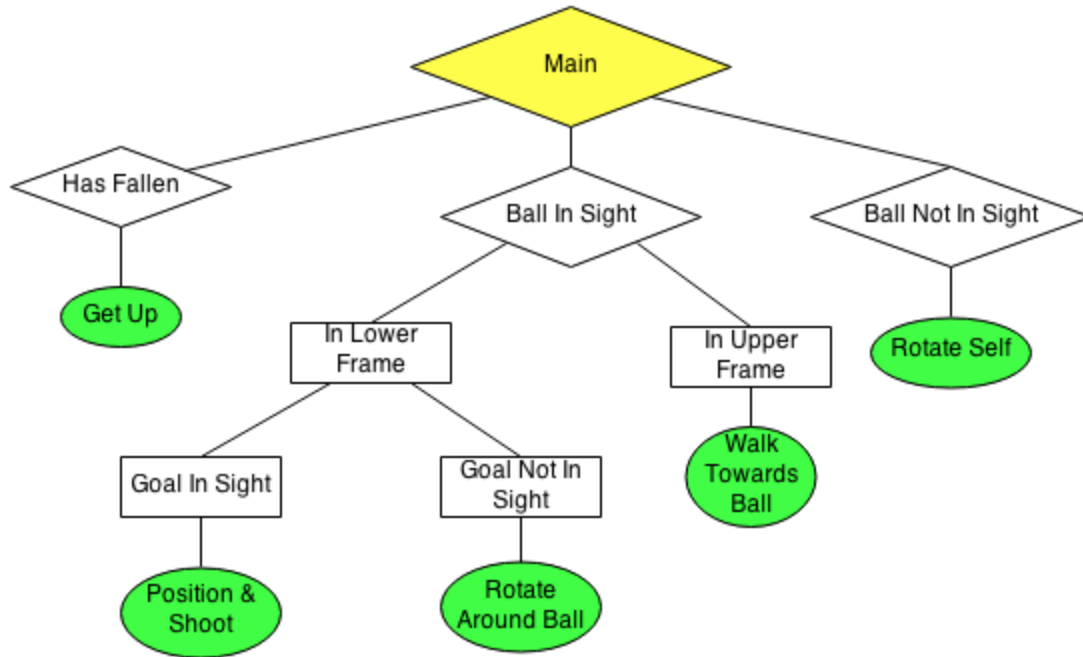


Figure 4.5.1

Starting from the top of the tree the robot must circle until it finds the ball. This rotation should only take seconds for a full scan of the field. If at any cycle the ball becomes present in the head or the torso camera the rotation is killed and the ball is either approached or the goal is located. For the former scenario the algorithm described in the walk section is used, with a new theta calculated each time. The robot will in this case be facing the ball head-on as it disappears from the upper frame into the lower one. During testing it was clear that if the ball was close enough to be in the lower frame the robot should start positioning to score or pass. This gives a metric as to how far away the robot is from the ball and the threshold is always constant.

With the ball in the lower frame the robot would start to position themselves to align with the goal. This motion keeps the ball in front of the robot until the goal comes into view. The goal must be within a range of the center of the screen. This range is quite wide since the goal itself accepts a spectrum of kicks. Another reason for not having a small window is so that the robot will carry out an action without having to re-position due to a slight miscalculation. A percentage of error is to be expected for the kick to the goal. When testing with smaller range a robot make take hundreds of cycles before it is confident enough to kick and score a goal. Although this previous method provides a more accurate end result, the chance of interception or other ball loss becomes increasingly higher.

After alignment the player will do a final approach to the ball. This approach utilizes the same walk algorithm as before but uses a slower walk speed to keep the robot steadier. Specifically the speed is 26% of the normal speed. During every cycle at this point the robot makes sure that the goal is still also in an acceptable range and then moves forward. This being so far down the behavior tree makes the behavior the most precise.

At the very end of this behavior tree the robot should be at the ball, ready to kick. If the ball coordinate returned is more than 460(of a total of 480) pixels in the Y directions from the top, then the robot is close enough to attempt a kick. This takes into account another aspect of the cycles. More than likely the robot will finish the last step from the previous cycle in which they were already moving, bringing the ball even closer to the foot of the player. The code calls a type of stop-movement that lets the previous call return to a steady state. This allows for the player to be stable by the time the next call is made. Otherwise the player may be on one foot, tilted, or even having fallen. After the kick the robot will once again chase the ball and complete the action again. There is also a chance that the ball gets just close enough, but also out of range, in this case the robot will of course start to look for it again.

The behavior class also provides auxiliary methods created specifically for the class. A method called `freePath()` determines whether the robot has a free path to the goal either now or sometime in the future. The method takes in a low and high Y value on the screen and the coordinates of any and all enemies. If any points lay within that range the method will alert the rest of the behavior tree that the robot needs to do a non-scoring kick.

A version of the offender code is available where the robot looks for blocked paths and calculates whether it should still attempt to score a goal or dribble the ball upfield for a later attempt. This version uses the `freePath()` method and while positioning to align with the goal it can tell if any enemy robot in the field will align to block the goal once the goal is directly in front of the player. The issues that arise with this code in execution are overlaps in ranges. An example of this being that the robot has a low and a high value to stay in-between when aligning to score, at the same time there is a range of values that correspond to freedom of kicking. certain combinations of these would cause a deadlock of decisions and neither can be picked. In simpler terms logic for a third decision does not exist but neither does an obtainable decision.

This behavior does not account for any saving of world states. For example if the robot does loses sight of the ball in the right edge of the frame he will still rotate left-as is default. By implementing a behavior tree rather than a finite state machine(FSM) the robot would not have a decent way of keeping track of other players and objects outside of the frame of vision.

Section 5: Design Challenges and Solutions

The first semester of Senior Design revolved around solving problems related to the red ball tracker and the kick animation. After those major roadblocks were overcome, much coding took place bringing its own many bugs with it. This section discusses the debugging that was done for the code.

After about a week of OpenCV usage, one of the robots started having trouble with its vision. The robot connected to port 9560 would not be able to detect the ball. Tracing the problem brought to light the fact that the code only ever expected to return one set of coordinates, since there is only one ball in the environment. It was however returning an array with values, which the function discarded to return a null value.

The array was unexpected, but as it was only happening to the robot facing the yellow goal it was then discovered that the red ball tracker we wrote in OpenCV was also detecting the yellow goal. Now that there were two objects being detected on the screen, the function applying the color thresholding and finding the coordinates of the centroids present in the image returns an array with two sets of coordinates. Previously the color had been set to orange to have a stronger differentiation between it and the pink belts that are worn by one of the teams.

This change in color resulted in both the yellow goal and the orange ball being picked up by the “red” ball tracker. It was still very early after starting to use OpenCV, so we were still very unfamiliar with it. The flaw finally came to light as a mistake in the saturation value for the thresholding of HSV files. Instead of setting it to 255 as it should be, we had simply plugged in the saturation value of the color we wanted to detect, which in this case was 235. It was close enough for the detection to work, but caused the edge case where the tracker failed us.

The next set of issues came from implementing concurrent execution in the software. When running it with more than one robot the different process created for them did not start with the exception of one. The logic that each process was supposed to run through included while loops that would keep the execution within the first process and never actually spawn the other processes.

To address this issue all the while loops were taken out of the logic meaning a robot would in one iteration of the process ask for its next logic step, take it, and

then exit. The loop that was running inside each process which kept going through each behaviour tree repeatedly was moved outside the process to encapsulate the commands spawning each processes.

After further familiarization with parallel processing in Python, all the processes in the software were rewritten to be threads. This caused a noticeable speed-up in the performance of the simulator. The Webots simulator functions in such a way that depending on workload real-time execution is slowed down, and previously the processes had brought Webot down to 0.30x its normal speed, now it was running at a manageable 0.70x.

Slowdowns in the simulation are problematic. When slowdowns occur, i.e. at 0.30x, the walking works less reliable, introducing a greater amount of wobble in the robots movement and increasing the times robots fall over significantly. This puts serious hardware constraints on anyone wanting to run the simulator with the official team size and create a 3 vs 3 robot soccer scenario. According to Aldebaran, the creator of NAO and its associated Webots variant, running the simulation with six robots will put a full load on an Intel i7 core. As we noticed even with an i7, a certain slowdown is unavoidable, with all of the six robots the simulation will not run at real time speed anymore.

One of the behaviors designed for the robots is a detection-act cycle of falling down and getting back up. To implement the fall detection, the memory of the robot is queried for values on the feet sensors. Each foot of the robot has eight sensors which give feedback on the pressure they register. When a robot lifts its leg during walking the respective foot's sensors read out values of zero for all four of them. The idea behind implementing the falling detecting was to check all sensors and when all of them would remain zero for the duration of the order of magnitude of a second, it would inform the method for fall detection that the robot has actually fallen. This did not work because when a robot fell, the values for its feet sensors did not in fact have a value of zero

The reason the fall detection did not work in such a way is that the NAO robots have an event queue handling certain events. One of the events is fall detection. Due to the already large complexity of this project implementing event listeners in the code was not one of the goals. Not having a fall detecting event listener lead to the decision in the first place to monitor the robots stability and position in space by means of the force resisting sensors at the bottom of its feet.

When the falling event is detected on the robot, movement is disabled. However the reading of sensors is also disabled. Therefore when a robot falls the feet sensors do not read zero as expected but remain at the value they were currently at. This realization led to checking the FSR periodically and if the value on all sensors had not changed it would mean that the robot had fallen. This implementation worked flawlessly.

One note on the FSRs is that there is a bug in Webots for NAO. The rear-left sensor on the right foot always returns a value of zero, no matter what state the robot is in. The value for the sensor was queried for several different robots with different ports assigned to them in different modes of movement. The result was identical leading to the conclusion that the bug is within Webots and not in our code.

After the falling detection worked it the robots with that code enabled would still not get up. Getting up is supposed to be an easy endeavor in Webots, calling the StandInit posture in the API will make any robot get up from the ground. Nonetheless the robots with falling detection would remain lying on the soccer field like real soccer players in the endzone. The solution was to send the robot any sort of movement command, even for just a millisecond. Afterwards the robot would be receptive to the posture command again and get up. Having fixed this bug our falling down detecting/getting up action loop was now complete and working correctly. Lastly there was an issue where the behaviour for falling and getting up would work, but when implemented on several robots it would fail in such a way that once players were down only one could get up during which every other players' movement was halted. It was caused by the posture command being a blocking call, which seems to be implemented in Webots in such a way that it blocks no matter which robot it originates from.

Blocking calls can be avoided by calling `.post` on the proxy first before calling the command. The program line finishes execution while the action is executed on the robot.

The issues that were tackled in the debugging phase lead to it being possible to implement working soccer teams. These teams are able to recover from failure leading to longer sequences of robot interaction and more interesting play. This is an improvement over the very initial team behaviors that always ended with every robot on the ground within the duration of a minute.

By overcoming the challenges that appeared and creating solutions that took the character of this project into account, the robots being programmed can rely on a good backbone of functions to realize good soccer play.

Section 6: Online Documentation

The second major requirement of this project, besides delivering robot soccer code for Webots, was to create resources for future students. For this purpose our sponsor Dr. Sukthankar created a Google Sites page called “ucfrobocup”, available at <https://sites.google.com/site/ucfrobocup/>.

It was handed off with the instruction to create documentation on all the relevant aspects of our project for newcomers to Webots, leaving design and organization of the content to us.

6.1 Simulation Setup

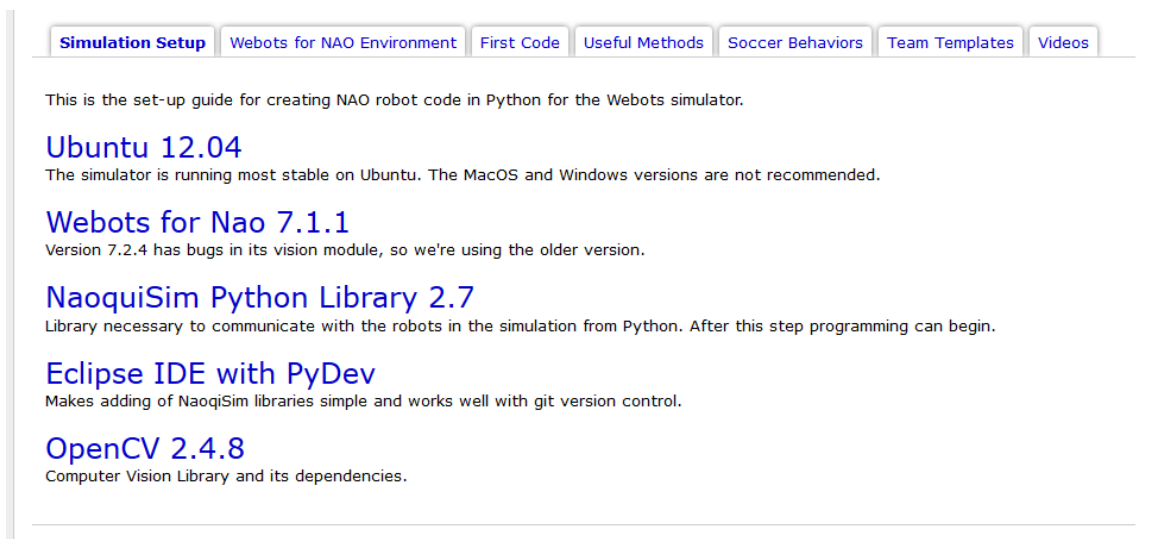


Figure 6.1.1

The website is using the wiki template that Google provides. It was edited to use a horizontal navigation bar instead of the default sidebar. The first tab is called Simulation Setup and covers the instructions for all the software needed to get a project running in Webots, with OpenCV being an extra requirement for using our software.

On this first page are links to the setup of Ubuntu, Webots for NAO, Naoqi Python library, Eclipse, Sublime, and OpenCV. Of those Eclipse and Sublime are optional, technically source code can be written in a text editor and run through the console, but are improvements to the development process that are nice to have. OpenCV is

optional for the early code examples given on this website, but are required later on when object tracking is used in actual soccer behaviors.

A lot of the recommended software is not the latest version. This is true for Ubuntu 12.04 and Webots. To a smaller extent Python could be regarded under this criteria as well, with the release of version 3.0 which is not compatible with 2.x versions, however Ubuntu 12.04 comes preloaded with version 2.7. So for most users, the version of python will not be a concern.

Ubuntu 12.04 is being used as it is the specific version that Webots was developed for. Going by the feedback on the forums that Aldebaran provides, it is still the most stable and bug-free platform to operate on.

Webots for NAO 7.1.1 is one release behind the newest version. The rationale for using this older version once again is performance based. For the first month of the project the video capturing of the robots never worked right which manifested itself in the functionality of the red ball tracker, which half the time did not initialize or return any valid values.

The installation process of all the aforementioned software is very straightforward, which aligns with our goals for a simple intro to Webots Robot Soccer. None of the packages require any knowledge of build tools, with the exception of OpenCV which uses CMake. The OpenCV section provides a simple script that can be run for the installation, eliminating the difficulty of set-up that would otherwise exist.

6.2 Webots for NAO Environment

Many issues with the Webots for NAO simulator can be avoided by using the right version of it. Our team learned this the hard way as we struggled with trackers and inconsistencies with the loading of code onto the robots. This is information that needs to be passed on to future participants in Robot Soccer research, to avoid the same mistakes happening again.

This page of the website details the process of setting up the actual simulation software. When a license is obtained for Webots, the licensee is emailed a registration code. It is not readily apparent that it cannot be used for activation in the software, but is instead for Aldebaran's website validation which in turn makes the users website account credentials the login for Webots.

In addition to the steps needed for loading up and validating the simulator, there is also general information on Webots that will help to inform future decisions. This includes such details as the fact that Webots can run a maximum of 7 robots. Also noteworthy is a bug in Webots that can leave processes active in memory, which leads brings the simulation to a halt after reloading the environment eight times. At that time those processes need to be manually terminated, for which a script is provided, and Webots needs to be rebooted. This is an unavoidable step and an added difficulty in working with the simulator.

A distinction that gets quickly lost within all the advice on the Aldebaran forums is between Webots Pro, Webots Edu, and Webots for NAO. Webots Pro is the main version of the Webots simulator, Webots Edu the discounted version for education purposes and Webots for NAO is an implementation of the simulator specific to the NAO robot. There are a few distinctions between them, specifically between the NAO version and the other two, limiting the scope of the former.

The Pro/Edu version allow users to customize robots or to add their own templates. The Pro/Edu version come with the source code of the Naoqi API allowing the users to edit and compile it for themselves. The Pro/Edu version allows the programming of supervisors.

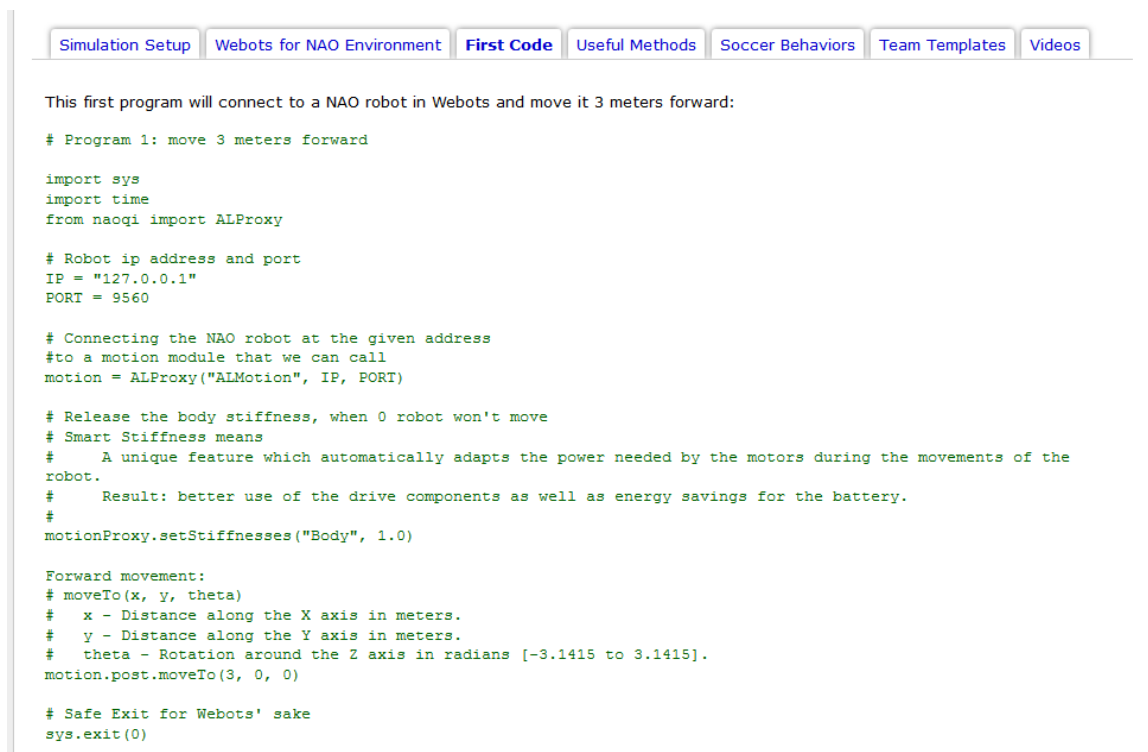
Webots for NAO has none of the above features. This lead to confusion during the early phase of this project since research on the forums often explained features that would have been useful but ended up not being available to us. Prominently on that list is the access to the Naoqi API source. Early outlines for the work to be done on the tracking abilities of the robots considered rewriting the API to extend the trackers' detection ability beyond the red ball.

Similarly when a better walk animation was sought, writing a genetic algorithm to train the robot was considered. Much of the documentation mentioned GPS positions of the robot that could be called. Eventually we realized that the GPS capabilities require a controller that can only be enabled through the supervisor, which is not a part of Webots for NAO

Webots for NAO is good software for robot simulation, but it is important to be aware of its limitations. This content on the UCF Robocup website accomplishes communicating exactly that.

6.3 First Code

Starting to code in a new environment can be a very daunting task, more so in such a unique field as robotics. To help new students make sense of the general structure of Webots Python code quickly and to form an understanding of basic methods without having spent hours poured over the documentation to extract that information first, we provide a section with a steps of coding baby steps anybody can take with access to the software.



```
Simulation Setup | Webots for NAO Environment | First Code | Useful Methods | Soccer Behaviors | Team Templates | Videos

This first program will connect to a NAO robot in Webots and move it 3 meters forward:

# Program 1: move 3 meters forward

import sys
import time
from naoqi import ALProxy

# Robot ip address and port
IP = "127.0.0.1"
PORT = 9560

# Connecting the NAO robot at the given address
#to a motion module that we can call
motion = ALProxy("ALMotion", IP, PORT)

# Release the body stiffness, when 0 robot won't move
# Smart Stiffness means
#   A unique feature which automatically adapts the power needed by the motors during the movements of the
#   robot.
#   Result: better use of the drive components as well as energy savings for the battery.
#
motionProxy.setStiffnesses("Body", 1.0)

Forward movement:
# moveTo(x, y, theta)
#   x - Distance along the X axis in meters.
#   y - Distance along the Y axis in meters.
#   theta - Rotation around the Z axis in radians [-3.1415 to 3.1415].
motion.post.moveTo(3, 0, 0)

# Safe Exit for Webots' sake
sys.exit(0)
```

Figure 6.3.1

Fittingly this section is called First Code. It contains three examples of very simple Webots code. There is an abundance of comments which explain the functionality of methods used in great detail.

The first example is of a program that makes a robot walk three meters forward. It shows the target audience how an ip address and port is specified to communicate with the robot. Afterwards a proxy object is created, which is a link to the robot. Calling its method will create change in the state of the robot it is linked to. Before

the code exits, the method `moveTo(3, 0, 0)` is called which makes the robot walk forward for three meters. Then the simple program terminates.

There are two other programs documented in this section; one has code for tracking the ball with the NAO's head, the other one for kicking a ball. These three programs will give users quick results in their first moments with the simulator and should give them the confidence to carry on with more challenging behaviors.

6.4 Useful Methods

The methods that ended up being used in this project make up a small subsection of all the methods available. One reason is similar functionalities, another the documentation of deprecated functions.

Finding which calls worked nicely and what did not was the main effort expended in the very early phase of the project. This accumulated knowledge presented on the website will enable future students to jump ahead on the learning curve. Getting ahead this way will lead to more complex code being developed sooner. Sharing this insight into the API that we gained was a good and obvious choice.

6.5 Soccer Behaviors

Whereas the First Code section had simplistic examples and the Useful Methods section was covering more advanced methods in an abstract way, this section brings both together to inform on the creation of Soccer Behaviors.

This means applying the right method to create stock behaviors that one could observe in actual soccer games. A few examples are presented in addition to a Github link that provides many more. The scope of this section is big as there are many different situation arising in a soccer that can be implemented as a stock behavior.

To give a few examples there are:

- a goalie tracking the ball and moving into position to block the goal
- a striker moving towards the ball and then kicking it
- a player circling a ball till he is in sight of the opposing team's goal
- a defender moving in between an opposing player and the ball
- a player a ball to a teammate

Currently the first three behaviors are available on the website. With these many team behaviors can be designed, which is the intention of this section. In such a way each example succeeds in becoming a building block for a team strategy.

6.6 Team Templates

This is where the previous code snippets come together to form a functioning team. We have access to soccer behaviors, now they just need to be tied together with a logical order of execution. The obvious, most accessible way is to use behavior trees or finite state machines. We have provided a framework to make their implementation very straightforward. Now such reactionary behaviors as “When you see the ball, chase it. If not, spin in circles till you do” can be constructed.

Here we provide more links to Github, as running our code allows for the realisation of the concept described and will potentially inspire students to create derivative teams based on these ideas. Some of the team behaviors already available will be dissected in detail to give students an insight into the thought process behind creating soccer playing robot teams.

By reaching this section and having build on the foundation of the previous ones, a student will be in almost the same position as Team Soccer Robots was at the end of the second semester of Senior Design. This opens up a lot of potential for future project, as our contribution means a jump start for anybody using the simulation, either in the context of a robotics class, another senior design project, or for personal edification.

6.7 Videos

The final section is meant as a visual aid. For someone who does not have all the required software available, glancing over the documentation will only have limited merit as the most reward lies in the application of all the described concept and the actual implementation of the code on the website.



Figure 6.7.1

Having videos featured on the site gives the cursory visitor a focal point to attach all their impressions. In addition it gives good visual feedback to teams implementing our code, as to what a successfully working team looks like and might inspire solutions during potential troubleshooting.

As the Senior Design fair showed, Videos tend to have the greatest effect on people when used effectively. We conclude that their inclusion will bolster the attractiveness of this website for anyone seeking information on robot soccer.

6.7 Website Evaluation

Through various iterations during the second half of Senior Design, the websites that was created to fulfill the secondary requirement of this project meet all

expectations. As a resource for future students it is a complete resource that provides a superior learning experience. It scales from easy content to complexity effortlessly and provides important information necessary to advance from one level of competency to the next.

The final meeting with the Sponsor verified that the website meet expectations and all requests regarding finetuning content were heeded accordingly. The UCF Robocup website is now a good teaching tool ready for use.

Section 7: Administrative Content

Administration is very limited in several ways. Fiscally speaking no additional funds needed to be added since the only software to be purchased was Webots. Consulting was done via Astrid Jackson every week and most other issues were resolved internally.

7.1 Finance

There are several expenses that our team considered during the initial planning phase of the project. The Nao robot is distributed by RobotsLab for \$15,999[13]. The purchase was deemed infeasible by the team, as well as the sponsor. Aside from the cost, another reason leading to this decision was the fact that our main focus is the development of software for the robots, which can be adequately achieved without the hardware.

Project Component	Cost
Nao Robot	\$15,999.00
Google Sites for Group Wiki and Student Wiki	Free
Google Docs	Free
Asana	Free
Instagantt	Free
Eclipse with PyDev	Free
Webots	Provided by Sponsor
Choreographe	Provided by Sponsor
Github	Free*
Bugzilla	Free
	*for open-source use

Figure 7.1.1: Project Financial Obligations

Next in the line of financial considerations was the attendance of a RoboCup competition or conference. The locations for RoboCup 2013 and RoboCup 2014 are Eindhoven, Netherlands, and João Pessoa, Brazil, respectively. However there was no discussion of financing necessary, as Robocup 2013 already took place on June 24th, 2013, and Robocup 2014 will take place after the project has concluded, namely on July 19th, 2014.

Regarding conferences, the NAO Tech Day would have been a consideration. There, the community around Aldebaran's robots comes together and various presentations are given. Unfortunately this conference has also passed, having been held on June 4th, 2013, with no date for further conferences set. Consequently it did not factor into our budget either.

The software that the team is working with, as discussed in the previous section, becomes the last consideration for our expenses. License keys for the main software, Webots and Choregraphe, have been provided by our sponsor, Dr. Gita Sukthankar. The other pieces of our development process - Asana, Instagantt, Google Sites, Bugzilla, Github, and Java - are all freely available to us.

Therefore this project can in essence be achieved with no financial support. However there is potential to acquire financing to attend a local conference, should one arise, from the Student Governing Association of the University of Central Florida.

7.2 Consultants

To aid in the development of this project the group had Dr. Gita Sukthankar as a sponsor as well as her graduate student Astrid Jackson. Both of these consultants provided excellent insight into project relevant material and helped push the project in the right direction every step of the way.

7.2.1 Dr. Gita Sukthankar

The project idea was proposed and is running under the sponsorship of UCF faculty member Dr. Gita Sukthankar. As director of the Intelligent Agents Lab and given her extensive experience with multi-agent and multi-robot systems, the team can rely on knowledgeable support throughout the whole process.

Dr. Sukthankar received her Ph.D. and Masters in Robotics from Carnegie Mellon University and joined The University of Central Florida in 2007. She specializes in Artificial Intelligence and machine learning.

7.2.2 Astrid Jackson

Also assisting the successful completion of our goals is Ph.D. student Astrid Jackson. Previously employed with EA Games, she has designed Artificial Intelligence in sports titles of the aforementioned game developer. Sport and Artificial Intelligence are the overarching focus of our project; her experience will not be amiss as an adviser to our project.

7.2.3 Dr. Mark Heinrich

Overseeing the Senior Design class is Dr. Mark Heinrich. Dr. Heinrich received his Ph.D. in Electrical Engineering from Stanford University under Dr. John Hennessy. Since then he has had a faculty position at Cornell University before coming to UCF.

Section 8: Conclusion

The project was originally anticipated to be a straightforward project with simple goals, i.e. making robots play soccer. Instead it turned out that robotics and simulations are rife with complexities and difficulties. Bugs and limitation in the software added another challenge on top.

Despite the scope increasing beyond what was initially imagined possible, the project in itself turned out well. A framework has been laid for future work at UCF in the field of simulation robotics and all the target objectives of this group in particular were met in the end: Teams of robots were playing soccer in the simulation and resources are available on the web.

With the completion of this project the group members will have learned moderate Artificial Intelligence programming techniques, and learned how to efficiently use robotics simulation software. Dr. Sukthankar will have the necessary documentation for her robotics class, and proper knowledge about the software being used. Not only is this an achievement for the team, but it is a project with potential framework for others, thus benefiting the university.

Tackling group work adequately was also a large part of this undertaking; bringing individual skills to strengthen the team. Teamwork lessons learned will benefit the group in their post-graduate careers, since most work is unavoidably group oriented.

Appendix A

A.1 Document Citations

Stone, Peter. Manuela Veloso. Patrick Riley. *The CMUnited-98 Championship Simulator Team*. Computer Science Dept. Carnegie Mellon University. Pittsburgh, PA. 1999. pp 1-16.

Thomas Teyvobia. *Optimizing Arm to Leg Coordination and Walking Speed in the NAO Humanoid Robot*. General Robotics, Automation, Sensing and Perception ([GRASP](#)) Lab. University of Pennsylvania.

Ueda, Ryuichi. Shota Kase. Yuji Hasegawa. *Team Description of Team ARAIBO 2007*. Dept. of Precision Engineering, School of Engineering. The University of Tokyo. Tokyo, Japan. 2007. pp 1-10.

Gamma, Erich, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

A.2 References

1. www.robocup.org/organization-of-robocup/
2. <http://www.robocup.org/about-robocup/objective/>
3. www.robocup.org/robocup-soccer/small-size
4. www.robocup.org/robocup-soccer/middle-size
5. www.robocup.org/robocup-soccer/humanoid
6. <http://www.cyberbotics.com/about> masthead for Cyberbotics.

7. http://ode-wiki.org/wiki/index.php?title=Main_Page Main page for ODE Wiki Site.
8. https://community.aldebaran-robotics.com/doc/1-14/software/webots/webots_index.html#what-it-is-not Aldebaran Webots documentation.
9. <http://osteele.com/images/2008/git-transport.png>
10. <http://www.cyberbotics.com/cdrom/common/doc/webots/guide/section1.2.html>
Cyberbotics preferences.
11. http://www.chrisoleary.com/projects/Soccer/Essays/FreeKickMechanics_DavidBeckham.html
12. <http://www.jssm.org/vol6/n2/1/v6n2-1pdf.pdf>
13. <http://www.robotslab.com/Robot/NAO%20H25>