

# **BURST Team Report**

## **RoboCup 2009 SPL**

Eran Polosetski<sup>1</sup>, Vladimir (Vova) Sadv<sup>1</sup>, Alon Levy<sup>1</sup>, Asaf Shilony<sup>1</sup>, Anat Sevet<sup>1</sup>, Elad Alon<sup>1</sup>, Meytal Traub<sup>1</sup>, Gal Kaminka<sup>1</sup>, and Eli Kolberg<sup>2</sup>

<sup>1</sup> Computer Science & <sup>2</sup> Computer Engineering Departments,  
Bar Ilan University, Israel

December 11, 2009

[galk@cs.biu.ac.il](mailto:galk@cs.biu.ac.il)

<http://shwarma.cs.biu.ac.il/robocup/>

# Table of Contents

Introduction.....	3
Software Architecture.....	3
Modules.....	3
Motion.....	5
Locomotion.....	5
Kicking engine.....	6
Vision.....	7
Color calibration.....	7
Strategy.....	9
Behaviors.....	9
Path Planning.....	9
Utilities.....	10
References.....	10

## Introduction

Team BURST (Bar-Ilan University Robotic Soccer Team [1]) participated in the RoboCup 2009 standard platform league. BURST is the first senior (non-Junior) RoboCup team from Israel, ever. The team is composed of two previously-independent RoboCup efforts at Bar Ilan University, lead by Kaminka (Computer Science department) and Kolberg (Computer Engineering department). Both team leaders have considerable experience in RoboCup (as participants, organizers, league chairs, and symposium co-chairs). All student members of the team have relevant experience in robotics.

We have a long history of publications, in and out of RoboCup forums, and in and out of robotic soccer (dozens of publications in the last three years, alone). We take RoboCup-based research very seriously, and have focused research goals that we hope to achieve by utilizing the robotic platform, and our participation in the league. Specifically, we are interested in: (i) anytime object-recognition (as a basis for anytime visual SLAM); (ii) using reinforcement learning to generate robust and speedy biped and quadped gaits; and (iii) decision-making architectures for humanoid robots.

## Software Architecture

Our software architecture is decomposed into components of actuation, vision, communication and high level behavior. They are implemented as follows:

- Actuation, Low level Sensors, Walking - We use the NaoQi client-server architecture (provided by Aldebaran) to access the lowest level modules and the actuation: namely, joint movement and walking. Most actual motions we use were generated by the Choregraphe tool.
- Vision – We use vision code created by Northern Bites.
- Communication - A C++ module running in the NaoQi process exports all relevant variables to shared memory (we needed the speed for out of process on the same robot communication), while another module records these results to a gzipped file. We use ALMemory/Shared Memory/sockets to communicate, using the best means for each option.
- High level behavior – We created a python main loop supporting FSM and event-based programming. This included keeping a world state, generating events and calling appropriate handlers (the actual behavior code).

Our architecture was based on the following ideas:

1. Python as a high level, easy to learn easy to produce working code quickly, language. Python is open source, very high level, mature and suitable for general purpose programming as well.
2. Don't use threads for high level behaviors to simplify debugging. Instead, use Deferreds.
3. Event based main loop. Any external event mapped to an event source (events trigger waiting deferred objects).

## Modules

The robot ran two executables, the Aldebaran provided NaoQi executable and our Burst python code. Aldebaran's NaoQi connected to the robot sensors/motors and to the camera. We added a module to NaoQi called imops that copied all the sensor data to shared memory, and ran the vision code.

In our burst executable we used an event based approach, in a single thread of execution. Multiple concurrent actions were enabled by a deferred based solution, based on the twisted python library.

In order to allow easy writing of sequential logic (do A, then B) we used a mechanism called Deferred. This allowed shorter code but was harder to debug than we expected and possibly hindered more than assisted for behavior composition.

We used a high level player object, which executed a FSM governing the various game states, and when appropriate called a main behavior which could call other behaviors. Each behavior was a Deferred object, and could as a result be composed by sequence pending events such as finishing a walk segment, seeing a specific object, or just passage of time.

For simulation we used Webots 6, with the Aldebaran supplied controller.

For an explanation of what a deferred is see the relevant page on the twisted wiki [4]. Programming style wise it means you can write code as following:

```
Kickball().onDone(onKickBallDone)
```

or without requiring another function definition, but losing on verbosity:

```
Kickball().onDone(lambda _: doSomethingHere)
```

Both styles result in the Kickball method doing something immediate (just computational) and returning – presumably it will initiate some action based on current perception/world state, and using either deferreds itself or another callback mechanism register a wait for completion of some underlying event (movement towards ball, kick itself), and return the Deferred object. The Deferred object has the onDone method, which itself returns immediately. The EventManager which is called by the MainLoop will eventually recognize the firing of the deferred, which will call the onKickBallDone function.

Our code base was subdivided into the following executables. The division was mainly a result of using Aldebaran as an access to the low-level sensor state and for sensor control:

- Burst – a python module, includes our main event loop running at 20Hz.
- imops – a module for Aldebaran's executable, includes two logical parts:
  - vision code – taken from Northern Bites. Recognized the location of the ball and goal posts. We did not use any other features due to lack of time.
  - shared memory copying code – the preferred information transfer to burst, including sensor data and vision output.

Burst (lib/burst python module) was further subdivided into the following submodules (there are many more modules in the code base, but we only describe those that were actually used, the rest were dead code as far as the competition is concerned):

- actions:
  - searcher – head turning and body turning to find a specific object, in practice one of goal, goal post or ball.
  - penalty\_kicking – specific behavior for penalty kick.
  - journey – any walk consisting of multiple straight and turn segments. We did not use arc (out of lack of time).
  - headtracker – head tracking behavior
- moves – moves such as kicking and leaping, and walk parameters.
- world – world state. Ball location, goal location.
- personal – change of constants per robot. Allows for fixes that are robot specific in one place instead of spread around the code.
- eventmanager – contains the MainLoop object, and the EventManager implementation.
- Players – The main behaviors used.

# Motion

## *Locomotion*

During the preparations to the contest, we examined different possible walking engines as candidates for our team locomotion control. Two main development paths were considered: creation of custom walk engine (using the underlying Aldebaran DCM command scheduler) and using hand-tuned standard NaoQi Motion module. Several problems with our custom engine and time limitations driven us to use the “budget” possibility of standard NaoQi module.

Our basic straight walk was a hand-tuned NaoQi Motion gait. The speed of our gait was 12 cm/sec while being remarkably stable on the carpet. There were two main problems with this gait: relatively inaccurate path (the robot seemed to be traveling in a curve after about 70cm) and lack of closing loop via video (since NaoQi walk command are queued in advance and we couldn't get the NaoQi engine to change direction/speed whilst walking)..

The reason for inaccuracy is by Nao design. The only joint in pelvis is located asymmetrically on one side and on the other side there is a plastic plate to compensate. This compensation, while probably being fine in some situations is clearly not enough for long straight walking on the carpet. The robot tends to walk in a curve. The strength of this tendency differs for different robots, carpets and, probably, battery states. We tried to overcome this issue in two ways: First, we created different parameter sets for each of our robots, utilizing asymmetric parameters exposed by NaoQi. The second approach was to limit a straight walk commands to 50-100 cm, the inaccuracies are much smaller for those distances. Of course those distance limitations made our overall motion slower, so the trade-off between desired accuracy and speed had to be considered.

Just before the competition, Aldebaran came out with “learning” mode for their walking engine. We evaluated this mode and concluded that the resolution for learned parameters deltas proposed by this method is too large to contribute anything to our gait. So, while we think that machine-learning methods are highly capable for gait design, still the specific method proposed by Aldebaran is not good enough just yet from our experience. We hope that next NaoQi versions will give a better answer on this subject.

Another motion pattern we used was a standard NaoQi side-stepping. It has same inaccuracy issues as the straight walk. We used side-stepping for the final robot placement near the ball. Those inaccuracies resulted a large amount of “thrashing” behavior by robot near the ball from time to time. To overcome this issue we had to lower desired position accuracy, which, in turn, affected our kick accuracy. Therefore we tried to minimize usage of this motion method as we could.

Turning was also based on the standard Aldebaran methods. Oddly enough, our “circle strafe” method discussed below gave better results in terms of speed than this basic turning. But the accuracy and resolution of possible angles it provided was not good enough, so we decided to utilize Aldebaran software for our turning motion.

The only motion type in use that was not based on NaoQi motion was the “circle strafing” walking pattern. The basic idea of this pattern is side-stepping around some point while preserving this point as robot heading destination. It proved itself as a very helpful behavior for bringing the robot into kicking position to the gate (in this case the central point was the ball). To create this motion pattern we used the Choreograph software from Aldebaran to record joint commands and than hand-tuned the motion patterns to preserve chosen circle diameter as well as stability on carpet.

Next year we plan to proceed with development of our custom walking controller, while examining

open-source walking controllers available (B-Human and others) as candidates for examples and starting points.

## ***Kicking engine***

In our kicker behavior, once the attacker found himself in close proximity to the ball it acted in one of two options: either it positioned itself against the center of the goal, or it stopped with some angle towards the goal. The first option enables the attacker to create a an accurate and strong kick with high probability of scoring, while the latter enables the attacker to act more quickly, with the trade-off of reducing the probability of scoring, due to missed or weak shots.

In order to maximize the kicking position, once the robot reaches the area of the ball, it uses side strafing, circle strafing, forward and backwards movements in order to position itself in the right distance from the ball with the right angle to the center of the goal. When done, it kicks the center of the ball. On the other hand, in order to act more quickly, once the attacker is in the area of the ball (but not in the right angle) it calculates the point on the ball, on which the attacker should kick in order to compensate for the angle (similar to billiards angle shots). In order to create such angle shots, we distance the kicking foot from the balancing foot, while shifting the center of mass accordingly and therefore, create more degrees of freedom.

Our basic straight kick was first constructed using the choreograph tool. We were interested in first getting the robot balanced, with the the kicking foot tilted backwards. This is important since you want the robot to be able to kick the ball even when pushed by the opponent. Secondly, we focused on creating a smooth (using smooth interpolation), yet very fast forward movement of the kicking foot without touching the ground, such that it will be extended as far as possible without loosing balance. This was helpful for building a strong kick that could be made even when the robot was a bit far from the ball. Thus, we were able to add a some length to our kicking area.

We later used the naojoints tool that we created in order to extend the width of the kicking area. We did so by creating a new kicking style, where the kicking foot was as far as possible from the balancing foot, without loosing balance. Then, we calculated the interpolation between the two kicks in order to create a parametrized kick for different positions along the x-axis, and by that extended the width of the kicking area. Note, that the latter kick was less balanced and weaker than the original kick and thus, the further the robot is from the ball on the x-axis, the weaker the shot will be along with less balance. Also, the kicking area was certainly not a rectangle, since the corners are too far away even for the parameterized kick. Therefore, we treated the kicking area as a half ellipse, where the two radii were the y-axis and half of the x-axis. However, this was just an approximation.

With a parameterized shot in our hands, all we needed to do is measure the angle between the front of the robot and the trajectory of the ball, while using different parameters when the robot is perfectly in front of the ball. We have taken extensive measurements and calculated averages using the robot's ball tracking ability to measure the angle between the robot and the ball, after it was kicked. This way, we had a parameterized kick aiming at the center of the goal even when the ball is not exactly between the robot and the goal.

## Vision

We used vision code courtesy of Northern Bites [2] with very few changes. Basically this means the usual colorspace → color-table, run length encoding, blobbing and object detection. Their implementation is a single module, which made it necessary for us to export the required variables through ALMemory / Shared memory.

We placed the vision code in a module called imops which was dynamically loaded by NaoQi at runtime. This allowed the vision code to run unaltered, since it ran as a module in the NB code base, and still let us run our high level code in a separate executable. The output from the NB module was passed through shared memory.

The NB Vision module is fully described in their team report, but an outline is provided here for completeness. The module provided existence of objects in the current frame and if so the location in image coordinates. The image was first converted from color space via a color table to a set of colors per pixel. The color table had to be calibrated per field during the competition, using a GUI tool written by NB for that purpose. The color table itself is a 128x128x128 8bit color index table, 2MiB in total. The next stage is blobbing and then object inference, filtering out too small objects to avoid noise. For goal posts the left/right certainty is computed as well.

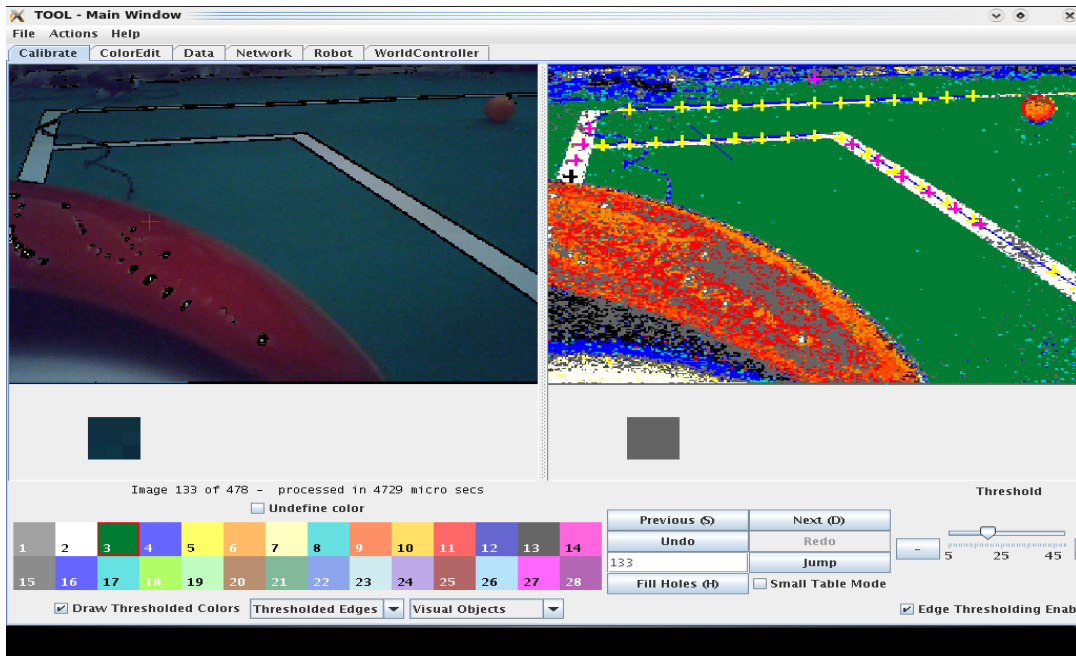
### ***Color calibration***

In order to calibrate the color-table we used a tool which was built by Northern Bites team (a.k.a the TOOL).

In the original TOOL, although having a palette of 28 colors, only 14 colors were available due to a minor bug. After fixing it and adding the option to work with a palette of 28 colors, we were able to get fine results from the calibration. Due to the use of 28 colors we were able to calibrate the pixels more carefully and therefore to distinguish between orange chairs / red robot shoulders / the ball. Another example is confusion between blue jeans and the blue gate.

We used approximately 3000 photos of the fields during the RoboCup tournament in order to calibrate our color-table. The TOOL requires manual marking of pixels, however once partially calibrated it shows the detected objects (ball, gates and field lines).

The main problem with this calibration process is that the marking of pixels is manual (very time consuming) and also that it is not very robust – when marking pixels in a new image it affects all other images (without the user knowing how exactly it changed). For example, if we marked pixels in 100 pictures and the detection was great, marking pixels in the 101 image can make the detection in the previous 100 pictures worse.



On the figure we can see the calibration TOOL and how using a larger palette of colors helped us to distinguish between the red robot shoulder and the ball. The ball is marked with an orange circle in order to show us that it was recognized by the calibration tool as the ball. The vertical and horizontal lines are marked with “+” while the horizontal and vertical lines are colored differently.

We wish to thank Northern Bites for publishing their code and allowing other teams to learn and use parts of it. Their vision routines worked great for us.

We wish the color-table calibration process could be more robust and automatic in the future and we intend to invest time to achieve this goal.

## Strategy

For RoboCup 2009 we didn't manage to create working localization. This obviously affected our high-level behaviors. Our players spent most of the time tracking the ball and searched for the opponent goal only when near the ball and intending to kick.

Our behavior engine was written in python especially for the RoboCup event using an event based programming model and a deferred mechanism (the later is based on our team's experience with [3,4]). This facilitates faster development as compared to a NaoQi module, and by using shared memory we achieve comparable speeds.

## Behaviors

In the actual RoboCup 2009 competition we used 2 roles only – a striker and a goalie. We worked on a 3<sup>rd</sup> role (a defender) but didn't finish it in time. Here is a high-level description of our 2 roles:

### 1. Striker

- Search ball (including turning the robot around)
- Constantly head-track the ball while moving towards it (ignoring our position in the environment, but using sonars to avoid obstacles)
- When close to ball look for opponent goal and circle-strafe around the ball till aligned with goal
- Kick the ball towards the goal (either using a straight kick or an angled kick)

### 2. Goalie

- Search ball (without turning the robot body around)
- Constantly head-track the ball
- If the ball is moving towards the goalie line and its speed is high enough to risk our goal - leap. Once on the ground, get up and try to return to the middle of the goal (using the distance from the goal post, as we don't have localization). During the competition we changed this behavior since the goalie took a long time to return to mid-goal, didn't return accurately enough and didn't react to the ball position while returning. So instead of returning to the mid-goal, we changed the goalie role to a striker role.

For the penalty kick we altered these two roles to be more suitable for the time-limited penalty kick. The change to the goalie role was to increase its sensitivity to ball location changes, to allow fast enough response to fast penalty kicks. The change to the striker role was to disable the circle-strafing routine, since it took too much time and the robot was already supposed to be quite aligned towards the goal.

## Path Planning

A considerable effort has been made on studying motion path planning. Given starting coordinates (X,Y,Yaw), target coordinates (X,Y,Yaw) and set of all the gaits defined for the robot, we built a motion planner that proposes a combination of walking commands that minimizes total travel time. This project was performed by group of outstanding undergraduate students.

## Utilities

A number of utilities were developed to ease development of the behavior code, and to allow easy experimenting with snippets of code on simulated or real robots. These include:

1. A standalone implementation of the soap based protocol used for out of process communication with NaoQi.
2. An ipython based shell for issuing any of our commands / behaviors.
3. Same shell was enhanced to include utilities to graph specific variables or periodically run any bits of code, visualization of localization data and of vision algorithms.

## References

1. Team BURST Homepage, <http://shwarma.cs.biu.ac.il/robocup/>
2. Northern Bites Homepage, <http://robocup.bowdoin.edu/blog/>
3. Twisted Matrix, <http://twistedmatrix.com/trac/>
4. Twisted Deferreds, <http://twistedmatrix.com/documents/current/core/howto/defer.html>