The University of Pennsylvania Robocup 2013 SPL Nao Soccer Team

Alan Aquino, Christopher Akatsuka, Tatenda Mushonga,
Yida Zhang, Yizheng He, and
Daniel D. Lee

General Robotics Automation, Sensing and Perception (GRASP) Laboratory
University of Pennsylvania, Philadelphia, PA 19104

Abstract. This paper presents the organization and architecture of a team of soccer-playing Nao robots developed at the Univ. of Pennsylvania for the 2012 Robocup competition. This effort represents completely custom software architecture developed from scratch. All sensory and motor functions are prototyped and run in Lua on the embedded on-board processors. High-level behaviors and team coordination are implemented using state machines. The locomotion engine allows for omni-directional motions and uses sensory feedback to compensate for external disturbances. An echolocation system helps robots to localize based on perceived goalie position even with two yellow goals.

1 Introduction

Robocup 2012 represents the continuation of a long-standing tradition of robotics for the UPennalizers of the Univ. of Pennsylvania. The UPennalizers participated in the Robocup Sony Aibo league and made the quarter-finals each year starting in 1999. After a brief hiatus from 2007 through 2008 , the UPennalizers began competing in Robocup once more in 2009. The UPennalizers swiftly adapted the existing code base to the Aldebaran Nao platform, simultaneously improving and expanding the base set of sensory and motor skills [1]. Having translated our code base from Matlab to Lua, a scripting language which wraps C, before Robocup 2011, the UPennalizers continued to test this restructured code base at Mexico 2012. One notable problem which the 2012 competition presented was the issue of two yellow goals. The solution presented by the Upennalizers utilizes echolocation through sound emmitted by the goalie.

2 Software Architecture

The software architecture for the robots is shown in Figure 1. The architecture used is an expansion of the architecture used by the UPennalizers in the previous Robocup competition. This architecture uses Lua as a common development platform.

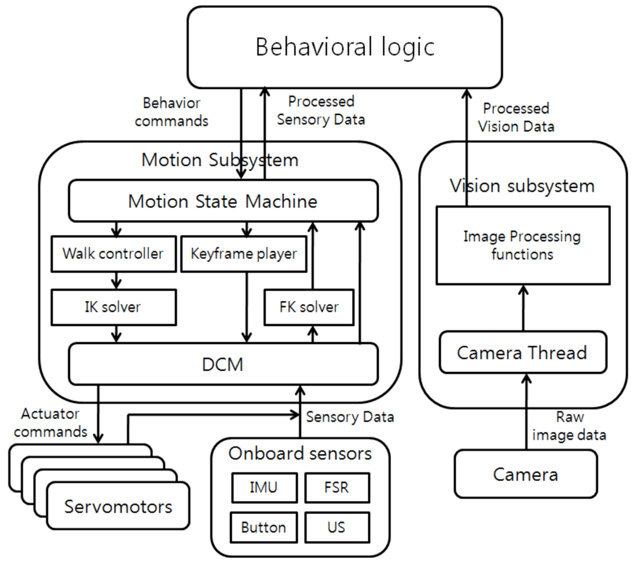

Fig. 1. Block Diagram of the Software Architecture.

Low-level interfaces to the hardware level (via NaoQi or our own custom controllers) are implemented as compiled C libraries that can be called from Lua scripts. The routines provide access to the camera and other sensors, such as joint encoders and the IMU. They also provide interfaces to modify the joint angles, stiffnesses and LED's.

The system is comprised of two main modules. One Lua process contains the main behavioral control of the robot along with the motion control through NaoQi. The second process contains the image processing pipeline. Separating out time consuming vision processing allows us to maintain an update rate of 100 Hz for motion control and to run image processing with the remaining processing power. This decoupling of motion and vision processing allows our robots to maintain more stability and robustness when walking.

Shared memory is used as the main form of interprocess communication. Any important control and debugging information is stored as shared memory which is accessible from any module. This also allows for real-time, on demand debugging and monitoring without any change or impact on the system. Debugging can be done locally or remotely, and the information is accessible both through Lua and Matlab interfaces.

The Lua routines consist of a variety of modules, layered hierarchically:

Body Responsible for reading joint encoders, IMU data, foot sensors, battery status, and button presses on the robot. It is also responsible for setting motor joints and parameters, as well as the body and face LED's.

Camera Interface to the video camera system, including setting parameters, switching cameras, and reading raw YUYV images.

Vision Uses acquired camera images to deduce presence and relative location of the ball, goal, field lines, and other robots.

World Models world state of the robot, including pose and filtered ball location.

Game StateMch Game state machine to respond to Robocup game controller and referee button pushes.

Head StateMch Head state machine to implement ball tracking, searching, and look-around behaviors.

Body StateMch Body state machine to switch between chasing, ball approach, dribbling, and kicking behaviors.

Keyframe Keyframe motion generator used for scripted motions such as getup and static kick motions.

Walk Omni-directional locomotion module.

Sound An echolocation system allows players to relocalize based on the perceived position of the goalie, which emits a pseudo-random sound.

3 Vision

Our algorithms used for processing visual information are similar to those used by other Robocup teams in the past. Since fast vision is crucial to the robots' behaviors, these algorithms are implemented using a small number of compiled C++ routines.

The main processing pipeline segments the highest-resolution color images from the camera, forms connected regions, and recognizes objects from the statistics of the colored regions. The color segmentation routine classifies individual pixels in the image based upon their YCbCr values. Based upon a number of training images, a Gaussian mixture model is used to segment the YCbCr color cube into the following colors:

- Orange (Ball)
- Yellow (Goal)
- Green (Field)
- White (Lines)

Once pixels in the image are classified according to their colors, they are segmented into either connected components or edge regions. After the image is segmented, the regions are classified into relevant objects by comparing various image statistics of the regions. These statistics include the bounding box of the region, the centroid location, and the chord lengths in the region. In this manner, the location of the ball and goal posts are detected.

Field line recognition decreases the need for robots to actively search for landmarks, enabling them to chase the ball more effectively. The first step in our line identification method is to find white pixels that neighbor pixels of "field green" color. Once these pixels are located, a Hough transform is used to search for relevant line directions.

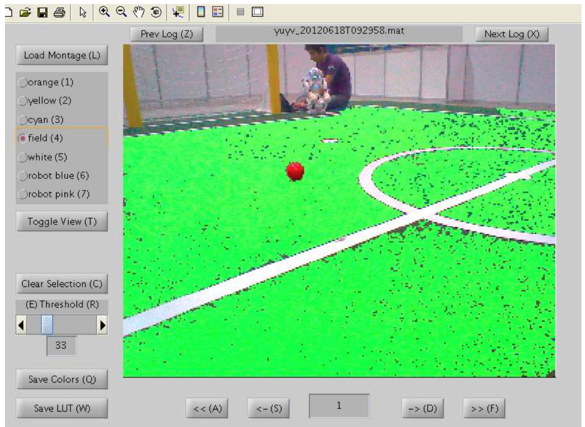

Fig. 2. Graphical Interface for Color Segmentation

In the Hough transform, each possible line pixel (x, y) in the image is transformed into a discrete set of points (θ_i, r_i) which satisfy:

$$x \cos \theta_i + y \sin \theta = r_i \tag{1}$$

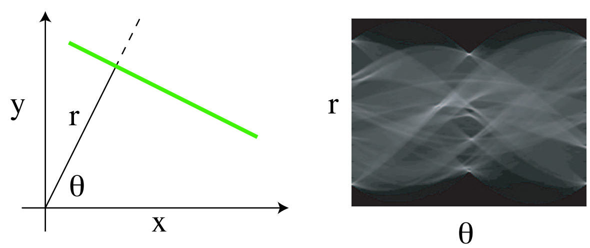

Fig. 3. Hough transformation for field line detection in images.

The pairs (θ_i, r_i) are accumulated in a matrix structure where lines appear as large values as shown in Figure 3. To speed the search for relevant lines, our implementation only considers possible line directions that are either parallel or perpendicular to the maximal value of the accumulator array. Once these lines are located, they are identified as either interior or exterior field lines based upon their position, then used to aid in localization.

3.1 Debugging

a, Monitor

To debug the vision code, we developed a tool to receive image packets from an active robot and display them. To this end, we broadcast YUYV images, as well as two 'labeled' images. The YUYV images represent what a robot is literally seeing at any given time, and the labeled images depict what the robot thinks it is seeing at that same time. We programmed a GUI in Matlab which receives these packets, reconstruct them, and then displays them for the user to see. Through the use of this debugging tool, it is possible for us to test and improve our color look up tables with ease.

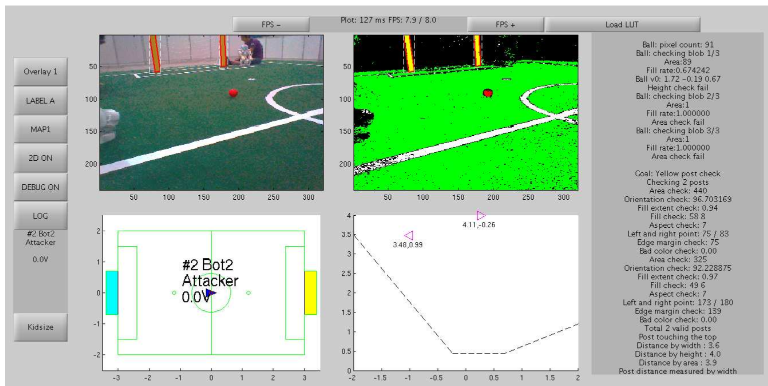

Fig. 4. Monitoring software developed to debug vision.

b, Camera Parameter Setting

Since vision depends highly on the quality of pictures from the camera, setting camera parameters (i.e. Exposure, Contrast, and Saturation) properly is crucial to the developing and debugging of vision code. To get better images and change parameters easily, the camera driver was modified and a lua script developed. The new camera driver uses Linux mmap functions to store image data. This enables the vision system to sample pictures at 30 frames per second from both cameras, where the unmodified camera driver samples at around 25 fps). This greatly improved the picture quality in motion. The Lua script supports camera parameter switching while NaoQi is running on the robot, and it broadcasts images from both cameras so that the impact of parameter switching on vision code can directly be seen on a local machine through the use of debugging tools.

c, Logging and Camera Simulator

Logging information allows the user to log vision data without affecting the currently running system in any way. These vision data are usually taken when the robot is running in a real competition environment and are thus of debugging value. We used MATLAB as our main logging program. The data we record includes:

– Time Stamp
– Joint Angles
– IMU Data
– YUYV Image
– Currently Selected Camera

All information is stored in log files and can be displayed on the monitor.

To better test our vision code, we developed the camera simulator in MATLAB. Instead of getting images from the robot, the simulator takes images from previous logs (generated by the logging tool) and pushes these data into the shared memory. Image processing codes can then run based on the logged images. This tool enables the user to debug the vision code without the use of a robot.

4 Localization

The problem of knowing the location of robots on the field is handled by a probabilistic model incorporating information from visual landmarks such as goals and lines, as well as odometry information from the effectors. Recently, probabilistic models for pose estimation such as extended Kalman filters, grid-based Markov models, and Monte Carlo particle filters have been successfully implemented. Unfortunately, complex probabilistic models can be difficult to implement in real-time due to a lack of processing power on board the robots. We address this issue with a new pose estimation algorithm that incorporates a hybrid Rao-Blackwellized representation that reduces computational time, while still providing for a high level of accuracy. Our algorithm models the pose uncertainty as a distribution over a *discrete* set of heading angles and *continuous* translational coordinates. The distribution over poses (x, y, θ), where (x, y) are the two-dimensional translational coordinates of the robot on the field, and θ is the heading angle, is first generically decomposed into the product:

$$P(x, y, \theta) = P(\theta)P(x, y|\theta) = \sum_i P(\theta_i)P(x, y|\theta_i) \tag{2}$$

We model the distribution $P(\theta)$ as a discrete set of weighted samples $\{\theta_i\}$, and the conditional likelihood $P(x, y|\theta)$ as simple two-dimensional Gaussians. This approach has the advantage of combining discrete Markov updates for the heading angle with Kalman filter updates for translational degrees of freedom.

When this algorithm is implemented on the robots, they quickly incorporate visual landmarks and motion information to consistently estimate both the heading angle and translational coordinations on the field as shown in Fig. 5.

4.1 Sound

With the addition of the "two yellow goals" rule, relocalization after kidnap becomes more complicated. To address this problem, we make use of an echolocation system through sound emitted by our goalie. Through analyzing the arrival

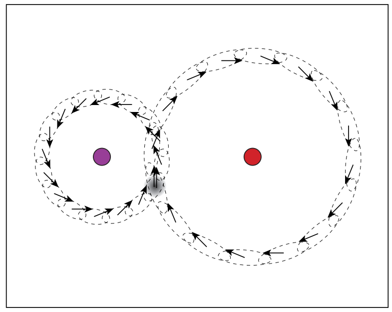

Fig. 5. Rao-Blackwellized probabilistic representation used for localization.

times in each of the receiver's microphones, the position of the goalie relative to the receiver can be determined. This information can then be fed to our filters to help the player relocalize to the correct side of the field.

5 Motion

Motion is controlled by a dynamic walk module in addition to predetermined scripted motions. One main development has been a bipedal walk engine that allows fast, omni-directional motions.

The walk engine generates trajectories for the robot's center of mass (COM) based upon desired translational and rotational velocity settings. The module then computes optimal foot placement given this desired body motion. Inverse kinematics are used to generate joint trajectories so that the zero moment point (ZMP) is over the support foot during the step. This process is repeated to generate alternate support and swing phases for both legs.

IMU feedback is used to modulate the commanded joint angles and phase of the gait cycle to provide for further stability during locomotion. In this way, minor disturbances such as carpet imperfections and bumping into obstacles do not cause the robot to fall over.

Depending on the surface and robot, a number of parameters need to be tuned. These include the body and step height, ZMP time constant, joint stiffnesses during various phases of the gait, and gyroscopic feedbacks. Through tuning these parameters to a new surface, the motion engine can adapt to various environs.

This past year, mid-gait parameter switching was introduced into the motion engine. Through this addition, we are able to make use of several sets of parameters which are honed to a specific purpose. During the 2012 competition, this functionality was used to allow for a different set of parameters for orbit. In this way, we were able to have large steps for our orbit state while still having quick, light steps for standard movement.

5.1 Kicks

Our kick engine makes use of gyroscopic stabilization to maintain balance over the course of a powerful kick. Each kick can now be tuned from a single configuration file, allowing adjustment of the torso, leg, and foot positions, as well as the angle through which the foot swings.

New for the 2012 competition, we introduced a walking kick to be used as a dribble. Our walk kick engine, upon receiving a ”walk kick request,” will plan the following two steps such that the first foot will allow the robot to have better balance while the second foot kicks forward slightly. In this way, we are able to kick the ball out of the path of opposing attackers, and dribble the ball out of the center circle during kick-off.

6 Obstacle Detection

Through reading data from the Ultrasound sensors mounted on the chest of Aldebaran Nao humanoids, we can make informed decisions as to the presence of an obstacle in the robot's path. The construction of the Nao platform incorporates two ultrasound sensors on the humanoid's chest; one angled to the robot's left and one angled to the robot's right. Depending on which of these sensors has been triggered, our logic can infer the location of the obstruction: To the left, to the right, or in the center of the path. From this information, it is possible to command the robot to stop when it is approaching an object, and even to cause it to navigate around the obstacle.

For the 2012 competition, we made use of this information to stop players before collision with other players. Smaller tolerances were used for the attacking role than for the other roles, and no stopping logic was implemented on the goalie. This information is also used to force a walk kick when an obstacle is very close to a robot pursuing the ball. In this manner, we provided less opportunity for opponents to gain control of the ball.

7 Behaviors

Behaviors are controlled by finite state machines which are updated at 100Hz. In our state machine implementation, each state in a state machine contains an entry function, an exit function, and a body. The entry function specifies any actions which need to be taken when the finite state machine enters a particular state; for example, turning the head if it enters the $headScan$ state. The exit specifies any actions which need to be taken on exit from a particular state, such as planting both feet on the ground when exiting the $bodyWalk$ state. The body of the state contains anything which needs to be updated and any decisions which need to be made within a particular state. The body of a state is where the state machine queries the environment to determine if the state of the field or of the robot has changed.

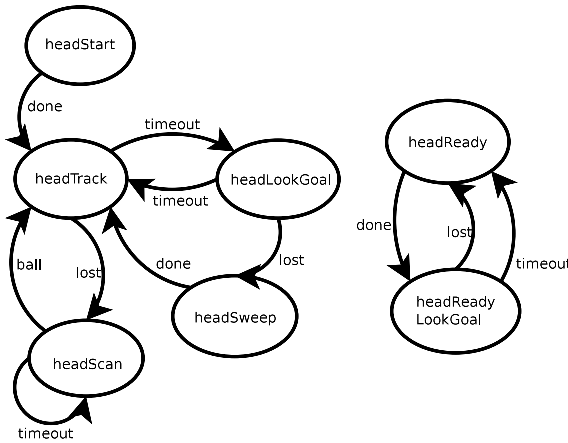

Fig. 6. Head Statemachine

The head state machine is simple: Either the head is looking for the ball, looking at the ball, or finding the goal posts. In example, if the head is tracking the ball and loses it, it throws a *ballLost* event and transitions to the *headScan* state, wherein the head begins to scan the field for the ball.

The body state machine is more complex than the head state machine, as the body has more degrees of freedom than does the head. The main state is the *bodyPosition* state which keeps track of the estimated position of the robot, the ball, and the goals. During the *READY* state, the body state machine transitions to the *BodyGotoCenter* state, which commands the robot to return to its 'home' position. During game play, the body state machine transitions between the *BodySearch* state, which causes the robot to search for the ball if it has been lost; the *BodyOrbit* state, which causes the robot to maneuver around the ball until it is in a good position to score a goal; the *BodyApproach* state, which causes the robot to get close enough to the ball to kick it effectively; the *BodyKick* state, which executes the dynamic kick engine; and the *BodyPosition* state, as described earlier.

7.1 Changing Behaviors

The code base is structured hierarchically, with the state machine definition residing within the `BodySM` and `HeadSM` files. This is where the actual states and state transitions are defined. To change a transition, all that needs to be done is to change the transition line in the state machine file. Each state resides in its own file, making the changing of states simple.

7.2 Team Play

To make full use of a team of four robots, it is imperative to assign each robot a role. We do this through the development of 'team play' code, which functions

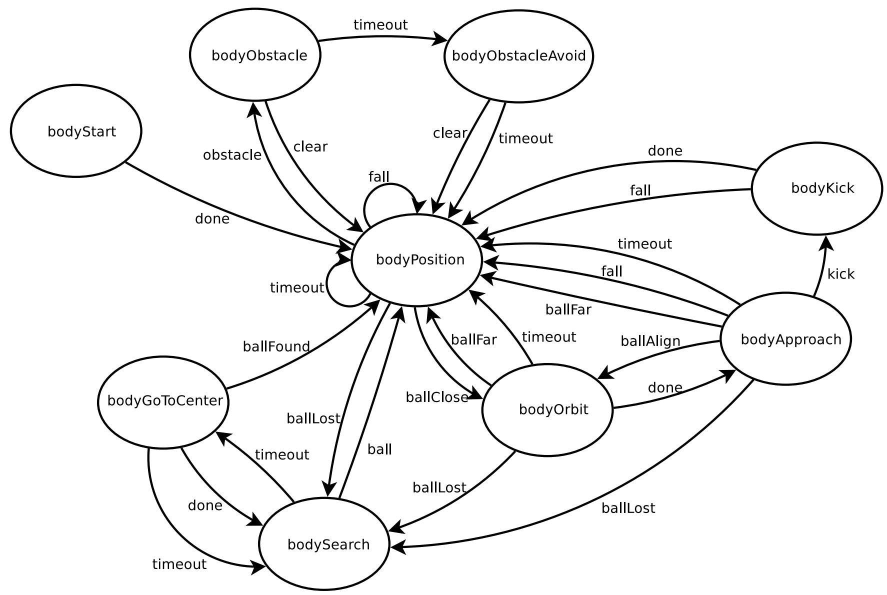

Fig. 7. Body Statemachine

alongside our localization code. The different roles each have specific home positions on the field, and each role causes the robot to behave differently. The roles we have defined for our team are as follows:

Attack The attacking robot goes directly to the ball, as long as it wasn't in the defensive penalty box.

Defend The defending robot positions itself between the ball and defensive goal area.

Support The supporting robot follows the attacking robot upfield, but stays at a respectable distance away–usually about midfield.

Goalie The goalie stays near the defensive goal to clear the ball when it comes close.

Because the primary function of a soccer player is to move the ball and attempt to score, we make that our top priority in our strategy. Therefore, our robot 'team members' communicate with each other as to each's relative proximity to the ball. If our supporter is nearer the ball than our attacker, they will switch roles; the supporter will behave like the attacker, and the attacker will behave like the supporter. This holds true if the ball becomes nearer to the defender than any other player; the defender will 'switch roles' with the attacker, and try to move the ball down field. In this way, team members can reach the ball much faster if they behave as a unit instead of as individuals.

8 Summary

The UPennalizers kept to previous standards of reaching the Quarter-Finals at Robocup 2012, Mexico City. Through switching to a Lua-based code base, we were able to keep our update speeds to 100Hz, stabilizing our motion and allowing us to move faster than ever before. The addition of sound-based localization allwed us to stay localized even under the confusing environment of two yellow goals, and the addition of a walk kick and parameter switching kept motion quick and competitive.

Our code has recently been re-released under the GNU public license, and we hope it will be of use to future teams.

References

1. Aldebaran Robotics. `http://www.aldebaran-robotics.com`.
2. Lua. `http://www.lua.org`.
3. MATLAB. `http://www.mathworks.com`.