UPennalizers
Robocup 2013 Standard Platform League
Team Description Paper

Christopher Akatsuka, Alan Aquino, Sarah Dean,
Yizheng He, Tatenda Mushonga, Yida Zhang, and
Dr. Daniel Lee

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

Abstract

This paper presents the organization and architecture of a team of soccer-playing Nao robots developed by University of Pennsylvania's Robocup SPL team. It also documents the efforts gone into improving the code base for the 2013 competitive season. All sensory and motor functions are prototyped and run in Lua on the embedded on board processors. High-level behaviors and team coordination modules are implemented by Lua using state machines. The locomotion engine allows for omni-directional motions and uses sensory feedback to compensate for external disturbances. The cognition module helps robot to detect landmarks and localize in a symmetric environment. Through the year, improvements were made across all of the various modules.

1 Introduction

In 1999, two years after the first international Robocup meet, the University of Pennsylvania formed the UPennalizers autonomous robot soccer group and began stepping up to the challenges put forth by the competition. While the league was still utilizing four-legged Sony Aibos, the UPennalizers made the quarterfinal rounds every year through 2006 before taking a brief two-year hiatus in 2007. The team reformed and returned in 2009 to begin competing in the Standard Platform League with Aldebaran Naos, taking on bipedal motion alongside improved vision techniques and advanced team behaviors.

Continuing its streak of making the international quarterfinals through 2012, the UPennalizers were challenged in 2013 with training a team of entirely new undergraduates without its seasoned veterans from previous years. The decision was also made to merge code bases with the University's graduate Robocup group, Team DARwIn, in order to share robot behaviors, knowledge between team members, and operating skills between both leagues. The new team went on to take first place at the 2013 US Open, and took the Consolation Cup at Robocup 2013 Eindhoven, finishing Rank 11 of 22.

2 Software Architecture

A high-level description of the software architecture for the Naos is shown in Figure 1. The current architecture is an expansion upon the previous years work. It uses Lua as a common development platform to interface between all modules.

Low-level interactions with hardware are implemented using compiled C libraries in conjunction with the Nao's on-board hardware controller (NaoQi) or custom controllers. These in turn, are called via Lua scripts, and allow for control over motor positions, motor stiffnesses, and LED's. Sensory feedback is also handled similarly, allowing users to get data from a variety of sources such as the Nao's two on-board cameras, foot weight sensors, the inertial measurement unit (IMU), and the ultrasound microphones.

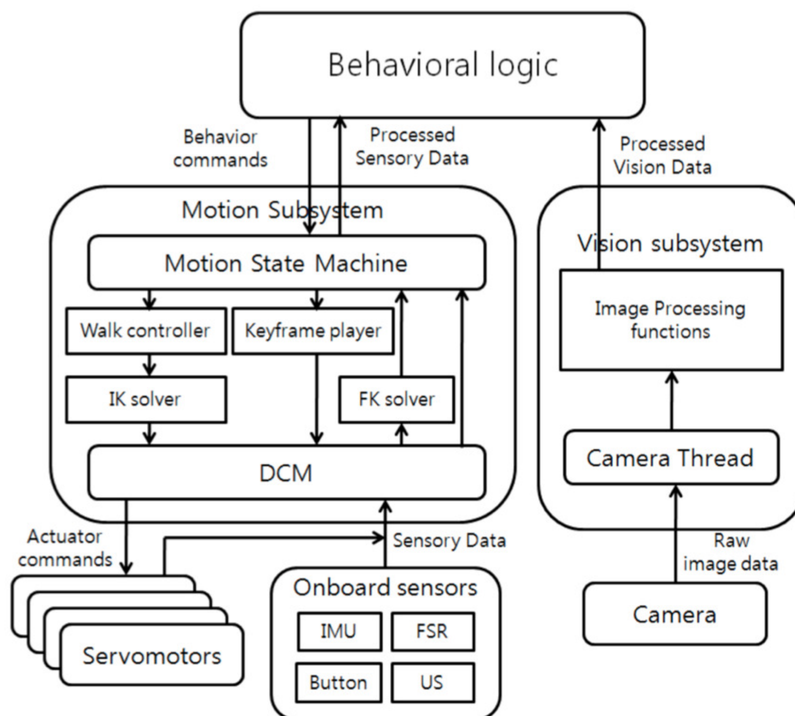


Fig. 1. Block Diagram of the Software Architecture.

The system maintains a constant update speed of 100Hz, and is decoupled into two separate pipelines. The main process handles motion control and behavior control, while an auxiliary process is dedicated solely to cognition processing. This decision, made last season, allows for more efficient handling of the Nao's on-board single-core Intel Atom Z530 clocked at 1.6 GHz. The cognition engine runs off of the remaining processing power not used by the main modules, and as a result, the Naos were noted to be much more stable and robust than in previous years.

Inter-process communication is accomplished via shared memory. Important information such as ball distance, position on the field, and game state are examples of shared memory variables. Any module can write and read to shared memory. In addition, any operator connected to a Nao via secure shell can monitor the data stored in the shared memory module without any change or impact on the running system, allowing for real-time on-the-fly debugging and analysis through either Lua or MATLAB.

A variety of software is used to run our Naos. We utilize Lua 5.1.15 and LuaJIT 2.0.1 for high level interactions, MATLAB R2013a for debugging vision and localization information, C/C++ to run low-level processes, and Webots 7.1.2 for simulation purposes.

2.1 Software Modules

The main modules accessed by our Lua routines are as follows, layered hierarchically:

Camera Direct interface to the cameras located in the forehead (upper) and mouth (lower); controls switching frequency and bundling of images in YUYV format.

Vision Interprets incoming images; based on the user-created color table and camera parameters, the module passes on information relating to the presence and relatively location of key objects such as the ball, defending goal posts, attacking goal posts, field lines, field corners, and other robots.

World Models the robot's state on the field, including pose and filtered ball position;

Body Handles physical sensory information and functions; reads joint encoders, IMU data, foot weight sensors, battery voltage, and chest button presses, but can also set motor positions, stiffnesses, and LED's.

Motion Dictates general movements on the Nao; i.e. sitting, standing, diving

Walk Controls omni-directional locomotion; takes in hand-tuned parameters and applies them to a zero-moment point (ZMP) based walk engine.

Kick Maintains intra-robot stability during kick movements; different kick settings can be loaded to allow for powerful standing kicks, quick walk-kicks, and decisive side-kicks.

Keyframes Lists scripted positions for certain movements; getting up from front and back falls is done by feeding the Body module a series of motor positions and timings.

Game State Machine Receives and relays information from the Game Controller; information from the GSM such as game state determines behavior among all robots on the field during certain times of the game.

Head State Machine Controls head movements; different conditions determine when to switch into ball searching, ball tracking, and simply looking around.

Body State Machine Dispatches movement instructions; conditions from all previous modules will cause the Nao to switch between chasing after far away balls, performing curved approaches to line up for shots, dribbling, and performing kicks when the ball is close enough.

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 Mex routines.

During calibration, a Gaussian mixture model is used to partition the YCbCr color cube into the following colors:

– Orange (Ball)

– Yellow (Goals)

– Green (Field)

– White (Lines)

Using a number of trained images, resulting in a color look-up table. While the robot is running, the main processing pipeline segments the highest-resolution color images from the camera by classifying individual pixels based upon their YCbCr values. Connected regions are recognized as either connected components or edge regions, and objects are recognized from the statistics - such as the bounding box of the region, the centroid location, and the chord lengths in the region - of the colored regions. 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 line identification 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.

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:

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

The pairs (θ_i, r_i) are accumulated in a matrix structure where lines appear as large values as shown in Figure 2. 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.

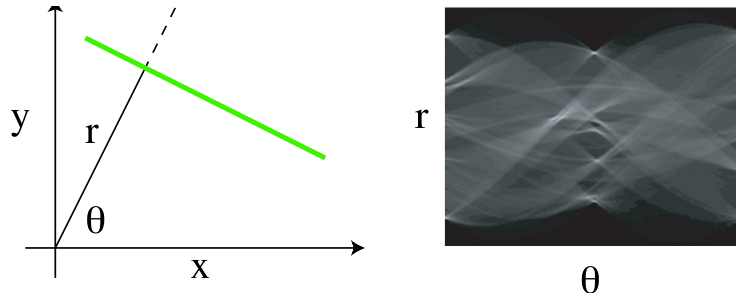


Fig. 2. Hough transformation for field line detection in images.

3.1 Calibrating and Debugging

3.1.1 Monitoring

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.

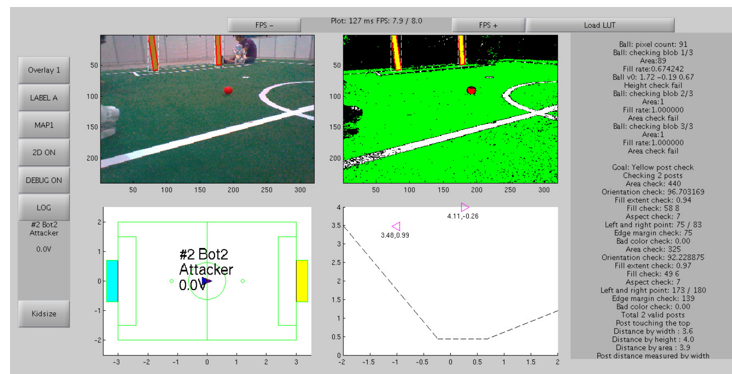
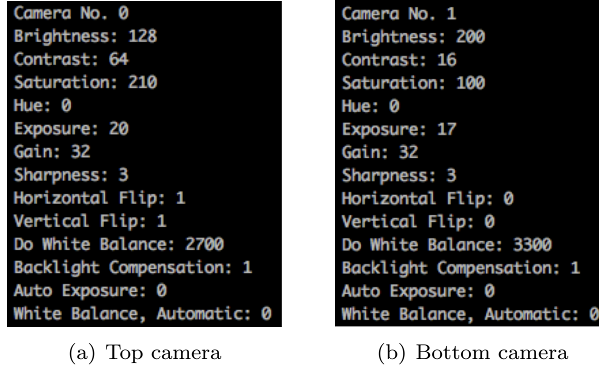


Fig. 3. Monitor for Debugging.

3.1.2 Setting Camera Parameters

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 was developed. Figures 4(a) and 4(b) display a set of camera parameter values.

These parameters should make the top and bottom camera visually appear as similar as possible because both cameras feeds are converted using the same colortable.

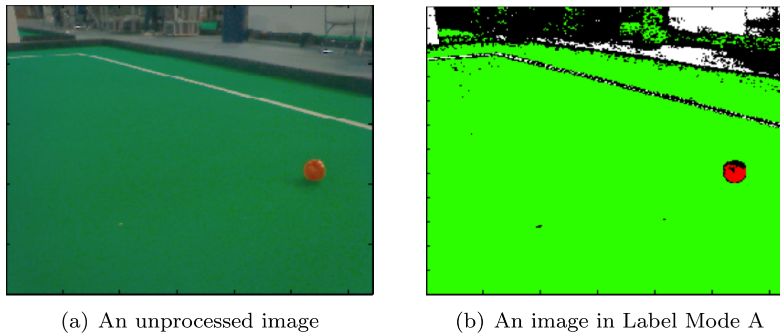


(a) Top camera (b) Bottom camera

Fig. 4. Example camera parameters.

3.1.3 Colortables

After camera parameters are set, pictures are taken from both cameras. Color segmentation training is then conducted through a Colortable Selection Tool, where colors of interest are associated with specific YCbCr values by a single click. Depending on the threshold value used, connected regions to the pixel clicked are also highlighted if their YCbCr values are close. In order to eliminate noise, we first process image packets with color definitions using a Gaussian mixture model that analyzes the probability density function of defined pixel values in conjunction with Bayes' Theorem, which expands boundaries of the color classes. As seen in the transition from Figure 5(a) to Figure 5(b), defined colors are displayed as a single shade in Label Mode A. Undefined colors show as black colors.



(a) An unprocessed image (b) An image in Label Mode A

Fig. 5

Next, we merge the pixels in 4x4 blocks through XOR operation assuming that target objects that are large enough that they won't be eliminated. As seen in the transition from Label Mode A to Label Mode B in Figure 6, which is the product of these XOR's, most of the eliminated pixels were either black, undefined pixels, or noise pixels.

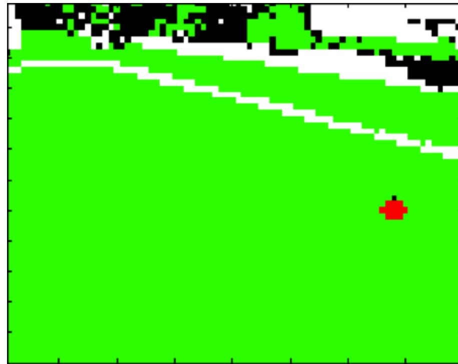


Fig. 6. An image in Label Mode B

3.1.4 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

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.

3.2 Updates to Vision Code

3.2.1 Changes in Line and Corner Detection

More checking routines have been added to line and corner detection to eliminate false-positives. Normally after first conducting a basic white and green pixel

check, white blobs are put through various tests to determine whether they are lines or corners. As seen in Figure 7, these tests are crucial as there is often a lot of white noise in the background that would result in false-positives. Line and corner tests include ground checks, cross checks, line overlap checks, length-width ratio thresholds, horizon checks, and an in-field check.

One important addition to our line tests was a distance check. Line detection from far away is useless anyways due to size fluctuation caused by noise and/or not enough white pixels surviving the transition from Label Mode A to Label Mode B. Lines that appear too close or too far away pose the risk of throwing off the robot's localization.

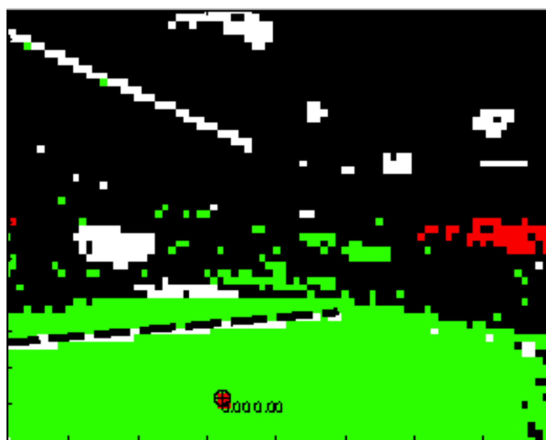


Fig. 7. Only a true white line detected in Label Mode B

In regards to corner detection, a new addition to our tests was the elimination of the center circle by means of location on field. False corners are often detected on the surface of the center circle, causing localization to be inaccurate. Ultimately, line and corner detection are low-weighted contributions to localization that are designed to pick up *most* lines and corners rather than *all* lines and corners in order to avoid detecting even a *few* false lines and corners.

3.2.2 Changes in Spot and Ball Detection

Spot and ball detection code were updated to include successful tests from line and corner detection. Such changes include the addition of a distance check and a check for field opposite the part of a ball or spot partially cut off by the camera at the periphery of its vision. Both changes made the detection of false balls much more unlikely and did not seem to hinder the detection of true balls while in competition.

3.2.3 Changes in Goal Detection

The variance in goal distance calculation has always been a crucial source of error in vision and localization. As a quick fix to the problem with the Nao robot miscalculating its distance away from goal posts by a uniform number from any point on a field, we implemented a goal distance factor. This factor multiplies the distance from the goal that the localization code uses. Before every match, we simply needed to update the factor after placing the robot at different distances from the goal posts.

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 pose estimation algorithm that incorporates a hybrid Rao-Blackwellized representation that reduces computational time, while still providing 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:

$$P(x, y, \theta) = P(\theta)P(x, y|\theta) = \sum_i P(\theta_i)P(x, y, |\theta_i) \quad (2)$$

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 Gaussian. This approach has the advantage of combining discrete Markov updates for the heading angle with Kalman filter updates for translational degrees of freedom.

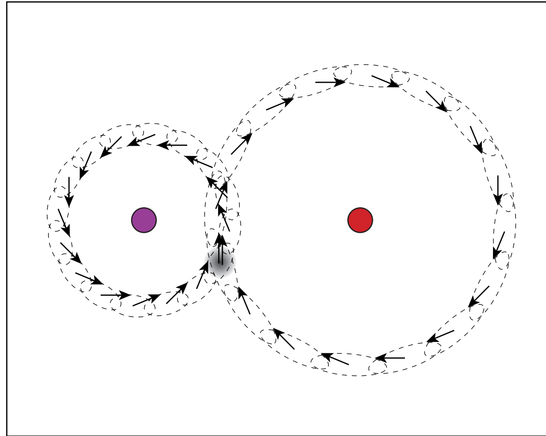


Fig. 8. Rao-Blackwellized probabilistic representation used for localization.

When this algorithm is implemented on the robots, they quickly incorporate visual landmarks and motion information to consistently estimate both the heading angle and translation coordinates on the field as shown in Figure 8. Even after the robots are lifted ('kidnapped') by the referees, they quickly re-localize their positions when they see new visual cues.

4.1 Particle Initialization

Our algorithm utilizes 200 particles to estimate the position of the robot. Properly initializing the positions of the particles helps improve the accuracy of the localization algorithm. Before the game starts, in the Ready state, the particles are initialized on the sides of the defending half of the field, as shown in Figure 9. In the Set state, if the robot is not manually replaced, its particles are initialized near the possible initial positions defined in our game strategy. Besides, during the game, when a robot falls down, its localization particles' heading angles are reinitialized.

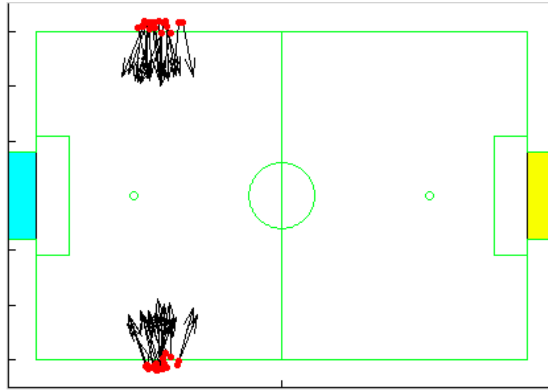


Fig. 9. Initialization of particles before game starts.

4.2 Odometry, Landmark Observation and Re-sampling

A Kalman filter is implemented to track the continuous change on the position and weight of each particle. The filtering is a product of two steps: the motion model update and the measurement update. The motion model update - also referred to as the odometry update - utilizes the robot kinematics to update the particle filter as the robot walks around the field. Given the joint angles of the robot, forward kinematics is used to compute the location of the robot's feet as it walks. The change in translation and rotation of the body of the robot are computed based on the position of the feet, as shown in Figure 10, and used to update the particle filter.

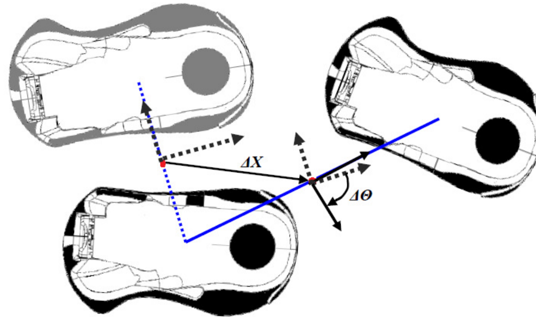


Fig. 10. Visualization of the odometry calculation after one step.

The measurement model refines this estimate using sensory inputs, such as vision-based landmark detection. As previously mentioned in this paper, our vision model is able to detect and calculate the three-dimensional position of different landmarks, including: goal posts, field lines, and corners. The measurement model incorporates these data to adjust the particle positions and their

weights in the filter. Due to the difference in reliability of different landmark detections, we incorporate different types of observed landmark positions differently. For instance, while goal post detection is used to correct both the position and heading angles, corner and line detections are mainly used to correct the heading angles since the variance in their position calculation is relatively large.

Our algorithm re-samples all of the particles every 0.1 seconds. We use the stratification method to redraw all of the particles so that the ones with higher weight will stay. Figure 11 illustrate the result of our algorithm. While the robot is moving on field, the particles are drawn in our debugging tool.

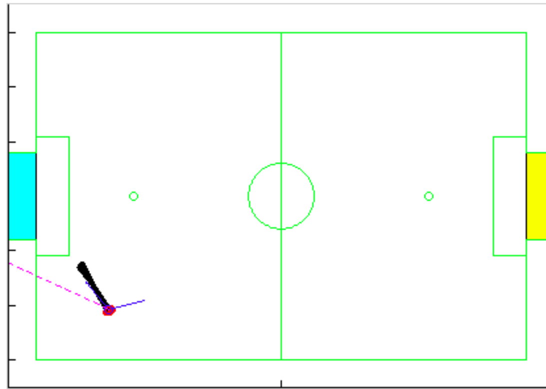


Fig. 11. The robot takes in and weighs landmarks to establish an accurate estimation of its position on the field.

4.3 Error Correction

One great challenge in the Standard Platform League is the symmetric field. Under ideal circumstances where the robot's starting position is known, the basic particle filter approach alone is enough to keep track of the correct robot pose. However, noise in the motion model, inevitable false positive detections of landmarks, and falling down, will all eventually cause the robot to converge on a pose that is symmetrically opposite the true location. This year, the problem is further complicated by the increase in the size of competition fields, which results in higher variance in vision detection. To address this problem, we use the team correcting mechanism based on goalie ball information.

For most of the cases, the goalie stays in the penalty box, and it stays close to the defending goal posts. Therefore, among the five players on the field, the goalie is most confident about its location, as well as the detected ball position. During the game, if a player robot and the goalie see the ball simultaneously but they believe the ball is on different sides of the field, it is very likely that the player robot's localization is flipped. Under such circumstances, its particles will be flipped according to the center of the field.

Moreover, since the robots are very likely to generate localization error when they fall over near the center of the field, we label robots that fall near the center

as "confused players". Such players will not make direct shots when they see the ball. Instead, they will dribble or walk-kick the ball until the goalie sees the ball and confirms their positions.

5 Motion

Motion is controlled by a dynamic walk module combined with predetermined scripted motions. One main development has been a bipedal walk engine that allows for 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 (IK) 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.

For our 2013 season, the underlying walk engine described above was not altered; the only changes were made to parameter files dictating a few controllable variables. Depending on the surface of play, a number of these parameters need to be tuned. These include the body and step height, percentages of single- and double-support, velocity and acceleration limits, and gyroscopic feedback. These parameters are tuned by hand, and a skilled operator is able to watch a robot stumble on a new surface and know exactly what needs to be tweaked. We also opted to use a slow and stable walk that remained mostly unchanged throughout the week, opting to dedicate our efforts to behavioral and localization improvements.

```
----------------------------------------------
-- Stance and velocity limit values
----------------------------------------------
walk.stanceLimitX={-0.10,0.10};
walk.stanceLimitY={0.09,0.20};
walk.stanceLimitA={-0*math.pi/180,40*math.pi/180};

walk.velLimitX={-.04,.05};
walk.velLimitY={-.02,.02};
walk.velLimitA={-.4,.4};
walk.velDelta={0.02,0.02,0.15}

--Foot overlap check variables
walk.footSizeX = {-0.04,0.08};
walk.stanceLimitMarginY = 0.035;
```

Fig. 12. Example parameters for one of our walk files.

5.1 Kicks

Our kicks this year are a combination of scripted keyframes and ZMP-based kicks. Of our three kicks – standing, walk, and side – only the walk-kick utilizes the new ZMP engine. The old-fashioned style kicks are created by specifying motor positions and timings, and must be carefully tuned by hand in order to ensure balance, stability, and power. The new kicks are inherited from our merge with Team DARwIn. Similar to how the walk engine calculates joint positions in response to motion requests of the COM and ZMP, our newer kick calculates the way that the robot needs to balance in order to perform faster and more powerful kicks.

While we utilized a mix of a keyframed standing and keyframed walk-kick during the US Open to great success, after transitioning to the ZMP walk-kick, we used this newer kick solely during our matches in Eindhoven. This allowed us to have greater control over the ball, and react quicker than opponent robots which would approach a ball and take excessive time during their keyframe motions to do a kick.

5.2 Keyframing

A keyframe file consists of a series of frames, snapshots of the 22 motor positions along with a timing by which those positions must be reached from the previous frame. Though the motors natively read and write radians to their encoder, we use degrees and convert them later for better readability.

```
angles = vector.new({
0.1, 25.5,
109.8, 11.0, -88.9, -21.4,
-13.7, -0.3, 17.1, -5.6, 5.2, 7.6,
0.0, -1.8, 14.6, -1.1, 4.9, -2.7,
109.9, -10.2, 88.7, 19.9,
})*math.pi/180,
duration = 0.400;
```

The motors in order are:

1. HeadYaw
2. HeadPitch
3. LShoulderPitch
4. LShoulderRoll
5. LElbowYaw
6. LElbowRoll
7. LHipYawPitch
8. LHipRoll
9. LHipPitch
10. LKneePitch
11. LAnklePitch
12. LAnkleRoll
13. RHipYawPitch
14. RHipRoll
15. RHipPitch
16. RKneePitch
17. RAnklePitch
18. RAnkleRoll
19. RShoulderPitch
20. RShoulderRoll
21. RElbowYaw
22. RElbowRoll

We utilize keyframed motions for two types of kicks, and also for our get-up motions. Like our walk, keyframes are hand-tuned based upon experimentation. To prolong the life of our robots, we do most of the heavy keyframe testing in Webots and then port it to the robots and perform final checks to verify full functionality.

6 Behavior

Finite state machines (FSMs) dictate the behaviors on our Naos and allow them to adapt to constantly changing conditions on the field. Updated at a speed of 100Hz, FSM's are analogous to flow charts. Our implementation of an FSM consists of a file that defines the transitions (`BodyFSM.lua` and `HeadFSM.lua` for the body and head, respectively) and a series of larger files that define specific states (i.e. `bodyPosition.lua` or `headSweep.lua`). A specific state consists of three main functions entry, update, and exit.

As their names suggest, the entry and exit functions specify actions that need to be taken when a robot first enters a state or when it finally completes a state. An example of a typical entry action is `print(_NAME..entry)`, which sends a simple print statement to the main feed and tells an operator what state a robot is currently in. Exit statements tend to be empty and are simply there to facilitate state machine functionality, but occasionally contain an action such as telling the Motion module to command a stance when leaving *bodyIdle*.

After entering and before exiting, the Nao will constantly cycle through the body of a state (the update function), querying the environment until certain conditions are met. During *bodySearch*, for example, the robot will rotate in place until either a) the ball is spotted and causes a transition to *bodyPosition* to determine how far away the ball is; b) it times out after a certain amount of time has been spent updating, and will transition to *bodyGoToCenter* to move the robot towards the center of the field in hopes of finding a ball.

Non-goalie behaviors are described here because they apply to a majority of the robots on the field (4 out of 5). The goalie will utilize the same transition file as a regular player, but instead uses a series of states unique only to the goalie.

6.1 The Body Finite State Machine (BodyFSM)

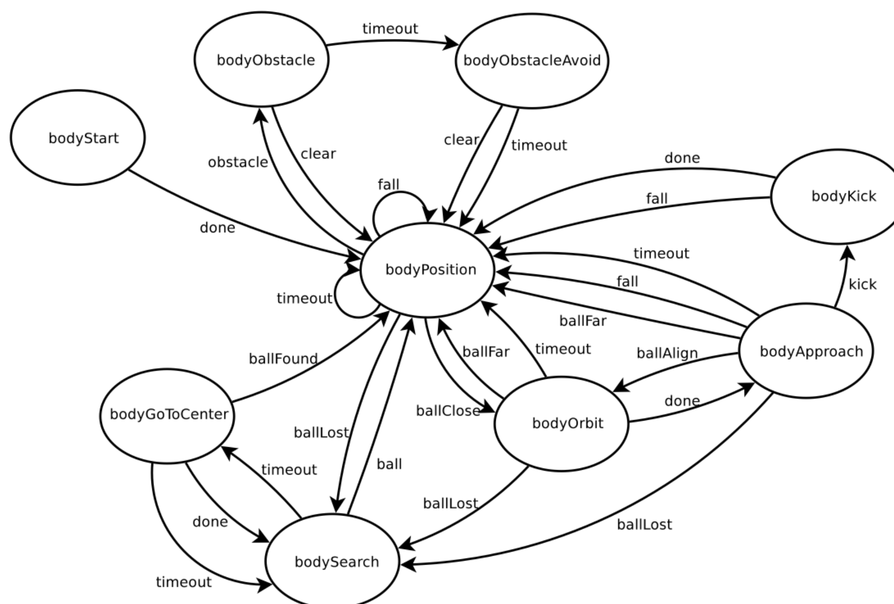


Fig. 13. Body State Machine for a non-goalie player.

The specific body states used in our 2013 code are as follows:

bodyAnticipate Goalie specific: Prepare for the ball to come within range.

bodyApproach Align for kick.

bodyChase Ball sighted; run for ball and slow as distance decreases.

bodyDribble Dribble the ball.

bodyGotoCenter Return to the center of the field, defined as $(0, 0)$.

bodyIdle Initial state when the main code is started up. Nao will be sitting awaiting button press or game state change to 'Ready'.

bodyKick Perform a standing kick.

bodyObstacle Obstacle detected.

bodyObstacleAvoid Sidestep or stop movement until the obstacle clears.

bodyOrbit Make fine adjustments to trajectory before kicking.

bodyPosition Main body state; most states will transition back here.

bodyPositionGoalie Main body state for the goalie.

bodyReady Clears temporary variables and prepares the robot to start a new half.

bodyReadyMove After a goal has been scored or when game state

is 'Ready', returns the robot to its initial position on the field.

bodySearch Revolve and search for the ball.

bodyStart Initial state when game goes to 'Playing'; handles kickoff.

bodyStop Stops the robot completely.

bodyWalkKick Perform a kick while in motion.

bodyUnpenalized Commands the Nao to stand back up and walk into the field after being unpenalized.

6.2 The Head Finite State Machine (HeadFSM)

Because the head has far fewer degrees of freedom, it is much less complex than the FSM used for the body. Its overall functionality, however, remains the same as the body state machine.

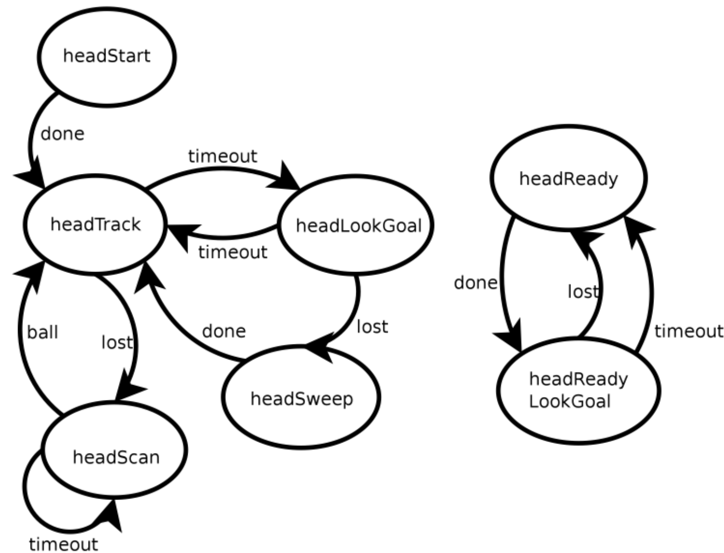


Fig. 14. Head State Machine for a non-goalie player.
Left : used while playing / Right : Used during READY state

The specific head states used in our 2013 code are as follows:

headIdle Initial state after main code is run; wait for game change.

headKick During *bodyApproach*, keep the head tilted down towards the ball.

headKickFollow Follow the ball after a kick.

headLookGoal Look up during approach to find the attacking goal posts.

headReady Localize during *BodyReadyMove* by finding lines.

headReadyLookGoal When in the initial position, look towards the attacking goal posts to localize.

headScan Look around for the ball.

headStart Initial state after the game state changes to 'Playing'.

headSweep Perform a general search, with a priority on finding goal posts.

headTrack Track the ball, moving or stationary.

headTrackGoalie Goalie-specific: Track the approaching ball.

6.3 Changing Behaviors

Adding new states is fairly simple to do. First, a declaration of a new state, followed by its relevant transitions, must be set in the head file (`BodyFSM.lua` or `HeadFSM.lua`).

```
require('NEW_STATE')
..
sm:add_state(NEW_STATE)
..
sm:set_transition(NEW_STATE, 'return-condition',
    NEXT_STATE)
sm:set_transition(PREVIOUS_STATE, 'return-
   condition', NEW_STATE)
```

Then the new state file must be placed in the same folder as the head file, and must contain an entry, update, and exit function.

```
function entry()
#actions
..
function update()
#actions
..
function exit()
#actions
```

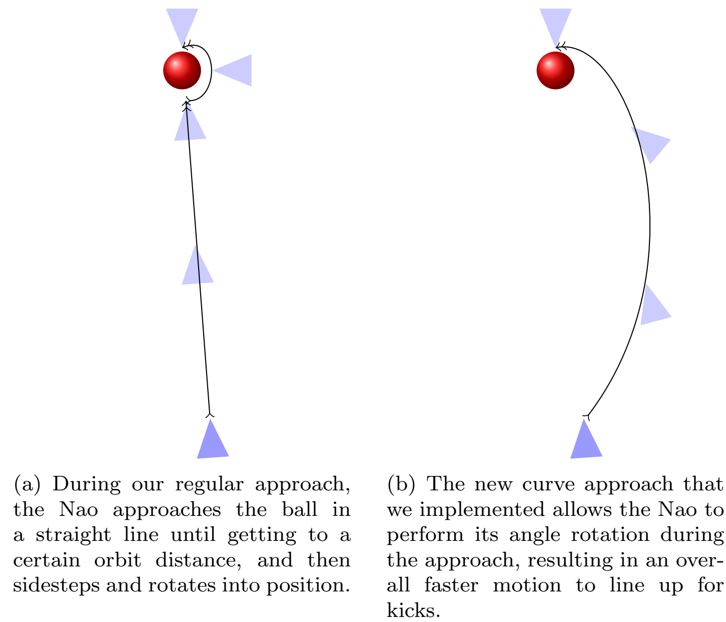
In this way, new behaviors can quickly be added or existing ones modified as required.

6.4 Updates in State Machines for 2013

For the Body State Machine, one important improvement is the approach method. Instead of the traditional direct approach method, this year we implemented the curvature approach method, as illustrated in 15(a), which enables the robots to quickly reach and kick the ball. We built it through careful calculation of the robot's approaching path: basically speaking, the desired position of the robot

in each cycle of the state machine is changing with the attacking angle. As the robot gradually rotates to face the goal, its destination moves closer to the ball, which results in a curved path.

Obstacle detection code was improved in preparation for this years competition. Data read from the ultrasound sensors allows informed decisions as to the presence of an obstacle in the robots path to be made. Significant filtering of the input data, as well as relative as opposed to absolute measurements, allows us to generate a relatively reliable signal. From this information, it is possible to perform avoidance maneuvers if necessary.



(a) During our regular approach, the Nao approaches the ball in a straight line until getting to a certain orbit distance, and then sidesteps and rotates into position.

(b) The new curve approach that we implemented allows the Nao to perform its angle rotation during the approach, resulting in an overall faster motion to line up for kicks.

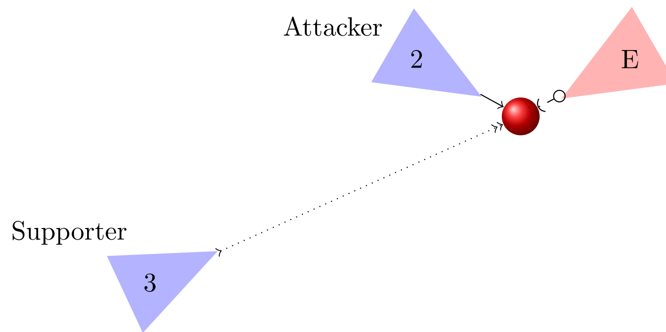
Fig. 15. Difference between our original and our improved approach.

6.5 Team Play

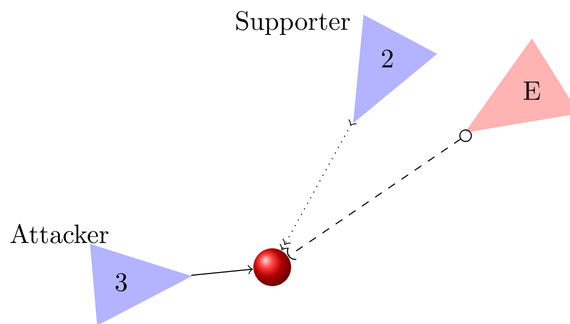
To make efficient use of the field, we have divided our team of five robots into 4 separate and distinct roles. These various roles have differing starting positions, and inhabit different parts of the field after kickoff. Our roles are as follows:

Goalie	1	Stays in and around the defensive goal to clear the ball when it comes close.
Attacker	2	Goes directly towards the ball and kicks.
Supporter	3	Follows the attacking robot up-field, but stays at a respectable distance away—usually about midfield.
Defender	4	The defending robot positions itself between the ball and defensive goal area.
Defender Two	5	Performs double duty with the first defender, but has a different initial position.

Our primary strategy is to constantly keep the ball moving down-field. To encourage this, the four general players (non-goalies) are constantly communicating over Wi-Fi, sharing their global position, relative distances to the ball, and current roles. Our code works in such a way that the role of Attacker changes often during a game, based on ETA's to the game ball.



(a) The ball is closest to Nao 2, and so it is currently the Attacker. Nao 3 sights the ball, but because its distance is second farthest away, it becomes the Supporter.



(b) After the ball changes position and becomes closest to 3, it now becomes the Attacker. The Nao that was formerly an attacker, now being second furthest away, assigns itself the role of Supporter.

Fig. 16. Illustration of how roles change between team members.

Take for example, this situation. Following kickoff, Nao #2, initially assigned as the Attacker, gets the ball into the opponent half. An opponent defender steals the ball away, and with a powerful kick, sends it back into our half. If Defender Two (Nao #5) finds the ball stopped closest to him, he will inform the team that he is switching into the Attacker role. The other three general players will then check how far they are to the ball, and assign themselves roles in order of ascending distance to the ball. The next closest robot, regardless of number and initial role, would become the new Supporter, while the two furthest away would become Defenders One and Two.

In this way, the team can reach and move the ball much quicker and with more efficiency by behaving as a dynamic unit.

7 Summary

While the UPennalizers broke their annual tradition of making it to the quarter-final matches every year, the team has continued to keep pace with the rest of the league. With a new batch of undergraduates ready to pass on their knowledge to new team members in the fall, the UPennalizers' future looks as bright as ever.

Our 2013 demo code has been released on our website under the GNU public license, and we hope that it will be of use to future teams.