To appear in *Proceedings of the RoboCup International Symposium 2013* (RoboCup 2013), Eindhoven, Netherlands, July 2013.

The 2012 UT Austin Villa Code Release

Samuel Barrett, Katie Genter, Yuchen He, Todd Hester, Piyush Khandelwal, Jacob Menashe, and Peter Stone

Department of Computer Science
The University of Texas at Austin
{sbarrett, katie, hychyc07, todd, piyushk, jmenashe, pstone}@cs.utexas.edu
http://www.cs.utexas.edu/~AustinVilla

Abstract. In 2012, UT Austin Villa claimed the Standard Platform League championships at both the US Open and the 2012 RoboCup competition held in Mexico City. This paper describes the code release associated with the team and discusses the key contributions of the release. This release will enable teams entering the Standard Platform League and researchers using the Naos to have a solid foundation from which to start their work as well as providing useful modules to existing researchers and RoboCup teams. We expect it to be of particular interest because it includes the architecture, logic modules, and debugging tools that led to the team's success in 2012. This architecture is designed to be flexible and robust while enabling easy testing and debugging of code. The vision code was designed for easy use in creating color tables and debugging problems. A custom localization simulator that is included permits fast testing of full team scenarios. Also included is the kick engine which runs through a number of static joint poses and adapts them to the current location of the ball. This code release will provide a solid foundation for new RoboCup teams and for researchers that use the Naos.

Keywords: RoboCup, Nao, SPL, UT Austin Villa

1 Introduction

Developing intelligent behavior for robots is difficult due to the time and effort needed to create and debug code for robots. This code release attempts to reduce that burden for researchers that use humanoid robots. Specifically, it contains the code from the UT Austin Villa team that competes in RoboCup, the Robot Soccer World Cup. The international research initiative behind RoboCup pushes towards advancing robotics and artificial intelligence by using the game of soccer as a test domain. Soccer requires robots to deal with real world issues such as teamwork, localization, motion, and vision all in real time while interacting with an environment that is not under their control. The long-term goal of RoboCup is to build a team of 11 humanoid robot soccer players that can beat the best human soccer team on a real soccer field by the year 2050 [5].

RoboCup is organized into several leagues, including both simulation leagues and leagues that compete with physical robots. This paper describes the code used by the 2012 championship team in the Standard Platform League (SPL)[1]. All teams in the SPL compete with identical Aldebaran Nao humanoid robots robots[2], making it essentially a software competition. This format allows for teams to share ideas and code, such as this code release, to speed the development of capable, intelligent robots.

UT Austin Villa has competed in the Standard Platform League with the Nao robots every year since the Nao was introduced in 2008. Through these years, we have built a substantial code infrastructure for robot soccer that served as the base for our championship in 2012 [2, 1]. This paper describes the partial release of that code base, including its software architecture, stream-lined vision processing, localization, localization simulator, kick engine, and debug tool. The high level strategy and behavior code is omitted, but is described in [3, 1].

2 Code Release

This section describes the steps necessary to use the released code[3] and adapt it to new situations. More documentation is included in the code release in the form of a README and comments in the code. The vision system is based on colors, which it interprets through the use of color tables. Generating these color tables is done with the provided UTNaoTool, using a combination of painting colors onto the images and annotating objects in the images as described in Section 4. In addition, as the included behavior only directly walks towards and kicks the ball, it will be necessary to create more complex behaviors. Further behaviors can be created using a task hierarchy that is written in Lua for rapid development and testing.

The main components of the code are contained in the `core` directory, with modules separated into their respective directories, but note that the behavior code is contained in the `lua` directory. The main access point to modules is the `processFrame` function, and the main control of the vision and cognition process is primarily handled by the `processFrame` function in the file `lua/init.lua`. The main control of the motion process is handled by `processFrame` in `MotionCore.cpp`. The NaoQi interface is primarily found in `interfaces/nao/src/naointerface.cpp`.

In 2012, UT Austin Villa won the Standard Platform League at the 16th International RoboCup Competition in Mexico City, Mexico. Twenty-five teams entered the competition, and games were played with four robots on each team. Our code release allows new teams to ramp up quickly with modules that have been demonstrated to support a RoboCup championship.

In principle, this code release can be useful for projects other than RoboCup and on robots other than the Nao. The included tools for color-based vision are easy and quick to use. The localization algorithms used are general and can be

---

1 http://www.tzi.de/spl/

2 http://www.aldebaran.com/

3 `http://www.cs.utexas.edu/~AustinVilla/?p=downloads/source_code_and_binaries`

extended to use other types of landmarks, while coordinating with other robots. Furthermore, the most useful tool included is the general infrastructure of the memory and modules. This infrastructure permits creating and replaying logs in a simple and customizable manner, and these logs are vital for understanding and debugging behaviors on robots. In addition, it allows for decoupling the code modules, while still maintaining good computational performance. This infrastructure is described in more depth in the following section.

3 Software Architecture

When dealing with robots, it is important that the code be both easily debugable and testable. The design's key element is to enforce that the environment *interface* and its *logic* are kept distinct (Figure 1), and that all communication is done via shared *memory*. The interface simply extracts sensor data into memory and then sends commands to the joints, whether the system is a robot, simulator, or debug tool. This design allows the same logic to be applied regardless of the underlying system being used. The advantages of this architecture are discussed further in [1].

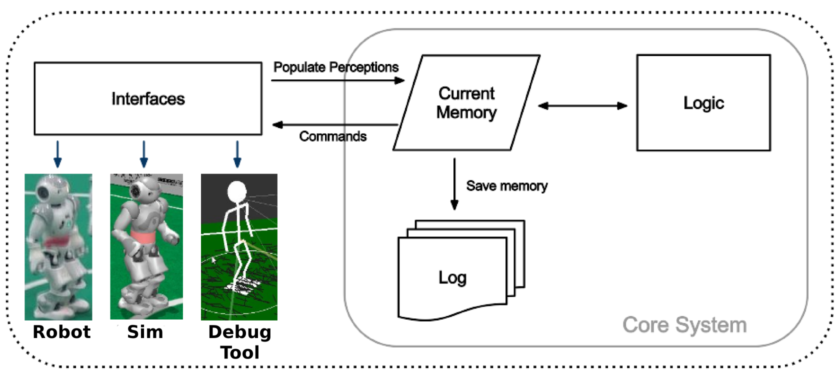


Fig. 1. Overview of the UT Austin Villa software architecture, which separates interface, memory, and logic.

Furthermore, for better modularity, the logic is organized into a number of modules that each take care of specific functionality. These modules are discussed in depth in Sections 4–6. In addition, the memory is split into memory blocks that group related pieces of information and allow us to analyze which modules rely on each piece of information. Modules communicate via the memory blocks; for example, the higher level strategy behavior talks to the walk and kick engines in the form of motion request blocks and get information about odometry and the current status of the kick in the form of information blocks.

To take advantage of the modularity of our code, the execution is split into three separate processes:

1. Interface: this code runs at 100 hz, is called by naoqi (or a simulator), and copies the current sensor information to memory and writes out the current motion commands
2. Motion: this code runs at 100 hz and converts high level commands into direct commands for the motors
3. Vision: this code runs at 30 hz and processes the incoming imaging and selects the high level commands

Splitting execution into three processes allows us to restart crashed processes, as well as debug each process separately. In addition, it aids in development as changing code in the motion and vision processes does not require the slow procedure of restarting the NaoQi interface on the robots. However, this design requires us to store our memory structure in shared memory. The majority of the information is kept separate for each of the processes, so mutex locks are only required when copying local information into these shared blocks.

The walk engine included in the release was developed by the B-Human team from the University of Bremen [8], using the code for interfacing with this walk engine released by Bowdoin College's Northern Bites team [6]. All the other major modules, comprising the large majority of the codebase, were developed by the Austin Villa team. These modules are described in the following sections.

4 Vision

The vision system is dedicated to the task of detecting the ball, goals, field lines, and robots in the camera images and reporting their relative distance and bearings to the robot. The team's vision system divides the object detection task into 4 stages, each of which is carried out on both cameras. These stages are listed below:

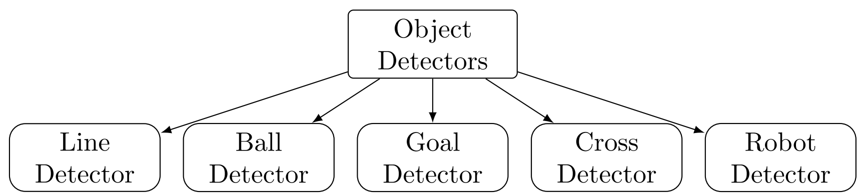
1. **Segmentation** - The raw image is read and segmented using a color table.
2. **Blob Formation** - The segmented image is scanned using horizontal and vertical scan lines, and "blobs" are formed for further processing.
3. **Object Detection** - The blobs are merged into different objects.
4. **Transformation** - The information given by the pose of the robot is used to generate ground plane transformations of the line segments detected.

In addition, our tool's vision analysis system provides a set of key features for debugging and tuning color calibrations. In particular, the log annotation and color table generation features are useful for creating and analyzing color tables with measurable accuracy. Since the vision system is based on segmented images, proper construction of color tables is key to its effectiveness.

The overall framework for these processing steps is outlined in detail along with high level descriptions of the color table analysis system in our previous papers [2, 1]. We therefore focus on new implementation techniques that are found in this code release, the first of which is the arrangement of object detection methods into segmentation and detector modules.

The vision system relies on a handful of primary classes:

- `TransformationProvider` - This class converts body position estimates into transforms for mapping pixel values into world locations and vice versa.
- `Classifier` - Classifies pixels by color and constructs vertical and horizontal runs of contiguous color. This is the segmentation phase.
- `BlobDetector` - Merges adjacent runs of common color into a number of blobs. This is the blob formation phase.
- `[Line,Ball,Goal,Cross,Robot]Detector` - These detector classes run in sequence to transform blobs into complete object detections. Each detector is responsible for finding relevant blobs, performing sanity checks, and indicating results to the global world object array. Various statistics such as bearing and image position are stored as well.



It should be noted that the cross detector, which is responsible for discerning the cross directly in front of the goal box, was used only for the goalie as a localization aid. In general it can be difficult to discern the cross from other lines on the field due primarily to occlusion, however in the setting of the goalie we are able to enforce more restrictive assumptions on what qualifies as the cross. Other field lines are too ambiguous to provide the goalie with enough certainty in its position. Goal posts provide a good alternative to field lines as localization anchors, however they often lie outside the goalie's field of view and would thus require a complete head turn to maintain positional certainty. The cross overcomes both of these drawbacks by providing a unique localization anchor that is consistently between the goalie and the main area of play.

The vision system also implements selective high-resolution scanning based on subsampled pixel values, which is used most effectively for locating the ball across large distances. In normal operation, each of the listed object detectors relies on a 320x240 subsampled segmented image to ensure that vision processing runs at the hardware-enforced framerate of 30 Hz. At any point a detector may request a high-resolution scan of the top camera image for a particular color from the Classifier. The Classifier then reclassifies all regions surrounding subsampled pixels of the detected color using the full 640x480 resolution allowed by our implementation. Generally only small regions of the image are reclassified, thus providing the benefits of full resolution images with virtually no hit to performance. By using this technique we are able to reliably detect the ball at distances up to 5 meters, up by a factor of 2 over detections with subsampled images. High resolution scans of the goal are also enabled, which similarly improves goal detection distances.

5 Localization

Keeping track of the location of the robot is important in all robot tasks, especially in the RoboCup domain. The goal of the localization system is to take in the detected objects from vision, and output the location of the robot, the ball, the robot's teammates, and the robot's opponents. Having a good world model like this enables the robot to make smart strategic decisions during the game. Like many other teams at the competition [7, 4], since 2011 we have used a multi-modal 7-state Unscented Kalman Filter (UKF) for localization [2]. The UKF provides a number of benefits compared to other approaches such as Monte Carlo localization as it is computationally efficient and enables easy sharing and integration of ball information between teammates.

One of the challenges of the RoboCup Standard Platform League in 2012 is that there are no unique markers anywhere on the field. Therefore, the robots must do a good job of integrating both odometry and vision estimates of landmarks while recovering from noise added by bumps, falls, and kidnapping. Since the field is symmetrical, the robot has no information to recover its orientation on its own. Instead, the robot takes advantage of shared information from teammates to resolve which direction it is facing. Our solution to resolving this issue was to have robots check whether their teammates thought the ball was in the same location or in the symmetrically opposite location. It is important to listen to more than one teammate, as we do not want one teammate that is going the wrong way to convince the entire team to start going the wrong direction. This functionality is described in the `processSharedBall` and `checkSharedBallForFlips` functions in the file `localization/UKFModule.cpp`.

However, testing and debugging localization with full teams of robots is quite challenging. Therefore, we wrote a localization simulator to implement, test, and debug our solutions, shown in Figure 2. The simulator can be run from the world window of the tool included in the code release. The simulator takes advantage of the modularity of our code. Instead of interfacing with the code at the perception and motor command layers, it populates the code with simulated output from the vision module, and then uses the code's output to the kicking and walking modules to move the robots in the simulation. The simulator proved useful not just for localization, but also for testing various strategies and behaviors.

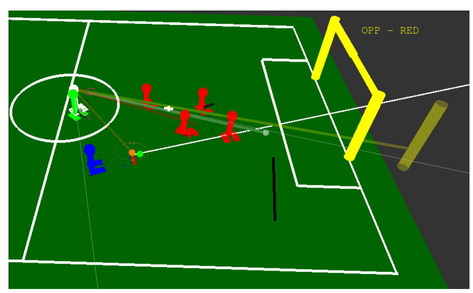


Fig. 2. The localization simulator.

The main interface into the localization module is in the `processFrame` function in the file `localization/UKFModule.cpp`, and the updates from observations is handled by the `processObservations` function. The UKF has many parameters that affect how it handles observations and odometry updates. These parameters were initially optimized using a number of long logs of the robot's observations and movements. Additional tuning was performed by hand to handle issues that were discovered during test games.

6 Kick Engine

The code release includes our static standing kick engine, which is found in the file `motion/KickModule.cpp`. This kick engine utilizes a series of keyframe states that the robot moves through at specified times. For each state, we define a state time, a joint movement time, the desired center of mass in x,y,z space, and the desired kicking foot position in x,y,z space. These values are defined for each state in the file `lua/cfgkick.lua`. The robot's joints are controlled using inverse kinematics to reach each desired state in the allocated joint movement time. In particular, our kick engine utilizes the states shown in Table 1.

State	Description
Stand	stand at the same height as when walking
Shift	shift the center of mass away from the kicking foot
Lift	lift the kicking foot
Align	align the kicking foot
Spline	swing the kicking foot through the ball
ResetFoot	reset the kicking foot such that it is centered under the body
FootDown	place the kicking foot back on the ground
ShiftBack	shift the center of mass back towards the center of the body
FinishStand	stand at the same height as when walking

Table 1. States utilized by our static kick engine

In the Spline state, splines are used to compute the path for the kicking foot to follow as it moves forward through the ball. We use splines in this state to obtain a smooth path for the kicking foot to follow, as informal experimentation showed that a smooth kicking foot trajectory resulted in a more consistent kick. You might note that two additional states — Kick1 and Kick2 — are defined in the file `lua/cfgkick.lua`. The state time for these states is set to 0 though, so these states are not currently utilized by the kick engine (although the target for the end of the Spline state uses the desired swing foot position defined for the Kick2 state). However, one could remove the Spline state by setting its state time to 0, and give positive state times to the Kick1 and Kick2 states, to obtain a kick that does not use splines to compute the path for the foot to follow as it moves through the ball.

The kick engine obtains the desired kick distance by controlling the amount of time needed to move the foot during the Spline state. For RoboCup 2012, the only static standing kicks that the robots used were straight kicks ranging between 1.5 meters and 3.5 meters. Hence, 190 ms were spent in the Spline state for the 3.5 meter kicks, while 300ms were spent in the Spline state for the 1.5 meter kicks.

The kick engine alters the desired swing foot position for the Align and Spline states of the kick engine based on the desired kick distance and the current ball position with respect to the kicking foot. The complete details of this kick engine are available in [2], and the improvements to it for 2012 are available in [1].

7 Conclusion

This paper describes the partial release of 2012 RoboCup SPL champions UT Austin Villa's 2012 code base. The code release includes UT Austin Villa's software architecture, stream-lined vision processing, localization, localization simulator, motion engine, and debug tool. This release omits the strategy and high level behaviors for playing soccer. Importantly, all of the code was developed on a flexible architecture developed previously, that enables easy testing and debugging of code. This code release is intended to serve as a useful foundation for any team entering the RoboCup SPL competition as well as any researchers using the Nao robots or similar platforms.

Acknowledgments

This code release builds upon the work of all of the past UT Austin Villa teams. We thank all the previous team members for their important contributions, especially Michael Quinlan and Mohan Sridharan. This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. LARG research is supported in part by NSF (IIS-0917122), ONR (N00014-09-1-0658), and the FHWA (DTFH61-07-H-00030).

References

1. S. Barrett, K. Genter, Y. He, T. Hester, P. Khandelwal, J. Menashe, and P. Stone. UT Austin Villa 2012: Standard Platform League world champions. In X. Chen, P. Stone, L. E. Sucar, and T. V. der Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*. Springer Verlag, 2012.
2. S. Barrett, K. Genter, T. Hester, P. Khandelwal, M. Quinlan, P. Stone, and M. Sridharan. Austin Villa 2011: Sharing is caring: Better awareness through information sharing. Technical Report UT-AI-TR-12-01, The University of Texas at Austin, Department of Computer Sciences, AI Laboratory, January 2012.
3. S. Barrett, K. Genter, T. Hester, M. Quinlan, and P. Stone. Controlled kicking under uncertainty. In *The Fifth Workshop on Humanoid Soccer Robots at Humanoids 2010*, December 2010.
4. G. Jochmann, S. Kerner, S. Tasse, and O. Urbann. Efficient multi-hypotheses unscented Kalman filtering for robust localization. In *RoboCup 2011: Robot Soccer World Cup XV*. 2012.
5. H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. RoboCup: The robot world cup initiative. In *Proceedings of The First International Conference on Autonomous Agents*. ACM Press, 1997.
6. O. Neamtu, W. Dawson, E. Googins, B. Jacobel, L. Mamantov, D. McAvoy, B. Mende, N. Merritt, E. Ratner, N. Terman, J. Zalinger, J. Morrison, and E. Chown. Northern Bites code release, 2012. `https://github.com/northern-bites`.
7. M. Quinlan and R. Middleton. Multiple model Kalman filters: A localization technique for RoboCup soccer. In *2009 RoboCup Symposium*, 2009.
8. T. Röfer, T. Laue, J. Müller, A. Fabisch, F. Feldpausch, K. Gillmann, C. Graf, T. J. de Haas, A. Härtl, A. Humann, D. Honsel, P. Kastner, T. Kastner, C. Könemann, B. Markowsky, O. J. L. Riemann, and F. Wenk. B-Human team report and code release, 2011. `http://www.b-human.de/downloads/bhuman11_coderelease.pdf`.