L3M*Joint Team report
Participation in the 2010 RoboCup SPL league

Vincent HUGEL and Nicolas JOUANDEAU
hugel@lisv.uvsq.fr and n@ai.univ-paris8.fr

*Les Trois Mousquetaires – Los Tres Mosqueteros – The Three Musketeers

Table of Contents

1 Introduction

In 2010 the L3M team participated in the NAO SPL league for the second time. The team was originally composed of French members and became a joint team with two Spanish Universities in 2010 after the German Open.

The team is now composed of two French Universities, the University of Versailles (UVSQ) represented by the LISV Laboratory[1], the University Paris 8 (UP8) represented by the LIASD Laboratory[2], two Spanish Universities, la Universidad de Murcia, represented by el Grupo de Investigación de Ingeniería Aplicada (GIIA), la Universidad Politécnica de Valencia, represented by el Instituto Universitario de Automática e Informática Industrial (AI2), and one French Engineering School, ISTY, that is a Science and Technology Engineering School that depends on the University of Versailles. The researchers involved this year were Vincent Hugel (team leader, LISV), Nicolas Jouandeau (LIASD), Pierre Blazevic (head of ISTY), Patrick Bonnin (ISTY), Humberto Martínez Barberá (GIIA), Juan Francisco Blanes Noguera (AI2). The team students were Juan Jose Alcaraz Jimenez (GIIA), Manuel Muñoz Alcobendas (AI2), Pau Muñoz Benavent (AI2), Aldenis Garcia Martinez (UP8), Loïc Thimon (UP8) and Vincent Clavel (ISTY).

The French part of the team also participated in the 2010 RomeCup and the 2010 German Open that took place in Magdeburg. These participations were very useful to focus on the real problems of vision tuning and to fix them.

In RoboCup 2010 the L3M team won 2 games, did 1 draw and lost 3 games, see table 1. L3M scored 5 goals. This was enough to pass the first round but not the second one. L3M ranked 3rd in the second pool out of 4 teams. The final ranking of our team is between the 9th and the 12th place, at the end of the first half, out of 23 teams in competition. This is not a bad result taking into account the high level of the tournament, and the difference of human and financial resources between teams.

Game	Teams	Score
1st	L3M - HTWK	0-2
2nd	L3M - Northern Bites	2-0
3rd	L3M - Zadeat	2-0
4th	L3M - CHITA Hominids	1-1
5th	L3M - Austin Villa	0-5
6th	L3M - CMurfs	0-2

Table 1: L3M soccer games results

This second participation was the opportunity to set up a joint team and to improve the behaviors of last year [1].

[1] Laboratoire d'Ingénierie des Systèmes de Versailles, Laboratory specialized in System Engineering, Robotics and Mechatronics
[2] Advanced Computer Science Laboratory of Saint Denis, Laboratory specialized in computer science

2 Module architecture

In addition to the two modules that were used in the embedded software of 2009, namely *mvt* module and *vision* module [1], a new module – named *comm* – was created to deal with the communication with the gamecontroller and the communication between team robots. The UDP protocol in burst mode is used for communications.

The *comm* module is waiting for messages from the gamecontroller. Once a message is received it is analyzed and useful information is copied into a shared memory. This is done simply by calling the Aldebaran *InsertData* method on a memory proxy whose type is $ALPtr < ALMemoryProxy >$. The information used so far from gamecontroller are game state, team colour, kickoff team, penalty, and score. Shared memory is updated when a new value is received from the gamecontroller.

In the *mvt* module callbacks are defined and being subscribed, i.e. associated with changes in shared memory. The Aldebaran method *subscribeToMicroEvent* is used on a memory proxy to do that. There are four callbacks, to deal separately with game state, team colour, kickoff team and penalty. Score information is not used yet. When a new value is written in the shared memory by the *comm* module, the appropriate callback is run in the *mvt* module to copy the information in the local memory space so that the decision engine inside the *mvt* module can update the robot's state with the last information provided by the gamecontroller.

The *comm* module is also running threads that are awaiting messages from other robots. Messages received are copied into shared memory for further use by the *mvt* module. The *comm* module contains bound methods that can be used by the *mvt* module to send messages to other robots. The sending method calls a *sendto* function for each of the robots. The broadcast sending mode was not used because it posed serious problems we could not fix.

3 Velocity control for locomotion path planning

Last year the L3M robots were not programmed to track and follow the ball in real time. This year our robots can track the ball and head for it while adjusting their path towards the ball taking into account the relative position of the poles that were detected and refreshed. First we used the position control walking primitives from Aldebaran to update the path to be followed by the robot in real time. Unfortunately these primitives were responsible for frequent and strange jolts that nearly made the robot collapse. Path tracking in these conditions was not reliable. That is why velocity control walking primitives from Aldebaran were utilized instead.

Three kinds of path tracking algorithms were designed. The first path tracking algorithm was designed to head for the ball and to halt when arrived close to the ball with one of the feet in front of the ball. The objective of the second path tracking behavior is to align the robot with the ball-target direction behind the ball in the direction of the target. This is useful to attack the opponent goal when it is defined as target, but it can also be useful to get the robot aligned with any other target like a virtual point for placement purposes. The third and last path tracking behavior consists of a aligning the robot with the ball and the target but the robot must place itself between the ball and the target. This is obviously useful for defense purposes.

The three functions *UpdateGotoBall()*, *UpdateAlignedTarget()*, *UpdateAlignedTargetForDefense()* given below are called by the behavior decision engine to update the path that the robot must follow. All these three functions first call the *UpdatePerceptionServoHeadToBall)()* function that is used to servo the head to the ball. This function calls the *DoPerception()* function that gets visual information from the *vision* module and calculates angles and distance to objects detected in the image.

```
void Behavior_robot::UpdatePerceptionServoHeadToBall(void) {
    DoPerception();
    // servo head
    if (ball_seen) {
        PtrMvt->fMotion->post.angleInterpolationWithSpeed(std::string("HeadYaw"),
        ball_horiz_angle_for_servo,0.85);
        PtrMvt->fMotion->post.angleInterpolationWithSpeed(std::string("HeadPitch"),
        ball_vert_angle_for_servo,0.85);
    }
}

void Behavior_robot::UpdateGotoBall(void) {
    UpdatePerceptionServoHeadToBall();
    if ((ST_nb_opp_poles > 0) && (ball_distance < 1.2))
       trajBallDone = UpdateTrajectoryGotoBallAlignedTarget(opp_goal_horiz_angle);
    else
        trajBallDone = UpdateTrajectoryGotoBall();
    if (ball_distance > 0.45)
        obstacleAvoidance();

    DealWithSaturation();
    PtrMvt->fMotion->setWalkTargetVelocity (fx, fy, ftheta, freq);
}
```

When the distance to the ball is less than 1.2 $[m]$, and at least one opponent pole has been detected recently or a few seconds earlier and stored in the pole history, there is a subcall to the function that deals with aligning the robot with ball and target to attack the opponent goal. If no pole was detected the robot simply goes to the ball. In case the robot approaches the ball at a distance less than 0.45 $[m]$, the obstacle avoidance procedure is run. For velocity control we use the *setWalkTargetVelocity()* from Aldebaran *Motion* module that requires four parameters as inputs that we named fx, fy, $f\theta$, and $freq$.

- fx is the step length along x-axis as a fraction of maximal step length along x. A negative value means a backward step. The maximal step length is set to 0.04 $[m]$.
- fy is the step length along y-axis as a fraction of maximal step length along y. A negative value means a sideways step to the right. The maximal step length is set to 0.04 $[m]$.
- $f\theta$ is the angle between the feet as a fraction of maximal step angle. A positive value result in a left turn(anti-clockwise) and a negative value results in a right turn(clockwise). The maximal step angle is set to 0.698 $[rad]$, which is 40 $[deg]$.

- $freq$ is the step frequency as a fraction of linear interpolation between minimal step frequency and maximal step frequency. One cycle is considered to be a phase of double leg support followed by a phase of single leg support.

fx, fy, $f\theta$ are internal variables defined as object members whose range is $[-1, 1]$. $freq$ range is $[0, 1]$. The $freq$ parameter was always set to 1. The *DealWithSaturation()* function makes it sure that fx and fy parameters for velocity control stay inside their respective range while respecting their proportion relative to each other. The velocity-controlled trajectory is updated in the *UpdateTrajectoryGotoBall()* – described in 3.1 – and *UpdateTrajectoryGotoBallAlignedTarget()* – described in 3.2.

```
void Behavior_robot::UpdateAlignedTarget(void) {
    UpdatePerceptionServoHeadToBall();
    trajBallDone = UpdateTrajectoryGotoBallAlignedTarget(angle_target);
    DealWithSaturation();
    PtrMvt->fMotion->setWalkTargetVelocity (fx, fy, ftheta, freq);
}
```

This function always calls *UpdateTrajectoryGotoBallAlignedTarget()* described in section 3.2.

```
void Behavior_robot::UpdateAlignedTargetForDefense(void) {
    UpdatePerceptionServoHeadToBall();
    trajBallDone = UpdateTrajectoryGotoBallAlignedTargetForDefense(angle_target);
    DealWithSaturation();
    PtrMvt->fMotion->setWalkTargetVelocity (fx, fy, ftheta, freq);
}
```

This function always calls *UpdateTrajectoryGotoBallAlignedTargetForDefense()* described in section 3.3.

3.1 Velocity-controlled trajectory to go to the ball

This section describes the algorithm used in the *UpdateTrajectoryGotoBall()* function.

In the case where the robot does not see the ball or has not seen it after the refreshment period has elapsed, the robot halts – fx, fy and $f\theta$ are set to 0 – if it is at a distance to the ball less than 2 $[m]$ or if it is rotating – the angular velocity rate component $f\theta$ is not 0. It the robot is not rotating and if it is enough far away from the ball, the robot continues to walk according to the last values set in fx and fy velocity parameters, this is to have a chance to catch the ball while walking before halting for searching the ball again.

In case the robot has detected the ball the following instructions are executed:

- the robot checks if it has arrived close to the ball. The test is the following:

$$|d_B.\cos\alpha_B^{horiz}| < d_B^{close} \quad and \quad |d_B.\sin\alpha_B^{horiz}| < d_B^{close} \tag{1}$$

where d_B is the distance to the ball, α_B^{horiz} the horizontal angle between the robot's longitudinal axis and the ball, and d_B^{close} is a fixed threshold under which the ball is considered as close enough for the robot to be arrived at the ball. $d_B^{close} = 0.25$ $[m]$.

- the robot calculates the target angle, namely α_T^{horiz}, it needs to turn to have its longitudinal axis aligned with the target point. The default value of the target angle is the ball horizontal angle. However it is possible to specify that the robot arrives at the ball with the left or right foot in front of it in order to be ready for kicking the ball. In this case the target angle is modified when the robot is at a distance from the ball below 0.6 $[m]$ by adding a correction angle to the ball horizontal angle. This correction angle is obtained by considering a virtual target point for the robot. This target point is at 0.05 $[m]$ from the ball perpendicular to ball heading direction. The target point will be on the right of the ball if the kicking foot selected is the left foot and vice-versa.

- the robot checks if it only needs to execute a rotation motion. This will be the case once it has arrived close to the ball, or when the ball horizontal angle is greater than 85 $[deg]$ in absolute value. In this last case it is preferable to rotate first before heading for the ball.

- In case only a rotation motion is required, fx and fy are set to 0, otherwise they are adjusted as follows:

$$\begin{gathered}\beta = (d_B - d_B^{close})/(d_B^{dec} - d_B^{close}) \\ if(\beta < 0.2) \quad \beta = 0.2 \\ fx = \beta.\cos\alpha_T^{horiz} \\ fy = \beta.\sin\alpha_T^{horiz}\end{gathered}$$

  where d_B^{dec} is a distance under which the robot will start to decelerate because the weight coefficient β will be less than 1. d_B^{dec} is set to 0.47 $[m]$. The weight coefficient β is saturated to minimal value of 0.2 to prevent the robot from stepping on the spot.

- Next, a minimal angle α_{min} and a maximal angle α_{max} are used to deal with the target angle to turn. If the target angle is below α_{min} then fy and $f\theta$ are set to 0. This means that only forward velocity is used for the trajectory to the ball. If the target angle is greater than α_{max} the target angle is saturated to α_{max}. α_{min} is set to 10 $[deg]$ when one of the feet has been selected for positioning and kicking. When there is no requirement for positioning one of the feet for kicking, α_{min} is set to 16 $[deg]$ when the ball is very close at a distance of 0.17 $[m]$.

- The minimal angle is subtracted from the target angle, taking sign into account. To get a value between 0 and $1 - \alpha_{min}/\alpha_{max}$ the target angle is then divided by α_{max}.

- $f\theta$ is then calculated with a parabolic profile as follows:

$$f\theta = k_\theta.\alpha_T^{horiz}.|\alpha_T^{horiz}| \tag{2}$$

  where k_θ is a coefficient used to scaled up/down the parabolic shape. k_θ is fixed to 1.5. To prevent the robot from stepping on the spot, a minimal value of $f\theta$ is defined as:

$$f\theta_{min} = \alpha_{min}/(3.max\_step\_angle) \tag{3}$$

It is used as follows:

$$
\begin{aligned}
&if\ f\theta < 0.0 \\
&\qquad if \quad f\theta >= -f\theta_{min} \quad f\theta = -f\theta_{min} \\
&\qquad else\ if\ f\theta <= f\theta_{min} \quad f\theta = f\theta_{min}
\end{aligned}
$$

- the function returns true when the ball is in sight and parameters fx, fy and $f\theta$ are 0.

3.2 Velocity-controlled trajectory to align the attacking robot with ball and target

This section describes the algorithm used in the *UpdateTrajectoryGotoBallAlignedTarget()* function. This function requires a target angle as input, named α_T^{horiz}. Usually this angle is the angle between the robot's longitudinal axis and the opponent goal direction.

The algorithm is similar to the one described previously. In case the robot has detected the ball the following instructions are executed:

- the robot checks if it has arrived close to the ball.
- the robot updates the target angle, namely α_T^{horiz}, as:

$$\alpha_T^{horiz} = 2.\alpha_B^{horiz} - \alpha_T^{horiz} \tag{4}$$

- α_T^{horiz} is saturated between -60 $[deg]$ and 60 $[deg]$. Because the angle is between -180 $[deg]$ and 180 $[deg]$, there is a risk of sideways oscillation when the angle changes sign around 180 $[deg]$. This must be taken into account in the angle saturation to prevent the robot from oscillating.
- if the robot has NOT arrived close to the ball, the strategy for the robot consists of moving along a curve while changing the robot's heading so that it converges to the ball-target direction:
  - if $|\alpha_T^{horiz}| < 48$ $[deg]$ then fx and fy are set in the same way as for tracking the ball only. The angle used for fx and fy is α_T^{horiz}.
  - Otherwise fx and fy are set to 0. Only a rotation motion will be triggered.
  - The angle used for calculating $f\theta$ next is set to ball horizontal angle.
- if the robot has arrived close to the ball,
  - angle difference $\Delta\alpha = \alpha_T^{horiz} - \alpha_B^{horiz}$ is calculated.
  - if $|\Delta\alpha| < 12$ $[deg]$ then fx and fy are set to 0.
  - else
    - $\Delta\alpha$ is saturated between $-\pi/2$ and $\pi/2$.
    - $fx = k_x.\sin\Delta\alpha.\sin\alpha_B^{horiz}$
    - $fy = -k_y.\sin\Delta\alpha.\cos\alpha_B^{horiz}$

    with $k_x = k_y = 1$.
  - The angle used for calculating $f\theta$ next is set to the target angle given as input to the function.
- $f\theta$ is calculated as in section 3.1 with a scaling coefficient $k_\theta = 0.6$.

3.3 Velocity-controlled trajectory to align the defending robot between ball and target

This section describes the algorithm used in the *UpdateTrajectoryGotoBallAlignedTargetForDefense()* function. This function requires a target angle as input. The target point here can be the own goal center. In this situation, the robot must defend its own field so the robot will place itself between the ball and the target, and approach the ball.

The algorithm is the same as in section 3.2, with $k_x = k_y = 1.5$.

4 Obstacle avoidance

The obstacle avoidance implementation is very simple. To avoid obstacles the robot uses its two ultra-sonar sensors (US) located on the chest. There is one sensor on the right and another one on the left, and they point a little bit outwards.

4.1 Ultra sonar configuration

The devices used to get access to US sensor values are the following:

```
    fSensorKeys.push_back(std::string("Device/SubDeviceList/US/Left/Sensor/Value"));
    fSensorKeys.push_back(std::string("Device/SubDeviceList/US/Right/Sensor/Value"));
```

The source code used to enable US sensors using a non-periodic configuration – this means that a request for sensor reading has to be sent – is given below:

```
void MovementModule::EnableUS() {
    int DCMtime;
    //Get Time
    try {
        DCMtime = dcmProxy->getTime(0);
    } catch (const AL::ALError &e) {
        throw ALERROR(getName(),
        "EnableUS()", "Error on DCM getTime : " + e.toString());
    }

    USCommands.arraySetSize(3);
    USCommands[0] = std::string("US/Actuator/Value");
    USCommands[1] = std::string("Merge");
    USCommands[2].arraySetSize(1);
    USCommands[2][0].arraySetSize(2);
    USCommands[2][0][0]= 4.0;
    USCommands[2][0][1]= DCMtime;

    try {
```

```
        dcmProxy->set(USCommands);
    } catch (const AL::ALError &e) {
        throw ALERROR(getName(),
        "EnableUS()", "Error on sending US to DCM : " + e.toString());
    }
}
```

Sometimes however the sensors seem not to work any more. It is possible to disable them and enable them again using this non periodic configuration during the play.

US sensor values are read every 100 $[ms]$ using *GetValues()* method on fast memory access pointer in a post-DCM-thread[3] callback. The post-DCM-thread callback runs every 10 $[ms]$. The configuration for US sensor reading was non periodic. This means that a request for US sensor values is sent in the post-DCM-thread every 100 $[ms]$ after US sensor values have been read:

```
        int DCMtime;
        //Get Time
        try {
            DCMtime = dcmProxy->getTime(0);
        } catch (const AL::ALError &e) {
            throw ALERROR(getName(),
            "callbackEveryCycle()", "Error on DCM getTime : " + e.toString());
        }

        USCommands[2][0][1] = DCMtime;
        try {
            dcmProxy->set(USCommands);
        } catch (const AL::ALError &e) {
            throw ALERROR(getName(),
            "callbackEveryCycle()", "Error with DCM set : " + e.toString());
        }
```

4.2 Dealing with obstacle detection

An obstacle is considered as detected if both sensors give a value greater than 0.1 $[m]$ and if at least one of them give a value less than 0.45 $[m]$. Then to determine whether the obstacle is on the left or on the right both values are compared. If the difference between both values is small, the obstacle is considered to be in front of the robot.

```
void Behavior_robot::obstacleAvoidance() {
    float USLeft = PtrMvt->sensorValues[4];
    float USRight = PtrMvt->sensorValues[5];
```

[3]extract from Aldebaran documentation. DCM stands for *Device Communication Manager*. The DCM is the NAO software module, part of the NaoQi system, that is in charge of the communication with every electronic devices in the robot (boards, sensors, actuators ...) with the only exception of the sound (in or out) and the camera. It manages the main communication line: the USB link with the ChestBoard. [...]. Modules like *Motion* and *Leds* directly send commands to actuators using the DCM, while extractors and other modules use sensor results returned by the DCM in ALMemory

```
    if ( (USLeft < 0.45 || USRight < 0.45) && (USLeft > 0.1 && USRight > 0.1) ) {
     //If the robot is close to an obstacle
        if (USRight < USLeft) { //The obstacle is on the right
            fx = cos(60.0*convert_from_deg_to_rad);
            fy = sin(60.0*convert_from_deg_to_rad);
        } else { //The obstacle is on the left
            fx = cos(-60.0*convert_from_deg_to_rad);
            fy = sin(-60.0*convert_from_deg_to_rad);
        }
        if (fabs(USRight-USLeft) < 0.02) {
        //The obstacle is in front of the robot
            fx=0;
            fy=1;
        }
    }
}
```

To avoid the obstacle when it is on the right or on the left, the obstacle avoidance function overwrites fx and fy with a ± 60 $[deg]$ oblique change of direction.

This strategy was enough to avoid other robots. Of course it would be useful to use vision feedback to detect obstacles more precisely, especially robots lying on the floor. This will be done in the next future.

5 Localization

The localization used was only based on poles. White lines are not used yet in the localization process. The robot detects poles in the image and stores information of distance and horizontal angle for two blue poles and two yellow poles. The respective positions of detected poles relative to the walking robot are updated thanks to the robot's odometry.

When 2 poles of the same colour are stored and a new pole with similar colour is detected, it is compared with the 2 poles already in memory. When 2 poles are detected they are first being compared to check if they can be considered as the same. The comparison consists of calculating the distance between the poles using horizontal angle and distance and applying a threshold distance of approximately one third of the real goal pole inter-distance.

Then 3 cases may arise. The first case is when there is no pole in the memory pool, the second case assumes there is only pole in the memory pool, and the last case appears when there are 2 poles in the pool.

In the first case, the new pole is immediately stored.

In the second case, if the new pole is likely to be the same as the pole already recently stored P_1, the P_1 is replaced by a pole with greatest distance and averaged horizontal angle. If P_1 is too old the new pole replaces P_1. If the new pole cannot be considered as the same as the pole already stored, the new pole is stored as the second pole of the same goal.

In the last case, the new pole P_{det} is compared with the first pole stored P_1. If the new pole is likely to be the same as P_1, P_1 is replaced by a pole with greatest distance and averaged horizontal angle. If P_1 is too old the new pole replaces P_1. If P_{det} is not likely to be the same as P_1, it is compared with P_2 and the same checking takes place. In the case where P_{det} cannot match P_1 neither P_2 three situations may arise: P_{det} will replace P_1, P_{det} will replace P_2, or P_1 and P_2 will remain unchanged. The decision in this case consists of selecting the two poles whose inter-distance is closest to the real goal pole inter-distance, as described below:

- if $|P_1P_{det} - d_P| < |P_1P_2 - d_P|$
- then
  - if $|P_2P_{det} - d_P| < |P_1P_{det} - d_P|$
  - then P_{det} replaces P1
  - else P_{det} replaces P2
- else
  - if $|P_2P_{det} - d_P| < |P_1P_2 - d_P|$
  - then P_{det} replaces P1
  - else P_1 and P_2 unchanged

where d_P is the real distance between goal poles. $d_P = 1.5$ $[m]$.

6 Vision

6.1 L3M Vision Manager

In our previous L3M report [1], we explained the pixel classification we were commonly using. The technique was based on selecting special sets of images based on human expertise. Classification of each pixel colour for each object type was done offline and manually. It required more than one hour to fix all types of classification (including ball, yellow and blue poles and green at least). As colour pixel classification highly depends on lighting conditions that can strongly vary within 10 minutes, the time needed for such classification was a real drawback of our vision system. To accelerate our camera tunning parameters procedure and our colour classification procedure, we developed a new L3MVisionManager (like other teams that are using very attractive calibration interfaces [4, 5]). The goal was to develop a client vision interface, that allows us:

- to tune camera parameters (like telepathe, but where telepathe shows camera outputs with 160×120 unresizable window, we would like to have 640×480 or more, with centering and zooming possibilities),
- to realize on-line pixel colour classification
- to enhance previously defined colour classification (while keeping or modifying camera parameters)

- to see vision-system analysis in real time.

Figure 1 shows the main tabs of the L3MVisionManager. The first tab is labeled `Cam`. It allows us to set camera parameters on a wide screen. The zoom enables a neat view of special details. When the checkbox `set-zoom-center` is checked, each click centers the camera view. By checking the `show-zoom-center` box, the zoom center is shown with a green cross. Zooming in (*respect.* out) is accessible via the press button labeled `zoom-in` (*respect.* `zoom-out`). Zooming functions are also bound to mouse wheel to accelerate zoom handling. By such mechanism, it is possible to zoom in or out in the current view and to move zoom centering in the image. Figure 2 shows a `zoom-in` view with the green centering cross.

The second tab is used to construct the LUT. The previous `zoom-in` and `zoom-out` mechanism is again available. Pixels that are inside a range of three channels values can be selected. Such selection is centered on the last pixel selected by mouse clicking. To increase possibilities of pixels groups construction, a group can be defined with single pixel surrounding and with multiple pixels surrounding. Pixels groups can also be set in one single image or with multiple images. By selecting multiple images pixels group with single pixel surrounding, the selection stretches naturally to colour shapes. This function appears to be very useful because in `kVGA`, colour variations are very fast and catching all pixels types associated with a shape is difficult. As surrounding can be adjusted, it allows the user to fit surrounding values to fulfill a shape. Surrounding values can be grown, reduced, translated to upper or lower values. When applying such modifications to surrounding values, modifications are discretized into steps that are composed with press buttons labeled 1,2,4 and 8. Pixels groups can be added to or subtracted from the current LUT. Five pixel-types, that correspond to ball, blue-goal, yellow-goal, green and white, can be set. LUT can be saved in a binary file with a header that contains all camera parameters. When loading a file that contains a previously defined LUT, it is possible to load camera parameters or not. The resulting colour pixel image can be viewed in the main window (*i.e.* figure 3 top) or in a smaller window on the right while the original appears in the main window (*i.e.* figure 3 bottom).

The third tab presented in Figure 4 shows the resulting interpretation of the scene in real time. Small blue circles show the field's limits, white boxes are places where poles can be, horizontal blue bars show blue pole bases, big blue circles with center show ball detection (knowing that circle's radius corresponds to ball's radius) and yellow values represent the distance from head to the virtual horizontal plane passing through the camera the center, and the distance from head to the virtual horizontal plane passing through the bottom of the current camera view (available only if behavior is active, otherwise it displays null values as in this case).

For the implementation of these new tool functionalities, code is shared between VisionModule (embedded executions) and L3MVisionManager (remote executions, *i.e.* on a personal computer). To view camera information grabbed in the NAO, a new vision-service is started inside VisionModule. Figure 5 shows our client-server architecture where embedded functions are waiting for incoming requests of `getCameraParams`, `setCameraParams`, `getImage` and `getAdditionnalInfos`.

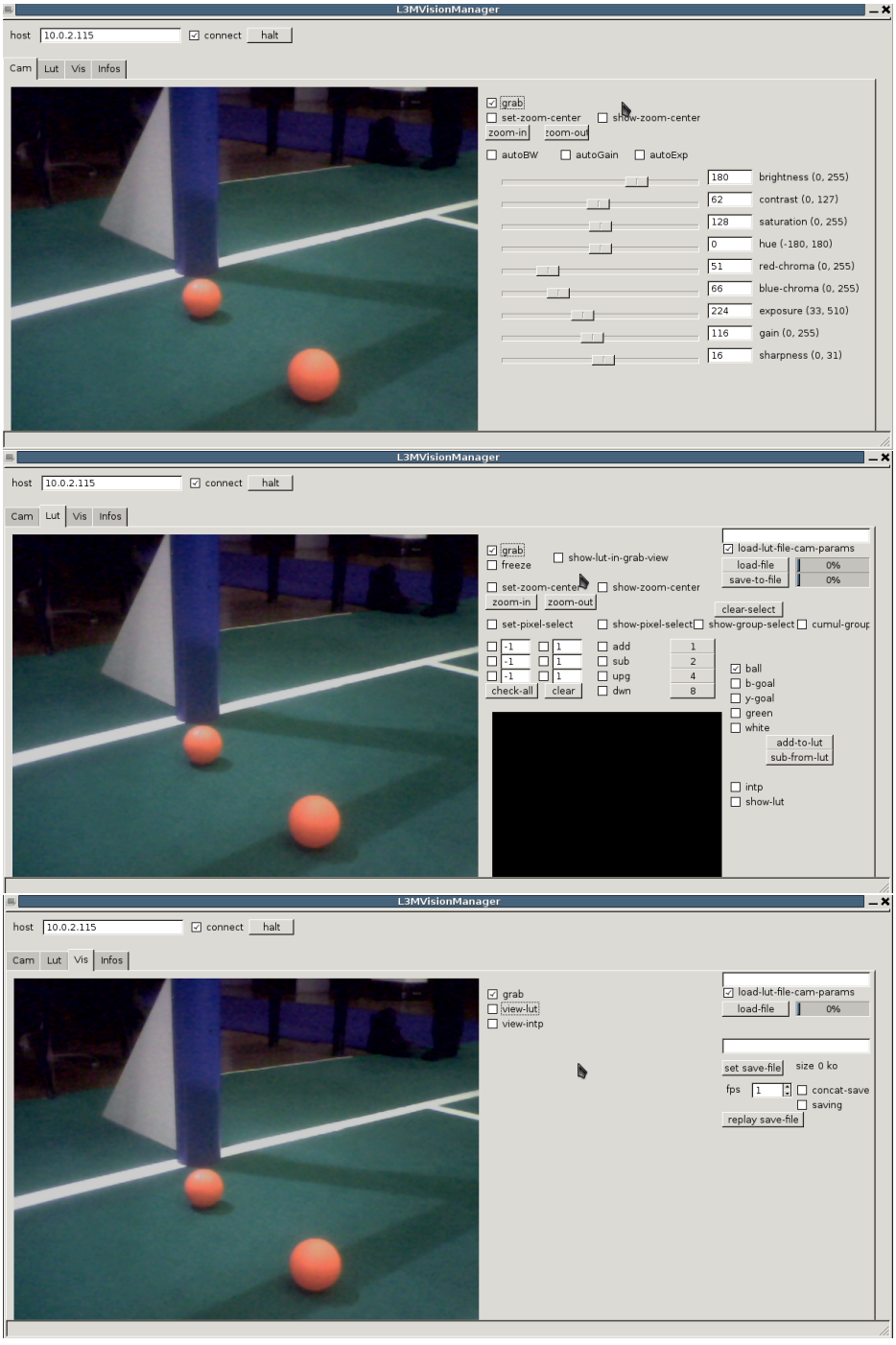

Figure 1: Three main tabs of L3MVisionManager. The first tab shows the tuning camera parameters interface, the second shows the colour classification interface and the third shows the realtime vision system analysis interface. The L3MVisionManager contains four tabs, respectively labeled `Cam`, `Lut`, `Vis` and `Infos`. The fourth tab (not shown here) summarizes logs in the buffered text area.

Such an architecture allows us to minimize the cpu usage of embedded modules and then to de-

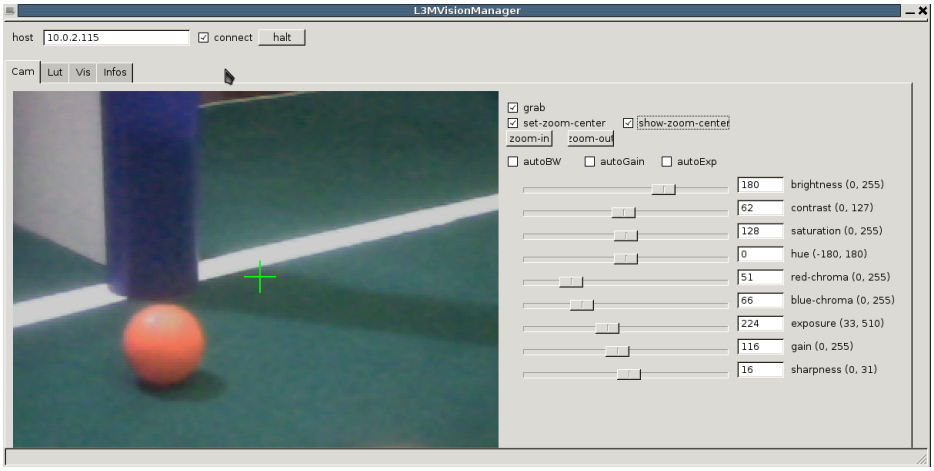

Figure 2: Here is a `zoom-in` view. The green cross shows the center of the view. This cross helps the user centering the view in accordance with its desires. The cross can be displayed or hidden.

crease response time for each `getImage()`. As time for sharing camera images is time consuming, it is possible to simultaneously connect only one client. Clients follow a communication protocol with the vision server that is waiting for requests. As time needed to grab a new image in `kVGA` takes less than 10ms, the transfer of such an image takes approximatively 60ms, and its interpretation presented in figure 4 takes less than 10ms on a personal computer. Clients send a request for a new image every 100ms (fixing the frame rate of L3MVisionManager to 10 fps).

6.2 Vision calibration

According to our previous report [1], we started from the procedure that stands for setting camera parameters, enhanced with sharpness (that is set to 2 to help distinguish objects clearly, and minimize objects distortions regarding LUT characteristics). The whole parameters set is saved in the LUT-file generated with our L3MVisionManager. At each start, the VisionModule loads this LUT-file and sets all camera parameters. This procedure allows us to link camera parameters automatically with LUT characteristics that define pixels included in the LUT.

6.3 Vision processing

6.3.1 Choosing color space

As in our previous version, the *vision* module is in charge of objects identification, that can be retrieved by other modules. Again we timed the *getImage()* that has greatly changed with the new naoqi version (now v1.6.0). Tables 2 and 3 show averages of camera's data access while using naoqi functions. For each column, the left part shows average, and the right part gives standard deviation. Classical color space are presented where `hsy` is an embedded version mixing `hsv` and `hsl` color spaces and where `yuv422` is the only native color space.

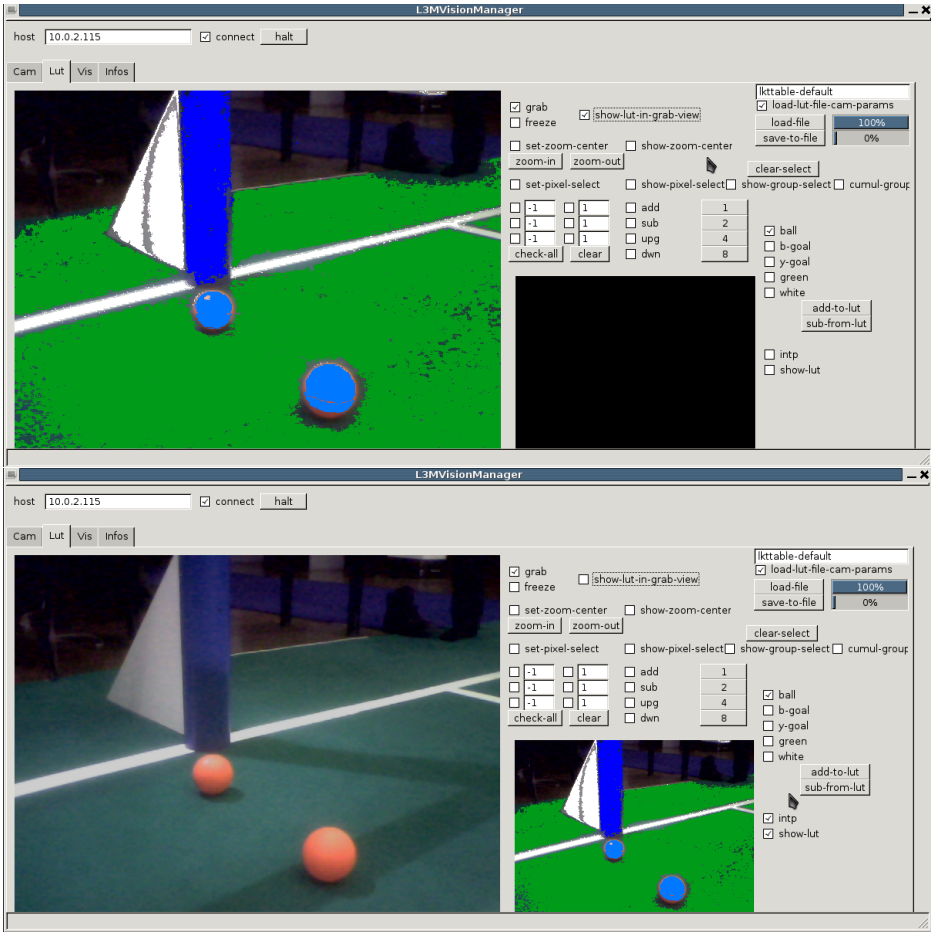

Figure 3: The resulting colour pixel image can be displayed in the main window (see top view) or in the right window (see bottom view).

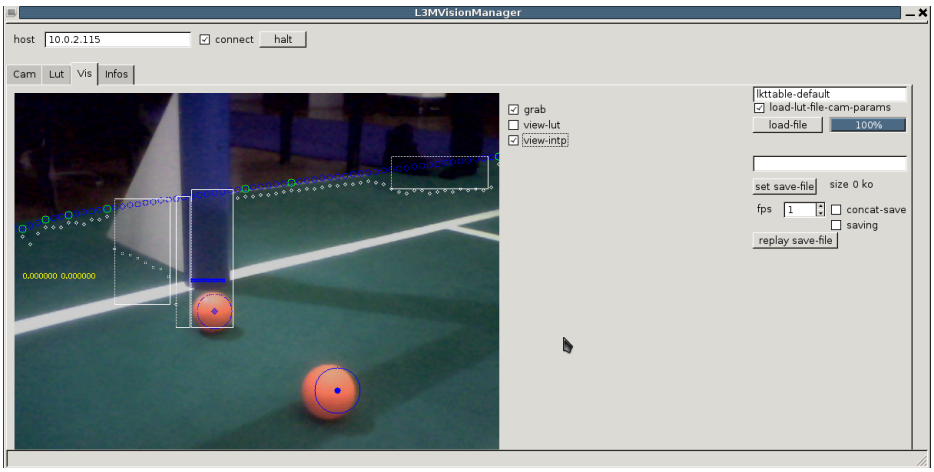

Figure 4: The resulting interpretation of the scene appears in real time.

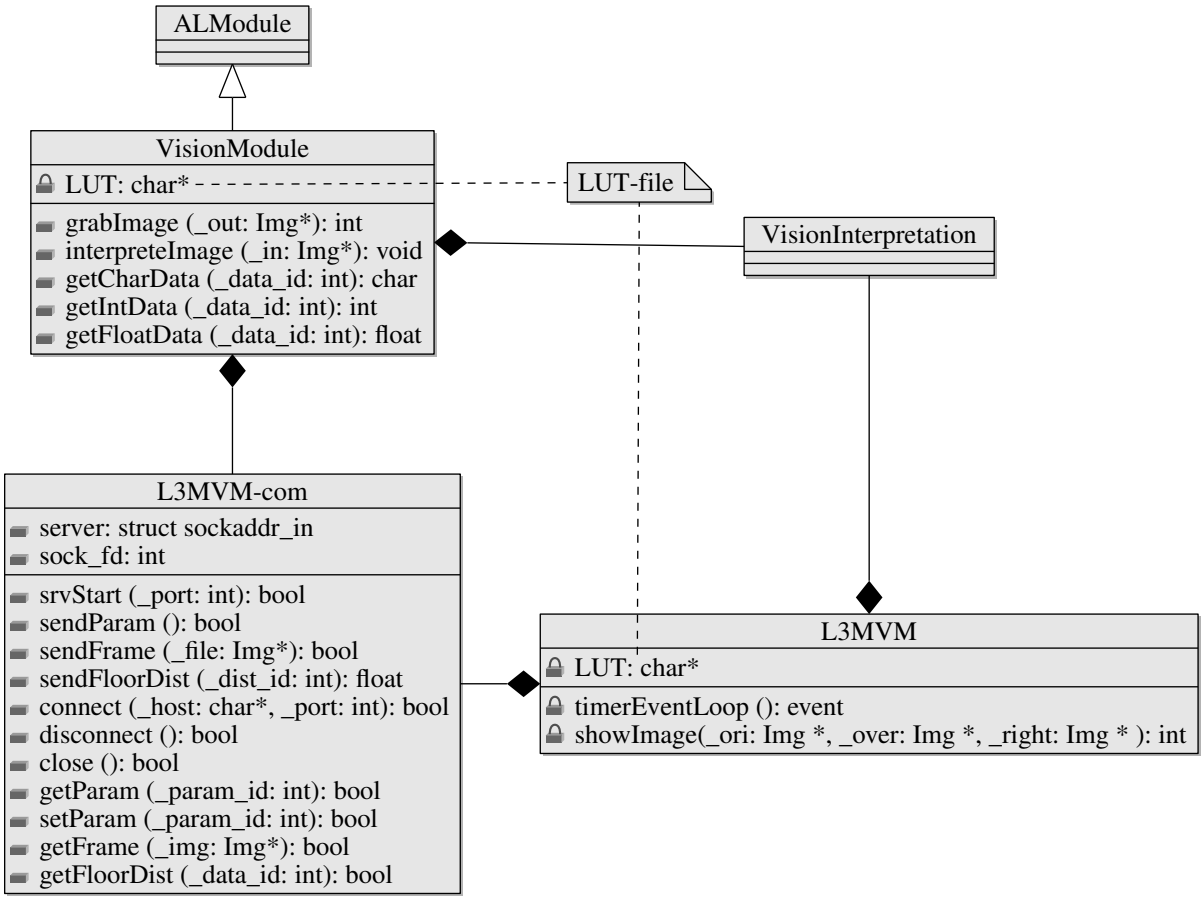

Figure 5: The L3MVM architecture makes it possible to share code between embedded vision module (*i.e.* VisionModule class) and IHM vision manager interface (*i.e.* L3MVM class). The L3MVM-com class is dedicated to unify communications between client and server. It implements all client and server functions to set parameters and test vision functions. The VisionModule is using server functions when L3MVM uses the client parts. VisionInterpretation class contains all interpretation treatments. It is used in both client and server parts. Such a mechanism reduces the time needed by L3MVisionManager to show interpretation results, by minimizing embedded workload. In L3MVM class, the timerEventLoop automatically and periodically generates one getFrame event that leads to display the received image. Depending on VisionInterpretation functions requested, this image is displayed after treatment and interpretation. Such an architecture also needs the same LUT file in both client/server parts (*i.e.* L3MVM/VisionModule).

Table 2: times in ms to get, copy and release a `kVGA` image using main color spaces

color space	*getImage()*		copy		*releaseImage()*	
`yuv422`	< 0.001	+/-< 0.001	0.230	+/-0.077	0.641×10^{-3}	+/-0.215×10^{-3}
`yuv`	0.055	+/-0.019	0.052	+/-0.019	0.053×10^{-3}	+/-0.053×10^{-3}
`hsy`	0.147	+/-0.049	0.048	+/-0.016	0.047×10^{-3}	+/-0.047×10^{-3}
`rgb`	0.095	+/-0.031	0.049	+/-0.016	0.042×10^{-3}	+/-0.042×10^{-3}

Table 2 presents basic operators of image processing. The first and the last operators (*i.e.* `getImage()` and `releaseImage()`) are directly called from naoqi. The operator named copy

is transforming input colors into CLUT entries. For `yuv422` color space, this means that data are unpacked and transformed. Table 2 shows that `yuv422` is the fastest but also shows that copy operator is highly time consuming. For all color spaces, `releaseImage()` is negligible even if it takes 3 times more for non native color spaces.

Table 3: times in ms to grab `kVGA` image using main color spaces

color space	grab with copy		grab without copy	
`yuv422`	0.231	+/-0.077	0.001	+/-< 0.001
`yuv`	0.107	+/-0.038	0.055	+/-0.019
`hsy`	0.195	+/-0.065	0.147	+/-0.049
`rgb`	0.144	+/-0.047	0.095	+/-0.031

Table 3 shows the time difference between simple image grabbing and image grabbing with information buffering to CLUT entries space. It shows that the buffering process takes approximatively 50ms except for the `yuv422` color space that takes around 4 times more. This lead us to use the `yuv422` color space without buffering.

6.3.2 Image interpretation

Based on a pixel classification, the image interpretation is done by subsampling a `yuv422` image to obtain an acceptable processing time (*i.e.* close to $20ms$). Our vision process achieves a vertical segmentation directly from the `yuv422` image (similar way to other teams like [3] that wisely uses such segmentation). The process is divided in different steps and a subsampling grid is defined for each step.

- first, we set the convex hull of the soccer field (*i.e.* based on the green shape identified by green entries in the CLUT). Then the convex hull is compared to the green shape to define possible poles in the image.
- second, inside the convex hull, ball colour shapes are collected thanks to vertical segments, and according to a threshold, segments are regrouped to define multiple ball regions. Each ball region is sampled to be ordered in a list by size with others. Each ball region is associated with its ball pixel density, moments for shape identification and pixel-types surroundings informations. With all these informations, each ball region can be subclassed in the ordered list. The best ball region is considered as the real ball in the image.
- finally, possible poles boxes are subsampled. Each possible pole is associated with a pole colour density, moments for shape identification, pole's tilt and surroundings informations of their base. The best pole is considered as the real pole in the image. The process is able to select two poles of the same colour, identifying only one pole if there is one in front of NAO and two poles if there are two.

This process is parametrized by four subsampling grids : the first sets the green subsampling, the second sets possible ball regions subsampling inside the green region, the next sets each ball region subsampling to fix each ball region characteristics and the last sets each possible pole subsampling to fix each pole's characteristics.

Table 4: subsampling times in ms while identifying heights of field pixel-types

subsampling	*1*		*2*		*4*		*8*		*16*		*32*	
1	193.0	2.6	108.7	20.5	48.1	1.1	23.7	< 1	11.7	< 1	5.8	< 1
2	96.3	1.3	51.0	4.7	23.8	< 1	12.0	< 1	6.0	< 1	2.8	< 1
4	48.4	< 1	23.6	< 1	11.9	< 1	6.1	< 1	3.0	< 1	1.6	< 1
8	24.2	< 1	11.8	< 1	6.1	< 1	3.1	< 1	1.5	< 1	<1	< 1

Table 5: subsampling times in ms while computing the convex hull of field pixel-types

subsampling	*1*		*2*		*4*		*8*		*16*		*32*	
1	210.4	16.1	95.1	7.5	47.9	7.5	24.1	10.1	14.5	4.2	7.1	2.9
2	98.6	8.8	47.9	5.1	24.2	6.6	12.2	7.7	7.4	2.6	3.1	< 1
4	49.5	8.6	24.2	7.5	12.3	7.7	7.6	27.2	4.3	2.3	2.8	2.2
8	25.2	12.0	12.5	9.7	6.3	8.5	3.2	3.8	1.7	< 1	<1	< 1

Table 6: subsampling times in ms while computing moments of each pixel-types

situation	*1*		*2*		*4*		*8*		*16*		*32*	
first	200.1	2.4	100.5	5.3	58.5	4.7	26.9	4.9	12.2	< 1	6.2	< 1
second	228.7	18.3	104.8	1.0	52.3	1.2	26.3	< 1	13.2	< 1	6.3	< 1

Table 7: subsampling times in ms for vertical ball segmentation inside field convex hull

subsampling	*1*		*2*		*4*		*8*		*16*		*32*	
1	186.2	< 1	93.1	< 1	47.0	1.2	23.5	< 1	11.5	< 1	5.7	< 1
2	186.4	< 1	93.0	1.0	46.6	< 1	23.4	< 1	11.8	< 1	5.8	< 1
4	202.1	19.2	94.5	3.7	46.8	< 1	23.2	< 1	12.3	< 1	5.8	< 1
8	186.2	1.2	106.7	8.5	46.8	< 1	23.4	< 1	11.7	< 1	5.7	< 1

Table 8: subsampling times in ms while computing density of each pixel-types

situation	$(1,1)$		$(1,2)$		$(2,2)$		$(2,4)$		$(4,4)$		$(4,8)$	
second	211.5	< 1	104.2	1.3	52.2	< 1	26.3	< 1	14.5	3.2	7.7	3.4

Table 4 shows the time spent to extract the green pixel-type heights in a `kVGA` image. First column (and *respect.* first line) indicates the vertical (*respect.* horizontal) subsampling value used. A value of 2 means that half columns (or lines) are considered. Table 5 shows the additional time needed to compute the convex hull of green pixel-type heights. In Tables 4 and 5, the first column shows average time, and the second column gives standard deviation. Based on these results, we chose subsampling $(8, 2)$ so that 1 in 8 columns and 1 in 2 lines are scanned.

Table 6 shows times to compute inertial moments of each pixel type in the entire image. It shows that such an operation is time consuming for two classical situations: the first that contains only one ball and the second that contains one ball and one pole. In this case, moments are computed for each pixel type, which is useless (at least for the green parts). Situations `first` and `second` contains close objects that gives us computing time needed for a classical view with close objects

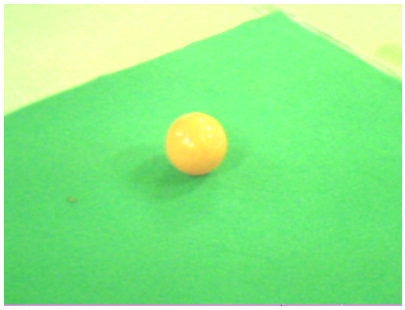

Figure 6: A simple ball on the green field.

(like in Figure 6). Results show that such operations have to be reduced to local area that are candidates to shape identification like a ball and a pole. Our previous $(8, 2)$ choice is maintained and puts us in the 8th column of this table. These times indicate an upper bound of the time needed to compute ball moments.

Table 7 shows times needed to compute the convex hull of the green and to regroup ball vertical segments inside the hull. As segments are checked between their neighbors vertically, it shows that a vertical subsampling is inefficient. The horizontal subsampling takes less columns into account, so generates less segments and have better results than the vertical one. Again, staying in the 8th column is fair enough. $(8, 1)$ is preferred to $(8, 2)$ and others whereas a vertical subsampling of 1 is more precise that others.

Table 8 shows times needed to perform all treatments, including the final step, that subsamples possible poles boxes. Using the first situation makes the same results as table 6 (logic to a situation without pole). The subsampling mentioned here is fitted on the rectangular shape of axis aligned boxes. Finally, subsampling poles with $(4, 4)$ is considered as admissible.

Considering all these subsampling parameters, an important question is to know the quality of the resulting image processing provided. For each step, we have chosen the most admissible value with the better possible quality. The final time of 14.5ms (with more or less 3.3ms) gives us a maximum time needed for image processing. Figure 4 shows the result of such processing, where poles boxes candidates are notified by white rectangles, pole bases by horizontal blue segments and balls by big blue circles.

7 Individual behaviors

The robot can search for and detect ball and poles, attack the opponent goal and defend its own goal by turning about the ball. It can kick the ball more or less quickly.

The robot starts to look for the ball by moving its head then by turning in place. Once it has found the ball the robot goes to it. When only the ball has been detected, the robot follows the trajectory described in section 3.1. When one opponent pole has been detected the robot will curve its trajectory to align its longitudinal axis with the ball and the opponent pole (see section 3.2). When two opponent poles have been detected the robot will curve its trajectory towards the line joining the ball and the goal center – middle of the two poles, this is done when the robot

is not too far away from the ball. Once arrived at the ball if the robot has two opponent poles stored in memory it switches to the kicking procedure. If it has only one opponent pole in sight or none in sight it starts to scan with its head to look for opponent poles. After scan if there is no pole the robot turns about the ball by 90 [*deg*] and start head scanning again. If there is only one opponent pole, the robot aligns itself with the ball and the pole, then the kicking procedure is triggered.

The kicking procedure first consists of looking down at the feet to check if the ball is close enough. If not the robot moves to get closer. If the robot is enough close to the ball, the lateral distance between the ball and closest foot to the ball is calculated (this is done in pixel units right in the image), and the robot moves sideways to place its closest foot in front of the ball. The robot is then ready to kick the ball. The kicking motion was defined off line. It cannot be interrupted. The kick is made slow when the opponent goal is far away from the robot, this is to avoid to kick the ball out of the field. When one opponent pole or the opponent goal is enough close the kick is made fast.

If the robot does not detect any opponent pole but detects an own pole the robot adopts a defending strategy. It goes to the ball and once close enough it starts to turn around the ball to place itself between the own pole and the ball. No kick is triggered after positioning, the robot looks for the opponent goal by scanning the environment with its head.

The goalkeeper behavior is an adaptation of the defending player. When the ball approaches the own goal too much the goalkeeper goes to the ball and places itself between the ball and its goal. It was decided not to trigger any kick to avoid kicking the ball out of the field. Maybe it was a bad choice. However this strategy proved useful as it had prevented our team from having too many goals scored into our own goal.

8 Collective behaviors

The collective behavior used in the ball passing challenge is defined by the following procedure. At each time step, each player defines its actions and its next state according to its current state value. Initially, all players are in INIT state.

Such a procedure is tedious in more complex collective situations. Even for this simple one, it is hard to tune. To use them during the match, collective behaviors must demonstrate positive rewards against individual behaviors. In the ball passing challenge, it is a duty to use such collective behavior. Thus, we restricted collective behaviors to the ball passing challenge.

```
collectiveBallPassingChallenge ()
  if state is INIT
    state = closestFromTheBall ( delta, self )
  else if state is KICKER
    gotoball and kick the ball towards other
    if a kick just ended :
      send KICK to other
      state = ASSISTANT
  else if state is ASSISTANT
    if state ( other ) differs to KICKER
      state = INIT
    else if KICK message just received
      wait for 3 sec
      state = KICKER
```

In the `collectiveBallPassingChallenge` procedure, all possible situations are summed up in three cases : `INIT`, `KICKER` and `ASSISTANT`. In `INIT` state, each one has to define who is the closest to the ball depending on more or less `delta` distance value, where draws are solved by `self` id value (*i.e.* the smallest `self` id become the closest NAO to the ball). The function `closestFromTheBall` compares all distances from the ball and returns `KICKER` for the closer and `ASSISTANT` for the other. This procedure can be easily extended to more than two players by modifying the condition `state ( other )` by the condition `state of one other`. To simplify implementation, we used a `wait for 3 sec` function. This last one can be replaced with a wait for a ball stop. This first solution of collective behavior shows that further developments have to be done in the area.

9 Future improvements

9.1 Solving shortcomings

The first developments will aim to solve shortcomings of the current software implementation. This means:

- better detection of ball when it is partly obstructed, especially when the ball is at the feet.
- more reliable detection of poles to avoid false positives. False positives of blue poles inside blue sweatbands must be avoided.
- quicker procedure for searching for the ball involving specific and adapted head movement and walking displacements for field exploration.
- better placement around the ball when it comes to turn about it by angle close to 180 $[deg]$.

9.2 More important developments

The main next developments will focus on setting up a distributed architecture for the robots to exchange roles dynamically. Developments will also deal with improving vision algorithms, and localization algorithms using line detection.

- intercommunication between robots, with dynamic role exchange. This involves the implementation of a distributed network architecture
- enhance scene interpretation with colour group detection
- localization based on line detection, application to goalkeeper positioning with respect to goal area limits
- better collision avoidance
- increase walking speed
- improve locomotion stability by keeping balance in case of collision or walking across irregular terrain
- improve kicking motion stability by keeping balance or interrupting it.

References

[1] RoboCup 2009 SPL French L3M team report.

[2] Nicolas Jouandeau, Patrick Bonnin, Vincent Hugel and Pierre Blazevic. "From color groups to scene interpretation". *4th Workshop of Humanoid Soccer Robots, 2010*

[3] B-Human Team Report and Code Release 2010 (*and private conversations*).

[4] NAO-Team Humboldt 2009 (*and private conversations*).

[5] Robocup 2009 TeamChaos Documentation (*and private conversations*).