Dutch Aibo Team: Technical Report RoboCup 2005

Jürgen Sturm [a] Arnoud Visser [a]
[a] *Universiteit van Amsterdam*

Niek Wijngaards [b]
[b] *DECIS Laboratory / Thales Research & Technology Netherlands*

http://www.dutchaiboteam.nl/

Abstract. In this report we describe the algorithms implemented by the Dutch Team in the Sony 4-Legged League. In particular, we describe the new modules on the Aibo side (self-localization, role switching policies, vision) as well as some useful tools we created for the developer side (storing binary snapshots, improved color table creation and performance measurements). Our approach in the variable lighting challenge is explained, which allows to compensate for a wide range of varying lighting conditions. The range of lighting conditions during the challenge was too small to test our algorithm to the extreme, we expect that with this algorithm we can play soccer both indoors and outdoors. This component is still under ongoing development and we hope that next year we can demonstrate fully autonomous self-configuration under arbitrary lighting conditions.

1. Introduction

In the second year of participation of the Dutch Aibo Team we started with merging our 2004 code base and the German code from 2004. Due to the quality of the German documentation [8] and code our ground for this year's improvements are in practice completely based on the German code release from 2004. This allowed us to concentrate our resources on new developments (bearing in mind our small and constantly changing team composition: 20 second-year undergraduates, 2 graduate students, no post-graduates or post-docs).

Outline The following gives a brief summary of the contents of this report: In section 2 you find our modifications and improvements for the soccer code of the modules which run on the Aibos. In section 3 we explain the descriptions of the extensions we added to the developer debugging application (called RobotControl). In section 4 you find the description of our approaches to master the technical challenges, especially the variable lighting challenge in subsection 4.1. Finally we give a brief discussion in section 5 and end with an outlook for next year's participation and our ongoing research.

2. Solutions for Playing Soccer

The mission at the RoboCup of the 4-legged robots is to play soccer. The German Team has split this mission into several tasks. For each task different solutions can be invented. Each solution corresponds with a *module*, and a number of *modules* form an information processing network that plays soccer. The design of the Germans allows switching between modules that are solutions of the same task at runtime. We slightly adapted many of the original modules to comply with the new rules, as illustrated in figure 1. We included the new field dimensions and beacon positions, rebuilt the field lines lookup tables, implemented the new game controller protocol and adjusted the behaviors.

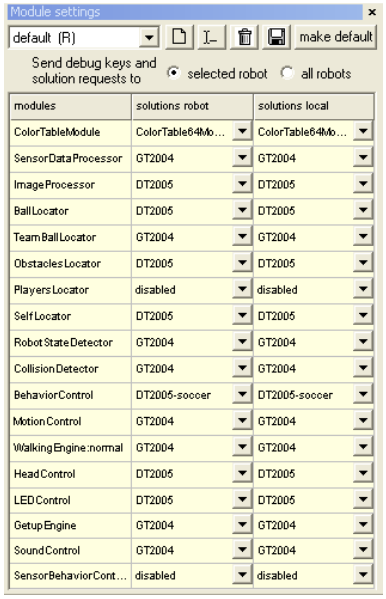

Figure 1. Screenshot of the 'Module settings' toolbar of RobotControl with the modules modified by the Dutch Team.

2.1. Dynamic Roles allocation

The original role switching policy was not robust: firstly, with the WLAN turned off the two attacking robots ended up in the same behavioral state and kept playing in exactly the same manner, and as of then they walked next to each other and pushed each other. Secondly, even with the WLAN turned on, critical roles were not assigned uniquely and it also happened frequently that two robots were in striker mode where they obstructed themselves mutually. Therefore we designed and implemented a new dynamic role allocation module, which reduces the number of striker conflicts considerably.

Before the computation starts, a number of flags and variables are evaluated. Firstly, it is determined whether or not the wireless link is working. The link is considered to be broken when for more than 5 seconds there is no other team player broadcast received. Secondly, it is determined whether the player is on the own field half or not. Thirdly, the estimated time to reach the ball is computed, also taking into account the angles between the player, the ball and the opponent goal[1].

[1]Because even if one player is closer to the ball than another but stands between the ball and the opponent goal it would not be wise to assign him the striker role.

Without WLAN Without an active wireless link the new policy encourages that the players try to stay in their innate roles, but whenever the ball comes closer than a certain threshold, they become temporarily striker. This is easy to implement for the second and fourth player (innate roles defensive and offensive supporter) because they have distinguished places on the field, but player three may change from field halve and should play wherever its support is required more. Therefore, player three has innate role defensive supporter when playing on its own half, and offensive supporter when playing on the opponent's half. Player three leaves on the opponent's half the initiative to player four, because this player only takes over the striker role when the ball is close (1m), whereas player four already feels responsible when it sees the ball from further away (2.5m). When the ball is kicked away, all players return to their innate role. This leads to a balanced team play (at least less obstructions occur compared with the original policy).

With WLAN When the wireless link is functioning, roles can be assigned smarter. When a robot doesn't see the ball, it decides (on the basis of the transmitted robot poses of its teammates) whether it is the hindmost player. If this is the case, it becomes the defensive supporter (meaning that the robot will walk back to the own field half). Otherwise the offensive supporter role is assigned to it. It is now temporarily possible that two robots have the offensive supporter role.

When, on the other hand, the ball is seen, each robot mentally assigns the striker role to the robot that is closest to the ball (based on the estimated time to reach the ball). Then the hindmost player is assigned the defensive supporter role and when there is another player remaining[2], it is assigned the supporter role (which probably makes it walk to the opponent's field half).

2.2. Behaviors

We slightly adapted many of the original behaviors to comply with the new rules. The most radical modification we like to mention is the rewritten goalkeeper behavior (see the red states in figure 2) and a new kick (the smashKick).

We wrote an odometry test behavior that basically makes the robot walk 4m straight ahead and subsequently turn 180 degrees. By this means we discovered that almost all our robots had a (heavy) deviation to the right when walking straight on, so we tried (but without success) to use the self-learning walking-parameter engine of the German Team, then tried (and failed again) to write a manual odometry error compensation. Finally we had no choice but to occasionally let the robot recover from bad odometry updates (around every 10 seconds, depending on the situation) by adding additional re-localization phases to the behaviors that, of course, cost expensive additional time while playing.

2.3. Self-localization

We started adapting the self-locator to the new rules (especially the new field size), which took us a long time because of our unfamiliarity with the lower levels of the GT2004 code. With the aid of the Statistics toolbar (described in section 3.1) we finally succeeded to have the module running.

During the debugging process several interesting new ideas came up, which now could be tested against the original algorithm. Our first hypothesis was that the localization error would decrease when the estimated distance to a landmark was used in the observation model. Our second hypothesis was that convergence would be faster if multiple subsequent landmark percepts could be used in the *resampling* stage. Our third hypothesis was that the precision would increase if the final pose estimate is checked against the odometry, to prevent sudden jumps.

[2]In a four-player game there should of course always be one last player, but just as well it could be that it is currently penalized and taken out of the game. So the question remains to which of the two remaining roles (offensive/defensive supporter) you want to give priority. We chose to play defensive, always assigning the defensive supporter role first.

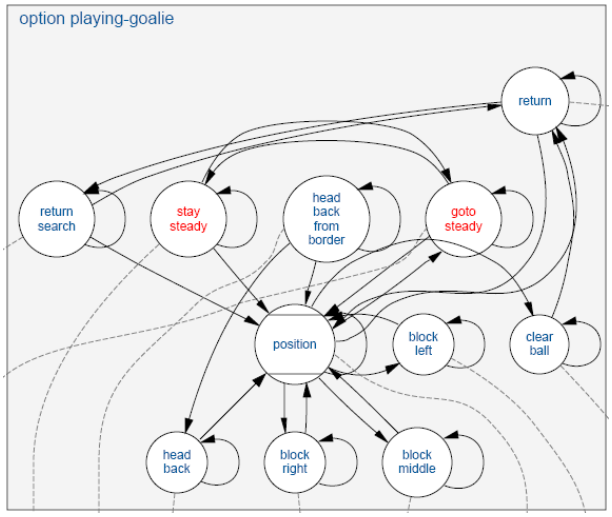

Figure 2. Option *playing-goalie* implements the Dutch goal keeper behavior.

Distance in the Observation Model We included the estimated distance to the landmarks in the observation model by calculating the quality $q_{measured}$ by the following Gaussian similarity formula:

$$q_{measured} = s(distance_{measured}, bearing_{measured}, distance_{expected}, bearing_{expected}) = \exp^{-25(1-b)^2 d^2} \quad (1)$$

$$\text{where } d = 1 + |\frac{distance_{expected} - distance_{measured}}{distance_{expected}}| \text{ and } b = |\frac{|bearing_{expected} - bearing_{measured}| - \pi}{\pi}|.$$

This formula is equivalent with equation (3.29) from [8], when $d = 1.4$. The quality $q_{measured}$ of each observation of a landmark is used to update the running estimate $quality_{landmarks}$ according to the equation (3.33) from [8]. The probability p^i of the particle i of the particle filter [2], representing a hypothesis of the posterior robot pose, is calculated as the product of four independent quality estimates:

$$p^i = q^i_{fieldlines} \cdot q^i_{border} \cdot q^i_{goallines} \cdot q^i_{landmarks} \quad (2)$$

The q^i_{border} no longer is the running estimate of the quality of measurements of the white wall of previous competitions, but of the measurements of the border outside the green field.

Landmark Flashback Buffer When the sight on the field is bad (for example due to bad calibration, people running around, after a kidnap or while a game is running) it takes unnecessarily long time before the particle filter converges to the right position. This is because of the stochastic nature of particle filtering: whenever a landmark is detected all unlikely particles on the field are removed while likely particles are duplicated. However, if there are no subsequent landmark percepts within a few seconds, the position is lost again. A solution to this problem is to store seen landmarks and decrease their reliability in a few seconds. We adjusted the parameter in such a way that roughly every landmark is used twice.

So, instead of using only the current observation y_t at time t, we maintain a small buffer for each type of landmark, with the N latest observations y_f and their corresponding time frames f. Each observation y_f has a probability of

$$p(t, f) = Percept_{Change} \cdot Percept_{Decay}^{(t-f)} \tag{3}$$

to be used to update the running estimate $q_{landmarks}$. The constant $Percept_{Change} = 0.15$ prevents that recent but not subsequent observations dominate the running estimate. The constant $Percept_{Decay} = 0.99$ takes care that an observation is forgotten after a few hundred frames.

Pure Odometry Parallel Filter Probably also because of the stochastic nature of the particle filter and wrong perceptions the particle filter occasionally diverges and suddenly jumps around over the field. Although in most cases the robot finds its location back within a second, this can lead to undesired behavior. Moreover, these outliers can easily be detected because in such situations the computed robot-pose validity is extremely low compared to earlier readings[3]. The validity of the robot pose is calculated with equation (3.38) from [8]. Whenever the validity of the pure odometry filter drops below the validity of the particle filter it is reinitialized to the robot pose and the validity of the particle filter (when everything is fine this is what happens in almost every frame). When the validity of the pure odometry filter is better than the particle filter, the own robot pose and validity are kept, although the latter drops to zero in seconds. Whichever filter produces the robot pose with the bigger validity is chosen to supply this frame's robot pose.

Results The original GT2004 module, adapted for the field size, achieved an average localization error of 21.3 cm and 8.27 degrees. The average kidnap recovery time was 4.97s (σ=4.45s).

Taking advantage of the distance information, the precision of the self-locator module increased to a 14.3 cm average error. The recovery time after a kidnap dropped down to 2.16s.

This flashback buffer led to an increased kidnap recovery speed, especially when seldomly visible landmarks were detected. The kidnap recovery time dropped to 2.09s (without taking advantage of the earlier mentioned new distance-to-landmark likelihood estimation), but only slightly decreased the average positional error: 17.3 cm.

With the odometry filtering turned on, we achieved the best average error of only 12.8 cm and 4.64 degrees.

Using all three extensions to the original algorithm simultaneously, both the precision and recovery time do not further increase; we obtain an average error of 13.5 cm[4] and 5.02 degrees. The average kidnap recovery time was around 2.14s with a standard deviation of 0.71s. This means that with these modifications we achieved a very stable self-localization, without impairing the reaction time or quality to kidnaps.

In the appendix a table with the actual measurements of these experiments can be found.

3. Tool development

On the developer side we extended the German RobotControl application. New components could easily be added to this framework because of its remarkably modular design. The benefit of this approach, compared

[3]However, this could of course also be the sign of a kidnap, but keep in mind that the particle filter needs some seconds to recover anyway and become stable, so it's after all not so bad to believe for a short period of time in the "blindly" updated position until switching in one step to the new position.

[4]The attentive reader notices that odometry filtering alone already achieved a lower average error. This is unfortunately due to occasional noise in the measurement procedure (e.g., sometimes the robot recognizes slightly more landmarks than in another run). More reliable numbers can be gained by repeating the measurements several times.

to many separated PC-tools, is that all components may then directly communicate with the Aibos both on the field or the simulated in RobotControl or even only listen to the playback of earlier-recorded logfiles.

Although RobotControl already incorporates a lot of tools, new developments on Aibo modules can only take place in a controlled way when a corresponding developer window is created to log the progress. We found it useful to add a few tools to extend its functionality.

3.1. Statistics Toolbar

Our biggest problem this year was the poorly functioning self-locator module on the Aibos. Partly this was due to the transfer of our code after a month's effort from one student project to the next. The first project tea, [5] adapted the source code for the new field dimensions at the beginning of this year, but was not satisfied with the self-localization results and indicated missing dependencies. Unfortunately, the search for these dependencies progressed slowly, because the code was functioning at that moment, but due to the lack of measurement tools it was unclear how poorly. Improvements on the code weren't progressing, until the creation of the tool for measuring and testing the performance of the image processor (producing the landmark percepts) and the self-locator module that integrates them into the filtered estimation of the robot pose.

We have created a statistics toolbar that reads the position of the Aibo supplied by an external source. The statistics toolbar is independent of the source, so this could be originating from any reference system. This can be, for example, the robot poses supplied by the simulator (a.k.a. the oracle), or pre-defined points distributed over the field.

The German Universities have already build reference systems that can be used as independent sources, but details are scarcely given. For instance, [7] relies on a laser range finder that detects a pipe attached to the Aibo's back to reproduce the experiments described in [4]. At Darmstadt [1] a ceiling camera is used that detects and tracks the positions of multiple robots simultaneously. Both methods rely on the availability of additional hardware and have probably never been tested outside the laboratories where they were developed. A third possibility is to use a special image processor and self-locator that can determine the robot pose using a check board pattern on the wall, but constantly switching modules is not a neat solution and moreover, the check board can only be seen from a restricted area on the field [3].

The statistics toolbar can use any reference system to test the performance of the perception and the localization. Good results have been gained with a simple set of 10 fixed locations on the field, as illustrated in figure 3. The robot is moved manually from position to position once every 60 seconds, while the statistics toolbar does its analysis.

With every message from the image processor the statistics toolbar calculates the set of landmarks (see figure 4) that the robot should see (completely visible within the calculated viewing area) as well as landmarks, which are definitely invisible (lie completely outside the viewing area). The viewing area is generated by looking at a rectangle with a dimension slightly bigger than the landmark. This rectangle is placed orthogonal to the viewing direction, at the location that the reference system indicates. This procedure is equivalent with the generation of the area with grids of scan lines for the perception modules (see Section 3.2.1 in [8].

This information is compared to the actual perceived landmarks and counters are calculated to track the number of true positives, true negatives, false positives and false negatives. Receiving perceptual and positional information packages from the robots is by nature completely asynchronous, meaning that we can never assume that both perceptual and positional information arrive at the same point in time. Therefore we developed a ring buffer that primarily holds all incoming information for several seconds. Whenever the

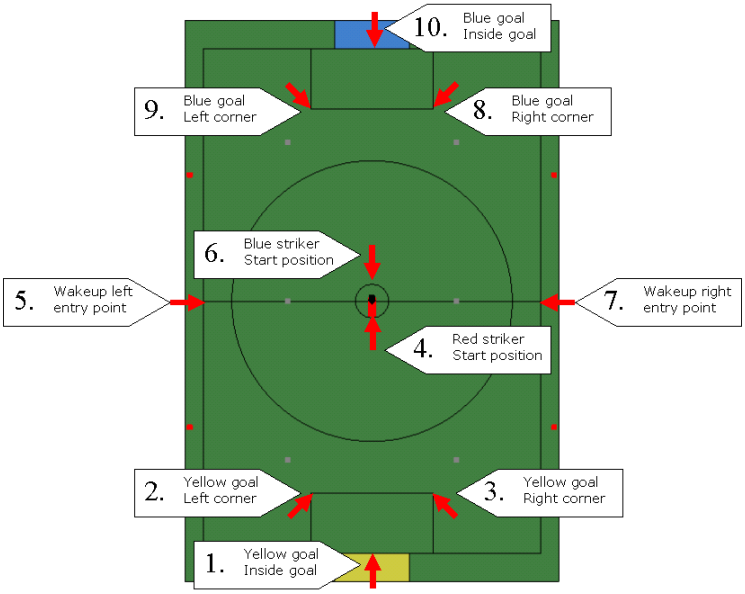

Figure 3. 10 measurement positions on the field.

dialog bar is requested to repaint (more or less once in a second), a computationally cheap update algorithm spreads information[5] from the oldest (in the meantime) modified frame to the most recent one.

The amount of false positives is dangerous for the performance of the self-locator, while the ratio of true positives to the sum of true positives and false negatives gives an indication of the landmark recognition rate of the image processor.

On our last official test before the championship with the 10 pre-defined points we achieved the following performance output. The measurements at each point started after a 20 seconds kidnap recovery time.

```
Flags:
  precision visible flags: 26% (80 FN : 29 TP)
  precision invisible flags: 99% (1 FP : 1566 TN)
Yellow goal:
  precision visible goals: 67% (11 FN : 23 TP)
  precision invisible goals 99% (2 FP : 389 TN)
Blue goal:
  precision visible goals: 75% (9 FN : 27 TP)
  precision invisible goals 99% (2 FP : 374 TN)
Selflocator:
  average error: 63.4655 mm, 2.93795 deg
```

While the flag recognition rate is rather low (probably due to the blur of the fast moving camera in combination with the small size of distant flags), goals are recognized at a high rate (probably because of their big size and easy structure). Fortunately the false positive rate is extremely low.

[5]Camera matrices, true position of the robot, statistic counters (capturing information like true/false positives/negatives).

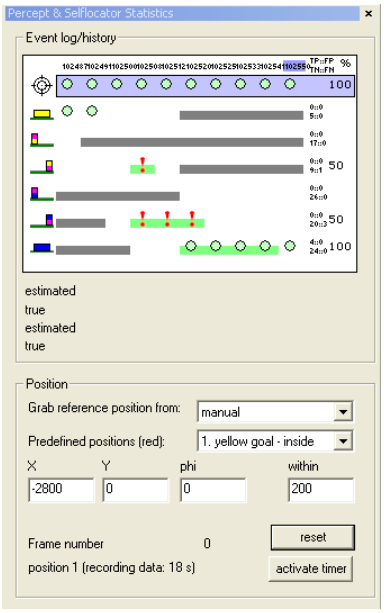

Figure 4. Screenshot of the statistics toolbar.

The self-locator module determined the robot's position within a perfect range around the true position (6.35 cm and 2.94 degree average error in localization). Tests after the championship, as shown in the appendix, could not reproduce this level of accuracy, but are acceptable considering the manual procedure.

3.2. Color table creation

The original color table creation tool required, apart from a good instinct and a calm hand, a lot of patience until the color-to-color class lookup table consisting of 64^3 elements was filled in sufficiently correctly. Even then, the produced color tables were prone to error because some regions in the color table were constantly relabeled.

To reduce the need of a calm hand an automatic flood fill algorithm was implemented.

To overcome the relabeling problem, the new toolbar keeps track of the weight of all color class assignments per entry (color cube) in the table. The weight is computed in such a way that in the end every mouse click of the user gets the same weight, no matter how big the selected (and added) region of pixels is. This has the advantage that a click on the (relatively) small orange ball counts as much as a click on the (relatively) big green carpet. This natural approach keeps the influence of a color class in the color space linearly balanced on the amount of user interaction (regarding this color class). The difference between these approaches is illustrated in figure 5.

In figure 5 first a pixel on the orange ball is selected. The flood fill algorithm searches automatically for comparable pixels in the neighborhood. A round region is found, together with some noise in the corners of the yellow goal. The user wants to correct this, and selects a yellow pixel in the goal as typical 'yellow'. The automatic flood fill finds many more comparable pixels this time. For the unweighted algorithm this means that the example of the large yellow area overrules the small orange example, which results in assigning YellowGoal to the highlights on top of the ball. With the weighted algorithm the ball keeps its color class assignment.

In the appendix a quick tutorial on how to create a color table can be found.

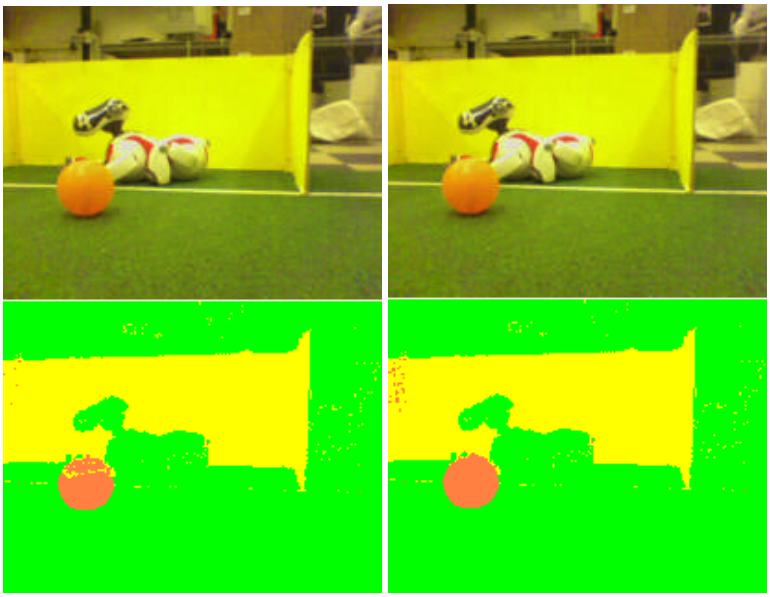

Figure 5. Resulting color segmentation using unweighted (left) and weighted (right) color class assignment, after two assignments (first the small orange region, then the big yellow region). The green region is unassigned.

3.3. Binary Image Deployment

RobotControl allows memorystick creation with the current Aibo binaries in the `/build/MS/` path of the project tree. Apart from the fact that this procedure is very time-consuming (almost two minutes per memorystick, plus the shut-down and start-up time of an Aibo) it only allows the creation of the most current binaries of the project tree.

The *Binary Image Deployment dialog* solves these problems: it smartly checks whether the files which need to be copied are different from those on the memorystick (leading to copy times below 10s) and even allows for software and configuration to be sent wirelessly (via the FTP protocol) which in turn speeds up the deployment process enormously. Moreover, it is possible to save binary images in a single `.tgz` archive file that can be stored and conserves in this way the state of development. One binary image roughly occupies 4MB and can easily be sent via e-mail or stored on project sites.

This tool greatly facilitated the co-operation between the different Aibo projects spread out over the Netherlands.

4. Technical challenges

We signed up for participation in all three technical challenges.

However, the SLAM challenge module couldn't be finished in time (and still hasn't been). For the open challenge we wanted to show a predator-prey demonstration running on the Aibos, but we failed to produce working memorysticks in Osaka (undetected installation problems with the Tekkotsu framework). The variable lighting code was prepared and tested thoroughly, however an uneven floor and extremely bad

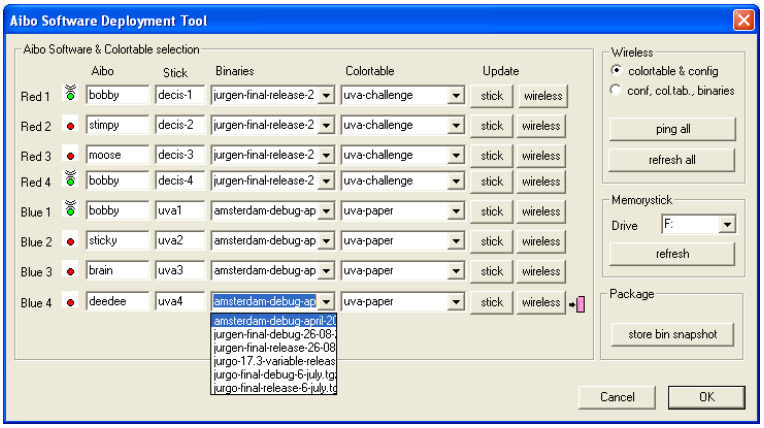

Figure 6. Screenshot of the deployment dialog.

luck made the ball roll - already after the first shot of our robot - around in a circle and finally, under the back of the (sleeping) defending robot, where our robot couldn't grab it any more.

Despite this unfortunate situation, our approach works under a huge amount of lighting conditions, is very simple, computationally cheap and even allows a whole soccer game being played under varying lighting conditions (such as necessary for an outdoor demonstration), therefore we want to explain it below in more detail.

4.1. Variable Lighting

The vision module segments incoming images using the color-to-color class lookup table. Inspired by the multicolor table approach by Mantz [6] and Slamet [9], the Dutch Aibo Team came with the idea to handle varying lighting conditions with color segmentation. The main principle is to supply the robot with multiple color tables and a module that selects the most appropriate one both in a stable way and with low latency. Along with the switching of color tables, the camera settings can subsequently also be adapted, letting the Aibo change the shutter speed, white balance and the camera gain factor.

The question remains how to detect that the current color table / camera settings combination is at a certain moment not appropriate any more. We chose to ground this decision on a fast and outlier-protected average over the Y-channel of the image (basically the brightness)[6]. Averaging is needed, because there are large natural fluctuations in the brightness for a robot playing soccer. Although in most cases the biggest part of an image consists of the green carpet and the background, it might happen that the robot is currently handling the ball or running into obstacles, of which it might (not) be aware of. We therefore decided to use a double filtering both in the spatial (per image) and temporal (over multiple images) dimension to obtain stable readings[7].

For the spatial dimension, we use a pixel grid[8] spread uniformly over the input image. From the selected pixels, we compute the median (using a histogram) and feed it into a sliding average filter over the last frames[9]. This produces fairly stable readings, as can be seen in figure 7.

[6]This does not take into account the effects of a changed white balance, but we think that for the moment it is safe to assume that the robot (and field) is not suddenly kidnapped from an outdoor to indoor environment or vice versa.

[7]As stable as possible. Of course the robot might be mistaken when the sight is obstructed for a long time. However, as long as it is stuck, there is probably no well-suited color table/camera-settings combination anyway.

[8]In Osaka: 10x10 grid points.

[9]In Osaka: 5 frames.

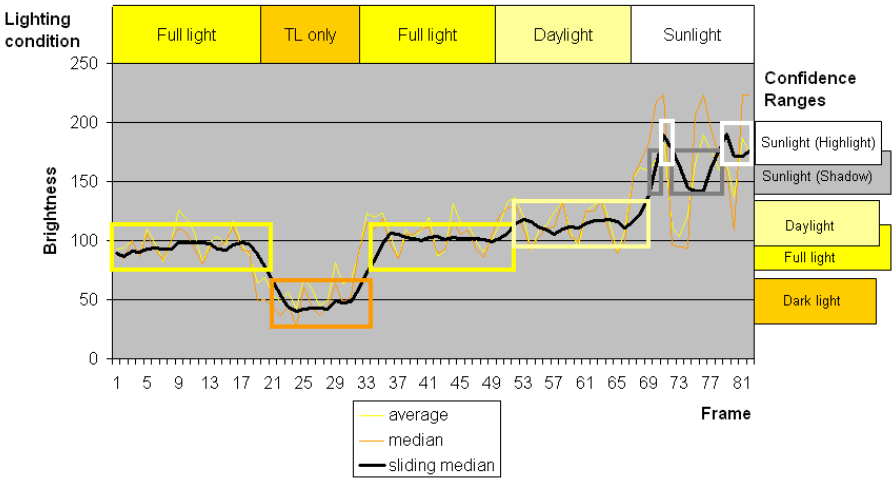

Figure 7. Brightness filtering under variable lighting conditions.

To create figure 7, we changed the lighting conditions quite drastically for five times, as indicated at the top of the figure. At the center of the image the brightness is indicated. The yellow line represents the average brightness per frame. The orange line indicates the median average brightness per frame. The jumps correspond to the head movements of the robot (head sweep, robot is searching for the ball). The sliding average filter stabilizes these readings very well. The horizontal bars on the right give the (estimated) confidence range of different color tables.

Whenever the sliding average leaves a specific threshold, the color tables are adapted. In reality also the camera settings (gain, shutterspeed) are adjusted to keep the brightness in a convertible range between 50 and 100, but in figure 7 the camera settings are kept constant to illustrate our concept.

To make this possible, the confidence values for a given color table/camera-setting combination have to be estimated. Therefore, after building a color table under a certain lighting condition, we let the robot play with the ball for a short time, while it collects the filtered values in a second histogram. We now can compute a certain quantile[10] (extending from the center of the probability mass) that we consider to be the validity range of the newly produced color table/camera-settings. This is supervised learning.

After having done this for several conditions (darker and lighter), we end up with a complete color table selection table. It turned out that the performance is somewhat better when the validity ranges are tuned manually afterwards, for example if brightness averages drop below 30 this means noisy images, while if the average rises above 100 the images might contain ugly highlights and moreover it would be recommendable anyway to switch to a higher shutter speed that would result in clearer images. Whenever the robot detects that the sliding average leaves the validity range at the bottom or at the top, it switches (if possible) one setting downwards or upwards respectively. After that, a counter is initialized to prevent switching again before the sliding average filter is completely flushed[11].

In Osaka we used 5 different color tables; 2 of them for darker lighting conditions (we produced them by dimming half/all of the field lamps) and 2 of them for brighter conditions. As we were not allowed to record logfiles with the additional lights turned on, we recorded some logfiles under normal lighting with decreased

[10] In Osaka: 90%.

[11] With our settings in Osaka, this still allowed up to 6 switches per second.

shutter speeds that produced seemingly brighter circumstances. Below you see the configuration file we used for the challenge:

```
# variable lighting config file
#
# format:
# <name> <YAvgRangeFrom> <YAvgRangeTo> ..
#        .. <WhiteBalance> <Gain> <ShutterSpeed> <ColorTable>
#
# coltables 4/5 are estimated, therefore the confusing names
# (the names stand for the camera settings while recording)
#
# darkest to brightest
1/complete/high/slow 50 85  indoor  high   slow col1.c64
2/half/high/mid      50 85  indoor  high   mid  col2.c64
3/normal/high/fast   35 70  indoor  high   fast col3.c64
4/normal/high/mid    60 90  indoor  high   fast col4.c64
5/normal/medium/fast 35 70  indoor  medium fast col5.c64
```

Apart from the unlucky demonstration itself, we believe (and will demonstrate) that with this algorithm it is possible to play outdoor with changing sunlight and weather conditions. Furthermore, it allows us to do so without expensive spotlights and eventually even remove the window shades in our robot laboratory.

Continuing on this trajectory, a master student will henceforth examine self-calibrating possibilities and implement one. The complex furnishing of a robot laboratory (and actually most places) should make special color-tagged landmarks superfluous, so we hope that some camera sweeps from given positions on the field will be sufficient for precise map building. Even if the processing power of the Aibos is limited, the whole process has only to be done once at the beginning and maybe even computations could be split up and solved by its teammates.

5. Conclusions and discussion

The main problem of the Dutch Aibo Team, which is in its second year of existence, is its insufficient manpower, unaligned research directions of its supporting institutes and a missing long-term perspective for its future. As the main overview is held and main work is accomplished by (short-term) student projects, all experience and skills get lost for the Dutch Aibo team as regularly as its members change. These issues have been recognized, and initiatives to incorporate this effort in long-term research programs on a National and European scale have been implemented.

However, when considering the small team size we are completely satisfied with the results we achieved this year at the RoboCup World Championship 2005. We left the tournament after several fair games in the intermediate preliminary rounds and took a lot of interesting inspirations back home. Although we have to acknowledge that part of our success surely was due to the excellent basis we had from the German Team code, nevertheless we think we gained a lot of experience, built up RoboCup competition skills and established in many places important links to other scientists, research groups and communities.

6. Acknowledgements

The authors wish to thank the official Dutch AIBO Team representation at the RoboCup 2005 for their enthusiasm, cooperation and progress: Jürgen Sturm (University of Amsterdam), Stefan Leijnen and Silvain van Weers (University of Utrecht), Nico Assink (Saxion Universities of Professional Education), and Niek Wijngaards (DECIS Lab, Coach). We like to thank David Knibbe (University of Amsterdam) for his comments on the final version of this article.

Our participation in the RoboCup 2005 event (as well as the German Open 2005 event) could not have been possible without the help from many groups and persons from the Dutch AIBO Team. Although the overall team consists of 11 research groups (see www.dutchaiboteam.nl/partners for an overview), we especially wish to thank the following groups and their people for their contributions and involvement in our AIBO soccer and RoboCup activities: Frank Dignum and the students from this year's Software Project (University of Utrecht), Pieter Jonker and Paul van Rossum (Delft Technical University), Arnoud Visser and his students (University of Amsterdam), Bart de Boer and Gert Kootstra (University of Groningen), Theo de Ridder and Henk van Leeuwen (Saxion Universities of Professional Education), and our team coach Niek Wijngaards (DECIS Lab / Thales Research and Technology Netherlands). Further, we would like to thank our sponsors, who made the journey to Osaka possible.

References

[1] R. Brunn and M. Kunz. Ein globales sichtsystem zur unterstützung der entwicklung autonomer fussballroboter. Master's thesis, Technische Universität Darmstadt, 2005.

[2] A. Doucet, N. de Freitas, and N. Gordon. *Sequential Monte Carlo in Practice*. Springer Verlag, New York, 2001.

[3] Uwe Düffert. Quadruped walking: Modeling and optimization of robot movements. Master's thesis, Humbolt-Universität zu Berlin, 2005.

[4] Jens-Steffen Gutmann and Dieter Fox. An experimental comparison of localization methods continued. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'02)*, Lausanne, Switzerland, October 2002.

[5] Woiyl Hammoumi, Vladimir Nedovic, Bayu Slamet, and Roberto Valenti. Improving self-localisation and behaviour for aibo's soccer-playing robots. Technical report, Universiteit van Amsterdam, February 2005.

[6] F.A. Mantz, P.P. Jonker, and W. Caarls. Thinking in behaviors, not in tasks; a behaviour-based vision system on a legged robot. In *Proceedings of the Robocup 2005 Symposium, Osaka, Japan*, July 2005.

[7] T. Röfer, H.-D. Burkhard, U. Düffert, J. Hoffmann, D. Göhring, M. Jüngel, M. Lötzsch, O. v. Stryk, R. Brunn, M. Kallnik, M. Kunz, S. Petters, M. Risler, M. Stelzer, I. Dahm, M. Wachter, K. Engel, and A. Oster. *GermanTeam RoboCup 2003*. Online, 199 pages, 2003.

[8] T. Röfer, T. Laue, H.-D. Burkhard, J. Hoffmann, M. Jüngel, D. Göhring, M. Lötzsch, U. Düffert, M. Spranger, B. Altmeyer, V. Goetzke, O. v. Stryk, R. Brunn, M. Dassler, M. Kunz, M. Risler, M. Stelzer, D. Thomas, S. Uhrig, U. Schwiegelshohn, I. Dahm, M. Hebbel, W. Nisticó, C. Schumann, and M. Wachter. *GermanTeam RoboCup 2004*. Online, 299 pages, 2004.

[9] B. Slamet and A. Visser. Purposeful perception by attention-steered robots. In *Proc. 17th Dutch-Belgian Artificial Intelligence Conference, BNAIC'05*, Brussels, October 2005.

[10] J. Sturm and A. Visser. Improvements in monte-carlo localization for a 4-legged robot. Technical report, September 2005. http://www.science.uva.nl/ arnoud/research/aibo/Sept2005.html.

Appendix: Self-Localization Results

This table describes the specific measurements [12] underlying our conclusions regarding the precision of our self-localization in section 2.3.

	activated new features:				
	all off	flashback	distance	odometry	all on
average localization error:	213 mm 8.27 deg	173 mm 5.77 deg	143 mm 5.88 deg	128 mm 4.65 deg	135 mm 5.03 deg
recovery time at position:					
inside yellow goal	1.0 s	1.8 s	2.8 s	0.6 s	0.5 s
left corner penalty area (red team)	2.1 s	3.1 s	2.7 s	2.9 s	3.1 s
right corner penalty area (red team)	3.1 s	1.3 s	0.8 s	1.0 s	1.9 s
left re-entry point	14.1 s	4.5 s	1.9 s	3.3 s	3.0 s
red striker kickoff position	2.0 s	1.1 s	0.9 s	2.0 s	2.2 s
right re-entry point	8.4 s	3.2 s	3.0 s	2.1 s	2.5 s
blue striker kickoff position	3.4 s	1.9 s	3.0 s	2.1 s	2.1 s
right corner penalty area (blue team)	3.6 s	2.0 s	1.7 s	2.5 s	2.2 s
left corner penalty area (blue team)	10.5 s	1.3 s	2.7 s	2.0 s	1.8 s
inside blue goal	1.5 s	0.7 s	2.1 s	1.3 s	2.1 s
recovery time average:	4.97 s	2.09 s	2.16 s	1.98 s	2.14 s
recovery time std deviation:	4.45 s	1.17 s	0.82 s	0.83 s	0.72 s

[12] This result is the predecessors of the measurements presented in [10]. These measurements were recorded by direct communication with the Aibo on the field, which means slightly different circumstances for each run. A more fair comparison between the algorithms can be made with the playback of a recorded logfile, as shown in [10].

Appendix: Manuals

HOWTO use the Software Deployment Tool

In this part of the appendix you will learn how to set up RobotControl when you use it for the first time, write your first memory stick, get your Aibo up and running and finally connect to it wireless.

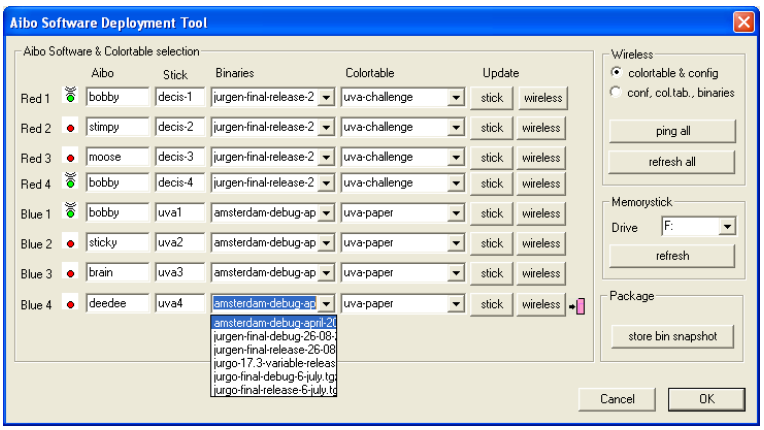

Figure 8. Screenshot of the deployment dialog.

Firstly, you have to create a new WLAN configuration (click on the "add connection" symbol in the WLAN toolbar. Give it a name, and enter the IPs of the robot(s), as well as the ESSID, netmask, optionally the WEP key, WLAN channel and set the AP mode to 1 (infrastructure only mode). Then close the dialog with the OK button.

Then click on "Deploy Software". Think about which robot you'd like to have (red or blue, player 1 goalkeeper, player 2 defensive supporter, player 3 and 4 offensive supporter). Aiboname and Stickname need not to be correct, but be sure to select an existing binary image and a colortable for your stick. Also keep in mind that the memory stick will be set up with the IP address corresponding to the player number) you entered in the WLAN dialog.

Now click on the corresponding "stick" button - et voilà! Plug the memorystick in the Aibo and turn it on. After a few seconds, the Aibo should get up. Now check in the WLAN dialog the "connect" box, then click on "Connect" in the WLAN toolbar. The Aibo should reply with its name, battery state etc. Now you can issue further commands by the various toolbars, for example, bring the robot in the "ready" state by clicking on the "ready" symbol in the Game toolbar.

HOWTO Record a logfile and make a colortable

Set up the wireless settings (the "Edit and Copy" button) in RobotControl [13], check the *connect box*, then click on ok. Open the "Deployment Dialog", select the correct memorystick drive letter, select a debug binary image for the corresponding robot and write it to the stick. Close the dialog. Start the Aibo, then try to connect.

Select in the debug toolbar the SendJPEGImage key, select "send every n msec", enter something such as 1000 in the text field, and press send. Now you should receive images from the robot. Enable recording in

[13] Especially IP, wireless SSID and WEP key.

the logfile toolbar. Move the robot around the field, show it every flag from different directions and distances. Show it the goals, the lines and the ball, also in combination with red and blue robots. Try to record around 100-200 images. Disconnect from the robot, save the logfile and shut-down the robot.

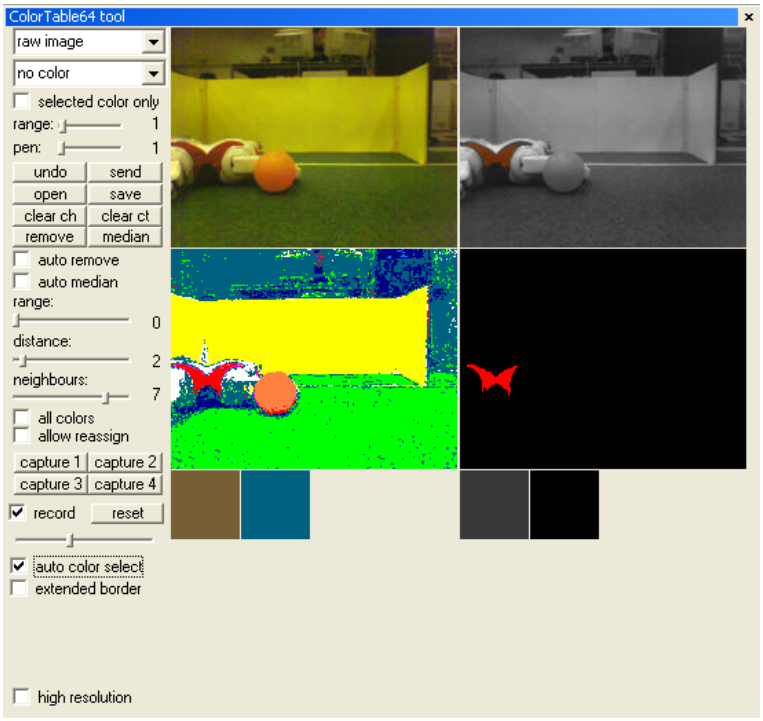

Figure 9. Screenshot of the colortable64 dialog.

Now open the colortable toolbox and load an existing colortable (if you have one), that provides you with a head start. Now click in the middle of color planes such as the yellow region of the flags and goals, or the white stripes etc. If the automatic floodfill doesn't select the object well, reload the image in the logfile toolbar and try again with a changed color distance threshold. You can either select the assigned color class manually or let the floodfill select the class for you, then the colorclass which occurs the most in the object gets the polygon. With the right mouse button you commit the selected pixels to the color matrix and color table. After a few images, the colortable should be good enough to classify most pixels correctly. After that, walk through the rest of the logfile and check whether everything is classified correctly, otherwise add pixels again. When a simulated robot is enabled, you can see how well it already detects landmarks, lines and the ball in the image viewer.

Finally, don't forget to save the colortable in the `/Deployment/Colortables/` folder; then they become available in the deployment dialog. You can also save the underlying complete matrix for later extensions (but keep in mind that they weigh around 15MB on your harddisk).

HOWTO use the Statistics Toolbar

In this section we give a brief introduction to the Statistics Toolbar. You will learn how to measure the performance of the image processors as well as of the selflocator modules.

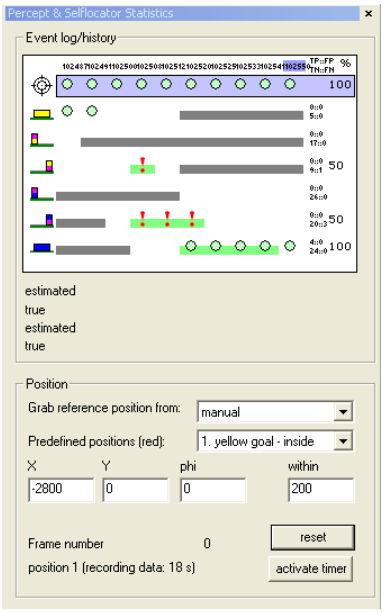

Figure 10. Screenshot of the statistics toolbar.

As the statistics toolbar knows the robot position and orientation, it calculates which landmarks should be fully visible from the current pose, and which are definitely not. A visible (invisible) landmark is indicated by a green (grey) background in the corresponding frame column. When the robot reports a seen landmark, a circle is painted at the corresponding frame column/landmark row. A red exclamation mark indicates a vision mismatch (either because the robot sees an actually invisible landmark or doesn't see a visible landmark). This serves also as the basis for the calculation of the true positives (correctly seen landmarks), true negatives (correctly not seen landmarks because they are invisible) as well as the error counts (the false positives/negatives). The selflocator analyzer of the Statistics Toolbar computes the distance of the true position to the robot's estimate and aggregates this in an average error estimate. Whenever the error is smaller than the threshold (default 20 cm), this is indicated by a green (otherwise red) circle in the corresponding frame column. At the rightmost column you find summarized counters for the true/false positives/negatives as well as percentages of the overall performance per landmark and localization.

In the simulator, using the oracle as position source

The easiest way to get a feeling for the use of the statistics toolbar is by running the simulator. Activate single robot (for example red 2) in the simulator toolbar, check "send Oracle", reset and then start the simulator. You should now see the messages being processed by the Statistics toolbar. The columns represent a single frame (on the top you can see the frame number); the first row contains the information about the selflocator, and the next ones about the various landmarks. Click on a single cell to get some additional information about the event, like the estimated and true position (of the robot/landmark), the (average) error, the state of visibility.

In the real world, using the test parcours

If you want to test your modules in the real world, you have three possibilities. The first posibility is to directly enter the robot's position manually in absolute field coordinates (in mm and degrees). Connect to your robot, place it on the selected spot, make it send landmark percepts (sendPercepts debug key) and its robot pose (sendWorldState) every 300ms. Now you should see how the robot performs at this location. Of course, you want to test the overall performance, so one spot is not enough. This brings us to possibility number two. We defined ten points on the field, with high relevance for the image processor and selflocator. The toolbar can guide you and the robot automatically

around the field. Set up everything as before, and additionally select in the "Xabsl2 monitor and tester" the "walk-slam-route" option (activate the "to physical robot" checkbox). The idea is that the robot is placed on the ten different positions for one minute each. To help you get the timing right, both the toolbar as well as the robot can give you exact indications what to do when. To get started, you have to synchronize the robot with RobotControl. Therefore, click simultaneously on the "activate timer" button and the button on the Aibo's back. In the toolbar, a message appears (with a big font, good to read): "Position 1: yellow goal, inside". In the mean time the robot will have lowered its head and is looking to the ground. Now place the robot at the indicated position. In the Statistics Toolbar, you should see a timer counting down 20 seconds before the true recording for this position starts. After this period, the robot raises its head and starts localizing for 40s. As soon as the selflocalization error drops below the threshold for the first time, the kidnap recovery time is recorded and written to the console. When the 40 seconds are over, the robot lowers its head again (and RobotControl produces a beep sound) meaning that it should be transported to the next position. After the whole sequence of 10 positions is completed, a brief textual summary on the image processor and selflocator performance is written to the console of RobotControl.

Last and a bit more tricky is the third possibility, namely to completely trust on the robot's own localization capabilities. This allows you test the performance during a real game, and to estimate the influence of the image processor module on the selflocator. The third possibility basically means that now it assumed that the robot is always right about his own position estimate (which is definitely not always the case), and generate with this assumption the counters of true/false positives/negatives per landmark for the image processor.