1 Introduction

This document reports a snapshot of the status of the ISocRob team of soccer robots at the end of RoboCup'98 competition. The nature of this report is rather technical, assuming knowledge of the more general context of the SocRob project [4] and the ISocRob Team [2] in particular.

The outline of this report comprises a technical description of each robot, beginning with the perception of the environment (vision) all the way to the wheel motion, crossing the high-level behavior issues. But prior to this information flow based description, some generalities about the robot are described.

2 Generalities

The physical dimensions of the robot are $410mm \times 410mm$ when projected onto the ground, and height $360mm$. The computational power of the robot is centered on a AMD-K6 based conventional PC motherboard. The expansion boards attached to the motherboard are a VGA S3 graphics card, a Realtek Ethernet 10Mb/s card, a VideoLogic Captivator PCI frame grabber board, and a custom-made motor control board (based on a `LM629` motor controller, connected with a `LM18200` power amplifier).

The operating system used is Linux [1] (RedHat 5.2 distribution). The software was developed in C. There is a custom-made driver (kernel modules, named `robot.o`) which interface the motor controller board. This driver creates a *char device* at `/dev/srm` (major 60, minor 0). A function library (`liblm.a`) was developed to allow a higher level of abstraction over the motor control. The main code is divided into two files: `hal.c`[1], which contains a simple high-level API (Application Interface) that abstracts the interface with all sensors and actuators of the robot, and `kontrol.c`[2], which contains the main deliberative code, including image processing routines. There is also a parameter file (`param.dat`) which contains numeric parameters for the software. These parameters avoid re-building the executable each time a parameter is changed (useful for parameter adjustment).

[1] `hal` is the acronym of *Hardware Abstraction Layer,* which have nothing to do with the name of the computer appearing in "2001: A Space Odyssey" computer.

[2] The letter K in its name is not a typo, but rather a style of writing that appeared in late '80 and early '90 in the world of games, demos, and BBS', usually called "elite". As an historical side-note, this style is connoted with a "cool mood", replacing "E" by "3," "S" by "Z," "C" by "K," and so on...

3 Perception

The only transducer used by the robot is a vision camera. It is a Philips XC731/340 color CCD camera designed for video-conferencing, sold bundled with a VideoLogic Captivator PCI frame grabber board based on the Brooktree Bt848 chip [3]. Both devices are used as the hardware supporting vision perception.

The camera connects to the frame grabber through a SVHS[3] cable which transmits a PAL video signal. The camera viewing angle is 51×39^o featuring a CCD with a resolution of 512×582 pixels. The illumination sensitivity is rated under 10 lux. This camera has a manual focus knob and an automatic iris and white balance self-adjusting. The automatic white balance can be turned off by a switch. Since it deforms the RGB space, depending on the light conditions, sometimes it is useful to switch it off. Another annoying feature of the camera/frame grabber set was the mangled video output signal after turning the robot power on. It is usually necessary to switch the camera power off and on again.

The CCD chip position relative to the robot is 285mm above the floor level, facing front, 100mm ahead the wheel axis, tilted 14^o (approx.) down with respect to a horizontal plane. The orientation of the camera can be manually adjusted, and was usually calibrated *ad-hoc* in the following way: the robot is positioned in the goal position of one side, and the camera tilt is adjusted in order to make about 30–50% of the other side goal visible.

The frame grabber is directly connected to the motherboard PCI bus, and since it has no local RAM, the grabbing is done directly into the motherboard RAM (or to the graphics card internal frame buffer). In order to to so, the Bt848 chip is equipped with a programmable DMA controlled by a small RISC processor. This combination allows some rather sophisticated modes of grabbing, including various color formats, on-the-fly image clipping and scaling, and so on. However the developed software only made minimal use of these features. Note that exhaustive use of this features can easily lead to an extremely unportable code and hardware dependant.

4 Image Acquisition and Feature Extraction

The hardware-level frame grabber driver is the bttv package (for Linux). It compiles to a kernel providing an API accessible through a char device (`/dev/bttv`) mostly using the `ioctl()` system call. This device is opened

[3]This means that the video signal is separated in its luminance and chrominance components.

with an usual `open()` system call. To acquire a frame, a specific `ioctl()` command has to be issued. After this function execution, a single frame (containing both even and odd fields) is grabbed to a previously allocated space of memory (which has to be locked into the physical memory, *i.e.,* non-swappable memory). The overhead for this capture is considerable: the driver has to turn on the frame grabber DMA, wait for a frame start[4], grab it, wait for the end, and turn the DMA off. Since the vision loop runs in a single executing thread, the time elapsed in this process is added to the time spent in image processing and next action decision, resulting in a frame rate in the range of 10–15 frames *per* second.

The format of the acquired image is 200×150 pixels coded in `RGB555`, *i.e.,* five bits for each R, G, or B color component (each pixel occupies two bytes). Note that this resolution is independent from the CCD matrix: this matrix is scanned and converted into a analog (PAL) video signal, where one pixel row in the matrix corresponds to a video scan line, and the horizontal discretization is converted into an analog signal; moreover, the frame grabber is able to sample this analog video signal into arbitrary resolutions, where the chosen resolution is 200×150; therefore this whole process can be seen as arbitrarly sampling an analog image, retaining however some contraints imposed by the PAL video format.

The feature extraction is mostly based on color discrimination and center-of-mass determination. Figure 1 shows the representation of the RGB space used to derive to expressions for the color discrimination algorithm.

Let the vector $(r\ g\ b)^T$ denote a particular color in the RGB color space, assuming that each component is already normalized to the interval $[0, 1]$. The Y axis is generated by the unit vector $\hat{y} = (1\ 1\ 1)^T/\sqrt{3}$. spawning a 1-dimensional linear sub-space. The orthogonal complement[5] of this sub-space is a plane orthogonal to the Y axis. To define such a plane we need two arbitrary orthogonal vectors: making the $r = 0$ we get[6] $\hat{u} = (0\ 1\ \ -1)/\sqrt{2}$. And to complete an orthonormal basis, we can find the third unit vector from the external product of the other two: $\hat{v} = \hat{y} \times \hat{u} = (-2\ 1\ 1)^T/\sqrt{6}$.

The hue of a color can be found from the angle it makes with respect to this basis along the $\hat{y}$ axis. Therefore, in order to define the region in RGB space corresponding to an interval of hue values, we need two delimiting

[4]Note that the camera video scan runs asynchronous to the image processing software loop.

[5]Given a linear space $\mathcal{L}$, the orthogonal complement of a linear sub-space $\mathcal{S}$ with respect to $\mathcal{L}$ is another linear sub-space $\mathcal{C}$ such that $\forall_{x \in \mathcal{S}}\ \forall_{y \in \mathcal{C}}\ x \cdot y = 0$. In other words, $\mathcal{C}$ is the sub-space of all vectors orthogonal to all vectors of $\mathcal{S}$.

[6]We have to satisfy $\hat{u} \cdot \hat{y} = 0$ and $\|\hat{u}\| = 1$, leaving us with a degree of freedom, which can be, for instance, zeroing the first component.

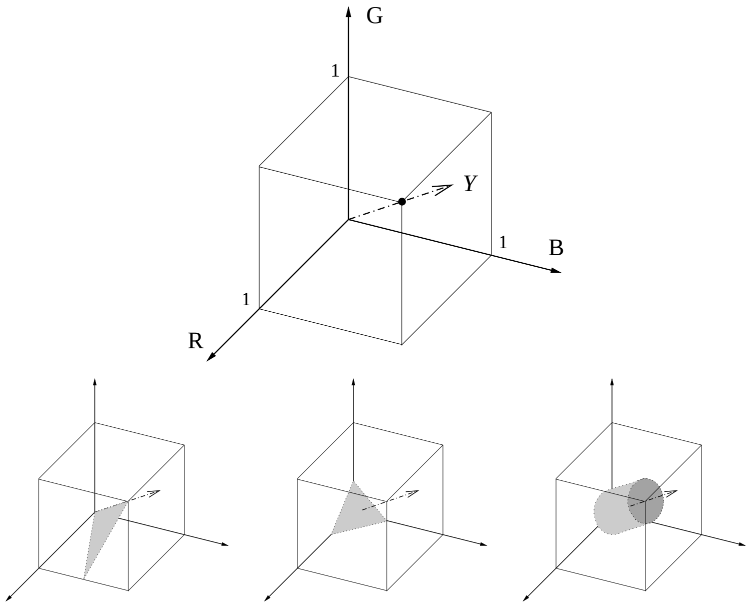

Figure 1: An RGB space representation; each component is normalized to the range 0–1. In the upper half there is a general view of the space along with luminance axis. In the lower half the sub-sets of constant hue, value, and saturation are shown.

planes, which intersect along the Y axis (see figure 2). These two planes delimit two symmetric "cake slices". Let us define each one of these two planes by its orthogonal unit vector[7]. This unit vector not only defines a plane (dividing the whole RGB space into two regions), but also a direction[8]. From the internal product of a vector by this defining unit vector, one can determine at once in what region the point lies. To find out whether a given color, say x, is inside a slice defined by $\hat{p}_1$ and $\hat{p}_2$, two inequalities have to be met: $x \cdot \hat{p}_1$ greater or lesser than zero, and $x \cdot \hat{p}_1$ less or greater than zero, respectively. Figure 3 illustrates this idea. Taking slice A, it is defined by the intersection of the area "ahead" $\hat{p}_1$ with the area "behind" $\hat{p}_2$. This means that the following inequalities are simultaneously satisfied, when the point x belongs to the area A:

$$A : \begin{cases} x \cdot \hat{p}_1 > 0 \\ x \cdot \hat{p}_2 < 0 \end{cases} \tag{1}$$

[7]Note that these planes are linear subspaces, since they all pass through the origin, and thus they can be defined only by a orthogonal unit vector.

[8]Not all surfaces can be directed, e.g. Möbius srip.

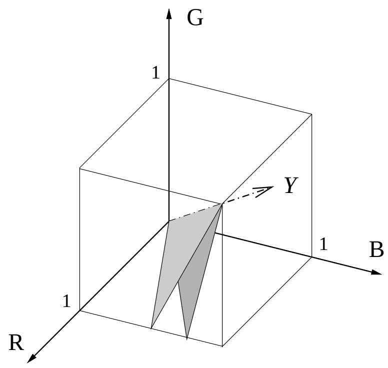

Figure 2: A range of hue angles corresponds to a "slice" in the RGB space, with respect to the Y axis. This slice is delimited by two planes.

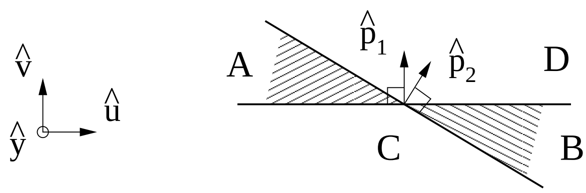

Figure 3: The geometry of the problem looking along the Y axis. Two planes, defined by the orthogonal unit vectors $\hat{p}_1$ and $\hat{p}_2$, split the RGB space into four "slices." Each one of these slices is defined by two inequalities (see text for full details).

Similarly, areas B, C, and D, are defined by

$$B : \begin{cases} x \cdot \hat{p}_1 < 0 \\ x \cdot \hat{p}_2 > 0 \end{cases} \quad C : \begin{cases} x \cdot \hat{p}_1 < 0 \\ x \cdot \hat{p}_2 < 0 \end{cases} \quad D : \begin{cases} x \cdot \hat{p}_1 > 0 \\ x \cdot \hat{p}_2 > 0 \end{cases}$$

It is important to stress that the computational load to detect a color is relatively low. Since the vectors $\hat{p}_1$ and $\hat{p}_2$ only depend on the θ_{min} and θ_{max}, which delimit the range of hue angles, they can be pre-computed, and for each pixel and each area, only two internal products and a sign check have to be computed[9], in the vision processing loop.

Besides detecting the hue of pixels using this method, a second constraint has to be introduced. The closer a given point in the RGB space is to the Y axis, the more its hue is sensitive to noise. To quantify this effect, take the Cartesian representation (x, y) of an arbitrary polar vector (ρ, θ) (in 2

[9] This results in six multiplications and four additions.

dimensions[10]). Think about θ as being the hue, and ρ the distance of the considered vector to the $\hat{y}$ axis.

$$\begin{cases} x = \rho \cos\theta \\ y = \rho \sin\theta \end{cases} \tag{2}$$

Differentiating and using (2) again, we get

$$\begin{cases} dx = x \ d\rho/\rho - y \ d\theta \\ dy = y \ d\rho/\rho + x \ d\theta \end{cases} \tag{3}$$

from which we can remove $d\rho$ by subtracting one equation from the other, after multiplying each of them by an appropriate factor:

$$d\theta = \frac{x \ dy + y \ dx}{x^2 + y^2} = \frac{x \ dy + y \ dx}{\rho^2} \tag{4}$$

Using 2 and representing the pertubation (dx, dy) in polar coordinates $(dz \ \cos\alpha, dz \ \sin\alpha)$, we get

$$d\theta = \frac{dz}{\rho} \sin(\theta + \alpha) \tag{5}$$

Unless $\sin(\theta+\alpha) = 0$, the closer to zero ρ is, the more sensitive (with respect to dz) θ is. This illustrates the increasingly noise sensitivity of the hue, whenever it approaches the $\hat{y}$ axis.

For this reason, a second constraint imposes a minimum distance from the point to the Y axis. In other words, a color must have a minimum chrominance in order to be accepted inside a color slice. Recalling figure 1, this means it has to be outside a cylinder of an appropriate radius, with symmetry axis over the Y axis.

A way to do this check is to impose a minimum norm to the chrominance vector. The chrominance vector is the projection of the color in RGB space, onto the $(\hat{u}, \hat{v})$ space, *i.e.*, the orthogonal complement of the Y axis. To formalize this idea, the luminance (x_y) and the chrominance $(x_c)$are computed using the expressions

$$\begin{cases} x_y = (x \cdot \hat{y}) \, \hat{y} \\ x_c = x - x_y \end{cases} \tag{6}$$

(note that $x = x_y + x_c$), and then $\|x_c\|$ has to be greater than a specified minimum chrominance value.

[10] It corresponds to the projection onto the orthogonal complement of the Y axis, which forms a plane orthogonal to $\hat{y}$.

To summarize the color detection algorithm, the specification of a color to be detected is defined by the tuple

$$\langle \theta_0, \delta\theta, c_{min} \rangle \tag{7}$$

from which the two unit vectors that define the delimiting planes are extracted:

$$\begin{cases} \hat{p}_{min} = \hat{u}\cos(\theta_0 - \delta\theta) + \hat{v}\sin(\theta_0 - \delta\theta) \\ \hat{p}_{max} = \hat{u}\cos(\theta_0 + \delta\theta) + \hat{v}\sin(\theta_0 + \delta\theta) \end{cases} \tag{8}$$

These expressions account for the pre-computations that can be made prior to scanning the image for color detection. For each pixel, when the following equations are simultaneously satisfied (for any color), the pixel is positively detected:

$$\begin{cases} \|x_c\| > c_{min} \\ x \cdot \hat{p}_{min} < 0 \\ x \cdot \hat{p}_{max} > 0 \end{cases} \tag{9}$$

The parameters in (7) are configurable through the parameter file. There are four color specification slots — orange (ball), yellow goal, blue goal, and green (floor) — indexed from 0 to 3. The parameter symbolic names are:

Variable	θ_0	$\delta\theta$	c_{min}
Symbol	`th0.`n	`dth.`n	`cmin.`n

where n stands for the color index.

The center of mass is used to estimate the position of several objects, namely the ball (orange) and each one of the goals (yellow and blue). The center of mass is computed over the whole image, using the usual expression (note that from now on, (x, y) denotes coordinates in the image):

$$(x_{CM}, y_{CM}) = \frac{\sum_{x,y}(x,y)p(x,y)}{\sum_{x,y} p(x,y)} \tag{10}$$

where $p(x, y)$ denotes the predicate (1 if true, 0 if false) of whether the pixel with coordinates (x, y) is positively detected as being of a specified color.

Because of the importance of an efficient ball tracking, an additional feature is also taken into account, besides the ball center of mass. Since the camera geometry is fixed, there is a direct (non-linear) mapping between the ball distance (with respect to the robot) and the y coordinate in the image. Moreover, there is another non-linear mapping between the ball distance and its apparent radius. For simplicity, the mapping between y_{CM} and estimated ball radius r_b is approximated by an affine function:

$$r_b \approx r_{max} \frac{y_{max} - y_{CM}}{y_{max}} + r_{min} \frac{y_{CM}}{y_{max}} \tag{11}$$

where r_{min} and r_{max} are the ball radius (in pixels) in the image at the top of the image and at the bottom, and y_{max} is the number of pixel rows. It is assumed that the y coordinate in the image starts from 0 at the top row and grows downwards until y_{max}. The r_{min} and r_{max} parameters are calibrated by hand, with the ball positioned at the top and at the bottom of the image[11]. The parameter code names for r_{min} and r_{max} are `yu` and `yd`, respectively.

This ball radius estimation is used to add a restriction to the ball detection criteria. In addition, some care is taken when the ball falls outside the camera field of view. The expected ball area is reduced when this happens. Finally, it is given a safe margin, resulting in a threshold which depends on the ball position in the image. When the number of pixels detected as belonging to the ball exceeds this threshold, the ball is considered positively detected.

The ball position estimation is additionally subjected to a low-pass filter to reduce noise. This filter is a simple first order IIR filter:

$$(x_b^t, y_b^t) = \alpha(x_b^{t-1}, y_b^{t-1}) + (1 - \alpha)\,(x_{CM}^t, y_{CM}^t) \tag{12}$$

where the $\alpha \in [0, 1]$ parameter specifies how much the ball trajectory is smoothed.

Finally, in order to enhance the reaction time of the robot, a simple prediction of the ball future trajectory is computed. This prediction is used in some points of the code instead of the filtered estimate. The prediction is first-order linear:

$$(x_p^t, y_p^t) = \beta y_b^t \left[(x_b^t, y_b^t) - (x_b^{t-1}, y_b^{t-1})\right] + (x_b^t, y_b^t) \tag{13}$$

where the β parameter specifies how sensitive is the prediction (parameter name: `pred`). The protuct by y_b^t is supposed to vary the prediction sensitivity along the y axis, *i.e.,* the closer the ball is to the robot (higher y_b), the more sensitive the prediction is.

With respect to the goal detection, only the center of mass coordinates of their position are taken into account.

Two more features are extracted from the image: the quantity of white and the quantity of black. For the detection of black, the luminance and the chrominance norms have to satisfy

[11] Note that all these considerations assume that the ball is at the ground level. To put the ball "at the top of the image" means to move it farther away from the robot, until its center of mass reaches the top of the camera image, and similarly for the ball "at the bottom of the image."

$$
\begin{cases} \|x_y\| < y_{thresh}^{black} \\ \|x_c\| < c_{max} \end{cases} \tag{14}
$$

(see (6) for the definitions of x_y and x_c), where y_{thresh}^{black} is a threshold. Similarly, for the white detection, the expression used is:

$$
\begin{cases} \|x_y\| > y_{thresh}^{white} \\ \|x_c\| < c_{max} \end{cases} \tag{15}
$$

The parameter names for these thresholds are:

Variable	c_{max}	y_{thresh}^{black}	y_{thresh}^{white}
Symbol	`bt-cmax`	`b-ymax`	`w-ymin`

5 Behavior State Machine

The behavior of each robot can be either *goal-keeper* or *forward.* The choice of the behavior is hard-wired in the parameter file. Each behavior is implemented by a state machine. The state machine diagram for these two behaviors can be found in figures 4 and 5.

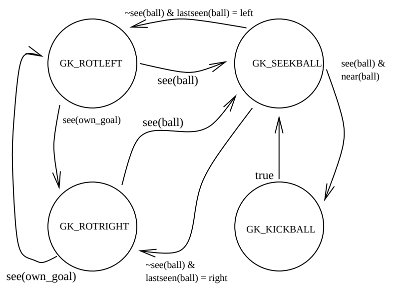

Figure 4: *Goal-Keeper* behavior state machine diagram.

The predicate `see()` corresponds to a positive identification of a color object. As stated above, the supported color objects are `ball`, `own_goal`, and `other_goal`. The colors of these two last items are exchangeable, depending on the side of the robot. The function `lastseen(ball)` can be either `left` or `right` according to the side, relative to the robot, where the ball was last seen. The predicate `near(ball)` corresponds to a threshold over the estimated ball distance (parameter name `kick-dist`, compared with y_c), and `timeout` corresponds to a pre-defined timeout period.

In a given time, a robot is in a specific state. This state specifies what it shall do in the *sense-think-act* loop. In other words, for each image frame, the actuation (motor control) is computed depending on both the present state and the sensor readings. When a transition condition occurs, the present state changes, affecting the next acquired frame.

The rationale behind the *Goal-Keeper* is to let it swing left to right periodically in a "ping-pong" fashion until it finds the ball. Once the ball is "locked" in its field of view, the robot tracks it, and if the ball approaches a certain distance, it kicks, moving forward and back quickly. The states for the *Goal-Keeper* are:

`GK_ROTLEFT` **and** `GK_ROTRIGHT` — The robot rotates left or right around its geometric center, using the parameter `vrot` as differential speed (v_d, see section 6 for further details);

`GK_SEEKBALL` — The robots follows the ball, using a simple proportional controller over the ball center of mass horizontal displacement x_b, relative to the middle line of the image $x_{ref} = x_{max}/2$:

$$\begin{cases} v_c = 0 \\ v_d = \frac{x_b - x_{ref}}{k_{dscale}} \end{cases} \tag{16}$$

where k_{dscale} is a scaling parameter (named `dscale` in the parameter file);

`GK_KICKBALL` — In this state the robot performs a pre-programmed trajectory that consists of advancing forward at speed `kick-speed` during `kick-time` (parameter names), and going backwards with the same speed and time. Then, the state machine jumps into the `GK_SEEKBALL` state. Note that the robot only jumps into this state when the parameter `kick?` is non-zero (this parameter is used to prevent the robot to "kick," mostly for testing purposes).

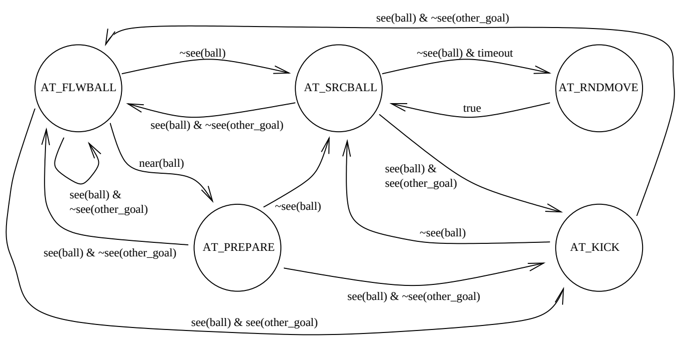

Figure 5: *Forward* behavior state machine diagram.

Concerning the *Forward* behavior, the idea is to search the ball, move towards it, position itself in such a way it sees the ball and the goal of the other team, both in the field of view, and carry the ball towards that goal. Here follows a description of the *Forward* states:

`AT_FLWBALL` — The ball is visible, and advances towards it, using a simple proportional controller over the ball center of mass. The controller policy is:

$$\begin{cases} v_c = 100 \\ v_d = \frac{x_p - x_{max}/2 \pm x_{max}/4}{k_{dscale}} \end{cases} \tag{17}$$

where the the $\pm$ sign depends on which side the goal of the other team is. This implies a small offset relative to the ball direction in order to allow the robot to turn around the ball. Note that this state is used when the other goal is not visible. Otherwise, the state `AT_SHOOT` is used;

`AT_SRCBALL` — Search the ball rotating around the geometric center of the robot, to the side where the ball was last seen. Its differential velocity (v_d) is specified by the parameter `vrot`;

`AT_RND` — This state intends to make the robot perform a random move, in order to avoid searching indefinitely for an hidden ball. However, this movement was not implemented yet;

`AT_PREPARE` — This state performs a pre-programmed robot trajectory, trying to get around the ball. During this movement, no image processing is done (*i.e.,* the robot is "blind"). This trajectory uses the parameters `prep-vel`, and `prep-t` as reference values for speed and time, and `prep-dist` for the distance to the ball from which the robot should start the trajectory;

`AT_SHOOT` — This state transports the ball towards the other side goal. To do so, the following control policy is used:

$$\begin{cases} v_c = v_{kick} \\ v_d = \frac{1.5 x_b + 0.5 x_c^{other}}{2(2 - x_{ref}/2) k_{dscale}} \end{cases} \tag{18}$$

where v_{kick} corresponds to the shoot speed specified by the parameter `kick-speed`, and x_c^{other} is the center of mass of the color of the other team goal. The differential velocity v_d is weighted between x_b and x_c^{other} (by 75% and 25%) in order to bias the robot approach direction somewhere between the center of the ball and the center of the opposite goal[12] and

The weights 1.5 and 0.5

6 Motor Control

The robot uses a differential drive topology, based on two independent DC motors. Each motor is coupled to a 132 mm diameter, 12 mm wide wheel. Each motor+wheel system is independently controlled in a velocity feedback loop, using an encoder and a dedicated IC (`LM629`). This IC implements a PID controller parameterized not only by the PID coefficients (k_p, k_i, and k_d) but also by a ramp acceleration parameter (`accel` in the parameter file).

Given the common and differential mode velocities, the left and right wheel values are derived as usual:

$$\begin{cases} v_l = v_c + v_d \\ v_r = v_c - v_d \end{cases} \tag{19}$$

The reference values sent to the wheel controllers are further multiplied by the scale factor `vel` (in the parameter file).

References

[1] Linux online. URL: `http://www.linux.org`, 1998.

[2] P. Aparicio, R. Ventura, P. Lima, and C. Pinto-Ferreira. *RoboCup-98: Robot Soccer World Cup II*, chapter ISocRob - Team Description. Springer-Verlag, Berlin, 1999.

[3] Ralph Metzler. Bttv — a linux driver for bt848 based frame grabbers. URL: `http://www.thp.uni-koeln.de/~rjkm/linux/bttv.html`, 1998.

[4] R. Ventura, P. Aparicio, P. Lima, and L. Custódio. IsocRob — Intelligent Society of Robots. In *Collection of RoboCup99 Team Description Papers*, August 1999.

[12] The center of mass of the visible portion of the ball, to be exact, because of a possible occlusion of an opponent goal-keeper.