DETAILED VISION DOCUMENTATION

Real-time Visual Perception for Autonomous Robotic Soccer Agents

RoboCup '99
Thomas Charles Karpati
Prof. Rafaello D'Andrea
Dr. Jin-Woo Lee

Revision 2.
Revision Date: 5/26/99

RoboCup (The World Cup Robot Soccer) is an attempt to promote AI and robotics research by providing a common task for evaluation of various theories, algorithms, and agent architectures. In order for the robot (physical robot and software agent) to play a soccer game reasonably well, wide range of technologies need to be integrated and numbers of technical breakthroughs must be accomplished. The range of technologies spans both AI and robotics research, such as design principles of autonomous agents, multi-agent collaboration, strategy acquisition, real-time processing and planning, intelligent robotics, sensor fusion, and so forth. [6]

Note on Terms

The following terms are used extensively throughout the document.

Blobs: The term blob refers to a region of the image where all of the pixels in that region are of a homogeneous value. The surrounding area of the image is of a different value.

Blob Identifier Image: The blob identifier image is a binary image that contains blobs. The blob analysis module can compute blob features on grayscale images but needs the binary image to locate the regions on which to compute features.

Connected Components: Pixels in an image that are neighboring and all contain the same pixel value. The neighbors can be defined by using a 4 connected lattice or an 8 connected lattice. Two regions of equal pixel values but separated by a different value are not considered to be of the same connected components.

Interesting Color: Colors that are being actively sought during the vision processing.

Oriented Team Marker: A team marker that has had an orientation marker registered to it.

Primary Color: One of the three colors that are used to locate everything on the playing field. The colors are orange, blue, and yellow. Orange identifies the ball, and blue and yellow identify the robots and the team that they belong to.

Robot Profile: The current position, orientation, linear velocity, and rotational velocity for the robot in question. These variables are the current values of the linear tracking filter and the information that is sent to the artificial intelligence computers.

Robot State: The position and orientation of the robot in equation.

Secondary Color: Any other colors that are used for gathering information about the current state of the robots. This includes information such as orientation. The secondary colors can be any color other than the primary colors.

Team Marker: A marker that is placed on top of the robot to identify the team that the robot belongs to.

Section 1 Table of Contents

Section 2 RoboCup: Robot World Cup Initiative

The RoboCup competition is a program designed to promote robotics research through out the world by defining a standard problem from which several issues can be addressed. The competition requires a cross-discipline effort including aspects of Electrical Engineering and Mechanical Engineering as well as Computer Science. To produce a competent team; the system requires mechanics, digital and analog electronics, control theory, system theory, algorithms, artificial intelligence, and real-time sensory processing. The goal of the RoboCup Initiative is to build a system that can play a game of soccer using robots for the players based upon the rules stated in Appendix A.

Section 3 System Overview

The RoboCup system is designed to play a game of soccer in a way that resembles an actual soccer game played by human players. The system is completely autonomous and free of human intervention during game play. The system contains four main components:

- The vision system
- The artificial intelligence system
- The wireless communication to the robots
- The robots

The artificial intelligence broadcasts commands over a wireless network to the robots, which in turn carry out the commands. The vision system determines the current game state and sends the state to the AI computers for further processing and determination of a strategy to be executed.

The system is designed to run at a rate of 60 Hz, and the robots are designed to achieve a maximum velocity of 2 m/s and a maximum acceleration 2.94 m/s^2. The system has been implemented and is functional, yet runs at a frame rate of 40-45Hz.

Section 4 Vision System Overview

The vision system consists of everything from the lighting and the markers on top of the robots, through the acquisition and processing of the visual data, to the transmission of the processed data to the artificial intelligence computers. The vision system processes color images from a CCD camera to locate and track the robots and the ball. It consists of segmenting the image, locating the objects that are considered to be interesting, identifying, and tracking these objects. This information is then sent over the network connection to the artificial intelligence system.

The field conforms to fairly stringent constraints to aid in the visual processing of data. It is uniformly illuminated at levels between 700 – 1,000 lux, and the number of colors on the field is limited to a few.

The block diagram given in figure 1 describes the system. A camera is mounted above the field to perceive the global state of the system. Each robot is marked with a team marker that is used to identify the team that the robot is a member of. These colors are set forth in the RoboCup rules and are either yellow or blue. The ball is an orange colored golf ball. Aside from the team markers other markers may be placed on top of the robot, such as an orientation marker, or identification marker. Each frame is grabbed from the camera and processed using a color segmentation algorithm to separate the colors that are deemed to be interesting. The interesting colors are then processed according to the classification that each colored object on the field can fall into. These colors are:

- Orange signifying the ball
- Yellow signifying one team
- Blue signifying the other team
- White signifying the walls, field lines, and the orientation of the robots
- Green signifying the playing field
- Black signifying the physical robot covers

Of these six colors only four are considered to be interesting. They are orange, yellow, blue, and white. The other color regions of the image are thrown away. The location of the ball, and the locations and orientation of the robots are computed. The robots are identified and ordered to specify which robot corresponds to which team marker. The ball and robot states, composed of the position and orientation in the case of the robots, is filtered using a tracking/prediction filter to reduce the amount of measurement error that is inherent in the digitization of the image. The states of the ball and the robots are sent to the AI system over a UDP network connection for strategy processing and review.

The system hardware is comprised of:

- 1 Sony DXC-9000 3-CCD Color Camera with Zoom Lens
- 1 Matrox Genesis Image Processing Board with 64 MB of SRAM and Grab and Display Modules
- 1 Century Optics Wide Angle Adapter

The system software is comprised of:

- Image Segmentation
- Blob Analysis
- Robot Orientation Determination
- Robot Identification and Linear Filtering
- Network Functions to Transport Data Across UDP/IP Connection

The system is built up of 3 separate programs. They are the calibration program, the vision system proper, and the vision system display client. The calibration program allows for color thresholds to be determined in a graphical way. The vision system proper acquires and processes the digitized frames and disperses the resulting data. The vision system display client provides graphical visualization of the processed data from the vision system proper by acquiring data from the network and plotting this data in a window. The vision system display client requires no special hardware or software.

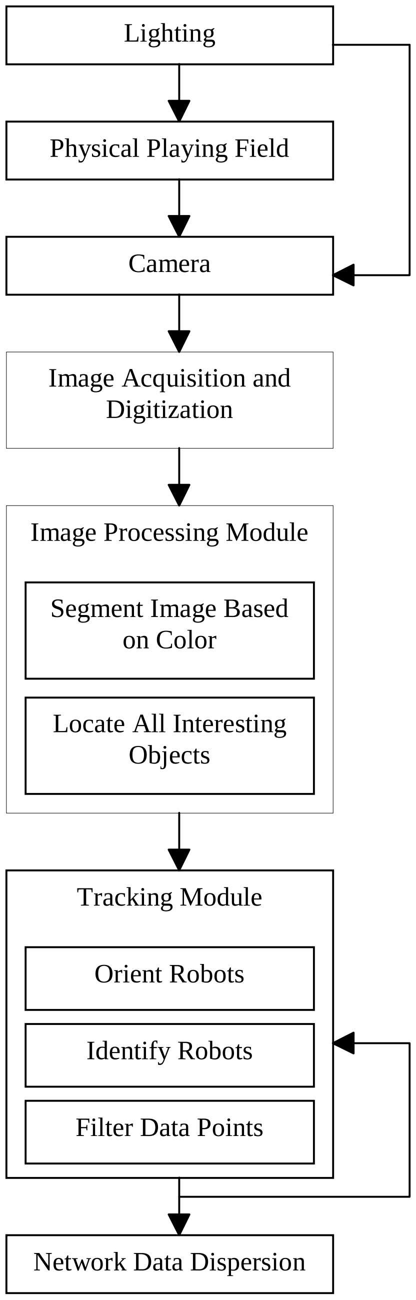


Figure 1. Vision System High Level Block Diagram

There are several modes of operation for the vision system. The processed data can be sent to either one or two artificial intelligence systems. It can also be sent optionally to an arbitrary number of vision display clients. Two artificial intelligence systems provide the capability for two teams to compete with the use of a single vision system. The system also allows for the use of difference images before segmentation to eliminate field noise, and also allows for the optional use of filtering of tracked objects.

Section 5 RoboCup Vision System Requirements

The vision system must be able perceive the current game state. This means that it must identify the objects on the field and track them in real-time with a minimal delay. It needs to separate the ball, two different team markers, and any additional robot markings from the rest of the field in a reliable way. Each robot on the Cornell team needs to be able to be identified uniquely in some manner, whether physically if the robots are visually heterogeneous or in software if the robots are visually homogeneous. This data needs to be transferred to the AI system. The vision system also needs to be implemented within the time frame given. The time frame is 9 months.

Section 6 High Level Vision System Functional Description and Analysis

The vision system was analyzed and the requirements were determined by, and influenced the requirements of the system as a whole. The camera was determined to be a single color camera, which can image the entire field at a height of at least 2.5 meters. The coloring of the additional markers on top of the robots was determined to be of a uniform color and the robots to be visually homogeneous. The vision frame rate was determined to be 60 Hz processing an entire frame (both even and odd fields) of data. Each robot needs to be able to have the orientation determined, and uniquely identified by the system. A separate workstation was chosen to process all of the vision data and the information needs to be sent to a separate computer over a network connection for all artificial intelligence processing. The operating system was chosen to be Microsoft Windows NT. The vision system needs to be able to be calibrated as conditions change and needs to have a user interface to allow for changing of the system state.

6.1 Global Vision Analysis

A global vision system was chosen to allow a suitable camera without size and processing constraints. The robot requirements specify that on-board robot vision requires that the robot height must be within 22.5 cm. An on-board vision system also requires either the camera data must be processed on the robot, or sent to a central processing workstation over wireless communication. Processing vision data requires a large amount of processing power and the processed data needs to be merged in some comprehensive way to allow for an organized strategy to be determined by the artificial intelligence system. To eliminate these problems, a single camera is used to resolve the state of the game. This eliminates both the need to process vision data and merge this data into a unified game state, and the need to process the data on the robots themselves. The implementation of this system also allows it to be completed and fully tested within the time frame given. The use of the single camera thus required that the camera have a high enough resolution

Table 1. Spatial Error

Dimentions of the Field		Length (meters)	Width (meters)
		2.74	1.525
Camera Resolution			
Height	Width	Resolution Along Field Length(meters)	Resolution Along Field Width (meters)
200	200	0.01370000000	0.00762500000
320	240	0.00856250000	0.00635416667
512	512	0.00535156250	0.00297851563
640	480	0.00428125000	0.00317708333
800	600	0.00342500000	0.00254166667
1024	768	0.00267578125	0.00198567708

$$I = R + G + B$$

$$I = R + G + B$$

$$I = R + G + B$$

and frame rate to locate the robots with minimal error. The error consists of two parts; spatial error and temporal error. To resolve object locations within a centimeter of spatial error, the camera resolution needs to at or greater than 320x240. This analysis is given in Table 1. The frame rate of the camera and top speed of the robots in intertwined. In order to increase temporal error that is inherent in the vision system the frame rate of the camera needs to be fast enough that the maneuverable area of the field by the robots is minimal, but yet still maintain a feasible frame rate. The maneuverable area is defined by the area that the robot can cover within two consecutive frames. This area is inversely proportional to the frame rate. This

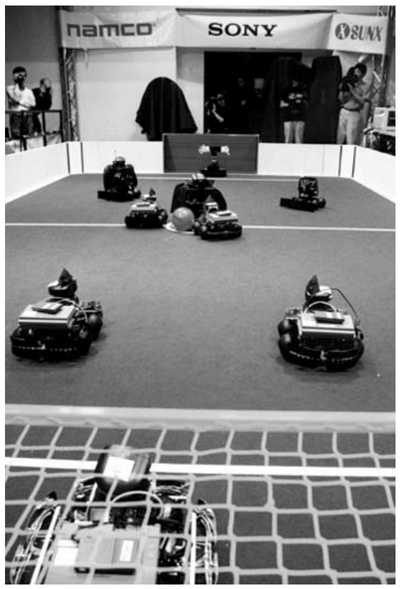
Table 2. Temporal Error

Camera Frame Rate	Area Covered Traveling at 2 m/s (meters^2)	Area Covered Traveling at 4 m/s (meters^2)
PAL: 25Hz	0.020096	0.080384
NTSC: 30Hz	0.013956	0.055822
60Hz	0.003489	0.013956

analysis is given in Table 2. The use of a single camera requires that the camera must be able to image the entire field at a height of at least 3 meters as specified by the RoboCup regulations. However, the RoboCup lab requires that the camera be able to image the entire field at a height of 2.5 meters. This fixes the lower bound on the lens field of view to be at 63.598°. This specifies that the camera lens must be a wide-angle lens. The wide-angle lens will introduce a fair amount of barrel distortion that needs to be compensated for.

6.2 Color Analysis

The RoboCup regulations specify all objects in terms of color. This and other RoboCup teams using color vision systems motivated the selection of a color vision system for use. A grayscale system will flatten the color information preserving the intensity information and eliminating the color information of the vision data. Specifically, a very small distance in grayscale space separates the color orange (ball) and the color green (field). Thus a grayscale system would be prone to segmentation errors due to noise in the color values of either the ball or the field. The distance between the two colors on a 255-color scale is about 20 divisions. This is insufficient because this is the most important differentiation that must be made. A typical image that may be acquired during the game is presented to the right. The ball and the field are not differentiable with a high degree of confidence. The conversion from the RGB color space to the grayscale color space is given by



$$I = R + G + B$$

This is a non-affine transformation. Colors that have been captured with a grayscale camera cannot be recovered. 3 CCD color cameras capture images based on the RGB color space. Upon converting orange (RGB coordinates of [255, 0, 0]), green (RGB coordinates of [0, 255,0]), and blue (RGB coordinates of [0, 0, 255]) to grayscale using equation 1, the intensity value of all of these colors is 85. Thus differentiating these three colors in terms of intensity is not reliable and highly prone to image noise. Grayscale images are also highly sensitive to lighting variations because this is the information that is captured by a grayscale camera. The colors that are present in an image of the field can be considered to be clusters in the used colorspace. In grayscale, these clusters are all located along a single axis since grayscale is a linear colorspace. In RGB colorspace, the clusters are located in 3-space and thus are spread about this three dimensional volume with more space between the colors. Since color cameras capture in RGB space this separation is preserved through the digitization of the color image. The conversion to grayscale does not preserve this information.

6.3 Color Based Segmentation and Tracking

To identify the objects on the field that are marked as interesting, color segmentation is done to separate the objects from the rest of the field and spurious objects that are located off of the field. The items of each color deemed interesting are separately processed and tracked. The segmentation procedure is required be robust to noise and lighting variations across the field. The tracking procedure is required to be invariant to occasional spurious data points and measurement error inherent in the image processing procedure. The tracking procedure is dependent upon the robot markings, and the identification of the robots. The tracking procedure allows for ball and robot prediction into future frames to compensate for delays in the system.

Table 3. Vision System Tasks

Image Resolution	640	480
Camera Frame Rate (s)		0.016667
Number of Channels	3	
Number of Colors	4	
	Color Thresholds without Difference Images	Color Thresholds with Difference Images
Image Processing		
Number of Difference Images		1
Number of Mask Generations		1
Number of Image Merges		1
Number of Color Thresholds	12	12
Number of Logical Operations	4	4
Number of Blob Analysis	4	4
Tracking		
Number of Ball to Track	1	1
Number of Cornell Robots to Track	5	5
Number of Opponent Robots to Track	5	5
Number of Orientation Blobs to Track	5	5
Total Number of Objects to Track	**16**	**16**

6.4 Robot Marking

The vision system processes in linear time with respect to the number of colors that are determined to be interesting. To decrease the vision processing time, any additional markers that are placed on top of the robots are to be of a uniform color. This method also leaves enough separation in colorspace that the primary colors can be reliably segmented. If each robot is marked with a separate color that uniquely determines the robot, five new secondary colors are now searched in the colorspace decreasing the distances between color clusters in the colorspace. For reliable and robust color separation, these distances need to be maximized. The robots involved are not all bi-directional and capable of turns of arbitrary radius. To determine the direction that the robot is facing, an orientation marker is placed on the top of the robot that allows the vision system to realize the direction that the robot is facing. The use of the path that the robot is traveling is not appropriate for orientation determination since the robot can change orientation without traversing a path. The robot is also able to travel in both forward and backward directions. Since not all robots are bi-directional, the robots are required to have a specific direction at times during game play. This direction cannot be determined from the forward and backward motion of the robots since the single marker on the robot is a radially symmetric ping-pong ball placed on the top of the robot. A second marker is used to eliminate the symmetry of the top of the robot and be able to determine the orientation of the robot in question.

6.5 Robot Identification

Commands from the artificial intelligence system are broadcast to the robots and each robot receives a command that is determined by that robot position on the field. In order to ensure that the commands are sent to the correct robots on the field, each robot that is being tracked needs to be identified. A mapping is required to be performed between the blobs that are located by the vision system to the numbers that the robots have been programmed to respond to, and the ordered list that is stored in the AI program. The blobs that are located by the image processing are unordered and initially arranged as they are found by raster scan order.

6.6 Data Dispersion to Vision Clients

The vision system is separated from the artificial intelligence computers and is to be dispersed to the AI computer and any other systems that processing the vision data. The possible use of multiple clients necessitates a communication system that allows for multiple computers to access information from the vision computer. A local area network is built from the vision and artificial intelligence computers to allow for the efficient transmission of data to all clients. This system allows for multiple computers to be physically connected to the vision system. The communication link from the vision system and clients needs to be built on top of this physical layer. A serial or parallel line allows for only a single computer to access the data from the vision computer, whereas the network does not contain this property. An Ethernet link was decided based upon the high bandwidth and availability of device driver abstraction. The Matrox libraries are compiled for both DOS and Windows NT, but Windows NT also contains the networking protocol suite for UDP/IP. Thus Windows NT was the operating system platform selected. The use of Microsoft Windows NT also eliminates direct communication to the ports of the computer

Section 7 Hardware Determination and Analysis

In order to capture images at an update rate of 60 Hz, the Sony DXC-9000 was chosen. This camera has a resolution of 640x480 and updates at 60 frames per second, using a full frame transfer. The camera transmits data using a VGA signal. While the VGA signal is analog, the data that is used to create the analog signal is digital, thus the SNR is very small. The processing of the data is dependent on the amount of data that is to be analyzed. At the target frame rate, dedicated hardware was determined to be needed in order to process at a high enough bandwidth. A frame rate of 60 Hz is fast enough that operating system scheduling and efficient algorithm coding is a major concern when processing on a computer. To connect the camera, the Matrox Genesis board was selected because a 60 Hz transfer rate has been tested and verified with the Sony camera. An alternative DSP board, the Coreco Cobra/C6, was not able to verify the compatibility of the board with the Sony camera running at the frame rate. The Matrox genesis board also included an extensive library of image processing function calls. The hardware specifications are included in Appendix B.

Section 8 Algorithmic Constraints Due to Hardware Selection

The selection of the Matrox Genesis DSP board added a significant number of constraints to the determination of the image processing algorithms. Each call to the Genesis board has an overhead of .5 ms.

With this overhead, 32 calls to the DSP board will result in the entire processing window being wasted on overhead. The result is that the number of calls to the image processing board must be minimized. The speed and efficiency of the image processing functions are also dependent on the type of data that the image buffer contains. Binary type buffers need to be used as often as possible to increase the speed of processing. Also, any colorspace conversions, while attractive, cannot be computed on the board at the specified frame rate. The price of the vision system hardware prohibited the purchase of a system for each team. Thus the system must be able to accommodate visual processing for both teams. This includes the dispersion of the data to two computers simultaneously.

Section 9 Algorithm and Network Protocol Determination

The vision algorithms consist of an image processing module and a tracking module. The image processing module is comprised of an image segmentation stage and a blob analysis stage. The tracking module is comprised of blob orientation, identification, and filtering stages.

9.1 Image Segmentation

The image processing stage of the vision system answers the question of what is in the image, and where is the object located in the image. The image processing algorithms need to be developed with attention based toward optimized speed and accuracy. Each frame of data contains 921,600 bytes of data. Thus the system must be able to have a throughput of at least 52.7MB of data per second. A single pass through the image requires a throughput of 105.4MB of data per second. Several image processing algorithms were examined during the algorithm determination. Each algorithm was considered for speed on the Matrox Genesis board. These include color histogram backprojection with blob aggregation, an distance classifier with thresholding, and color thresholding. The color thresholding algorithm was selected for both feasibility and execution speed.

9.1.1 Color Histogram Backprojection with Blob Aggregation

The color histogram backprojection algorithm is set forth by Michael Swain and Dana Ballard in [10]. This algorithm is the equivalent to a correlation of the object model and the image in histogram space. This algorithm identifies the objects that have a similar color histogram to the model images and then localizes the blobs based upon the size of the blob of a certain color.

The algorithm consists of two parts: a ratio histogram is projected onto the image and then a box sum is convolved across the entire image. Both the model (here images of the ping pong balls, and the golf ball) and the image multidimensional histograms are computed. The histograms are computed in the *rg-by-wb* color space.

Color	Binary Value From Color Thresholding (R G B)
Black	(0 0 0)
Blue	(0 0 1)
Green	(0 1 0)
Cyan	(0 1 1)
Red	(1 0 0)
Magenta	(1 0 1)
Yellow	(1 1 0)
White	(1 1 1)

The ratio histogram is defined to be:

$$R_i = \min\left(\frac{M_i}{I_i}, 1\right)$$

where R_i is the histogram ratio, i is the bin number, M_i is the model histogram, and I_i is the image histogram. This ratio is indexed based on the number of bins in the histogram. This ratio histogram is then backprojected onto the image by replacing the image value with the value of R_i that the image point references. A box sum is then convolved across the image and the blob centers produce peaks in the image. The algorithm is executed for four different models. They are the ball, the two team colors, and the orientation color. After the computation of the box sum, we will be looking for the maximum value when searching for the ball, 5 peaks when we are searching for any of the other target objects. The output from the image processing algorithm will be the locations of the 16 blob centers that are found in the image.

The histogram backprojection algorithm is robust, yet computationally intensive. At a resolution of 640x480 the golf ball comprises a width of nine pixels of an area of approximately 68.55 pixels^2. The sizes of the ping pong ball in the image are of a small size ratio when compared to the rest of the image; approximately $2.23175\text{x}10^{-4}$. This means that the peaks that are determined by the algorithm are not significantly above the noise floor that is produced by the rest of the image. Conversion of the captured RGB space image to the rg-by-wb color space is prohibitively expensive and thus cannot be computed. Maintaining the RGB color space implies that the backprojection of histogram ratios back onto the image involve a three-dimensional look up table. The size of this table is 256^3 = 16777216 entries. Considering the anticipated non-uniform lighting at the competition across the field, histogram backprojection will possibly fail to locate robots in brighter of darker areas of the field. RGB space is highly sensitive to lighting variations and thus the histograms of similarly colored objects on different portions of the field will not coincide with the model histogram, evaluating to a low match value. The algorithm requires for each color to localize:

- One Histogram Mapping
- One Image Division
- One Minimum Calculation
- One LUT mapping
- One Convolution (Box Sum)
- One Peak Determination

9.1.2 Distance Classifier with Thresholding

A distance based classifier was considered for implementation to classify each pixel by the distance metric to each of the expected values all the colors marked as interesting. The distance metric used is defined as

$$D = \sum_{3} |P - T|$$

To eliminate the classification of uninteresting colors to those colors that are marked as interesting a threshold is implemented. The classifier is implemented using the following equation

$$\begin{Bmatrix} d = \min(dist_i) & , d < threshold \\ 0 & , otherwise \end{Bmatrix}$$

The classifier involves three steps. First the distance of the pixel color to all target colors is computed. Then the pixel is classified based on the target class that it is closest to. If the distance is greater than a maximum distance threshold, the pixel is thrown away. This classifier can be achieved using a radial bias or k-means classifier for classification target vectors that vary through time. This algorithm requires:

- Three Image Subtractions with Absolute Value for Each Color (One Subtraction for Each Color Channel)
- One Triadic Image Addition for Each Color (One Addition for Each Color Channel)
- One Minimum Calculation
- One Binary Thresholding

For the minimum of three colors that need to be localized this results in:

- Nine Image Subtractions with Absolution Value
- Three Triadic Image Additions for Each Color
- Three Minimum Calculations
- Three Binary Thresholds

As can be noted, the algorithm runs in time O(n) in the number of colors that are classified. This algorithm cannot be completed in the target amount of time.

9.1.3 Color Thresholding

A color thresholding algorithm was selected as the image processing algorithm for both execution speed and feasibility. Each color channel is thresholded in RGB space and classified by whether or not the pixel is contained within the cube that is delimited by the thresholds. This algorithm is well suited since the colors delimited by the RoboCup federation are well separated in color space. This algorithm produces binary type data buffers quickly to allow for increased processing speed further into processing. The algorithm requires:

- Three Binary Thresholds for Each Color
- One Logical Operation for Each Color

For the minimum of three colors that need to be localized this results in:

- Nine Binary Thresholds
- Three Logical Operations

The color thresholding segmentation procedure can be shown to be equivalent to the distance based algorithm with a redefinition of the distance metric that is used. The contours of the original distance based classifier produce a cube in the colorspace as shown in figure 3, projected on to a plane for visual clarity. The color thresholding procedure produces the same contour lines rotated to align the sides of the cube to be parallel to the axes of the colorspace illustrated in figure 4. The original classifier deviates from the norm 2 distance (Euclidean distance) by at most 9.3%. The color thresholding classifier deviates from the norm 2 distance by at most 44.2%. These deviations are located at the points that radiate out from the center of the contour at 45° from the axes. All of the colors that are defined by the RoboCup Federation are located at corners of the RGB color cube (Figure 5) and the shortest distances between any two colors are along the color axes where this deviation is minimal. In the ideal case of marker coloring the entire colorspace can be split into eight regions using a single segmentation along each axis. In the segmentation of the colorspace into eight regions, each color then occupies a volume that results from the splitting of the plane with 3 planes. Thus each color band only needs to be segmented a single time and the colors can be uniquely identified by the binary values given in Table 4. Each term refers to whether the volume is in the positive of negative half of the cube along the specified axis (Figure 6). The color thresholding requires much less processing time than the other distance classifier and due to the tight timing constraint fits into the time window. While a maximum of nine thresholds are required, several thresholds can also be merged together to reduce the number of binary thresholds from nine to a much smaller

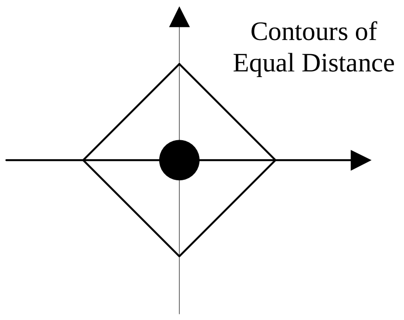


Figure 3. Contours for Norm 1 Classifier

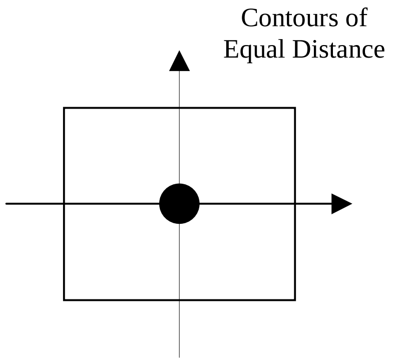


Figure 4. Contours for Color Thresholding

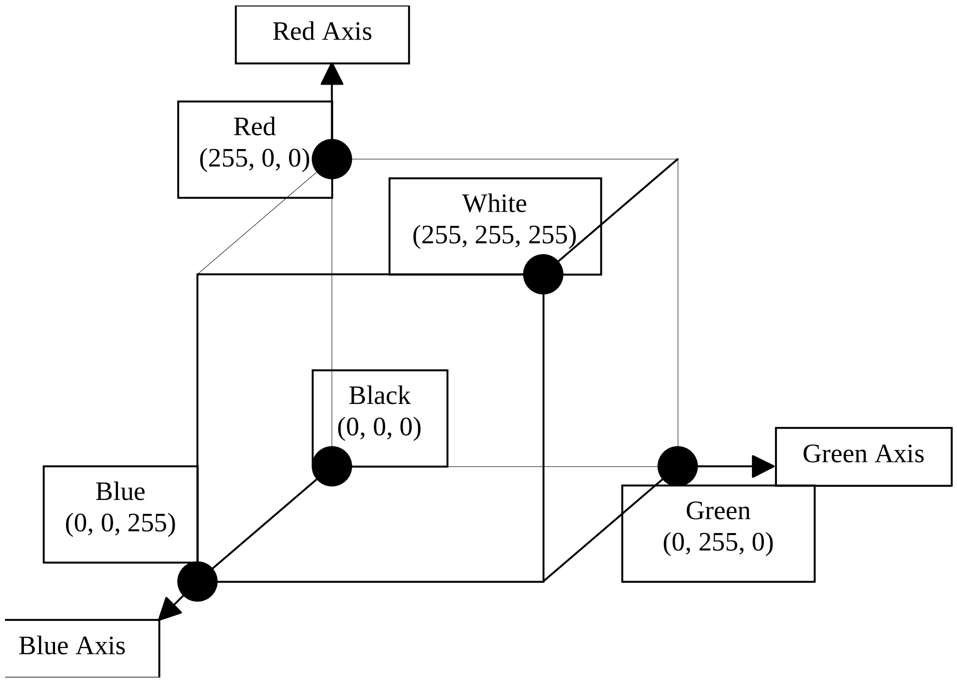


Figure 5. RGB Color Cube with RoboCup Colors

number, with a lower bound of three. This is dependent upon the actual coloring of the objects. To allow for variations in the coloring of objects the thresholds are set during the calibration procedure and are loaded into the vision program during runtime and only close thresholds are merged together allowing for volume overlap. However algorithmic optimizations use all nine thresholds for the primary colors. This also allows for not having to merge thresholds, and more robust calibration of the system, at the price of a speed hit.

Table 4. RGB Binary Values

Color	Binary Value From Color Thresholding (R G B)
Black	(0 0 0)
Blue	(0 0 1)
Green	(0 1 0)
Cyan	(0 1 1)
Red	(1 0 0)
Magenta	(1 0 1)
Yellow	(1 1 0)
White	(1 1 1)

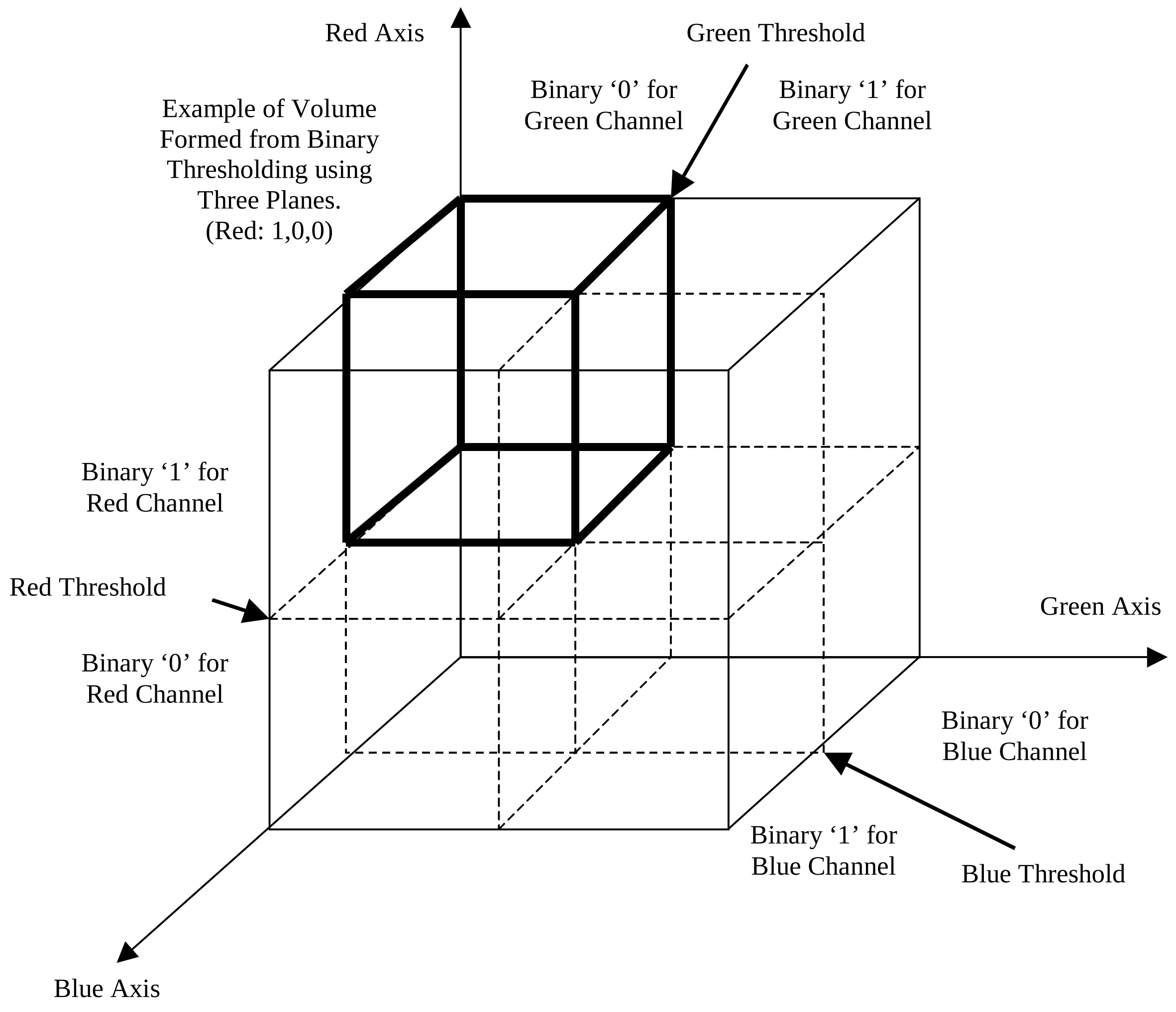

Red Axis
Green Threshold
Binary ‘0’ for Green Channel
Binary ‘1’ for Green Channel
Example of Volume Formed from Binary Thresholding using Three Planes. (Red: 1,0,0)
Binary ‘1’ for Red Channel
Red Threshold
Binary ‘0’ for Red Channel
Green Axis
Binary ‘0’ for Blue Channel
Binary ‘1’ for Blue Channel
Blue Threshold
Blue Axis

Figure 6. Example of Binary Coding of Sub-volumes based on Single Band Thresholds

9.2 Blob Analysis

All regions that have been classified as belonging to a specified color result in binary blobs in the blob identifier image for the colors that are considered interesting. Blob features are computed to enable further processing of the binary identifier image. These features are center of gravity in both x and y coordinates, blob area, and blob perimeter. The center of gravity for each blob identifies the location of the blob in image coordinates, the area and perimeter allow for the computation of the compactness of each blob. During blob analysis all pixels that evaluate to a binary one in the identifier image are connected to the neighboring pixels, and the features are computed for these connected components.

9.3 Blob Size and Shape Filtering

The list of blobs that is the result of the image processing is filtered by both size and shape parameters. Due to the fact that the markers for all objects on the field are of a specified size, any blobs that are not within the size constraints are thrown away as noise. Every marker is also either circular or square in shape. To further filter noise from the image processing step, the compactness of all the blobs are computed from the perimeter and area features of each blob. The compactness of both a circle and a square of ideal shape is approximately 1. Any blobs that differ significantly from this compactness are thrown away as spurious data points. This step eliminates a significant portion of the noise in the image.

9.4 Orientation Determination

A separate marker that is placed on top of the robots, eliminating the symmetry of the robot covers, determines the orientation of each robot. Following blob analysis, each blob that is classified as an orientation marker is registered to the blobs that are classified as team markers for the team that is being tracked. The registration performs an exhaustive greedy search of all combinations of orientation and team marker blob positions and any orientation marker that is within the specified distance to a team marker is registered to the corresponding team marker, resulting in an unordered list of oriented robot blobs.

The relative locations of the team markers and the orientations of the markers determine the orientation of the robot. In order to accurately determine the orientation of the robot, the team marker and the orientation markers are to be spatially located as far from each other as possible within the bounds of the robot covers. The position of the markers on the robots relative to the actual orientation of the robot is dependent upon the shape of the robot covers that are used. The robot covers for both teams are described in the figure to the right (Figure 7). The orientation markers themselves are sheets of paper that are placed on the covers of the robots. To change the color of the orientation marker, a new sheet of the specified color must be printed or purchased and cut to proper size. Ping pong balls, those used for the team markers, are not used since the ping pong balls would be required to be painted to match certain colors and the color of paper is easier to control. The use of paper affects the robot orientation due to parallax error in the images. The physical height of the paper on the robot colors and the height of the robot marker are not in the same horizontal plane and thus requires that the parallax error be corrected prior to determination of robot orientation. Robots are oriented using the formula

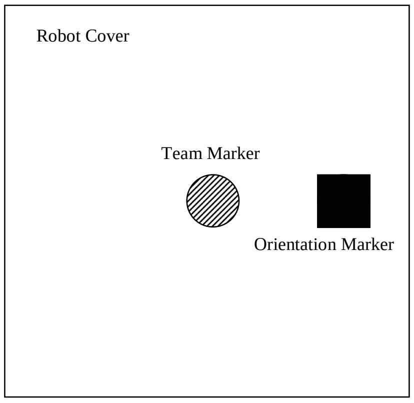


Figure 7. Robot Cover Showing Orientation

$$\theta = \tan^{-1}\left(\frac{y_m - y_o}{x_m - x_o}\right)$$

The orientation is performed with respect to the positive x-axis in the image coordinates and is independent of the placement of the markers on the robot covers. The corresponding robot dependent orientation is resolved in the artificial intelligence system. The result of the orientation stage is to produce a list of robot states

9.5 Robot Identification

Because all of the robots are visually homogeneous the individual robots need to be identified to ensure that the correct commands are sent to the appropriate robots. The identification step uses the temporal continuity that is inherent in the physical state of the system being tracked. The robots can only traverse a small portion of the field between frame grabs and this information is exploited in the identification step. The unordered list of team marker blobs is registered to an ordered list of team marker blobs that was computed based on the information in the previous frame. This registration is performed using an exhaustive greedy search algorithm, which attempts to register all team marker blobs with the identified robots from the previous frame based on a distance measure. For each of the robots on the field, the distance to all of the unidentified team marker blobs is computed and the blob with the minimum distance is registered to the robot in question. The blob is considered to be identified and the robot state is updated in the system. The difference between the robot position in the current frame and in the previous frame is used to compute the robot linear and rotational velocity. This is stored in the ordered list of robot profiles.

9.6 Linear Filtering and Prediction for Robot Identification

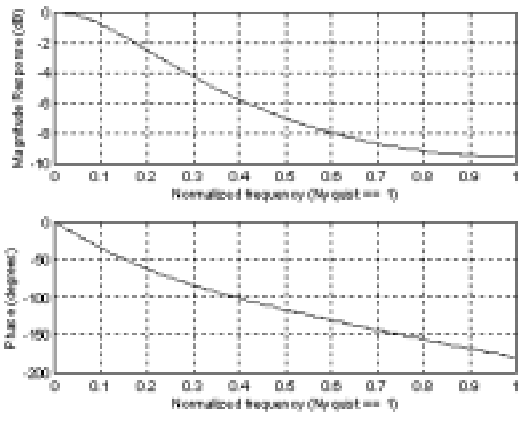
Inherent in the image capture, digitization, and processing; noise in introduced into the system. This noise is due to pixelation, sensor noise, and segmentation noise. The results of the processing can then be modeled as x'(t) = x(t) + ν(t), where ν(t) is gaussian white noise. To eliminate this noise a linear tracking filter is used to smooth the position and orientation information in both the ball and robot states. The fundamental trade-off of this increased accuracy in the state information is the delay that is due to sudden changes in the state information. As the standard deviation of the noise in the state information is reduced, the lag in the tracking filter is increased. However, since a single pixel shift in the location of an orientation blob produces a 10-15° shift in the orientation of the robot, the filter is necessary to provide accurate object profile information to the artificial intelligence system. The filter selection for the ball was chosen to be a linear prediction filter since the ball has constant velocity and no input forces when traveling freely across the field. A Kalman filter was also considered for tracking of the robot states since the robots can travel in non-linear patterns on the field and have input forces that are subject to the commands that they are given. The Kalman filter was not selected due to increased complexity and computation required of the filter, time to project completion, and speed of the tracking system. Kalman filtering requires approximately twice the computation time of the linear filtering algorithms and the system tracking rate is fast enough that the system position and velocity profiles are linearized about the current time in the system. The filtering computes the position and velocity information and this is kept in the system as an object profile.

The linear tracking filter equations are

$$\dot{y}[n] = \dot{y}[n-1] + \frac{h}{T}\left(x[n] - y[n-1]\right)$$
$$y[n] = y[n-1] + g\left(x[n] - y[n-1]\right)$$

The propagation of the filtering equations from the previous frame allows for a predicted robot location in the current frame. Comparing all of the current robot states to the predicted robot profiles identifies the robots. The match is based on the distance that the robot has traveled, and the change in velocity between the two frames. Thus the robot identification is dependent on continuity in both robot position and robot velocity. The current robot velocity is measured as the distance between the position of the robot in the previous position and the current team marker blob that is being considered. The result of the identification using the tracking filter is to keep a point on the field that from being identified as a robot while a robot moves past that point. The distance that the point travels from one frame to the next is nearly zero and thus would win in the greedy algorithm that only uses the position information from one frame to the next.

The identification stage of robot tracking is sensitive to errors from the image processing. If a team marker is not segmented in the image processing module, due to such errors as severe color distortion or improper calibration, The robot profile for the current frame is interpolated from the previous robot profile. This allows for a reasonable approximation to the correct robot profile since the robots are changing slowly with respect to the frame rate of the vision system. During the next frame, the interpolated robot profile is used to search through the robot states to determine the identification. If an appropriate robot state is not found for five frames the robot profile is considered to be invalid, and the tracking for that robot is dropped.



Robots that have been dropped from tracking are reacquired based on oriented robot states that do not register to any robot states that are currently being tracked. Once all of the team states have been registered and identified, any remaining robots that should be on the field but are not being tracked are assigned a robot state, and the robot profile initialized to the current position of the team marker on the field. Since the robots move with arbitrary motions about the field, there is no reliable way to assign the robot identification to the robot state that is under consideration. Thus the identification of a robot number to the robot state is done in an arbitrary way. Hence, if two separate robots are not segmented properly and have been dropped from tracking, the reassignment of robot numbers to these robots is done by which robot position is first in term of raster scan order of the field image. This is the main cause of robot misidentification.

The linear filtering[2] of the robots reduces the amount of noise in the robot positions and orientation by eliminating high frequency components of the change in data. As the position of the input to the linear filter varies through time, the filter generates a value that represents an approximation based on the current and all previous measurements of the position or data. The bandwidth of the filter determines the amount of noise that is allowed to pass through the filter. The decrease in bandwidth, while reducing the variance of the output, causes a lag in the estimated value for position or orientation. This is apparent in the phase response of the filter. Thus, sharp changes in data are reflected in the output after a significant amount of time. The fact that the filter is second order necessitates that the response of the filter contains dampening that needs to be accounted for.

The filter performs without steady-state error for the case of constant-velocity of the tracked parameter. While the ball has constant velocity for times when the ball freely rolling along the field, the robots do not usually have constant velocity during game play. During game play, the velocity of the robots is constant because of the frame rate that the vision system runs at. Expanding the position of the robot in one dimension in terms of its Taylor Series produces

$$x(t) = x(t_n) + T\dot{x}(t_n) + \frac{T^2}{2!}\ddot{x}(t_n) + \ldots$$

The robots are designed to have maximum accelerations of around 3m/s^2. For the anticipated update rate of 60 Hz, this results in the acceleration term for the expansion being

$$\frac{T^2}{2!}\ddot{x}_{\max} = \frac{(1/60)^2}{2!}(3 \, {}^{m}\!/_{s^2}) = 4.166666\times10^{-4}\,m$$

This term is small enough that the acceleration of the robot is not necessary as one of the filter states. For the case of a constant acceleration, the filtered position and orientation will lag behind the actual value of the robot by a value of

$$b^* = -\frac{\ddot{x}T^2}{h} = -4.990\times10^{-3}\,m$$

the result of the truncation of the acceleration term in the Taylor Series expansion.

The transient error of the resulting signal is the sum of all the lag errors for a step change in velocity given by

$$D_{y_n} = \sum_{n=0}^{\infty}(y_n - x_n)^2$$

The total error is thus the error due to input noise and the filter lag and given by the cost function

$$Error = VAR(y_n) + \lambda(D_{y_n})$$

where the λ is a Lagrange multiplier that determines the importance of each portion of the error. The resulting filter gains are related by

$$h = \frac{g^2}{2-g}$$

9.7 Network Protocol and Model

For efficient transmission of data across the network connection with minimal delay, the UDP transmission protocol was selected. UDP is a member of the TCP/IP network layer protocol suite and is widely available and included in the Windows NT socket libraries. Opposed to the TCP/IP transmission protocol, the UDP protocol does not provide for either guaranteed data delivery or acknowledgement of the data being received at the destination socket. TCP/IP was not selected due to these features that provide for reliable data transmission. For any data to be transmitted across a TCP connection the receiver is notified and an acknowledgement is returned. Then the data is packetized and each packet is sent over the network in turn once the reception of the previous packet is acknowledged. This form of reliable delivery is very slow and does not fit our needs since the calls to transmit the data blocks all processing on the transmitting and receiving machine. The vision and the artificial intelligence computers are connected together using a local area network that can be disconnected from all other network traffic and computers. This eliminates a significant amount to network noise, and completely eliminates all networking problems such as dropped packets due to misrouting, maximum number of hops, and so forth. The UDP protocol is thus sufficient for the transmission of data from one computer to an adjacent computer on the network.

The communication between the vision system and the artificial intelligence computers is performed using a client server model, where the artificial intelligence computer places a request for data and the vision system transmits the data to the address that requested the data. This method allows the artificial intelligence system to obtain the most recent information that is available from the vision system. Other problems such as network buffer backup are also eliminated using this model. The network buffer may be backed up if the vision computer simply sends data without the artificial intelligence system being ready to read the data from the buffer. This occurs when the artificial intelligence program is started after the vision system and does not read data from the network buffer during the initialization.

Section 10 Detailed Algorithmic Analysis and System Implementation

Image capture and processing is done on the Matrox Genesis DSP board. This includes the image capture, image segmentation and blob analysis using the Genesis Native Libraries. The tracking is done on the host processor. Data dispersion over the network is also done on the host computer. For each network connection that the system is expecting, a separate thread is used to disperse the data.

The vision system uses four main structures to organize the data that is processed. These four main structures are:

- `genesisStruct`: All devices, threads, and buffers that are used on the DSP board for image processing.
- `blobAnalysisStruct`: All threads, buffers, and result structures that are used for blob analysis
- `trackingStruct`: All structures and arrays that are used for object tracking and filtering
- `calibrationData`: All system parameters resulting from calibration. This holds all constants that are dependent on the environment and which can change.

10.1 System Initialization

The system is calibrated on initialization using a calibration file that is loaded during system start up. The file contains all of the necessary system calibration parameters. These include the color thresholds for the ball and the two robot colors, the starting and stopping positions of the image window, the starting and stopping positions of the field in the image, the number of robots on each team, the physical scale factor to convert pixels to meters, the camera height in meters, and the robot height in meters. Many of these parameters can be set once and do not need to be changed. The calibration file format can be found in Appendix C. The remaining system parameters are computed using these values taken from the calibration file.

To eliminate unnecessary processing time in the vision system loop, all buffers that are used for image processing on the DSP board are allocated beforehand. Buffer allocations on the board are synchronous function calls, and block system execution until the required memory has been allocated. The size and type of these buffers is known and thus all buffers are allocated at system startup.

The blob analysis features are also declared in the system initialization. These features are added to the blob analysis feature buffer. The structure used to transfer the data from the DSP board to the host computer are declared in an array that is large enough to hold the amount of data that is ever expected to result from the blob analysis stage. The blob analysis module is also instructed to ignore run information and to use 4-connected to connect neighboring pixels. A run is a horizontal sequence of blob pixels. The 4-connected connection scheme defines pixels as being connected if neighboring blob pixels are to the top, bottom, left, or right of the pixel in question. Blob pixels that are on the diagonal are not considered to be connected.

The tracking information is initialized to zero initially and there are no objects that are currently being tracked by the system. During system startup, the user is allowed to optionally select objects that are on a single captured frame to initialize tracking information and assign initial robot identification numbers. If this step in omitted, then the system assigns robot identification numbers based on the order in which the objects are located in the image.

During system initialization, all network threads are allocated and started. The threads are used to service network requests that are presented to the vision system and to perform data interpolation if necessary.

10.2 Image Capture and Preparation

The image capture of the frame from the camera is done in a double buffering fashion. This allows for a frame to be captured into memory on the DSP board while processing takes place. Without double buffering, the capture of a frame of data needs to be done after processing of the data is completed. As the frame is captured into memory, processing of data is halted.

The vision system uses a circular buffer containing two separate buffers for the acquisition of data. More buffers can also be used. After the completion of processing a buffer of data, the input buffer is marked as free and the next frame grab is initiated into the free buffer. The second buffer now contains new data and marked as the working buffer, and the processing moves to the working buffer. Since the image processing loop takes more time to complete then the transfer of data from the camera to the board, the vision system does not need to wait for the completion of the frame grab and immediately begins processing to new buffer of unprocessed data.

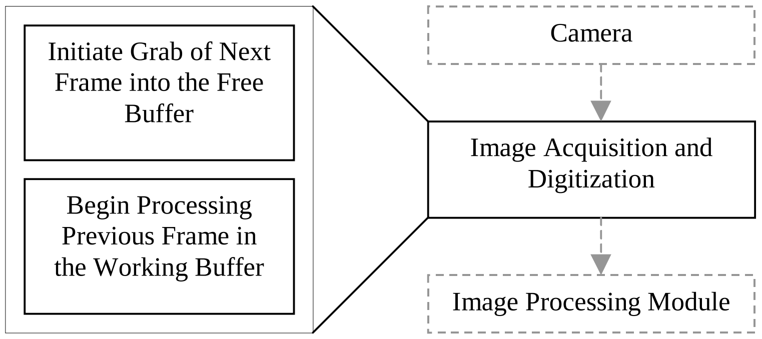

Initiate Grab of Next Frame into the Free Buffer
Begin Processing Previous Frame in the Working Buffer
Camera
Image Acquisition and Digitization
Image Processing Module

Figure 8. Image Acquisition and Digitization Breakdown

Since the field only takes up a portion of the image, the field portion of the captured frame is windowed and used for processing. The location and dimensions of the window is located in the calibration information that is read at system initialization. This windowed image is stored as a child buffer of the original image. Separate child buffers of the windowed child buffer provide access to each of the color channels. This is illustrated in Figure 9. The child buffers allow for processing of image regions and bands without needing to transfer image data to a separate buffer, increasing the system throughput.

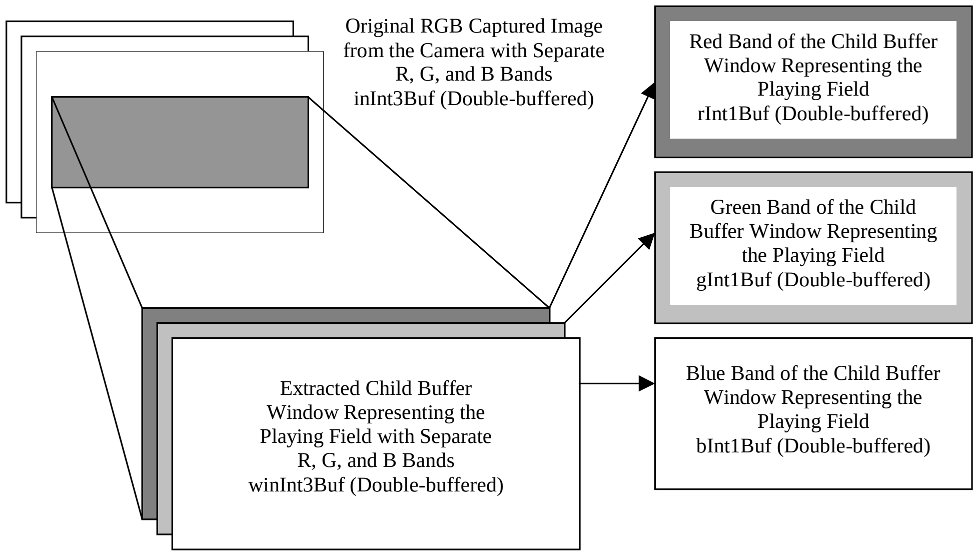

Original RGB Captured Image from the Camera with Separate R, G, and B Bands inInt3Buf (Double-buffered)
Red Band of the Child Buffer Window Representing the Playing Field rInt1Buf (Double-buffered)
Green Band of the Child Buffer Window Representing the Playing Field gInt1Buf (Double-buffered)
Blue Band of the Child Buffer Window Representing the Playing Field bInt1Buf (Double-buffered)
Extracted Child Buffer Window Representing the Playing Field with Separate R, G, and B Bands winInt3Buf (Double-buffered)

Figure 9. Image Windowing and Band Extraction

10.3 Color Based Segmentation

The color based segmentation performs segmentation using the windowed buffer from the capture buffer which has been marked as the working buffer in the acquisition stage. The segmentation allows for the optional computation of a difference image to mask the parts of the field that are not the objects that are attempting to be tracked. An image of the field without robots and without the ball is captured and stored in system memory. The current frame is then subtracted from this objectless image and absolute value is applied. Regions of the difference image that are large in any of the color bands are regions where there are high color differences. This is a good indication that the ball or one of the robots is present at that region. The camera flicker typically will not allow unchanged regions of the image to have a difference image value of zero. The difference image locates the positions of the ball and robots on the field, but will not produce a constant color across the image. The colors of the objects that are in the difference are dependent on both the color of the object and also the color of the area in the reference image. For example, an orange ball will produce an orange object when it is subtracted from the dark field, but will produce a cyan colored object when it is subtracted from the white wall.

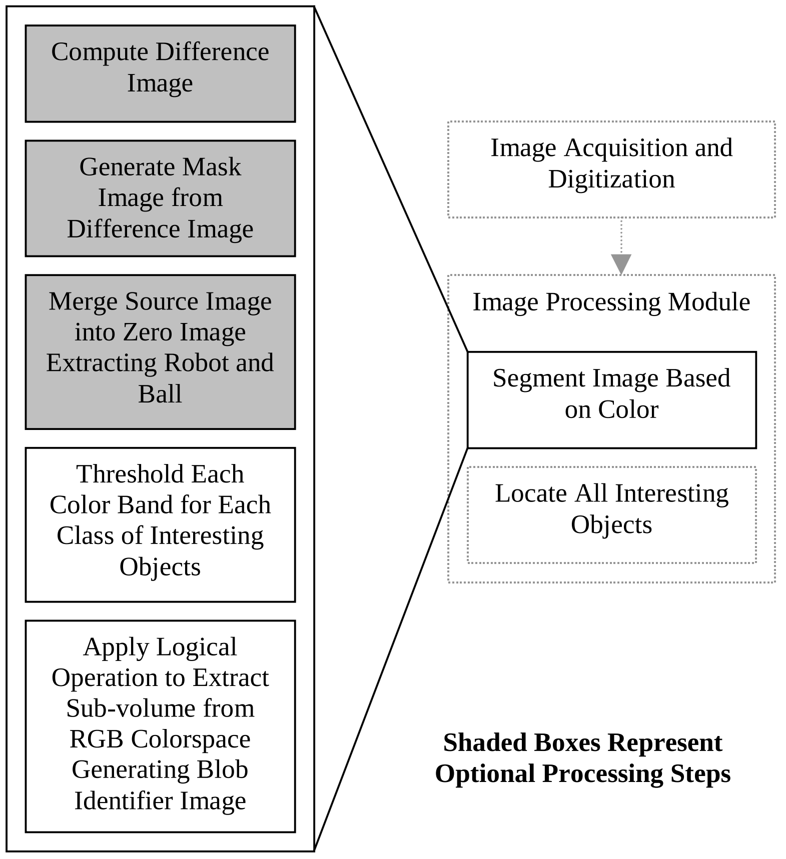

Compute Difference Image
Generate Mask Image from Difference Image
Merge Source Image into Zero Image Extracting Robot and Ball
Threshold Each Color Band for Each Class of Interesting Objects
Apply Logical Operation to Extract Sub-volume from RGB Colorspace Generating Blob Identifier Image
Image Acquisition and Digitization
Image Processing Module
Segment Image Based on Color
Locate All Interesting Objects
Shaded Boxes Represent Optional Processing Steps

Figure 10. Segmentation Step Breakdown

$$Field[79,83,96] - OrangeBall[255,74,73] = Orange[176,9,23]$$
$$Wall[255,255,255] - OrangeBall[255,74,73] = Cyan[0,181,182]$$

Thus the color constancy needs to be resolved to ensure that the colors will remain constant as the objects move across the field. To resolve this issue, regions of the difference image which indicate objects are replaced by the respective portions of the source image sing a binary mask generated from the difference image. Once the difference image are formed, a binary image where the pixels that above a threshold are set to 0x00 and those below are set to 0xFF is generated from the difference image. The image that is generated is a binary mask, which is color channel dependent. In order to generate a mask that is independent of the color bands, these masks are merged together using an AND operator. This produces a single band mask where at least one of the color bands differ from the reference image by at least the threshold are set to 0xFF. Values of 30 to 80 work well to eliminate most of the static image from the difference image. Once the mask image is generated, the source image is merged with a constant image of value 0x00. This produces an image where regions of the source image that contain color differences from the robotless reference image pass the source image, and the remainder of the image is constant 0x00. This produces a merged image such as shown in figure 11.The result of the difference image is then used for the color thresholding. If difference imaging is not used, then the image processing begins with the color thresholding. The color thresholding process converts each of the integer (8-bit) color band images into a binary image based on the threshold for the color band. The binary images allow for faster processing of data due to the reduced data throughput by a factor of eight. Each color that is interesting is segmented separately from the other colors. The bands are also thresholded using different values to allow for different thresholds to be used for each color. The resulting binary images designate the blobs in the color band where the regions fall into the threshold volumes on each band. The results are shown in Figure 13 below.

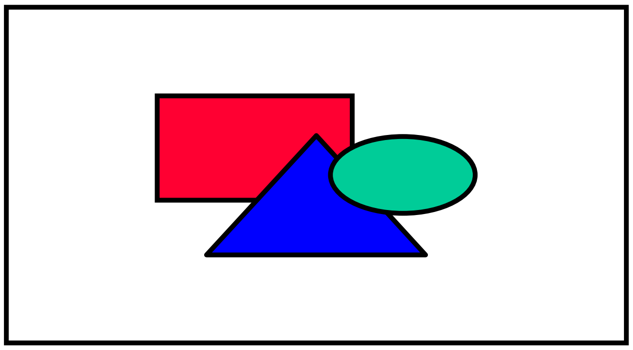


Figure 11. Difference and Source Image Merge Result

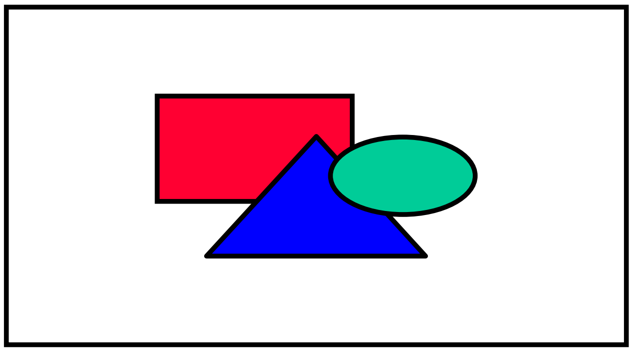


Figure 12. Result of Logical Operation

The colored regions can be extracted using a simple logical operation that selects either the positive or negative parts of the binary images depending on the color. This is shown in Figure 12 to the above. The thresholding produces 12 binary images, one for each color channel for each interesting color. To perform the logical operation for all the interesting colors results in four logical operations, where the Boolean function to be performed is taken from Table 4 in Section 9.1.3. The opcodes for the separate logical operations were derived to perform the necessary function since the operations are not defined in the Genesis library. The Texas Instruments’ C80

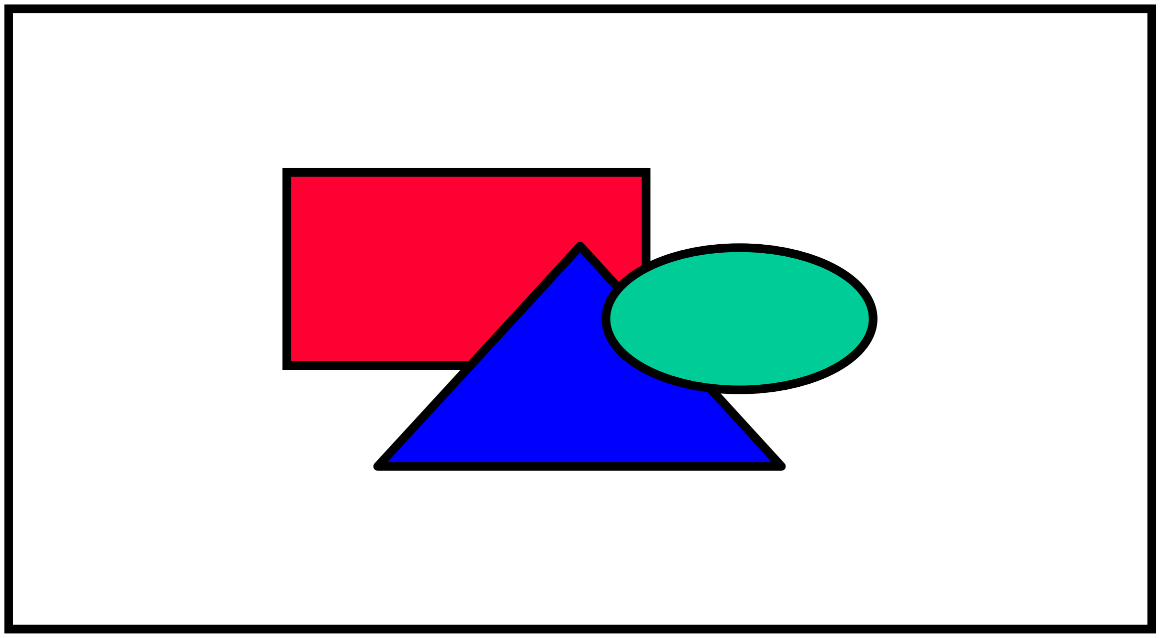


Figure 13. Results of Binary Thresholding for the Ball

processor allows for any logical operation to be performed by determining the proper opcode. This custom opcode generation is described in the manual TMS320C80 (MVP) Parallel Processor User's Guide and also described in Appendix D. These opcodes are defined in the file "`defines.h`". Due to the function call overhead of the Genesis Native Library the thresholding is done such that the positive binary images for all bands is produced for the color that is being separated. This results in the logical operation for all the colors being an AND of all color bands. Since the Boolean function that the logical operation performs is the same, all the buffers are processed at the same time with a single call to the DSP board, saving 1.5 ms in overhead time, and allowing all primary colors to be segmented separately. The results from all color band thresholding for a specific band is placed into a child buffer. Then the logical operation is performed on the parent buffer, which performs the logical operation on all of the results at once. The child buffering is illustrated in Figure 14. The result of the logical operation is a group of binary identifier images that are passed to the blob analysis module. These images are arranged such that the blob analysis needs to process only a single buffer.

The processing times on the Matrox board are presented below in Table 5 where the times include the .5ms

Table 5. Image Segmentation Times

Operation	Subsample	Execution Time (ms)
Convert from Integer to Binary (1 operation)	1	2.79
Convert from Integer to Binary (1 operation)	2	0.836
Logical Operation (all colors)	1	3.334
Logical Operation (all colors)	2	1.52

overhead required for all processing function calls to the board. From the times, it is shown that the image must be subsampled in order for the processing to be completed in close to the required amount of time. This subsampling doubles the error in the blob position computation. After subsampling, each pixel corresponds to 8.5625 mm. This increased error represents an error of .313% of the field length and .561% of the field width. This error is smaller than the error that is introduced into the system during calibration.

Table 6. Image Segmentation Times

Operation	Number of calls	Execution Time (ms)	Total Execution Time (ms)
Differnce Image Preprocessing			
Generate Difference Image	1	2.680	2.680
Generate Binary Mask for All Bands	1	1.613	1.613
AND operation on Binary Masks	1	1.153	1.153
Merge Source Image and Mask	1	2.579	2.579
Total Time			**8.025**
Color Thresholding and Segmentation			
Conversion from 8-bit to 1-bit	9	0.836	7.524
Logical operation on 1-bit	1	1.520	1.520
Total Time			**9.044**
Total Time for Image Segmentation			**17.069**

The list of times for the image segmentation procedure is given in Table 6. The image segmentation is performed using the function call "`processFrame()`", and can be found in the file "`imageProcessing.c`". The function process frames using the Genesis Native Library calls:

- `ImIntDyadic()`: Generate the Difference Image
- `ImIntBinarize()`: Generate the Mask Image for All Bands
- `ImIntTriadic()`: AND operation on Binary Masks
- `ImIntTriadic()`: Merge Source Image and Mask
- `ImBinConvert()`: Conversion from 8-bit to 1-bit binary. Color Thresholds
- `ImBinTriadic()`: Logical Operation to Extract Sub-volume

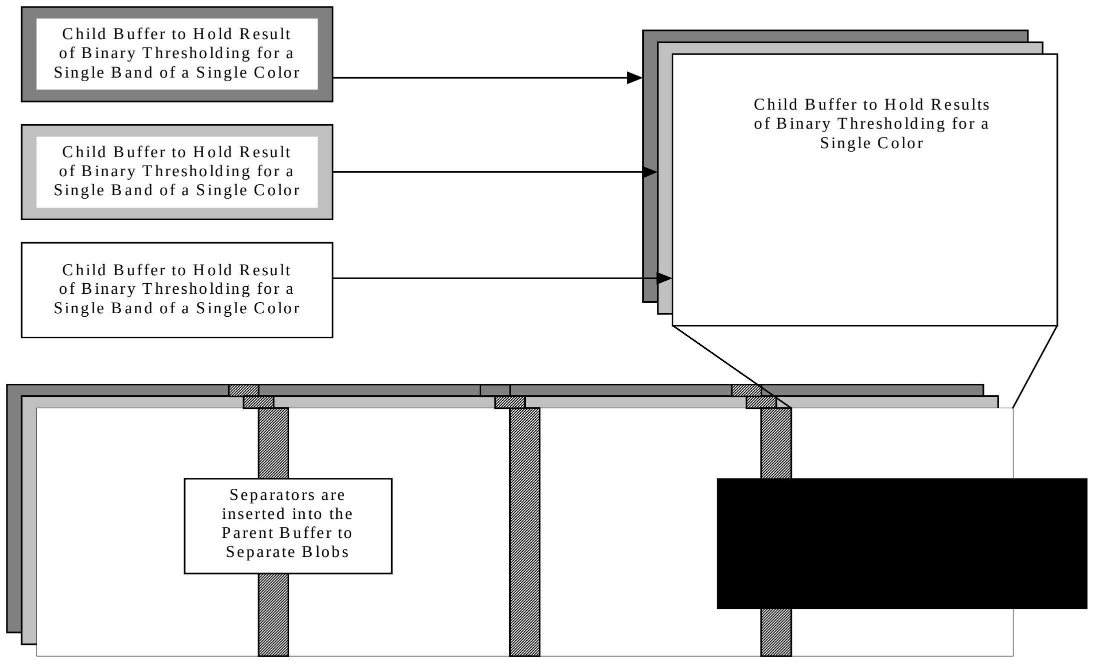

Child Buffer to Hold Result of Binary Thresholding for a Single Band of a Single Color
Child Buffer to Hold Result of Binary Thresholding for a Single Band of a Single Color
Child Buffer to Hold Result of Binary Thresholding for a Single Band of a Single Color
Child Buffer to Hold Results of Binary Thresholding for a Single Color
Separators are inserted into the Parent Buffer to Separate Blobs

Figure 14. Illustration of Buffer Management for Logical Operation

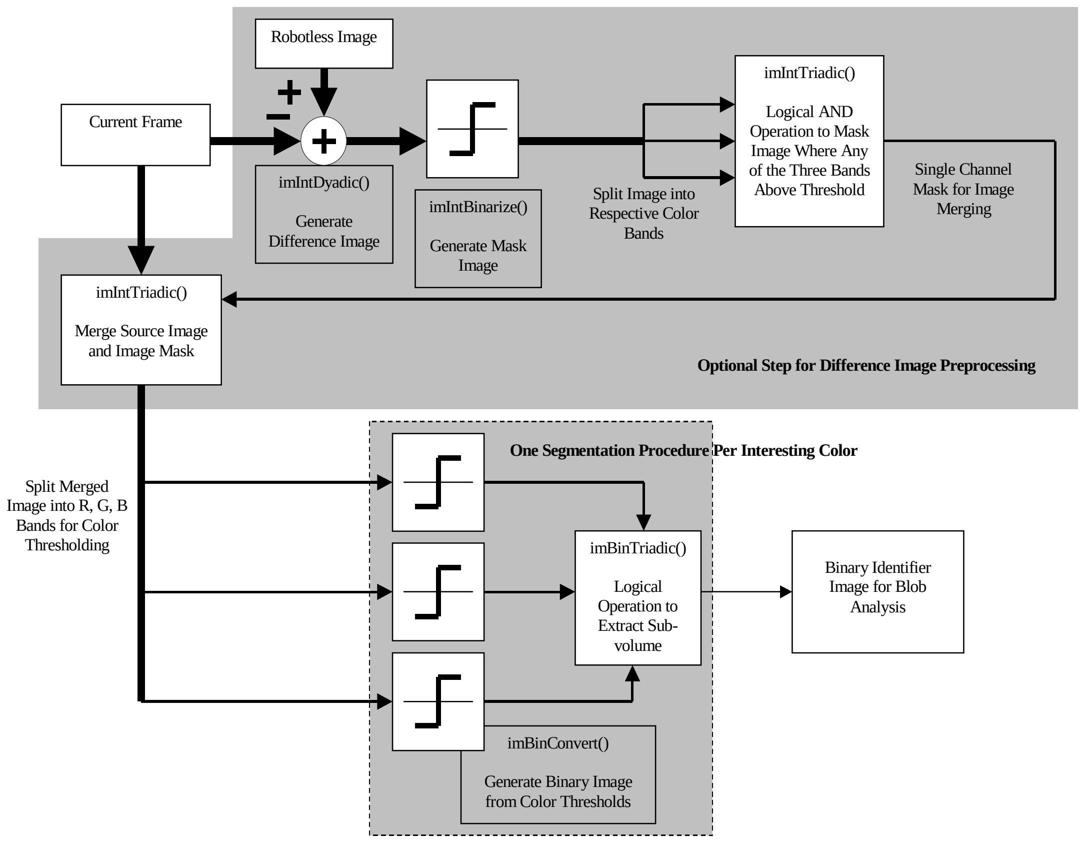

Robotless Image
+
−
Current Frame
imIntDyadic() Generate Difference Image
imIntBinarize() Generate Mask Image
Split Image into Respective Color Bands
imIntTriadic() Logical AND Operation to Mask Image Where Any of the Three Bands Above Threshold
Single Channel Mask for Image Merging
imIntTriadic() Merge Source Image and Image Mask
Optional Step for Difference Image Preprocessing
One Segmentation Procedure Per Interesting Color
Split Merged Image into R, G, B Bands for Color Thresholding
imBinTriadic() Logical Operation to Extract Sub-volume
Binary Identifier Image for Blob Analysis
imBinConvert() Generate Binary Image from Color Thresholds

Figure 15. Image Segmentation Block Diagram

10.4 Blob Analysis

The blob analysis locates all of the blobs that are the result of the sub-volume extraction in the color segmentation step. The blob analysis stage connects all of the pixels that are a binary '1' together if they are neighboring each other. Once all of the pixels are connected, the requested features are computed for each blob. These features are area, perimeter, and center of gravity. The blob analysis is performed on the single binary image that is the result of the color segmentation step. This allows for a single call to compute the blob features. The child buffers for each individual color are separated to ensure that blobs are not connected which correspond to different sub-volumes. Once the blob analysis is completed, the results are transferred to the host computer from the DSP board. The results are transferred in one step by transferring data into a structure that contains more features then those that are requested to be computed. These additional features in the structure are invalid and subsequently ignored in the tracking module. The structure is of type IM_BLOB_GROUP1_ST. It contains the fields that are described in Code Segment 1. The blob analysis computes only the binary features of the blob analysis image. Other than binary features, it can also compute grayscale features; however, the grayscale feature computation takes more processing time than the binary features, and a grayscale image of the field in also not available. Than would require an additional image processing step to perform the conversion to grayscale. The binary features also perform well enough for the locating the blobs. The blob analysis allows for sub-pixel results when computing the center of gravity. This is due to the averaging that is used for the computation.

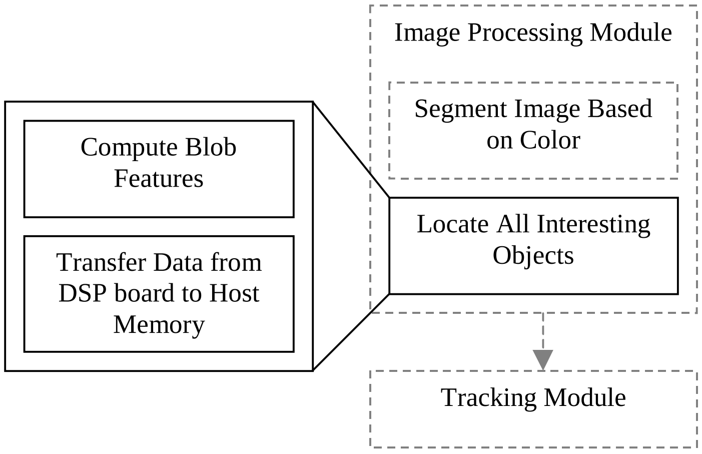


Image Processing Module
Segment Image Based on Color
Locate All Interesting Objects
Compute Blob Features
Transfer Data from DSP board to Host Memory
Tracking Module

Figure 16. Blob Analysis Breakdown

$$center\_of\_gravity\_x = \frac{\sum_{i}^{\infty} x_i}{blob\_area}$$

$$center\_of\_gravity\_y = \frac{\sum_{i}^{\infty} y_i}{blob\_area}$$

The blob area is simply the number of pixels that comprise a blob. The perimeter of the blob is the total number of edge pixels in a blob. The times for the blob analysis and transfer are provided in Table 7. The times for the blob analysis are a rough estimate; the times are data dependent and vary depending on the number and size of the blobs. The blob analysis is computed using the Genesis Native Library function "`imBlobCompute()`" and the results are retrieved using the "`imBlobGetResults()`"function.

```
/* Blob analysis results --- group1 */
typedef struct
{
   unsigned short number_of_blobs;
   unsigned short label_value;
   unsigned long area;
   unsigned short box_x_min;
   unsigned short box_y_min;
   unsigned short box_x_max;
   unsigned short box_y_max;
   unsigned short number_of_holes;
   unsigned short number_of_runs;
   float perimeter;
   float length;
   float breadth;
   float center_of_gravity_x;
   float center_of_gravity_y;
} IM_BLOB_GROUP1_ST;
```

Code Segment 1. Blob Analysis Result Structure

Table 7. Blob Analysis Computation and Transfer Times

Operation	Time to Complete using difference images(ms)	Time to Complete without using difference images (ms)
imBlobCompute	2.636	16.999
imBlobGetResult	2.102	6.695
Total Blob Analysis Time	**4.738**	**23.694**

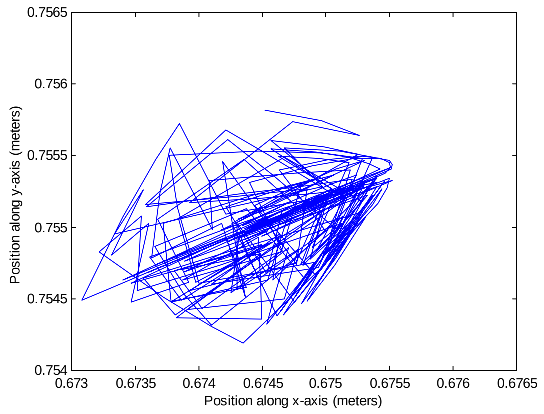


Figure 17. Position for a Stationary Object

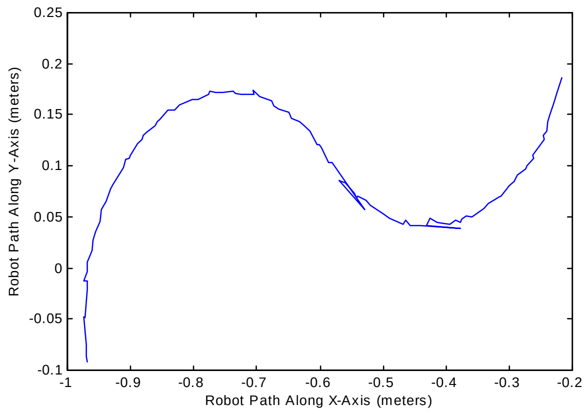


Figure 18. Position for a Moving Object

The blob analysis and transfer are performed in the function "`analyseBlobs()`" and is found in the file "`imageProcessing.c`"

The error in the blob analysis is typical small. The variance for blob positions is 7.4988x10^{-5} meters along the x-axis and 3.2395x10^{-5} meters along the y-axis. X-y plots for the positions of a robot is shown above. The model for the blob positions can be described as $x^{'}(t) = x(t)+v(t)$ where $v(t)$ is the measurement noise with mean of 0 and variance as stated above.

10.5 Initial Blob Filtering and Preprocessing

Once the blob results have been transferred to the board, the list of blobs is initially filtered and preprocessed. The filtering consists of size and shape filtering, and the preprocessing consists of separating the blobs that correspond to the different colors from each other. The buffer that contains the blob identifier image contains four child buffers, one for each color. The blob analysis processes the parent buffer to reduce the amount of overhead for the function call. To separate the blobs that correspond to the different colors, each blob that is of the specified size and shape is then examined for its position in the blob analysis identifier image. The size and shape of each blob is first examined to reduce the amount of time to compare the blob location values. If the blob falls within a certain child buffer, then the blob is added to the list of blobs for that color. Once a blob has been added to the list of blobs for a particular color, the offset from the child window location in the blob identifier image is removed. This results in four separate lists of blobs. The blob features are no longer needed, and only the center of gravity is used. Each blob is stored as a list of points in the window using a structure of type point. The point structure contains only the x and y positions of the blob relative to the upper left hand corner of the window and are in image coordinates. The initial filtering and blob preprocessing is found in the function "`preprocessBlobResults()`" and is found in the file "`track.c`" The filtering constants are defined in the file "`defines.h`". These include the maximum and minimum size for blobs and the maximum compactness. The blob areas after filtering are between 5 and 35. The lower bound allows for partial results from image segmentation, yet is large enough to remove most of the noise from the blob identifier image.

Code Segment 2. Point Structure

```
struct Point {
// all values represent pixels
        float x;
        float y;
};
```

The upper bound allows for slightly larger blobs that can result from pixelation of the image and dilation from setting the thresholds lower than necessary. The ideal blob area is 17.1399 pixels.

$$\pi\left(\frac{object\_raduis}{meters\_to\_pixel\_conversion}\right)^2 = \pi\left(\frac{.02m}{.0085625}\right)^2 = 17.1399\, pixels$$

The blob compactness is close to one for the objects that are being located. The compactness is defined as:

$$\frac{perimeter^2}{(4\pi \times area)}$$

A circle has the minimum compactness value of 1 and a square has a compactness value of 1.2732. To allow for slightly irregular shapes to result from the segmentation process, all objects with a compactness value of greater than 2 are discarded as noise. The filtering and preprocessing removes a significant amount of noise from the system.

After the blobs have been filtered for size and shape, the barrel distortion and parallax error is removed from the blob locations. The imaging distortion is removed after filtering to reduce computation for blobs that are not possibly objects being sought.

The barrel distortion is removed using a third order polynomial fit to the distortion introduced in the lens. The barrel distortion is a function of the lens itself and thus is independent of both the camera height and the location of the field in the image.

The parallax error is a function of the camera height and the height of the object under consideration. The parallax error is produced by the incident angle from the camera to the object on the field. The error is zero at the point directly below the camera and increases as the object moves radially away from the camera and as the height of the object increases. Figure 19 describes parallax error. Since different classes of objects have different heights, the position obtained from the blob analysis is the product of the projection of the object center onto the image plane. The difference in height of the object requires that all objects must be

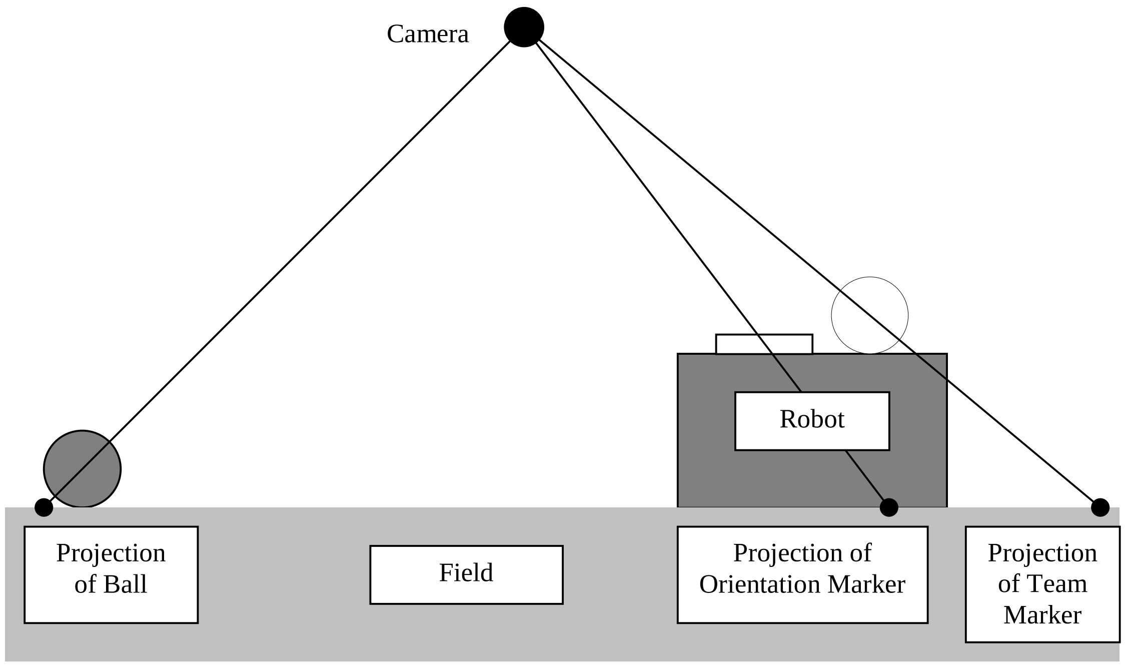


Figure 19. Parallax Error

projected onto the same plane and have the differences in height accounted for. In particular, the orientation markers and the team markers have different heights and thus is subject to error if the parallax error is not removed previous to the orientation determination. The parallax error is removed by undoing the projection of the objects. The projection is removed using the equation

$$d' = \frac{d}{h} \times h' = P$$

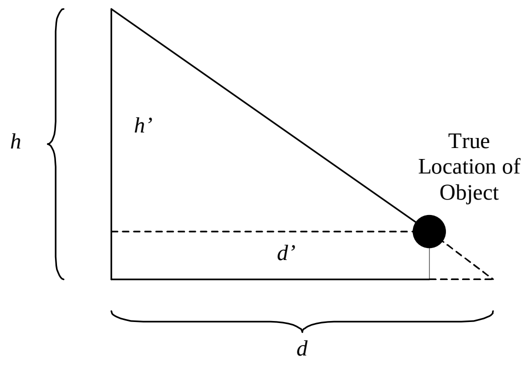


Figure 20. Removal of Parallax Error

d' is the true radial distance from the center of the lens to the object . *d* is the radial distance from the camera to the projection of the object. *h* is the distance from the camera lens to the field, and *h'* is the distance from the object to the camera. The transformation is a scalar and is identified below as *P*. The origin of the distances needs to be with respect to the location of the center of the camera, thus the distances need to be offset to place the origin under the camera. After removing the projection, the coordinates are converted back to image coordinates. Thus with the coordinate transform, the equation becomes

$$(d - offset)P + offset$$

The transformation can be performed separately and thus removed the need to convert from rectangular coordinates to polar coordinates. After rearranging the above equation, the transformation becomes

$$d' = d \cdot P + offset(1 - P)$$

Both the offset and the scalar *P* can be computed beforehand. The scalar P is computed for each class of objects; ball, team marker, and orientation marker. These scalars are computed during the system initialization and are stored in the calibration information. The storage of the scalars is in an array that is indexed using the constants BALL, ROBOT, and ORIENT, which are defined in "`defines.h`". The parallax error removal is found in "`paralaxCorrect()`" in "`transformation.c`".

The list of points from the filtering is then sent to the tracking module.

10.6 Object Tracking

The tracking module tracks all of the objects on the field. The position of the ball is determined and filtered

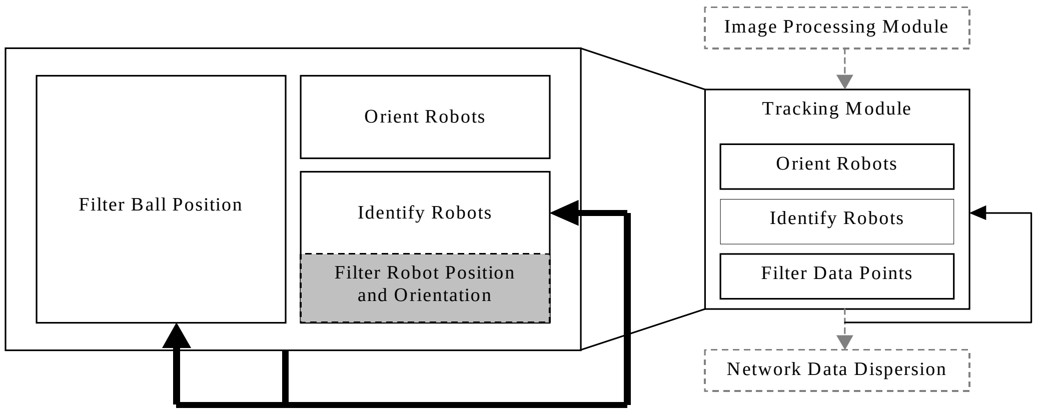


Figure 21. Tracking Module Breakdown

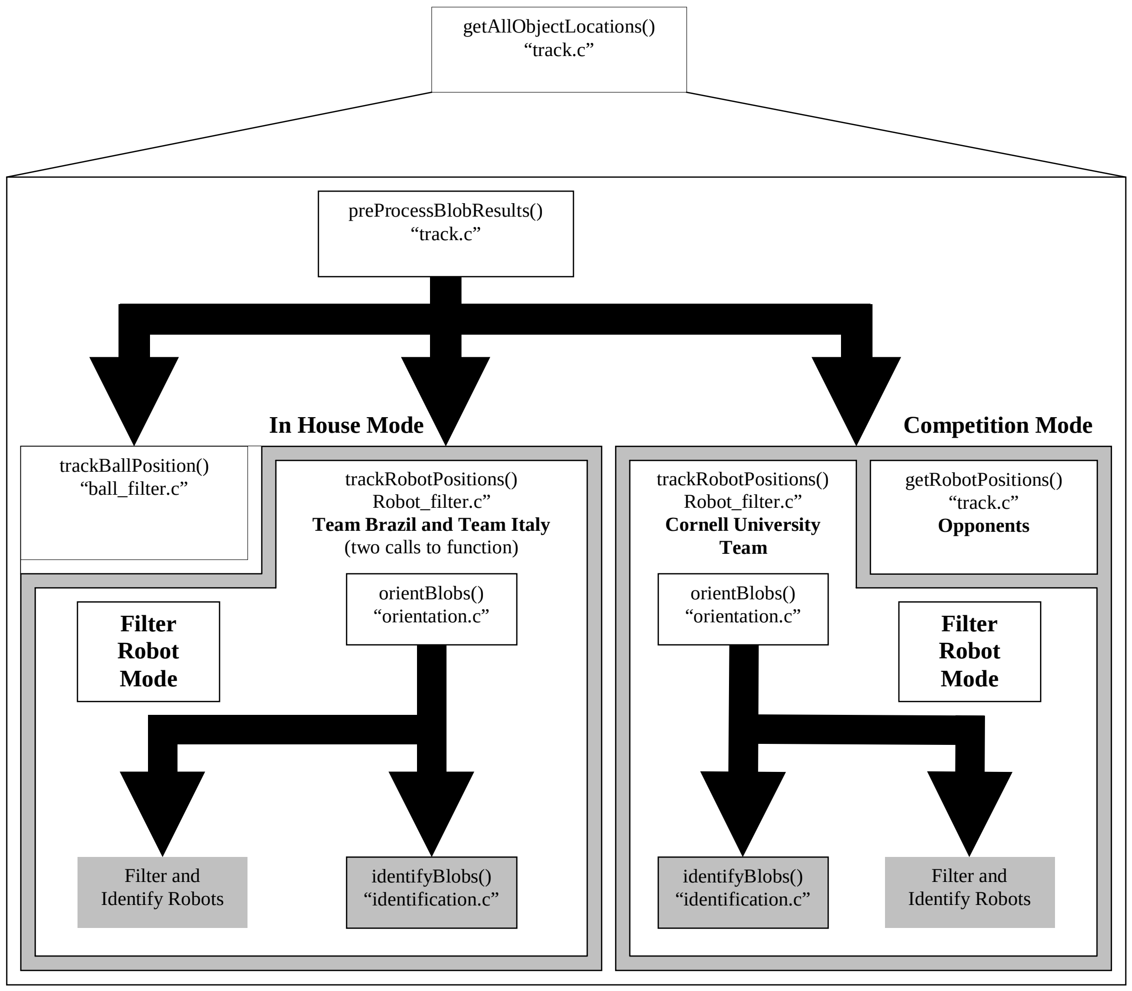


Figure 22. Modes for Object Tracking

using a linear tracking filter. The robots are oriented, identified, and filtered also using a linear tracking filter. The list of points that represent the ball is sent directly to the tracking filter to have the position of the ball estimated. The blobs that represent the robots and the orientation markers are first registered together, then the oriented blobs are sent to the tracking filter to be identified and have the robot positions and orientation estimated.

Object tracking is initiated with a call to "`getAllObjectPositions()`" in the file "`track.c`" Subsequent function calls are handled dependent upon the different modes of object tracking available.

10.7 Robot Orientation

The orientation procedure registers orientation blobs to team marker blobs by using an exhaustive search. For each team marker blob, every available orientation blob is examined for possible registration. The registration consists of examining the Euclidean distance from a team marker blob to all orientation blobs. If the distance is significantly larger than the physical distance from the orientation marker to the team marker on the top of the robot, then the next orientation marker is considered. If an orientation marker is found, then the angle between the team marker blob and the orientation marker blob is computed, the results are placed into a state structure, and the orientation blob is removed from the list. The state structure holds the position of the team marker blob and also the angle to the registered orientation marker. If an orientation marker is not found, then the orientation for the team marker is set to NOT_ORIENTED. The computation of the orientation of the robots uses the information already computed during the registration of the orientation and team marker blobs, allowing reduced computational complexity. The orientation registration has complexity of O(nm) where n is the number of team marker blobs found and m is the number of orientation marker blobs. The orientation of the robots contains significant amounts of error. The variance for the orientation of the robots is .0017 radians or 5.79.7402x10^{-2}°. These statistics are for the orientation for a stationary object. This error implies that the orientation of the robot orientation should be filtered even if the position of the robots is not. To reduce the amount of error in the orientation, the orientation markers are placed on the edge of the robot covers. This increases the distance between the two markers. The orientation determination can be modeled as

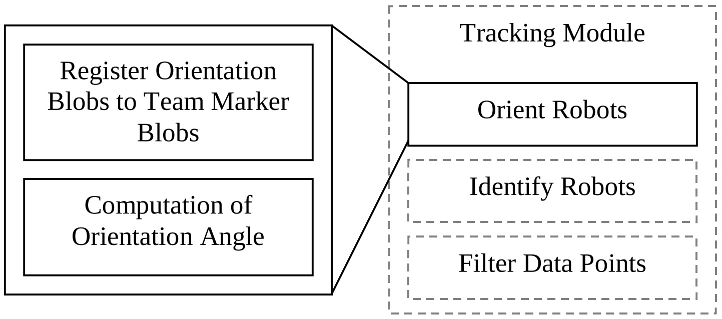

Register Orientation Blobs to Team Marker Blobs
Computation of Orientation Angle
Tracking Module
Orient Robots
Identify Robots
Filter Data Points

Figure 23. Robot Orientation Breakdown

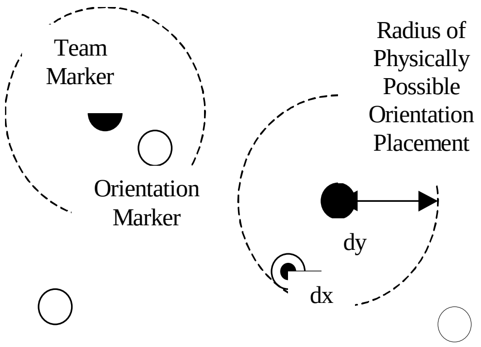


Figure 24. Registration of Orientation

$$\theta = \tan^{-1}\left(\frac{\left(y_m(t)+v_{ym}(t)\right)-\left(y_o(t)+v_{yo}(t)\right)}{\left(x_m(t)+v_{xm}(t)\right)-\left(x_o(t)-v_{xo}(t)\right)}\right)$$

The terms $v_{ym}(t)$, $v_{yo}(t)$, $v_{xm}(t)$, and $v_{xo}(t)$ are the noise that is introduced into the system due to measurement and pixelation. The noise terms are uncorrelated and have similar distribution. The mean is zero and the variance is as stated above in section 10.4. The equation can be separated into two parts. The part that represents the true orientation and a part that represents the noise in the orientation.

Code Segment 3. State Structure

```
struct state {
// position is in pixels
            float x;
            float y;
// orientation is in radians
            float w;
};
```

$$\theta = \tan^{-1}\left(\frac{\left(y_m(t)-y_o(t)\right)+v_y(t)}{\left(x_m(t)-x_o(t)\right)+v_x(t)}\right)$$

The noise terms then have zero mean and variance that is twice the variance given above in section 10.4. An approximation to the orientation error can be given by

$$\theta \approx \sin^{-1}\left(\frac{7.49\times10^{-5}}{.03}\right) = 2.5\times10^{-3}$$

Where 7.49x10^{-5} is the variance of the error in the position estimate and .03 is the distance between the two markers. The variance is then 2.5x10^{-3} radians or .1430°.

The orientation procedure can be found in "`orientBlobs()`" in the file "`orientation.c`"

10.8 Robot Identification with Linear Filtering

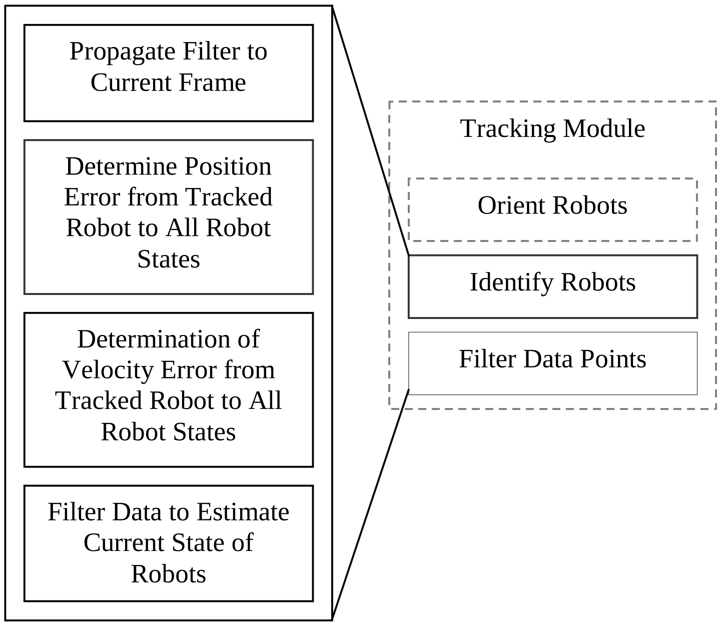

Propagate Filter to Current Frame
Determine Position Error from Tracked Robot to All Robot States
Determination of Velocity Error from Tracked Robot to All Robot States
Filter Data to Estimate Current State of Robots
Tracking Module
Orient Robots
Identify Robots
Filter Data Points

Figure 25. Identification and Filtering Breakdown

The robot identification can be done optionally with the linear filtering to aid in the identification. With the linear filtering, robot misidentification is less prominent. The filter allows for both the position and the velocity of the robots to be continuous. At the beginning of the identification and filtering process, the current filter state is updated to predict the robot location in the current frame. The equation for the update of the filter is

$$y[n] = y[n-1] + T * \dot{y}[n-1]$$

The units that are used for the filtering equations are pixels for the position of the robots and pixels per frame for the velocity. Since the velocity information is in pixels per frame, the time scale *T* for the update equations in one. This update of the filter state provides an estimate as to where the robots are expected to be on the field and how quickly they are moving during the current frame. The robot identification registers team marker blobs based on the closeness to this predicted state.

Each team marker blob that is located in the current frame is attempted to be registered with the robots that are located and tracked from all of the previous frames. The registration is performed using an exhaustive search of all the team markers that have been found. If the position difference between two successive frames differs by more than the physically realizable distance, then the team marker blob is disregarded. The registration is based upon the difference that is possible between the predicted robot location and velocity and the position and velocity of objects that are observed in the current frame. The team marker that has the minimal difference is accepted and registered as long as the position is physically realizable. The difference measure that is used is the sum of the change in position and velocity between two frames. The change is position from the current frame and the predicted position is computed and allows for the change in position between the two frames. Since the filtering is working in pixels per frame for velocity, these values are also the current velocity of the blob that is in question. The difference between this velocity and the velocity of the predicted state is then computed. These are summed to get the error term that is the metric for robot identification. The equations used for position error are:

$$dx = x_{current} - x_{predict}$$
$$dy = y_{current} - y_{predict}$$
$$E_{position} = \sqrt{dx^2 + dy^2}$$

The equations that are used for velocity error are:

$$dv_x = dx - dx_{predict}$$
$$dv_y = dy - dy_{predict}$$
$$E_{velocity} = \sqrt{dv_x{}^2 + dv_y{}^2}$$

The total error used for robot registration is

$$E_{total} = E_{position} + E_{velocity}$$

Once a robot is found and registered to the ordered list of actual robots, the new robot position is filtered to generate an estimated position and velocity that is sent to the artificial intelligence computers. The new state is based upon the observed position in the current frame and all previous observations with diminishing weight given to the previous frames.

The identification procedure gives precedent to the states that have orientation markers registered to them. The oriented robot states are a good indication that a robot is located at that location. Once all of the oriented robot states are examined for robot identification, the unoriented robot states are then examined. The information is then placed into a profile structure. The profile structure contains the position, orientation, and linear and rotational velocity of the object. This information contains the current state of the filter for that object. If any of the robots in the ordered list are still left to be currently located on the field, the remaining robot profiles are then extrapolated. After identification, the tracking state for a robot is marked as either TRACKING or TRACKING_NO_ORIENT if no orientation information is available but has been located on the field. If the robot state is extrapolated, the tracking state for the robot is set to the number of frames that the profile has been extrapolated. These states are defined in the file "`defines.h`".

The filtering allows for later extrapolation of the robot profile into the future. The filtering operates on both the robot position and the robot orientation. The robot orientation filtering in complicated by the fact that the robot orientation is angular. The assumption made is that the robot can only rotate about its axis at a rate of π/2 radians per cycle. Without this assumption the direction of rotation is

Code Segment 4. Profile Structure

```
struct profile {
// position is in pixels
            float x;
            float y;
// orientation is in radians
            float w;
// linear velocity is in pixels per cycle
            float dx;
            float dy;
// rotational velocity is in radians per cycle
            float dw;
};
```

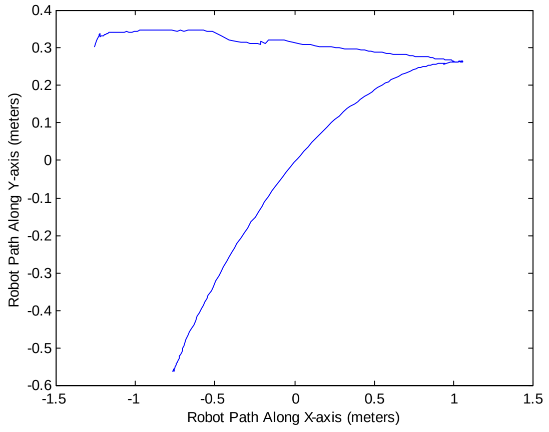


Figure 26. Robot Path with Filtering

impossible to determine from two discrete measurements. If the direction of the rotation is consistent with the difference between the measured angle and the predicted angle and within $\pi/2$ radians, then the orientation information is filtered. If at any point the filtered orientation is outside the range $-\pi$ to π, then the orientation is recentered to fall into the range.

The filtering is found to reduce the amount of error. The variance of the errors has been essentially reduced by a factor of two. The variance is 3.2429×10^{-5} along the x-axis and 5.3253×10^{-6} along the y-axis.

10.9 Network Implementation

Transmission of the data over the network requires that the data be formatted and placed into packets. The packet structure contains all of the relevant information about the position and velocity of the ball and all of the robots that are on the field. The structure is found to the {CODE}. All of the data is given in Cartesian coordinates. The data that is sent to the artificial intelligence computers is the data that is used for the tracking filters with the exception of the opponent robots, which are not filtered. The inclusion of the additional information in the opponent robot structure allows the same packet structure to be used for the vision display client. The orientation and rotational velocity is considered to be invalid for the opponent robots when the data is sent to the artificial intelligence.

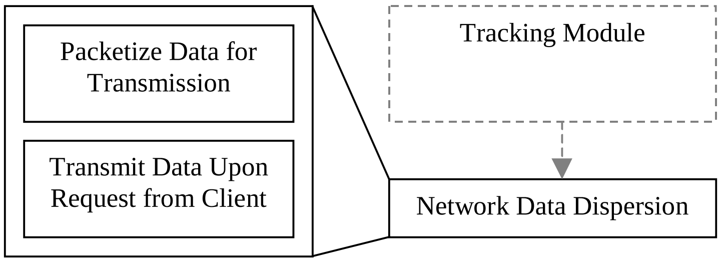

Packetize Data for Transmission
Transmit Data Upon Request from Client
Tracking Module
Network Data Dispersion

Figure 26. Network Dispersion Breakdown

The data packetization is determined by the mode that the vision system is currently operating in. If the vision system is in IN_HOUSE mode, then the robot position, velocity, orientation, and rotational velocity is computed for all of the robots on the field. All of the data for the Brazil robots and only the position of the Italy robots is sent to the Brazil artificial intelligence computers. All of the data for the Italy robots and only the position of the Brazil robots is sent to the Italy robots. Thus the artificial intelligence computers are not given an additional advantage when they are competing against each other. The data that is sent to the computers in IN_HOUSE mode is similar to what is to be expected during competition. These two sets of packets are then sent to the artificial intelligence computers upon the request for new data. In competition mode, only one set of data is packetized and sent to the artificial intelligence computers upon a request for more data. If the vision display client is being run, then the all of the data that has been collected for the ball, and the robots is packetized and sent to the vision display client upon a request for the data. This data contains the position, velocity, orientation, and rotational velocity of all the robots on the field, if the system is in IN_HOUSE mode and all information for the ball and Cornell robots, and the position only for the opponent robots if the system is in competition mode.

The network connection is established by opening a server port on the vision computer and servicing a request for data. If the vision system is in IN_HOUSE mode, the Brazil artificial intelligence computer connects on the port BRAZIL_PORT and the Italy artificial intelligence computer connects on the port ITALY_PORT. The artificial intelligence computer when the system in running in competition mode connects on the port VISION_PORT. If a vision desplay client is running, the client connects on the port DISPLAY_PORT. The port numbers are defined in the file “defines.h”.

The packet structure contains the number of opponent robots that are found on the field. This number (oNumber) contains the amount of the opponent robot information that is valid in the array. The number of the team robots (fNumber) contains the validity of each of the robot information that is in the array. Since the friendly robots are ordered according to the physical robot numbers that are assigned to them, the number of the robot on the field is considered to be valid is the corresponding bit in the 8-bit number is a ‘1’. If the bit is a ‘0’ then the robot information is considered to be invalid. This can result if the robot is not on the field, or it the robot track has been dropped.

To service a network request for data, the vision system thread waits for a request to come in from another computer. This request is a single byte that is identified as SUBMIT_REQUEST. Once this byte has been received, the vision system sends the appropriate data to the address that the request came from.

Code Segment 5. Network Data Structure

```
typedef float pos_t;
typedef float vel_t;
typedef float ang_t;
typedef float rot_t;

struct ballTrack_net{
        pos_t x;
        pos_t y;
        vel_t dx;
        vel_t dy;
};      /* 128 bits */
        /* 16 bytes */

struct oRobotTrack_net{
        pos_t x;
        pos_t y;
        vel_t dx;
        vel_t dy;
        ang_t w;
        rot_t dw;
};      /* 192 bits */
        /* 24 bytes */

struct fRobotTrack_net{
        pos_t x;
        pos_t y;
        vel_t dx;
        vel_t dy;
        ang_t w;
        rot_t dw;
};      /* 192 bits */
        /* 24 bytes */

typedef struct localNetworkStruct{
        struct ballTrack_net ball;
        struct fRobotTrack_net fRobot[5];
        char fNumber;
        struct oRobotTrack_net oRobot[5];
        char oNumber;
} netdata_t;    /* 2064 bits */
                /* 258 bytes */
```

10.10 Vision System Threads

The vision system is multi-threaded. One thread contains all of the processing and the tracking proceedures. The other threads are for servicing network requests and for the user interface. Only the required number of threads are built to reduce the amount of distributed processing. The network service threads are assigned such that there is one thread servicing each of the ports that are open on the machine. Thus during IN_HOUSE mode, there are two network service threads, and in competition mode there is only one network thread. The network data structures that are sent to the artificial intelligence computers are global variables that are accessible to all of the threads. To prevent sending data over the network, and writing data to the data structures, a mutex is used for mutual exclusion. There is one mutex for each network data structure. Thus the mutexes are locked for the shortest amount of time, reducing the waiting time to read the data and copy it onto the network.

References

[1] Birchfield, S. “Elliptical Head Tracking Using Intensity Gradients and Color Histograms”. IEEE Conference on Computer Vision and Pattern Recognition, Santa Barbara, California, June 1998.

[2] Brookner, E. Tracking and Kalman Filtering Made Easy, John Wiley & Sons, Inc. New York, 1998

[3] Folta, F., Van Eycken, L., and Van Gool, L. “Shape Extraction Using Temporal Continuity”, WIAMIS, 1997, pp. 69-74, Louvain-la-Neuve Belgium (1997)

[4] Han, K., and Veloso, M. “Reactive Visual Control of Multiple Non-Holonomic Robotic Agents”

[5] Horswill, I., and Barnhart, C. “Unifying Segmentation, tracking, and Visual Search”.

[6] Kitano, H. et. al. “RoboCup: A Chanllenge Problem for AI and Robotics. “, RoboCup-97: Robot Soccer World Cup I. Springer-Verlag, Germany, 1998. Pp. 1-19

[7] Poynton, C. “A Guided Tour of Color Space”, New Foundations for Video Technology (Proceedings of the SMPTE Advanced Television and Electronic Imaging Conference). San Francisco, Feb. 1995, 167-180.

[8] Shafarenko, L., Petrou, M., and Kittler, J. “Histogram-Based Segmentation in a Perceptually Uniform Color Space” IEEE Transactions on Image Processing. Vol. 7, No. 9, September 1998. Pp. 1354-1358

[9] Shi, J., and Malik, J. “Motion Segmentation and Tracking Using Normalized Cuts”, International Conference on Computer Vision, January 1998.

[10] Swain, M. and Ballard D. “Color Indexing”, International Journal of Computer Vision, 7:1, 1991. Pp. 11-32

[11] Veloso, M., Stone, P. and Han, K. “The CMUnited Robotic Soccer Team: Perception and Multiagent Control”.

[12] Veloso, M., Stone, P., Hun, K., and Achim, S. “Prediction, Behaviors, and Collaboration in a Team of Robotic Soccer Agents”, Submitted to ICMAS ’98 in November 1997.

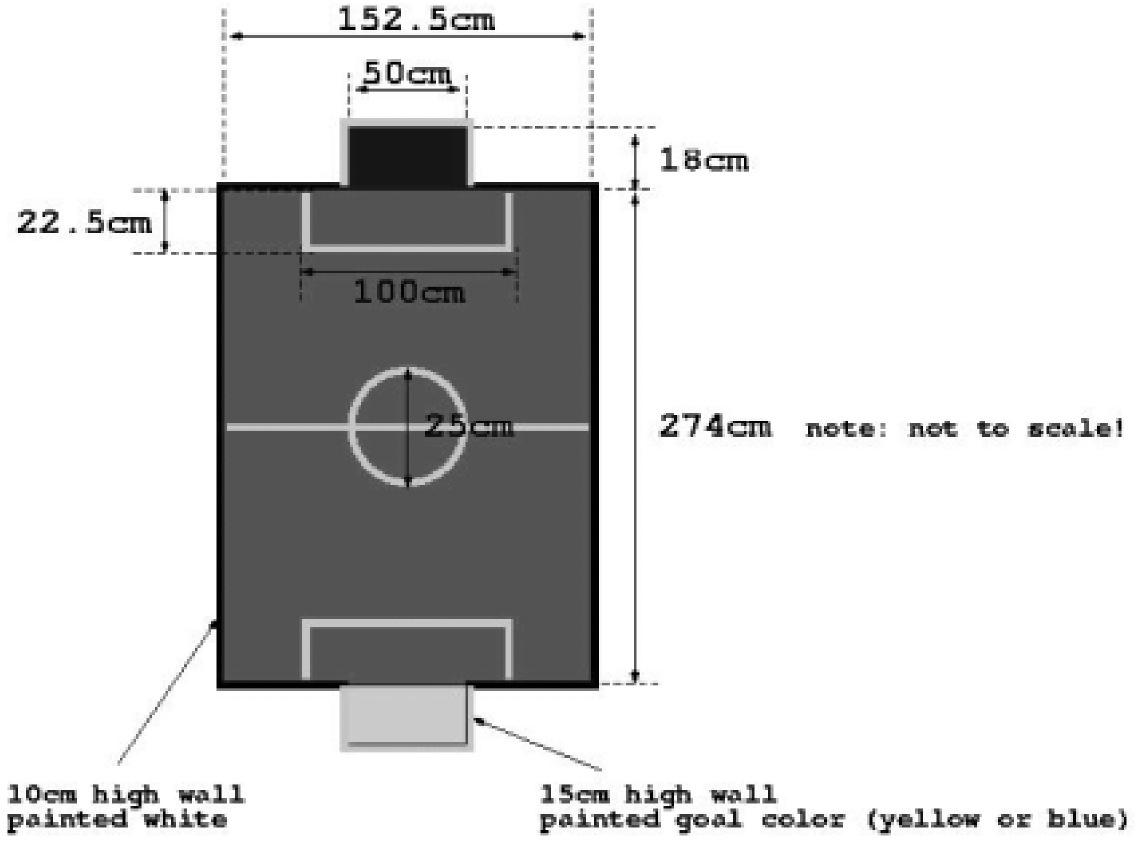
RoboCup Small Size League (F-180) Rules and Regulations

These are the official rules for 1999 RoboCup F-180 League play (revised 20 January 1999).

A. 1 Playing Field

Surface

A table tennis table is the official surface for matches. The size and color of the table is defined as the International Table Tennis Federation (ITTF) standard. ITTF regulations concerning the height of the table surface above the floor do not apply. We will provide an appropriate height for the RoboCup competition, which will be chosen to aid the global vision systems. Dimensions are 152.5cm by 274cm; the color is matte green. Every effort shall be made to ensure that the table is flat, however, it is up to individual teams to design their robots to cope with slight curvatures of the surface. (please refer to ITTF Regulations for more details on the table).



Here is a picture of an older version of the field. Note that the goals are different now.

Walls

Walls shall be placed all around the field, including behind the goals. The walls shall be painted white and shall be 10cm high. Behind the goals walls will be 15cm high and painted one of the two appropriate goal colors.

Four small panels are positioned in the corners to avoid balls getting stuck. As shown in the figure below, they are located 3cm from the corner for each axis. Green stripes 1cm wide are painted on the corners to aid robots in visual identification of the edge of the panel.

Goals

The width of each goal is 50 cm, (approximately 1/3 of the width of the shorter end of the field). The goal is 18 centimeters deep. The wall continues behind the goal but increases to a height of 15cm. There is no safety net over the goal, nor is there a horizontal goal bar. Robots can not fall off the playing area because of the wall. The wall and area behind the goal line will be painted either yellow or blue. It should be noted that a robot may use the area behind the goal.

Defense Zone

A defense zone is created around each of the goals. It extends from the front of the goal to 22.5cm into the field. The zone is 100 cm wide. Only one robot from each team may enter this area. Brief passing and accidental entry of other robots is permitted, but intentional entry and stay is prohibited.

Once a defending robot (goal keeper) has hold of the ball or is facing and in contact with the ball then all attacking robots must leave the area. The attacking robot can not interfere with the goal keeper. Given the size of the defense zone a robot is said to be in the defense zone if any part of it is within the area.

Also, an attack robot can not intentionally interfere with the movement of the defenders robot in the defense zone. A robot can not be used to block the movement of the goal keeper.

Table markings/colors

- The field shall be dark green. ITTF's color regulation is flexible, there may be slight differences in the color of the table. Robot designers should take this fact into consideration.
- Walls are white.
- A 1 centimeter thick white line will be painted across the table (the center line), with a center circle 25 centimeters in diameter placed at the center.
- The border of the defense zone will be painted in white, with a width of 1cm.
- The area behind the goal is either dark blue or yellow (one end is dark blue and the other yellow).

A. 2 Robots

Area

The total floor area occupied by a robot shall not exceed 180 square centimeters.

18cm rule

The robot must fit inside an 18cm diameter cylinder. For rectangular robots the diagonal length would be the constraint. Also note that some shapes, even though no dimension exceeds 18cm will not fit within an 18cm cylinder (e.g. an 18cm x 18cm x 15cm triangle). 15cm by 9.9cm rectangular robots will fit. 15cm by 12cm rectangular robots are too big even though their area is 180cm^2.

ONE TIME 1999 EXCEPTION: In RoboCup-99, robots that competed in RoboCup-98 will be allowed to compete, even if they fail the 18cm rule, provided they cannot be easily modified to comply. This exception will be eliminated in 2000.

Height

If the team is using a global vision system robot height is restricted to 15 cm or less. Otherwise the robot height must be 22.5 cm or less. Height restrictions do not apply to radio antennae and visual markers.

Marking/Colors

Coloring is a very sensitive issue, because it is very difficult to guarantee exactly which colors will be used at the competition. Colors also change depending on lighting. Even though the organizers will make a sincere effort to provide standard colors, the designers should design their robots to cope with variations in color. Each team will be given certain amount of time to fine-tune their robot for the actual fields and settings on a day before the competition.

Markings for robots need to enable visibility from above (for global vision) and from the playing field (for mobile vision). To support this, each robot will be marked using a single colored ping-pong ball (provided by the RoboCup organization) mounted at the center of their top surface. Unless the shape or drive mechanism of the robot does not allow it, the marker should be located at the center of rotation of the robot. If this is not possible the relationship between the marker placement and the axis of movement/rotation must be advertised before the competition.

All of the robots defending the yellow goal will display a yellow ping pong ball. All of the robots defending the blue goal will display a blue ping pong ball. Each team must be able to use either color as the primary color.

For mounting purposes, ping pong balls will be drilled with two small holes (2 mm in diameter) through their axis to provide for mounting on a spindle. Each robot will be fitted with a spindle for holding the balls. Note: the robot's antenna may be used for this purpose.

Other than the official markers, no external colors of the robot, including the body may be goal-blue, goal-yellow, field-green, or ball-orange. While it is not specifically required, black is the recommended color for the body of the robot.

Teams may use any additional markers they wish, provided:

- The color of the marker is not goal-blue, goal-yellow, field-green or ball-orange. Participants should strive to find colors as different as possible from these "official" colors.
- The marker does not cause the robot to exceed any of the horizontal dimension (area) restrictions described above.
- They provide multiple copies of their markers at the competition so other teams may use them to calibrate their vision systems.

Inspection

The robots will be examined by the referee before the game to ensure that they meet these constraints. As one test for area compliance, the robots must fit into an 18cm diameter cylinder. Whilst being inspected each robot must be at its maximum size; anything that protrudes from the robot must be extended. Except as allowed under "conflict resolution" below, ANY VIOLATION OF DIMENSION OR COLORING CRITERIA MAY DISQUALIFY THE ROBOT FROM COMPETITION.

Team

A team shall consist of no more than 5 robots.

A. 3 Ball

An orange golf ball provided by the RoboCup organization shall be used. An example of the official ball will be sent to requesting teams.

A. 4 Pre-game setup

Organizers will make every effort to provide the teams access to the competition area at least two hours before the start of the competition. They will also strive to allow at least one hour of setup time before each game. Participants should be aware, however, that conditions may arise where this much time cannot be provided

A. 5 Length of the game

The games consist of the first half, break, and the second half; each is 10 minutes. Each team will be allowed some set up time at the start of the game. Before the beginning of the second half, teams will switch sides (including their blue or yellow team markers). However, if both teams agree that switching sides is not necessary, they will not be required to switch.

A. 6 Timeouts/delay of game

Each team will be allocated four five-minute timeouts at the beginning of the game. In case a team is not ready to start at the scheduled time, they may use their timeouts to delay the game up to 20 minutes.

During a game, timeouts will only be granted during a break in play.

A. 7 Substition and removal of damaged robots

In general, substitutions are only allowed for damaged robots during a break in play. However, if in the opinion of the referee, a damaged robot is likely to cause serious harm to humans, other robots or itself the referee will stop the game immediately and have the damaged robot removed. In this case, the game will be restarted with a free kick for the opposing team (the team that did not have the damaged robot). If there is no immediate danger however, the referee may allow the game to continue.

To replace a robot by substitute at other times the following conditions must be observed:

- a substitution can only be made during a stoppage in play.
- the referee is informed before the proposed substitution is made,
- the substitute is placed on the field after the robot being replaced has been removed,
- the substitute is placed on the field in the position on the field from which the replaced robot was removed.
- a substitution can only be made during a stoppage in play.

A. 8 Wireless Communication

Robots can use wireless communication to computers or networks located off the field. Participants shall notify the organizers of the method of wireless communication, power, and frequency by the 1st of May. The tournament committee shall be notified of any change after that registration as soon as possible.

In order to avoid interference, a team should be able to select from two carrier frequencies before the match. The type wireless communication shall follow legal regulations of the country where the competition is held.

A. 9 Global Vision System / External Distributed Vision System

The use of a global vision system or an external distributed vision systems are permitted, but not required, to identify and track the position of robots and balls. This is achieved by using one or more cameras.

Cameras positioned above the field will be mounted on a beam suspended from the ceiling. The beam will be positioned 3 meters above the table. If both teams agree, and the hosting facilities allow it, another height may be used. Cameras may not protrude more than 15cm below the bottom of the beam. The placement of cameras is performed on a game by game basis, and the teams choose camera positions by tossing a coin to find which team places a camera first. The use of a global vision system shall be advertised at the time of registration, and detailed arrangements shall be discussed with the RoboCup organizing committee.

The local organizer will inform all participants of the camera attachments required to use the beam provided.

A. 10 Lighting

A description of the lighting will be provided by the local organizer. The intent is to provide 700-1000 LUX uniform light.

A. 11 Goal Keepers

Each team may designate one robot as a goal keeper. The goal keeper can hold and manipulate a ball for up to 15 seconds within its penalty area. After releasing the ball the keeper must not recapture the ball until it touches an opponent or a member of its own team outside the penalty area. If the ball is released by the keeper and it reaches the half way line without touching any other robot, the opponent is given an indirect free kick positioned anywhere along the half way line (borrowed from Futsal rule).

Any of the robots may change roles with the goal keeper (and thus be permitted to manipulate the ball) provided the referee is informed before the change and that the change is made during a stoppage in play.

A. 12 During play

- All time for stoppages will be added to the end of the half they occur in.
- The ball has to go forwards at a kick-off or the kick-off will be restarted.
- In general, movement of robots by humans is not acceptable. However, at kick-offs and restarts one member of the team is allowed on the pitch to place robots. Gross movement of robots is not allowed, except before kickoffs, to place the designated kicker for a free kick or to ensure robots are in locations required for penalty and free kicks. Humans are not allowed to free stuck robots except during a stoppage in play, and then they should move the robots only far enough to free them.
- The ball may be lifted during play. However, the height of the ball from the table must not endanger spectators, the referees or human team members!! If the ball crosses the goal line 15cm above the table, the goal is disallowed and a free kick is awarded to the defending team.

A. 13 Kick-off/Restart/Stop

Before a kickoff, or once play has been stopped for other reasons (e.g. a foul or penalty kick) robots should cease movement until play is restarted by the umpire.

For the start or restart of the game the umpire will call verbally, or by whistle, and the operator of the team can send signals to robots. The signal can be entered through a keyboard attached to a server being used on the side lines. No other information, such as strategy information may be sent. Also, the keyboard operator my not send information during play. This paragraph only applies during play, strategy revision during half time and timeouts is permitted.

A. 14 Robot positions at kick-off/Restart

Kick-off

All robots shall be in located on their side of the field.

Penalty Kick

Only a goal keeper shall be in the defense zone, and the ball shall be located at the specified position (45cm from the goal along the lengthwise centerline of the field). All other robots shall be located at least

15 cm behind the robot which kicks the ball. Robots other cannot move until the referee signals the resumption of play (by whistle, etc.).

Free Kick

Free kicks are taken after a foul or a stoppage in play. If the free kick is taken after a foul the ball is placed at the point where the foul was committed. If the free kick is taken after a stoppage in play, the ball remains in place. If the ball is within 15cm of a wall or the defense zone line, the ball will placed 15cm from the wall or defense zone.

All robots must be placed 15cm from the ball. If the kick was awarded as the result of a foul, a human from the team awarded the kick may place one robot near the ball. None of the robots may move until play is resumed by the referee.

Restart after a goal

The non-scoring team will be awarded the kick-off. The restart after the goal shall adopt the same formation as the kick-off.

Robots may be moved to their starting positions by hand.

Throw-in

When the ball departs the field, the ball will be returned immediately to the field, and located approximately 5cm inside of the wall, in front of the closest robot of the team which did not push the ball out of play to where the ball went out of bounds.

During this period, the robots can continue to move, and the time counting continues.

A. 15 Fouls

The following fouls are defined:

Lack of progress

If it is deemed by the referee that the game has stopped then a free kick is awarded to the team which last touched the ball. A game is considered stopped if the ball has not been touched by a robot for 20 seconds and it appears that no robots are likely to hit the ball.

Non-moving robots

If the referee determines that a robot is not moving for a period of 20 seconds or longer, he will remove it from play and give the robot a yellow card. Participants may repair the robot and ask that it be put back in play if they desire. A second failure of the same robot to move for 20 seconds will result in a red card and permanent removal from the same. Goal tenders and robots further than 20cm from the ball will not be penalized.

Multiple Defense

When more than one robot of the defending side enters the defense zone and substantially affects the game a foul will be called, and a penalty kick will be declared.

Ball Holding

A player cannot 'hold' a ball unless it is a goal keeper in its penalty area. Holding a ball means taking a full control of the ball by removing its entire degrees of freedom; typically, fixing a ball to the body or surrounding a ball using the body to prevent accesses by others. In general 80% of the ball should be outside the a convex hull around the robot. This is up to the referee to judge whether a robot is holding the ball. In general another robot should be able to remove the ball from another player. It a robot is deemed to be holding the ball then a free kick will be declared. If this happens in the defense zone by the defense team, a penalty kick will be declared.

Court Modification

Modification or damage to the court and the ball is forbidden. Should this occur, the game is suspended and the appropriate restoration is done immediately before the game resumes.

Robot Halting

All the players must be halted prior to kick-off or restarting of the game. The judges check or adjust the placements of the players and declares the completion of adjustment 5 seconds before indicating a kick-off or a restart action. During this 5 seconds, the players can not move.

Charging/Attacking

Unless striving for a ball a player must not attack another. In case the umpire clearly observes such an act, it is regarded as a violent action. Then the umpire presents a red card to the responsible player ordering it to leave the game. The judgment is one based on an external appearance. In general, it is unacceptable for multiple robots to charge a single robot, and it is also unacceptable to hit the back of a robot even if it has the ball and it is unacceptable to push along the table another player. The exact interpretation of what is acceptable is left to the referee.

During play, if a player utilizes a device or an action which continuously exerts, or whose primary purpose appears to be, serious damages to other robot's functions, the umpire can present a yellow card as a warning to the responsible player, and order it to go outside the court and correct the problem. Once the correction is made, the robot can resume to the game under an approval by the umpire. In case the problem is repeated, the umpire presents a red card to the responsible player telling it to leave the game. This rule could be invoked on a robot should it continuously charge a robot whilst attempting to tackle the other robot.

Offside

The offside rule is not adopted.

Fair play

Aside from the above items, no regulations are placed against possible body contacts, charging, dangerous plays, obstructions etc. However, it is expected that the aim of all teams is to play a fair and clean game of football.

A. 16 Conflict Resolution

Resolution of dispute and interpretation of ambiguity of rules shall be made by three officials, who will act as umpires, designated prior to the match. The umpires shall not have any conflict of interest to teams in the match. The umpires may consult with the tournament officials of the RoboCup for resolving conflicts. Ambiguities shall be resolved by referring to FIFA official regulations, where appropriate. Specific modifications to the rules to allow for special problems and/or capabilities of a team's robots may be agreed to at the time of of the competition, provided a majority of the contestants agree.

Appendix B Hardware Specifications

B.1 Sony DXC-9000 3CCD Color Video Camera

Optical System:	Pickup device ½-inch CCD, interline transfer type
Effective Picture Elements:	659x494
Lens Mount:	½-inch bayonet type
Signal Format:	NTSC standard format
	Scanning
	525 lines, 2:1 interlace
	VGA format
	Scanning
	640x480, 1/60 non-interlace
Horizontal Resolution:	Horizontal: 700 TV lines
	Vertical: 480 TV lines
Sensitivity:	2,000 lux (F5.6, 3200K)
Signal-to-Noise Ratio:	58 dB
Gain Control:	AGC and 0 to 18 dB in units of 1 dB
White Balance:	Automatic
	Manual: Red gain and green gain adjustable individually
Electronic Shutter Speed:	Step mode and variable mode
Video Output Signals:	Composite: 1.0Vp-p, 75 ohms
	RGB: 1.0Vp-p, 75 ohms
Input/output Connectors:	VIDEO OUT: BNC type, 75 ohms
	DC IN/VBS: 12-pin
	REMOTE: mini-DIN 8-pin
	RGB /SYNC: D-sub 9-pin
	LENS: 6-pin connector for 2/3-inch lens
Power Supply:	12V DC
Dimensions (w/h/d):	79x72x145 mm
Mass:	790 g
Mounting Screw:	U1/4”, 20 UNC
	4.5 ± 0.2 mm (ISO standard)
	0.197 inches (ASA standard)

B.2 Sony VCL-714BXEA Zoom Lens

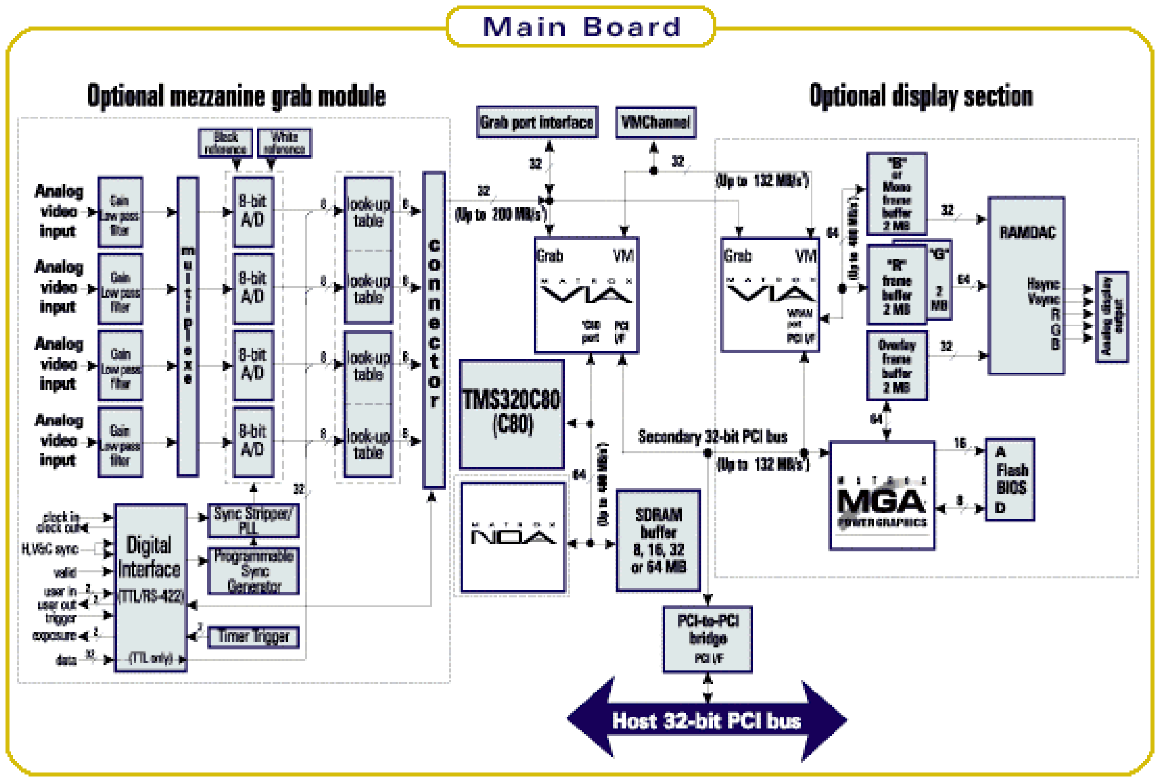
Focal Length:	7.5 mm to 105 mm
Iris:	1.4 to 16, and C
Field of View (at 1.1 m):	W: 660 mm to 880 mm
	T: 47 mm to 63 mm
Focus Range:	∞ to 1.1 m
Mount:	Bayonet mount
Dimensions:	195x82.5x124 mm

Weight:	1,120 g

B.3 Matrox Genesis

Processor:	1 Texas Instruments TM320C80 (‘C80) 32-bit RISC master processor (MP) with floating point unit. The ‘C80 contains four parallel processors (PP) which are 32-bit fixed point DSPs with 64-bit instruction words
Texas Instruments ‘C80 Clock Rate:	50 MHz
Processor Memory Management:	peak 400MB/s @ 50MHz for data transfer between on-board and off-board memory
SRAM:	64MB
Grab Module:	included
Analog Interface:	4 software selectable video inputs
	4 8-bit analog-to-digital converters
Sync Generator:	Sync and timing FPGA provides control for synchronization, triggering, exposure, and inputs and outputs
Video Adjustment:	Software programmable input gain, offset, and references
	Phase adjustment: 0°-270°, 90° increments
	Fully configurable and configurable input LUTs
	Four 256 x 8-bit
	Four 8K x 16-bit
Display Module:	included
Frame Buffers:	Dual-screen mode available
	220 MHz RAMDAC
NOA (Neighborhood Operations ASIC):	included
Software:	Matrox Genesis Native Library
	Matrox Imaging Library Lite (MIL-Lite)
	Matrox Active MIL

- Matrox Intellicam



-

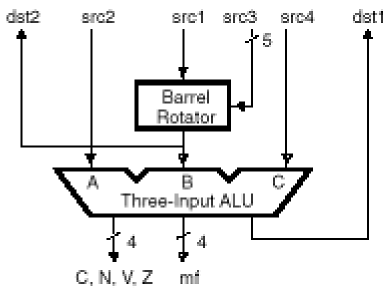
Appendix C Sample Calibration File

```
This is the number of paramaters
25
This is the color thresholds for the image segmentation
Thy Ball
#Red_threshold_ball
174
#Green_threshold_ball
157
#Blue_threshold_ball
196
Thy Robots
#Red_threshold_y_robot
176
#Green_threshold_y_robot
116
#Blue_threshold_y_robot
184
Them Robots
#Red_threshold_b_robot
254
#Green_threshold_b_robot
254
#Blue_threshold_b_robot
165
This is the size and position of the child buffer
#X_start
4
#X_stop
316
#Y_start
42
#Y_stop
212
These are the field dimensions in pixels of the image
#Field_x_start
4
#Field_y_start
42
#Field_x_stop
318
#Field_y_stop
212
This is the number of friendly robots
#Number_of_fbots
1
#Number_of_obots
1
This is the physical paramaters of the field
#X_scale
.0101488999185
#Y_scale
-.0103457867208
#X_offset
```

```
157.32
#Y_offset
85.7185
This is for parallax error correction (in meters)
#Camera_height
2.59
#Robot_height
0.15
```

Appendix D PP Guide for Determining ALU Opcodes for Triadic Functions

The ‘C80 allows for the explicit assignment of opcodes for logical and arithmetic functions. The procedure for determining the opcode, or the 32-bit value that resides in register d0 of the PP for the instruction class EALU||ROTATE, can be found discussed in great detail in the TMS320C80 (MVP) Parallel Processor User’s Guide and briefly on page 65-66 in the Matrox Genesis Native Library User’s Guide. The opcode derivation supports all 256 Boolean and arithmetic functions that are possible with 3 inputs. The EALU||ROTATE allows for four input operations with a parallel rotate as illustrated below. The output from the barrel rotator can optionally be written to a separate destination.



For logical functions the appropriate Karnaugh map must be constructed. For the function (A&~B&~C) the Karnaugh map is:

A \ BC	00	01	11	10
0	F0 0	F2 0	F6 0	F4 0
1	F1 1	F3 0	F7 0	F5 0

Figure 2. Karnaugh Map for (A&~B&~C)

From the map it can be seen the output should only be a logical ‘1’ when the input A is ‘1’ and all of the other inputs are ‘0’. This function corresponds to the operation that will extract the orange sub-volume from the RGB color cube.

Each entry in the table is then read off in the order {F7 F6 F5 F4 F3 F2 F1 F0} and left shifted by 19. Thus the function is essentially:

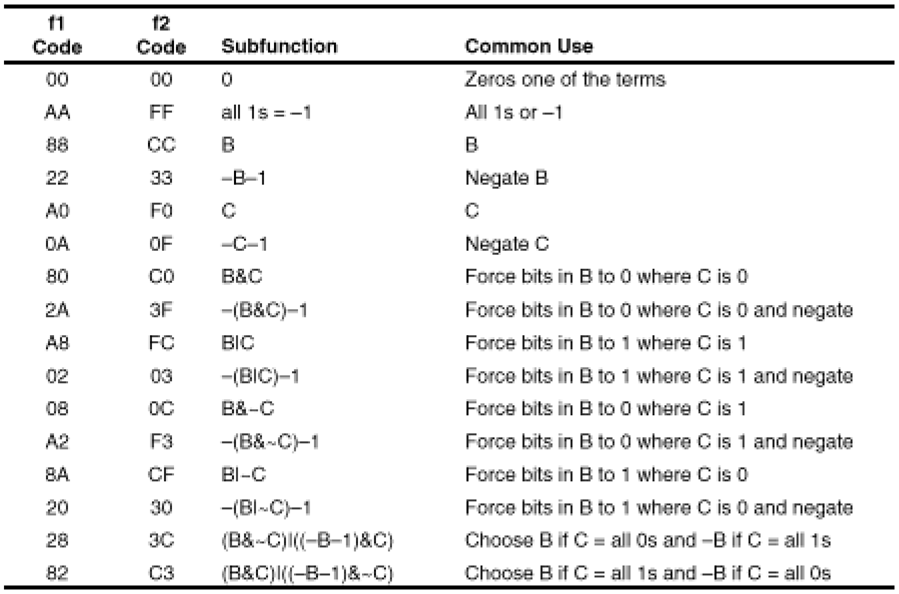
(F0&(~A&~B&~C) | F1&(A&~B~C) | F2&(~A&B&~C) | F3&(A&B&~C) |
F4&(~A&~B&C) | F5&(A&~B&C) | F6&(~A&B&C) | F7&(A&B&C))[+1|+cin]

For arithmetic functions the following functions can be formed

A&f1(B,C) + f2(B,C) [+1|+cin]

where the functions f1(B,C) and f2(B,C) are selected from the Table 1. The ALU opcode is determined by f1 XOR f2 left shifted by 19.

Table 1. Possible f1(B,C) or f2(B,C) Functions



f1 Code	f2 Code	Subfunction	Common Use
00	00	0	Zeros one of the terms
AA	FF	all 1s = –1	All 1s or –1
88	CC	B	B
22	33	–B–1	Negate B
A0	F0	C	C
0A	0F	–C–1	Negate C
80	C0	B&C	Force bits in B to 0 where C is 0
2A	3F	–(B&C)–1	Force bits in B to 0 where C is 0 and negate
A8	FC	B|C	Force bits in B to 1 where C is 1
02	03	–(B|C)–1	Force bits in B to 1 where C is 1 and negate
08	0C	B&~C	Force bits in B to 0 where C is 1
A2	F3	–(B&~C)–1	Force bits in B to 0 where C is 1 and negate
8A	CF	B|~C	Force bits in B to 1 where C is 0
20	30	–(B|~C)–1	Force bits in B to 1 where C is 0 and negate
28	3C	(B&~C)|((–B–1)&C)	Choose B if C = all 0s and –B if C = all 1s
82	C3	(B&C)|((–B–1)&~C)	Choose B if C = all 1s and –B if C = all 0s

Appendix E Optimizing Genesis Native Library Applications (Version 1.3)

This document will help you estimate the expected performance of Genesis on a particular application, and help you optimize your code so that you reach the expected performance. Although most of the general advice given here is likely to remain valid, particular details of individual functions may change with each release of the Genesis software, since improvements may be made, and/or new functions added. *You should always look for the most recent version of this document.*

The first step in writing an efficient application is to pick the functions that can do the job as fast as possible. To do this you need to know the execution time of individual functions. This is not as simple as it sounds, because the execution time of each processing function varies with the size and data type of the images processed, and can be affected by many other parameters. The execution time of some functions is even dependent on the actual content of the image. To address these issues, several types of information are provided:

- Actual benchmarks for each function with common data types.
- Simple rules for estimating the performance of cases not listed.
- A more-detailed discussion of functions whose performance is more difficult to estimate.

There are also some general points, described below, that can be applied to all processing functions. These will help you avoid some of the basic mistakes which can make your code inefficient.

E.1 Overheads

All functions have a fixed overhead, which means that they become less efficient when operating on small images. The overhead is not exactly the same for all functions, and it may be reduced in the future, but it is typically about 0.5 ms for each function call. This means that you cannot process an image in less time, no matter how small the image is. (Note that this overhead applies to processing functions; simple control functions have less overhead, and are discussed later.) This overhead has several implications:

When adjusting a known benchmark to a different image size, you cannot simply scale it according to the number of pixels in the image. You should first subtract the overhead, then scale the benchmark, then add the overhead again. For example, if a function with a 0.5 ms overhead takes a total of 2.5 ms for a 512x512 image, it will take 1.0 ms for a 256x256 image, and 8.5 ms for a 1024x1024 image.

When your application only requires that one or more regions of interest (ROIs) be processed, it might actually turn out to be more efficient to process the whole image than to define and process several ROIs separately. If your images themselves are quite small, you should consider packing several into a larger buffer and processing them all at once.

Simple asynchronous control functions (such as *imBufPutField()*), and other functions that don’t operate on images, have a lower overhead. It is typically 0.2 ms for these functions. However, synchronous functions (such as *imBufGetField()* or *imBufChild()*) are slower, and take about 0.5 ms under Windows NT (they are slightly faster under DOS).

You should try to avoid synchronous functions as much as possible within time-critical loops. Allocate all buffers outside of loops unless you have no choice. When using synchronous functions to read back processing results, use as few calls as possible. Sometimes you have the option of reading back results individually using one function call each, or reading a whole group of results at once. The latter method will be more efficient, and is particularly important for blob analysis and pattern matching where many results are usually produced.

E.2 Estimating Missing Benchmarks

When the particular case you need is supported by the library but not listed in the benchmark table, you can usually estimate its performance using a few simple rules. This document does *not* describe how to estimate the performance of functions that are not in the standard library, but that you might be interested in implementing as a custom function if the performance justifies it. (To implement custom functions on the C80 you need the Genesis Native Library Developer's Toolkit (DTK), as well as the appropriate code development tools from Texas Instruments. This is a fairly complex task, but can often provide a significant performance increase if several standard library functions can be combined into a single custom function.)

I/O bound functions

A function is I/O bound when performance is limited by the speed at which it can access data in memory. The processors will be idle some of the time while they are waiting for data to be transferred on-chip. It is useful here to know that processing functions using the PPs always work by transferring the image from external memory into on-chip memory, a block at a time. Each block is processed in on-chip memory, then the results are transferred back to external memory. Processing and I/O can be overlapped, so the PPs are not kept waiting for data to process, unless they process faster than data can be transferred (then the operation is said to be I/O bound).

If a function is described as I/O bound in the benchmark table, or if the I/O figure given is close to 300 MB/s which indicates that the function is I/O bound, it is simple to estimate benchmarks for cases not listed explicitly. You should simply consider the total number of bytes of I/O per pixel (counting all source and destination buffers), and scale the benchmark accordingly. For the most accurate estimate you should also consider the function overhead. For example, consider the I/O bound operations of *imIntDyadic()*. Given the 8-bit case, let's try to estimate the performance for the 16-bit case. The total I/O in the 8-bit case is 3 bytes/pixel, and it is 6 bytes/pixel in the 16-bit case (exactly twice as much). The function overhead can be calculated as 0.4 ms from the first two columns. The calculated performance for the 16-bit case is therefore (3.0–0.4)×2 + 0.4 = 5.6 ms, which is in agreement with the measured value. If, say, the two source buffers were 8-bit and the destination were 16-bit, the total I/O would be 4 bytes/pixel and you would expect the performance to lie somewhere between the listed 8- and 16-bit cases. In fact it should be (3.0–0.4)×4/3 + 0.4 = 3.9 ms.

Compute bound functions

Most compute bound functions are listed explicitly in the benchmark table, because you would need to know how the operation is coded in PP assembly language in order to estimate the performance. However, this doesn't apply to some functions. In particular, neighborhood operations which support different sized kernels (such as convolution or morphology) usually take a certain amount of time for each kernel element. Therefore you should look up the benchmark for a kernel similar in size to yours, then scale the benchmark according to the number of kernel values. For example, the C80 processing rate for a general 5x5 16-bit convolution (25 kernel values) is quoted as 7.65 MPix/s. The estimated performance for a 7x7 convolution (49 kernel values) would therefore be 7.65×25/49 = 3.90 MPix/s, which agrees very well with the measured value. This rule normally works well, but beware of particular kernels that might be carried out with optimized PP code and give quite different performance from the general case. For example, kernels whose values are all the same (or where only the center value is different) are faster on the C80 than kernels with completely arbitrary values. The situation with the NOA is similar but the rules are different. Pixel type (8- or 16-bit), coefficient type (8- or 16-bit), and kernel symmetry (horizontal and/or vertical) all affect performance considerably on any given size of kernel. This is discussed in more detail later.

E.3 Parallelism

There are times when you could execute several operations in parallel, by sending the commands to different threads, but this will not always reduce the total execution time. When you send commands to different processing nodes, they will truly run in parallel and they will have no impact on each other as long as each node processes buffers allocated in its own memory. However, when only one node is involved, things are more complicated.

First, to benefit from parallelism, each function must use different resources. For example, there is no point in executing two functions at the same time if they both use the PPs. If both try to allocate all the PPs (which is the default behavior) they will simply execute serially, since one function will have to wait for the other to finish completely before it is granted access to the PPs. If you use *imThrControl()* to limit each thread to, say, two PPs, both functions will run at the same time but each at only half speed (so there is no net gain in performance). Note that this is true whether the functions are compute bound (so each runs at half speed because it only has half of the processors) or I/O bound (where each runs at half speed because it only gets access to memory half of the time).

If two functions use different resources (for example the PPs and MP, or PPs and NOA) there will still be no advantage to executing them in parallel if both are I/O bound, since the available memory bandwidth is the limiting factor. However, if one or both functions are not I/O bound, there should be an advantage to running then in parallel. One common case involves processing an image while copying the previously-processed image to the display. When this is done serially (i.e. processing and copy commands are send to the same thread), the transfer time is simply added to the processing time. When this is done is parallel (i.e. the copy command is sent to another thread, with the proper synchronization), the transfer begins as soon as processing completes, but it does not prevent processing of the next image from starting. This way some or all of the transfer time is hidden (depending on how I/O intensive the processing function is). For more details on how to implement this, look at the Native Library examples.

It is also important to realize that using too many threads, and especially the extra synchronization functions that more threads usually implies, can make your application less efficient. There are certainly big performance gains to be had sometimes by using extra threads to exploit parallelism in your application, but you should generally try to solve the problem using the smallest number of threads that gives you the parallelism you need.

E.4 Benchmarks

Below is a table of actual benchmarks for 512x512 images, using a 50 MHz C80 with NOA . To keep the table to a reasonable size, some cases have been omitted if they can easily be estimated from other cases using the rules given elsewhere in this document. If a function has options that are not mentioned in the table (e.g. replace vs. transparent overscan), assume that the option makes no significant difference to the timing.

The table gives actual execution time (in milliseconds), processing rate (in MPixels/second), and the memory bandwidth required (in MBytes/second). The latter figure is useful when considering the impact of concurrent operations such as grabbing and transfers to display. If the operation is I/O bound, or nearly so, it will be slowed down by other operations that also need access to memory. You should assume that the sustainable bandwidth on processing memory (SDRAM) is 300 MB/s (the peak rate is 400 MB/s). Hence if grabbing and other operations require, say, 30 MB/s, they will slow down an I/O bound processing function by about 10% if executed in parallel. The effect on a compute bound function should be negligible. The quoted I/O rates are approximate values taking into account any extra I/O that might be performed internally to hold intermediate results etc.

Function overheads are included in the measured execution times, so you must take them into account when scaling the numbers to a different image size (see the section on overheads for details). Overheads are *not* included in the processing or I/O rates, so you can more easily scale these to different image sizes. The difference between the measured execution time and the expected time calculated from the processing rate and image size is the actual fixed overhead for that function.

Note that some of the numbers may change in future releases as improvements are made. *You should always look for the most recent version of these benchmarks.* Entries marked with an asterisk (*) are further explained after the table. Entries marked with † are data dependent, and performance will vary from image to image.

Function and options	**Time (ms)** *with overhead*		**Rate (MPix/s)** *without overhead*		**I/O (MB/s)** *without overhead*	
imBinConvert()						
8-bit to/from binary	1.4		260		290	
16-bit to/from binary	2.3		133		280	
32-bit to/from binary	4.4		65		270	
imBinMorphic()*						
IM_ERODE/IM_DILATE	10.10.1.	10.10.1.	10.10.1.	10.10.1.	10.10.1.	10.10.1
IM_3X3_RECT_1, 1 iteration	1.0	1.3	470	500	120	125
IM_3X3_RECT_1, 2 iterations	1.3	1.4	290	500	72	125
IM_3X3_RECT_1, 3 iterations	1.7	1.5	210	500	53	125
All operations except IM_MATCH						
General 3x3	2.2	1.3	170	500	42	125
General 5x5	4.1	1.4	76	500	19	125
General 7x7	6.8	1.5	43	500	11	125
General 9x9	10.4	1.9	27.6	320	7	81
General 11x11	14.7	2.5	19.2	215	5	54
IM_MATCH						
General 3x3	17.8	3.1	15.2	143	17	160
General 5x5	25.8	3.2	10.4	141	12	160
General 16x16	70.7	5.5	3.78	75.6	4	85
General 32x32	288	15.8	0.92	23.8	2	51
imBinTriadic()						
No inputs	0.6		980		120	
One input	0.8		580		140	
Two inputs	1.0		420		160	
Three inputs	1.2		320		160	
imBlobCalculate()*†						
8-bit, 9 blobs, total area 5% of image						
Area only	~2.7					
Area + binary COG	~3.0					
Area + gray COG	~3.6					
8-bit, 100 blobs, total area 25%						
Area only	~9					
Area + binary COG	~12					
Area + gray COG	~16					
imBufClear()						
Binary	0.7		940		120	
8-bit	1.1		350		350	
16-bit	1.7		190		380	
32-bit	3.1		95		380	
imBufCopy()						
IM_PROC to IM_PROC (on same node)						
Binary	0.9		580		150	
8-bit	2.0		170		340	
16-bit	3.5		88		350	
32-bit	6.4		44		350	
IM_PROC to IM_DISP (VIA driven)*						

8-bit, PCI write	3.6	86	86
8-bit, PCI read	4.6	66	66
8-bit, VM	3.1	118	118
IM_PROC to IM_HOST (VIA driven)*			
8-bit	3.6	86	86
IM_HOST to IM_PROC (VIA driven)*			
8-bit	3.3	99	99
imBufGet()*			
8-bit	29.0	9.2	9.2
imBufPut()*			
8-bit	5.1	61	61
imBufPack()†			
8-bit, none tagged	0.6	1500	190
8-bit, all tagged	2.7	110	240
8-bit, 35% tagged (circular mask)	1.4	270	230
16-bit, 35% tagged (circular mask)	2.2	150	230
32-bit, 35% tagged (circular mask)	3.5	85	240
imFloatConvert()			
8-bit to float	4.6	67.0	330
16-bit to float	12.1	22.7	140
32-bit to float	7.1	38.4	310
float to 32-bit	7.1	38.4	310
imFloatDyadic()			
IM_ADD, IM_SUB	12.4	21.7	260
IM_SUB_ABS	15.7	17.1	200
IM_MIN, IM_MAX	27.5	9.6	110
IM_MULT	10.5	25.8	310
IM_DIV	40.7	6.5	78
IM_SQUARE_ADD	19.0	14.0	170
IM_ATAN2	900	0.29	3.5
imFloatMac1()	11.4	23.7	190
imFloatMac2()	19.0	14.0	170
imFloatMonadic()			
IM_ADD, IM_SUB	7.1	38.4	310
IM_SUB_ABS, IM_SUB_NEG	12.3	21.8	170
IM_MIN, IM_MAX	24.8	10.7	85
IM_MULT, IM_DIV	7.1	38.4	310
IM_DIV_INTO	38.0	7.0	56
imFloatUnary()			
IM_NEG	7.1	38.4	310
IM_ABS	12.3	21.8	170
IM_SQUARE	7.1	38.4	310
IM_SQRT	53.7	4.9	39
IM_CUBE	12.9	20.7	170
IM_LOG, IM_EXP, IM_SIN, IM_COS, IM_TAN, IM_ATAN	700 – 900	~0.3	~2.5
IM_CBRT	1700	0.15	1.2
imGenWarpLutMatrix()			
IM_CTL_PRECISION = 0	63.5	4.2	17
IM_CTL_PRECISION > 0	74.0	3.6	14
imIntBinarize()			
IM_IN_RANGE/ IM_OUT_RANGE			
8-bit	2.1	148	300
16-bit	3.9	75	220
32-bit	7.4	37	190

All other conditions						
8-bit	1.9		173		340	
16-bit	2.7		114		340	
32-bit	4.3		67		340	
imIntConnectMap()						
8-bit to 8-bit	7.0		40		80	
8-bit to 16-bit	7.3		38		110	
imIntConvert()						
8-bit to 16-bit	3.2		93		280	
8-bit to 32-bit	4.5		64		320	
16-bit to 32-bit	5.2		55		330	
16-bit to 8-bit (truncate)	3.2		94		280	
Unsigned 16-bit to 8-bit (IM_CLIP)	3.5		84		250	
Signed 16-bit to 8-bit (IM_CLIP)	5.8		48		140	
Signed 16-bit to 8-bit (IM_ABS_CLIP)	4.5		63		190	
32-bit to 8-bit (IM_CLIP)	6.5		43		210	
32-bit to 16-bit (IM_CLIP)	6.6		39		250	
imIntConvertColor()						
8-bit IM_RGB_TO_HSL	36.4		7.2		43	
8-bit IM_HSL_TO_RGB	44.6		5.9		35	
8-bit IM_RGB_TO_H	30.0		8.9		35	
8-bit IM_RGB_TO_L	6.5		43		170	
8-bit IM_RGB_TO_I	5.7		61		240	
8-bit IM_L_TO_RGB	5.9		57		230	
8-bit IM_MATRIX, 3x1 no clip	5.3		49		200	
8-bit IM_MATRIX, 3x1 unsigned clip	7.7		34		140	
8-bit IM_MATRIX, 3x1 signed clip	9.0		29		120	
8-bit IM_MATRIX, 3x3 no clip	14.5		18.1		110	
8-bit IM_MATRIX, 3x3 unsigned clip	22.3		11.8		71	
8-bit IM_MATRIX, 3x3 signed clip	26.3		10.0		60	
imIntConvolve()*						
8-bit *(see detailed description later)*						
IM_SMOOTH	10.10.1.	10.10.1.	*10.10.1.*	*10.10.1.*	10.10.1.	10.10.1
IM_SHARPEN						
IM_SHARPEN2						
IM_HORIZ_EDGE						
IM_VERT_EDGE	9.7	2.3	28.8	160	58	320
IM_SOBEL_EDGE	8.6	2.3	32.3	160	65	320
IM_PREWITT_EDGE	9.4	2.3	29.8	160	60	320
IM_LAPLACIAN_EDGE	7.3	2.3	38.3	160	77	320
IM_LAPLACIAN_EDGE2	4.7	2.3	62.2	160	120	320
IM_ROBERTS_EDGE	11.3	11.3	24.2	24.2	48	48
	11.3	11.3	24.2	24.2	48	48
General kernel	8.6	2.3	32.3	160	65	320
3x3	9.5	2.3	29.7	160	60	320
5x5	3.0	3.0	101	101	200	200
7x7						
9x9						
11x11	9.7	2.3	28.8	160	58	320
Symmetric kernel	26.0	4.0	10.4	80	21	160
5x5	44.9	5.5	5.96	55	12	110
7x7	69.4	8.2	3.84	35	7.7	70
9x9	99.2	11.8	2.68	24	5.4	48
11x11						
All 1's kernel (16-bit output)	26.0	3.2	10.4	107	21	215
	44.9	4.9	5.96	62	12	125

5x5	69.4	5.8	3.84	52	7.7	105
11x11	99.2	6.8	2.68	44	5.4	87
Effect of normalization options:						
No Shift (16-bit output)	5.7	3.2	52	107	156	315
Clip/Absolute	8.5	6.8	35	44	105	130
imIntConvolve()*	–2.1	+0.0				
16-bit *(see detailed description later)*	+0.7	+0.0				
IM_SMOOTH						
IM_SHARPEN						
IM_SHARPEN2				**NOA**	**C80**	**NOA**
IM_HORIZ_EDGE	*10.10.1.*	10.10.1.	***10.10.1.***	74	84	300
IM_VERT_EDGE				74	130	300
IM_SOBEL_EDGE				74	68	300
IM_PREWITT_EDGE	13.3	4.2	20.9	74	150	300
IM_LAPLACIAN_EDGE	8.8	4.2	31.6	74	140	300
IM_LAPLACIAN_EDGE2	16.1	4.2	16.9	23.6	94	94
IM_ROBERTS_EDGE	7.5	4.2	37.2	23.6	94	94
	7.6	4.2	36.5	74	130	300
General kernel	11.6	11.6	23.6	74	68	300
3x3	11.6	11.6	23.6	52.4	210	210
5x5	8.8	4.2	31.6			
7x7	16.1	4.2	16.9			
9x9	5.4	5.4	52.4	74	84	300
11x11				32	31	125
Symmetric kernel				16.1	16	65
5x5	13.3	4.2	20.9	9.8	9.4	39
7x7	35.2	9.0	7.65	6.6	6.3	26
9x9	68.3	17.0	3.91			
11x11	112.8	27.6	2.36	65	31	260
All 1's kernel	168.6	40.8	1.57	47	16	190
5x5				30	9.4	120
11x11	35.2	4.8	7.65	22	6.3	87
Effect of normalization options:	68.3	6.3	3.91			
Shift	112.8	9.4	2.36	65	99	260
Clip/Absolute	168.6	13.0	1.57	32	60	130
	11.2	4.8	24.7			
	18.3	9.0	15.1			
	+0.0	+0.0				
	10.10.1.	**10.10.1.**	***10.10.1.***			
imIntCorrelate()						
8x8 model, STEP = 1	305		0.86		2.6	
8x8 model, STEP = 2	227		0.24		0.7	
16x16 model, STEP = 1	1090		1.16		3.5	
16x16 model, STEP = 2	307		0.85		2.6	
imIntCountDifference()						
8-bit	2.0		160		320	
16-bit	3.7		79		320	

32-bit	6.9		40		320	
imIntDistance()						
IM_CITY_BLOCK						
8-bit	12.4		21.9		88	
8-bit to 16-bit	14.6		18.6		130	
All other cases	19.2		13.9		98	
imIntDyadic()						
IM_DIV						
8-bit	18.6		14.4		43	
16-bit	29.1		9.1		55	
32-bit	47.3		5.6		67	
IM_DIV_FRAC						
8-bit to 16-bit	32.3		8.2		33	
16-bit to 32-bit	52.7		5.0		40	
IM_MULT_MSB						
32-bit	13.0		21		250	
All other operations (I/O bound)						
8-bit	3.0		102		310	
16-bit	5.6		51		310	
32-bit	10.7		26		310	
imIntErodeDilate()*			**C80**	**NOA**	**C80**	**NOA**
Unsigned 8-bit	10.10.1.	10.10.1.				
Kernel IM_3X3_RECT_0			62	160	120	320
Kernel all zero						
5x5			11.4	132	23	265
7x7			6.0	91	12	180
9x9	4.7	2.2	3.8	62	7.7	125
11x11			2.5	44	5.1	87
General kernel (not all zero)	23.7	2.5				
3x3	44.7	3.5	14.1	160	28	320
5x5	69.4	4.8	5.2	64	11	130
7x7	105	6.7	2.7	32	5.4	64
9x9			1.6	19.6	3.3	39
11x11	19.2	2.2	1.1	13.2	2.2	26
	50.7	4.7				
Unsigned 16-bit	97.8	8.7				
Kernel IM_3X3_RECT_0	161	14.0	35	75	140	300
Kernel all zero	240	20.6				
5x5			5.5	64	23	260
7x7			3.0	48	12	190
9x9	7.9	4.0	1.9	38	8	150
11x11			1.3	32	5	130
General kernel (not all zero)	47.3	4.6				
3x3	88.1	6.1	13.9	75	56	300
5x5	141	7.5	5.2	62	21	250
7x7	207	8.9	2.7	32	11	130
9x9			1.6	19.3	6.5	77
11x11	19.5	4.1	1.1	13.0	4.4	52
	51.0	4.8				
Overhead for signed data (all cases)	98.3	8.8				
	161	14.2				
	240	20.8				
	+0.9	+0.0				
imIntFindExtreme()						
IM_MIN or IM_MAX (only one)						

Unsigned 8-bit	1.2	310	310
Unsigned 16-bit	2.0	160	320
32-bit	3.6	80	320
IM_MIN and IM_MAX (both)			
Unsigned 8-bit	1.8	190	190
Unsigned 16-bit	3.1	96	190
32-bit	4.5	64	250
imIntFFT()			
2-d (forward or reverse)	240	1.1	
1-d (i.e. horizontal pass only on 512 lines)	105	2.5	
imIntFlip()			
Flip or 180 degree rotate			
8-bit	2.1	150	300
16-bit	3.5	83	330
32-bit	6.7	41	330
90 or 270 degree rotate			
8-bit	2.7	110	220
16-bit	5.6	49	200
32-bit	12.3	22	170
imIntGainOffset()			
8-bit			
No clip or offset	2.9	102	310
Clip but no offset	3.1	97	290
Clip and offset	5.8	49	200
16-bit			
No clip or offset	5.5	51	310
Clip but no offset	5.5	51	310
Clip and offset	6.9	40	320
imIntHistogram()*			
8-bit	2.7	123	120
10-bit, short table	4.0	89	180
10-bit, long table	8.9	33	65
12-bit, short table	10.2	33	66
imIntLabel()			
Few blobs (fastest)	9.2	31	180
Many blobs (slower)	≥15	≤17	≤100
imIntLocateEvent()†			
100 events			
Number only	2.0	172	170
With X, Y positions	2.7	146	150
10000 events			
Number only	6.2	46	46
With X, Y positions	11.5	25	25
imIntLutMap()*			
Default method			
8-bit to 8-bit	2.1	150	300
8-bit to 16-bit	2.7	114	340
8-bit to 32-bit	4.3	67	340
14-bit to 8-bit (16 KB table)	4.2	73	220
13-bit to 16-bit (16 KB table)	4.3	68	270
12-bit to 32-bit (16 KB table)	7.0	41	250
16-bit to 8-bit†	~26	~10	~30
16-bit to 16-bit†	~35	~7	~30
16-bit to 32-bit†	~43	~6	~30
Method with work buffer			

16-bit to 8-bit	21	12.4	110
16-bit to 16-bit	24	11.0	130
16-bit to 32-bit†	~50	~5	~90
imIntMac1()			
8-bit to 8-bit	3.1	96	190
8-bit to 16-bit	3.2	92	280
16-bit to 16-bit	3.6	82	330
imIntMac2()			
8-bit to 8-bit	3.2	95	280
8- and 16-bit to16-bit	4.8	60	300
16-bit to 16-bit	5.5	52	310
imIntMonadic()			
IM_DIV			
8-bit	15.1	17.9	35
16-bit	25.8	10.3	41
32-bit	47.2	5.6	45
IM_DIV_FRAC			
8-bit to 16-bit	27.2	9.8	29
16-bit to 32-bit	49.9	5.3	32
IM_MULT_MSB			
32-bit	12.6	21	170
All other operations (I/O bound)			
8-bit	2.0	163	330
16-bit	3.6	82	330
32-bit	6.8	41	330
imIntProject()			
0.0 degrees			
8-bit to 32-bit	2.1	147	150
16-bit to 32-bit	2.6	117	230
32-bit to 32-bit	3.9	75	300
90.0 degrees			
8-bit to 32-bit	1.8	184	180
16-bit to 32-bit	2.4	126	250
32-bit to 32-bit	4.3	67	270
imIntRank()			
IM_3X3_RECT			
8-bit median	11.1	24.6	49
16-bit median	24.3	11.0	43
IM_3X3_CROSS			
8-bit median	8.4	32.8	66
16-bit median	20.3	13.2	53
IM_3X3_X			
8-bit median	6.5	43.2	86
16-bit median	12.5	21.6	86
IM_1X5			
8-bit median	6.4	44.2	88
16-bit median	11.9	22.8	91
IM_5X1			
8-bit median	8.8	31.1	62
16-bit median	14.5	18.6	74
imIntRecFilter()			
8-bit to 16-bit (no Dst2)	5.5	52	260
8-bit to 16-bit (with Dst2)	6.2	46	270
16-bit to 16-bit (no Dst2)	6.1	46	280
16-bit to 16-bit (with Dst2)	7.5	37	260

imIntScale()						
IM_INTERPOLATE						
8-bit 2×2	3.8		82		103	
8-bit 4×4	3.0		108		115	
8-bit 8×8	2.9		112		115	
8-bit arbitrary factor	6.6		42			
16-bit arbitrary factor	7.8		36			
IM_NO_INTERPOLATE						
8-bit arbitrary factor	3.7		81			
16-bit arbitrary factor	5.1		56			
imIntSubsample()*				NOA	**C80**	**NOA**
IM_INTERPOLATE	*10.10.1.*	10.10.1.	***10.10.1.***			
8-bit (any supported factor)				235		
16-bit (any supported factor)				117		
IM_NO_INTERPOLATE						
8-bit 2×2	2.3	2.1	140	470	260	340
8-bit 4×4	3.4	3.2	86		220	
16-bit 2×2				200	260	300
16-bit 4×4	1.1	1.1	350		240	
	0.7		690			
	1.9	1.8	170			
	1.1		380			
imIntThickThin()						
8-bit						
Single 3x3 kernel	21.2		12.8		25	
8-band 3x3 kernel	154		1.73		27	
16-bit						
Single 3x3 kernel	41.7		6.4		25	
8-band 3x3 kernel	308		0.86		27	
imIntTriadic()						
All operations are I/O-bound						
2 bytes of I/O per pixel	1.9		170		340	
3 bytes of I/O per pixel	2.7		110		340	
4 bytes of I/O per pixel	3.7		80		320	
5 bytes of I/O per pixel	4.6		63		310	
6 bytes of I/O per pixel	5.6		51		310	
imIntWarpLut()*						
Nearest neighbor						
8-bit	~10		~27		~160	
16-bit	~11		~25		~200	
32-bit	~13		~20		~240	
Bilinear interpolation						
8-bit	~21		~12		~80	
16-bit	~22		~12		~100	
imIntWarpPolynomial()*						
Nearest neighbor						
8-bit	~8		~39		~80	
16-bit	~10		~30		~120	
32-bit	~18		~15		~120	
Bilinear interpolation						
8-bit	~20		~14		~30	
16-bit	~29		~9		~40	
Bicubic interpolation						
8-bit	~70		~3.8		~8	
16-bit	~92		~2.9		~12	

imIntZoom()*	**C80**		**C80**	NOA	**C80**	NOA
IM_INTERPOLATE		10.10.1.				
8-bit (any supported factor)	6.5		44	180		
16-bit (any supported factor)	6.7		42	120		
IM_NO_INTERPOLATE						
8-bit 2x2	2.2	2.5	150	220	190	270
8-bit 4x4	2.0	3.2	170	275	180	310
16-bit 2x2	3.0		105	135	130	340
16-bit 4x4	3.6		125	160	130	350
		2.2				
		2.0				
		3.0				
		2.6				
imJpegDecode()†*			**C80**	**NOA**	**C80**	**NOA**
8-bit lossless	*10.10.1.*	10.10.1.	~11	~45	~17	~70
8-bit lossy			~18			
	~23	~6.8				
	~14					
imJpegEncode()†*			**C80**	**NOA**	**C80**	**NOA**
8-bit lossless	*10.10.1.*	10.10.1.	~14	~100	~21	~150
8-bit lossy			~16			
	~19	~3.3				
	~16					
imPatFindModel()†						
Speed IM_HIGH, accuracy IM_MEDIUM						
32x32 model	~26					
64x64 model	~12					
128x128 model	~8					
256x256 model	~14					
Speed IM_MEDIUM, accuracy IM_HIGH						
32x32 model	~26					
64x64 model	~12					
128x128 model	~12					
256x256 model	~25					

***** Means that the function is discussed further below.
† Means that the performance is data dependent.

E.5 Specific Functions

The behavior of some functions is more complicated that the benchmark table indicates, and they need to be discussed individually.

imBinMorphic()
The pre-defined kernel IM_3X3_RECT_1 has been specially optimized for the C80, and there is a particularly large performance gain when multiple iterations are required. Therefore, with the C80 you should decompose large erosions or dilations into multiple iterations of IM_3X3_RECT_1 whenever possible. However, with the NOA it is just as good to define large kernels directly and use only one iteration. In general, binary morphology is so fast with the NOA that the function overhead is very important on small images, and it might sometimes be faster to disable the NOA and use the C80 instead. However, there is a way to reduce the NOA setup overhead for the second and subsequent passes with a given kernel (see the documentation for the control fields supported by the function).

imBlobCalculate()

Performance here is data dependent, and the more blobs are present the longer it takes. If your images are noisy, so that after thresholding many spurious blobs are present, it is usually best to clean up the thresholded image with morphology before doing blob analysis. To make sure this cleaning stage runs as fast as possible, threshold your original image into a 1-bit binary buffer.

The benchmarks in the table were obtained with the saving of run information disabled (i.e. after calling *imBlobControl()* with IM_BLOB_SAVE_RUNS set to IM_DISABLE). If you save runs (which is the default behavior) processing time will be slightly longer.

Another important point in blob analysis it to read back as many results as you can at once. This means making use of the data structures, IM_BLOB_GROUP1_ST etc., that hold a whole group of blob results. See the example program BLOB.C for usage.

imBufCopy()

Performance of PCI transfers between the host and Genesis are dependent on the host PCI chipset, so you should do your own tests to measure the performance on your system. See also the following discussion on *imBufGet()* and *imBufPut()*. Note that VM performance depends on the VM channel clock speed. This can be set to 25 or 33 MHz in the GENESIS.INI file.

The performance of both PCI and VM transfers varies somewhat with the width of the image being transferred.

imBufGet()/imBufPut()

In these functions the data transfer is driven by the host CPU. Writes to Genesis memory are much faster than reads, although actual figures are dependent on the host PCI chipset. *imBufGet()* is normally adequate for small amounts of data such as a histogram result (which is typically a few Kbytes), but for whole images it is much faster to allocate a host buffer and use *imBufCopy()*. On some systems *imBufPut()* might be the fastest way to get even large amounts of data from the host to the Genesis, but it ties up the host CPU. *imBufCopy()* should provide comparable speed and does not tie up the host CPU.

imIntConvolve()

The main point about convolution is that the 8-bit case runs considerably faster than the 16-bit case. For the C80 the 8-bit case requires that the source buffer be 8-bit unsigned, and that the kernel values all lie in the 8-bit signed range [–128, +127]. (Note that the type of the kernel buffer itself is irrelevant; the actual kernel values will be tested to see if the 8-bit convolution can be used.) Another requirement for the 8-bit case on the C80 is that it must be possible to use a 16-bit signed accumulator without causing possible overflows. This means that the sum of the positive kernel values must not exceed 128, and the sum of the negative kernel values must not exceed –128. If not all of the 8-bit requirements are met, the 16-bit benchmarks will apply.

Note that many predefined kernels use specially optimized C80 code, and execute faster than the equivalent kernel that you define yourself. Always use a predefined kernel if you can.

On the NOA the 8-bit case is not so restricted. Any 8-bit type (for image or kernel) is handled at the same speed, and there is no limit on the sum of kernel values. If you have 16-bit data with an 8-bit kernel, or 8-bit data with a 16-bit kernel, the speed will be somewhere between the listed 8- and 16-bit benchmarks. The slowest case of all is that of 16-bit data *and* a 16-bit kernel (and this is what is used for the listed 16-bit NOA benchmarks). Note that the NOA 8-bit benchmarks were made in exact mode (the computation control field, IM_CTL_COMPUTATION, was set to IM_EXACT so that no approximations were made). Some symmetric kernels will show a slight improvement in fast mode (where the computation control field is set to IM_FAST). The NOA 16-bit benchmarks for symmetric kernels were made in fast mode. However, if your data is 14-bit or less, you will get the same performance in exact mode as long as you specify the number of input bits.

You will see that the NOA takes advantage of symmetric kernels (positive or negative symmetry, in either the horizontal or vertical direction). The symmetric benchmarks listed assume both horizontal and vertical symmetry (which is quite common for real convolution kernels). The C80 cannot take advantage of kernel symmetry, but it does have specially optimized code for kernels whose values are all 1. Furthermore, the time for an all 1's kernel is only weakly dependent on the kernel size. Kernels whose values are some other constant (optionally with a different center value) are only slightly slower on the C80 than the same-sized all 1's kernel. The NOA does not handle constant kernels as a special case, but they are faster than general kernels because of their symmetry.

The number of special cases is simply too large to describe here. If convolution is a time critical operation for you, you should benchmark your particular kernel to see how fast it runs. When using the NOA, you should also consider using fast mode. This significantly improves the speed of some cases with only a slight loss of precision. Note also that there is a way to reduce the NOA setup overhead for the second and subsequent passes with a given kernel (see the documentation for the control fields supported by the function).

imIntErodeDilate()
With the NOA any symmetric kernel will run at the same speed as the all-zero case. With the C80 there is no advantage to symmetric kernels (only the all-zero case has been optimized). As you can see, large all-zero kernels are better implemented as multiple passes with IM_3X3_RECT_0 on the C80 (assuming they can be decomposed this way), but this is not true for the NOA. Note also that there is a way to reduce the NOA setup overhead for the second and subsequent passes with a given kernel (see the documentation for the control fields supported by the function).

imIntHistogram()
Performance here is very dependent on the type of the input data (i.e. the size of the histogram result buffer). For 12-bit or larger data, you should consider not using every pixel in the histogram calculation (see the control fields which enable subsampling). You might also consider shifting the data down to 8 or 10 bits before performing the histogram (see the control field which specifies the number of bits you want to use).

imIntLutMap()
Here the performance is very good when the lookup table is no bigger than 16KB, because then the table fits entirely into the C80's internal RAM (assuming all four PPs are used, otherwise the table must be proportionally smaller). Performance with very large LUTs is also data dependent, so you may need to try both the available methods to see which performs best in your case. If you have 16-bit data, you should consider using an interpolated LUT mapping (which uses a smaller table).

imIntSubsample()
The quoted processing rates are in terms of *input* pixels/second. Since the output image is smaller, the processing rate will be lower in terms of *output* pixels/seconds. Note that the NOA can be used only for subsampling factors of 1 or 2, but there is only a significant speedup on images larger than about 512x512. This is because the higher processing rate of the NOA is offset by a bigger overhead. On small images it might be better to disable the NOA.

imIntWarpLut()
All figures for this function are approximate since performance is weakly dependent on the actual transformation (i.e. the values in the Xlut and Ylut buffers).

imIntWarpPolynomial()
All figures for this function are approximate since performance is weakly dependent on the actual transformation. Replace overscan may also be slightly faster than transparent overscan since in the former case some parts of the output buffer can simply be replaced with a constant value (if that region originated outside the source buffer).

imIntZoom()

The quoted processing rates are in terms of *output* pixels/second. The NOA can be used only for zoom factors of 1, 2 or 4. The processing rate is higher for the NOA but the overhead is also higher. Hence on small images it might be better to disable the NOA.

imJpegEncode()/imJpegDecode()
The times are slightly data dependent, and in the lossy case are also affected by the Q factor (higher compression produces slightly faster times). You can also improve performance a little by increasing the restart interval (increase the value of IM_JPEG_RESTART_ROWS from its default of 32). When you also need to transfer the compressed image on or off the board, you should also consider the time taken by *imJpegReadBuf()/imJpegWriteBuf()*. However, these functions can be overlapped with the actual encoding or decoding by running them in another thread. In this case they add very little extra time.

Appendix F Vision Code Structure

F.1 ball_filter.c

Header File: track.h

int trackBallPosition (struct trackingStruct* track, struct calibrationData* calibData)

Description: Filters the ball position and linear velocity in image coordinates. The barrel distortion and parallax error are also removed. If the ball is not found the data is extrapolated from the previous frames. The position is clipped if the filter overshoots the image size.

Inputs: struct trackingStruct* track

Contains all of the information necessary for tracking.

struct calibrationData* calibData

Contains all of the calibration information including the size of the window and the field in terms of image coordinates.

Return Value: The number of ball blobs that were found. This is either zero or one.

F.2 barreldist.c

Header File: distortion.h

void brlDistCorrect(struct point* pointIn, int number, struct calibrationData* calibData)

Description: Removes the barrel distortion. The distortion is a function of the lens and the distance that the pixel is radially from the center pixel in the image.

Inputs: struct point* pointIn

The list of points that are to be corrected in terms of image coordinates.

int number

The number of points that are to be corrected.

struct calibrationData* calibData

Contains all of the calibration information including the polynomial that is used to remove the barrel distortion.

Return Value: void

F.3 calibration.c

Header Files: calibration.h

struct calibrationData* readCalibrationFile(struct calibrationData* calibData, char* filename)

Description: Reads a calibration file and computes the other necessary parameters from the file. This function allows for reading of either the old or the new calibration file format. If a valid structure is not passed as an input to the function, it will allocate a new structure on the heap. Otherwise, the data will be overwritten using the values read from the file.

Inputs: struct calibrationData* calibData

Contains all calibration information.

char* filename

The name of the file that is to be read for calibration information.

Return Value: The address of the structure that contains the valid information.

F.4 fileLogging.c

Header File: vision.h

struct fileLogStruct* initializeLogFile(struct fileLogStruct* log, char* filename)

Description: Initializes the log file struct. Opens the appropriate file and writes the header information in a format such that Matlab will regard the header as comments. If a valid structure is not passed as an input to the function, it will allocate a new structure on the heap. Otherwise, it will reinitialize everything, and reopen the file.

Inputs: struct fileLogStruct* log

Contains the pointer to the open file and the current date.

char* filename

The name of the file to open for writing vision data.

Return Value: The address of the structure that contains the valid information.

void outputPositions (struct fileLogStruct* log, struct trackingStruct* track)

Description: Writes the appropriate information about the objects that are being tracked into the file. The data for each frame appears as a single line on the output. The first four entries correspond to the ball. The next six entries correspond to either the Brazil or Cornell robots. The remaining six entries correspond to either the Italy or opposing robots.

Inputs: struct fileLogStruct* log

Contains the pointer to the open file that the data is being written to.

struct trackingStruct* track

Contains the tracking information for all objects that can be in the field.

Return Value: void

void freeLogFile (struct fileLogStruct* log)

Description: Closes the file that is being written to and frees the allocated memory for the structure on the heap.

Inputs: struct fileLogStruct* log

Contains the pointer to the file log structure.

Return Value: void

F.5 identification.c

Header File: identification.h

void identifyBlobs (struct profile* orderedRobots, struct profile* oldRobots, int numRobots, struct state* newRobots, int numNew, int trackingRobot[])

Description: Identifies the robot blobs based on the current position of the blobs and where the robots were located in the previous frame. This function only identifies the robots, but does not do any filtering on the data. The velocity of the robots is not taken into account, only the position.

Inputs: struct profile* orderedRobots

The profiles for the ordered list of robots that correspond to the initial ordering of robot positions. This is the new data that has been ordered.

struct profile* oldRobots

The old data referring to the locations of the robots in the previous frame.

int numRobots

The number of elements that are in the array pointed to by oldRobots.

struct state* newRobots

The new blobs that map to the team marker color and are candidates for robot identification. These blobs are oriented and contain position information in terms of image coordinates and orientation in terms of radians.

int numNew

The number of elements that are in the array pointed to by newRobots.

int trackingRobot[]

An array of type int that stores the current state of tracking for the robots from the previous frame. Updates to this array are done in place.

Return Value: void

F.6 imageProcessing.c

Header File: vision.h

void processFrame(struct genesisStruct* genesis, struct calibrationData* calibData, struct timingStruct* timing, int num);

void processFrame(struct genesisStruct* genesis, struct calibrationData* calibData, int num);

Description: Performs the color segmentation algorithm. The algorithm can optionally perform difference imaging to clean the image. The function can also perform timing on all of the genesis function calls.

Inputs: struct genesisStruct* genesis

Contains all buffers and pointers necessary to perform color segmentation on the genesis board.

struct calibrationData* calibData

Contains all calibration information including the color threshold values.

struct timingStruct* timing

Contains the timing statistics for all processing functions that take place on the DSP board.

int num

The current buffer index that is marked as processing in the circular grab buffer.

Return Value: void

void analyseBlobs(struct genesisStruct* genesis, struct blobAnalysisStruct* blobAnalysis, struct timingStruct* timing);

void analyseBlobs(struct genesisStruct* genesis, struct blobAnalysisStruct* blobAnalysis);

Description: Performs the blob analysis and transfers the results back to the host computer. The function can also perform timing on all of the genesis function calls.

Inputs: struct genesisStruct* genesis

Contains the results of the color thresholding.

struct blobAnalysisStruct* blobAnalysis

Contains all of the buffers, threads, and the result buffer in host memory to perform the blob analysis.

struct timingStruct* timing

Contains the timing statistics for the processing functions that perform blob analysis.

Return Value: void

F.7 initialization.c

Header File: vision.h

struct trackingStruct* initializeTrackingStruct (struct trackingStruct* track)

Description: Initializes the tracking structure. If a valid structure is not passed as an input to the function, it will allocate a new structure on the heap. Otherwise, it will reinitialize the variables. All tracking profiles are initialized to zero and all tracking states are set to NOT_TRACKING.

Inputs: struct trackingStruct* track

The structure that is to be initialized.

Return Value: The address of the structure that contains the valid information.

struct genesisStruct* initializeGenesisBoard(struct genesisStruct* genesis, struct calibrationData* calibData)

Description: Allocates all the threads, the camera, the genesis board, and buffers that are to be used for the image processing. The buffers and allocated and the child buffers are assigned. If a valid structure is not passed as an input to the function, it will allocate a new structure on the heap. Otherwise, the variables in the structure are overwritten and reallocated.

Inputs: struct genesisStruct* genesis

The structure that is to be initialized.

struct calibrationData* calibData

Contains the calibration information including the image buffer sizes and the window sizes and position.

Return Value: The address of the structure that contains the valid information.

struct blobAnalysisStruct* initializeBlobAnalysis (struct genesisStruct* genesis, struct blobAnalysisStruct* blobAnalysis)

Description: Allocates the blob analysis buffers, and threads. The child buffers are assigned. If a valid structure is not passed as an input to the function, it will allocate a new structure on the heap. Otherwise, the variables in the structure are overwritten and reallocated.

Inputs: struct genesisStruct* genesis

Contains the address of the genesis device

struct blobAnalysisStruct* blobAnalysis

The structure that is to be initialized.

Return Value: The address of the structure that contains the valid information.

struct timingStruct* initializeTiming(struct timingStruct* timing)

Description: Initializes the timing variables to zero. If a valid structure is not passed as an input to the function, it will allocate a new structure on the heap. Otherwise, the timing variables are all set to zero.

Input: struct timingStruct* timing

Contains all of the timing information.

Return Value: The address of the structure that contains the valid information.

struct statisticsStruct* initializeStatistics(struct statisticsStruct* statistics)

Description: Initializes the vision system statistics to zero. If a valid structure is not passed as an input to the function, it will allocate a new structure on the heap. Otherwise, the statistics are overwritten.

Input: struct statisticsStruct* statistics

Contains all of the vision system statistics.

Return Value: The address of the structure that contains the valid information.

void specifyInitialPositions (struct genesisStruct* genesis, struct trackingStruct* track)

Description: Allows for the specification of the initial ball and robot positions. It also assigns each robot on the field a number, which corresponds to the robot ID number. The robots and ball positions are assigned based on a single frame that is captured at the beginning of the game.

Inputs: struct genesisStruct* genesis

Contains the buffers for color segmentation and the grab buffers.

struct trackingStruct* track

Contains the tracking information for the robots and the ball.

Return Value: void

void initializeSingleProfile(struct profile* p, int number)

Description: Initializes an array of profile structures to be all zero.

Inputs: struct profile* p

Array containing profile structures to be initialized

int number

The number of elements in the array p

Return Value: void

void freeBlobAnalysis(struct blobAnalysisStruct* blobAnalysis)

Description: Frees all of the memory that is allocated for the blob analysis. These are the buffers and the threads. The entire blobAnalysis structure is then freed.

Inputs: struct blobAnalysisStruct* blobAnalysis

The structure that is to be deallocated.

Return Value: void

void freeGenesisBoard(struct genesisStruct* genesis)

Description: Frees all of the memory that is allocated for the genesis board. These are the buffers, the device, camera, and the threads. The entire genesis structure is then freed.

Inputs: struct genesisStruct* genesis

The structure that is to be deallocated.

Return Value: void

void freeAllStructures(struct genesisStruct* genesis, struct blobAnalysisStruct* blobAnalysis, struct trackingStruct* track, struct timingStruct* timing, struct statisticsStruct* statistics, struct calibrationData* calibData)

Description: Frees all of the dynamically allocated structures and the respective dynamically allocated pointers inside those structures.

Inputs: struct genesisStruct* genesis

Contains the genesis board device, camera, threads, and buffers.

struct blobAnalysisStruct* blobAnalysis

Contains the blob analysis buffers.

struct trackingStruct* tracking

Contains the tracking information.

struct timingStruct* timing

Contains the timing information.

struct statisticsStruct* statistics

Contains the vision system statistics.

struct calibrationData* calibData

Contains the calibration information.

Return Value: void

F.8 networkFunctions.c

Header File: udp_network.h

int sendData (struct netData* nds, struct localNetworkStruct* lns, int length)

Description: Sends the tracking information over the UDP network.

Inputs: struct netData* nds

Contains all the information relating to the port that is opened including the address and port number to send the information to.

struct localNetworkStruct* lns

The structure that will be sent across the network.

int length

Length of the network packet.

Return Value: The amount of data that could actually be transmitted.

int recvData (struct netData* nds, struct localNetworkStruct* lns, int length)

Description: Receives tracking data over the network. This function blocks until the data has been received and read from the buffer.

Inputs: struct netData* nds

Contains all the information relating to the port that is opened including the sending address and port number.

struct localNetworkStruct* lns

The structure that the tracking data will be read into.

int length

Length of the network packet that is to be received.

Return Value: The amount of data that could actually be received.

int recvRequest (struct netData* nds, char* buffer, int length)

Description: Wait for and read a request for the transmission of data over the network. This will block until the request comes in. If the data does not correspond to SUBMIT_REQUEST then the request is not serviced and an error is returned.

Inputs: struct netData* nds

Contains all the information relating to the port that is opened including the sending address and port number.

char* buffer

The piece of memory that the request will be read into.

int length

Length of the network packet that is to be received.

Return Value: The amount of data that could actually be received.

int sendRequest (struct netData* nds, char* buffer, int length)

Description: Send a request over the network for the transmission of new tracking data.

Inputs: struct netData* nds

Contains all the information relating to the port that is opened including the address and port number to send the information to.

char* buffer

The piece of memory that will be sent across the network.

int length

Length of the network packet to be transmitted.

Return Value: The amount of data that could actually be transmitted.

struct netData* initializeNetworkClient(struct netData* sns, unsigned short port)

Description: Initialize the network client data structure. This is the structure that contains all of the information that is needed for the artificial intelligence computers to use the local network. If a valid structure is not passed as an input to the function, it will allocate a new structure on the heap. Otherwise, it reinitializes the existing structure.

Inputs: struct netData* sns

Contains the addresses and port numbers that indicate where to get the data from.

unsigned short port

The port that the connection has to be made to on the vision computer.

Return Value: The address of the structure that contains the valid information.

struct netData* initializeNetworkServer(struct netData* cns, unsigned short port)

Description: Initializes the network server data structure. This is the structure that contains all of the information that is needed for the vision system to process and service requests over the local network. If a valid structure is not passed as an input to the function, it will allocate a new structure on the heap. Otherwise, it reinitializes the existing structure.

Inputs: struct netData* cns

Contains the address and port numbers that the request for information had come from.

unsigned short port

The port number that the socket binds to for waiting on requests.

Return Value: The address of the structure that contains the valid information.

void destroyNetwork(struct netData* cns)

Description: This closes the network port and frees the memory that was allocated during the initialization.

Inputs: struct netData* cns

Contains all the information for network communication.

Return Value: void

F.9 orientation.c

Header File: orientation.h

int orientBlobs (struct state* tRobotProfile, struct point* robot, int numRobot, struct point* orient, int numOrient)

Description: Registers orientation and team marker blobs together and computes the orientation angle between the two of them. If an orientation blob is not registered, then the orientation angle is set to NOT_ORIENTED. All orientation angles are in radians.

Inputs: struct state* tRobotProfile

The state of the supposed robot. This is the position and orientation.

struct point* robot

The array of team marker blobs.

int numRobot

The number of elements in the array robot.

struct point* orient

The array of orientation blobs.

int numOrient

The number of elements in the array orient.

Return Value: The number of team marker blobs that have orientation markers registered to them.

F.10 output.c

Header File: vision.h

void debugOutput (struct trackingStruct* track, struct calibrationData* calibData)

Description: Print the tracking information and profile for all of the objects currently being tracked to the console window.

Inputs: struct trackingStruct* track

Contains the tracking information and filter.

struct calibrationData* calibData

Contains the calibration information including how many robots should be on the field.

Return Value: void

void screenOutput (struct genesisStruct* genesis, struct trackingStruct* track, struct calibrationData* calibData, int bufferNumber)

Description: Outputs the current blobs which fall into each sub-volume that is being looked for to the genesis display section and can be viewed on the monitor.

Inputs: struct genesisStruct* genesis

Contains the buffers that correspond to the display section.

struct trackingStruct* track

Contains the tracking information that we wish to look at and the locations of the objects so the labels may be placed appropriately.

struct calibrationData* calibData

Contains the calibration information including the number of robots to be expected on the field.

int bufferNumber

The current buffer index that is marked as processing in the circular grab buffer.

Return Value: void

void imageCapture(struct genesisStruct* genesis, int mode)

Description: Captures a frame from the display section and saves it to disk as a TIFF image. Hitting the ‘9’ key on the number pad captures the image.

Inputs: struct genesisStruct* genesis

Contains the buffers that are to be captured to disk.

int mode

Whether the capture is done during the initialization or the processing stage.

Return Value: void

F.11 packetize.c

Header File: vision.h

void packetizeData (struct trackingStruct* track, struct calibrationData* calibData, struct localNetworkStruct* lns, HANDLE* mutex, int* dirty, int mode)

Description: Places the data that results from the tracking into the appropriate structure for transmission to the artificial intelligence computers. The type of data that is sent is dependent on the mode variable and the type of vision system build.

Inputs: struct trackingStruct* track

Contains the information that needs to be packetized.

struct calibrationData* calibData

Contains the calibration information including the number of robots to be expected on the field.

struct localNetworkStruct* lns

The structure that will be sent across the network.

HANDLE* mutex

The mutual exclusion lock that prevents the data from being read while new data is being written.

int* dirty

Flag that states whether the data in the localNetworkStruct is dirty of if it is new data.

int mode

The type of computer that the packet will be transmitted to.

Return Values: void

F.12 robot_filter.c

Header File: track.h

int trackRobotPositions (struct trackingStruct* track, int mode, struct calibrationData* calibData);

Description: Orient, identify, filter, and track the robot positions in image coordinates. Also filters the linear and rotational velocity. The barrel distortion and parallax error are also removed. If a robot is not located but is physically in the field, then the data is extrapolated from the previous frames. The position is clipped if the filter overshoots the frame size.

Inputs: struct trackingStruct* track

Contains all of the data for tracking, and filtering.

int mode

Specifies the data that needs to be tracked.

struct calibrationData* calibData

Contains all of the calibration information including the filter constants.

Return Value: Returns a ‘0’ always.

F.13 track.c

Header File: track.h

void getAllObjectPositions(struct trackingStruct* track, struct blobAnalysisStruct* blobAnalysis, struct calibrationData* calibData)

Description: Wrapper that performs all of the function calls for object tracking.

Inputs: struct trackingStruct* track

Contains all of the data for tracking and filtering.

struct blobAnalysisStruct* blobAnalysis

Contains the results of the blob analysis.

struct calibrationData* calibData

Contains the calibration information including the size if the windows and the minimum and maximum that a blob can be.

Return Value: void

void preprocessBlobResults(struct trackingStruct* track, IM_BLOB_GROUP1_ST* results, int sizeX)

Description: Size and shape filtering of the blobs to eliminate any spurious data points that may result in the image processing. The blobs are then processed from the common list of all blob features. The blobs are separated based on color.

Inputs: struct trackingStruct* track

Contains all of the data for tracking and filtering.

IM_BLOB_GROUP1_ST* results

The buffer that the results of the blob analysis will be copied into.

int sizeX

The width of the windowed grab buffer that only contains the field.

Return Value: void

void getRobotPositions(struct trackingStruct* track, struct calibrationData* calibData)

Description: Copies the locations of the opponent robots into the tracking structure until no more blobs are found or until there are not supposed to be more players on the field.

Inputs: struct trackingStruct* track

Contains the tracking information.

struct calibrationData* calibData

Contains the calibration information including the number of robots to be expected on the field.

Return Value: void

F.14 transform.c

Header File: transformation.h

struct profile transformCoord (struct profile inPoint, struct calibrationData* calibData)

Description: Transforms the coordinates from image coordinates to field coordinates.

Inputs: struct profile inPoint

The profile that is to be converted from image to field coordinates.

struct calibrationData* calibData

Contains the calibration information including the transformation from image to field coordinates.

Return Value: The profile in field coordinates.

void paralaxCorrect (struct point* inPoint, int object, int number, struct calibrationData* calibData)

Description: Removes the parallax error that results from objects of different heights. The error removal is dependent on the classification of the object in question.

Inputs: struct point* inPoint

Array of point structures that need the parallax error removed.

int object

Type of object in question.

int number

Number of elements in the array inPoint.

struct calibrationData* calibData

Contains the calibration information including the parallax scalar and offset.

Return Value: void

F.15 vision.c

Header File: vision.h

void main(int argc, char **argv)

Description: Entry point into the vision system code. This function calls all of the other functions and has the main vision processing loop.

Inputs: int argc

The number of elements in the array argv.

char** argv

Array of the arguments to the executable.

Return Value: The exit status of the program.

void errHandler(void *success)

Description: Error handler for the genesis function calls. Exits the program upon error in the genesis native library calls. Outputs the error to the console window.

Inputs: void* success

Flag to mark whether or not there was en error in the processing loop.

Return Value: void

void commThread(void)

Description: The thread that services tracking data requests over the network in competition mode.

Inputs: void

Return Value: void

void commBrazilThread(void)

Description: The thread that will service all requests for data from the Brazil artificial intelligence computer.

Inputs: void

Return Value: void

void commItalyThread(void)

Description: The thread that will service all requests for data from the Italy artificial intelligence computer.

Inputs: void

Return Value: void

void commDisplayThread(void)

Description: The thread that will service all requests for data from the display client computer(s).

Inputs: void

Return Value: void

void uiThread(void)

Description: The thread that services all user interface commands to the processing thread.

Inputs: void

Return Value: void