Appendix A

Machine Vision for Navigation

The need for sensing in any truly autonomous robot is ubiquitous. Among the various sensors that can be applied, one of the most powerful and inexpensive is that of machine vision. Color-based region segmentation, where objects in the world are identified by specific color (but not necessarily uniquely), has proved popular in robotics and automation, because color coding is a relatively unobtrusive modification of the environment. With the coding, balls, goals, obstacles, other robots, as well as other object can be detected. As the primary sensory process for many mobile robots, vision goes hand-in-hand with navigation. A system may use a camera on a robot to detect relative obstacles for navigation, or in the case of many small robots, a camera fixed in the world space to detect both robots and nearby obstacles.

Although popular as a sensor due to low hardware cost, vision has sometimes proved difficult due to high processing requirements and a large input stream to sift through in order to generate perceptual information for higher levels in the system. Thus the problem is that of mapping an input video stream to a perceptually more salient representation for other parts of the agent. The representation popular in hardware and domain-specific approaches to this problem is to segment the video stream into colored regions (representing all or part of a colored object). This is the representation we also choose, as it has proved successful in many applications [50].

A.1 CMVision: The Color Machine Vision Library

A.1.1 Color Image Segmentation

By far the most popular approach in real time machine vision processing has been color segmentation. It is currently popular due to the relative ease of defining special colors as markers or object labels for a domain, and has proved simpler than other methods such as the use of geometric patterns or barcodes. Among the many approaches taken in color segmentation, the most popular employ single-pixel classification into discrete classes. Among these, linear and constant thresholding are the most popular. Other alternatives include nearest neighbor classification and probability histograms.

Linear color thresholding works by partitioning the color space with linear boundaries (e.g. planes in 3-dimensional spaces). A particular pixel is then classified according to which partition it lies in. This method is convenient for learning representations such as artificial neural networks (ANNs) or multivariate decision trees (MDTs) [13].

A second approach is to use nearest neighbor classification. Typically several hundred preclassified exemplars are employed, each having a unique location in the color space and an associated classification. To classify a new pixel, a list of the K nearest exemplars are found, then the pixel is classified according to the largest proportion of classifications of the neighbors [15]. Both linear thresholding and nearest neighbor classification provide good results in terms of classification accuracy, but do not provide real-time performance using off-the-shelf hardware.

Another approach is to use a set of constant thresholds defining a color class as a rectangular block in the color space [50]. This approach offers good performance, but is unable to take advantage of potential dependencies between the color space dimensions.

A final related approach is to store a discretized version of the entire joint probability distribution. So, for example, to check whether a particular pixel is a member of the color class, its individual color components are used as indices to a multi-dimensional array. When the location is looked up in the array the returned value indicates probability of membership. This technique enables a modeling of arbitrary distribution volumes and membership can be checked with reasonable efficiency. The approach also enables the user to represent unusual membership volumes (e.g. cones or ellipsoids) and thus capture dependencies between the dimensions of the color space. The primary drawback to this approach is its associated high memory cost.

A.1.2 Color Spaces

The color space refers to the multidimensional space the describes the color at each discrete point, or pixel, in an image. The intensity of a black and white image is a segment of single dimensional space, where the value varies from its lowest black value to its highest at white. Color spaces generally occupy three spaces, although can be projected into more or fewer to yield other color representations. The common RGB color space consists of a triplet of red, green, and blue intensity values. Thus each color in the representation lies in a cube with black at the corner (0,0,0), and pure white at the value (1.0,1.0,1.0). Here we will describe the different color spaces we considered for our library, including RGB, a projection or RGB we call fractional YRGB, and the YUV color space used by the NTSC and PAL video standards, among other places.

In our choice of appropriate color spaces, we needed to balance what the hardware provides with what would be amenable to our threshold representation, and what seems to provide the best performance in practice. At first we considered RGB, which is a common format for image display and manipulation, and is provided directly by most video capture hardware. It's main problem lies in the intensity value of light and shadows being spread across all three parameters. This makes it difficult to separate intensity variance from color variance with a rectangular, axis aligned threshold. More complex threshold shapes alleviate this problem, but that was not possible in our implementation. An equally powerful technique is to find another color space or projection of one that is more appropriate to describe using rectangular thresholds.

This limitation lead us to explore a software transformed RGB color space we called fractional RGB. It involves separating the RGB color into four channels, intensity, red, green, blue. The color channels in this case are normalized by the intensity, and thus are fractions calculated using the following definition:

$$Y' = \tfrac{(R+G+B)}{3} \tag{A.1}$$
$$R' = \tfrac{R}{Y'} \tag{A.2}$$
$$G' = \tfrac{G}{Y'} \tag{A.3}$$
$$B' = \tfrac{B}{Y'} \tag{A.4}$$

The main drawback of this approach is of course the need to perform several integer divides or floating point multiplications per pixel. It did however prove to be a robust space for describing the colors with axis-aligned threshold cubes. It proves useful where RGB is the only available color space, and the extra processing power is available.

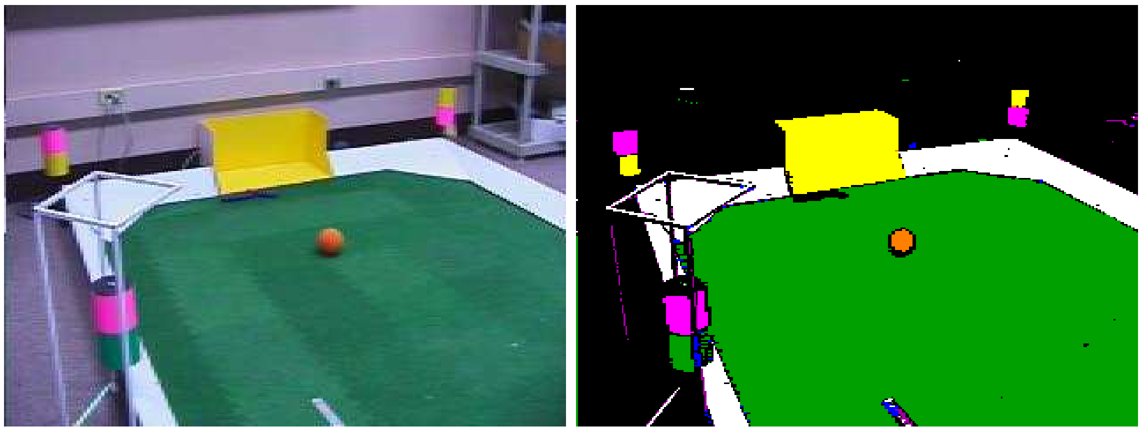


Figure A.1: A video image and its YUV color classified result

The final color space we tried was the YUV format, which consists of an intensity (Y) value, and two chrominance (color) values (U,V). It is used in video standards due to its closer match with human perception of color, and since it is the raw form of video, it is provided directly by most analog video capture devices. Since intensity is separated into its own separate parameter, the main cause of correlations between the color component values has been removed, and thus is a better representation for the rectangular thresholds. This is because the implementation requires axis aligned sides for the thresholds, which cannot model interactions among the component color values. Thus YUV proved to be robust in general, fast since it was provided directly by hardware, and a good match for required assumptions of component independence in our implementation. An example YUV histogram, with a threshold shown outlining a target yellow color is given in figure A.2

One color space we have not tried with our library is HSI, or hue, saturation, intensity. In its specification, hue is the angle on the color wheel, or dominant spectral frequency, saturation is the amount of color vs. neutral gray, and I is the intensity. Although easy for humans to reason in (hence its use in color pickers in painting programs), it offers little or no advantage over YUV, and introduces numerical complications and instabilities. Complications primarily arise from the angle wrapping around from 360 to 0, requiring thresholding operations work on a modular number values. More seriously, at low saturation values (black, gray, or white), the hue value becomes numerically unstable, making thresholds to describe these common colors unwieldy, and other calculations difficult [74]. Finally, HSI can be approximated by computing a polar coordinate version of the UV values in YUV. Since YUV is available directly from the hardware, it is simplest just to threshold in the pre-existing YUV space, thus avoiding the numerical problems HSI poses.

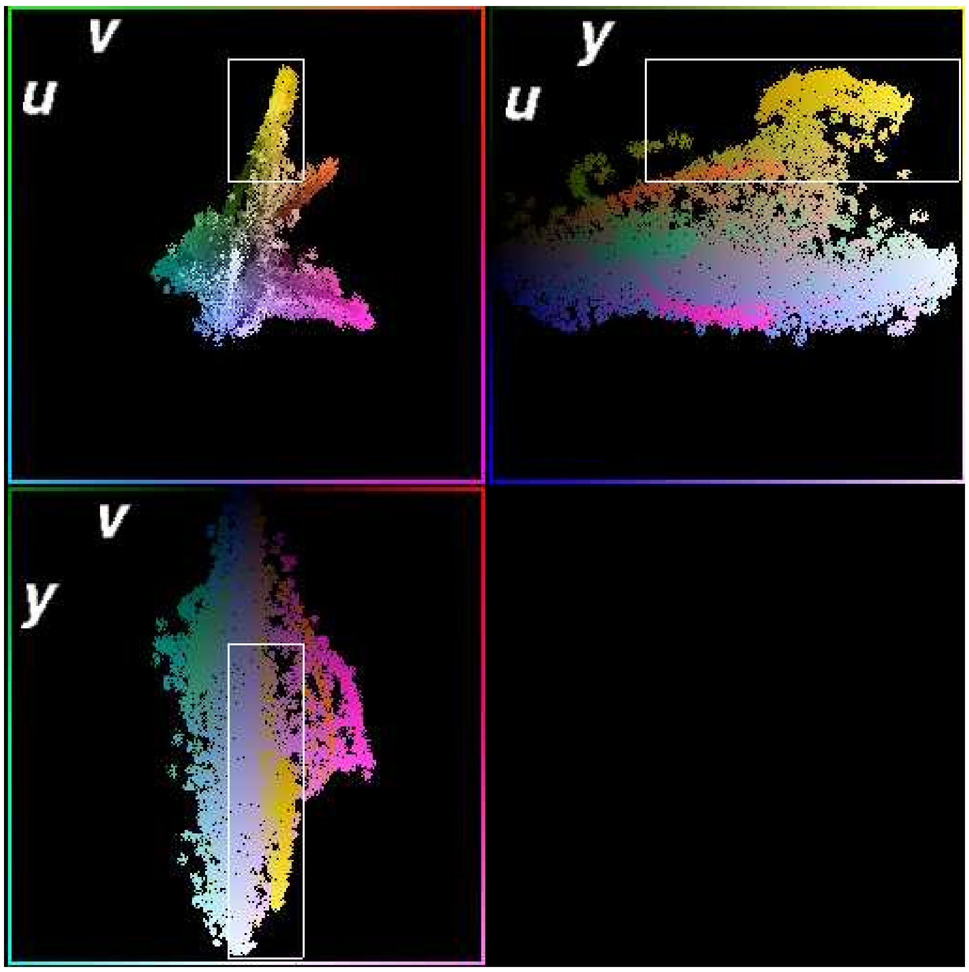


Figure A.2: A YUV histogram with a threshold defined

A.1.3 Thresholding

The thresholding method described here can be used with general multidimensional color spaces that have discrete component color levels, but the following discussion will describe only the YUV color space, since generalization of this example will be clear. In our approach, each color class is initially specified as a set of six threshold values: two for each dimension in the color space, after the transformation if one is being used. The mechanism used for thresholding is an important efficiency consideration because the thresholding operation must be repeated for each color at each pixel in the image. One way to check if a pixel is a member of a particular color class is to use a set of comparisons similar to

```
if ((Y >= Ylowerthresh)
   AND (Y <= Yupperthresh)
   AND (U >= Ulowerthresh)
   AND (U <= Uupperthresh)
   AND (V >= Vlowerthresh)
   AND (V <= Vupperthresh))
   pixel_color = color_class;
```

to determine if a pixel with values `Y, U, V` should be grouped in the color class. Unfortunately this approach is rather inefficient because, once compiled, it could require as many as 6 conditional branches to determine membership in one color class for each pixel. This can be especially inefficient on pipelined processors with speculative instruction execution.

Instead, our implementation uses a boolean valued decomposition of the multidimensional threshold. Such a region can be represented as the product of three functions, one along each of the axes in the space (Figure A.3). The decomposed representation is stored in arrays, with one array element for each value of a color component. Thus class membership can be computed as the bitwise `AND` of the elements of each array indicated by the color component values:

```
pixel_in_class = YClass[Y]
             AND UClass[U]
             AND VClass[V];
```

The resulting boolean value of `pixel_in_class` indicates whether the pixel belongs to the class or not. This approach allows the system to scale linearly with the number of pixels and color space dimensions, and can be implemented as a few array lookups per pixel. The operation is much faster than the naive approach because the the bitwise `AND` is a significantly lower cost operation than an integer compare on most modern processors.

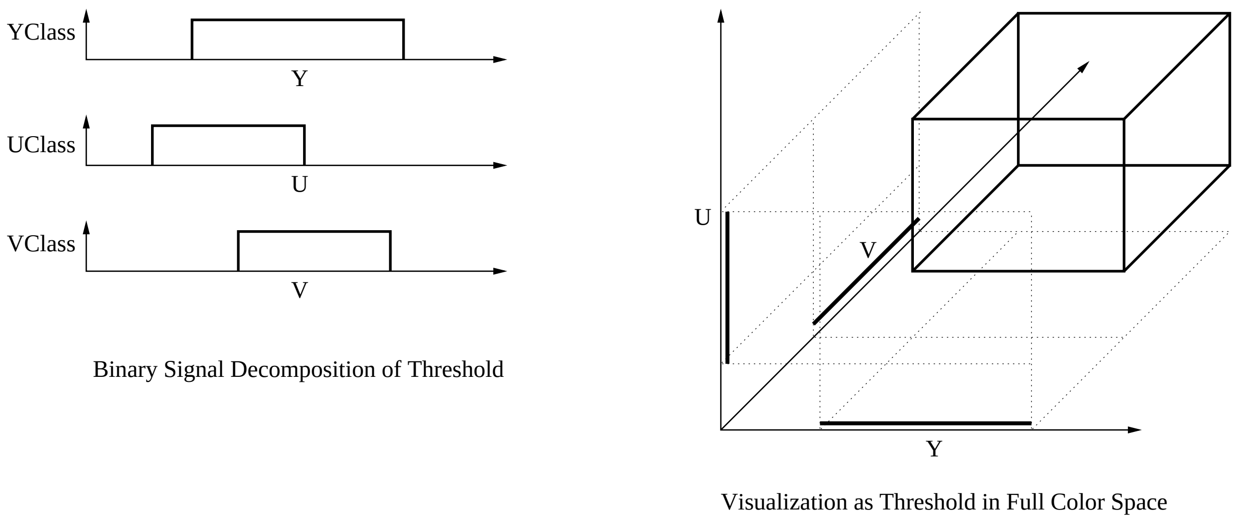

YClass
Y
UClass
U
VClass
V
Binary Signal Decomposition of Threshold
U
V
Y
Visualization as Threshold in Full Color Space

Figure A.3: A 3D region of the color space represented as a combination of 1D binary functions.

To illustrate the approach, consider the following example. Suppose we discretized the YUV color space to 10 levels in each each dimension. So "orange," for example might be represented by assigning the following values to the elements of each array:

```
YClass[] = {0,1,1,1,1,1,1,1,1,1};
UClass[] = {0,0,0,0,0,0,0,1,1,1};
VClass[] = {0,0,0,0,0,0,0,1,1,1};
```

Thus, to check if a pixel with color values `(1,8,9)` is a member of the color class "orange" all we need to do is evaluate the expression `YClass[1] AND UClass[8] AND VClass[9]`, which in this case would resolve to 1, or `true` indicating that color is in the class "orange."

One of the most significant advantages of our approach is that it can determine a pixel's membership in multiple color classes *simultaneously.* By exploiting parallelism in the bit-wise `AND` operation for integers we can determine membership in several classes at once. As an example, suppose the region of the color space occupied by "blue" pixels were represented as follows:

```
YClass[] = {0,1,1,1,1,1,1,1,1,1};
UClass[] = {1,1,1,0,0,0,0,0,0,0};
VClass[] = {0,0,0,1,1,1,0,0,0,0};
```

Rather than build a separate set of arrays for each color, we can combine the arrays using each bit position an array element to represent the corresponding values for each color. So, for example if each element in an array were a two-bit integer, we could combine the "orange" and "blue" representations as follows:

```
YClass[] = {00,11,11,11,11,11,11,11,11,11};
UClass[] = {01,01,01,00,00,00,00,10,10,10};
VClass[] = {00,00,00,01,01,01,00,10,10,10};
```

Where the first (high-order) bit in each element is used to represent "orange" and the second bit is used to represent "blue." Thus we can check whether `(1,8,9)` is in one of the two classes by evaluating the single expression `YClass[1] AND UClass[8] AND VClass[9]`. The result is `10`, indicating the color is in the "orange" class but not "blue."

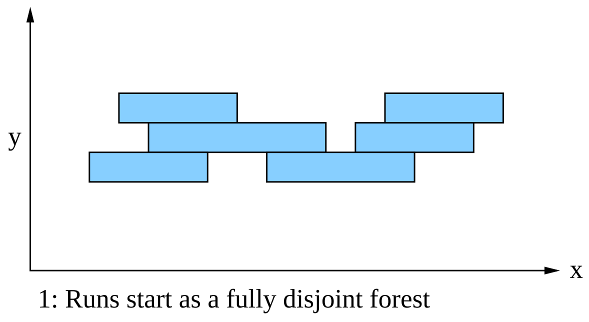
In our implementation, each array element is a 32-bit integer. It is therefore possible to evaluate membership in 32 distinct color classes at once with two `AND` operations. In contrast, the naive comparison approach could require 32×6, or up to 192 comparisons for the same operation. Additionally, due to the small size of the color class representation, the algorithm can take advantage of memory caching effects.

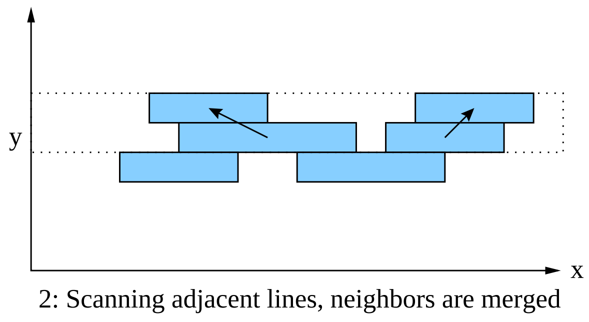
A.1.4 Connected Regions

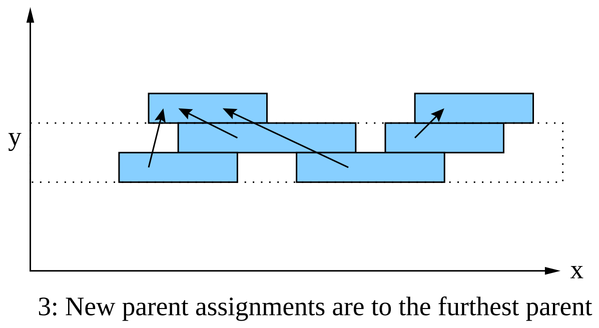
After the various color samples have been classified, connected regions are formed by examining the classified samples. This is typically an expensive operation that can severely impact real-time performance. Our connected components merging procedure is implemented in two stages for efficiency reasons.

The first stage is to compute a run length encoded (RLE) version for the classified image. In many robotic vision applications significant changes in adjacent image pixels are relatively infrequent. By grouping similar adjacent pixels as a single "run" we have an opportunity for efficiency because subsequent users of the data can operate on entire runs rather than individual pixels. There is also the practical benefit that region merging need now only look for vertical connectivity, because the horizontal components are merged in the transformation to the RLE image.

The merging method employs a tree-based *union find* with path compression. This offers performance that is not only good in practice but also provides a hard algorithmic bound that is for all practical purposes linear [82]. The merging is performed in place on the classified







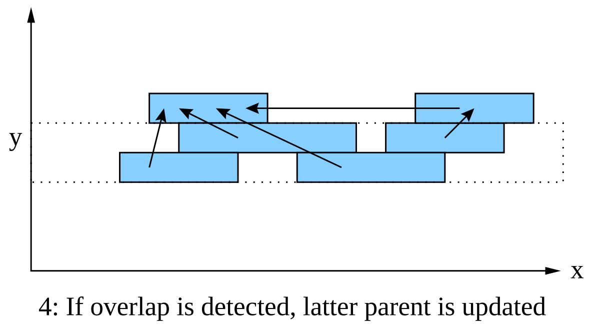


Figure A.4: An example of how runs are grouped into regions

RLE image. This is because each run contains a field with all the necessary information; an identifier indicating a run's parent element (the upper leftmost member of the region). Initially, each run labels itself as its parent, resulting in a completely disjoint forest. The merging procedure scans adjacent rows and merges runs which are of the same color class and overlap under four-connectedness. This results in a disjoint forest where the each run's parent pointer points upward toward the region's global parent. Thus a second pass is needed to compress all of the paths so that each run is labeled with its the actual parent. Now each set of runs pointing to a single parent uniquely identifies a connected region. The process is illustrated in Figure A.4).

A.1.5 Extracting Region Information

In the next step we extract region information from the merged RLE map. The bounding box, centroid, and size of the region are calculated incrementally in a single pass over the forest data structure. Because the algorithm is passing over the image a run at a time, and not processing a region at a time, the region labels are renumbered so that each region label is the index of a region structure in the region table. This facilitates a significantly faster lookup. A number of other statistics could also be gathered from the data structure, including the convex hull and edge points which could be useful for geometric model fitting.

After the statistics have been calculated, the regions are separated based on color into separate threaded linked lists in the region table. Finally, they are sorted by size so that high level processing algorithms can deal with the larger (and presumably more important) blobs and ignore relatively smaller ones which are most often the result of noise.

A.1.6 Density-Based Region Merging

In the final layer before data is passed back up to the client application, a top-down merging heuristic is applied that helps eliminate some of the errors generated in the bottom up region generation. The problem addressed here is best introduced with an example. If a detected region were to have a single line of misidentified pixels transecting it, the lower levels of the vision system would identify it as two separate regions rather than a single one. Thus a minimal change in the initial input can yield vastly differing results.

One solution in this case is to employ a sort of grouping heuristic, where similar objects near each other are considered a single object rather than distinct ones. Since the region statistics include both the area and the bounding box, a density measure can be obtained.

The merging heuristic is operationalized as merging pairs of regions, which if merged would have a density is above a threshold set individually for each color. Thus the amount of "grouping force" can be varied depending on what is appropriate for objects of a particular color. In the example above, the area separating the two regions is small, so the density would still be high when the regions are merged, thus it is likely that they would be above the threshold and would be grouped together as a individual region.

A.1.7 Performance

CMVision was tested at several resolutions using pre-captured data from an overhead camera. For the test setup, a 60-image sequence was taken at 3 resolutions, up to a maximum of 640x240 (interlaced NTSC video), and stored in a memory buffer so that the library performance could be isolated from image capture overhead. The YUV colorspace colorspace (native NTSC YCrCb422 format) was used for image capture and thresholding. The thresholding, connected components, region extraction, and region sorting methods were run (density-based region merging was disabled) were run 1000 times to generate averaged timings. The tests were run on a 2.4 GHz Athlon computer, and compiled using g++ with optimization enabled. The results are shown in Table A.1.7. As can be seen, CMVision can run quickly on a modern computer, leaving plenty of processing time for other tasks. Multiple 60 Hz video sequences can be processed on a single machine, and are limited by the bandwidth between the capture cards and main memory[1] instead of the vision processing itself. Alternatively, it can be applied on lower-speed embedded processors such as those likely to be found onboard small robots.

Frame Size (w x h)	Processing Time (ms)	Throughput (frames/sec)
640x240	0.968333	1032.70
320x240	0.498148	2007.44
160x120	0.126860	7882.71

Table A.1: Performance of the CMVision low-level color vision library at various resolutions

[1]Bandwidth for a single steam of digitized NTSC video can reach 18 MB/s without counting overhead. The maximum theoretical bandwidth of a PCI bus is 127 MB/s, although typically much less can be achieved in practice with multiple PCI devices.

A.1.8 Summary of the CMVision Library

The CMVision library is a system to accelerate low level segmentation and tracking using color machine vision. In creating it, we evaluated the properties of alternative approaches, choosing color thresholding as a segmentation method, and YUV or fractional YRGB as robust color spaces for thresholding. The created system can perform bounded computation, full frame processing at camera frame rates. The system can track up to 32 colors at resolutions up to 640x480 and rates at 30 or 60Hz without specialized hardware. Thus the primary contribution of this system is that it is a software-only approach implemented on general purpose, inexpensive, hardware. This provides a significant advantage over more expensive hardware-only solutions, or other, slower software approaches. The approach is intended primarily to accelerate low level vision for use in real-time applications where hardware acceleration is either too expensive or unavailable.

Building on this lower level, CMVision has been applied to several applications, including Carnegie Mellon's RoboCup small-size league team entries (since 2000), as well as the Sony legged (Aibo) league entries (1999-2005). As applied in the CMDragons system, CMVision has been used to track robots one two cameras at 60Hz on a single computer, while leaving sufficient processing time available for robot behaviors and motion planning. The next section describes the pattern detection system built for the CMDragons robot system.

A.2 Pattern Detection

Fast pattern detection and identification is a fundamental problem for many applications of real-time vision systems. The desirable characteristics for a solution are that it requires little computation, localizes a pattern robustly and with high accuracy, and can identify a large number of unique pattern identifiers so that many of these markers can be tracked within a field a view. We will present a system that can accurately track a broad class of patterns both accurately and quickly, when used with a suitable low level vision system that can return calibrated coordinates of regions in a image. Both pattern design and the detection algorithm are considered together to find a solution meeting the above criteria. Along the way, assumptions are verified to make informed choices without relying on guesswork, and allowing similar systems to be designed on a solid experimental and statistical basis.

Object identification and tracking is one of the most important current applications of machine vision. Much work has focused on object detection and tracking for complex or variable objects, such as faces, cars, and doors. While much progress has been made, many of the algorithms require substantial amounts of processing and are less accurate than can be achieved with patterns specifically designed for detection. Thus many current applications of vision-based object detection and tracking use customized patterns, such as in automated part placement or package routing systems. Although the use of customized patterns prevents the system from being usable in every environment, in many cases requiring a pattern to be used is not a major limitation. Of course, if the pattern can be specified in order to suit the capabilities and limitations of the machine vision system and detection algorithm, in return we expect very high performance from that system. Specifically, the detection for the pattern should be fast and highly accurate; especially when compared to more general object detection and tracking systems. We will describe such a system in the following sections. For low level vision, we will employ the freely available [17] CMVision [18] library. It performs color segmentation and connected components analysis to return colored regions at real time rates without special hardware or dedicating the entire CPU to the task.

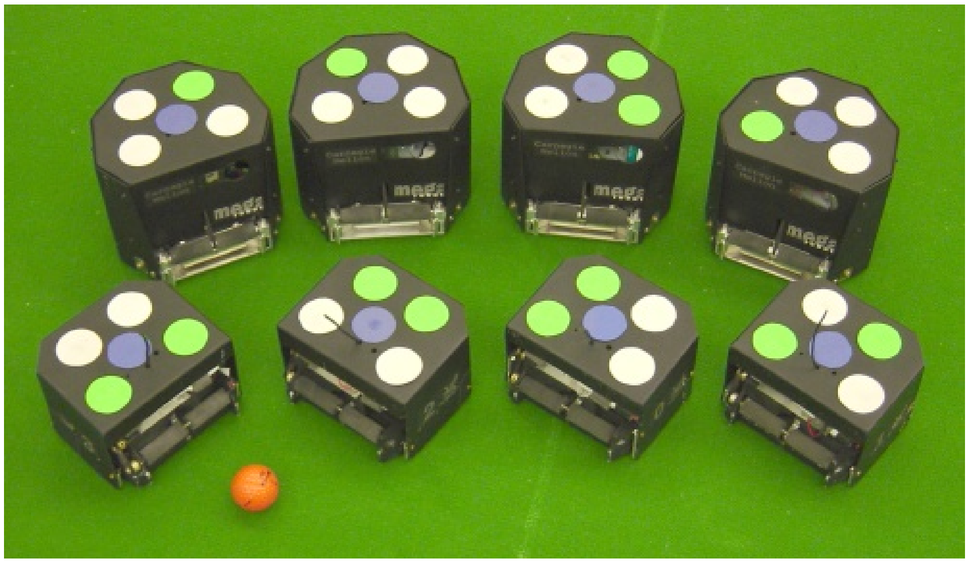


Figure A.5: The CMDragons'02 Robots. More recent robots use use the same marker pattern.

The environment in which most of this work has been done is the RoboCup [55] F180 "Small Size" League, where robots up to 18cm in diameter play soccer on a 2.8m by 2.3m carpeted soccer field. The game is played with two teams with five robots each, and uses an orange golf ball for the ball. One team must have a 40mm blue colored circle centered on the top of its robot while the other team must have a 40mm yellow patch. Teams may add extra patches and colors to the top of their robot to aid in tracking, so long as those colors are differentiable from the three standard colors (orange golf ball, yellow and blue team patches). The robots from our team, CMDragons'02 [22], can be seen in Figure A.5. Each team can control its robots either onboard, or offboard, and cameras are allowed to be placed above the field. Thus, most teams use a single overhead camera, with an offboard PC interpreting the camera signal and sending commands to the robots via a radio link.

This environment thus poses a tracking problem for up to 11 small objects in known planes (in this case the possibly different, but known, heights above a ground plane). Few other environments currently demand accurate, multiple pattern detection at very high speed, but one such environment is Virtual or Augmented Reality. For these environments, patterns are tracked in order to localize head mounted displays and locate objects in the physical environment that are mapped into the virtual environment [26, 27]. Detection must be fast and accurate to minimize observable lag and jitter in the visualization. Due to work in these two environments, fast, accurate, multiple pattern detection has become better understood and more practical. Thus we expect many more applications for such tracking systems in the future.

Another aspect that makes the RoboCup F180 environment challenging is the high speeds of the tracked objects, since the robots move quite quickly relative to their size. Robot speeds peaking in excess of 2m/s are not uncommon, and ball speeds (via robot kicking mechanisms) can reach up to 5m/s. Thus we feel this is a good testbed for a tracking system. Two other vision tracking and identification systems for the F180 league have been described in [78] and [45]. Each describes a working system used by a team, but neither motivates the choice of pattern by a thorough analysis of the underlying feature error, or attempts to generalize detection to other similar patterns in order to compare their performance. In this paper we will outline the choices and trade-offs made in designing an identification and tracking pattern by gathering real and simulated data at each step so that informed tradeoffs can be made. We hope that this will help others to implement similar high performance tracking systems both within RoboCup and in many other environments where a similar problem exists. Such designs should not have to rely on any guesswork.

In the first section, we will motivate the type of patches chosen from which to build patterns, and examine their error distributions when viewed from a camera. In the second section, we will describe several common patterns and a broad class that includes most of the patterns. We will motivate the use of this most popular class for its simplicity and accuracy, and for which an efficient generic detection algorithm can be created. In the following section, the performance of several such patterns will be examined in simulation. Finally the best performing pattern from simulation will be evaluated on a real-time vision system.

A.2.1 Single Patches

In order to build up a detection pattern, we must have some simpler building block on which to build. We will use simple colored patches whose position can be calculated accurately. To detect orientation, multiple patches can be employed. For single patches, the simplest design to detect is to use a regular geometric shape of a single color. Detection in the vision system can be carried out on a binary or multiclass threshold image from which connected regions of common color class can be extracted. This approach is common, and is known to be quite efficient, so this is the approach we will use. The next variable to determine is patch shape. We chose circles, because they guarantee rotational invariance, and analytical corrections for the projective distortions of their image centroids are known [44]. In addition, they are compact, minimizing the length of the border with other regions, where thresholding is most difficult. In experiments, other regular shapes such as squares, hexagons, and octagons, perform roughly on par with circles. However, they do not offer any benefits to motivate their use in light of the analytical guarantees for circles.

The more difficult parameter to determine is what size of patch to use. When the dimensions of the overall pattern are known, this still leaves the question of whether it is better to have a pattern with a few large patches, or more patches where each is of a smaller size. To address this, we created a test setup where a small moving platform would carry three different sized white patches 2 meters across the field of view of a camera looking down from 3 meters. The platform moved at a slow constant speed (about 23mm/sec) allowing large amounts of data to be gathered from a variety of locations across the field. Using this setup, we gathered positional data at 30 samples/sec for 40mm, 50mm, and 60mm circles. A total of 5 runs were gathered, each one having about 2570 data points. As a convention, we labelled the dimension along the primary direction of travel as x, and the dimension perpendicular to the direction of travel as y.

Although there is no ground truth from which to measure true error, the error can be estimated by smoothing the data with a large Gaussian kernel ($\sigma = 10$) and then comparing single samples with the smoothed version of the signal. The aggregate errors appear to follow a Gaussian distributions quite well, as can be seen in Figure A.6. However, more outliers occurred than would be expected in a pure Gaussian distribution, and the variance seemed to change noticeably between runs, and even varied over different segments in the course of a single run. The most surprising result however, is that the size of the patch had very little effect. The overall standard deviations were around 0.52mm in both x and y for all sizes with only slight (although significant) variation. The cumulative distributions of absolute error in x and y are shown in Figure A.7.

The estimated standard deviations for each patch size can be found in Table A.2, along

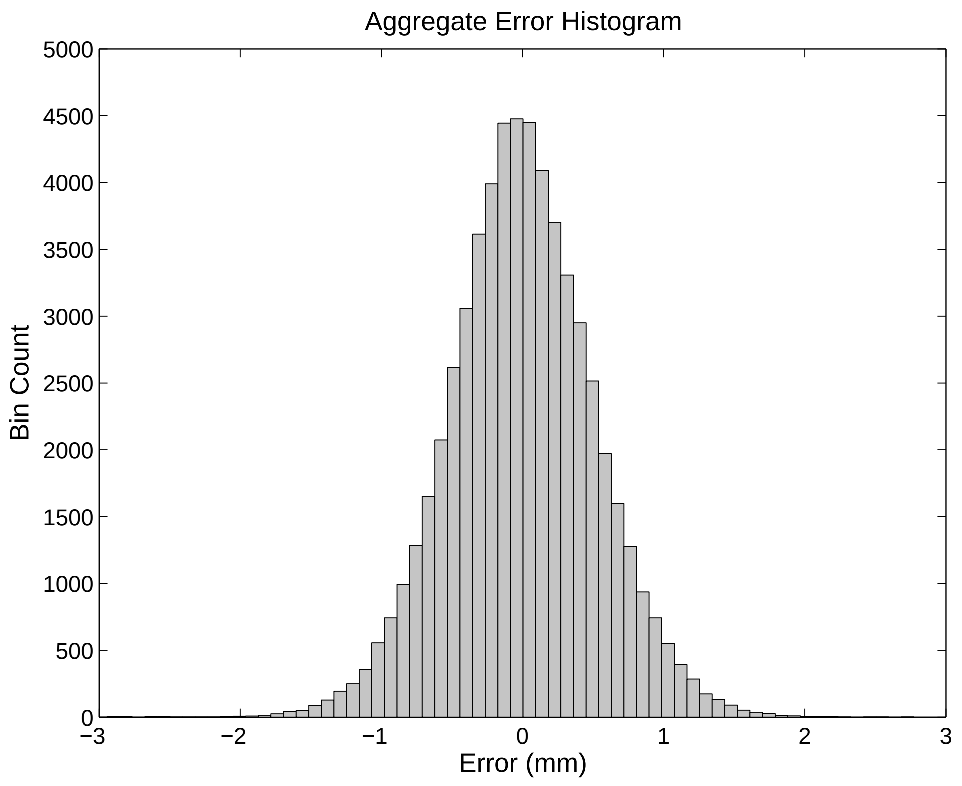
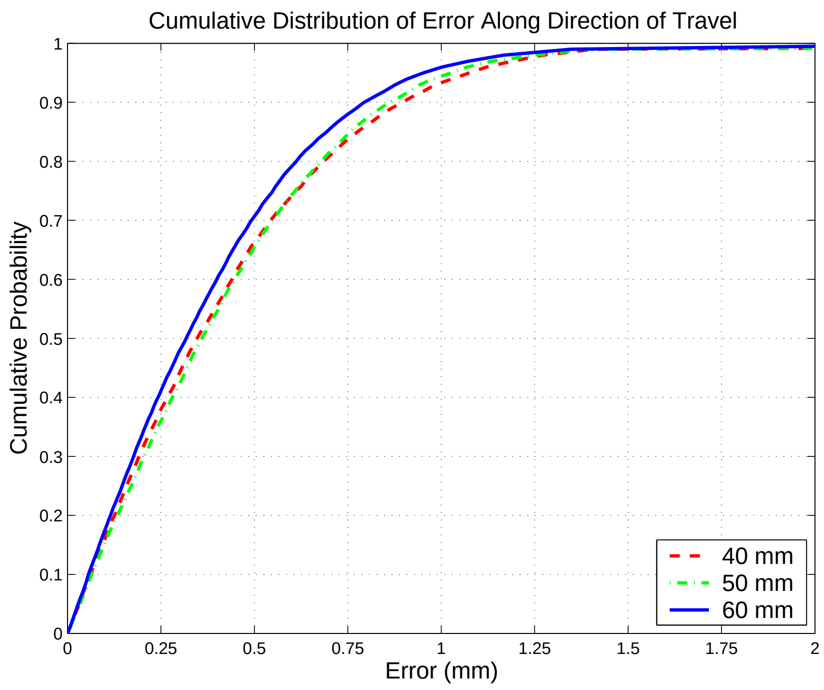


Figure A.6: Aggregate error distribution for all samples including all three patch diameters. This is actual data collected from the vision system.



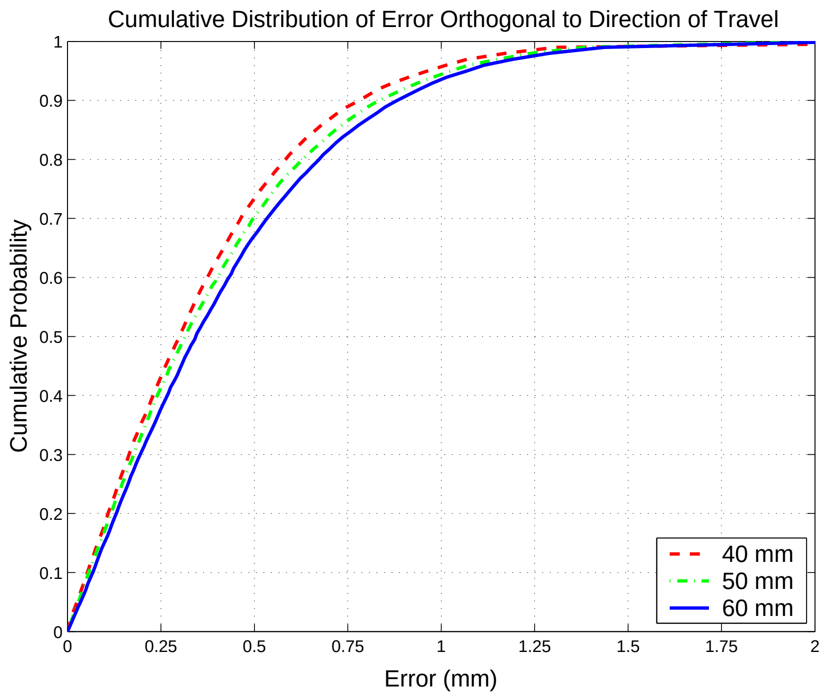


Figure A.7: Cumulative distributions of absolute error. Note that patch size does not have a large effect on error. Along the direction of travel, the largest patch size decreases error somewhat, but does worse than the smallest patch size perpendicular to the direction of travel.

Table A.2: The estimated standard deviations and 95% confidence intervals by patch size.

Diameter	σ_x	σ_y
40 mm	0.553, [0.546 – 0.561]	0.473, [0.467 – 0.480]
50 mm	0.543, [0.535 – 0.550]	0.504, [0.497 – 0.511]
60 mm	0.489, [0.482 – 0.496]	0.533, [0.526 – 0.541]

with 95% confidence intervals for the standard deviation. For hypothesis testing, we used the non-parametric Wilcoxon signed-rank test due to its robustness to outliers and lack of strong assumptions about the distributions being tested. The significant results (in all cases, $p < 0.0001$) were that along the direction of travel (x), the 60mm diameter circle had significantly less error than both the 50mm and 40mm patches. Perpendicular to travel (y) however, error *increased* with patch size, with all means being significantly different. Combined error in x and y indicated that the 50mm patch was slightly worse than the other two, with $p = 0.02$ against each of the other patches. Given the number of data points (over 10,000) however, we do not consider a difference at $p = 0.02$ ultimately conclusive. In addition, the difference in error from best to worst is less than 5%, which is much less variation than expected since the largest patch has 2.25 times the area of the smallest patch.

Thus the conclusion we can draw are that patches should be large enough they can be detected reliably, but need not be made any larger for purposes of accuracy. This is important in that it is contrary to conventional wisdom about region detection. It seems that other factors, such as quantization at edges due to pixels, play a larger part in determining the error than the area of the region. As we will show later, we can do somewhat better by adding more patches rather than using fewer patches and increasing their size.

A.2.2 Patterns and Detection

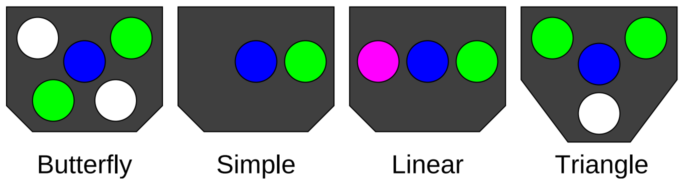


Figure A.8: Examples of common tracking patterns from the RoboCup F180 environment.

Now that the basis for choosing patches has been established, we can use this knowledge to evaluate tracking patterns. One source for many different ideas are the various patterns used by the over 20 teams in the RoboCup F180 League. The rules for the patterns on the top of the robots in that league has naturally led to many tracking and identification patterns being tried. Examples of some of the more popular designs can be seen in Figure A.8. The approaches taken thus far generally fall into one of three broad categories. By far the most common type is like that shown above, which we call *patch based*, where in addition to the team marker patch in the center, one or more additional circular or rectangular patches are used to encode position and orientation. Patch based systems have the advantage that position and orientation detection can combine several features (patches in this case) with sub-pixel accuracy. One alternative to this is to have a key patch marking the center of the pattern, surrounded by radial "pie slices" of two or more colors. Orientation and identification can be performed by scanning at some constant radius from the central patch [45]. Unfortunately, by depending on features (color edges in this case) that are difficult to quickly detect with sub-pixel accuracy, it is difficult to get very accurate orientation using this method. Another type of tracking pattern that has been tried is to use a central patch with an nearby line feature. By finding the edge points of that feature, a least squares fit can be made to get an accurate orientation measurement. Unfortunately, both these classes of patterns do not offer a straightforward way to improve the position estimate. For a given pattern size, we would like to make the most of the space. Ideally, we would like all of the patches to contribute to position, orientation, and identification detection. In this regard, patch based systems tend to do quite well, which has led to them becoming the most common class of pattern used for tracking.

If we look again at Figure A.8, we can notice similarities that can aid in creating a generic detection algorithm. All are keyed by a colored patch (in the center of the pattern) indicating the presence of a pattern. Each patch occurs at some unambiguous angle radially from the key patch. Thus there is a distinct circular ordering of the non-key patches that can be calculated even in the presence of moderate noise. This means a generic detection algorithm can start by searching for the key patch, and then detecting and sorting the additional patches radially. All that needs to be done after that is to find the rotational correspondence of the additional patches with the geometric model of the pattern. In the case of the Simple pattern, the correspondence is trivial. For the Linear and Triangle patterns, the colors of patches can be used to disambiguate geometrically similar correspondences. Finally, in the case of the Butterfly pattern, geometric asymmetry can be used to find the rotational correspondence, assuming the distances between patches can be measured accurately enough. As shown in the previous section, this boils down to the difference of two Gaussians. When we consider the standard deviations determined in the previous section, distances that differ by over 10mm should be differentiated correctly with very high certainty. Using geometric asymmetry offers the benefit of freeing up the patch colors to encode only identification, rather that both identification and rotational correspondence. This gives the butterfly pattern (or any other geometrically asymmetric pattern) an identification advantage over symmetric patterns, as shown in Table A.3.

Table A.3: The number of uniquely identifiable patterns that can be detected using a certain number of colors (excluding the key patch and key patch color)

Pattern	2 colors	3 colors	4 colors	n colors
Butterfly	16	64	256	4^n
Simple	2	3	4	n
Linear	1	3	6	$n \cdot (n-1)/2$
Triangle	2	6	12	$n \cdot (n-1)$

Once the correspondence is established, the position and orientation estimates must be made. What we would like is to get near optimal detection but without resorting to iterative methods or other time consuming operations. Here we take a simple approach that turns out to be not only fast but in practice nearly indistinguishable from optimal formulations. First, the mean location of the patches is determined. This is an optimal estimate, although for many patterns this location is offset from the actual location we want to report (so it is not an optimal estimator of that point). After this mean position has been determined, we get displacement vectors between a pre-specified set of "orientation pairs" from the patches. These pairs should be well separated (because error decreases with distance), and different pairs that share a patch should be as orthogonal as possible (to avoid correlated errors). For the butterfly pattern, we use vectors between the four non-key patches. For the Triangle pattern we Similarly take the triangle edges formed by the pattern's three external patches. For the Linear and Simple pattern, only one nearly orthogonal pair exists. For the Linear pattern we choose the longest option of the opposite patches because this will minimize the error compared to the two shorter vectors that include the central key-patch. After the separation vectors are determined, they can be rotated into a consistent frame of reference because their angle relative to forward is known from the model of the pattern. Once all the vectors are lined up by the model, they can be added to form a single vector, and the arctangent calculated to get the angle measurement.

The motivation for adding the vectors comes from the observation that the angular error of a vector is roughly proportional to the separation of the patches when the separation distance (d) is much larger than the positional standard deviation (σ), or:

$$\sigma_\theta \approx \frac{\sqrt{2\sigma^2}}{d}$$

The term $\sqrt{2\sigma^2}$ comes from the subtraction of two Gaussians (since the separation distance is large this is roughly a 1D subtraction). The division by d is the result of arctangent being linear near the origin. In practice, we've found this approximation works well when $d > 10\sigma$. Finally, once the angle estimate is made, we can use this to project the mean of the patches to the coordinates of the patch that are to be reported (normally the origin of the patch model coordinate system).

For comparison, we also derived an iterative Maximum Likelihood (ML) estimation method that co-optimizes position and angle estimates assuming Gaussian positional error for the patches. Its full derivation is omitted here for brevity. First, it is a well known fact that minimizing sum-squared-error in 1D is identical to maximizing the log likelihood (and thus likelihood) of samples from a Gaussian error distribution. Since in the 2D case variances can be added, this correspondence carries over into the 2D case. Thus by minimizing the sum-squared-error of the measured position of patches from their model positions given the estimated pattern position and orientation, we can obtain an ML estimate. So for a pattern with n patches, we define the current estimate of robot position as r, marker locations as $v_i \dots v_n$, and patch locations from the pattern model as $p_i \dots p_n$. If we let $s = \sin(r_\theta)$ and $c = \cos(r_\theta)$, then we have following derivatives for sum-squared-error E:

$$\begin{aligned} x_i &= r_{ix} + cp_{ix} - sp_{iy} - v_{ix} \\ y_i &= r_{iy} + sp_{ix} + cp_{iy} - v_{iy} \end{aligned}$$

$$\begin{aligned} \frac{\partial E}{\partial r_{ix}} &= \sum_{i=1..n} 2x_i \\ \frac{\partial E}{\partial r_{iy}} &= \sum_{i=1..n} 2y_i \\ \frac{\partial E}{\partial r_\theta} &= \sum_{i=1..n} 2x_i \cdot (-sp_{ix} - cp_{iy}) + 2y_i \cdot (cp_{ix} - sp_{iy}) \end{aligned}$$

These partial derivatives, along with the obvious implementation of the error function itself, can be used to create an iterative ML estimation method using Newton's method.

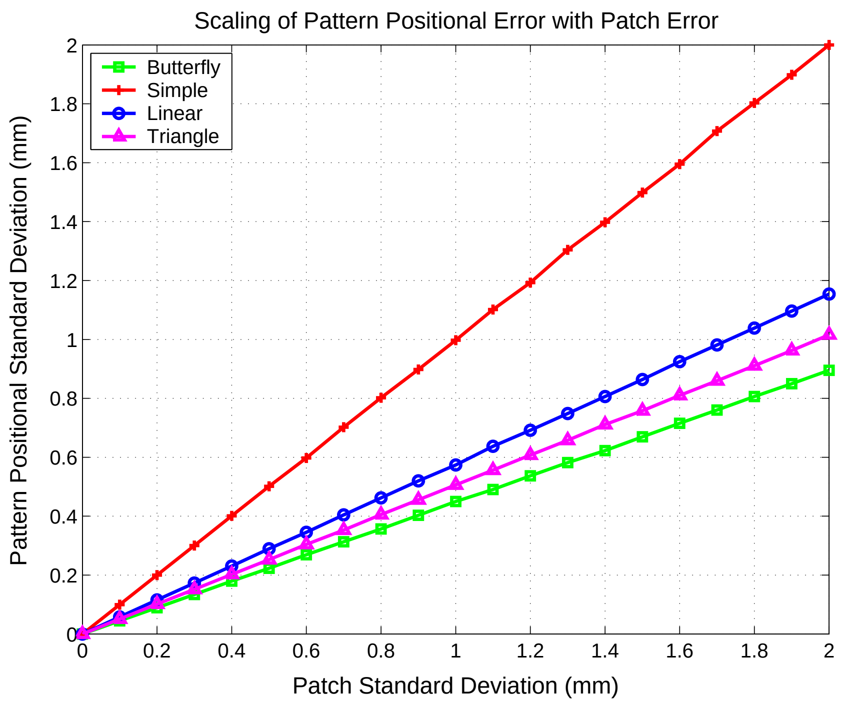
A.2.3 Pattern Comparison

In order to evaluate the patch-based patterns introduced earlier, a small simulator was created that would generate patch positions using a Gaussian error model for a pattern at random positions and orientations. Then the detection algorithm was run on the patches, and the resulting position and orientation measurements compared to the true values used to generate the input patches. The results for the simulation are shown in Figure A.9, plotted as pattern position and orientation standard deviation vs. input patch positional standard deviation. Each data point was generated from 100,000 simulated detections. One can easily see that multi-patch patterns have a distinct advantage for both position and angle measurements. The Simple pattern can fair no better than a single patch using the generic detection algorithm described in the previous section. It could perhaps benefit more from maximum likelihood detection, but this would make detecting the Simple pattern slower than detecting the more complicated patterns. The Butterfly pattern has the most accurate position estimation, followed by Triangle and Linear at somewhat decreased accuracy. For angular error, the Butterfly pattern again shows the lowest error, with Triangle close behind. With their multiple patches allowing several well separated orientation pairs to be used, they both perform much better than Linear or Simple, each of which only have a single orientation pair. Linear fares better than the Simple pattern because the separation distance for its orientation pair is twice that of the Simple pattern. Another dimension along which we might compare is execution time. The simple pattern's position could be identified the fastest, taking $0.5\mu s$ on a 900MHz Athlon, while the Butterfly pattern required $2.67\mu s$. As expected, the time was nearly proportional to the number of patches and orientation pairs. However, these times negligible compared to the roughly $1000\mu s$ it takes to threshold and segment a frame of video.

Since the Butterfly pattern worked best for both positional and angular error in simulation, we decided to make further tests to evaluate its performance using the iterative maximum likelihood detection and then measure the pattern's performance on a real vision system. To compare our generic detection algorithm with the ML estimate, we ran each of the detection methods in simulation. Even after 100,000 samples, no statistically significant differences could be detected with a Kolmogorov-Smirnov test, leading to the conclusion that at least for complicated patterns, the simple detection algorithm was indistinguishable in terms of accuracy from the ML estimation.

Finally, we evaluated the Butterfly pattern on a real vision system. We ran 5 runs similar to those for single patches but this time with a pattern being tracked rather than individual patches. The overall standard deviations were $\sigma_x = 0.3766$mm, $\sigma_y = 0.3432$mm, and $\sigma_\theta = 0.0070$rad. This was significantly better than a single patch ($p < 0.0001$), although not as low as predicted by simulation. The error was about 70% higher than predicted, which is most likely explained by some correlation in the patches' error (such as error from camera jitter). The pattern error was more consistent with a patch standard deviation of around 0.8mm, and thus could also have been due to outliers affecting the measurements.

Another possible problem (but one that is easily measurable) is correlation of errors over time. Typically filters assume that all readings are independent measurements, however this may not be the case for some sources of error. In Figure A.10, we show 2D scatter plots of adjacent readings for a single patch (top) and for a full pattern (bottom). The single patch shows structure indicating that errors are likely to repeat (the center diagonal stripe) or jump up or



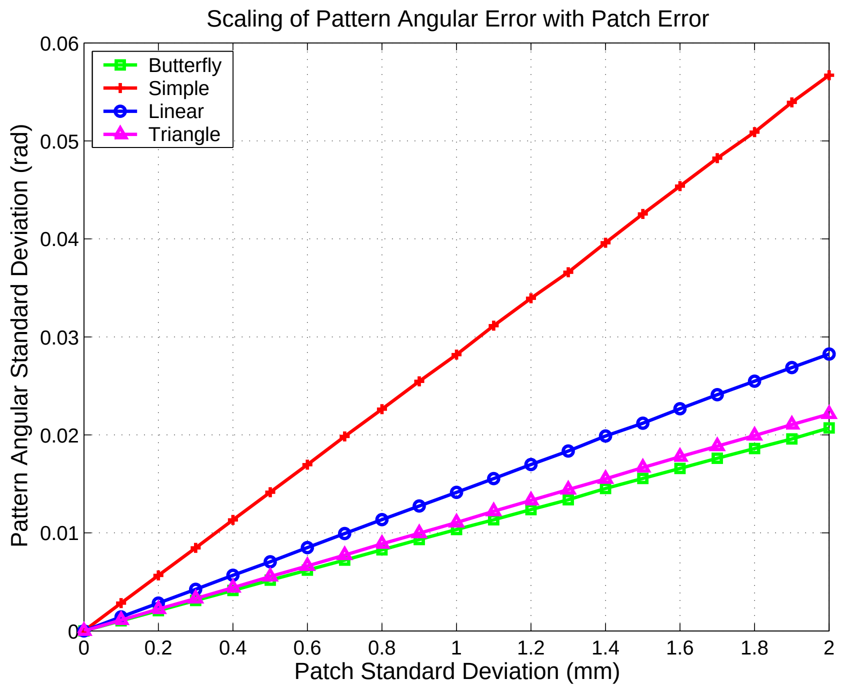
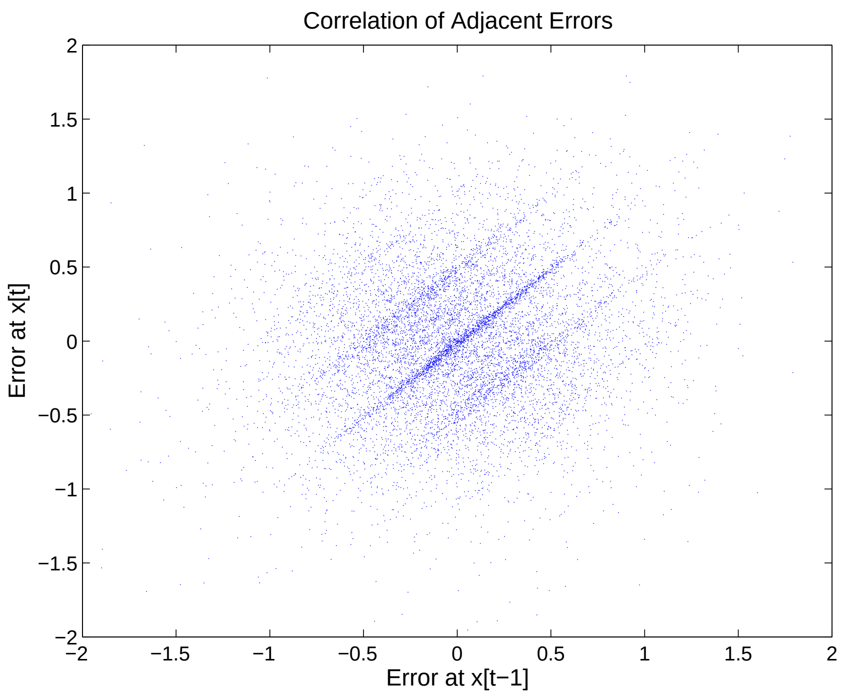


Figure A.9: Comparison of the positional and angular error of different patterns as the error of individual patches vary. For relatively small patch standard deviations, a linear relationship exists between the two, although the number of patches and layout of the pattern vary the factor.

down by a fixed amount (the upper and lower diagonal stripes). As best we could determine, this appears to be due to the binary color segmentation; As the patch moves across the camera image, pixels switch from background color to patch color (and back to background) abruptly, changing the location of the centroid by fixed amounts. The full pattern (bottom) does not display this structure (most likely because combining 5 patches made the structured error of individual patches small enough to make it unnoticeable). However the plot is still not an unbiased circular cloud, so adjacent readings are still somewhat correlated. With a few time steps separating readings, no observable correlations are present. Thus, assuming measurements are independent for patterns seems to be a reasonable simplification, although increasing the standard deviation to a more conservative estimate may be prudent. Assuming independence for patches may be more problematic, so for tracking single patches the extra complexity of modeling error correlation may be necessary.

A.2.4 Summary of Pattern Vision

This section presents the derivation of an efficient and highly accurate detection algorithm along with an analysis of the performance of many different patch-based patterns. It is designed to be paired with a low-level vision system (such as CMVision) to construct a global vision system for a multi-robot system. We first look at the performance of single patch detection, noting that size, although important for robust detection, does not have a large effect on the accuracy of the positional measurement of a patch. We presented a fast patch based detection algorithm along with an iterative ML variant, which perform similarly in terms of accuracy. We compared several patterns in simulation to find out how accurate their detection scaled with the error of the patches form which they were made. We then tested a pattern on a real vision system with positive results, and examined the assumption of independence on which higher levels of an object tracking system rely. In particular, the data supports the common assumption of objects with additive Gaussian noise on position, and provides measurements to guide the design of error tolerance into decision and navigation systems for robot agents.



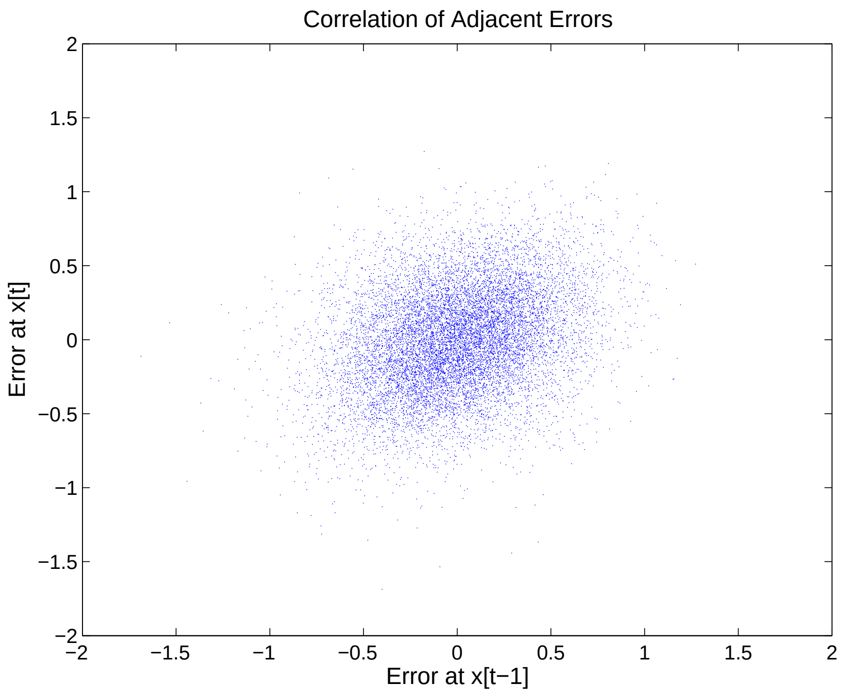


Figure A.10: 2D scatter plots of adjacent readings for single patches (top) and for a full Butterfly pattern (bottom). Uncorrelated readings would show up as a circular 2D Gaussian cloud. Note the structure in the plot for a single patch, and the unstructured but non-circular distribution for the full pattern.