

UG3 Introduction to Vision and Robotics

Vision Assignment

Clemens Wolff, Toms Bergmanis

March 7th, 2013

1 Introduction

The goal of this report is to propose an algorithm for object tracking. Object tracking, in general, is a challenging problem. Difficulties in tracking objects can arise due to abrupt object motion or changing appearance patterns of both the object and its surroundings and other factors. Tracking is usually performed in the context of higher-level applications that require the location and/or shape of the object in every frame. Typically, assumptions are made to constrain the tracking problem in the context of a particular application[1].

2 Methods

Several simplifying assumptions were made to constrain the tracking problem. It was assumed that:

1. objects to be tracked will be puck-like robots and that they will each appear in different shades of red or blue, or green.
2. camera will be set up in an angle not less than 45° with respect to the plane it is observing.
3. on the top of the robot will be a triangle indicating it's direction and that it will be in darker colour than the colour indicating the robot.
4. background of the image will be in colour other than any of the colours of the robots.

2.1 Detection of Robots

Input

I , a three channel image of dimensions $m \times n$ in the RGB colorspace.

Output

M , a $m \times n \times 3$ binary matrix where for each pixel P_{ij} of I , it holds that:

$M(i, j, 1) = 1 \leftrightarrow P_{ij}$ belongs to the red robot,

$M(i, j, 2) = 1 \leftrightarrow P_{ij}$ belongs to the green robot,

$M(i, j, 3) = 1 \leftrightarrow P_{ij}$ belongs to the blue robot.

Algorithm

1. Apply approximate RGB-normalisation to I , giving I_n :
 - For each pixel in I_n , calculate the sum S_{rgb} of the red, green, and blue values of that pixel.
 - If $S_{rgb} \neq 0$ (the pixel is not absolute black), set each of the pixel's red, green, and blue values to that value divided by S_{rgb} .
2. Calculate μ_r, μ_g, μ_b and $\sigma_r, \sigma_g, \sigma_b$, the means and standard deviations of the values in the three channels of I_n .
3. Assign each pixel P_{ij} in I to one of the robots or to the background:
 - Normalise P 's red, green, and blue values, giving P_n .
 - Calculate the probabilities p_r, p_g, p_b that P_n was generated by the gaussian distributions $\mathcal{N}_r = (\mu_r, \sigma_r), \mathcal{N}_g = (\mu_g, \sigma_g), \mathcal{N}_b = (\mu_b, \sigma_b)$.
 - Calculate P 's hue value h .
 - If h is within a certain range defined as red and p_r is sufficiently small, set $M(i, j, 1) = 1$ (similarly for ranges defined as green/blue and p_g/p_b . If none of these conditions are met, set $M(i, j, 1) = M(i, j, 2) = M(i, j, 3) = 0$.
4. Remove noise from each channel in M :
 - Set pixels to zero if they have fewer neighbours with value one than they have adjacent pixels with value zero.
 - Set zero-valued pixels to one if they have two one-valued horizontal or vertical neighbours.
5. Remove components that are distant from the main concentration of mass in each channel in M :
 - Compute the center of mass C of the channel.
 - Compute c_1, c_2, \dots , the centers of mass of each connected component in the channel.
 - Compute the mean distance δ of the c_k to C .
 - Set $M(i, j) = 0$ for all the pixels (i, j) in the components k that satisfy $c_k > \tau\delta$ for some fixed threshold τ .
6. Exploit the fact that all robots have similar sizes by setting every channel in M to all-zeros if the number of pixels set in that channel is smaller than the number of pixels set in the most populated channel by some margin.

Figure 1 shows a visualisation of the output matrix M .

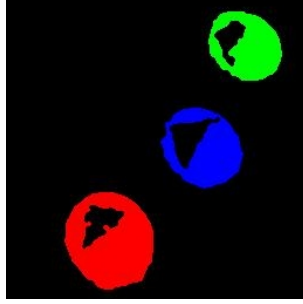


Figure 1: Result of Robot detection

2.2 Finding Robot Directions

Input

I , a three channel image of dimensions $m \times n$ in the RGB colorspace.

Output

$\Lambda = \{(c_r^m, c_r^t), (c_g^m, c_g^t), (c_b^m, c_b^t)\}$, a set where c_r^m is the center of mass of the red robot and c_r^t is the point towards which the robot is facing (similarly for the green and blue robots).

Algorithm

1. Get a matrix of robot masks M using the algorithm in Section 2.1. Let M_i be the i^{th} channel of M i.e. the set of points $\{M(a, b, i) | 1 \leq a \leq m, 1 \leq b \leq n\}$. Apply the remainder of the algorithm to each channel ξ in M .
2. Calculate the convex hull H of the points in the channel and create the set of pixels of I that are inside H : $P = \{p_{ij} | p_{ij} \in \xi \wedge M(i, j, \xi) = 1\}$
3. Calculate μ , the average rgb-value over P . Generate $\Pi = \{p | p \in P \wedge rgbvalue(p) < \mu\}$, the set of pixels in P that have a below-average rgb value.
4. The black triangles on the robots are the pixels in Π . Get rid of them by setting the relevant indices in M to zero.
Recompute the convex hull of M .
5. Repeat the previous step and remember the pixels in Π .
This reduces noise in M by giving a tighter estimate on the robot's pixels when the triangles were under-detected by the algorithm in Section 2.1.
Figure 2 shows the result of this step - a notable improvement in clarity of the triangles compared to Figure 1.
6. Update Λ : c_ξ^m is the center of mass of M_ξ , c_ξ^t is the center of mass of Π .
A line from c_ξ^m to c_ξ^t indicates the direction of the robot.

Figure 3 shows a visualisation of the output set Λ .

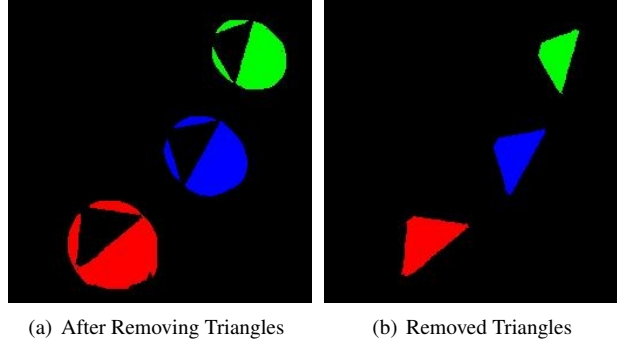


Figure 2: Triangle Detection via Local Thresholding

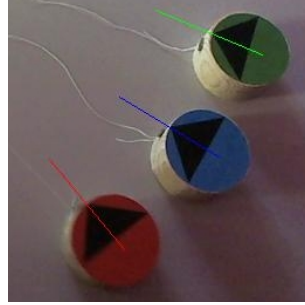


Figure 3: Detected Directions

2.3 Tracking Robots Over a Sequences of Frames

Input

$\Upsilon = \{I_1, I_2, \dots\}$, a sequence where each of the I_i is a three channel image of dimensions $m \times n$ in the RGB colorspace.

Output

Ω , a visualisation of the robot positions over Υ .

Algorithm

1. Use a median-filter to generate a background Ω from Υ .
 For each $1 \leq i \leq m, 1 \leq j \leq n$:
 - Create $\omega_{ij} = \{I_k(i, j) | I_k \in \Upsilon\}$, the set of the colors of the pixels at location (i, j) of all the images in Υ .
 - Set $\Omega(i, j) = \text{median}(\omega_{ij})$.
2. For each $I_i \in \Upsilon$:
 - Use the algorithm in Section 2.2 to get the set Λ . Let $\lambda = \{c | (c, -) \in \Lambda\}$.
 - Overlay Ω with a line from each element in λ_{i-1} to the corresponding element in λ_i , thus linking the centroids from image I_{i-1} to the centroids in image I_i .

Figure 4 shows a visualisation of the resulting track.

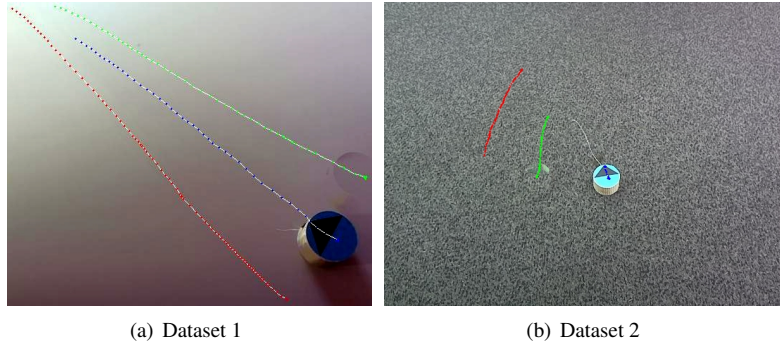


Figure 4: Output of Tracing Algorithm

3 Results

This section evaluates and visualises the performance of the three algorithms presented in Sections 2.1, 2.2, and 2.3. Table 1 describes the properties of the datasets used for this evaluation.

#	Background	Robot Size	Robot Color	Illumination
1	uniform, gray	large	saturated, dark	uniform, red hue
2	noisy, gray	small	faded, blue robot is cyan	histograms are bell-shaped
3	patterned, brown	large	saturated	daylight only
4	patterned, brown	large	saturated	daylight and artificial light

Table 1: Properties of evaluation datasets

3.1 Detection of Robots

The algorithm described in Section 2.1, worked perfectly on datasets 1 and 2. Evaluation on the third dataset led to the worst performance over all datasets, with $\sim 60\%$ of the occurrences of the blue robot being undetected and $\sim 40\%$ of the occurrences of the green robot being under-detected (leading to bad direction detection). The performance on the fourth dataset was interesting: the red robot was under-detected in $\sim 45\%$ of the cases (with the blue and green robots being found just fine) - while in the other datasets the red robot was usually detected with the highest confidence. Over all four datasets, about 10% of the robot instances were badly detected.

The fact that the color-detection algorithm works well on both datasets 1 and 2 leads to the conjecture that it is invariant under texture changes in the scene background and variations in robot-color saturation. The bad performance on dataset 3 can be explained by interference from the color of the scene background and by changes in scene illumination. The fact that the algorithm offers almost top-level performance on dataset 4 (captured on the same background as dataset 3) implies that the change in scene illumination is probably the largest influence on the algorithm’s performance.

This is in keeping with the intuition that daylight has more inherent variation than arti-

ficial light, thus introducing a higher degree of variability into the characteristics of the captured images.

3.2 Detection of Directions

Performance of robot direction detection, understandably, is heavily dependent on the performance of the detection of the robots. If the robots are well isolated by Section 2.1's algorithm, robot orientations are perfectly detected.

The algorithm is invariant under loose detection - false positive cases where some addition non-robot region is misleadingly detected as a robot. This is due to the algorithm's ability to filter-out noisy detections.

In case of under-detection - false negative cases where some parts of the image representing the robot where not detected - the algorithm breaks: the predicted direction is skewed towards the opposite side of the under-detection. This is due to the algorithm not putting strict circular or elipsoidal constraints on the shape of the robots: under detection is thus able to move the center of mass of either the triangle or the robot-convex-hull. The error introduced by this is proportional to the area of the under-detected region.

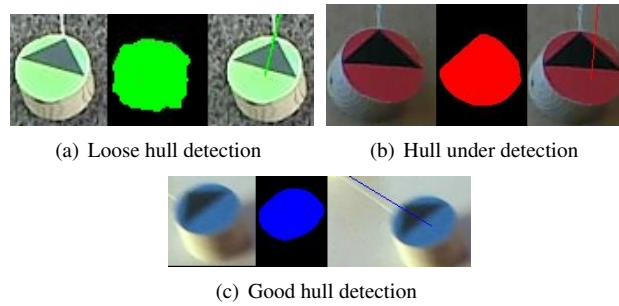


Figure 5: Direction detections for convex hulls of different qualities

3.3 Tracking of the robots

Section 2.3's algorithm to track robots over consecutive frames is trivial - a mere visualisation of half of the results of the robot-direction- detection algorithm presented in Section 2.2. The tracking algorithm's performance is therefore directly related to the performance of the robot-direction-detection algorithm and the same observations as in Section 3.2 apply: generally speaking, the algorithm performed well.

Figure 4 begets one additional observation related to the evaluation of the robot-tracking algorithm: both datasets considered in this report exhibit the property that one of the objects of interest does not move much for most of the frames. This entails that generating a background from data employing a simple frame-difference base approach (such as the median- filter used in Section 2.3) to perform background subtraction is bound to fail as one of the objects of interest will be considered a part of the background due to being mostly stationary. This is unfortunate since pre-processing the datasets with background subtraction would increase the accuracy of Section 2.1's algorithm by reducing noise and increasing resolution in the image.

4 Discussion

The algorithms in Sections 2.1, 2.2, and 2.3 operate under a limited number of reasonable simplifying assumptions and performed well across a range of datasets captured under very different conditions.

The robot color detection algorithm of Section 2.1 could be improved by adding more stringent conditions on the shape of the robots e.g. fitting a circle or ellipse to them rather than a convex hull. This should improve the subsequent performance of the direction-detection algorithm by reducing the amount of under-detections.

If assumptions about the size, scale, and shape of the robots can be made, Hough transforms or other shape-based techniques could be used to detect the robots. This would be more robust than the current color/pixel based approach but not invariant under scale or differences in camera positions.

Another way to improve the detection of the robots could be augmenting the current approach with the utilisation of second order spatial image statistics to pre-process the images. This would lead to a rough estimate of the robot positions with few false negatives, thus improving the currently utilised algorithms due to a reduced search space (implying a denser signal). A reduced search space also allows for computationally expensive but high-precision techniques to be used. One example of such a technique is a local color-based search starting from a small seed of pixels that are hypothesised to be part of a robot.

Direction detection could be improved by making it less dependent on the accuracy of the robot detection. One way to achieve this would be to base the direction calculation on properties of the detected triangles (e.g. direction = tangent line to the longest side) rather than on the triangles' centers of mass. This would increase robustness to under-detections.

Cross-frame tracking could be improved by abusing spatio-temporal proximity. One way to do this would be put a cap on how far the centroid of the robot can move in a certain frame interval, thus smoothing out the odd completely wrong prediction.

References

- [1] Yilmaz, A., Javed, O., and Shah, M., "Object Tracking: A Survey", *ACM Comput. Surv.* 38, 4, Article 13, 2006.

```

1 %%%%%%%%% file: main.m %%%%%%%%%
2 function res = main(path, image_type, start_offset, time_step)
3     if nargin < 2
4         image_type = 'jpg';
5     end
6     if nargin < 3
7         start_offset = 0;
8     end
9     if nargin < 4
10        time_step = 0.33;
11    end
12
13    _path = mfilename('fullpath');
14    [_path, _, _] = fileparts(_path);
15    addpath(fullfile(_path, 'algo'));
16    addpath(fullfile(_path, 'draw'));
17    addpath(fullfile(_path, 'test'));
18
19    files = dir(sprintf('%s/*.%s', path, image_type));
20    filenames = {files.name};
21    [~, num_files] = size(filenames);
22
23    [m n ~] = size(imread(sprintf('%s/%s', path, filenames{1})));
24    track_mask = zeros(m, n);
25
26    rc = zeros(num_files - start_offset, 2);
27    gc = zeros(num_files - start_offset, 2);
28    bc = zeros(num_files - start_offset, 2);
29
30    reds = cell(num_files - start_offset, 1);
31    greens = cell(num_files - start_offset, 1);
32    blues = cell(num_files - start_offset, 1);
33
34    for i = 1 + start_offset : num_files
35        image = imread(sprintf('%s/%s', path, filenames{i}));
36        [directions, centroids] = analyse_image(image);
37
38        r = uint16(centroids(1,:));
39        if r(1) > 0 && r(2) > 0
40            track_mask = overlay_cross(track_mask, 1, r(1), r(2));
41            rc(i,:) = [r(1), r(2)];
42        end
43
44        g = uint16(centroids(2,:));
45        if g(1) > 0 && g(2) > 0
46            track_mask = overlay_cross(track_mask, 2, g(1), g(2));
47            gc(i,:) = [g(1), g(2)];
48        end
49
50        b = uint16(centroids(3,:));
51        if b(1) > 0 && b(2) > 0

```



```

52     track_mask = overlay_cross(track_mask, 3, b(1), b(2));
53     bc(i,:) = [b(1), b(2)];
54 end
55
56     reds{i} = image(:,:,1);
57     greens{i} = image(:,:,2);
58     blues{i} = image(:,:,3);
59     imshow(overlay_mask(image, directions));
60
61     pause(time_step);
62 end
63
64 rc = rc(any(rc,2),:);
65 gc = gc(any(gc,2),:);
66 bc = bc(any(bc,2),:);
67
68 red_median = median(cat(3, reds{:}), 3);
69 green_median = median(cat(3, greens{:}), 3);
70 blue_median = median(cat(3, blues{:}), 3);
71
72 bg = cat(3, red_median, green_median, blue_median);
73
74 bg = overlay_polygon(bg, rc, [255, 255, 255]);
75 bg = overlay_polygon(bg, gc, [255, 255, 255]);
76 bg = overlay_polygon(bg, bc, [255, 255, 255]);
77 bg = overlay_mask(bg, track_mask);
78 imshow(bg)
79 end
80
81
82 %%%%%%%%% file: analyse_image.m %%%%%%%%%
83 function [pretty_mask, varargout] = analyse_image(image)
84     [num_rows, num_cols, num_channels] = size(image);
85     blob_mask = mask_colors(image);
86     [convex_mask, ~] = mask_convex_regions(image, blob_mask);
87     [convex_mask] = demask_triangles(image, convex_mask, false);
88     [convex_mask, ~] = mask_convex_regions(image, convex_mask);
89     [~, centroids, ~, triangle_centroids] = ...
90         demask_triangles(image, convex_mask, true);
91     pretty_mask = overlay_rays(zeros(num_rows,num_cols,num_channels),...
92         centroids,triangle_centroids, 99, ...
93         'Color', [1 0 0; 0 1 0; 0 0 1]);
94     varargout{1} = centroids;
95     varargout{2} = triangle_centroids;
96     varargout{3} = convex_mask;
97 end
98
99
100 % this function can used to find both - triangles and masks without
101 % triangles. To find triangles it must be run with

```

```

102 % get_triangle_centroids=true. Also application of the function int the
103 % following fashion gives better results.
104 % [convex_mask, ~] = mask_convex_regions(image, blob_mask);
105 % [convex_mask]=demask_triangles(image, convex_mask, false);
106 % [convex_mask, ~] = mask_convex_regions(image, convex_mask);
107 function [mask, varargout] = demask_triangles(image, mask, ...
108         get_triangle_centroids)
109 [num_rows, num_cols, num_channels] = size(image);
110 centroids = zeros(num_channels, 2);
111 triangle_centroids = zeros(num_channels, 2);
112 trinagle_mask = mask;
113 for c = 1 : num_channels
114     channel = image(:,:,c);
115     channel_mask = mask(:,:,c);
116     channel_trinagle_mask = mask(:,:,c);
117     if ~any(channel_mask)
118         continue;
119     end
120     rgb_values = zeros(sum(channel_mask(:)), 1);
121     idx = 1;
122     for nr = 1 : num_rows
123         for nc = 1 : num_cols
124             if channel_mask(nr, nc) == 0
125                 continue;
126             end
127             rgb_values(idx) = channel(nr, nc);
128             idx = idx + 1;
129         end
130     end
131     mean_rgb = mean(rgb_values(:));
132     channel_mask(channel < mean_rgb) = 0;
133     mask(:,:,c) = channel_mask;
134     props = regionprops(channel_mask, 'Centroid');
135     centroid = props.Centroid;
136     centroids(c,:) = [centroid(2), centroid(1)];
137     if get_triangle_centroids
138         channel_trinagle_mask(channel > mean_rgb) = 0;
139         channel_trinagle_mask=filter_mask(channel_trinagle_mask);
140         trinagle_mask(:,:,c) = channel_trinagle_mask;
141         props = regionprops(channel_trinagle_mask, 'Centroid');
142         centroid = props.Centroid;
143         triangle_centroids(c,:) = [centroid(2), centroid(1)];
144     end
145 end
146 varargout{1} = centroids;
147 varargout{2} = trinagle_mask;
148 varargout{3} = triangle_centroids;
149 end
150
151

```

```

152 function [color_mask, varargout] = mask_convex_regions(image, mask)
153     [num_rows, num_cols, num_channels] = size(image);
154     color_mask = zeros(num_rows, num_cols, num_channels);
155     convex_centroids = zeros(num_channels, 2);
156     for c = 1 : num_channels
157         channel = mask(:, :, c);
158         if ~any(channel(:))
159             continue;
160         end
161         props = regionprops(channel, 'Centroid', 'ConvexImage', 'BoundingBox');
162         convex_props = regionprops(props.ConvexImage, 'Centroid');
163         convex_centroid = convex_props.Centroid;
164         convex_centroid = [convex_centroid(2) + props.BoundingBox(2), ...
165             convex_centroid(1) + props.BoundingBox(1)];
166         convex_centroids(c, :) = convex_centroid;
167         convex_image = props.ConvexImage;
168         [num_rows_convex, num_cols_convex] = size(convex_image);
169         for row = 1 : num_rows_convex
170             for col = 1 : num_cols_convex
171                 if convex_image(row, col) == 1
172                     newrow = round(row + props.BoundingBox(2));
173                     newcol = round(col + props.BoundingBox(1));
174                     color_mask(newrow, newcol, c) = 1;
175                 end
176             end
177         end
178     end
179     varargout{1} = convex_centroids;
180 end
181
182
183 function image_mask = mask_colors(image)
184     [num_rows, num_cols, ~] = size(image);
185     num_pixels = num_rows * num_cols;
186     image_mask = zeros(num_pixels, 3);
187     rgb = double(reshape(image, num_pixels, 3));
188     rgbN = double(reshape(normalise_rgb(image, 'approximate'), num_pixels, 3));
189     rN_sdev = std(rgbN(:, 1));
190     gN_sdev = std(rgbN(:, 2));
191     bN_sdev = std(rgbN(:, 3));
192     rN_mean = mean(rgbN(:, 1));
193     gN_mean = mean(rgbN(:, 2));
194     bN_mean = mean(rgbN(:, 3));
195     hsv = reshape(rgb2hsv(image), num_pixels, 3);
196     for c = 1 : num_pixels
197         rN = rgbN(c, 1);
198         gN = rgbN(c, 2);
199         bN = rgbN(c, 3);
200         hue = hsv(c, 1) * 360;
201         % current pixel is red

```

```

202     if (hue >= 330 || hue <= 30) && ...
203         (normal_prob(rN, rN_mean, rN_sdev) < 0.001)
204         image_mask(c,1) = 1;
205     % current pixel is green
206     elseif (hue >= 80 && hue < 180) && ...
207         (normal_prob(gN, gN_mean, gN_sdev) < 0.007)
208         image_mask(c,2) = 1;
209     % current pixel is blue
210     elseif (hue >= 150 && hue <= 270) && ...
211         (normal_prob(bN, bN_mean, bN_sdev) < 0.0000085)
212         image_mask(c,3) = 1;
213     end
214 end
215 image_mask = reshape(image_mask, num_rows, num_cols, 3);
216 image_mask = remove_noise(image_mask);
217 image_mask = remove_outliers(image_mask);
218 image_mask = enforce_similar_channel_areas(image_mask);
219 end
220
221
222 function x = normal_prob(val, mu, sigma)
223     x = 1.0 / (sigma * sqrt(2 * pi)) * exp(-(val - mu) ^ 2 / (2 * sigma ^ 2));
224 end
225
226
227 function image = remove_noise(image)
228     [~, ~, num_channels] = size(image);
229     for c = 1 : num_channels
230         channel = image(:,:,c);
231         channel = bwmorph(channel, 'majority', Inf);
232         channel = bwmorph(channel, 'bridge', Inf);
233         image(:,:,c) = channel;
234     end
235 end
236
237
238 % finds the connected components in each channel of |image| and removes those
239 % that are far away from the centroid of the pixels in that channel
240 % here, 'far away' means more distant than |distance_proportion_threshold| times
241 % the average distance of each connected component to the channel centroid
242 % this removes big areas of noise such as the big green blob inside of the black
243 % arrow of the robot in data/1/00000006.jpg
244 function image = remove_outliers(image, distance_proportion_threshold)
245     if nargin < 2
246         distance_proportion_threshold = 0.5;
247     end
248     [~, ~, num_channels] = size(image);
249     for c = 1 : num_channels
250         channel = image(:,:,c);
251         if ~any(channel(:))

```

```

252     continue;
253 end
254 channel_properties = regionprops(channel, 'Centroid');
255 channel_centroid = channel_properties.Centroid;
256 regions = bwconncomp(channel);
257 regions_properties = regionprops(regions, 'Centroid', 'PixelIdxList');
258 regions_centroids = {regions_properties.Centroid};
259 distances = cellfun(@(x) norm(x - channel_centroid), regions_centroids);
260 mean_distance = mean(distances);
261 for d = 1 : length(distances)
262     if distances(d) > mean_distance * distance_proportion_threshold
263         idx = regions_properties(d).PixelIdxList;
264         channel(idx) = 0;
265     end
266 end
267 image(:,:,c) = channel;
268 end
269 end
270
271
272 % we know that the robots are all about the same size - we can thus remove any
273 % channels in the mask that have a much smaller area than the other channels
274 % this catches some problems like the shadow of the blue robot in
275 % data/1/00000095.jpg being detected as a red blob
276 function image = enforce_similar_channel_areas(image, area_proportion_threshold)
277     if nargin < 2
278         area_proportion_threshold = 0.5;
279     end
280     [~, ~, num_channels] = size(image);
281     areas = zeros(num_channels, 1);
282     for c = 1 : num_channels
283         channel = image(:,:,c);
284         if ~any(channel(:))
285             continue;
286         end
287         region_props = regionprops(channel, 'Area');
288         areas(c) = region_props.Area;
289     end
290     max_area = max(areas(areas > 0));
291     for c = 1 : num_channels
292         area = areas(c);
293         if area == 0
294             continue;
295         end
296         if (area < max_area * area_proportion_threshold) || ...
297             (area > max_area / area_proportion_threshold)
298             image(:,:,c) = image(:,:,c) * 0;
299         end
300     end
301 end

```

```

302
303
304 % this filters out all connected regions but the biggest one
305 function mask = filter_mask(mask)
306     [x, y, num_channels] = size(mask);
307     for c = 1 : num_channels
308         channel = zeros(x, y);
309         channel = reshape(channel, x * y, 1);
310         blob_info = bwconncomp(mask(:,:,c));
311         blob_list = blob_info.PixelIdxList;
312         [nrows, ncols] = cellfun(@size, blob_list);
313         largest_blob_pixels = blob_list{find(nrows == max(nrows))};
314         for i = 1 : max(nrows)
315             channel(largest_blob_pixels(i)) = 1;
316         end
317         channel = reshape(channel, x, y);
318         mask(:,:,c) = channel;
319     end
320 end
321
322
323 % normalises the values of the red, green, and blue channels of |image| in order
324 % to eliminate illumination differences in the image
325 % formula used: {r, g, b} = {r, g, b} / sqrt(r^2 + g^2 + b^2)
326 % if 'approximate' is passed as an additional parameter, instead use
327 % {r, g, b} = {r, g, b} / (r + g + b)
328 % this is approximately two times faster than the exact normalisation
329 function normalised_image = normalise_rgb(image, varargin)
330     approximate = ~isempty(find(strcmpi(varargin, 'approximate')));
331     red = double(image(:,:,1));
332     green = double(image(:,:,2));
333     blue = double(image(:,:,3));
334     if approximate
335         euclid_rgb = red(:,:) + green(:,:) + blue(:,:);
336     else
337         euclid_rgb = sqrt(red(:,:).^2 + green(:,:).^2 + blue(:,:).^2);
338     end
339     red_norm = round(red(:,:) ./ euclid_rgb .* 255);
340     green_norm = round(green(:,:) ./ euclid_rgb .* 255);
341     blue_norm = round(blue(:,:) ./ euclid_rgb .* 255);
342     % some pixels are absolute black (r = g = b = 0) which causes division by
343     % zero errors during normalisation and NaN values in the normalised channels
344     % need to filter these values out
345     red_norm(isnan(red_norm)) = 0;
346     green_norm(isnan(green_norm)) = 0;
347     blue_norm(isnan(blue_norm)) = 0;
348     red_norm = uint8(red_norm);
349     green_norm = uint8(green_norm);
350     blue_norm = uint8(blue_norm);
351     normalised_image = cat(3, red_norm, green_norm, blue_norm);

```

```

352 end
353 %%%%%%%%% end of file: analyse_image.m %%%%%%%%%
354
355
356 %%%%%%%%% file: median_filter.m %%%%%%%%%
357 % generates a background image from a set of sample images
358 % subtracting the background from the sample images eases object detection
359 % does not work on these datasets because the blue/cyan robot does not move
360 % for most of the images i.e. will be considered part of the background
361 function background = median_filter(path, image_type, start_offset, step)
362     if nargin < 2
363         image_type = 'jpg';
364     end
365     if nargin < 3
366         start_offset = 0;
367     end
368
369     dim = 2;
370
371     files = dir(sprintf('%s/*.%s', path, image_type));
372     filenames = {files.name};
373     [~, num_files] = size(filenames);
374
375     reds = cell(num_files - start_offset, 1);
376     greens = cell(num_files - start_offset, 1);
377     blues = cell(num_files - start_offset, 1);
378     for c = 1 + start_offset : step : num_files
379         image = imread(sprintf('%s/%s', path, filenames{c}));
380         reds{c} = image(:, :, 1);
381         greens{c} = image(:, :, 2);
382         blues{c} = image(:, :, 3);
383     end
384     red_median = median(cat(dim + 1, reds{:}), dim + 1);
385     green_median = median(cat(dim + 1, greens{:}), dim + 1);
386     blue_median = median(cat(dim + 1, blues{:}), dim + 1);
387
388     background = cat(dim + 1, red_median, green_median, blue_median);
389 end
390 %%%%%%%%% end of file: median_filter.m %%%%%%%%%
391
392
393 %%%%%%%%% file: normalise_rgb.m %%%%%%%%%
394 % normalises the values of the red, green, and blue channels of [image] in order
395 % to eliminate illumination differences in the image
396 % formula used: {r, g, b} = {r, g, b} / sqrt(r^2 + g^2 + b^2)
397 % if 'approximate' is passed as an additional parameter, instead use
398 % {r, g, b} = {r, g, b} / (r + g + b)
399 % this is approximately two times faster than the exact normalisation
400 function normalised_image = normalise_rgb(image, varargin)
401     approximate = ~isempty(find(strcmpi(varargin, 'approximate')));

```

```

402
403 red = double(image(:,:,1));
404 green = double(image(:,:,2));
405 blue = double(image(:,:,3));
406
407 if approximate
408     euclid_rgb = red(:,:,) + green(:,:,) + blue(:,:,);
409 else
410     euclid_rgb = sqrt(red(:,:,).^2 + green(:,:,).^2 + blue(:,:,).^2);
411 end
412 red_norm = round(red(:,:,) ./ euclid_rgb .* 255);
413 green_norm = round(green(:,:,) ./ euclid_rgb .* 255);
414 blue_norm = round(blue(:,:,) ./ euclid_rgb .* 255);
415
416 % some pixels are absolute black (r = g = b = 0) which causes division by
417 % zero errors during normalisation and NaN values in the normalised channels
418 % need to filter these values out
419 red_norm(isnan(red_norm)) = 0;
420 green_norm(isnan(green_norm)) = 0;
421 blue_norm(isnan(blue_norm)) = 0;
422
423 red_norm = uint8(red_norm);
424 green_norm = uint8(green_norm);
425 blue_norm = uint8(blue_norm);
426 normalised_image = cat(3, red_norm, green_norm, blue_norm);
427 end
428 %%%%%%%%% end of file: normalise_rgb.m %%%%%%%%%
429
430
431 %%%%%%%%% file: overlay_circles.m %%%%%%%%%
432 % draws the circles defined by the centres in |centers| and radiuses in |radii|
433 % onto |image| and returns the modified image
434 % [0, 0] is the top left corner of the image
435 % if |radii| is a number, draw all circles with that radius
436 function image = overlay_circles(image, centers, radii, varargin)
437 % parse options
438 argc = size(varargin, 2);
439 c = 1;
440 while c <= argc
441     arg = varargin{c};
442     if strcmpi(arg, 'Color')
443         if c + 1 > argc
444             error('Color option should be followed by an integer tripplet');
445         end
446         colors = varargin{c + 1};
447         c = c + 1;
448     end
449 end
450 % option defaults
451 if ~exist('colors', 'var')

```



```

452     colors = [255 255 255];
453 end
454
455 num_circles = size(centers, 1);
456 if size(radii, 1) == 1
457     radii = repmat(radii, num_circles, 1);
458 end
459 if size(colors, 1) == 1
460     colors = repmat(colors, num_circles, 1);
461 end
462
463 centers = round(centers);
464 radii = round(radii);
465
466 [~, ~, num_channels] = size(image);
467 for c = 1 : num_channels
468     channel = image(:,:,c);
469     channel = overlay_circles_channel(channel, centers, radii, colors(:,c));
470     image(:,:,c) = channel;
471 end
472 end
473
474
475 function channel = overlay_circles_channel(channel, centers, radii, colors)
476     [xmax, ymax] = size(channel);
477     num_circles = size(centers, 1);
478     for c = 1 : num_circles
479         Xc = centers(c,1);
480         Yc = centers(c,2);
481         radius = radii(c);
482
483         for theta = 0 : 0.1 : 359
484             x = round(Xc + radius * cos(theta));
485             y = round(Yc + radius * sin(theta));
486             if x <= xmax && x > 0 && y <= ymax && y > 0
487                 channel(x, y) = colors(c,:);
488             end
489         end
490     end
491 end
492 %%%%%%%%% end of file: overlay_circles.m %%%%%%%%%
493
494
495 %%%%%%%%% file: overlay_cross.m %%%%%%%%%
496 % returns image with a cross centered on pixel (|x|, |y|) drawn in |channel|
497 % [0, 0] is the top left corner of the image
498 function image = overlay_cross(image, channel, x, y)
499     [h w ~] = size(image);
500
501     if channel == 1

```

```

502     other1 = 2;
503     other2 = 3;
504 end
505 if channel == 2
506     other1 = 1;
507     other2 = 3;
508 end
509 if channel == 3
510     other1 = 1;
511     other2 = 2;
512 end
513
514 if y + 1 < h
515     image(x, y + 1, channel) = 1;
516     image(x, y + 1, other1) = 0;
517     image(x, y + 1, other2) = 0;
518 end
519
520 if y - 1 > 0
521     image(x, y - 1, channel) = 1;
522     image(x, y - 1, other1) = 0;
523     image(x, y - 1, other2) = 0;
524 end
525
526 if x + 1 < w
527     image(x + 1, y, channel) = 1;
528     image(x + 1, y, other1) = 0;
529     image(x + 1, y, other2) = 0;
530 end
531
532 if x - 1 > 0
533     image(x - 1, y, channel) = 1;
534     image(x - 1, y, other1) = 0;
535     image(x - 1, y, other2) = 0;
536 end
537
538 image(x, y, channel) = 1;
539 image(x, y, other1) = 0;
540 image(x, y, other2) = 0;
541
542 end
543
544 %%%%%%%%% end of file: overlay_cross.m %%%%%%%%%
545
546
547 %%%%%%%%% file: overlay_mask.m %%%%%%%%%
548 % returns the result of putting |mask| onto |image|
549 % for each pixel that is set in some channel of |mask|, saturates the pixel in
550 % the equivalent channel of |image|
551 function image = overlay_mask(image, mask, varargin)

```

```

552 % parse options
553 argc = size(varargin, 2);
554 c = 1;
555 while c <= argc
556     arg = varargin{c};
557     if strcmpi(arg, 'GrayScale')
558         grayscale = 1;
559     elseif strcmpi(arg, 'Saturation')
560         if c + 1 > argc
561             error('Saturation option should be followed by a double');
562         end
563         saturation = varargin{c + 1};
564         c = c + 1;
565     elseif strcmpi(arg, 'Lightness')
566         if c + 1 > argc
567             error('Lightness option should be followed by a double');
568         end
569         lightness = varargin{c + 1};
570         c = c + 1;
571     end
572     c = c + 1;
573 end
574
575 % modify background image
576 if exist('grayscale', 'var')
577     gray_image = rgb2gray(image);
578     image = cat(3, gray_image, gray_image, gray_image);
579 end
580 if exist('saturation', 'var')
581     hsv_image = rgb2hsv(image);
582     hsv_image(:,:,2) = hsv_image(:,:,2) * saturation;
583     image = hsv2rgb(hsv_image);
584 end
585 if exist('lightness', 'var')
586     hsv_image = rgb2hsv(image);
587     hsv_image(:,:,3) = hsv_image(:,:,3) * lightness;
588     image = hsv2rgb(hsv_image);
589 end
590
591 % lay mask onto background image
592 num_channels = size(image, 3);
593 channels = 1 : num_channels;
594 for c = 1 : num_channels
595     channel = image(:,:,c);
596     mask_pixels = find(mask(:,:,c) == 1);
597     channel(mask_pixels) = 255;
598     image(:,:,c) = channel;
599     for d = setdiff(channels, c)
600         channel = image(:,:,d);
601         channel(mask_pixels) = 0;

```

```

602     image(:, :, d) = channel;
603 end
604 end
605 end
606 %%%%%%%%% end of file: overlay_mask.m %%%%%%%%%
607
608
609 %%%%%%%%% file: overlay_polygon.m %%%%%%%%%
610 % returns image with lines drawn from the nth point in |points| to the n+1th
611 % [0, 0] is the top left corner of the image
612 function image = overlay_polygon(image, points, color)
613     if nargin < 3
614         color = [255 255 255];
615     end
616
617     points = round(points);
618
619     num_channels = size(image, 3);
620     for c = 1 : num_channels
621         channel = image(:, :, c);
622         channel = overlay_polygon_channel(channel, points, color(c));
623         image(:, :, c) = channel;
624     end
625 end
626
627
628 % Bresenham's line algorithm (simplified version)
629 % http://en.wikipedia.org/wiki/Bresenham's\_line\_algorithm#Simplification
630 function channel = overlay_polygon_channel(channel, points, color)
631     [xmax, ymax] = size(channel);
632     for c = 1 : length(points) - 1
633         start = points(c, :);
634         stop = points(c + 1, :);
635         x0 = start(1);
636         y0 = start(2);
637         x1 = stop(1);
638         y1 = stop(2);
639
640         dx = abs(x1 - x0);
641         dy = abs(y1 - y0);
642
643         if x0 < x1
644             sx = 1;
645         else
646             sx = -1;
647         end
648         if y0 < y1
649             sy = 1;
650         else
651             sy = -1;

```

```

652     end
653
654     err = dx - dy;
655
656     while 1
657         if x0 > 0 && x0 <= xmax && y0 > 0 && y0 <= ymax
658             channel(x0, y0) = color;
659         end
660         if x0 == x1 && y0 == y1
661             break;
662         end
663         e2 = 2 * err;
664         if e2 > -dy
665             err = err - dy;
666             x0 = x0 + sx;
667         end
668         if e2 < dx
669             err = err + dx;
670             y0 = y0 + sy;
671         end
672     end
673 end
674 end
675 %%%%%%%%% end of file: overlay_polygon.m %%%%%%%%%
676
677
678 %%%%%%%%% file: overlay_rays.m %%%%%%%%%
679 function image = overlay_rays(image, from, to, length, varargin)
680 % parse options
681 argc = size(varargin, 2);
682 c = 1;
683 while c <= argc
684     arg = varargin{c};
685     if strcmpi(arg, 'Color')
686         if c + 1 > argc
687             error('Color option should be followed by an integer triplet');
688         end
689         colors = varargin{c + 1};
690         c = c + 1;
691     end
692     c = c + 1;
693 end
694 % option defaults
695 if ~exist('colors', 'var')
696     colors = [255 255 255];
697 end
698
699 num_rays = size(from, 1);
700 if size(colors, 1) == 1;
701     colors = repmat(colors, num_rays, 1);

```

```

702 end
703
704 for c = 1 : num_rays
705     if from(c,:) == to(c,:)
706         continue;
707     end
708     x0 = from(c,1);
709     y0 = from(c,2);
710     x1 = to(c,1);
711     y1 = to(c,2);
712     dx = x0 - x1;
713     dy = y0 - y1;
714     lambda = min(sqrt(length ^ 2 / (dx ^ 2 + dy ^ 2)), ...
715         -sqrt(length ^ 2 / (dx ^ 2 + dy ^ 2)));
716
717     x = x0 + lambda * dx;
718     y = y0 + lambda * dy;
719
720     image = overlay_polygon(image, [x0 y0; x y], colors(c,:));
721 end
722 end
723 %%%%%%%%% end of file: overlay_rays.m %%%%%%%%%
724
725
726 %%%%%%%%% file: random_image.m %%%%%%%%%
727 % returns a random image from some directory D and print the path to that image
728 % the function understands the following options:
729 % 'Quiet'          don't print the path to the image
730 % 'ImageType', C   look for images of type C (default = 'jpg')
731 % any remaining parameters are taken to be the path to D
732 % if D is not specified, default to a random sub-directory of "ug3_Vision/data"
733 function [image, varargout] = random_image(varargin)
734     TL_DIR = 'ug3_Vision';
735     BRANCH_DIR = 'data';
736     % parse options
737     argc = length(varargin);
738     c = 1;
739     while c <= argc
740         arg = varargin{c};
741         if strcmpi(arg, 'Quiet')
742             quiet = 1;
743         elseif strcmpi(arg, 'ImageType')
744             if c + 1 > argc
745                 error('ImageType option should be followed by a string');
746             end
747             image_type = varargin{c + 1};
748             c = c + 1;
749         else
750             path = arg;
751         end

```

```

752     c = c + 1;
753 end
754 % option defaults
755 if ~exist('quiet', 'var')
756     quiet = 0;
757 end
758 if ~exist('image_type', 'var')
759     image_type = '.jpg';
760 end
761 if ~exist('path', 'var')
762     path = random_dir(TL_DIR, BRANCH_DIR);
763 end
764
765 if ~strcmp(image_type(1), '.')
766     image_type = strcat('.', image_type);
767 end
768
769 image_path = random_file(path, image_type);
770 image = imread(image_path);
771
772 if ~quiet
773     idx = strfind(image_path, TL_DIR);
774     if isempty(idx)
775         idx = 1;
776     end
777     fprintf(1, 'image = %s\n', image_path(idx : end));
778 end
779 varargout{1} = image_path;
780 end
781
782
783 % looks for a directory |tl_dir| somewhere up from this file's location
784 % returns the absolute path to one of the directories in |tl_dir|/|branch_dir|
785 function path = random_dir(tl_dir, branch_dir)
786     cur_path = mfilename('fullpath');
787     tl_path = cur_path(1 : strfind(cur_path, tl_dir) + length(tl_dir));
788     branch_path = strcat(tl_path, branch_dir);
789     branch_path_contents = dir(branch_path);
790     branch_path_dirs = { };
791     for c = 1 : length(branch_path_contents)
792         elem = branch_path_contents(c);
793         if ~elem.isdir || strcmp(elem.name, '.') || strcmp(elem.name, '..')
794             continue;
795         end
796         branch_path_dirs{end + 1} = fullfile(branch_path, elem.name);
797     end
798     path = branch_path_dirs{randi([1 length(branch_path_dirs)])};
799 end
800
801

```

```

802 % returns a random file ending with |extension| found at |path|
803 function path = random_file(path, extension)
804     if nargin < 2
805         extension = '';
806     end
807
808     file_type = strcat('*. ', extension);
809     files = dir(fullfile(path, file_type));
810     filenames = {files.name};
811
812     path = fullfile(path, filenames{randi([1 length(filenames)])});
813 end
814 %%%%%%%%% end of file: random_image.m %%%%%%%%%
815
816
817 %%%%%%%%% file: run_on_data.m %%%%%%%%%
818 clear all;
819 clc;
820
821 if ~exist('TL_DIR', 'var')
822     TL_DIR = 'ug3_Vision';
823 end
824 if ~exist('IN_DIR', 'var')
825     IN_DIR = 'data';
826 end
827 if ~exist('FILTER_FN', 'var')
828     FILTER_FN = 'analyse_image';
829 end
830 if ~exist('OUT_DIR', 'var')
831     OUT_DIR = fullfile('res', strrep(FILTER_FN, '_', '-'));
832 end
833 filter_fn = str2func(FILTER_FN);
834 disp(sprintf('function: %s\ninput: %s\n', FILTER_FN, fullfile(TL_DIR, IN_DIR)));
835
836 curpath = mfilename('fullpath');
837 tpath = curpath(1 : strfind(curpath, TL_DIR) + length(TL_DIR));
838 inpath = strcat(tpath, IN_DIR);
839 outpath = strcat(tpath, OUT_DIR);
840 inpath_contents = dir(inpath);
841 inpath_dirs = {};
842 for c = 1 : length(inpath_contents)
843     elem = inpath_contents(c);
844     if ~elem.isdir || strcmp(elem.name, '.') || strcmp(elem.name, '..')
845         continue;
846     end
847     inpath_dirs{end + 1} = fullfile(inpath, elem.name);
848 end
849
850 num_dirs = length(inpath_dirs);
851 times = [];

```



```

852 for c = 1 : num_dirs
853     in_dir = inpath_dirs{c};
854     files = dir(strcat(in_dir, filesep, '*.jpg'));
855     file_names = {files.name};
856     out_dir = fullfile(outpath, strcat(IN_DIR, '-', num2str(c)));
857     if ~exist(out_dir, 'dir')
858         mkdir(out_dir);
859     end
860     num_files = length(file_names);
861     for d = 1 : num_files
862         file_name = file_names{d};
863         input = fullfile(in_dir, file_name);
864         disp(sprintf('[dir %d/%d] [file %d/%d]', c, num_dirs, d, num_files));
865         disp(sprintf('\tinput = %s', input(strfind(input, TL_DIR) : end)));
866         image = imread(input);
867         timer = tic;
868         mask = filter_fn(image);
869         elapsed = toc(timer);
870         times(end + 1) = elapsed;
871         output = fullfile(out_dir, file_name);
872         image = overlay_mask(image, mask, 'Saturation', 1);
873         imwrite(image, output, 'jpg');
874         disp(sprintf('\toutput = %s', output(strfind(input, TL_DIR) : end)));
875         disp(sprintf('\tprocessing time = %fs', elapsed));
876     end
877 end
878
879 disp(sprintf('\naverage processing time per image: %fs', mean(times)));
880 %%%%%%%%% end of file: run_on_data.m %%%%%%%%%

```