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. Difculties 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 whithin 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 $c_1, c_2, \ldots$, 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.

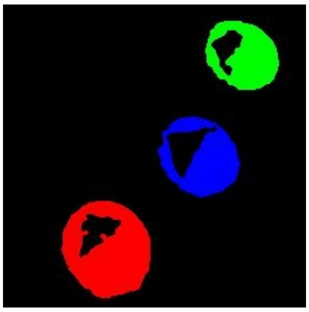


Figure 1: Result of Robot detection

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 ith 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: $P = \{p_{ij} | p_{ij} \in \xi \wedge M(i, j, \xi) = 1\}$

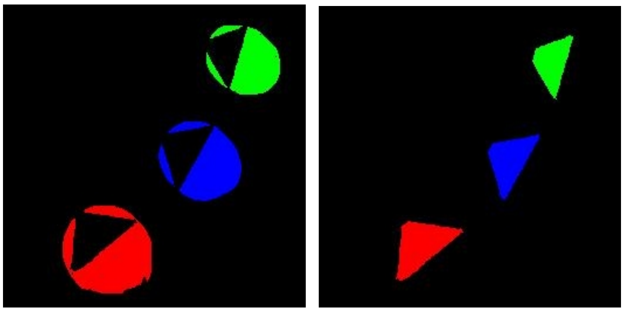
3. Calculate μ, the average rgb-value over P. Generate $\Pi = \{p | p \in P \wedge rgbvalue(p) < \mu\}$, the set of pixels in P that have a below-average rgb value.

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.



(a) After Removing Triangles (b) Removed Triangles

Figure 2: Triangle Detection via Local Thresholding

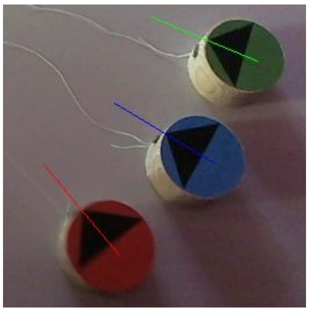


Figure 3: Detected Directions

2.3 Tracking Robots Over a Sequences of Frames

Input

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

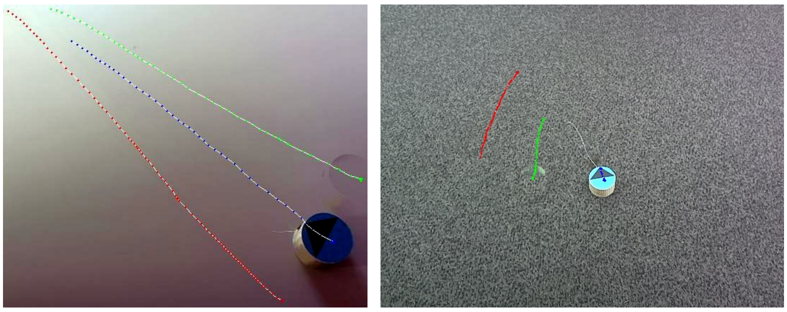
Output

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 $\Omega(i,j) = median(\omega_{ij})$.
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.



(a) Dataset 1 (b) Dataset 2

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

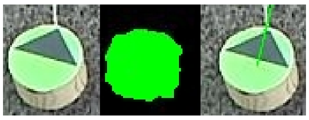
3.1 Detection of Robots

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 occurences of the blue robot being undetected and $\sim$40% of the occurences 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.

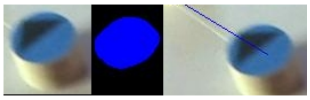
3.2 Detection of Directions

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.



(a) Loose hull detection

(b) Hull under detection



(c) Good hull detection

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, Hugh 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.

```
function res = main(path, image_type, start_offset, time_step)
    if nargin < 2
        image_type = 'jpg';
    end
    if nargin < 3
        start_offset = 0;
    end
    if nargin < 4
        time_step = 0.33;
    end

    files = dir(sprintf('%s/*.%s', path, image_type));
    filenames = {files.name};
    [~, num_files] = size(filenames);

    [m n ~] = size(imread(sprintf('%s/%s', path, filenames{1})));
    track_mask = zeros(m, n);

    rc = zeros(num_files - start_offset, 2);
    gc = zeros(num_files - start_offset, 2);
    bc = zeros(num_files - start_offset, 2);

    reds = cell(num_files - start_offset, 1);
    greens = cell(num_files - start_offset, 1);
    blues = cell(num_files - start_offset, 1);

    for i = 1 + start_offset : num_files
        image = imread(sprintf('%s/%s', path, filenames{i}));
        [directions, centroids] = analyse_image(image);

        r = uint16(centroids(1,:));
        if r(1) > 0 && r(2) > 0
            track_mask = overlay_cross(track_mask, 1, r(1), r(2));
            rc(i,:) = [r(1), r(2)];
        end

        g = uint16(centroids(2,:));
        if g(1) > 0 && g(2) > 0
            track_mask = overlay_cross(track_mask, 2, g(1), g(2));
            gc(i,:) = [g(1), g(2)];
        end

  b = uint16(centroids(3,:));
        if b(1) > 0 && b(2) > 0
            track_mask = overlay_cross(track_mask, 3, b(1), b(2));
            bc(i,:) = [b(1), b(2)];
        end

        reds{i} = image(:,:,1);
        greens{i} = image(:,:,2);
        blues{i} = image(:,:,3);
        imshow(overlay_mask(image, directions));

        pause(time_step);
    end

    rc = rc(any(rc,2),:);
    gc = gc(any(gc,2),:);
    bc = bc(any(bc,2),:);

    red_median = median(cat(3, reds{:}), 3);
    green_median = median(cat(3, greens{:}), 3);
    blue_median = median(cat(3, blues{:}), 3);

```

```
    bg = cat(3, red_median, green_median, blue_median);

    bg = overlay_polygon(bg, rc, [255, 255, 255]);
    bg = overlay_polygon(bg, gc, [255, 255, 255]);
    bg = overlay_polygon(bg, bc, [255, 255, 255]);
    bg = overlay_mask(bg, track_mask);
    imshow(bg)
end


function [pretty_mask, varargout] = analyse_image(image)
    [num_rows, num_cols, num_channels] = size(image);
    blob_mask = mask_colors(image);
    [convex_mask, ~] = mask_convex_regions(image, blob_mask);
    [convex_mask]=demask_triangles(image, convex_mask, false);
    [convex_mask, ~] = mask_convex_regions(image, convex_mask);
    [~, centroids, ~, triangle_centroids] = ...
                                demask_triangles(image, convex_mask, true);
    pretty_mask = overlay_rays(zeros(num_rows,num_cols,num_channels),...
                                centroids,triangle_centroids, 99, ...
                               'Color', [1 0 0; 0 1 0; 0 0 1]);
    varargout{1} = centroids;
    varargout{2} = triangle_centroids;
    varargout{3} = convex_mask;
end


% this function can used to find both - triangles and masks without
% triangles. To find triangles it must be run with
% get_triangle_centroids=true. Also application of the function int the
% following fashion gives better results.
%    [convex_mask, ~] = mask_convex_regions(image, blob_mask);
%    [convex_mask]=demask_triangles(image, convex_mask, false);
%    [convex_mask, ~] = mask_convex_regions(image, convex_mask);
function [mask, varargout] = demask_triangles(image, mask, ...
                                              get_triangle_centroids)
    [num_rows, num_cols, num_channels] = size(image);
    centroids = zeros(num_channels, 2);
    triangle_centroids  = zeros(num_channels, 2);
    trinagle_mask = mask;
    for c = 1 : num_channels
        channel = image(:,:,c);
        channel_mask = mask(:,:,c);
        channel_trinagle_mask =  mask(:,:,c);
        if ~any(channel_mask)
            continue;
        end
        rgb_values = zeros(sum(channel_mask(:)), 1);
        idx = 1;
        for nr = 1 : num_rows
            for nc = 1 : num_cols
                if channel_mask(nr, nc) == 0
                    continue;
                end
                rgb_values(idx) = channel(nr, nc);
                idx = idx + 1;
            end
        end
        mean_rgb = mean(rgb_values(:));
        channel_mask(channel < mean_rgb) = 0;
        mask(:,:,c) = channel_mask;
        props = regionprops(channel_mask, 'Centroid');
        centroid = props.Centroid;
        centroids(c,:) = [centroid(2), centroid(1)];
        if get_triangle_centroids
```

```
            channel_trinagle_mask(channel > mean_rgb) = 0;
            channel_trinagle_mask=filter_mask(channel_trinagle_mask);
            trinagle_mask(:,:,c) = channel_trinagle_mask;
            props = regionprops(channel_trinagle_mask, 'Centroid');
            centroid = props.Centroid;
            triangle_centroids(c,:) = [centroid(2), centroid(1)];
        end
    end
    varargout{1} = centroids;
    varargout{2} = trinagle_mask;
    varargout{3} = triangle_centroids;
end


function [color_mask, varargout] = mask_convex_regions(image, mask)
    [num_rows, num_cols, num_channels] = size(image);
    color_mask = zeros(num_rows, num_cols, num_channels);
    convex_centroids = zeros(num_channels, 2);
    for c = 1 : num_channels
        channel = mask(:,:,c);
        if ~any(channel(:))
            continue;
        end
        props = regionprops(channel, 'Centroid', 'ConvexImage', 'BoundingBox');
        convex_props = regionprops(props.ConvexImage, 'Centroid');
        convex_centroid = convex_props.Centroid;
        convex_centroid = [convex_centroid(2) + props.BoundingBox(2), ...
                           convex_centroid(1) + props.BoundingBox(1)];
        convex_centroids(c,:) = convex_centroid;
        convex_image = props.ConvexImage;
        [num_rows_convex, num_cols_convex] = size(convex_image);
        for row = 1 : num_rows_convex
            for col = 1 : num_cols_convex
                if convex_image(row, col) == 1
                    newrow = round(row + props.BoundingBox(2));
                    newcol = round(col + props.BoundingBox(1));
                    color_mask(newrow, newcol, c) = 1;
                end
            end
        end
    end
    varargout{1} = convex_centroids;
end


function image_mask = mask_colors(image)
    [num_rows, num_cols, ~] = size(image);
    num_pixels = num_rows * num_cols;
    image_mask = zeros(num_pixels, 3);
    rgb = double(reshape(image, num_pixels, 3));
    rgbN = double(reshape(normalise_rgb(image, 'approximate'), num_pixels, 3));
    rN_sdev = std(rgbN(:,1));
    gN_sdev = std(rgbN(:,2));
    bN_sdev = std(rgbN(:,3));
    rN_mean = mean(rgbN(:,1));
    gN_mean = mean(rgbN(:,2));
    bN_mean = mean(rgbN(:,3));
    hsv = reshape(rgb2hsv(image), num_pixels, 3);
    for c = 1 : num_pixels
        rN = rgbN(c,1);
        gN = rgbN(c,2);
        bN = rgbN(c,3);
        hue = hsv(c,1) * 360;
        % current pixel is red
        if      (hue >= 330 || hue <= 30) && ...
```

```
                (normal_prob(rN, rN_mean, rN_sdev) <  0.001)
                    image_mask(c,1) = 1;
        % current pixel is green
        elseif  (hue >= 80 && hue < 180) && ...
                (normal_prob(gN, gN_mean, gN_sdev) <  0.007)
                   image_mask(c,2) = 1;
        % current pixel is blue
        elseif  (hue >= 150 && hue <= 270) && ...
                (normal_prob(bN, bN_mean, bN_sdev) < 0.0000085)
                   image_mask(c,3) = 1;
        end
    end
    image_mask = reshape(image_mask, num_rows, num_cols, 3);
    image_mask = remove_noise(image_mask);
    image_mask = remove_outliers(image_mask);
    image_mask = enforce_similar_channel_areas(image_mask);
end


function x = normal_prob(val, mu, sigma)
    x = 1.0 / (sigma * sqrt(2 * pi)) * exp(-(val - mu) ^ 2 / (2 * sigma ^ 2));
end


function image = remove_noise(image)
    [~, ~, num_channels] = size(image);
    for c = 1 : num_channels
        channel = image(:,:,c);
        channel = bwmorph(channel, 'majority', Inf);
        channel = bwmorph(channel, 'bridge', Inf);
        image(:,:,c) = channel;
    end
end


% finds the connected components in each channel of |image| and removes those
% that are far away from the centroid of the pixels in that channel
% here, 'far away' means more distant than |distance_proprtion_threshold| times
% the average distance of each connected component to the channel centroid
% this removes big areas of noise such as the big green blob inside of the black
% arrow of the red robot in data/1/00000006.jpg
function image = remove_outliers(image, distance_proportion_threshold)
    if nargin < 2
        distance_proportion_threshold = 0.5;
    end
    [~, ~, num_channels] = size(image);
    for c = 1 : num_channels
        channel = image(:,:,c);
        if ~any(channel(:))
            continue;
        end
        channel_properties = regionprops(channel, 'Centroid');
        channel_centroid = channel_properties.Centroid;
        regions = bwconncomp(channel);
        regions_properties = regionprops(regions, 'Centroid', 'PixelIdxList');
        regions_centroids = {regions_properties.Centroid};
        distances = cellfun(@(x) norm(x - channel_centroid), regions_centroids);
        mean_distance = mean(distances);
        for d = 1 : length(distances)
            if distances(d) > mean_distance * distance_proportion_threshold
                idx = regions_properties(d).PixelIdxList;
                channel(idx) = 0;
            end
        end
        image(:,:,c) = channel;
```

```
    end
end


% we know that the robots are all about the same size - we can thus remove any
% channels in the mask that have a much smaller area than the other channels
% this catches some problems like the shadow of the blue robot in
% data/1/00000095.jpg being detected as a red blob
function image = enforce_similar_channel_areas(image, area_proportion_threshold)
    if nargin < 2
        area_proportion_threshold = 0.5;
    end
    [~, ~, num_channels] = size(image);
    areas = zeros(num_channels, 1);
    for c = 1 : num_channels
        channel = image(:,:,c);
        if ~any(channel(:))
            continue;
        end
        region_props = regionprops(channel, 'Area');
        areas(c) = region_props.Area;
    end
    max_area = max(areas(areas > 0));
    for c = 1 : num_channels
        area = areas(c);
        if area == 0
            continue;
        end
        if  (area < max_area * area_proportion_threshold) || ...
            (area > max_area / area_proportion_threshold)
                image(:,:,c) = image(:,:,c) * 0;
        end
    end
end


% this filters out all conneceted regions but the biggest one
function mask = filter_mask(mask)
    [x, y, num_channels] = size(mask);
    for c = 1 : num_channels
        channel = zeros(x, y);
        channel = reshape(channel, x * y, 1);
        blob_info = bwconncomp(mask(:,:,c));
        blob_list = blob_info.PixelIdxList;
        [nrows, ncols] = cellfun(@size, blob_list);
        largest_blob_pixels = blob_list{find(nrows == max(nrows))};
        for i = 1 : max(nrows)
            channel(largest_blob_pixels(i)) = 1;
        end
        channel = reshape(channel, x, y);
        mask(:,:,c) = channel;
    end
end


% normalises the values of the red, green, and blue channels of |image| in order
% to eliminate illumination differences in the image
% formula used: {r, g, b} = {r, g, b} / sqrt(r^2 + g^2 + b^2)
% if 'approximate' is passed as an additional parameter, instead use
% {r, g, b} = {r, g, b} / (r + g + b)
% this is approximately two times faster than the exact normalisation
function normalised_image = normalise_rgb(image, varargin)
    approximate = ~isempty(find(strcmpi(varargin, 'approximate')));
    red = double(image(:,:,1));
    green = double(image(:,:,2));
```

```
    blue = double(image(:,:,3));
    if approximate
        euclid_rgb = red(:,:) + green(:,:) + blue(:,:);
    else
        euclid_rgb = sqrt(red(:,:).^2 + green(:,:).^2 + blue(:,:).^2);
    end
    red_norm = round(red(:,:) ./ euclid_rgb .* 255);
    green_norm = round(green(:,:) ./ euclid_rgb .* 255);
    blue_norm = round(blue(:,:) ./ euclid_rgb .* 255);
    % some pixels are absolute black (r = g = b = 0) which causes division by
    % zero errors during normalisation and NaN values in the normalised channels
    % need to filter these values out
    red_norm(isnan(red_norm)) = 0;
    green_norm(isnan(green_norm)) = 0;
    blue_norm(isnan(blue_norm)) = 0;
    red_norm = uint8(red_norm);
    green_norm = uint8(green_norm);
    blue_norm = uint8(blue_norm);
    normalised_image = cat(3, red_norm, green_norm, blue_norm);
end


% returns image with a cross centered on pixel (|x|, |y|) drawn in |channel|
% [0, 0] is the top left corner of the image
function image = overlay_cross(image, channel, x, y)
    [h w ~] = size(image);

    if channel == 1
        other1 = 2;
        other2 = 3;
    end
    if channel == 2
        other1 = 1;
        other2 = 3;
    end
    if channel == 3
       other1 = 1;
       other2 = 2;
    end

    if y + 1 < h
        image(x, y + 1, channel) = 1;
        image(x, y + 1, other1) = 0;
        image(x, y + 1, other2) = 0;
    end

    if y - 1 > 0
        image(x, y - 1, channel) = 1;
        image(x, y - 1, other1) = 0;
        image(x, y - 1, other2) = 0;
    end

    if x + 1 < w
        image(x + 1, y, channel) = 1;
        image(x + 1, y, other1) = 0;
        image(x + 1, y, other2) = 0;
    end

    if x - 1 > 0
        image(x - 1, y, channel) = 1;
        image(x - 1, y, other1) = 0;
        image(x - 1, y, other2) = 0;
    end

    image(x, y, channel) = 1;
```

```
    image(x, y, other1) = 0;
    image(x, y, other2) = 0;

end


% returns the result of putting |mask| onto |image|
% for each pixel that is set in some channel of |mask|, saturates the pixel in
% the equivalent channel of |image|
function image = overlay_mask(image, mask, varargin)
    % parse options
    argc = size(varargin, 2);
    c = 1;
    while c <= argc
        arg = varargin{c};
        if strcmpi(arg, 'GrayScale')
            grayscale = 1;
        elseif strcmpi(arg, 'Saturation')
            if c + 1 > argc
                error('Saturation option should be followed by a double');
            end
            saturation = varargin{c + 1};
            c = c + 1;
        elseif strcmpi(arg, 'Lightness')
            if c + 1 > argc
                error('Lightness option should be followed by a double');
            end
            lightness = varargin{c + 1};
            c = c + 1;
        end
        c = c + 1;
    end

    % modify background image
    if exist('grayscale', 'var')
        gray_image = rgb2gray(image);
        image = cat(3, gray_image, gray_image, gray_image);
    end
    if exist('saturation', 'var')
        hsv_image = rgb2hsv(image);
        hsv_image(:,:,2) = hsv_image(:,:,2) * saturation;
        image = hsv2rgb(hsv_image);
    end
    if exist('lightness', 'var')
        hsv_image = rgb2hsv(image);
        hsv_image(:,:,3) = hsv_image(:,:,3) * lightness;
        image = hsv2rgb(hsv_image);
    end

    % lay mask onto background image
    num_channels = size(image, 3);
    channels = 1 : num_channels;
    for c = 1 : num_channels
        channel = image(:,:,c);
        mask_pixels = find(mask(:,:,c) == 1);
        channel(mask_pixels) = 255;
        image(:,:,c) = channel;
        for d = setdiff(channels, c)
            channel = image(:,:,d);
            channel(mask_pixels) = 0;
            image(:,:,d) = channel;
        end
    end
end
```

```
% returns image with lines drawn from the nth point in |points| to the n+1th
% [0, 0] is the top left corner of the image
function image = overlay_polygon(image, points, color)
    if nargin < 3
        color = [255 255 255];
    end

    points = round(points);

    num_channels = size(image, 3);
    for c = 1 : num_channels
        channel = image(:,:,c);
        channel = overlay_polygon_channel(channel, points, color(c));
        image(:,:,c) = channel;
    end
end


% Bresenham's line algorithm (simplified version)
% http://en.wikipedia.org/wiki/Bresenham's_line_algorithm#Simplification
function channel = overlay_polygon_channel(channel, points, color)
    [xmax, ymax] = size(channel);
    for c = 1 : length(points) - 1
        start = points(c,:);
        stop = points(c + 1,:);
        x0 = start(1);
        y0 = start(2);
        x1 = stop(1);
        y1 = stop(2);

        dx = abs(x1 - x0);
        dy = abs(y1 - y0);

        if x0 < x1
            sx = 1;
        else
            sx = -1;
        end
        if y0 < y1
            sy = 1;
        else
            sy = -1;
        end

        err = dx - dy;

        while 1
            if x0 > 0 && x0 <= xmax && y0 > 0 && y0 <= ymax
                channel(x0, y0) = color;
            end
            if x0 == x1 && y0 == y1
                break;
            end
            e2 = 2 * err;
            if e2 > -dy
                err = err - dy;
                x0 = x0 + sx;
            end
            if e2 < dx
                err = err + dx;
                y0 = y0 + sy;
            end
        end
    end
```

```
end


function image = overlay_rays(image, from, to, length, varargin)
    % parse options
    argc = size(varargin, 2);
    c = 1;
    while c <= argc
        arg = varargin{c};
        if strcmpi(arg, 'Color')
            if c + 1 > argc
                error('Color option should be followed by an integer tripplet');
            end
            colors = varargin{c + 1};
            c = c + 1;
        end
        c = c + 1;
    end
    % option defaults
    if ~exist('colors', 'var')
        colors = [255 255 255];
    end

    num_rays = size(from, 1);
    if size(colors, 1) == 1;
        colors = repmat(colors, num_rays, 1);
    end

    for c = 1 : num_rays
        if from(c,:) == to(c,:)
            continue;
        end
        x0 = from(c,1);
        y0 = from(c,2);
        x1 = to(c,1);
        y1 = to(c,2);
        dx = x0 - x1;
        dy = y0 - y1;
        lambda = min(sqrt(length ^ 2 / (dx ^ 2 + dy ^ 2)), ...
                    -sqrt(length ^ 2 / (dx ^ 2 + dy ^ 2)));

        x = x0 + lambda * dx;
        y = y0 + lambda * dy;

        image = overlay_polygon(image, [x0 y0; x y], colors(c,:));
    end
end
```