```
%%%%%%% file: main.m %%%%%%%
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
    _path = mfilename(’fullpath’);
    [_path, _, _] = fileparts(_path);
    addpath(fullfile(_path, ’algo’));
    addpath(fullfile(_path, ’draw’));
    addpath(fullfile(_path, ’test’));
    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
%%%%%%% file: analyse_image.m %%%%%%%
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
%%%%%%% end of file: analyse_image.m %%%%%%%
%%%%%%% file: median_filter.m %%%%%%%
% generates a background image from a set of sample images
% subtracting the background from the sample images eases object detection
% does not work on these datasets because the blue/cyan robot does not move
% for most of the images i.e. will be considered part of the background
function background = median_filter(path, image_type, start_offset, step)
    if nargin < 2
        image_type = ’jpg’;
    end
    if nargin < 3
        start_offset = 0;
    end
    dim = 2;
    files = dir(sprintf(’%s/*.%s’, path, image_type));
    filenames = {files.name};
    [~, num_files] = size(filenames);
    reds = cell(num_files - start_offset, 1);
    greens = cell(num_files - start_offset, 1);
    blues = cell(num_files - start_offset, 1);
    for c = 1 + start_offset :step: num_files
        image = imread(sprintf(’%s/%s’, path, filenames{c}));
        reds{c} = image(:,:,1);
        greens{c} = image(:,:,2);
        blues{c} = image(:,:,3);
    end
    red_median = median(cat(dim + 1, reds{:}), dim + 1);
    green_median = median(cat(dim + 1, greens{:}), dim + 1);
    blue_median = median(cat(dim + 1, blues{:}), dim + 1);
    background = cat(dim + 1, red_median, green_median, blue_median);
end
%%%%%%% end of file: median_filter.m %%%%%%%
%%%%%%% file: normalise_rgb.m %%%%%%%
```

```
% 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
%%%%%%% end of file: normalise_rgb.m %%%%%%%
%%%%%%% file: overlay_circles.m %%%%%%%
% draws the circles defined by the centres in |centers| and radiuses in |radii|
% onto |image| and returns the modified image
% [0, 0] is the top left corner of the image
% if |radii| is a number, draw all circles with that radius
function image = overlay_circles(image, centers, radii, 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
  end
  % option defaults
  if ~exist(’colors’, ’var’)
    colors = [255 255 255];
  end
  num_circles = size(centers, 1);
  if size(radii, 1) == 1
    radii = repmat(radii, num_circles, 1);
  end
  if size(colors, 1) == 1
    colors = repmat(colors, num_circles, 1);
  end
  centers = round(centers);
  radii = round(radii);
  [~, ~, num_channels] = size(image);
  for c = 1 : num_channels
    channel = image(:,:,c);
    channel = overlay_circles_channel(channel, centers, radii, colors(:,c));
    image(:,:,c) = channel;
  end
end
function channel = overlay_circles_channel(channel, centers, radii, colors)
  [xmax, ymax] = size(channel);
  num_circles = size(centers, 1);
  for c = 1 : num_circles
    Xc = centers(c,1);
    Yc = centers(c,2);
    radius = radii(c);
    for theta = 0 : 0.1 : 359
      x = round(Xc + radius * cos(theta));
      y = round(Yc + radius * sin(theta));
      if x <= xmax && x > 0 && y <= ymax && y > 0
        channel(x, y) = colors(c,:);
      end
    end
  end
end
%%%%%%% end of file: overlay_circles.m %%%%%%%
%%%%%%% file: overlay_cross.m %%%%%%%
% 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
%%%%%%% end of file: overlay_cross.m %%%%%%%
%%%%%%% file: overlay_mask.m %%%%%%%
% 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
%%%%%%% end of file: overlay_mask.m %%%%%%%
%%%%%%% file: overlay_polygon.m %%%%%%%
% 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
%%%%%%% end of file: overlay_polygon.m %%%%%%%
%%%%%%% file: overlay_rays.m %%%%%%%
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
%%%%%%% end of file: overlay_rays.m %%%%%%%
%%%%%%% file: random_image.m %%%%%%%
% returns a random image from some directory D and print the path to that image
% the function understands the following options:
```

```
%   ’Quiet’             don’t print the path to the image
%   ’ImageType’, C      look for images of type C (default = "jpg")
% any remaining parameters are taken to be the path to D
% if D is not specified, default to a random sub-directory of "ug3_Vision/data"
function [image, varargout] = random_image(varargin)
  TL_DIR = ’ug3_Vision’;
  BRANCH_DIR = ’data’;
  % parse options
  argc = length(varargin);
  c = 1;
  while c <= argc
    arg = varargin{c};
    if strcmpi(arg, ’Quiet’)
      quiet = 1;
    elseif strcmpi(arg, ’ImageType’)
      if c + 1 > argc
        error(’ImageType option should be followed by a string’);
      end
      image_type = varargin{c + 1};
      c = c + 1;
    else
      path = arg;
    end
    c = c + 1;
  end
  % option defaults
  if ~exist(’quiet’, ’var’)
    quiet = 0;
  end
  if ~exist(’image_type’, ’var’)
    image_type = ’.jpg’;
  end
  if ~exist(’path’, ’var’)
    path = random_dir(TL_DIR, BRANCH_DIR);
  end
  if ~strcmp(image_type(1), ’.’)
    image_type = strcat(’.’, image_type);
  end
  image_path = random_file(path, image_type);
  image = imread(image_path);
  if ~quiet
    idx = strfind(image_path, TL_DIR);
    if isempty(idx)
      idx = 1;
    end
    fprintf(1, ’image = %s\n’, image_path(idx : end));
  end
  varargout{1} = image_path;
end
% looks for a directory |tl_dir| somewhere up from this file’s location
% returns the absolute path to one of the directories in |tl_dir|/|branch_dir|
function path = random_dir(tl_dir, branch_dir)
    cur_path = mfilename(’fullpath’);
    tl_path = cur_path(1 : strfind(cur_path, tl_dir) + length(tl_dir));
    branch_path = strcat(tl_path, branch_dir);
    branch_path_contents = dir(branch_path);
    branch_path_dirs = {};
    for c = 1 : length(branch_path_contents)
        elem = branch_path_contents(c);
        if ~elem.isdir || strcmp(elem.name, ’.’) || strcmp(elem.name, ’..’)
            continue;
        end
        branch_path_dirs{end + 1} = fullfile(branch_path, elem.name);
    end
    path = branch_path_dirs{randi([1 length(branch_path_dirs)])};
end
% returns a random file ending with |extension| found at |path|
function path = random_file(path, extension)
    if nargin < 2
        extension = ’’;
    end
    file_type = strcat(’*’, extension);
    files = dir(fullfile(path, file_type));
    filenames = {files.name};
    path = fullfile(path, filenames{randi([1 length(filenames)])});
end
%%%%%%%% end of file: random_image.m %%%%%%%%
%%%%%%%% file: run_on_data.m %%%%%%%%
clear all;
clc;
if ~exist(’TL_DIR’, ’var’)
    TL_DIR = ’ug3_Vision’;
end
if ~exist(’IN_DIR’, ’var’)
    IN_DIR = ’data’;
end
if ~exist(’FILTER_FN’, ’var’)
    FILTER_FN = ’analyse_image’;
end
if ~exist(’OUT_DIR’, ’var’)
    OUT_DIR = fullfile(’res’, strrep(FILTER_FN, ’_’, ’-’));
end
filter_fn = str2func(FILTER_FN);
disp(sprintf(’function: %s\ninput: %s\n’, FILTER_FN, fullfile(TL_DIR, IN_DIR)));
curpath = mfilename(’fullpath’);
tlpath = curpath(1 : strfind(curpath, TL_DIR) + length(TL_DIR));
inpath = strcat(tlpath, IN_DIR);
outpath = strcat(tlpath, OUT_DIR);
inpath_contents = dir(inpath);
inpath_dirs = {};
```

```
for c = 1 : length(inpath_contents)
    elem = inpath_contents(c);
    if ~elem.isdir || strcmp(elem.name, ’.’) || strcmp(elem.name, ’..’)
        continue;
    end
    inpath_dirs{end + 1} = fullfile(inpath, elem.name);
end
num_dirs = length(inpath_dirs);
times = [];
for c = 1 : num_dirs
    in_dir = inpath_dirs{c};
    files = dir(strcat(in_dir, filesep, ’*.jpg’));
    file_names = {files.name};
    out_dir = fullfile(outpath, strcat(IN_DIR, ’-’, num2str(c)));
    if ~exist(out_dir, ’dir’)
        mkdir(out_dir);
    end
    num_files = length(file_names);
    for d = 1 : num_files
        file_name = file_names{d};
        input = fullfile(in_dir, file_name);
        disp(sprintf(’[dir %d/%d] [file %d/%d]’, c, num_dirs, d, num_files));
        disp(sprintf(’\tinput = %s’, input(strfind(input, TL_DIR) : end)));
        image = imread(input);
        timer = tic;
        mask = filter_fn(image);
        elapsed = toc(timer);
        times(end + 1) = elapsed;
        output = fullfile(out_dir, file_name);
        image = overlay_mask(image, mask, ’Saturation’, 1);
        imwrite(image, output, ’jpg’);
        disp(sprintf(’\toutput = %s’, output(strfind(input, TL_DIR) : end)));
        disp(sprintf(’\tprocessing time = %fs’, elapsed));
    end
end
disp(sprintf(’\naverage processing time per image: %fs’, mean(times)));
%%%%%%% end of file: run_on_data.m %%%%%%%
```