Assignment 2
CSE 5323 Computer Vision

Omar Abid
Oct. 19, 2015
211295573
Fall 2015
Professor: Minas Spetsakis

1. Play with Corners

In the first part of the assignment, two images from the same scene with slightly different positions were taken while ensuring nothing within the image moves more than a few pixels. A Harris Corner Detection Algorithm was used to find the corners as described below.

A Harris Corner Detection algorithm attempts to find large changes in intensity in both the X and Y direction by computing eigenvalues. The points of interest where the intensity change is large in both directions is considered to be a corner.

Mathematical Description of Harris Corner Detection:

1. Smooth the image using an isotropic Gaussian function, then convolve the derivative of the Gaussian with the image to get the x and y derivatives of the image.

$$Let\ G^{x}_{(\sigma)} = \frac{\partial G(x,y)}{\partial x}\ and\ G^{y}_{(\sigma)} = \frac{\partial G(x,y)}{\partial y} \quad (1)$$

$$where\ G(x,y) = Aexp(-(\frac{(x-y_0)^2}{2\sigma^2}) + (\frac{(x-y_0)^2}{2\sigma^2})) \quad (2)$$

   a. And the x and y derivatives of the image I_x, I_y.

$$I_x = \frac{\partial I}{\partial x} = G^{x}_{(\sigma)} * I \qquad I_y = \frac{\partial I}{\partial y} = G^{y}_{(\sigma)} * I \quad (3)$$

2. From this, we compute the squared derivative.

$$I^2_x = I_x.I_x \qquad I^2_y = I_y.I_y \qquad I_{xy} = I_x.I_y \quad (4)$$

3. Now we compute the Matrix R(x,y) for each pixel in the image.

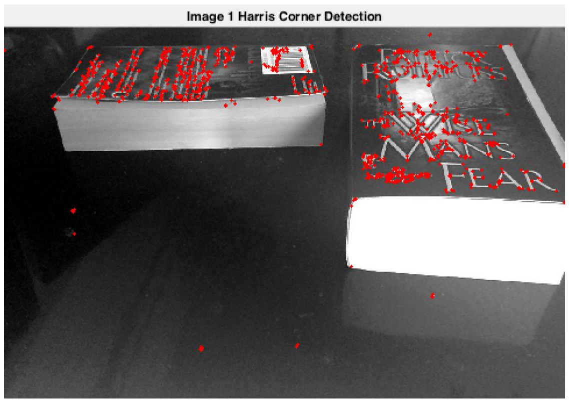
$$R(x,y) = [G(x,y) * I^2_x\ G(x,y) * I_{xy}\ G(x,y) * I_{xy}\ G(x,y) * I^2_y\ ] \quad (5)$$

4. We can now find the corners by thresholding for the smallest eigenvalue. For the assignment a Corner Response map J was computed for computational efficiency and J was chosen experimentally based on image output. Note that *k* is a constant determined experimentally, typically *k = 0.04-0.06.*

$$J = det\,(R) - k(trace(h))^2 \quad (6)$$

$$det\,(R) = \lambda_1\lambda_2 \quad and\ \ trace(R) = \lambda_1 + \lambda_2 \quad (7)$$

5. Following this, a builtin nonmax suppression function was run to eliminate a large number of points cluttering in a given corner of interest. Figure 1a shows the location of the detected points of interest and Figure 1b shows the corresponding image taken at a slightly different angle such that the image is a few pixels away than the original. Note that Figure 1b was generated after running the sum of squared differences to find the corresponding point.



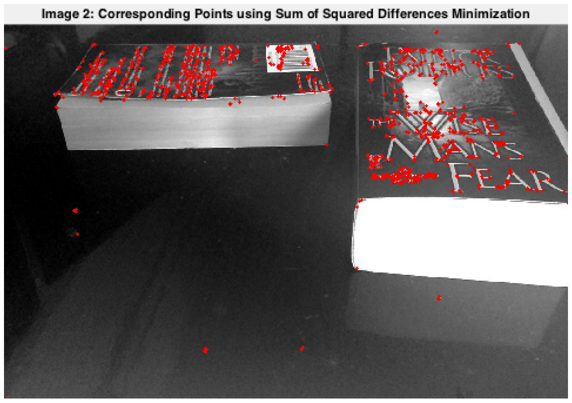


Image 2: Corresponding Points using Sum of Squared Differences Minimization

(a) *(b)*

Figure: 1a) Image 1 shows the original image with the corner points detected by the Harris Corner algorithm superimposed. Figure 1b) Image 2 taken at a slightly different angle calculated using the sum of squared differences equation (8).

To generate Figure 1b), a simple sum of squared differences was ran in order to find the corresponding point in the second image by searching a window [i-w,j -w],[i+w,j+w]. Where [i,j] is a point of interest (a place where a corner was detected in image 1, and *'w'* is the width in pixels of the search window. In the assignment, this parameter was determined experimentally to be '*w=5*'.

The sum of squared differences is defined as (8).

$$S[i_2,j_2] = \sum_{i=-p}^{i=p} \sum_{j=-p}^{j=p} (I_1[i_1+i,j_1+i] - I_2[i_2+i,j_2+i])^2 \qquad (8)$$

Where *p* defines the variable for the patch for which (8) is computed over, *(2p + 1)x(2p+1)*. And $[i_2,j_2]$ is corresponding point in the second image. Note that the value of $[i_2,j_2]$ is restricted in the range $i_1 - w \leq i_2 \leq i_1 + w \; and \; j_1 - w \leq j_2 \leq j_1 + w$. The value of $[i_2,j_2]$ within this range which minimizes S is chosen as the corresponding point in the second image since there is least variation here.

1.1 Implement a version of RANSAC

RANSAC or Random Sample Consensus is an iterative process of model fitting such as for fitting lines. In this assignment we use this algorithm to fit circles from a sample image.

The algorithm is described as such:

a) Run an edge detection algorithm.
   i. A canny edge detector taking an input image and returning a logical matrix the same size as the input image.

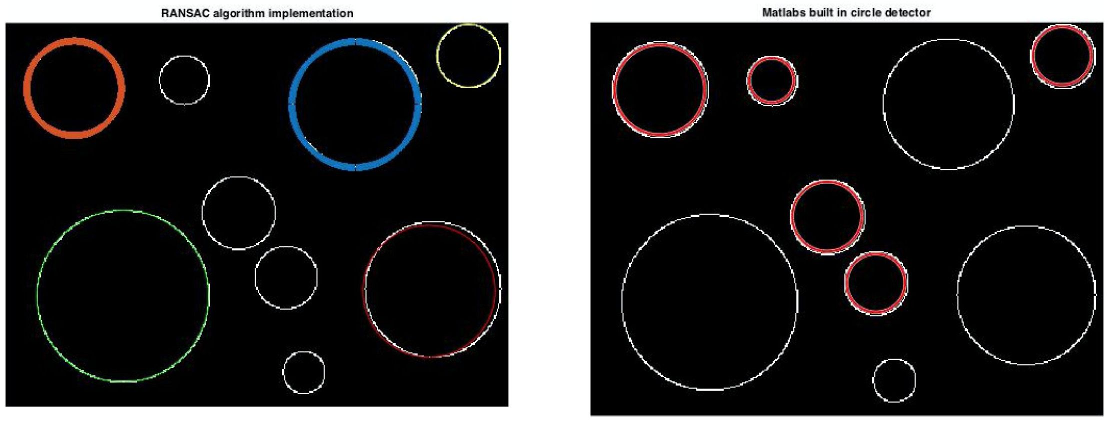
b) Select a triplet of distinct random points from the image returned from the edge detector algorithm.
    i. Generate a list of points given the returned image and store them in an array for use in part c).
c) Use a system of linear equations to fit and solve the equation of a circle fitting thru these points

$$(x_1 - h)^2 + (y_1 - k)^2 = r^2 \quad (1) \qquad (x_2 - h)^2 + (y_2 - k)^2 = r^2 \quad (2)$$
$$(x_3 - h)^2 + (y_3 - k)^2 = r^2 \quad (3)$$

Where (x_1,y_1), (x_2,y_2), (x_2,y_2) are the edge points detected from the canny edge detector. Note that Matlab's symbolic equation solver was used to solve the equations once in terms of h,k, and r, and these values calculated when necessary.

Also if the circle generated here was partially crossing past the boundaries of the image, it was discarded. Experimentally, out of N[100] times this was repeated, only 27 of those iterations when choosing random distinct points led to a circle within the boundaries of the image, hence this was an important step for efficiency.

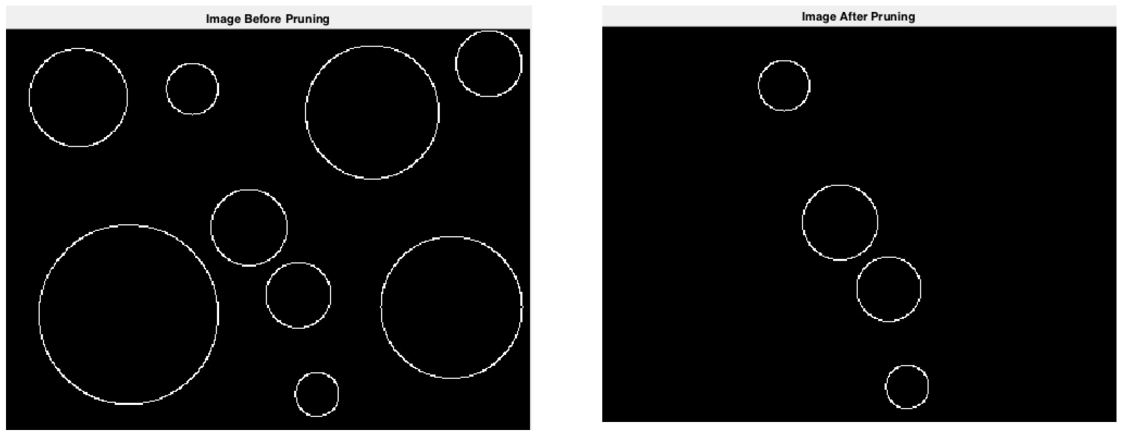
d) Count the number of points that are within W[3] pixels of the circle.
e) Steps (b)-(d) were repeated N[100] times and the circle that had the most points within W pixels (referred to as *vote_count* from now on) were chosen as the circle detected.
f) If the chosen circle from part (e) had *vote_count* < *threshold* then it was considered as noise and steps (b)-(e) were repeated again.
g) If the chosen circle had *vote_count* >= *threshold* then the points of the detected circle were removed and the procedure was repeated beginning with steps (b)-(g) to find C[5] circles.



(a) *(b)*

Figure 1.1a: (a) The RANSAC algorithm implementation done as an assignment, note that the largest circles are detected first since they have a higher probability of containing a point. (b) The built-in matlab circle detector, note that no clear

indication as to which algorithm was given. Note that only the first 5 circles detected were outputted.



(a) *(b)*

Figure 1.1b: (a) The original image used to generate Figure 1.1a and (b) the image after removing the detected circles (note this was done iteratively).

Appendix 1.1 – Code for ‘Playing with Corners’

```
%% Assignment 2 Question 1 Play With Corners

%% Get the images
rootDir = 'images/';

im_1 = imread([rootDir,'1.jpg']);
im_2 = imread([rootDir,'2.jpg']);
disp('Images successfully loaded');

%% Convert to Grayscale
im_1 = rgb2gray(im_1);
im_2 = rgb2gray(im_2);
[image_x_size,image_y_size] = size(im_1);
disp('Converted to Grayscale');

%% Compute Derivative of Gaussian given a sigma
sigma = 1;
hSize = [5 5]; % Gives the size


G1=fspecial('gauss',hSize, sigma);
[Gx,Gy] = gradient(G1);

%% Compute the x and y derivates of the image
disp('Computed the derivatives')
I_x = conv2(Gx,double(im_1));
I_y = conv2(Gy,double(im_1));

%% Compute Second Derivaties
I_x2 = I_x.*I_x;
I_y2 = I_y.*I_y;
I_xy = I_x.*I_y;

%% Compute sum of products of derivatives
I_x_2_average = conv2(G1,I_x2);
I_y_2_average = conv2(G1,I_y2);
I_xy_average = conv2(G1,I_xy);

%% Define the Image Gradient Magnitude and orientation
Image_Gradient_Magnitude = sqrt((I_x.^2)+(I_y.^2));
Image_Orientation = rad2deg(atan2(I_y,I_x));

%% Testing
I_x_2_average = I_x_2_average(5:end-4,5:end-4);
I_y_2_average = I_y_2_average(5:end-4,5:end-4);
I_xy_average = I_xy_average(5:end-4,5:end-4);

%% Let's define the matrix A = .... and compute matrix R
disp('Computing minimum eigenvalues at each pixel');
A = cell(image_x_size,image_y_size);
R = zeros(image_x_size, image_y_size);
kappa = 0.04; % Typically b/w 0.04-0.06
```

```
for i = 1:image_x_size
    for j = 1:image_y_size
        A{i,j} = [I_x_2_average(i,j) I_xy_average(i,j); I_xy_average(i,j)
I_y_2_average(i,j)];
        R(i,j) = det(A{i,j})-kappa*(trace(A{i,j}).^2);
    end
end

%% Set R = 0 as threshold
threshold = 5000;
for i = 1:image_x_size
    for j = 1:image_y_size
        if R(i,j) < threshold
            R(i,j) = 0;
        end
    end
end
disp(['Set R=',int2str(threshold),' as threshold'])

%% Add zeros around the borders for finding the non-suppression in 3x3
neighbourhood
x = zeros(image_x_size,1);
y = zeros(1,image_y_size+2);
R = computeNonMaxSuppression(R,Image_Gradient_Magnitude,
Image_Orientation);

disp('Finished finding the non maximimum suppression');

%% Set them as '1's and create a x and y vector matrix
x_point = [];
y_point = [];
for i = 1:image_x_size
    for j = 1:image_y_size
        if R(i,j) > 0
            R(i,j) = 1;
            x_point = [x_point i];
            y_point = [y_point j];

        end
    end
end
k = [x_point' y_point'];

%% Reduce the number of points by taking every 5th point
x_point = [];
y_point = [];
for i = 1:length(k)
    if (mod(i,3) == 0)
        x_point = [x_point k(i,1)];
        y_point = [y_point k(i,2)];
    end

end
```

```
disp('Store points in vector')


%% Calculate the sum of squared differences
w = 10; % Window width


%% Sum of squared differences
p = 4;
i_2 = zeros(length(x_point),1);
j_2 = zeros(length(y_point),1);
disp('Calculating SSD')

previous_percent_finished = 0;
num_corners = length(x_point);
count_interval = 25;

[tempXSize, tempYSize] = size(im_1);

for z = 1:num_corners  % For every corner do the following
    % Get the correspoding point (i1,j1) for the corner
    i_1 = x_point(z) ;
    j_1 = y_point(z) ;
    i_1 = min(i_1,tempXSize);
    j_1 = min(j_1,tempYSize);
    % Calculate the search window
    i_2_temp = max(i_1-w,1):1:min(i_1+w,tempXSize);
    j_2_temp = max(j_1-w,1):1:min(j_1+w,tempYSize);
    % Go over this window, each (a,b) is a corresponding (i2,j2) point
    % we must pick the one that minimizes S
    k = 1;
    search_area = [];
    S = zeros(length(i_2_temp)*length(j_2_temp),1);
    for a =i_2_temp
        for b =j_2_temp
            for i = -p:1:p
                for j = -p:1:p
                    if ((a+i)>0 && (a+i) < tempXSize && (b+j)>0 && (b+j)
< tempYSize && (i_1+i)>0 && (i_1+i) < tempXSize && (j_1+j)>0 && (j_1+j) <
tempYSize)
                        S(k) = S(k) + (double(im_1(i_1+i,j_1+j)) -
double(im_2(a+i,b+j))).^2;
                    end
                end
            end
            search_area = [search_area; a b];
            k = k + 1;
        end
    end

    [~,ind_value] = min(S);
    % The one that
    i_2(z) = search_area(ind_value,1);
    j_2(z) = search_area(ind_value,2);

    percent_finished = uint8((z/num_corners)*100);
    if ((percent_finished - previous_percent_finished) > count_interval)
        disp([int2str(percent_finished) '% Completed']);
        previous_percent_finished = percent_finished;
    end

end
disp('Finished SSD Calculation');



%% Plot the original and the corresponding image
subplot(1,2,1)
imshow(im_1)
title ('Image 1 Harris Corner Detection')
hold on; % Prevent image from being blown away.
plot(y_point,x_point,'r+', 'MarkerSize', 2);
subplot(1,2,2)
imshow(im_2)
title('Image 2: Corresponding Points using Sum of Squared Differences
Minimization')
hold on; % Prevent image from being blown away.
plot(j_2,i_2,'r+', 'MarkerSize', 2);
```

Appendix 1.2: Auxillary Function computeNonMaxSuppression used for the corner detection above.

```
function R = computeNonMaxSuppression(R_in,Image_Gradient_Magnitude,
Image_Orientation)
  E_W = 1;
  N_S = 2;
  NW_SE = 3;
  NE_SW = 4;
  R = R_in;
  [sizex, sizey] = size(Image_Orientation);
  for i = 1:sizex
      for j = 1:sizey
          if (Image_Orientation(i,j) >= -22.5 && Image_Orientation(i,j)
<= 22.5 ||...
              Image_Orientation(i,j) >= -180 && Image_Orientation(i,j) <=
-157.5 ||...
              Image_Orientation(i,j) >= 157.5 && Image_Orientation(i,j)
<= 180)
                  Image_Orientation(i,j) = E_W;
          elseif (Image_Orientation(i,j) >= 67.5 &&
Image_Orientation(i,j) <= 122.5 ||...
              Image_Orientation(i,j) >= -122.5 && Image_Orientation(i,j)
<= 67.5 )
                  Image_Orientation(i,j) = N_S;
              elseif (Image_Orientation(i,j) >= 112.5 &&
Image_Orientation(i,j) <= 157.5 ||...
              Image_Orientation(i,j) >= -67.5 && Image_Orientation(i,j)
<= 22.5 )
                Image_Orientation(i,j) = NW_SE;
              elseif (Image_Orientation(i,j) >= 22.5 &&
Image_Orientation(i,j) <= 67.5 ||...
              Image_Orientation(i,j) >= -157.5 && Image_Orientation(i,j)
<= -112.5 )
                Image_Orientation(i,j) = NE_SW;
          end
      end
  end

  for i = 1:sizex
      for j = 1:sizey
          if (Image_Orientation(i,j) == E_W)
              a = max(i-1,1);
              b = min(i+1,sizex);

              if ~(Image_Gradient_Magnitude(i,j) >=
Image_Gradient_Magnitude(a,j) && ...
                      Image_Gradient_Magnitude(i,j ) >=
Image_Gradient_Magnitude(b,j) )
                          R(i,j) = 0;
              end
          elseif (Image_Orientation(i,j) == N_S)
              a = max(j-1,1);
              b = min(j+1,sizey);

              if ~(Image_Gradient_Magnitude(i,j) >=
Image_Gradient_Magnitude(i,a) && ...
                      Image_Gradient_Magnitude(i,j ) >=
Image_Gradient_Magnitude(i,b) )
                          R(i,j) = 0;
              end
          elseif (Image_Orientation(i,j) == NW_SE)
              a1 = max(i-1,1);
              b1 = min(i+1,sizex);
              a2 = max(j-1,1);
              b2 = min(j+1,sizey);

              if ~(Image_Gradient_Magnitude(i,j) >=
Image_Gradient_Magnitude(a1,a2) && ...
                      Image_Gradient_Magnitude(i,j ) >=
Image_Gradient_Magnitude(b1,b2) )
                          R(i,j) = 0;
              end
          elseif (Image_Orientation(i,j) == NE_SW)
              a1 = max(i-1,1);
              b1 = min(i+1,sizex);
              a2 = min(j+1,sizey);
              b2 = max(j-1,1);

              if ~(Image_Gradient_Magnitude(i,j) >=
Image_Gradient_Magnitude(a1,a2) && ...
                      Image_Gradient_Magnitude(i,j ) >=
Image_Gradient_Magnitude(b1,b2) )
                          R(i,j) = 0;
              end
          end
      end
  end

end
```

Appendix 2.1 : Implementation a version of RANSAC

```
%% Question 2
rootDir = 'images/';
%rootDir = '';
im_1 = imread([rootDir,'circles_stage.png']);

im_1 = rgb2gray(im_1);
[x_size, y_size] = size(im_1);
im_4 = edge(im_1,'canny');
%im_4 = im_4(20:end-10,20:end-150);
im_4 = im_4(10:end-10,20:end-10);
im_1_original = im_4;

[x_size, y_size] = size(im_4);
imshow(im_4)

%% We will detect 5 circles
num_circles_to_find = 5;
my_circles = cell(num_circles_to_find,1);
c = 0;
while c < num_circles_to_find

    %% generate points list
    x_points =[];
    y_points = [];

    for i = 1:x_size
        for j = 1:y_size
            if (im_4(i,j))
                x_points = [x_points i];
                y_points = [y_points j];
            end
        end
    end


    %% Generate Distinct Points
    n=length(x_points);
    N = 100;
    r = cell(N,1);
    m=3; % Generate 3 distinct numbers
    for i = 1:N
        k=randperm(n);
        r{i}=k(1:m);
    end

    %% Generate N[100] points and their equations

    var_h = zeros(N,1);
    var_k = zeros(N,1);
    var_r = zeros(N,1);

    for i = 1:N
        %for j = 1:N
            k = r{i};

            xp_1 = x_points(k(1));
            yp_1 = y_points(k(1));
            xp_2 = x_points(k(2));
            yp_2 = y_points(k(2));
            xp_3 = x_points(k(3));
            yp_3 = y_points(k(3));


            % Fit a circle b/w the points

            var_h(i) = (xp_1^2*yp_2 - xp_1^2*yp_3 - xp_2^2*yp_1 +
xp_2^2*yp_3 + xp_3^2*yp_1 - xp_3^2*yp_2 + yp_1^2*yp_2 - yp_1^2*yp_3 -
yp_1*yp_2^2 + yp_1*yp_3^2 + yp_2^2*yp_3 - yp_2*yp_3^2)/(2*(xp_1*yp_2 -
xp_2*yp_1 - xp_1*yp_3 + xp_3*yp_1 + xp_2*yp_3 - xp_3*yp_2));
            var_k(i) = (- xp_1^2*xp_2 + xp_1^2*xp_3 + xp_1*xp_2^2 -
xp_1*xp_3^2 + xp_1*yp_2^2 - xp_1*yp_3^2 - xp_2^2*xp_3 + xp_2*xp_3^2 -
xp_2*yp_1^2 + xp_2*yp_3^2 + xp_3*yp_1^2 - xp_3*yp_2^2)/(2*(xp_1*yp_2 -
xp_2*yp_1 - xp_1*yp_3 + xp_3*yp_1 + xp_2*yp_3 - xp_3*yp_2));
            var_r(i) = -((xp_1^2*yp_2^2 - 2*xp_1^2*yp_2*yp_3 +
xp_1^2*yp_3^2 - 2*xp_1*xp_2*yp_1*yp_2 + 2*xp_1*xp_2*yp_1*yp_3 +
2*xp_1*xp_2*yp_2*yp_3 - 2*xp_1*xp_2*yp_3^2 + 2*xp_1*xp_3*yp_1*yp_2 -
2*xp_1*xp_3*yp_1*yp_3 - 2*xp_1*xp_3*yp_2^2 + 2*xp_1*xp_3*yp_2*yp_3 +
xp_2^2*yp_1^2 - 2*xp_2^2*yp_1*yp_3 + xp_2^2*yp_3^2 - 2*xp_2*xp_3*yp_1^2 +
2*xp_2*xp_3*yp_1*yp_2 + 2*xp_2*xp_3*yp_1*yp_3 - 2*xp_2*xp_3*yp_2*yp_3 +
xp_3^2*yp_1^2 - 2*xp_3^2*yp_1*yp_2 + xp_3^2*yp_2^2)*(xp_1^4*xp_2^2 -
2*xp_1^4*xp_2*xp_3 + xp_1^4*xp_3^2 + xp_1^4*yp_2^2 - 2*xp_1^4*yp_2*yp_3 +
xp_1^4*yp_3^2 - 2*xp_1^3*xp_2^3 + 2*xp_1^3*xp_2^2*xp_3 +
2*xp_1^3*xp_2*xp_3^2 - 2*xp_1^3*xp_2*yp_2^2 + 4*xp_1^3*xp_2*yp_2*yp_3 -
2*xp_1^3*xp_2*yp_3^2 - 2*xp_1^3*xp_3^3 - 2*xp_1^3*xp_3*yp_2^2 +
4*xp_1^3*xp_3*yp_2*yp_3 - 2*xp_1^3*xp_3*yp_3^2 + xp_1^2*xp_2^4 +
2*xp_1^2*xp_2^3*xp_3 - 6*xp_1^2*xp_2^2*xp_3^2 + 2*xp_1^2*xp_2^2*yp_1^2 -
2*xp_1^2*xp_2^2*yp_1*yp_2 - 2*xp_1^2*xp_2^2*yp_1*yp_3 +
2*xp_1^2*xp_2^2*yp_2^2 - 2*xp_1^2*xp_2^2*yp_2*yp_3 +
2*xp_1^2*xp_2^2*yp_3^2 + 2*xp_1^2*xp_2*xp_3^3 - 4*xp_1^2*xp_2*xp_3*yp_1^2
+ 4*xp_1^2*xp_2*xp_3*yp_1*yp_2 + 4*xp_1^2*xp_2*xp_3*yp_1*yp_3 +
2*xp_1^2*xp_2*xp_3*yp_2^2 - 8*xp_1^2*xp_2*xp_3*yp_2*yp_3 +
2*xp_1^2*xp_2*xp_3*yp_3^2 + xp_1^2*xp_3^4 + 2*xp_1^2*xp_3^2*yp_1^2 -
2*xp_1^2*xp_3^2*yp_1*yp_2 - 2*xp_1^2*xp_3^2*yp_1*yp_3 +
2*xp_1^2*xp_3^2*yp_2^2 - 2*xp_1^2*xp_3^2*yp_2*yp_3 +
2*xp_1^2*xp_3^2*yp_3^2 + 2*xp_1^2*yp_1^2*yp_2^2 -
4*xp_1^2*yp_1^2*yp_2*yp_3 + 2*xp_1^2*yp_1^2*yp_3^2 - 2*xp_1^2*yp_1*yp_2^3
+ 2*xp_1^2*yp_1*yp_2^2*yp_3 + 2*xp_1^2*yp_1*yp_2*yp_3^2 -
2*xp_1^2*yp_1*yp_3^3 + xp_1^2*yp_2^4 - 2*xp_1^2*yp_2^3*yp_3 +
2*xp_1^2*yp_2^2*yp_3^2 - 2*xp_1^2*yp_2*yp_3^3 + xp_1^2*yp_3^4 -
2*xp_1*xp_2^4*xp_3 + 2*xp_1*xp_2^3*xp_3^2 - 2*xp_1*xp_2^3*yp_1^2 +
4*xp_1*xp_2^3*yp_1*yp_3 - 2*xp_1*xp_2^3*yp_3^2 + 2*xp_1*xp_2^2*xp_3^3 +
2*xp_1*xp_2^2*xp_3*yp_1^2 + 4*xp_1*xp_2^2*xp_3*yp_1*yp_2 -
8*xp_1*xp_2^2*xp_3*yp_1*yp_3 - 4*xp_1*xp_2^2*xp_3*yp_2^2 +
4*xp_1*xp_2^2*xp_3*yp_2*yp_3 + 2*xp_1*xp_2^2*xp_3*yp_3^2 -
2*xp_1*xp_2*xp_3^4 + 2*xp_1*xp_2*xp_3^2*yp_1^2 -
8*xp_1*xp_2*xp_3^2*yp_1*yp_2 + 4*xp_1*xp_2*xp_3^2*yp_1*yp_3 +
2*xp_1*xp_2*xp_3^2*yp_2^2 + 4*xp_1*xp_2*xp_3^2*yp_2*yp_3 -
4*xp_1*xp_2*xp_3^2*yp_3^2 - 2*xp_1*xp_2*yp_1^2*yp_2^2 +
4*xp_1*xp_2*yp_1^2*yp_2*yp_3 - 2*xp_1*xp_2*yp_1^2*yp_3^2 +
4*xp_1*xp_2*yp_1*yp_2^2*yp_3 - 8*xp_1*xp_2*yp_1*yp_2*yp_3^2 +
4*xp_1*xp_2*yp_1*yp_3^3 - 2*xp_1*xp_2*yp_2^2*yp_3^2 +
4*xp_1*xp_2*yp_2*yp_3^3 - 2*xp_1*xp_2*yp_3^4 - 2*xp_1*xp_3^3*yp_1^2 +
4*xp_1*xp_3^3*yp_1*yp_2 - 2*xp_1*xp_3^3*yp_2^2 -
2*xp_1*xp_3*yp_1^2*yp_2^2 + 4*xp_1*xp_3*yp_1^2*yp_2*yp_3 -
2*xp_1*xp_3*yp_1^2*yp_3^2 + 4*xp_1*xp_3*yp_1*yp_2^3 -
8*xp_1*xp_3*yp_1*yp_2^2*yp_3 + 4*xp_1*xp_3*yp_1*yp_2*yp_3^2 -
2*xp_1*xp_3*yp_2^4 + 4*xp_1*xp_3*yp_2^3*yp_3 - 2*xp_1*xp_3*yp_2^2*yp_3^2
+ xp_2^4*xp_3^2 + xp_2^4*yp_1^2 - 2*xp_2^4*yp_1*yp_3 + xp_2^4*yp_3^2 -
2*xp_2^3*xp_3^3 - 2*xp_2^3*xp_3*yp_1^2 + 4*xp_2^3*xp_3*yp_1*yp_3 -
2*xp_2^3*xp_3*yp_3^2 + xp_2^2*xp_3^4 + 2*xp_2^2*xp_3^2*yp_1^2 -
2*xp_2^2*xp_3^2*yp_1*yp_2 - 2*xp_2^2*xp_3^2*yp_1*yp_3 +
2*xp_2^2*xp_3^2*yp_2^2 - 2*xp_2^2*xp_3^2*yp_2*yp_3 +
2*xp_2^2*xp_3^2*yp_3^2 + xp_2^2*yp_1^4 - 2*xp_2^2*yp_1^3*yp_2 -
2*xp_2^2*yp_1^3*yp_3 + 2*xp_2^2*yp_1^2*yp_2^2 + 2*xp_2^2*yp_1^2*yp_2*yp_3
+ 2*xp_2^2*yp_1^2*yp_3^2 - 4*xp_2^2*yp_1*yp_2^2*yp_3 +
2*xp_2^2*yp_1*yp_2*yp_3^2 - 2*xp_2^2*yp_1*yp_3^3 + 2*xp_2^2*yp_2^2*yp_3^2
- 2*xp_2^2*yp_2*yp_3^3 + xp_2^2*yp_3^4 - 2*xp_2*xp_3^3*yp_1^2 +
4*xp_2*xp_3^3*yp_1*yp_2 - 2*xp_2*xp_3^3*yp_2^2 - 2*xp_2*xp_3*yp_1^4 +
4*xp_2*xp_3*yp_1^3*yp_2 + 4*xp_2*xp_3*yp_1^3*yp_3 -
2*xp_2*xp_3*yp_1^2*yp_2^2 - 8*xp_2*xp_3*yp_1^2*yp_2*yp_3 -
2*xp_2*xp_3*yp_1^2*yp_3^2 + 4*xp_2*xp_3*yp_1*yp_2^2*yp_3 +
4*xp_2*xp_3*yp_1*yp_2*yp_3^2 - 2*xp_2*xp_3*yp_2^2*yp_3^2 + xp_3^4*yp_1^2
- 2*xp_3^4*yp_1*yp_2 + xp_3^4*yp_2^2 + xp_3^2*yp_1^4 -
2*xp_3^2*yp_1^3*yp_2 - 2*xp_3^2*yp_1^3*yp_3 + 2*xp_3^2*yp_1^2*yp_2^2 +
2*xp_3^2*yp_1^2*yp_2*yp_3 + 2*xp_3^2*yp_1^2*yp_3^2 - 2*xp_3^2*yp_1*yp_2^3
+ 2*xp_3^2*yp_1*yp_2^2*yp_3 - 4*xp_3^2*yp_1*yp_2*yp_3^2 + xp_3^2*yp_2^4 -
2*xp_3^2*yp_2^3*yp_3 + 2*xp_3^2*yp_2^2*yp_3^2 + yp_1^4*yp_2^2 -
2*yp_1^4*yp_2*yp_3 + yp_1^4*yp_3^2 - 2*yp_1^3*yp_2^3 +
2*yp_1^3*yp_2^2*yp_3 + 2*yp_1^3*yp_2*yp_3^2 - 2*yp_1^3*yp_3^3 +
yp_1^2*yp_2^4 + 2*yp_1^2*yp_2^3*yp_3 - 6*yp_1^2*yp_2^2*yp_3^2 +
2*yp_1^2*yp_2*yp_3^3 + yp_1^2*yp_3^4 - 2*yp_1*yp_2^4*yp_3 +
2*yp_1*yp_2^3*yp_3^2 + 2*yp_1*yp_2^2*yp_3^3 - 2*yp_1*yp_2*yp_3^4 +
yp_2^4*yp_3^2 - 2*yp_2^3*yp_3^3 + yp_2^2*yp_3^4))^(1/2)/(2*(xp_1^2*yp_2^2
- 2*xp_1^2*yp_2*yp_3 + xp_1^2*yp_3^2 - 2*xp_1*xp_2*yp_1*yp_2 +
2*xp 1*xp 2*yp 1*yp 3 + 2*xp 1*xp 2*yp 2*yp 3 - 2*xp 1*xp 2*yp 3^2 +
2*xp_1*xp_3*yp_1*yp_2 - 2*xp_1*xp_3*yp_1*yp_3 - 2*xp_1*xp_3*yp_2^2 +
2*xp_1*xp_3*yp_2*yp_3 + xp_2^2*yp_1^2 - 2*xp_2^2*yp_1*yp_3 +
xp_2^2*yp_3^2 - 2*xp_2*xp_3*yp_1^2 + 2*xp_2*xp_3*yp_1*yp_2 +
2*xp_2*xp_3*yp_1*yp_3 - 2*xp_2*xp_3*yp_2*yp_3 + xp_3^2*yp_1^2 -
2*xp_3^2*yp_1*yp_2 + xp_3^2*yp_2^2));

        %end
    end

    %% Remove those that are out of range
    equations_to_test = cell(N,1);
    count = 1;
    for i = 1:N
        x = var_r(i)*cosd(1:360)+var_h(i);
        y = var_r(i)*sind(1:360)+var_k(i);
        if ~(min(x) < 0 || max(x) > x_size || min(y) < 0 || max(y) >
y_size)
            equations_to_test{count} = [var_h(i) var_k(i) abs(var_r(i))];
            count = count + 1;
        end
    end
    count = count - 1;

    %% Now we go around all of our circles
    for z = 1:count
        %% Compute the x and y values
        x = equations_to_test{z}(3)*cosd(1:360)+equations_to_test{z}(1);
        y = equations_to_test{z}(3)*sind(1:360)+equations_to_test{z}(2);
        %% Set the range to give to the vote function
        x_range =
uint16(equations_to_test{z}(1)-equations_to_test{z}(3):1:equations_to_tes
t{z}(1)+equations_to_test{z}(3));
        y_range =
uint16(equations_to_test{z}(2)-equations_to_test{z}(3):1:equations_to_tes
t{z}(2)+equations_to_test{z}(3));
        %% Check around the x and y axis
        vote_count = zeros(count,1);
        W = 3; % Width to test.
        temp_sum = 0;
        for a = 1:length(x)
            temp_sum = temp_sum + voteCount(im_4,x(a),y(a),W);
        end
    end

    if temp_sum > 800
           c = c+1;
           vote_count(z) = temp_sum;
           [~, max_index] = max(vote_count);
           my_circles{c} =[equations_to_test{max_index}(1)
equations_to_test{max_index}(2) equations_to_test{max_index}(3)];
           im_4 = removePoints(im_4,my_circles{c});
    end
    disp(['count: ',int2str(c),' tempsum: ',int2str(temp_sum)]);
end

%% jklj
q =['r+','g+','y+','o+','b+'];
subplot(1,2,1)
imshow(im_1_original)
title ('RANSAC algorithm implementation')
hold on
for z = 1:num_circles_to_find
   % z = 3;
     x = my_circles{z}(3)*cosd(1:360)+my_circles{z}(1);
     y = my_circles{z}(3)*sind(1:360)+my_circles{z}(2);

     plot(y,x,q(z),'MarkerSize',5)
     hold on
end
subplot(1,2,2)
matlabCircleDetector(im_1_original);
title('Matlabs built in circle detector')

hold off
subplot(1,2,1)
imshow(im_1_original)
title('Image Before Pruning')
subplot(1,2,2)
imshow(im_4)
title('Image After Pruning')
```

Appendix 2.2 : Auxillary *voteCount* function for counting number of pixels that appear within a circle with a given radius and search window (W[3], chosen for this assignment).

```
function [count] = voteCount(data_in,x,y,w)
    count = 0;
    x = uint16(x);
    y = uint16(y);
    x_low = x-w;
    x_high = x+w;
    y_low = y-w;
    y_high = y+w;


    [x_size,y_size] = size(data_in);

    if (x_low <= 0)
        x_low = 1;
    end
    if (y_low <= 0)
        y_low = 1;
    end
    if (x_high >= x_size)
        x_high= x_size;
    end
    if (y_high >= y_size)
       y_high= y_size;
    end

    xrange = x_low:1:x_high;
    yrange = y_low:1:y_high;

    for i = xrange
        for j = yrange
            if (data_in(i,j))
                count = count + 1;
            end
        end
    end

end
```

Appendix 2.3: Auxillary function *removePoints*, to remove the points in the image which overlap with the detected circle. Used as a pruning process so that the same circle isn't detected more than once.

```
function image_out = removePoints(image_in,A)
    h = A(1);
    k = A(2);
    r = A(3);
    image_out = image_in;


    [x_size,y_size] = size(image_in);

    x = r*cosd(1:360)+h;
    y = r*sind(1:360)+k;
    x = uint16(x);
    y = uint16(y);

   w = 5;
    for i = 1:length(x)
        x_low = x(i) - w;
        x_high = x(i) + w;
        y_low = y(i) - w;
        y_high = y(i) + w;

        if (x_low <= 0)
            x_low = 1;
        end
        if (y_low <= 0)
            y_low = 1;
        end
        if (x_high >= x_size)
            x_high= x_size;
        end
        if (y_high >= y_size)
```

```
           y_high= y_size;
        end
        xrange = x_low:1:x_high;
        yrange = y_low:1:y_high;
        for a = xrange
            for b = yrange
                if (image_out(a,b))
                    image_out(a,b) = 0;
                end
            end
        end

    end
end
```

Appendix 2.4 : Auxillary function *matlabCircleDetector*, which uses matlab's builtin function to detect circles. Used to compare how well it works against the RANSAC algorithm implementation.

```
function matlabCircleDetector(image_in)
    A = image_in;
    [centers, radii] = imfindcircles(A,[15 70],'ObjectPolarity','dark');

    centers = centers(1:5,:,:);
    radii = radii(1:5);
    imshow(A)
    hold on
    h = viscircles(centers,radii);
end
```

References

Collins, R. (n.d.). Harris Corner Detector. Retrieved October 18, 2015, from http://www.cse.psu.edu/~rtc12/CSE486/lecture06.pdf

Forsyth, D., & Ponce, J. (2003). Computer vision: A modern approach. Upper Saddle River, N.J.: Prentice Hall.