```
clear; clc; close all;

%% Read the image from the folder and find the threshold using Otsu thresholding

load('camera_int from kuka camera.mat')
folder = 'Calibration Checker board/';
file = 'img_1';
extension = '.bmp';
ref_image_path = [folder,file,extension];

folder1 = 'new head position images/';
file1 = 'ninth_head';
extension1 = '.bmp';
test_image_path = [folder1,file1,extension1];
squareSize = 6;
use_same_size = 0;

%% Get the camera calibration values such as focal length, pricipal
%% axis points and skew in a matrix

% Generate intrinsic matrix with calibrated camera parameters
fx = cameraParams.FocalLength(1);
fy = cameraParams.FocalLength(2);
skew = cameraParams.Skew;
cx = cameraParams.PrincipalPoint(1);
cy = cameraParams.PrincipalPoint(2);
K = [fx,skew,cx;0,fy,cy;0,0,1];

%% Compute extrinsic parameters based on a reference image that is  image of a
%%calibration pattern on the screw holding tray

%read image
im = imread(ref_image_path);
% remove lens distortions
im_u = undistortImage(im, cameraParams);

% Find rotation and translation of World frame with respect to camera
% frame.
 [R, t] = compute_extrinsic_parameters(im, cameraParams, squareSize);
T = [R,t];

% Define transformation from measurement plane to the image plane (physical camera)
% homography from world to real image
H=K*[R(:,1:2),t];

%% Define transformation from measurement plane to the orthonormal image (virtual↙
image).
%% The virtual camera should be placed such that the principal ray observes the same↙
point M
%% as the real camera,and the image plane is parallel to the measurement plane.

% intrinsic parameters: generate intrinsic matrix for virtual camera

% set isotropic scale  fx'=fy'=f=(fx+fy)/2
```

```
f = (K(1,1)+K(2,2))/2.0;

K_new = K;
K_new(1,1) = f;
K_new(2,2) = f;

%% Extrinsic parameters: generate rotation and translation for virtual camera

% Visualize  image and world frame
view_extrinsics(H,squareSize,im);

% define rotation of world plane with respect to virtual fronto parallel camera
Xnew = [1;0;0];
Ynew = [0;1;0];
Znew = [0;0;1];
R_new = [Xnew,Ynew,Znew];

% Find the point on the measurement plane observed by the principal point
% and the camera-measurement-plane distance along the optical axis
[dsurf, P_w] = compute_pp_backprojection(H,cx,cy,T);

% virtual camera at same distance from P as physical one
tnew = -R_new*P_w+[0; 0; dsurf];

% homography from world to image of perpendicular view camera
HNew = K_new*[R_new(:,1),R_new(:,2),tnew];

%% compute scale of fronto parallel image (pix/mm)
scale_rect = f/tnew(3);

%% homography from image to rectified image
HRect = HNew*pinv(H);

%% APPLY HOMOGRAPHY TO TEST IMAGE
% WARNING homography Matrices need to be transposed before using them in Matlab↙
built-in functions!
%  The squares in the rectified image will appear as squares,
% and not as the rectangles observed in the original image. In the same way, the↙
circles in the rectified
% image appear as circles, and not as the ellipses observed in the original image.


%read image
colorImg = imread(test_image_path);
% image padding is done to obtain a image which is equal to the reference
% image dimensions that is 1088x1456 and if we subtract the given image dimenssions
% that is 452*452 then we get 636x1004 and half of this is 318x502 which is
%the zero padding around the image which will be at the centre
im_test= padarray(colorImg,[318 502],0,'both');
% remove lens distortions
im_test_u = undistortImage(im_test, cameraParams);

% use_same_size was stored during the homography computation
if(use_same_size)
```

```
    % define output image size and extents
    fixedView = imref2d(size(im_test));
    [im_test_rect, RB] = imwarp(im_test_u, projective2d((HRect')),'FillValues',↙
0,'OutputView',fixedView);
else
    [im_test_rect, RB] = imwarp(im_test_u, projective2d((HRect')),'FillValues', 0);
end

% visualize:
figure
subplot(1,2,1)
imshow(im_test_u)
hold on
title('Original image (undistorted) ')
hold off
subplot(1,2,2)
imshow(im_test_rect)
hold on
title('Perspective correction ')
hold off

set(gcf,'color','white');
fsize=get(gcf,'Position');
fsize(3) = 2*fsize(3);
set(gcf,'Position',fsize)


%% Binarize the above image

grayImg = rgb2gray(colorImg);
[counts,binLocations] = imhist(grayImg);
T = otsuthresh(counts);
%figure,imshow(grayImg);
Img = imbinarize(grayImg, T);
%figure,imshow(Img);

%% Detecting the diameter of the circular screws using Hough Circle transform
invImg = ~Img;
[centers,radii] = imfindcircles(invImg,[102 120], 'Sensitivity',↙
0.90,'EdgeThreshold', 0.03,'Method', 'TwoStage', 'ObjectPolarity', 'dark');
figure,imshow(invImg), hold on
viscircles(centers, radii);
len1 = size(centers,1);
count_Circles = 0;
for k = 1:len1
    plot(centers(k,1), centers(k,2), 'r+', 'MarkerSize', 1, 'LineWidth', 2);
    count_Circles = count_Circles+1;
end
Dia_Screw = 2*max(radii);

%% Hough line transform
if count_Circles == 0
    bwImg = imbinarize(grayImg);
    [edgeImg, threshOut] = edge(bwImg, 'Canny', 0.28,1.8);
    %figure,imshow(edgeImg);

    [Hmatrix, Theta, Rho] = hough(edgeImg, 'Theta', -90.00:7.00:83.00,↙
'RhoResolution', 2.32);
    Peaks = houghpeaks(Hmatrix, 2, 'threshold', 0.50, 'NHoodSize', [9 13]);
    lines = houghlines(edgeImg, Theta, Rho, Peaks, 'FillGap', 20, 'MinLength', 40);

    %%VISUALIZATION:
    figure;
    imshow(colorImg);
    hold on;
    for ii = 1:length(lines)
        xy = [lines(ii).point1; lines(ii).point2];
        plot(xy(:, 1), xy(:, 2), 'LineWidth', 2, 'Color', 'green');
    end
    dist_space =sqrt((lines(1).point2(2)-lines(2).point2(2))^2+(lines(1).point2(1)-↙
lines(2).point2(1))^2);
end


%% Results for the Screw head
 % here the above pixel values are converted into mm by scale_rect value
 % that is in pixel/mm

if count_Circles>0
    ScrewDiameter = Dia_Screw/scale_rect;
    sprintf("The diameter of the circular Screw head in mm  :%.2f ",ScrewDiameter)
else
    ScrewWidth = dist_space/scale_rect;
    sprintf("The width of the hexagon Screw head in mm:%.2f ",ScrewWidth)
end




%% Function space to the end

function [R, t] = compute_extrinsic_parameters(im_u, cameraParams, squareSize)

    % detect checkerboard corners
    [imagePoints, boardSize] = detectCheckerboardPoints(im_u);
    worldPoints = generateCheckerboardPoints(boardSize, squareSize);
    [R, t] = extrinsics(imagePoints,worldPoints,cameraParams);

    % WARNING rotation/homography Matrices obtained from Matlab need to be↙
transposed!
    R = R';
    t = t';
    % T = [R,t];
end

function [] = view_extrinsics(H,squareSize,im_u)
    Ow_c_h = H * [0,0,1]';
    Ow_c = Ow_c_h(1:2)/Ow_c_h(3);
```

```matlab
    % Xw vector in image plane
    x1_h = H * [squareSize*3,0,1]' ;
    x1 = x1_h(1:2)/x1_h(3);

    %Yw vector in image plane
    y1_h = H * [0,squareSize*3,1]' ;
    y1 = y1_h(1:2)/y1_h(3);

    % look at image and world axes to define rotation matrix
    figure
    imshow(im_u)
    hold on
    plot([Ow_c(1),x1(1)],[Ow_c(2),x1(2)],'r','LineWidth',4)
    plot([Ow_c(1),y1(1)],[Ow_c(2),y1(2)],'g','LineWidth',4)
    title('Undistorted image')
end



function [dsurf, P_w] = compute_pp_backprojection(H,cx,cy,T)
    % Back-projection of principal point from physical camera
    % to measurement plane (point P)
    % 2d homogeneous coordinates of Pw (on the plane)
    P_wh = pinv(H)*[cx,cy,1]';

    % 3D coordinates of Pw, where Pz=0
    P_w = [P_wh(1:2)/P_wh(3);0];

    % tnew = origin of world in virtual cam coordinates

    % 3D coordinates of P in physical camera frame
    P_c = T*[P_w;1];
    % camera to P distance
    dsurf = norm(-P_c);
end
```