EE 417 – Term Project

Term Project Report

Project 1 Title: 3D Reconstruction

Project 2 Title: Canny Edge Detector

Group Members:

Mert Kosan - 17472,

M. Mucahid Benlioglu - 17871

Supervisor(s): Mustafa Ünel

Date: 12.01.2017

Table of Contents

1. Summary of Report

We have done 2 project for the term project of EE417 (Computer Vision) course and this reports contains both reports of the project combined. First project is about 3D reconstruction of points from 2 views (2 stereo images). This project contains many computer vision algorithms such as calibration of camera, estimating camera matrices etc. Second project is about Canny Edge Detector, and in this project especially image processing algorithms and tools has been used.

In this report, both project has same structure. At the beginning, we will give summary and introduction of the projects, and after we will talk challenges we have encountered during the implementation and design of the project. Architecture, procedures that we have followed and result that we got from the projects will be explained. Discussion part will be available for each project, but this discussion parts will consist of individual comments and discussion from each group members. At section Appendix, we will put our implementation.

Both project is implemented in MATLAB 2016a and all codes are available in GIT repositories in public. Reader can examine our codes from below repositories.

Project 1: 3D reconstruction GIT repository:
https://bitbucket.org/pokemonteamcs310/3dreconstruction

Project 2: Canny Edge Detector GIT repository:
https://bitbucket.org/pokemonteamcs310/canny-implementation

2. Project 1: 3D Reconstruction

2.1.Abstract

In this project our focus was to estimate the 3D locations of objects using two stereo images. 3D reconstruction has always been a challenging achievement. Using 3D reconstruction, we can extract objects' 3D profile, also learn the 3D coordinates of any points. Therefore, 3D construction can be used in various areas of interest such as, geometrical analysis and design, industrial applications, computer graphics etc.

Since the computer vison is a reverse problem, in which we try to recover the lost data (i.e. the depth information) during the projection of 3D world to a 2D plane, 3D reconstruction of points using triangulation and recovering the depth information is one of the key aspects of the computer vision.

In this project, we have tried to address this key concept and tried to recover the lost dimension by using the corresponding features in pair of stereo images with a calibrated camera.

2.2.Introduction

This project is an application for recovering the lost 3D depth information on the image using stereo images and a calibrated camera. We followed some simple steps to address the issue and prepare a working application.

1. We have calibrated our camera, so that we can estimate important values that affect the reconstruction and image processing in general, such as tangential and radial distortion of the lens, focal length of the camera and principle point of the camera.
2. Matching a sparse set of points between stereo images by extracting some certain features from images and finding correspondences of these points.
3. Estimate the fundamental matrix, which will eventually be used in rest of the steps.
4. Re-detect the features from stereo images with a lower threshold so that we can obtain a denser set of points to track and plot.
5. Determine the 3D locations of the points using triangulation.
6. Plot the detected 3D point cloud, which is recovered *up to a scale*, in a graph.
7. Detect an object of known size, using the real size and calculated size recover the actual scale and plot the 3D points with metric distances instead of pixel distances.

Following sections will analyze the challenges that we have faced during the implementation, discuss about the algorithm of choice and implementation steps in more detail.

2.3.Challenges

First challenge was calibrating the camera accurately, since the accuracy of the estimation of camera parameters affect the 3D construction, we had to make sure a good calibration has done. We have previously calibrated one of our camera using a set of checkerboard patterns with the toolbox provided by Caltech, but to be able to use the built-in functions of MATLAB calibration results obtained by the camera calibration app had to be compatible with MATLAB's convention. Therefore, we decided to use MATLAB's camera calibration app with a new set of images.

Our first set of images did not produce a good calibration because the print quality of the checkerboard pattern was low and calibration app rejected most of the images. Then, we have changed our image set with higher quality checkerboard pattern, and took sufficient number of pictures of the pattern. Then we had to calibrate the camera with different subsets of the provided image set to reduce the reprojection error.

Next challenge was to decide which feature extraction algorithms to use, and which algorithms to use to estimate the essential matrix. After some short research and experiments, we decided to use Shi-Tomasi algorithm to detect the corners from the images and to match the features between two images we used Kanade-Lucas-Tomasi algorithm also RANSAC and MSAC over 8-point algorithm. The reason for these choices will be explained in more detail in the following sections.

We also encountered some challenges that we have not foreseen before the implementation. The most frustrating one was the result of the undistorted images. Resulting image obtained from the removing distortion process has black parts near edges, which is not part of the actual

image, but our corner detection algorithm detected multiple points on the edge of the black parts and created false feature extraction and feature tracking data. Surprisingly, our efforts to remove the outliers that do not fit the epipolar constraint did not leave them out and the resulting 3D reconstruction was corrupted. In order to prevent this corruption, we measured the approximate size and location of the actual image from one of the images and cropped the image accordingly, since the images are coming from the same camera and the distortion parameters are same, cropping operation fitted all 3 examples we used for this project.

2.4.Architecture

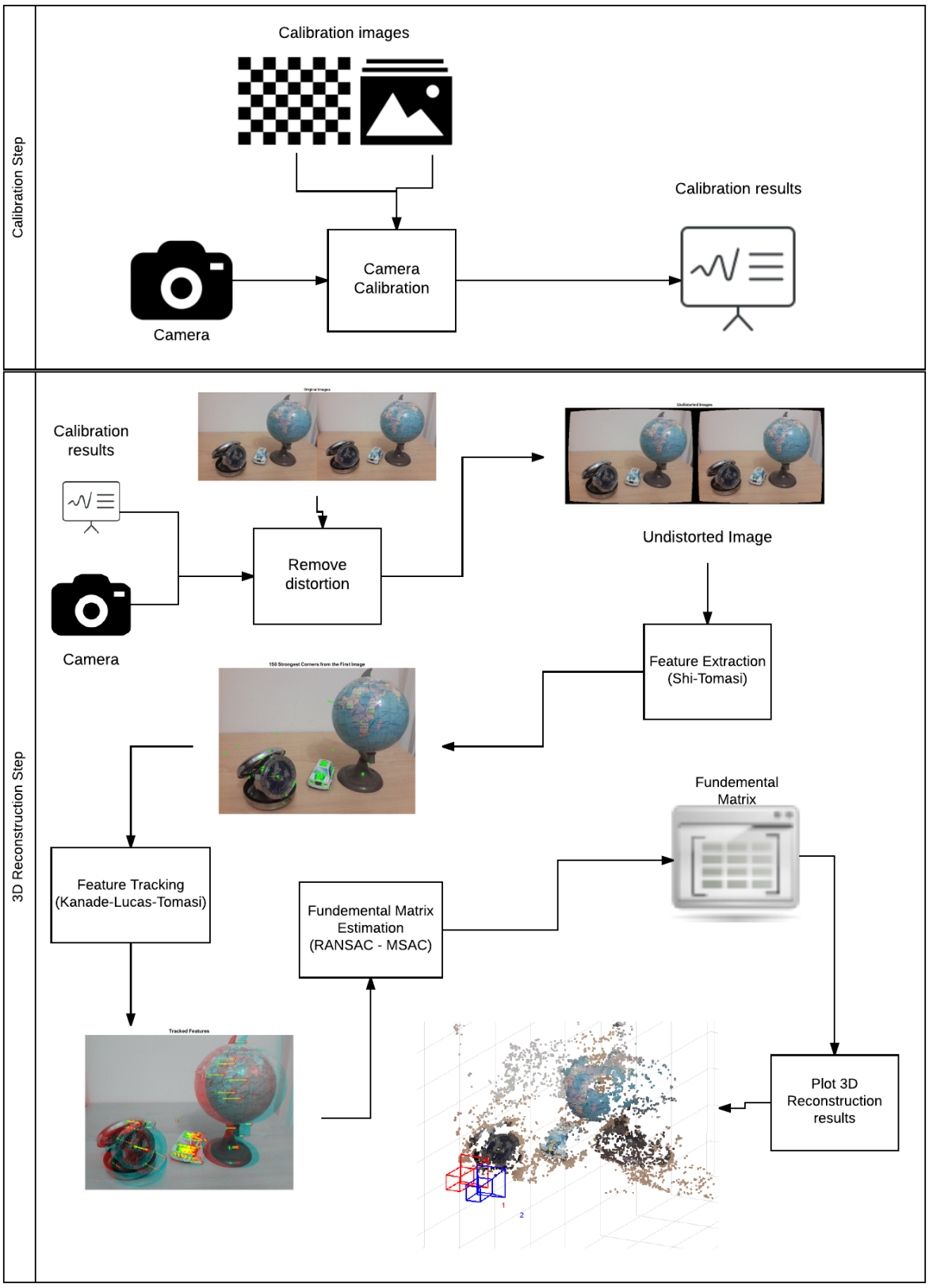


Figure 1. System Architecture for 3D reconstruction.

2.4.1. Implementation

2.4.1.1. Calibration Step

In order to make 3D reconstruction from given 2 views, we first need to calibrate our camera to estimate the camera parameters such as intrinsic parameters and distortion coefficients. Although previously we have calibrated our camera, we used Caltech's toolbox for it, but since we want the results to be compatible with MATLAB's built-in functions, we decided to recalibrate the camera with MATLAB's toolbox using a new set of images.

Because of the low-quality printing, our first set of images did not produce sufficiently good results, the contrast between black and white parts was not enough therefore, most of the images were rejected by the calibrator and that is when we decided to change the calibration image.

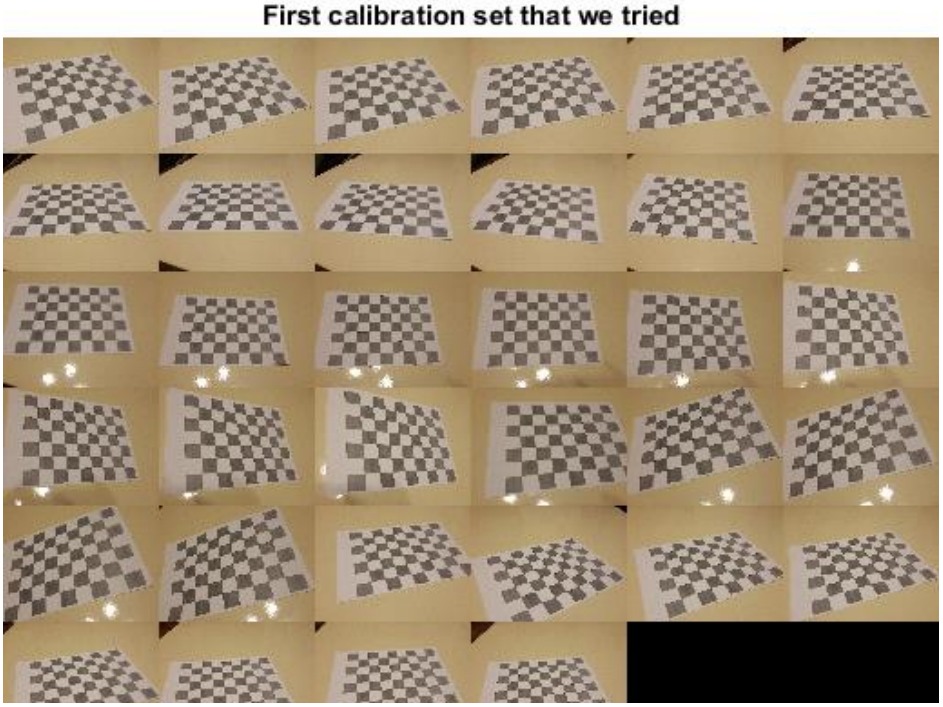


Figure 2. Calibration images with low quality checkerboard pattern.

After this set failed, we have prepared a new set of images with higher quality of the pattern. Before we calibrated the camera, we have skimmed all the images and left out the blurred images to prevent them to corrupt calibration data.

Figure 3. Blurred images

Then by using the rest of the images we have calibrated our camera and calculated the intrinsic parameters, distortion coefficients and extrinsic parameter. Our next step in the calibration was to check and correct the obvious errors and inspect the reprojection error. There were no obvious errors in the results (such as an image being behind the camera etc.) but reprojection error values were over one pixel on average, thus we have decided to discard the images that causes the most reprojection errors and recalibrated the camera. This time we have managed to pull the reprojection error below one pixel on average and satisfied with our results. We have saved the calibration results as ready-to-use data file and we also generated a script file to recalibrate the camera with the accepted images if necessary.

Figure 5. Initial reprojection error graph (images that result in dark blue errors removed from calibration)

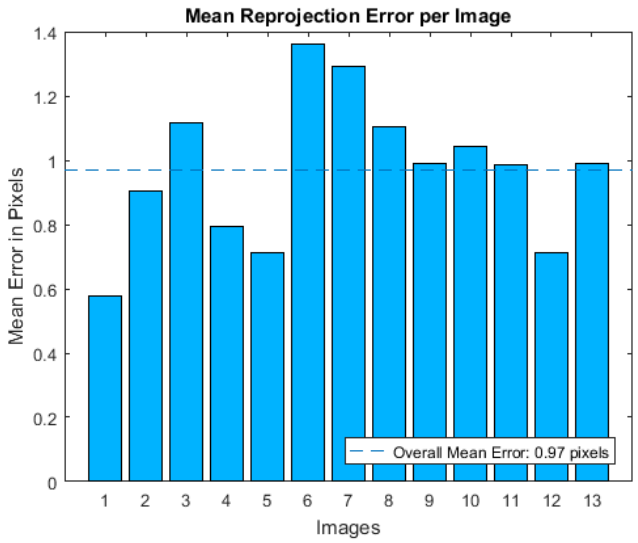


Figure 4. Reprojection error graph with final images.

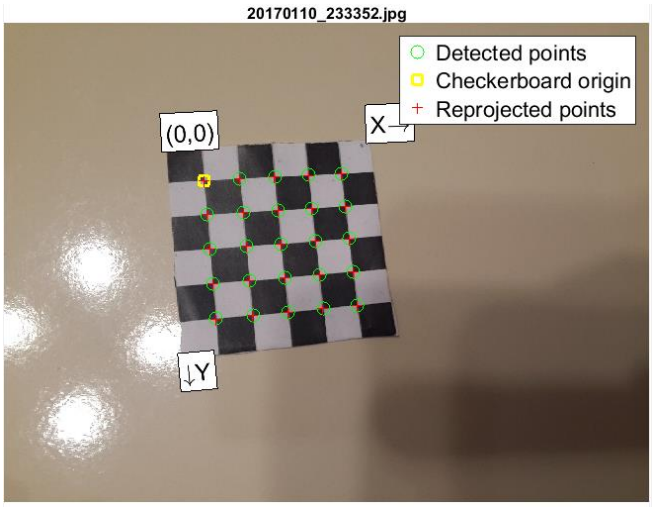


Figure 7. Example of calibration detections on one of the calibration images.

Figure 6. Resulting undistorted image from one of the calibration images.

2.4.1.2. Reconstruction Step

In the reconstruction step, the first thing we did right after loading images was to obtain the camera calibration results that we calculated in the calibration step. Then, by using that information we have undistorted the input images, since the lens distortion can affect the accuracy of the final reconstruction.

Although this process let us eliminate the radial distortion of the lens, it introduced new issues that affect the feature detection. Resulting undistorted images have black parts near edges, even though it is not the actual part of the image, if we do not specifically tell this to the algorithm, it will take the whole image and detect those useless black parts as features, which will eventually corrupt the final reconstruction results. Therefore, we have measured the location and the size of the actual image after removing distortion, and statically cropped the image to remove these black parts so that our feature extraction results becomes more robust. We have thought that, the static results that are obtained from an image (the image used to measure those results are not even one of the example images) may not apply to all images and still cause the same problem, and therefore, we may need to measure for each image. But, our results showed us that the results we have obtained fitted all the examples. After thinking about it, we have concluded that since we used the same camera for taking all the pictures, calculated distortion has to be more less the same, therefore one measurement satisfies all the images.

Secondly, we have extracted features from one of the images using Shi-Tomasi corner detection algorithm, which decides whether a feature is corner or not by checking the minimum eigen value of the corner matrix. After extracting the corners, we have tracked these features in the second image and determined the changes on the features using Kanade-Lucas-Tomasi

(KLT) algorithm, which calculates the weighted sums of the quantities $F'G, F'F, and\ (F')^2$ over the region of interest. (Lucas & Kanade, 1981).

As the third step, we have calculated the essential matrix using built-in functions provided us by MATLAB's computer vision toolbox. We first tried the normalized 8-Point algorithm, since we were familiar with it from our experiences during lectures and lab sessions. Results turned out to be good for the first and second examples, but when we applied the 8-Point algorithm to the 3rd example we have noticed that the resulting 3D point cloud does not produce satisfying enough results. After some research, we have seen that, MATLAB suggests us to use normalized 8-point algorithm if the feature points in both images match precisely. To produce better results and eliminate this problem, we have continued our research and discovered the Random Sample Consensus (RANSAC) algorithm, which is an iterative method to estimate parameters of a mathematical model from a set of data that contains outliers, when outliers are to be accorded no influence on the values of the estimates. (Strutz, 2016). The basic assumption of the algorithm is that the data consists of "inliers", which is the data that its distribution can be explained by some set of model parameters, though may be subjected to noise, and "outliers" that do not. (Random sample consensus, 2017). In our case, the difference between the inliers and outliers was whether the data fits the epipolar constraint (i.e. for each point observed in one image the same point must be observed in the other image on its epipolar line.) or not. By using this method, we have aimed to generalize our 3D reconstruction application to images with features that may not match precisely and have outlier aspects. Resulting reconstruction was satisfying after applying the RANSAC in our fundamental matrix and epipolar inliers estimations, but a further research showed us that we can improve the performance of estimation

(in terms of time) if we use Torr's suggested modification on RANSAC, which is MSAC (M-estimator Sample and Consensus). (Torr & Zisserman, 200).

Then, by using the camera calibration results, estimated fundamental matrix and epipolar inliers we have computed the rotation and the translation between cameras between the two images. Finally, after calculating the camera matrices as well, we have plotted the triangulated 3D points, which was initially up to a scale. But since we know there is a known object with known dimensions, we calculated the scale factor by using the real dimension of the object (globe) and the calculated dimension of it (valid for examples 1 and 2). As a result, we have obtained 3D point cloud where we can determine distances in centimeters.

2.5.Results

In this section, our 3D reconstruction method is shown on three different set of stereo images. Note that these results can be recreated by running the provided scripts. Also note that, due to the random nature of the RANSAC and MSAC algorithms reproduced results may not be exactly the same as this results, yet the results will have only unnoticeable changes and mostly be the same.

2.5.1. Example 1

Figure 7. Original Images for example 1.

Figure 8. After removing distortion from the image.

Figure 9. 150 Strongest Features extracted using Shi-Tomasi Algortihm.

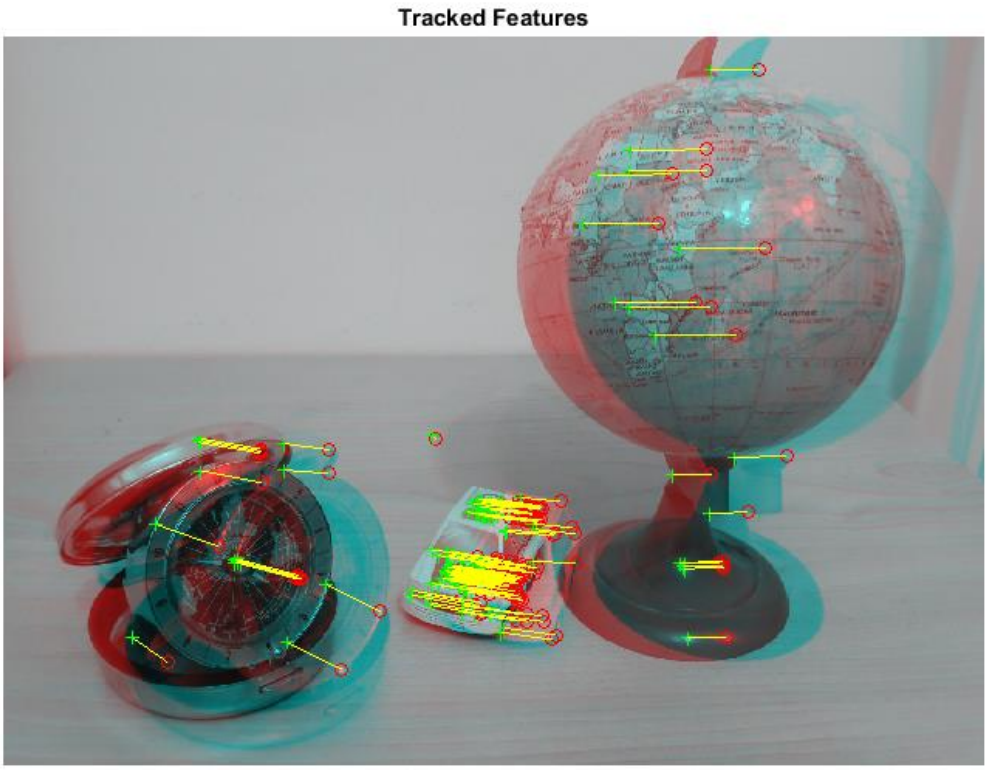


Figure 10. Result of feature tracking between the two images.

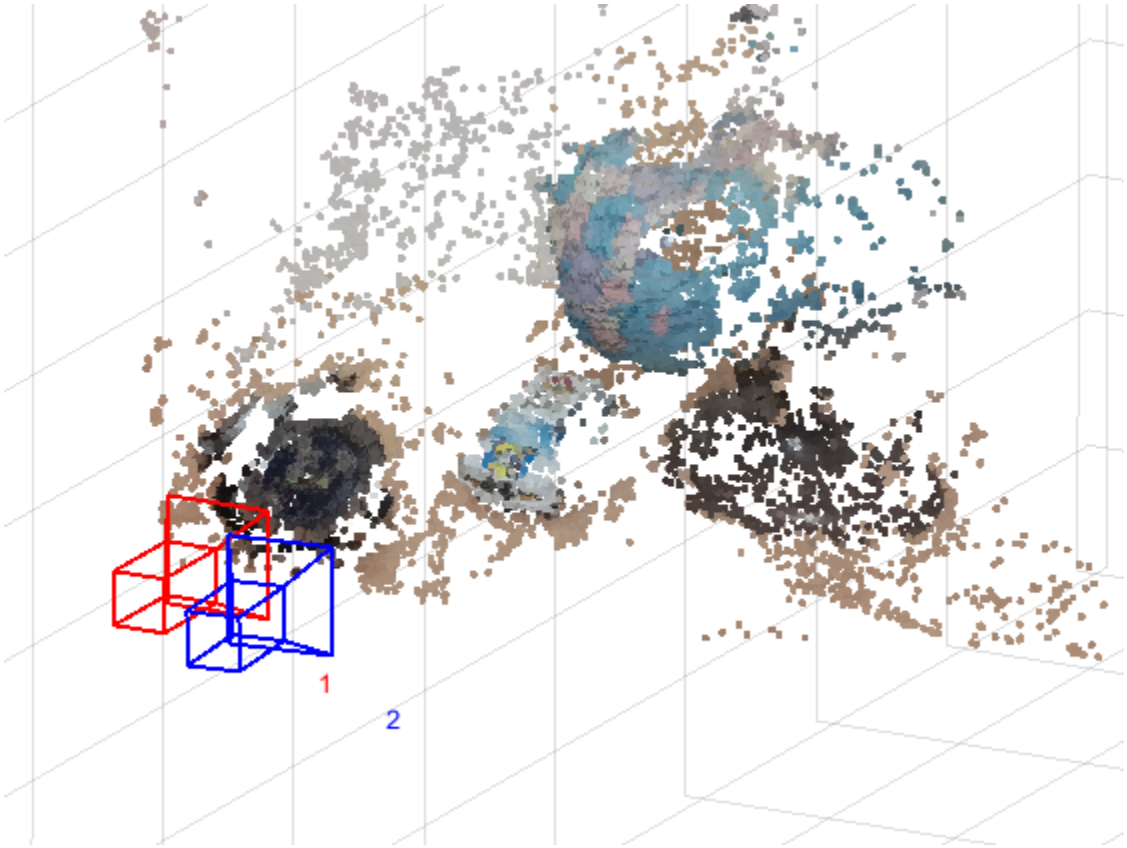


Figure 11. Reconstructed points in 3D plane up to a scale.

In here, we can only show the 3D points up to a scale, to determine the real distances between points and real coordinates of the points, we plot the best fitting globe on the image (since we know there is a globe feature in the images) and calculate the radius of the plotted globe. Since we know the real radius of the globe, we can determine the scale and calculate the real distances in centimeters.

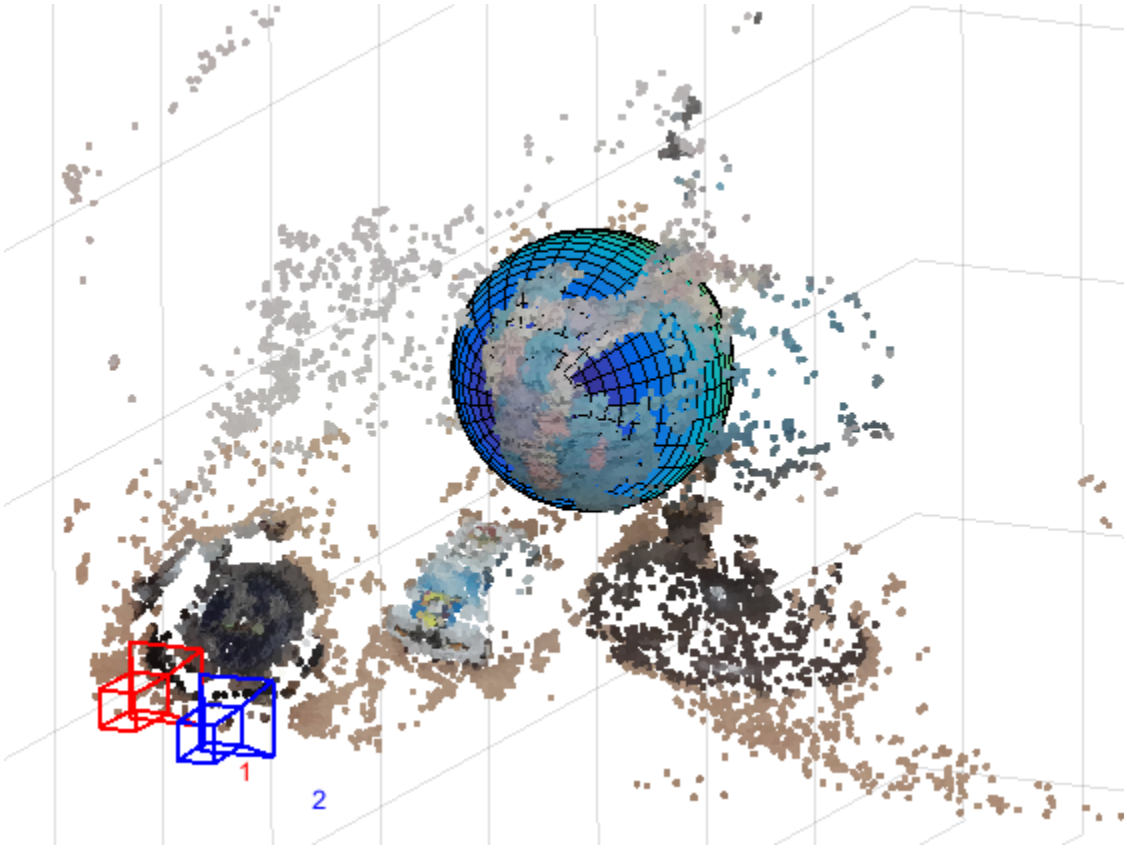


Figure 12. Finding the best fitting globe.

2.5.2. Example 2 (similar to example 1 with more objects)

Figure 13. Original images for example 2

Figure 14. After removing the distortion.

Figure 15. Detected Features using Shi-Tomasi Algorithm.

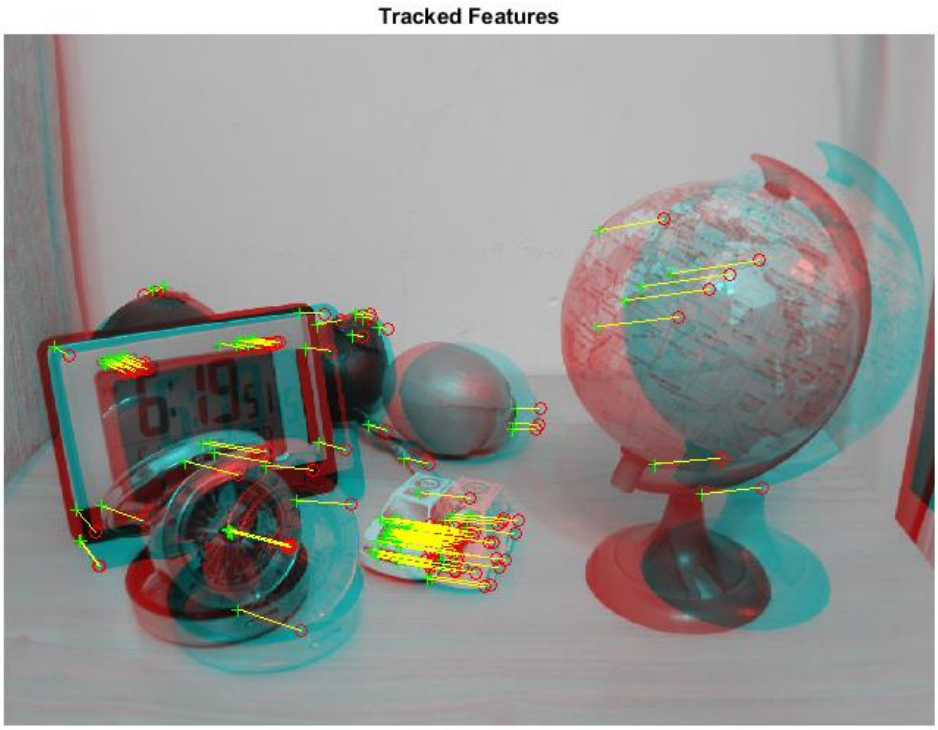


Figure 16. Correspondences of the detected features.

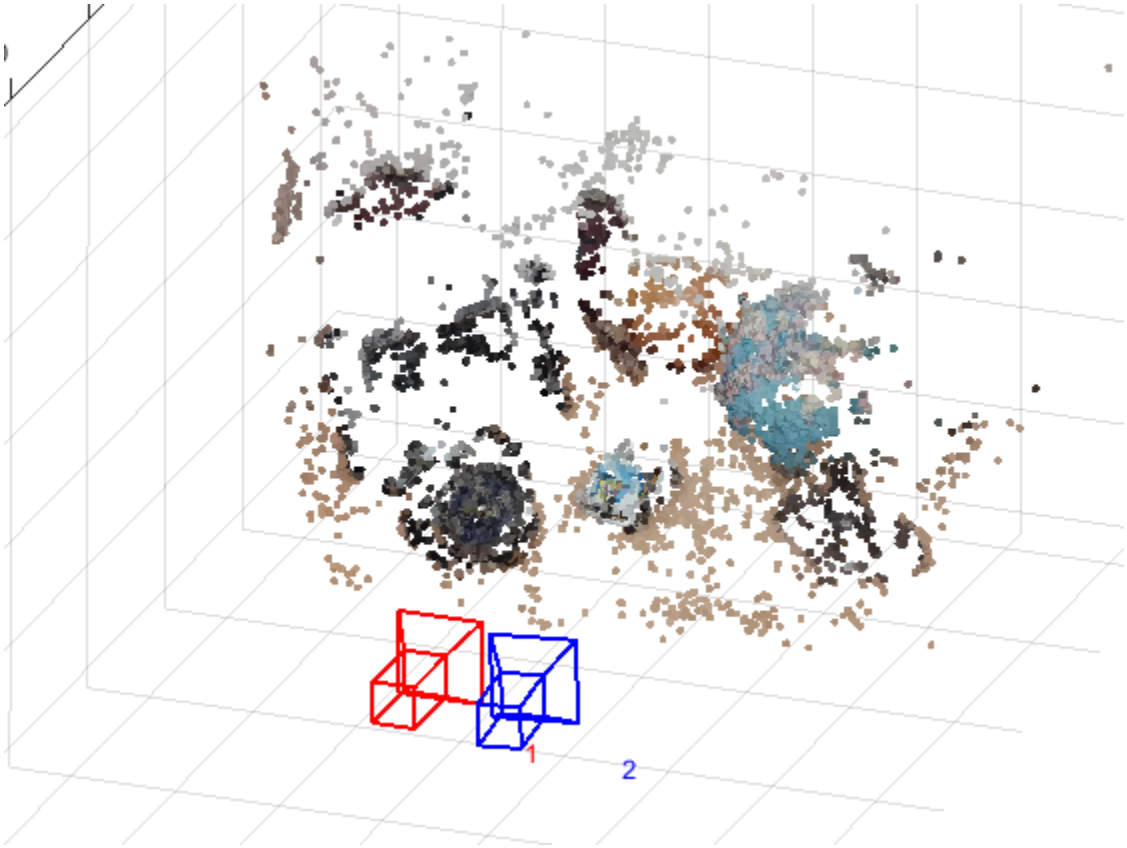


Figure 17. Reconstructed points

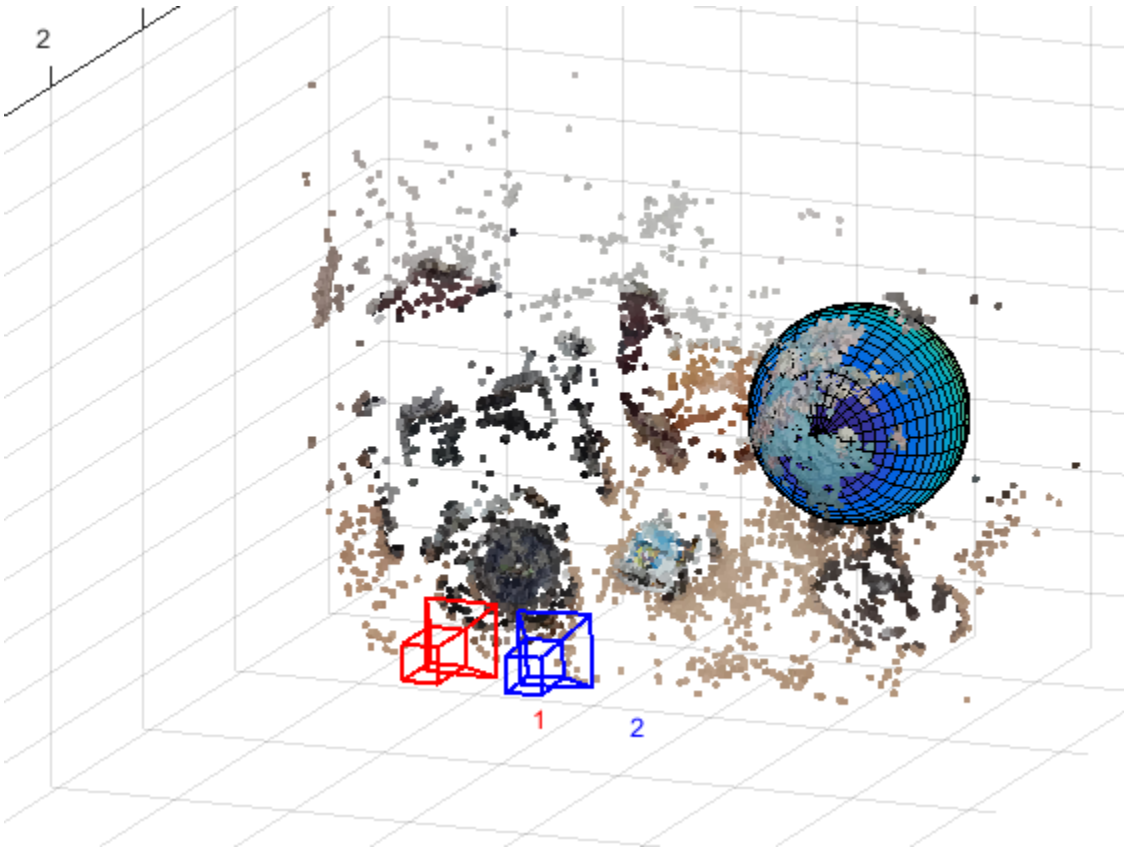


Figure 18. Reconstructed points with best fitting globe (for determining the scale)

2.5.3. Example 3 (human face)

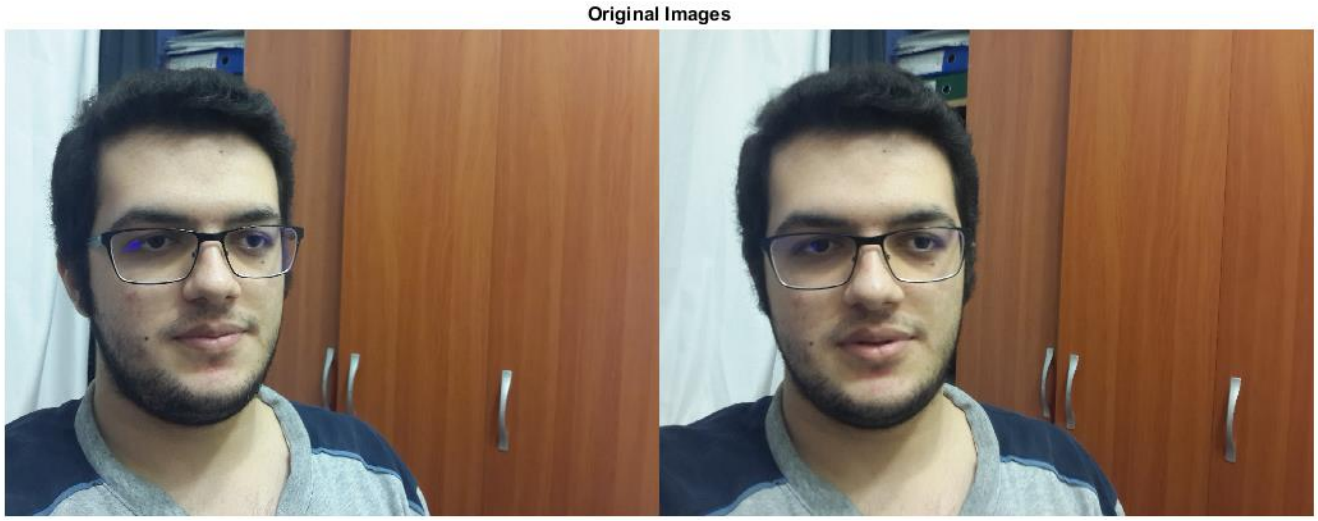


Figure 19. Original images for example 3.

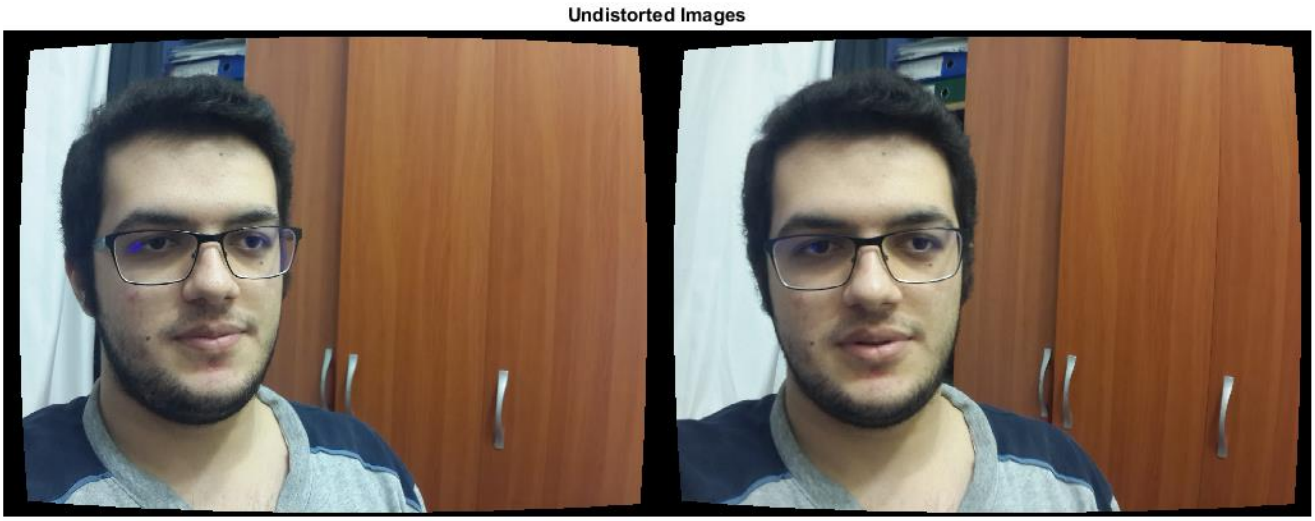


Figure 20. Undistorted version of the originals.

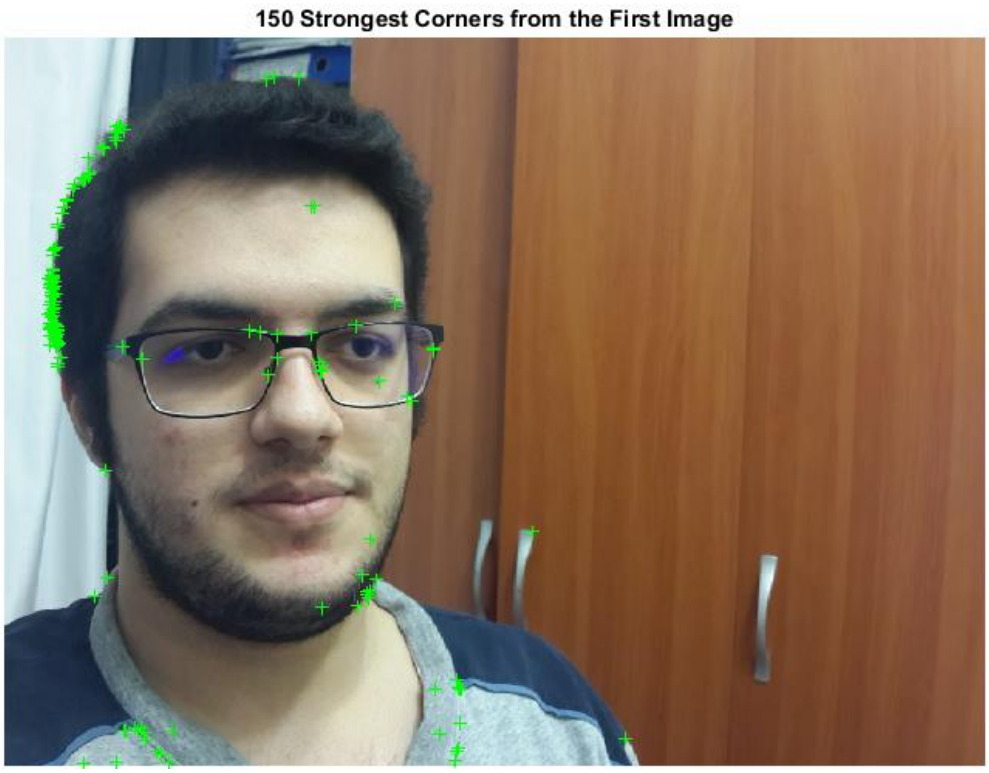


Figure 21. Strongest 150 corners detected with Shi-Tomasi algorithm.

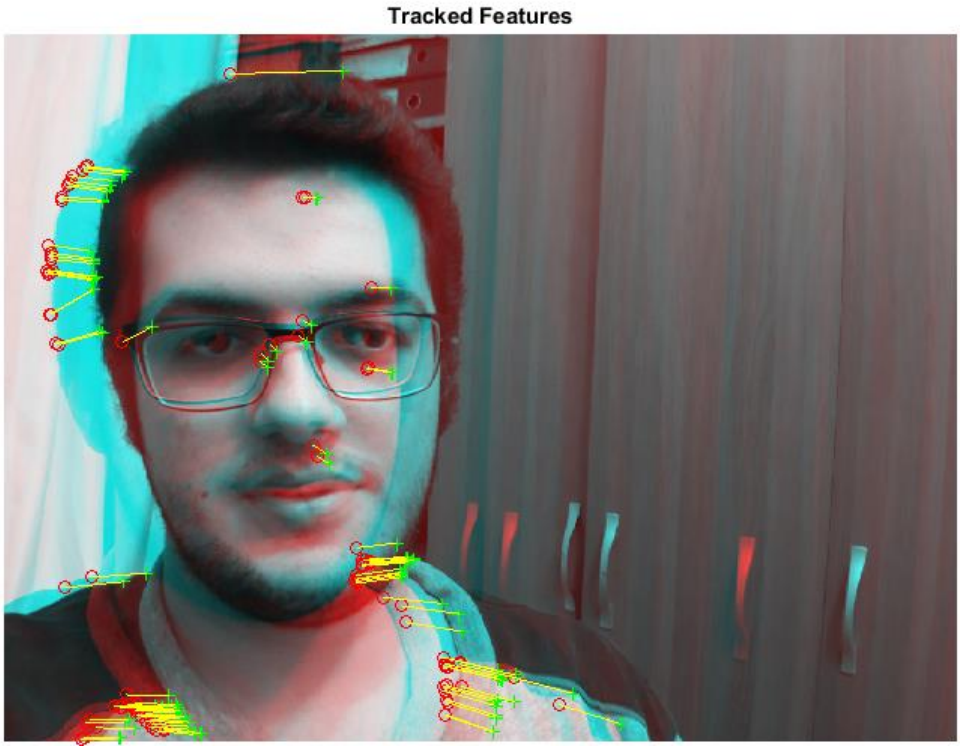


Figure 22. Correspondences of the detected features.

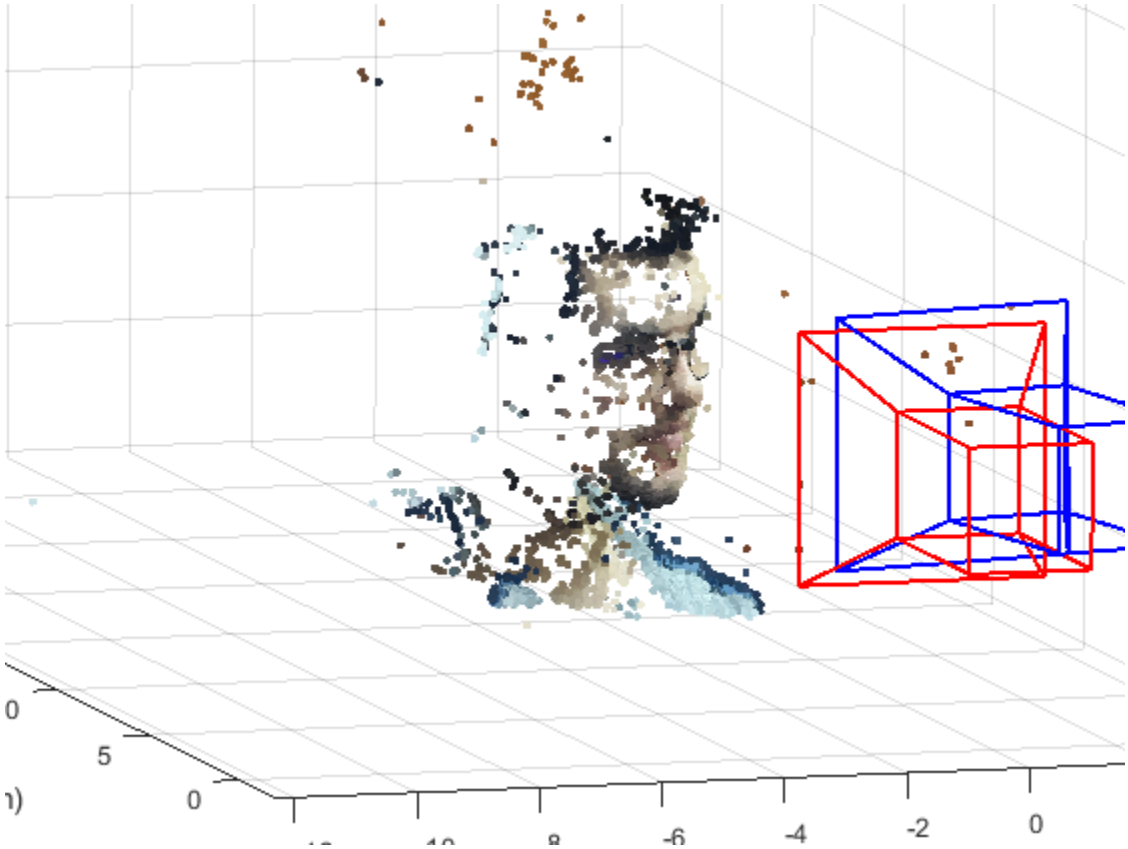


Figure 23. 3D reconstruction results of the points.

2.6.Discussion

2.6.1. Muhammed Mucahid Benlioglu

In this project, we have tried to triangulate 3D locations of feature points that are obtained from 2D stereo images. Although, we have mostly used the built-in functions that are provided by MATLAB's computer vision toolbox, to properly use those functions, we had to research on the algorithms that those functions implement. We were already familiar with some of the algorithms such as Shi-Tomasi algorithm for detecting corners using min eigen values of corner matrix, or 8-Point algorithm for fundamental matrix estimation, but we had to research the pros and cons of these algorithms and algorithms that produce better results than those. For example, we have learned that 8-Point algorithm produces good results if the detected features on images and their correspondences has to match precisely and if we input a stereo pair where there are some features that do not, it produces results that is not satisfying enough, thus we have made further research and overcame this results with different algorithms such as RANSAC. We have made researches on why is this algorithm is better than our initial choices or in which situation it is better to use this algorithm over others. We also made a research on feature tracking, and how Kanade-Lucas-Tomasi algorithm works. It was rather an easier research since the approach was similar to the optical flow problem we have dealt with during our lab sessions. As a result, we have successfully managed to obtain a 3D point cloud from the 2D stereo images

In my opinion, what most important thing that we have gained with this project is that we have introduced with new methods of feature extraction, feature tracking and fundamental matrix estimation. We have learned their differences pros and cons, which one to use in what situation, to maximize the optimality of the outcome.

2.6.2. Mert Kosan

In the project, we have tried to implement 3D reconstruction from 2D 2 images captured from different angle. To start to reconstruct 3D information from 2D images, we firstly calibrate our camera, which is rather important step to move on, then proceed to the main project steps. We kept the calibrated camera information (intrinsic parameters; focal length, distortion etc.) as a file, so we couldn't calibrate in every test of 3D reconstruction project.

We got rather good results, it is because in the project there are many hot-points and we focused on these hot-points and this gives us rather good results. On the below, I will discuss about these hot-points and how these hot-points affect project findings and our implementation.

1. Lens Distortion Problem: Owing to lens distortion, result would be rather bad, because lens distortion is a kind of error of camera. At the beginning, we eliminate the lens distortion to improve the result, and we got good results because of the fact that we undistorted the image.
2. Corresponding Points Problem: We used as many as corresponding point we can have. To do that we need to match points accurate and efficiently, for this purpose Shi-Tomasi corner detection algorithm and to find corresponding points Kanade-Lucas-Tomasi tracker has been used. In choosing good points is hot-point of the 3D reconstruction to estimate matrices which will build the 3D points. Algorithm that we have used helped to improve project results and findings which satisfies me.
3. Important Matrices: After we solved corresponding problem accurate, we were ready to estimate fundamental matrix and find epipolar geometry from that points. This fundamental matrix and epipolar geometry are important steps to find rotation matrix

and translation vector of camera. If we find and estimate these matrices accurately; 3D points, which is found by help of DLT (Direct Linear Transformation), will be found correctly. Our experiment and results show that these matrices or 3D points are found correctly (or nearly correct).

4. Visualization: Another hot-point of 3D Reconstruction is visualization, it is because we need to show visual findings as result to ensure the correctness of algorithm and method that we have used, to do that colors of resulting points are found from stereo images and if stereo image has globe (we used globe in our examples), we put symbolic globe to the result to see map on globe is correctly constructed or not. Resulting points and images are rather good and makes our implementation correct.

In my opinion, we achieved rather good results thanks to focusing on hot-point in the implementation. Also, in the result section, images show that improvements on implementation is needed, but this implementation is also good.

2.7. Appendix

2.7.1. Calibration Step (CalibrateCamera.m)

```
%% Info
%{
owners: mertkosan (Mert Kosan), mbenlioglu (Muhammed Mucahid Benlioglu)
created date: 10.01.2017

Camera Calibration using computer vision toolbox
%}

%% Calibration
% Define images to process
imageFileNames = {'CalibrationImages/2/20170110_233339.jpg',...
    'CalibrationImages/2/20170110_233343.jpg',...
    'CalibrationImages/2/20170110_233347.jpg',...
    'CalibrationImages/2/20170110_233350.jpg',...
    'CalibrationImages/2/20170110_233352.jpg',...
    'CalibrationImages/2/20170110_233409.jpg',...
    'CalibrationImages/2/20170110_233411.jpg',...
    'CalibrationImages/2/20170110_233412.jpg',...
    'CalibrationImages/2/20170110_233415.jpg',...
    'CalibrationImages/2/20170110_233416.jpg',...
    'CalibrationImages/2/20170110_233418.jpg',...
    'CalibrationImages/2/20170110_233422.jpg',...
    'CalibrationImages/2/20170110_233424.jpg',...
    };

% Detect checkerboards in images
[imagePoints, boardSize, imagesUsed] =
detectCheckerboardPoints(imageFileNames);
imageFileNames = imageFileNames(imagesUsed);

% Generate world coordinates of the corners of the squares
squareSize = 10;  % in units of 'mm'
worldPoints = generateCheckerboardPoints(boardSize, squareSize);

% Calibrate the camera
[cameraParams, imagesUsed, estimationErrors] =
estimateCameraParameters(imagePoints, worldPoints, ...
    'EstimateSkew', false, 'EstimateTangentialDistortion', false, ...
    'NumRadialDistortionCoefficients', 2, 'WorldUnits', 'mm', ...
    'InitialIntrinsicMatrix', [], 'InitialRadialDistortion', []);

% View reprojection errors
h1=figure; showReprojectionErrors(cameraParams, 'BarGraph');

% Visualize pattern locations
h2=figure; showExtrinsics(cameraParams, 'CameraCentric');

% Display parameter estimation errors
displayErrors(estimationErrors, cameraParams);
```

2.7.2. 3D Reconstruction Step (ObjectRecunstruction.m)

```
%% Info
%{
owners: mertkosan (Mert Kosan), mbenlioglu(Muhammed Mucahid Benlioglu)
created date: 10.01.2017

3D reconstruction of points from 2 views
%}

%% Read a Pair of Images
% Load a pair of images into the workspace.

clear all;close all;

imageDir = fullfile('StereoImages','3'); % 3 examples provided, replace 1
with
                                         % 2 or 3 for others
images = imageSet(imageDir);
I1 = read(images, 1);
I2 = read(images, 2);
figure
imshowpair(I1, I2, 'montage');
title('Original Images');

%% Load Camera Parameters

% Load precomputed intrinsic camera parameters using the Camera calibrator
% app (code for it is in CalibrateCamera.m)
load calibrationResults.mat

%% Remove Lens Distortion
% To improve the accuracy of the final reconstruction, we should remove the
% distortion from each of the images using the undistortImage function
% provided by the computer vision toolbox.

I1 = undistortImage(I1, cameraParams);
I2 = undistortImage(I2, cameraParams);

figure
imshowpair(I1, I2, 'montage');
title('Undistorted Images');

% black parts at the edge severely affect the results... Therefore it
% should be cropped to get rid of them, coordinates are measured statically
% from one pair, but it seems to fit every pair obtained from the same
% camera

I1 = imcrop(I1,[85 63 1900 1407]);
I2 = imcrop(I2,[85 63 1900 1407]);

figure
imshowpair(I1, I2, 'montage');
title('Undistorted Images after cropping black parts');
```

```
%% Find Point Correspondences Between The Images
% Detect good features to track using Shi-Tomasi feautres. Then we find the
% correspondig points between two images using Kanade-Lucas-Tomasi
% algorithm

% Detect feature points (Shi-Tomasi corners)
imagePoints1 = detectMinEigenFeatures(rgb2gray(I1), 'MinQuality', 0.1);

% Visualize detected points
figure
imshow(I1, 'InitialMagnification', 50);
title('150 Strongest Corners from the First Image');
hold on
plot(selectStrongest(imagePoints1, 150));

% Create the point tracker
% NumPyramidLevels: in this function KLT algorithm uses image pyramids,
% where each level is reduced in resolution by factor of 2 compared to
% previous level. This allows the algorithm to handle larger displacements
% of points between frames.
% MaxBidirectionalError: Track the point from previous frame to next then
% next to previous calculate the error between actual point and calculated
% one.
tracker = vision.PointTracker('MaxBidirectionalError', 1, 'NumPyramidLevels',
5);

% Initialize the point tracker
imagePoints1 = imagePoints1.Location;
initialize(tracker, imagePoints1, I1);

% Track the points
[imagePoints2, validIdx] = step(tracker, I2);
matchedPoints1 = imagePoints1(validIdx, :);
matchedPoints2 = imagePoints2(validIdx, :);

% Visualize correspondences
figure
showMatchedFeatures(I1, I2, matchedPoints1, matchedPoints2);
title('Tracked Features');


%% Estimate the Fundamental Matrix
% Caluclate fundemental matrix and find the inlier points that meet the
% epipolar constraint using the estimateFundamentalMatrix function

% Estimate the fundamental matrix
[fMatrix, epipolarInliers] = estimateFundamentalMatrix(...
  matchedPoints1, matchedPoints2, 'Method', 'MSAC', 'NumTrials', 10000);

% Find epipolar inliers
inlierPoints1 = matchedPoints1(epipolarInliers, :);
inlierPoints2 = matchedPoints2(epipolarInliers, :);

% Display inlier matches
figure
```

```
showMatchedFeatures(I1, I2, inlierPoints1, inlierPoints2);
title('Epipolar Inliers');

%% Compute the Camera Pose
% Compute the rotation and translation between the camera poses
% corresponding to the two images. Note that |t| is a unit vector,
% because translation can only be computed up to scale.

[R, t] = cameraPose(fMatrix, cameraParams, inlierPoints1, inlierPoints2);

%% Reconstruct the 3-D Locations of Matched Points
% Re-detect points in the first image using lower |'MinQuality'| to get
% more points. Track the new points into the second image. Estimate the
% 3-D locations corresponding to the matched points using the |triangulate|
% function, which implements the Direct Linear Transformation
% (DLT) algorithm. Place the origin at the optical center of the camera
% corresponding to the first image.

% Detect dense feature points
imagePoints1 = detectMinEigenFeatures(rgb2gray(I1), 'MinQuality', 0.001);

% Create the point tracker
tracker = vision.PointTracker('MaxBidirectionalError', 1, 'NumPyramidLevels',
5);

% Initialize the point tracker
imagePoints1 = imagePoints1.Location;
initialize(tracker, imagePoints1, I1);

% Track the points
[imagePoints2, validIdx] = step(tracker, I2);
matchedPoints1 = imagePoints1(validIdx, :);
matchedPoints2 = imagePoints2(validIdx, :);

% Compute the camera matrices for each position of the camera
% The first camera is at the origin looking along the X-axis. Thus, its
% rotation matrix is identity, and its translation vector is 0.
camMatrix1 = cameraMatrix(cameraParams, eye(3), [0 0 0]);
camMatrix2 = cameraMatrix(cameraParams, R', -t*R');

% Compute the 3-D points (direct linear transformation)
points3D = triangulate(matchedPoints1, matchedPoints2, camMatrix1,
camMatrix2);

% Get the color of each reconstructed point
numPixels = size(I1, 1) * size(I1, 2);
allColors = reshape(I1, [numPixels, 3]);
colorIdx = sub2ind([size(I1, 1), size(I1, 2)], round(matchedPoints1(:,2)),
...
    round(matchedPoints1(:, 1)));
color = allColors(colorIdx, :);

% Create the point cloud
ptCloud = pointCloud(points3D, 'Color', color);
```

```
%% Display the 3-D Point Cloud
% Use the |plotCamera| function to visualize the locations and orientations
% of the camera, and the |pcshow| function to visualize the point cloud.

% Visualize the camera locations and orientations
cameraSize = 0.3;
figure
plotCamera('Size', cameraSize, 'Color', 'r', 'Label', '1', 'Opacity', 0);
hold on
grid on
plotCamera('Location', t, 'Orientation', R, 'Size', cameraSize, ...
    'Color', 'b', 'Label', '2', 'Opacity', 0);

% Visualize the point cloud
pcshow(ptCloud, 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...
    'MarkerSize', 45);

% Rotate and zoom the plot
camorbit(0, -30);
camzoom(1.5);

% Label the axes
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis')

title('Up to Scale Reconstruction of the Scene');

%% Fit a Sphere to the Point Cloud to Find the Globe
% In examples 1 and 2 we know there is a globe (earth) therefore in order
% to better visulize the 3D shape we are plotting the best fitting globe by
% using pcfitsphere function, which is again provided by computer vision
% toolbox. Note: 3rd example does not contain any globe features, therefore
% resulting globe is irrelevant in that example...

% Detect the globe
globe = pcfitsphere(ptCloud, 0.1);

% Display the surface of the globe
plot(globe);
title('Estimated Location and Size of the Globe');
hold off

%% Metric Reconstruction of the Scene
% The actual radius of the globe is 5cm. We can now determine the
% coordinates of the 3-D points in centimeters.

% Determine the scale factor
scaleFactor = 5 / globe.Radius;

% Scale the point cloud
ptCloud = pointCloud(points3D * scaleFactor, 'Color', color);
t = t * scaleFactor;
```

```
% Visualize the point cloud in centimeters
cameraSize = 2;
figure
plotCamera('Size', cameraSize, 'Color', 'r', 'Label', '1', 'Opacity', 0);
hold on
grid on
plotCamera('Location', t, 'Orientation', R, 'Size', cameraSize, ...
    'Color', 'b', 'Label', '2', 'Opacity', 0);

% Visualize the point cloud
pcshow(ptCloud, 'VerticalAxis', 'y', 'VerticalAxisDir', 'down', ...
    'MarkerSize', 45);
camorbit(0, -30);
camzoom(1.5);

% Label the axes
xlabel('x-axis (cm)');
ylabel('y-axis (cm)');
zlabel('z-axis (cm)')
title('Metric Reconstruction of the Scene');
```

3. Project 2: Canny Edge Detector

3.1.Abstract

John F. Canny had been designed optimal edge detector in 1986. This edge detector is called his name, Canny Edge Detector. Canny states that edge detection has three main steps which are noise smoothing, edge enhancement and edge localization. To make these steps accurate, he used three criteria for this edge detector which are good detection, good localization, and single response constraint. With these three criteria, edge detector would work optimally (PennState).

There is no close-form for optimal filter but its kernel is like derivative of Gaussian filter. However, in traditional Canny edge detector, first Gaussian smoothing is applied and then first derivative of image along x and y (gradient of image) is found by Sobel (mostly), Prewitt or Roberts. And the most important feature of the Canny edge detector is that it has thinner edges because of edge localization and single response constraint and weak edges also can be used thus, edge if they have strong edges around them.

In this project, we have implemented Canny edge detector with some improvements upon a traditional Canny edge detector, at some point we have failed, at some point we have gotten good results. However, we were good at eliminating noise and unnecessary edges choosing good kernel for smoothing, derivation and threshold for edge elimination.

3.2.Introduction

Edge Detection is very important feature extraction method in image processing and computer vision, because as human beings, we can detect every object and their edges with our eyes, but can artificial intelligence or computers detect immediately or how can they detect the edges of the object in digital images? Images are made of numbers, and computers only work on these numbers, so processing of this detection is much longer than humans. However, the day passes, algorithms of edge detection are being really improved and very fast. Every scientists, mathematicians and engineers are trying to find optimal solution to this problem. Canny, in 1986, formalized some criteria to build optimal edge detector which is called Canny Edge Detector, which we were trying to build this solution in this project.

In Canny edge detector project, most of implementation is based on EE417 lecture slides, journal (Zhou, 2011), report (Yang Tao, 2015) and Wikipedia. Canny Edge Detector algorithm is implemented in MATLAB 2016a with license of Sabanci University. We have changed the traditional algorithm based on our experiments on Canny Edge Detector, and these changes are sometimes affecting our code in a good way, sometimes bad way. We tried to choose best way to proceed the next steps. However, algorithm has mainly 5 steps in it.

1. Pre-process: Input image has been changed to grayscale and double precision to make image more reliable to work on it.
2. Smoothing: Eliminating noise which is creating bad effect on edge detection algorithm, Gaussian filter is used for this purpose.

3. Derivation: To find edges of the image along x and y axes, then combine them. These derivations also help us to find directions and angles of each pixels which we used in the next steps.
4. Non-max Suppression: First privilege step of the Canny algorithm, which is focusing only on local maximum edges and this algorithm will make the edges thinner.
5. Hysteresis Thresholding: Second privilege step of Canny algorithm, which is keeping also weak edges which are continuation of strong edges. Alone weak edges are eliminated from image, this operation also can eliminate noise from image.

Above steps are main steps of the Canny algorithm, we have changed this algorithm a little bit to improve our code, in the next section, we will introduce the changes and how we succeed for failed.

The rest of reports, we will talk about challenges that we encounter during the project, system architecture, explanation of our implementation, procedure that we have followed, resulting images of our implementation and comparison with built-in MATLAB functions and discussion about the project from each group members; Mert Kosan and M. Mucahid Benlioglu.

3.3. Challenges

First challenge was determining the smoothing filter for eliminating noise, it because when eliminating noise, algorithm also smoothing the edges which is bad. It is because, this operation can also delete information from near of edges. Tao states that a bilateral filtering, which is edge preserving smoothing method, is better method for smoothing image rather than Gaussian filter. In bilateral filtering, edges and near edges smoothing kernel is different, weights are assigned differently. This different assignment of weight makes less smoothing on edges (Yang Tao, 2015). However, we have implemented bilateral filtering, Gaussian filter is still better filtering method rather than our implementation of bilateral filtering so that we kept Gaussian filter for smoothing method.

We also implement a function for convolution purpose which has $O(n^2)$ complexity, but it is very slow for large images, every convolution of whole image is approximately taking 5-6 seconds, it makes the code hard to test. Then we realized that there is built-in function in MATLAB (conv2) which is fast, I don't know the complexity but one whole image complexity takes approximately 1 second with this function. So, we used built-in function, but we will put our function to the appendix section of the report.

The second important challenge was chaging the derivation kernel in terms of size. Our derivation function has proposed for different type of kernel which are Sobel or Prewitt, however in our implementation and tests, we generally used Sobel operator for computing gradient of the image. The challenge was the size of the Sobel operator. Generally, 3x3 Sobel has been used for this purpose, we have tested 5x5 and 7x7 Sobel operator and both operator is better than 3x3, in some case 5x5 is better, in some cases 7x7. However, we have used 7x7 Sobel operator for derivation case and this makes our result better.

Another challenge was choosing threshold values dynamically, this is because there are no generic thresholds values for all images. That's why we tried to find this threshold value dynamically. We cannot say that we have perfect choice of these threshold values but Otsu Algorithm helped us to find high threshold, then we choose half of it as a low threshold value. We played with Otsu Algorithm a little bit by changing result of this algorithm with our experience of resulting images. However, our modified Otsu algorithm needs to be improved, we didn't change the algorithm by mathematical proofs, but our experiments make it change and result is not bad. Otsu algorithm is only looking histogram of the image, but we also looking size of the image (how many pixels it has), this is because based on our experiment, high quality images has strong, clear edges whereas low quality images have weak ones but they are expected to find by our algorithm, so if image has less pixels in it, we reduce the threshold values.

3.4. Architecture

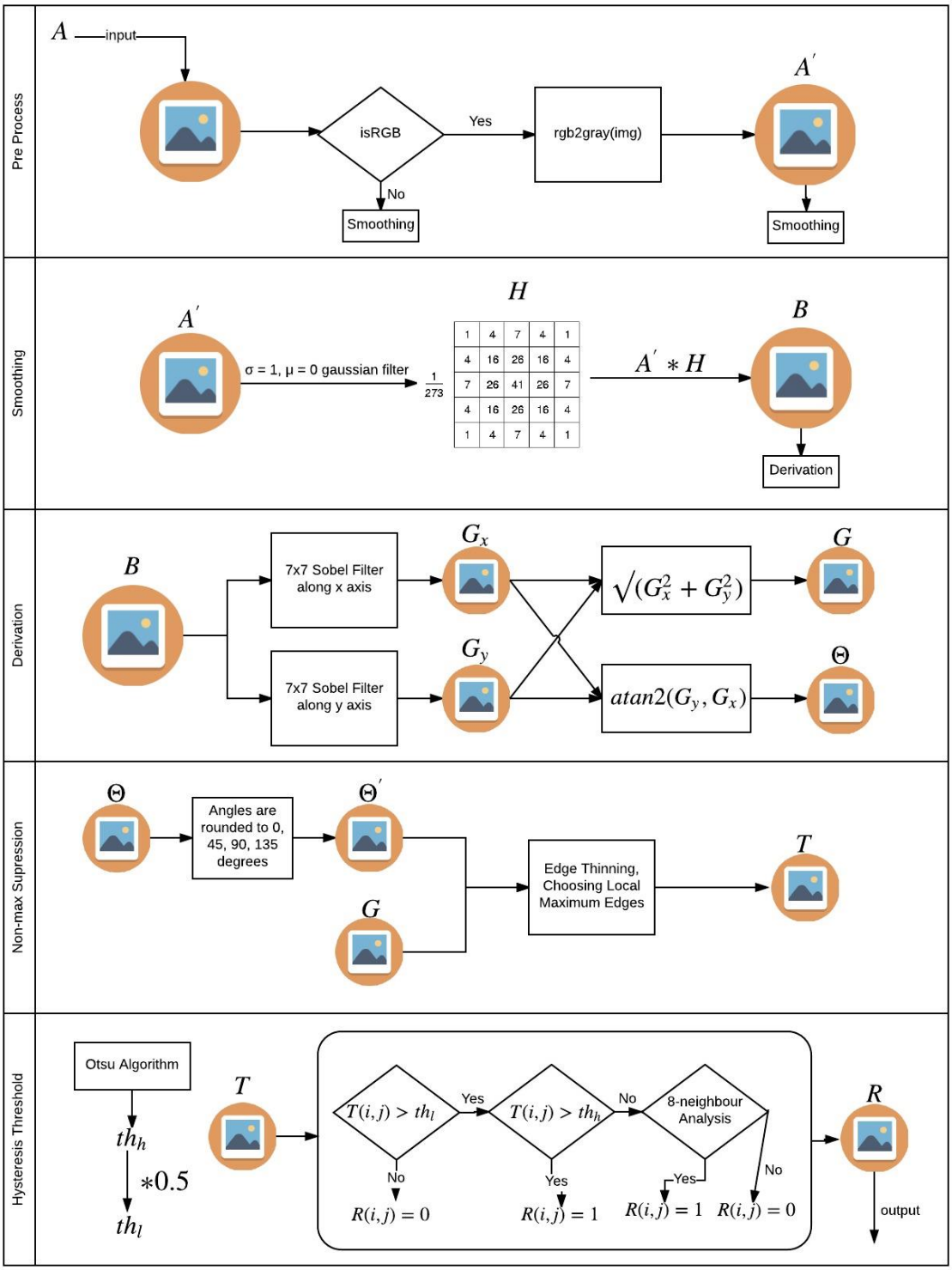


Figure 24.System Architecture of Canny Implementation

3.4.1. Implementation

3.4.1.1. Pre-Process

In pre-process section, we check the image is RGB or not. If the image is RGB, we turn into grayscale (rgb2gray built-in function of MATLAB has been used), because Canny implementation is edge detector and to find edges with using gradient method and smoothing the image, we need grayscale of the image. If is grayscale in the beginning, we kept as it is and move on the next phase, which is smoothing the image.

3.4.1.2. Smoothing

In smoothing section, we are smoothing the image to eliminate noise with zero-mean and sigma equals 1 Gaussian kernel (Figure 2), which has 5x5 window size. We are iterating the window along the image and convolving the image with this kernel (conv2 built-in function of MATLAB has been used). Result image is going to be derived with Sobel filters to find its gradients in the next phase, derivation.

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figure 25. zero-mean 5x5 Gaussian Kernel with sigma = 1

3.4.1.3. Derivation

In the derivation step, there is an improvement upon a traditional Canny edge detector which is explained in Introduction and Challenges sections (Zhou, 2011). This development is using the bigger window-sized Sobel filter which is cleaning more noise and detects edges more accurate. 7x7 Sobel filter has been used for both along x and y axes to gradient of the image, to find bigger Sobel filter Figure 5 formula has been used. Then hypotenuse of the gradients (Figure 3) and angles (directions) of each pixel (Figure 4) has been calculated.

$$G = \sqrt{G_x^2 + G_y^2}$$

Figure 26. Find Gradient of Image

$$\theta = atan2(G_y, G_x)$$

Figure 27. Find angles of each pixel

$$sobel(a\ x\ a) = cov2\big([1\ 2\ 1]' * [1\ 2\ 1], sobel(b\ x\ b)\big)\ \ (a = 2k + 3 = b + 2)$$

Figure 28. Find bigger Sobel kernel, cov2 is convolution function in MATLAB

3.4.1.4. Non-max Suppression

Before start of non-max suppression phase, we rounding the angles matrix to 45*k degrees, it because we want to deal with only rounding angles to determine which edges are necessary, which are not. Also at the end, direction is not important for us, the important thing is only line, it because in non-max suppression algorithm, we are looking both direction when we are dealing with angles. So, for our case 225 degree is also 45 degree (for all k = k + 180 degrees, same procedure applies), that's why at the end, we have only 0, 45, 90 and 135 degrees in angles matrix.

In the non-max suppression algorithm, we are iterating all pixels in normalized angles matrix. For each pixel, we will have 4 cases, these cases are 0, 45, 90 and 135 degrees. For example, for case 0; we will have get 3 values from gradient of image, these values are current, right, and left pixels (this is because these 3 three pixels creating 0 degree). If the maximum of these values is current pixel then current pixel value of thinner image (result of non-max suppression) is 1, otherwise 0. Same procedure is being applied on case 45, 90 and 135 based on their directions. At the end of algorithm, all edges in the image is now thinner, because local maximum edges are selected and this algorithm eliminates the non-max edges. However, in the result image, there are many edges which are locally max but globally very low, that's why in the next step, we will apply hysteresis thresholding to eliminate this weak edges and noise.

3.4.1.5. Hysteresis Thresholding

Before start of hysteresis thresholding, we are using Otsu algorithm to decide two threshold values, because there are no generic threshold values for each image, that's why we need to find these threshold values by dynamically based on image and result of Otsu algorithm, Also, threshold values are multiplied by maximum pixel value of the thinner gradient image, it because with this method thresholds are more image based.

In the hysteresis thresholding algorithm, at the beginning, we have result image of non-max suppression (thinner image) and two threshold values. Main reason to use two threshold value is try to not lose weak edges near strong edges, which are also valuable for the algorithm. Flow of the algorithm is iterating all pixel values of thinner image and for each pixel, we are comparing the pixel with threshold values. This comparison has three case, first one is that pixel is lower than low threshold which means this is not an edge. Second case is that pixel is equal to or higher than high threshold which means this is an edge. Third case is pixel is between the low and high threshold. In this case, we are looking 8 neighbor components, if one them is greater or equal than high threshold, it means this pixel is an edge, otherwise it is not an edge. Result image will be binary image, if we multiplied by 255 with each pixel, then result image is 8 bits image, then this image will be result of whole algorithm.

3.5. Results

In this section, we will introduce some results for each step of our algorithm in detail for one image and we will show you some result image of our implementation.

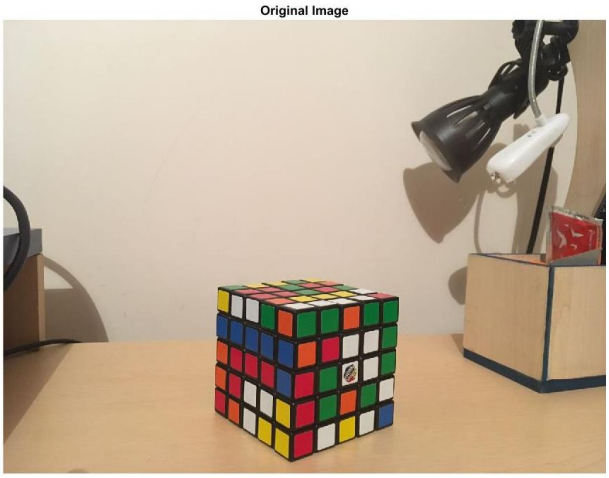


Figure i. Original Image (cube.jpg)

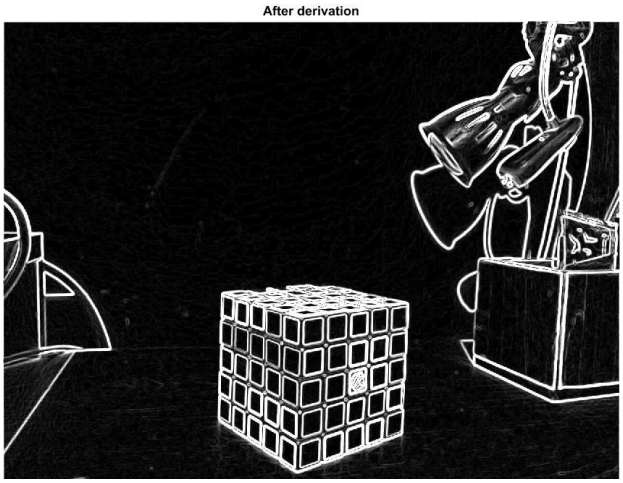


Figure ii. After Gaussian and Sobel filter

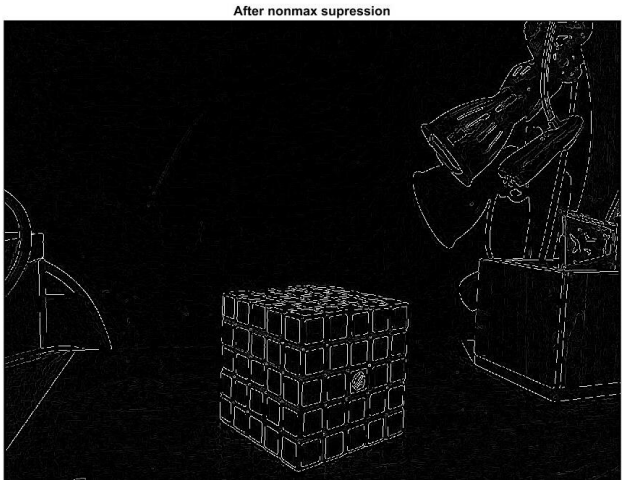


Figure iii. After Non-Max Suppression

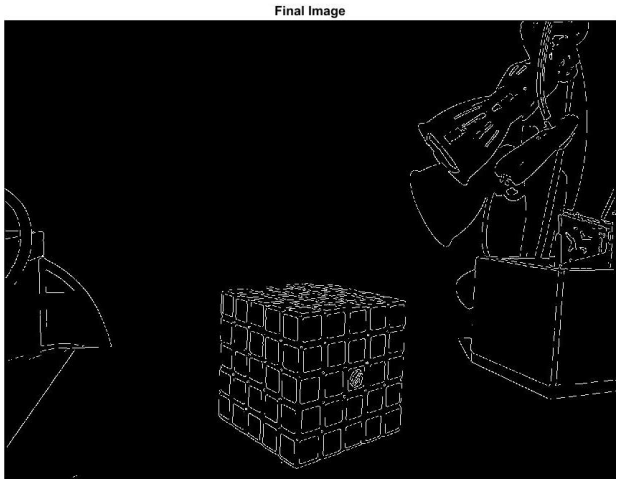


Figure iv. Final Image

Above images show that thinner operation (non-max suppression) is very good when you look at figure ii to figure iii. When you compare figure iii and figure iv, some noise effects and unnecessary edges on the table are lost as we are expected, so result is very good.

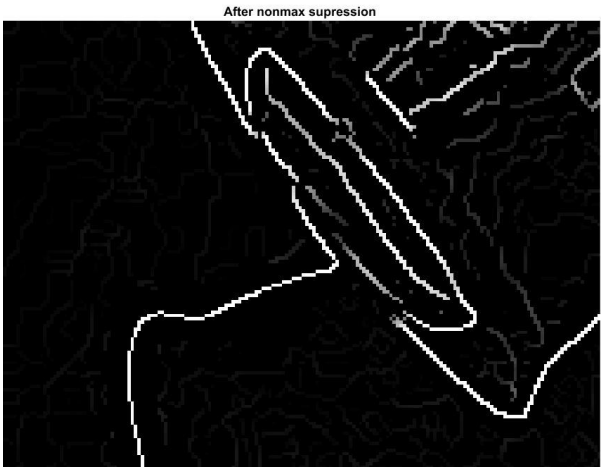
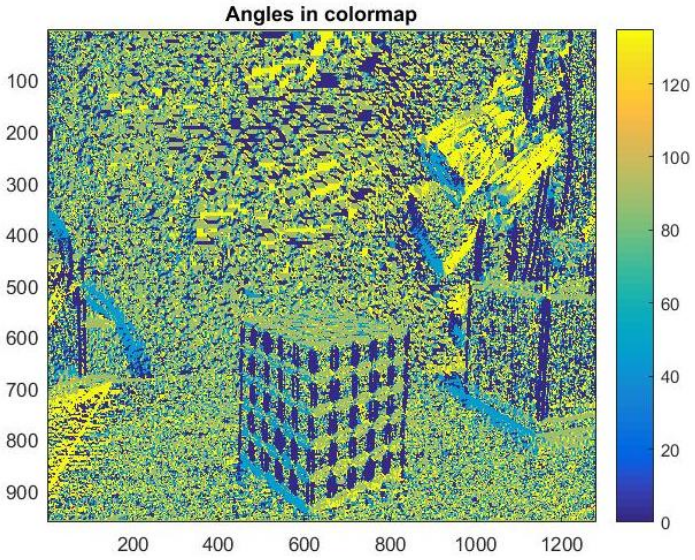


Figure v. Zoomed non-max suppression

Figure vi. Zoomed final image

Above images show zoomed on lamb version of cube image (cube.jpg). Left image is representing the image that is after non-max suppression operation, while right image is final image that is after hysteresis thresholding algorithm. Reader of this report can clearly see that left image has many weak edges, but hysteresis algorithm gets rid of this weak edges and noise to clear the image, and result image is expected image for our algorithm.



Above image shows the angles of each pixel on the image, for this purpose colormap function of MATLAB is used. These angles will be used in non-max suppression to make thin the edges.

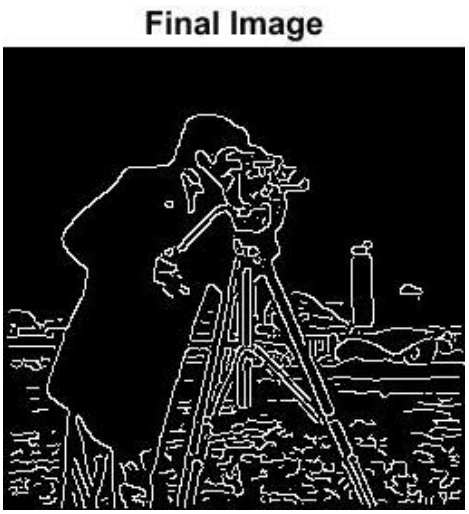


Figure vii. Final image (cameraman.tif)

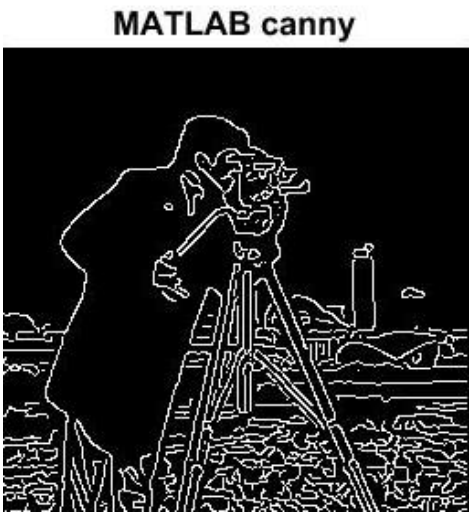


Figure viii. MATLAB Canny, cameraman.tif

Left image is our result, whereas right image is MATLAB built-in function Canny result. If we compare both them, we can say that our implementation and result are very accurate and good (image is 'cameraman.tif').

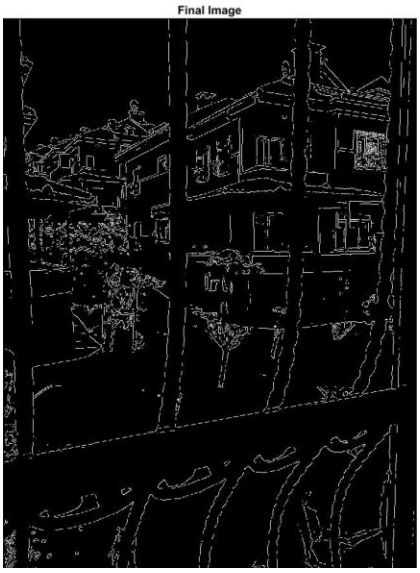


Figure ix. Final image (hapishane.jpg)

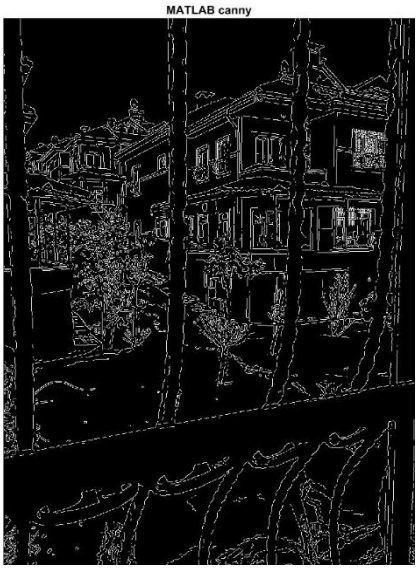


Figure x. MATLAB Canny, hapishane.jpg

Above comparison ('hapishane.jpg') shows that MATLAB Canny (right image) is more detailed than our implementation (left image), but our implementation extracts only important features and eliminates noise as much as it can.

3.6. Discussion

3.6.1. Mert Kosan

In the project, we tried to implement improved Canny edge detector rather than traditional Canny edge detector, which has been proposing optimal solution to edge detection algorithms. They have many similarities, however traditional one has some defects in it. Below, there some information about these defects, solutions if we have used and results of these solutions.

1. Gaussian filter, which is also smoothing the edges and these edges will be lost during the smoothing operation. So, we have researched the algorithm that will not smooth the edges. We have found bilateral filtering algorithm and we applied to the image rather than Gaussian filter. However, with Gaussian filter result was better or we could not apply bilateral filtering sufficiently, so we kept the Gaussian filter as it is.

2. For gradient calculation of the image along x and y axes, Sobel filter is used. In traditional canny, 3x3 Sobel filter has been used, however we have tested 5x5 and 7x7 Sobel filter on the image, and at the most case 7x7 Sobel filter much better than 3x3 in terms of finding edges and smoothing the image.

3. For thresholding problem, in Canny algorithm, there are two thresholds which controls the weak edges if they are near strong edges so that we keep the important weak edges. However, these thresholds are not generic for all images, so we applied Otsu algorithm to find high threshold. After that we determined the low threshold as a half of high threshold.

Our aim was to detect edges and make them thinner with non-maximum suppression and eliminate unnecessary edges and noise with hysteresis thresholding as Canny did, but also changed the size of gradient kernel, which really improves our results, and determine the threshold value dynamically for each image, it makes our code is generic at some point. However, this estimating the thresholding point code should be improved and tested on many images to see actual results.

The most important features (privileges of Canny Algorithm) of the Canny Algorithm are non-max suppression and hysteresis thresholding, that's why these algorithms are hot-point of our Canny implementation. They are carefully implemented and they have results well but also, they are the hot-points of the algorithm that should be improved.

In my opinion, our result is very good in terms of finding the important edges on the image rather than some noise effect edges of the image. Surely, there can be made many improvements on our algorithm, but the point or stage that we are on is rather good.

3.6.2. Muhammed Mucahid Benlioglu

Our focus in this project was to re-implement Canny edge detector that we used during the semester and while implementing try to find some methods to improve its effectivity. During implementation, after making some research we have concluded that we may introduce some level of improvement to the existing algorithm in three possible ways:

- We may change the filter used for derivative operation. Originally, Sobel operator with 3x3 kernel is used for the gradient operation. We know that Sobel operator acts as a crude approximation of Gaussian. By using this fact, although there are no officially stated Sobel kernels in higher dimension (e.g. 5x5 or 7x7) we can obtain kernels with the same idea. Our expectation was, with the increasing Sobel operator's kernel size, we can reach a better approximation of Gaussian thus obtaining better results.
- We can implement a smart smoothing step that apply less smoothing on edges and more smoothing on noises. The first step of Canny (and most of image processing operations) is to smooth the image to get rid of noise from the image, but this smoothing may cause us to lose some of the edges. Therefore, to differentiate the noise between edges, we have decided to use bilateral filtering instead of traditional Gaussian, but our implementation did not give sufficient results, therefore although the idea seemed good, we have decided to keep Gaussian filter as our smoothing step.
- We can implement a dynamic double thresholding mechanism that determines the minimum and maximum thresholds depending on the image. In MATLAB's implementation of Canny thresholds are entered statically or default values are used

(which are also static). Instead of using these static values, we have decided to implement a method that generates these thresholds depending on the image. For this we took Otsu's thresholding method as base, but resulting raw data from Otsu did not seem good, therefore we made slight modifications to keep the thresholds in between some acceptable levels.

In conclusion, we have obtained our own version of Canny algorithm, which can be further improved in terms of speed and effectiveness. Our comparison results showed that our implementation of Canny produces similar results with the MATLAB's version of Canny.

3.7. Appendix

3.7.1. Main Script (thecleverguy.m)

```
%{
owner: mertkosan (Mert Kosan), mbenlioglu(Muhammed Mucahid Benlioglu)
created date: 09.01.2017

Canny Edge Detector implementation
%}
close all; clear all; clc;

img = imread('cube.jpg');
figure; imshow(img); title('Original Image');
[imgNew] = prepare_image(img);

tic
[canny, thresh] = edge(imgNew, 'Canny');
figure; imshow(canny); title('MATLAB canny');
toc

gauss_kernel_size = 2;
gauss_kernel = 1/273.*[1 4 7 4 1; 4 16 26 16 4; 7 26 41 26 7; 4 16 26 16 4; 1
4 7 4 1];

kernel_size = 7;
type = 'Sobel';
[kernel_x, kernel_y] = return_derivation_kernel(type, kernel_size);

%{
  conv2 is far better than my convolution function in terms of speed
  so, i used conv2 and 'Same' parameters results same img size
%}
[imgS] = conv2(imgNew, gauss_kernel, 'SAME');
%convolution(imgNew, gauss_kernel, gauss_kernel_size);
figure; imshow(imgS); title('After Gauss Smoothing');

%derivation
[imgX] = conv2(imgS, kernel_x, 'SAME');
%convolution(imgS, kernel_x, kernel_size);
[imgY] = conv2(imgS, kernel_y, 'SAME');
%convolution(imgS, kernel_y, kernel_size);

%{
    merge two derivative image
    abs method also can be used but, this method is more sufficient to
    detect edges correctly
%}
imgXY = sqrt(imgX.^2+imgY.^2);
%imgXY = abs(imgX) + abs(imgY);
figure; imshow(imgXY); title('After derivation');
```

```
tic
%direction between two derivatives
angles = atan2(imgY, imgX) * 180 / pi;
normalized_angles = normalize_directions(angles);
figure; imagesc(normalized_angles); title('Angles in colormap'); colorbar;
toc

tic
%nonmax supression
thinner_imgXY = nonmax_supression(imgXY, normalized_angles);
thinner_imgXY = thinner_imgXY.*imgXY;
figure; imshow(thinner_imgXY); title('After nonmax supression');
toc

tic
%hythresis thresholding
%OTSU
[low_threshold, high_threshold] = otsu_thresholding(imgXY);
result_img = h_thresholding(thinner_imgXY, low_threshold, high_threshold);
figure; imshow(result_img); title('Final Image');
toc

%DONE CANNY EDGE DETECTOR IMPLEMENTATION
%IT CAN BE IMPROVED BY CHANGING NON-MAX SUPRESSION
%AND HYTHRESIS THRESHOLDING ALGORITHM (OTSU THRESHOLDING)
```

3.7.2.Pre-process Function (prepare_image.m)

```
%{
owner: mertkosan (Mert Kosan), mbenlioglu(Muhammed Mucahid Benlioglu)
created date: 09.01.2017

takes an image input and turns into gray scale if it is rgb, and turns
image pixel number uint8 to double
%}
function [last_img] = prepare_image(img)

if(size(img,3) == 3)
    img = rgb2gray(img);
end

img = im2double(img);
last_img = img;

end
```

3.7.3. Convolution Function (convolution.m)

```
%{
owner: mertkosan (Mert Kosan), mbenlioglu(Muhammed Mucahid Benlioglu)
created date: 09.01.2017

takes an image input and kernel with kernel size and make a convolution
operation image

normalized_kernel: all elements are divided by total of all elements, in
derivation case this total value is 1.

kernelsize: k value for 2k+1 x 2k+1 window
%}
function [out_img] = convolution(img, normalized_kernel, kernelsize)

[r,c] = size(img);
out_img = zeros(r,c);

for i=kernelsize+1:1:r-kernelsize
    for j=kernelsize+1:1:c-kernelsize
        subimage = img(i-kernelsize:i+kernelsize, j-kernelsize:j+kernelsize);
        out_img(i,j) = sum(sum(subimage.*normalized_kernel));
    end
end

end
```

3.7.4. Return Derivation Kernel Function (return_derivation_kernel.m)

```
%{
owner: mertkosan (Mert Kosan), mbenlioglu(Muhammed Mucahid Benlioglu)
created date: 09.01.2017

return kernel based on type
%}
function [kernel_x, kernel_y] = return_derivation_kernel(type, kernel_size)

narginchk(1,2);

%Sobel
sobel_kernel_x = [-1 0 1; -2 0 2; -1 0 1];
sobel_kernel_y = [1 2 1; 0 0 0; -1 -2 -1];

%Prewitt
prewitt_kernel_x = [-1 0 1; -1 0 1; -1 0 1];
prewitt_kernel_y = [1 1 1; 0 0 0; -1 -1 -1];

if(strcmp(type,'Sobel'))
    if nargin ~= 1
        if ~mod(kernel_size,2); kernel_size=kernel_size+1;end;
```

```
        for i=3:2:kernel_size-2
            sobel_kernel_x = 1/8 .* conv2([1 2 1]' * [1 2 1],
sobel_kernel_x);
            sobel_kernel_y = 1/8 .* conv2([1 2 1]' * [1 2 1],
sobel_kernel_y);
        end
    end
    kernel_x = sobel_kernel_x;
    kernel_y = sobel_kernel_y;
elseif(strcmp(type,'Prewitt'))
    if nargin ~= 1
        error('kernel_size paramater only valid for Sobel');
    end
    kernel_x = prewitt_kernel_x;
    kernel_y = prewitt_kernel_y;
else
    error('Wrong Kernel Type in : return_derivation_kernel!');
end

end
```

3.7.5. Normalize Directions Function (normalize_directions.m)

```
%{
owner: mertkosan (Mert Kosan), mbenlioglu(Muhammed Mucahid Benlioglu)
created date: 09.01.2017

takes a matrix which contains directions of the pixels in image, then at
the end, it turns angles into rounding angles with ignoring direction which
means for example 180 can be considered as 0. At the end there will be only
0, 45, 90 and 135 degrees for angles.
%}
function [normalized_angles] = normalize_directions(angles)

[r,c] = size(angles);
normalized_angles = zeros(r,c);

for i=1:1:r
    for j=1:1:c
       direction = angles(i,j);

       %normalize negative angles
       if(direction < 0); direction = direction + 360;
       end

       %normalize angle to nearest 45k angle
       if(direction > 337); normalized_angles(i,j) = 360;
       elseif(direction > 292); normalized_angles(i,j) = 315;
       elseif(direction > 247); normalized_angles(i,j) = 270;
       elseif(direction > 202); normalized_angles(i,j) = 225;
       elseif(direction > 157); normalized_angles(i,j) = 180;
       elseif(direction > 112); normalized_angles(i,j) = 135;
```

```
        elseif(direction > 77); normalized_angles(i,j) = 90;
        elseif(direction > 22); normalized_angles(i,j) = 45;
        else normalized_angles(i,j) = 0;
        end

        %normalize angles to 0,45,90,135, because k = k + 180 if you ignore
        %the direction of the angles.
        direction = normalized_angles(i,j);
        if(direction == 360 || direction == 180); normalized_angles(i,j) = 0;
        elseif(direction == 315); normalized_angles(i,j) = 135;
        elseif(direction == 270); normalized_angles(i,j) = 90;
        elseif(direction == 225); normalized_angles(i,j) = 45;
        end

    end
end
end
```

3.7.6. Non-max Supression Function (nonmax_suppression.m)

```
%{
owner: mertkosan (Mert Kosan), mbenlioglu(Muhammed Mucahid Benlioglu)
created date: 09.01.2017

takes input as ImgXY, which is combined image of derivatives of image,
and make edges thinner by nonmax_supression algorithm which main part of
Canny Implementation

takes input as normalized_angles which is result of normalize_directions.m
function
%}
function [thinner_ImgXY] = nonmax_suppression(ImgXY, normalized_angles)

[r,c] = size(ImgXY);
thinner_ImgXY = zeros(r,c);

for i=1+1:1:r-1
   for j=1+1:1:c-1
       current_value = ImgXY(i,j);
       switch normalized_angles(i,j)
           case 0
               max_value = max([current_value, ImgXY(i,j-1), ImgXY(i, j+1)]);
               if(current_value == max_value)
                   thinner_ImgXY(i,j) = 1;
               else
                   thinner_ImgXY(i,j) = 0;
               end
           case 45
               max_value = max([current_value, ImgXY(i+1,j-1), ImgXY(i-1,
j+1)]);
               if(current_value == max_value)
                   thinner_ImgXY(i,j) = 1;
               else
```

```
                thinner_ImgXY(i,j) = 0;
            end
        case 90
            max_value = max([current_value, ImgXY(i-1,j), ImgXY(i+1, j)]);
            if(current_value == max_value)
                thinner_ImgXY(i,j) = 1;
            else
                thinner_ImgXY(i,j) = 0;
            end
        case 135
            max_value = max([current_value, ImgXY(i-1,j-1), ImgXY(i+1,
j+1)]);
            if(current_value == max_value)
                thinner_ImgXY(i,j) = 1;
            else
                thinner_ImgXY(i,j) = 0;
            end
        otherwise
            thinner_ImgXY(i,j) = 0;
    end
  end
end

end
```

3.7.7. Otsu Thresholding Function (otsu_thresholding.m)

```
%{
owner: mertkosan (Mert Kosan), mbenlioglu(Muhammed Mucahid Benlioglu)
created date: 09.01.2017

    Modified OTSU algorithm by mbenlioglu and mertkosan
    THIS FUNCTION SHOULD BE IMPROVED!
%}
function [low_threshold, high_threshold] = otsu_thresholding(img)

[r,c] = size(img);
level = otsuthresh(imhist(img));

s = (r+c) /2;

if(s > 750)
    %%normalize level
    while level > 0.2
        level = level / 2;
    end
else
    %%normalize level
    while level > 0.1
        level = level / 2;
    end
end

high_threshold = level;
low_threshold = level / 2;
```

```
end
```

3.7.8. Hysteresis Thresholding Function (h_thresholding.m)

```
%{
owner: mertkosan (Mert Kosan), mbenlioglu(Muhammed Mucahid Benlioglu)
created date: 09.01.2017

takes an input as thinner_imgXY which is result of nonmax_supression.m
function, and we choose two threshold value eliminate low edges but not all
of them.
%}
function [result_img] = h_thresholding(thinner_imgXY, low_threshold,
high_threshold)

[r,c] = size(thinner_imgXY);
result_img = zeros(r,c);

maxvalue_in_img = max(max(thinner_imgXY));
low_threshold = low_threshold * maxvalue_in_img;
high_threshold = high_threshold * maxvalue_in_img;

for i=1:1:r
   for j=1:1:c
      value = thinner_imgXY(i,j);
      if(value >= high_threshold)
            result_img(i,j) = 1;
      elseif(value < high_threshold && value >= low_threshold)
          %look neighbours
          northwest = thinner_imgXY(i-1,j-1);
          west = thinner_imgXY(i,j-1);
          southwest = thinner_imgXY(i+1,j-1);
          south = thinner_imgXY(i-1,j);
          southeast = thinner_imgXY(i+1,j+1);
          east = thinner_imgXY(i,j+1);
          northeast = thinner_imgXY(i-1,j+1);
          north = thinner_imgXY(i-1,j);
          maxValue = max([northwest, west, southwest, south, southeast, east,
northeast, north]);
          if(maxValue >= high_threshold)
              result_img(i,j) = 1;
          end
      end
   end
end

%turn binary into 8bit image
result_img = uint8(result_img.*255);

end
```

References

Lucas, B. D., & Kanade, T. (1981). An Iterative Image Registration Technique with an Application to Stereo Vision (As cited in Wikipedia.org). *International Joint Conference on Artificial Intelligence*, (pp. 674-679).

PennState. (n.d.). Image Feature and Edge Detection. *Formal Design of an Optimal Edge Detector*.

Random sample consensus. (2017, January 11). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Random_sample_consensus

Shi, J., & Tomasi, C. (1994). Good Features to Track (As cited in Wikipedia.org). *IEEE Conference on Computer Vision and Pattern Recognition*, (pp. 593-600).

Strutz, T. (2016). *Data Fitting and Uncertainty (A practical introduction to weighted least squares and beyond) 2nd edition (As cited in Wikipedia.org).* Springer Vieweg.

Tomasi, C., & Kanade, T. (April 1991). Detection and Tracking of Point Features (As cited in Wikipedia.org). *Carnegie Mellon University Technical Report CMU-CS-91-132*.

Torr, P., & Zisserman, A. (200). MLESAC: A new robust estimator with application to estimating image geometry, Journal of Computer Vision and Image Understanding 78 (As cited in Wikipedia.org). 138-156.

Wikipedia. (n.d.). Canny edge detector.

Yang Tao, Q. Y.-h. (2015). *Improvement and Implementation for Canny Edge Detection.* Beijing: University of Chinese Academy of Sciences.

Zhou, P. Y. (2011). An Improved Canny Algorithm for Edge Detection. *Journal of Computational Information Systems*, 1516-1523.