IACV Homework Report

By: Mustafa Cagatay Sipahioglu

Person Code: 10899801

Enrollment Number: 220591

Table of Contents

Problem Statement

Scene. A **right circular cylinder** together with two (or more) circular **cross sections** and two (or more) **generatrix lines**. *A* right circular cylinder *is a surface, made of parallel lines, which is symmetric about a symmetry axis. A* circular cross section *of a right cylinder is a circumference centered on the symmetry axis and perpendicular to the symmetry axis. A* generatrix line *is a straight line, on the cylinder surface, which is parallel to its axis.*

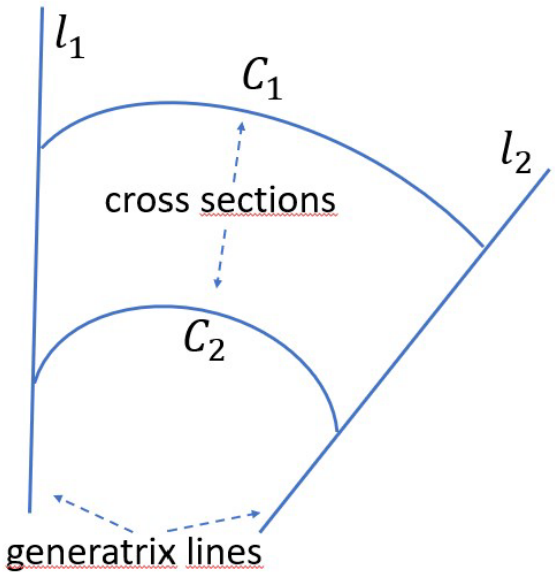


Image. A single image is taken of the above described cylinder by an uncalibrated, zero-skew, camera. (*Its calibration matrix depends on* **four** *unknown parameters, namely f_x , f_y and the two coordinates U_o, V_o of the principal point*). Two circular cross sections are visible, and their images C_1, C_2 are extracted. Two (parallel) generatrix lines of the cylinder are also visible, and their images l_1, l_2 are extracted.

Part 1: **Theory**

1. From C_1, C_2 find the horizon (vanishing) line h of the plane orthogonal to the cylinder axis.
2. From l_1, l_2, C_1, C_2 find the image projection a of the cylinder axis, and its vanishing point V.
3. From l_1, l_2, C_1, C_2 (and possibly h, a, and V), find the calibration matrix K.
4. From h, K, and V determine the orientation of the cylinder axis wrt the camera reference.
5. Compute the ratio between the radius of the circular cross sections and their distance.

Part 2: **Matlab**

1. Consider the image PalazzoTe.jpg. Using feature extraction techniques (**including** those implemented in **Matlab**) plus possible manual intervention, extract both the images l_1, l_2 of useful genetrix lines and images C_1, C_2 of useful circular cross sections.
2. Write a Matlab program that implements the solutions to problems 1 – 5
3. Rectification of a cylindric surface: Plot the **unfolding** of the part of the surface, included between the two cross sections, onto a plane.

Solutions to Questions 1.1, 1.2, 1.3, 1.4, 1.5 and 2.2 can be seen in sections 1, 2, 3, 4 and 5.
Solution to Question 2.1 can be found in sections 0A, 0B, 0C.
Solution to Question 3 can be found in section 3 (Rectification) and 6 (Unfolding).

Introduction

It is given that we are working on an image of a right cylinder. During our solution some lines and points on this cylinder will be useful. To establish a common nomenclature, the names given to these lines and points are shown in the images below both on the scene and the image.

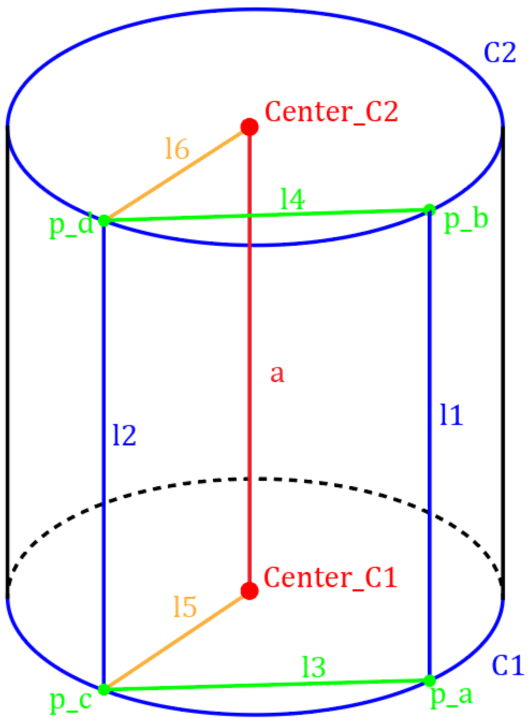


Figure 1: Right Cylinder Object (Scene)

Given an empty right cylinder, the ceiling image we are studying is defined by a surface section on the cylinder, between two arc pieces of circumferences C_1' and C_2' defined by points $p_a - p_c$ and $p_b - p_d$. It can be imagined that we are holding a camera to a right cylinder from below and seeing the farther surface of the cylinder from ourselves as depicted below.

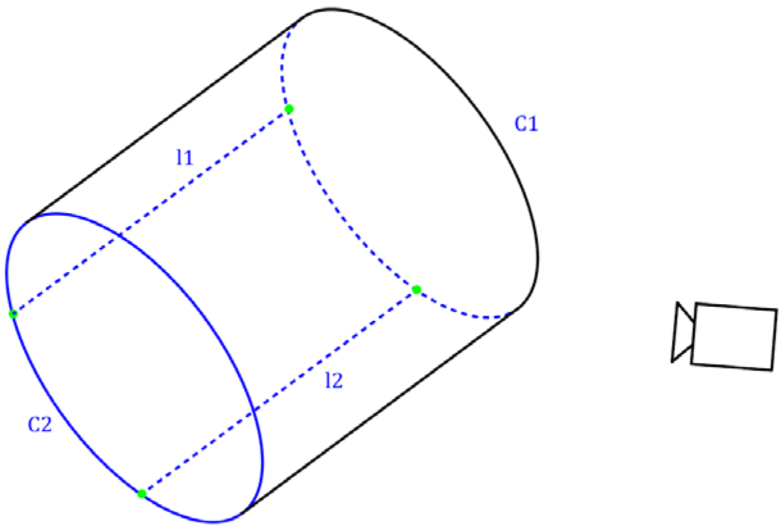


Figure 2: Qualitative Depiction of How the Object Was Photographed

The resulting image from such a scene can be depicted as below where V_i depicts the vanishing points found from the intersection of the similar colored parallel lines:

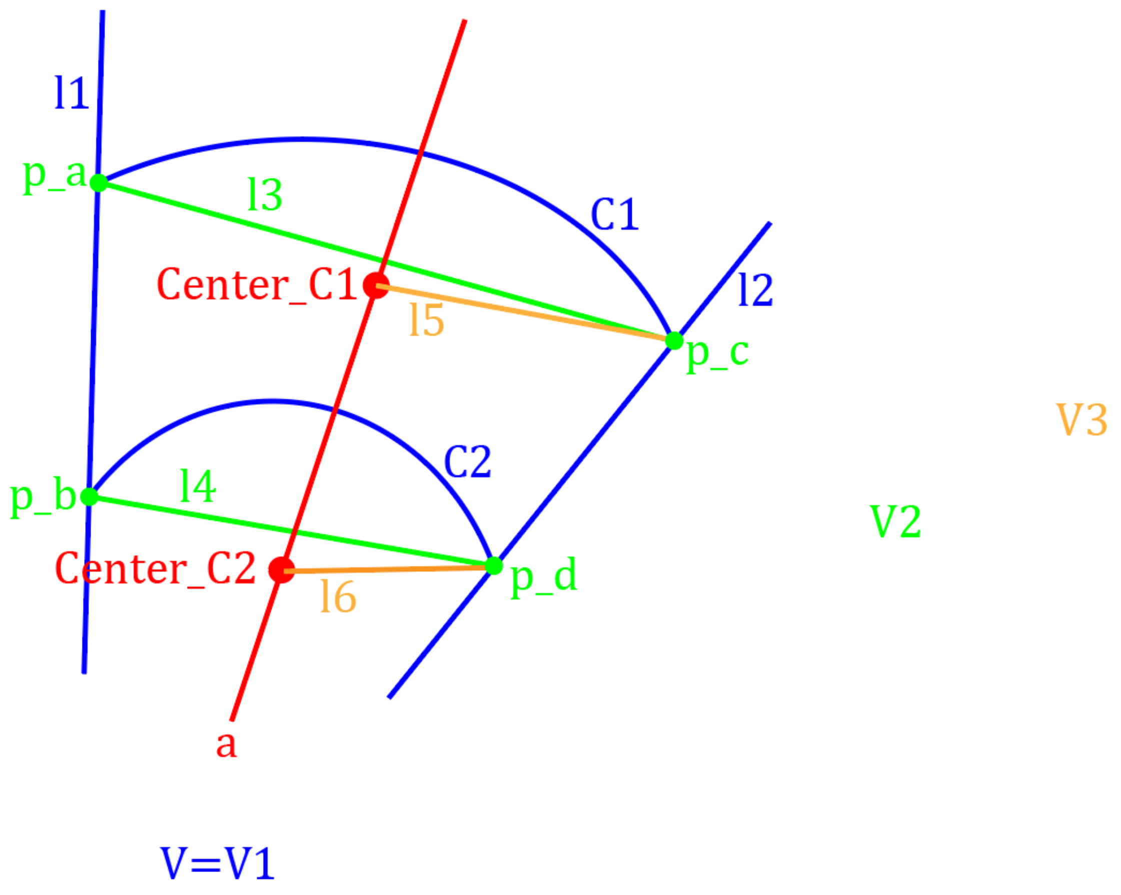


Figure 3: Defining Features of the Right Cylinder on the Image Plane

Having established a common nomenclature for most variable names, now we can start solving our problem. As to make this report easy to follow, the theoretical and MatLab solutions of each question is given simultaneously at their respective sections.

MATLAB VARIABLE NAMES: Please note that unless otherwise situated, IN THE MATLAB CODE every variable being talked about should be understood as the image version of that variable. So the variable $C1$ will correspond to the image of Conic 1 or variable I will correspond to the image of the circular point I.

0A. Feature Selection (P0A_FeatureSelector.m)

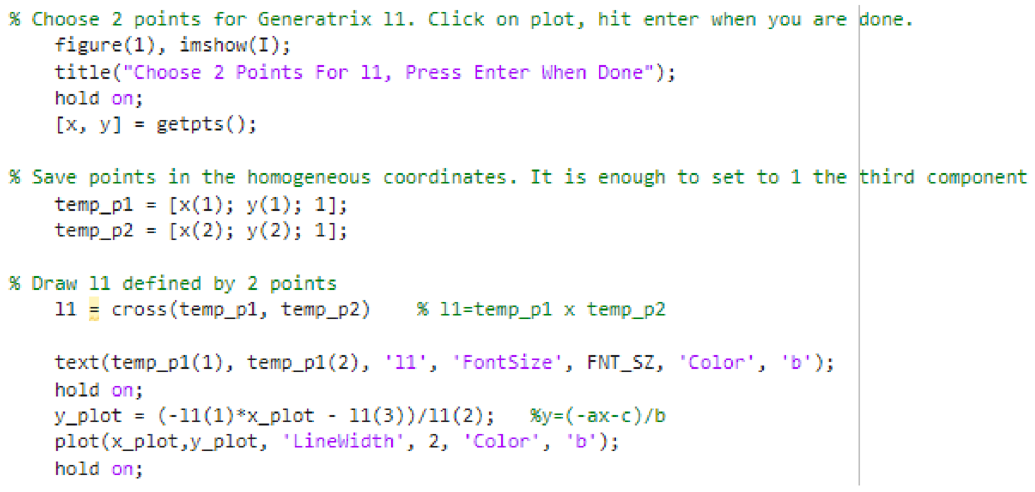
Visually inspecting our image, it is easy to see which edges can be selected to represent l_1', l_2', C_1', C_2'. A line equation is defined by 3 parameters and a conic equation is defined by 6. However, since we will express both in homogenous coordinates, by theory each can be represented by the ratio of their parameters. Therefore, the line equation is a 2DOF and the conic equation is a 5DOF system.

To define the lines l_1' *and* l_2', we can choose 2 points each on the image. Similarly, to find the conic parameters for the conic equations, we must choose 5 points on the conic image.

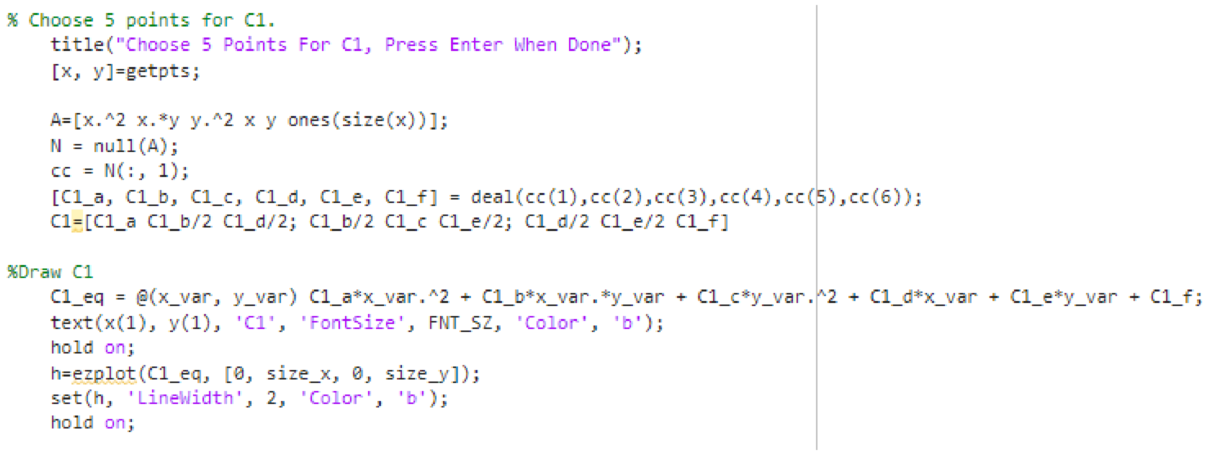
$$Line\ Equation: Ax + By + C = 0$$

$$Conic\ Equation: Ax^2 \ + Bxy + Cy^2 + Dx + Ey + F = 0$$

To solve for these parameters, we must choose 2 points on the lines and 5 points on the conics. Following this we can define the line equation easily by defining a line by the cross product of collinear point coordinates in homogenous coordinates. And we can define the conic equation by solving its equation symbolically for its parameters such that all 5 points chosen on the conic image satisfies its equation. These calculations can be done in MatLab as follows:



```
% Choose 2 points for Generatrix l1. Click on plot, hit enter when you are done.
    figure(1), imshow(I);
    title("Choose 2 Points For l1, Press Enter When Done");
    hold on;
    [x, y] = getpts();

% Save points in the homogeneous coordinates. It is enough to set to 1 the third component
    temp_p1 = [x(1); y(1); 1];
    temp_p2 = [x(2); y(2); 1];

% Draw l1 defined by 2 points
    l1 = cross(temp_p1, temp_p2)    % l1=temp_p1 x temp_p2

    text(temp_p1(1), temp_p1(2), 'l1', 'FontSize', FNT_SZ, 'Color', 'b');
    hold on;
    y_plot = (-l1(1)*x_plot - l1(3))/l1(2);   %y=(-ax-c)/b
    plot(x_plot,y_plot, 'LineWidth', 2, 'Color', 'b');
    hold on;
```



```
% Choose 5 points for C1.
    title("Choose 5 Points For C1, Press Enter When Done");
    [x, y]=getpts;

    A=[x.^2 x.*y y.^2 x y ones(size(x))];
    N = null(A);
    cc = N(:, 1);
    [C1_a, C1_b, C1_c, C1_d, C1_e, C1_f] = deal(cc(1),cc(2),cc(3),cc(4),cc(5),cc(6));
    C1=[C1_a C1_b/2 C1_d/2; C1_b/2 C1_c C1_e/2; C1_d/2 C1_e/2 C1_f]

%Draw C1
    C1_eq = @(x_var, y_var) C1_a*x_var.^2 + C1_b*x_var.*y_var + C1_c*y_var.^2 + C1_d*x_var + C1_e*y_var + C1_f;
    text(x(1), y(1), 'C1', 'FontSize', FNT_SZ, 'Color', 'b');
    hold on;
    h=ezplot(C1_eq, [0, size_x, 0, size_y]);
    set(h, 'LineWidth', 2, 'Color', 'b');
    hold on;
```

0B. Feature Extraction

For the feature extraction on this image, Canny Edge detection algorithm was used as follows. Firstly, the image was imported into MatLab. (The image rotation is done simply because MatLab can't read rotation metadata of the image and was importing it incorrectly) Then the image was converted to greyscale to simplify the edge detection process (Reducing color based noise, focusing on intensity and/or luminosity changes) and lastly "edges" were obtained using the built in Canny edge detection algorithm. Using the bwboundaries function we can convert the variable "edges" into an array of B where each element holds the set of coordinates for many points used to define some boundary.

```
% Rotate the image by -90 degrees using bilinear interpolation
originalImage = imrotate(imread('PalazzoTe.jpg'), -90, 'bilinear');

% Convert the rotated image to grayscale
grayImage = rgb2gray(originalImage);

% Perform edge detection using the Canny edge detector
edges = edge(grayImage, 'Canny');
[B, L] = bwboundaries(edges, 'noholes');
```

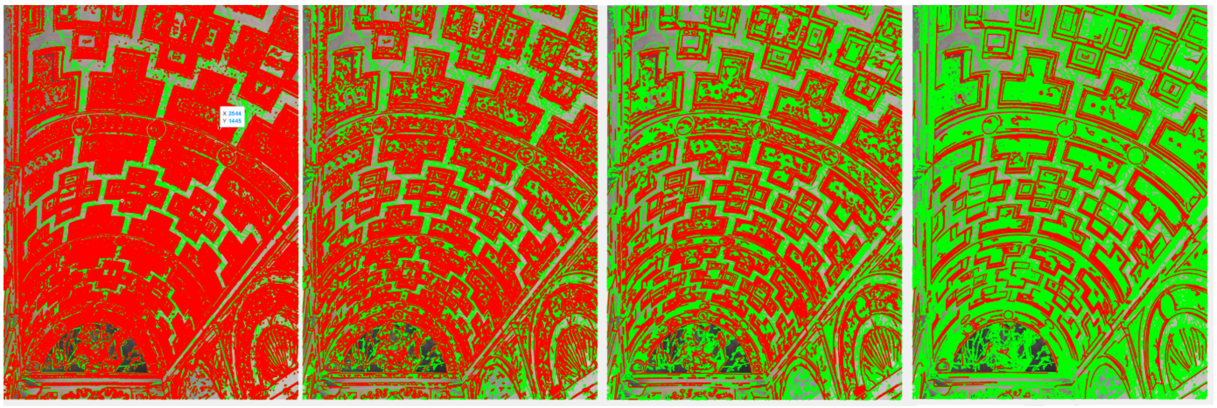
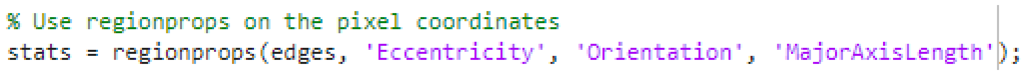


Figure 4: Edge Detection (Red) and Ellipse Detection (Green). Left the right: Edges with More Points Are Filtered.

Looking at Figure 4, we can see a blind attempt at trying to filter the extracted edges and ellipses with respect to the amount of points each edge is defined by. From such an attempt we can still learn a few things:

- The B variable blindly stores point arrays. We need to find a better way to measure each edge other than their array length.
- Since edges are just a group of points they can be any arbitrary shape, including conic-like, we do not necessarily need an ellipse detection algorithm.
- The features of the ceiling that look like crosses constitute edges with many points. The array length of each edge is a bad measure since it mostly filters these crosses.

Each element of the array B is an array of point coordinates that when connected define a boundary in the image. This distinction doesn't have to be in the shape of a line or doesn't even have to have a well-known geometric shape neither. However, each of these boundaries have some general geometric properties as shown below. These geometric properties can be extracted using the "regionprops" function in MatLab and they will be essential to how we process the useful edges.



```
% Use regionprops on the pixel coordinates
stats = regionprops(edges, 'Eccentricity', 'Orientation', 'MajorAxisLength');
```

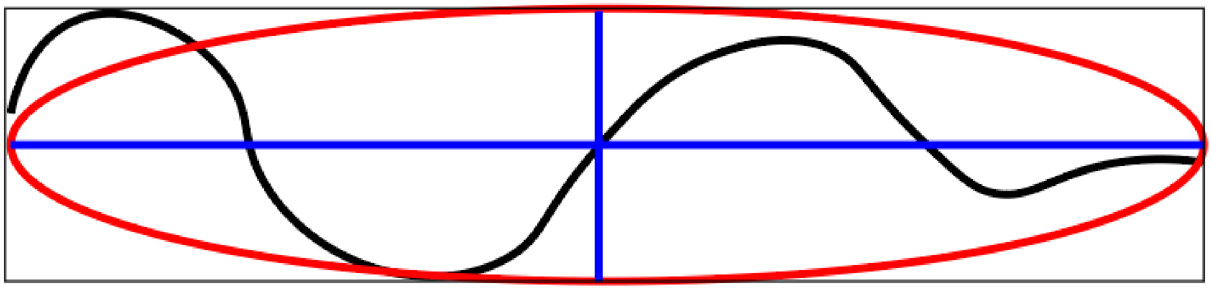


Figure 5: Arbitrary Boundary (Black); Ellipse Fitting in the Same Rectangle as the Edge (Red); Major and Minor Axes (Blue)

Looking at Figure 5, it is easier to imagine what some of these properties we can utilize are. Any edge can be treated using the parameters of the ellipse that has the same dimensions as the edge limits. (Such that they fit in the same rectangle) Some such properties are the eccentricity of this equivalent ellipse, the pixel length of the Major and Minor axes of this ellipse, and the orientation of the containing rectangle with respect to the horizontal image axis. The related documentation of “regionprops” can be read for the plethora of other properties that can be calculated for each edge, but these 3 properties were enough to extract l_1', l_2', C_1', C_2' from our image.

P0B_FeatureExtractor_Line.m: Now that we have extracted the properties for all our edges we can start eliminating them. In order to find the 2 edges that will define l_1', l_2' the following algorithm was enforced:

- Check for edges with eccentricity>0.995. (Eccentricity 0 represents a circle; Eccentricity 1 represents a line. We would like to eliminate all non "line like" edges by enforcing this property.)
- Within these "line like" edges:
    - Use the longest edge (Check Major Axis Length for comparison) with orientation between -85 and 80 degrees to define l_1'.
    - Use the longest edge (Check Major Axis Length for comparison) with orientation between 45 and 60 degrees to define l_2'.

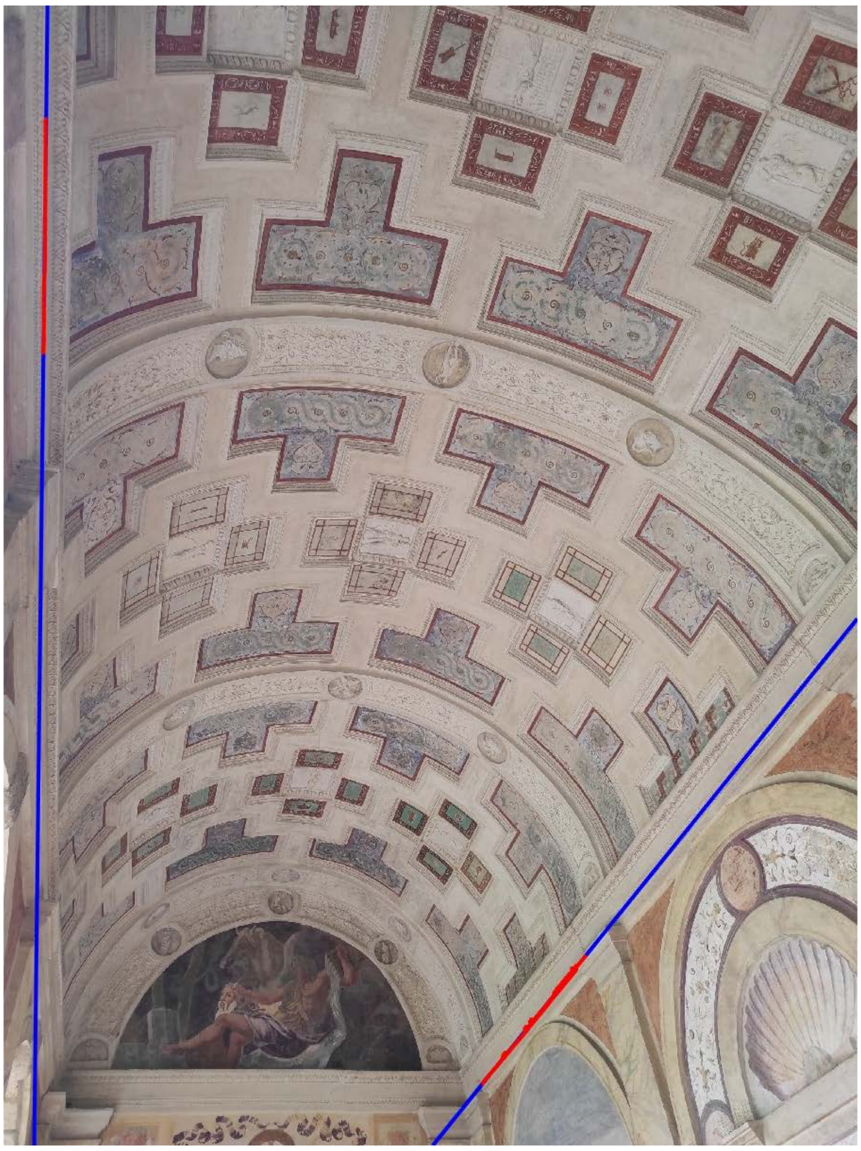


Figure 6: Extracted "Line Like" Edges (Red), Lines Fitted to These Edges (Blue)

Figure 6 shows the longest edges in each target orientation used to define the generatrix lines in red. After each of the longest edge in the desired region was found, a line was fitted to the coordinate arrays of these edges using "polyfit" in order to obtain the line parameters. The lines defined by these parameters are given in blue.

P0B_FeatureExtractor_Conic.m: Extracting C_1' and C_2' was relatively easier. After eliminating all the edges with their Major Axis Length smaller than 2000 pixels, exactly 2 edges were left behind as shown in Figure 7 in red.

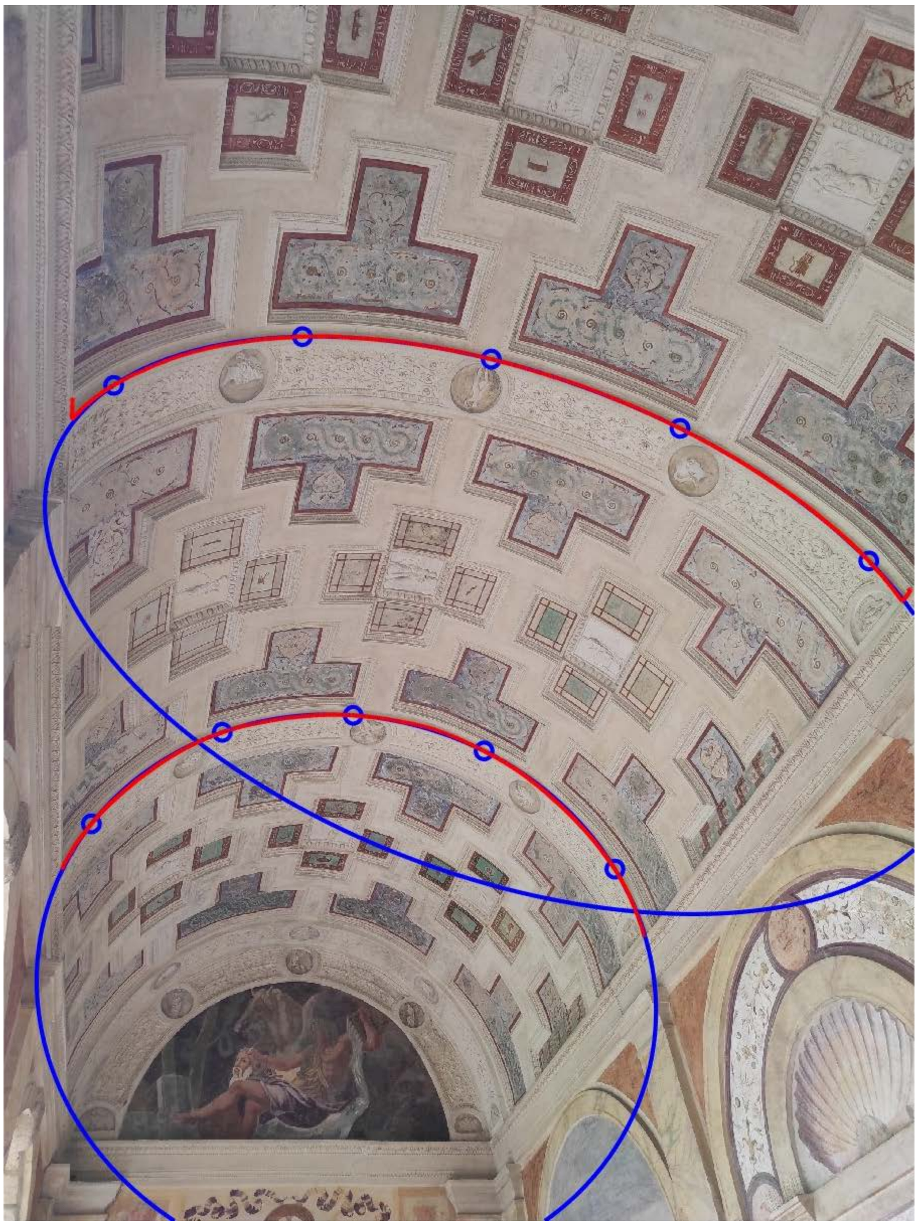


Figure 7: Extracted Edges with MajorAxisLength>2000 (Red), 5 Equidistant Points Chosen from These Edges (Small Blue Circles), Conics Fitted to These Points (Blue Lines)

After these edges were extracted 2 conics representing these elliptical looking edges must be found. Looking at the extracted edge near the top of Figure 7, it can be seen that this edge contains non-elliptical looking pieces at the start and the end. We can visually see that these points should not belong in our final conic. Due to these outlier points, fitting a conic by error minimization to the edge points will result in a faulty fitting. Therefore, an alternative approach was applied to fit the conics to these edges. 5 points equidistant on the x axis were sampled in the middle 90% portion of the red extracted edges. (No sampling was done in the beginning and the end of the edges to eliminate the outliers, sampling was done equidistantly to represent the conic better.) Using these 5 sample points for each conic, a conic was fitted with the same methodology used in the **P0A_FeatureSelector.m** case.

How Was the Elimination Thresholds Tuned?

In our opinion, the tuning of the elimination parameters is the biggest weakness of the Feature Extraction method as compared to the Feature Selection method. For example, in the extraction of conic looking edges it was quite convenient for the MajorAxisLength>2000 condition to leave us with the exact conic-looking edges we were looking for. To get the threshold of 2000 we had to increase the threshold by trial and error while checking the extracted edges by plotting. And if a single elimination parameter was not enough, we would have had to filter the extracted edges even more to eventually end up with our conic edges.

Another example can be given for the line extraction. We have initially tried higher eccentricity thresholds, even eccentricity=1 however this would filter edges that were very short and were not on the generatrix lines. This threshold was gradually reduced until we were left with many line-looking edges. Then these edges had to be further filtered using the orientations (Thresholds given initially by visual inspection and then fine-tuned by trial and error) and optimized using the major-axis lengths.

All in all, tuning the feature extraction thresholds and functions takes a lot of effort. The only advantage is that we can extract better l_1', l_2', C_1', C_2' parameters (In theory) than when done by Feature Selection instead. But even this is not guaranteed since there can be fitting errors after the correct edges were found (Like the outlier case that had to be fixed for conic fitting) or the correct edges to fit our lines/conics may be very hard to find.

0C. Choosing Between the Selected and Extracted Features

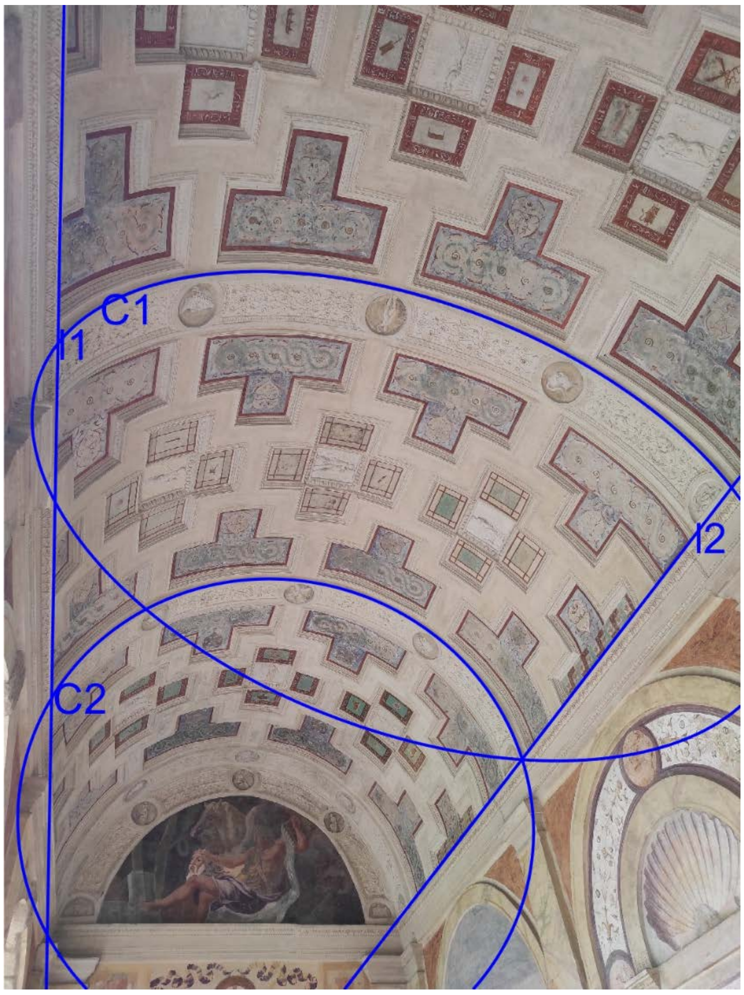
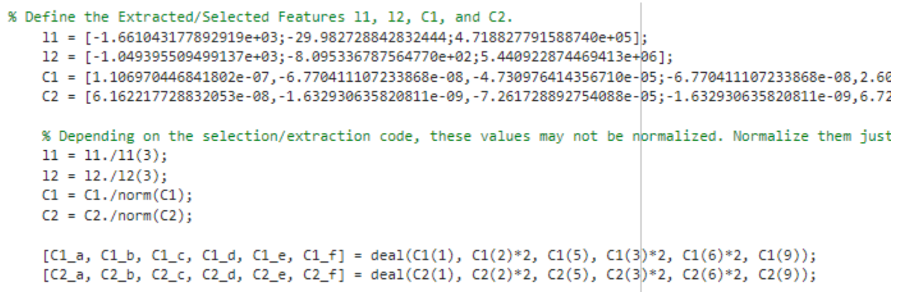


Figure 8: Feature Selected Image

In the end, the Feature Selection method was used to define the parameters of l_1', l_2', C_1', C_2'. This was done because the extracted l_1' seen in Figure 6 seemed doubtfully a good fit. (Even though it is pretty good, it seems to diverge from where the generatrix line is supposed to follow nearing the bottom of the image. We can speculate that this is because the extracted line is near the top of the image which do not offer a good extrapolation nearing the bottom of the image. In contrast the two points used to fit l_1' in the Feature Selection case were chosen carefully near the middle of the image with a large distance between them.)

The l_1', l_2', C_1', C_2' parameters calculated in FeatureSelector.m are copied and pasted to the beginning of our **P1_2_3_main.m** main code. They are normalized anyway since these data could have been obtained from many different functions that do not necessarily output in homogenous coordinates.



```
% Define the Extracted/Selected Features l1, l2, C1, and C2.
    l1 = [-1.661043177892919e+03;-29.982728842832444;4.718827791588740e+05];
    l2 = [-1.049395509499137e+03;-8.095336787564770e+02;5.440922874469413e+06];
    C1 = [1.106970446841802e-07,-6.770411107233868e-08,-4.730976414356710e-05;-6.770411107233868e-08,2.60
    C2 = [6.162217728832053e-08,-1.632930635820811e-09,-7.261728892754088e-05;-1.632930635820811e-09,6.72

    % Depending on the selection/extraction code, these values may not be normalized. Normalize them just
    l1 = l1./l1(3);
    l2 = l2./l2(3);
    C1 = C1./norm(C1);
    C2 = C2./norm(C2);

    [C1_a, C1_b, C1_c, C1_d, C1_e, C1_f] = deal(C1(1), C1(2)*2, C1(5), C1(3)*2, C1(6)*2, C1(9));
    [C2_a, C2_b, C2_c, C2_d, C2_e, C2_f] = deal(C2(1), C2(2)*2, C2(5), C2(3)*2, C2(6)*2, C2(9));
```

1. Horizon Line h Orthogonal to Cylinder Axis

Horizon/Vanishing line is a property of the images of plane scenes or a scene made up of parallel planes. Starting from a simpler case: Parallel lines on a scene intersect at a vanishing point in the image. So the intersection point of parallel lines maps to a point. Similarly, if we expand our definition to a higher dimension, it can be seen that random sets of parallel lines on a plane or parallel sets of lines in parallel planes intersect in multiple vanishing points in the image. All of these vanishing points can be observed to be on the same line, which defines the horizon/vanishing line. Similar to the mapping of the intersection of parallel lines being the vanishing point, intersection of parallel planes map to a line in the image, the horizon line.

We would like to find the horizon line orthogonal to the cylinder axis. This means the horizon line defined by the parallel cross sectional planes of the cylinder. Now using our definition above, this can be achieved by finding multiple vanishing points of parallel lines on these cross sections, however in our case there is an easier solution. Since we have 2 conic images of the circumferential cross sections of the scene, we can use another piece of special information: Conic images of circumferences on the same or parallel planes cross the horizon line at the images of the circular points. The images of the circular points are both on the horizon line and the conic images all intersect at these circular points. This means that the line passing through the intersection points of conics C_1' and C_2' gives us the horizon line.

The conics are defined by the following equations:

$$Conic\ 1: A_{C1}x^2 + B_{C1}xy + C_{C1}y^2 + D_{C1}x + E_{C1}y + F_{C1} = 0$$

$$Conic\ 2: A_{C2}x^2 + B_{C2}xy + C_{C2}y^2 + D_{C2}x + E_{C2}y + F_{C2} = 0$$

The (x,y) coordinates of the intersection points (images of circular points I' and J' and 2 more) satisfies both of these equations simultaneously. As a matter of fact as defined by the Bezout's theorem there are 4 (x,y) values simultaneously satisfying both equations, 4 intersection points. Furthermore, if only 2 of these 4 intersection points happen to be complex conjugates, then these two points indeed are the images of the circular points. (Bezout's theorem defines how to recognize which 2 solutions correspond to the images of the circular points if this condition wasn't satisfied as well. However, since we indeed get only 1 pair of unique complex conjugates in our case, this is the only part of the Bezout's theorem we will be using.)

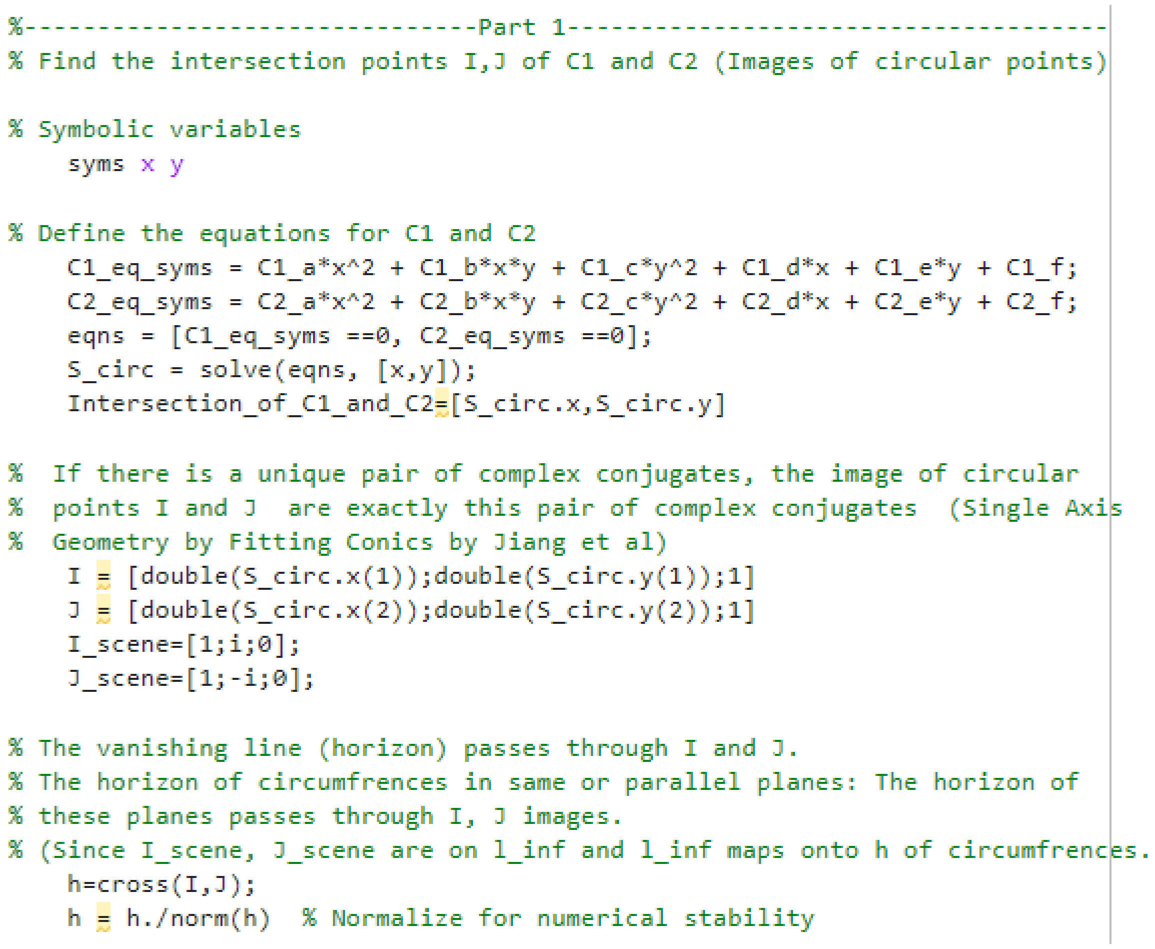
$$I', J'\ is\ the\ complex\ conjugate\ pair\ of\ solutions\ to\ the\ intersection\ of\ C_1'\ and\ C_2'$$

Finally, as defined in the second paragraph of this section, horizon line h is the line passing through both these points of the images of the circular points:

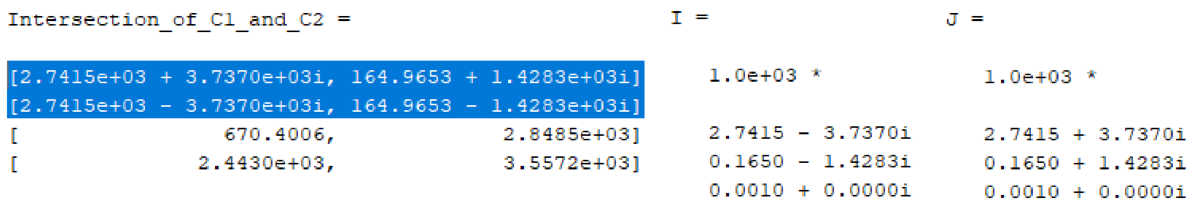
$$h = I' \times J'$$

Q1 MatLab Implementation (P1_2_3_main.m)

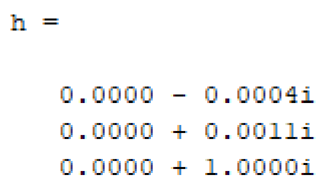
The MatLab implementation follows the theory solution without any problems. It can be noted that whenever the (x,y) points of intersection for the images of the circular points are found the points are defined in homogenous coordinates: as the linear algebra we are going to use for image processing is defined for homogeneous coordinates.



```
%------------------------------Part 1-----------------------------------
% Find the intersection points I,J of C1 and C2 (Images of circular points)

% Symbolic variables
    syms x y

% Define the equations for C1 and C2
    C1_eq_syms = C1_a*x^2 + C1_b*x*y + C1_c*y^2 + C1_d*x + C1_e*y + C1_f;
    C2_eq_syms = C2_a*x^2 + C2_b*x*y + C2_c*y^2 + C2_d*x + C2_e*y + C2_f;
    eqns = [C1_eq_syms ==0, C2_eq_syms ==0];
    S_circ = solve(eqns, [x,y]);
    Intersection_of_C1_and_C2=[S_circ.x,S_circ.y]

%  If there is a unique pair of complex conjugates, the image of circular
%  points I and J  are exactly this pair of complex conjugates  (Single Axis
%  Geometry by Fitting Conics by Jiang et al)
    I = [double(S_circ.x(1));double(S_circ.y(1));1]
    J = [double(S_circ.x(2));double(S_circ.y(2));1]
    I_scene=[1;i;0];
    J_scene=[1;-i;0];

% The vanishing line (horizon) passes through I and J.
% The horizon of circumfrences in same or parallel planes: The horizon of
% these planes passes through I, J images.
% (Since I_scene, J_scene are on l_inf and l_inf maps onto h of circumfrences.
    h=cross(I,J);
    h = h./norm(h)  % Normalize for numerical stability
```



```
Intersection_of_C1_and_C2 =                        I =                J =

[2.7415e+03 + 3.7370e+03i, 164.9653 + 1.4283e+03i]    1.0e+03 *          1.0e+03 *
[2.7415e+03 - 3.7370e+03i, 164.9653 - 1.4283e+03i]
[              670.4006,             2.8485e+03]    2.7415 - 3.7370i   2.7415 + 3.7370i
[            2.4430e+03,             3.5572e+03]    0.1650 - 1.4283i   0.1650 + 1.4283i
                                                    0.0010 + 0.0000i   0.0010 + 0.0000i
```



```
h =

    0.0000 - 0.0004i
    0.0000 + 0.0011i
    0.0000 + 1.0000i
```

2. Image Projection of the Cylinder Axis a' and Its Vanishing Point V

A projection of an image protects collinearity. We can use this principle to find the image of the cylinder axis: a. We know that the cylinder axis passes through the centers of the circumferences $Center_{C_1}$ and $Center_{C_2}$. Then since collinearity is preserved, the image of the cylinder axis must pass through the images of the centers of the circumferences $Center'_{C_1}$ and $Center'_{C_2}$. It is easy to calculate a center of a conic using the analytical formula given below:

$$Center\ of\ a\ Conic = (\frac{BE-2CD}{4AC-B^2}, \frac{BD-2AE}{4AC-B^2})$$

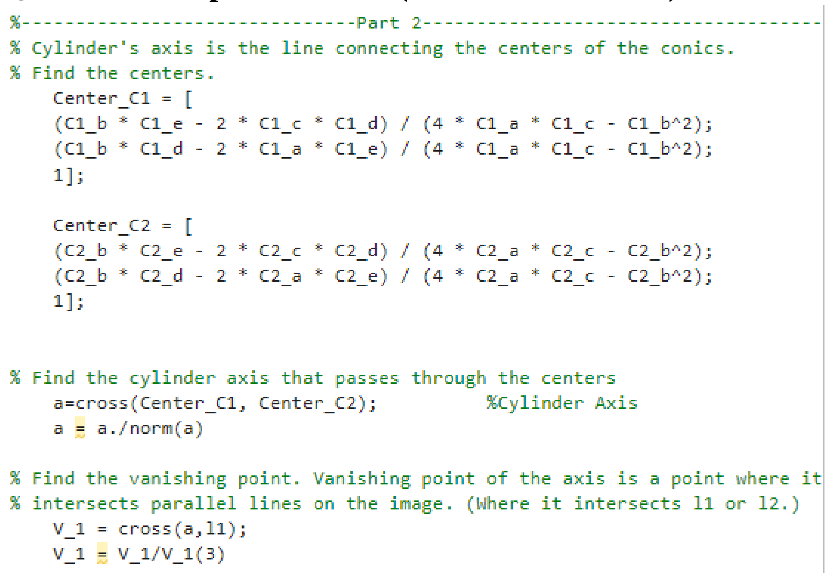
The image of the cylinder axis a'can be defined by the cross product of the images of the centers. (2 points on the image plane define a line via their cross product)

$$a' = Center_{C_1} \times Center_{C_2}$$

The vanishing point V of the image of the cylinder axis, can be found at the point the axis intersects a parallel line. Looking at Figure 1, we can see that a' is parallel with l'_1 and l'_2. Let's use the intersection with l'_1 to find V. (If our feature extractions/selections were perfect, a' would intersect both l'_1 and l'_2 at the same point, however since it is not a perfect there is a miniscule difference between where a' intersects l'_1 and l'_2. Therefore 1 of the lines must be chosen for calculation)

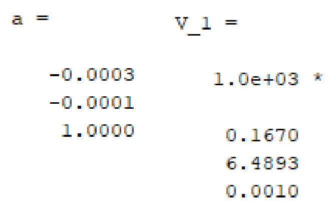
$$V = a' \times l'_1 = V_1$$

Q2 MatLab Implementation (Part1_2_3_main.m)



```
%------------------------------Part 2-----------------------------------
% Cylinder's axis is the line connecting the centers of the conics.
% Find the centers.
    Center_C1 = [
    (C1_b * C1_e - 2 * C1_c * C1_d) / (4 * C1_a * C1_c - C1_b^2);
    (C1_b * C1_d - 2 * C1_a * C1_e) / (4 * C1_a * C1_c - C1_b^2);
    1];

    Center_C2 = [
    (C2_b * C2_e - 2 * C2_c * C2_d) / (4 * C2_a * C2_c - C2_b^2);
    (C2_b * C2_d - 2 * C2_a * C2_e) / (4 * C2_a * C2_c - C2_b^2);
    1];


% Find the cylinder axis that passes through the centers
    a=cross(Center_C1, Center_C2);          %Cylinder Axis
    a = a./norm(a)

% Find the vanishing point. Vanishing point of the axis is a point where it
% intersects parallel lines on the image. (Where it intersects l1 or l2.)
    V_1 = cross(a,l1);
    V_1 = V_1/V_1(3)
```

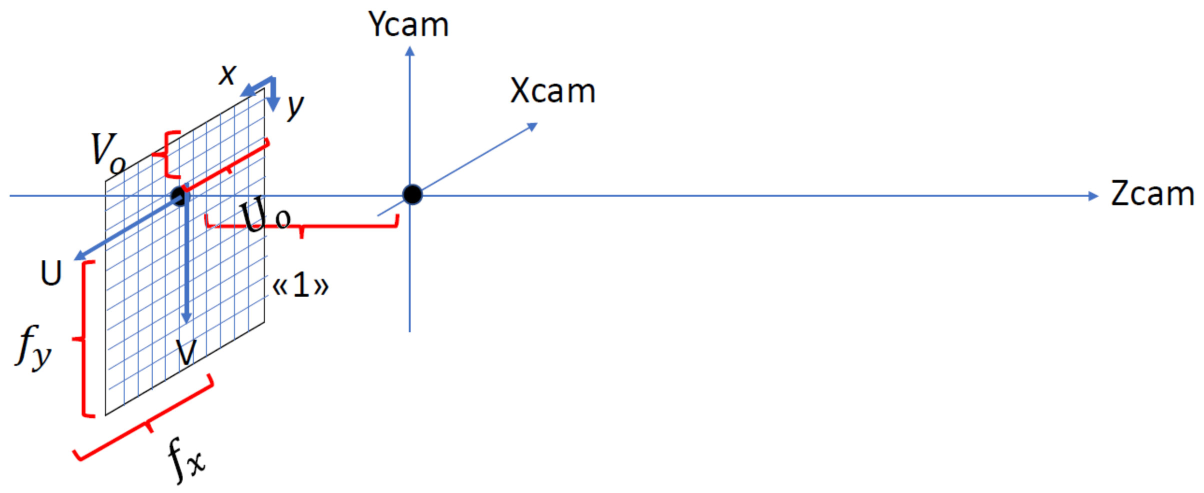
The MatLab implementation follows the theory without deviations. As always the points found from cross products are normalized to keep the homogenous coordinate form.



```
a =              V_1 =

   -0.0003          1.0e+03 *
   -0.0001
    1.0000           0.1670
                     6.4893
                     0.0010
```

3. Calibration Matrix *K*

Given the zero skew camera configuration in the camera coordinate system below:



The K matrix is defined as seen in the following equation. Please notice that this matrix is normalized with respect to the focal length f_z. (Such that $f_z = 1$ and 1 is equal to f_x horizontal or f_y vertical pixels.)

$$K = \begin{bmatrix} f_x & 0 & U_0 \\ 0 & f_y & V_0 \\ 0 & 0 & 1 \end{bmatrix}$$

To find the calibration matrix K we can start by finding the image of the absolute conic: w. This is beneficial since K and w are related with $w^{-1} = KK^T$ and we can find K using the Cholesky factorization of w. This means that we need to find w now. Luckily w is a very fundamental variable that comes up in many geometric relations. Firstly in a camera with zero skew w is defined by such a matrix as given below.

$$w = \begin{bmatrix} w_{11} & 0 & w_{13} \\ 0 & w_{11} & w_{23} \\ w_{13} & w_{23} & w_{33} \end{bmatrix}$$

This means that if we can find 4 linearly independent linear equations for w, we can solve this equation system to obtain w. As a matter of fact since we will be finding w in homogenous coordinates it will be normalized, therefore its DOF can be represented by the ratios of its parameters. (Similar to how we reduced a DOF for homogenous coordinate lines and conics in section “0A”) This means that we have a net DOF of 3 and 3 linear, linearly independent equations are enough to solve for w.

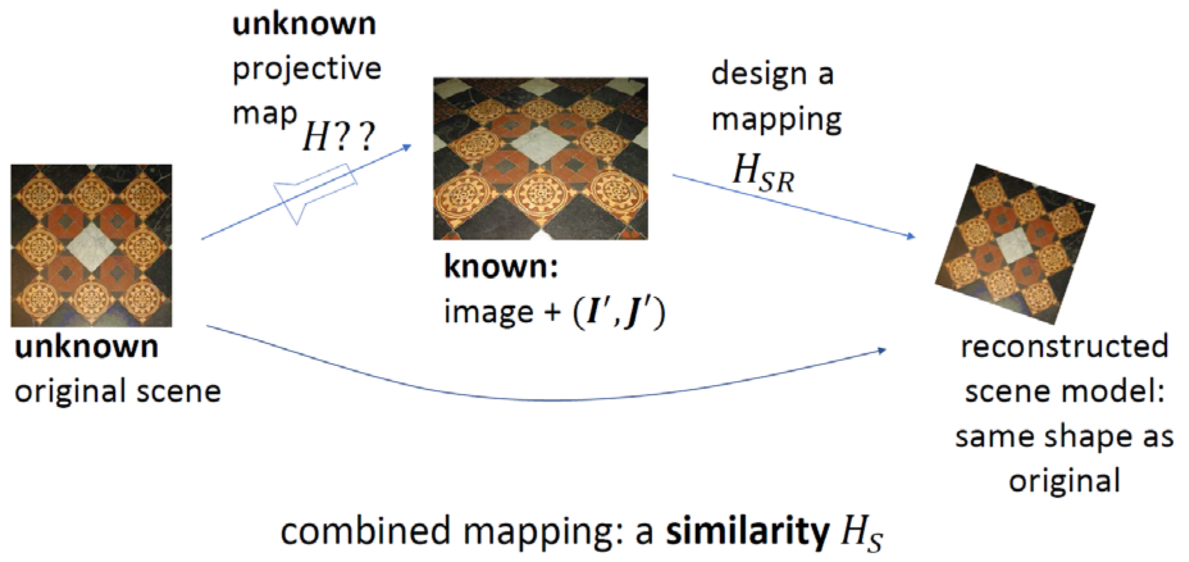


Figure 9: Definition of the 2D Reconstruction Problem

At this point it is beneficial to talk about the 2D reconstruction problem of which's overview can be seen in Figure 9. The problem goes as follows: We have got a scene (In our case cross section plane of a cylinder) and we take a photograph of this scene resulting in a 2D image. There exists a projective homography called H that maps any point on the original planar scene to the 2D image. This homography H is extremely hard to find and therefore it is very unlikely for us to reconstruct our image to how it was looking in the scene. However, it is possible if we want to reconstruct a model similar to the original scene. (May be scaled, may be rotated but the original planar scene will be understandable with the human eye.)

We know that the original scene and the model of the scene we are trying to construct are similar (Mapped by similarity mapping H_S) And we know that similarity mappings do not change the circular points. This means that the reconstructed model of the scene contains the original circular points $I\ and\ J$ and therefore any reconstruction homography $H_{SR} = H_{rect}$ will map $I' and\ J'$ back to $I\ and\ J$.

$$H_{rect} I' = I = \begin{bmatrix} 1 \\ i \\ 0 \end{bmatrix}$$

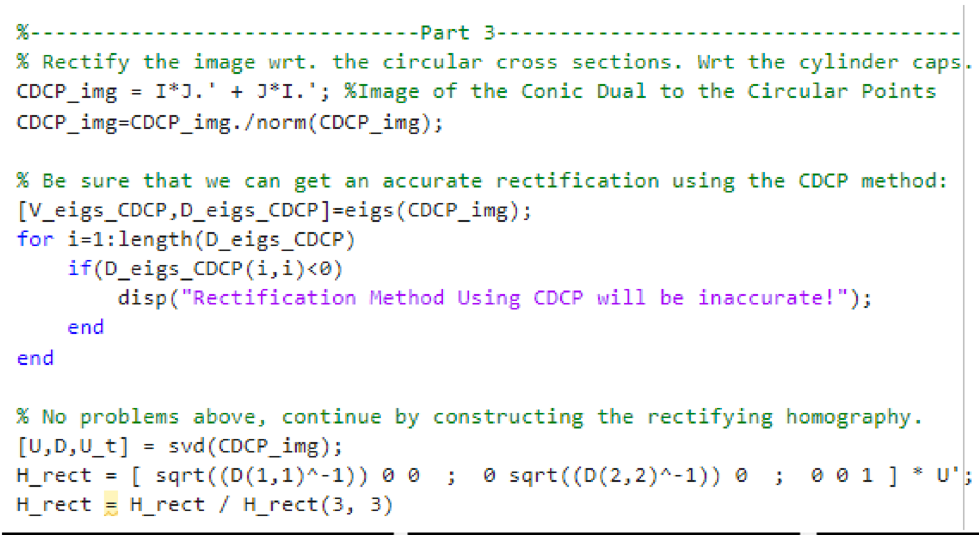
This fact can be extended over to the conic dual to the circular points and its image; and a reverse engineering method as follows, allows us to calculate H_{rect} if we know the images of the circular points.

$$C_\infty^{*\prime} = I' J'^T + J' I'^T$$

$$svd\left(C_\infty^{*\prime}\right) = U \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 0 \end{bmatrix} U^T = H_{rect}^{-1} C_\infty^* H_{rect}^{-T}$$

$$H_{rect} = \begin{bmatrix} 1/\sqrt{a} & 0 & 0 \\ 0 & 1/\sqrt{b} & 0 \\ 0 & 0 & 1 \end{bmatrix} U^T$$

In our case we indeed know the images of the circular points for the cross sectional planes. Therefore any H_{rect} we will calculate using the following methodology will be rectifying our image with respect to the cross sectional planes. (After rectification we should be looking squarely at the cross sections of the scene)



```
%-------------------------------Part 3-------------------------------------
% Rectify the image wrt. the circular cross sections. Wrt the cylinder caps.
CDCP_img = I*J.' + J*I.'; %Image of the Conic Dual to the Circular Points
CDCP_img=CDCP_img./norm(CDCP_img);

% Be sure that we can get an accurate rectification using the CDCP method:
[V_eigs_CDCP,D_eigs_CDCP]=eigs(CDCP_img);
for i=1:length(D_eigs_CDCP)
    if(D_eigs_CDCP(i,i)<0)
        disp("Rectification Method Using CDCP will be inaccurate!");
    end
end

% No problems above, continue by constructing the rectifying homography.
[U,D,U_t] = svd(CDCP_img);
H_rect = [ sqrt((D(1,1)^-1)) 0 0  ;  0 sqrt((D(2,2)^-1)) 0  ;  0 0 1 ] * U';
H_rect = H_rect / H_rect(3, 3)
```

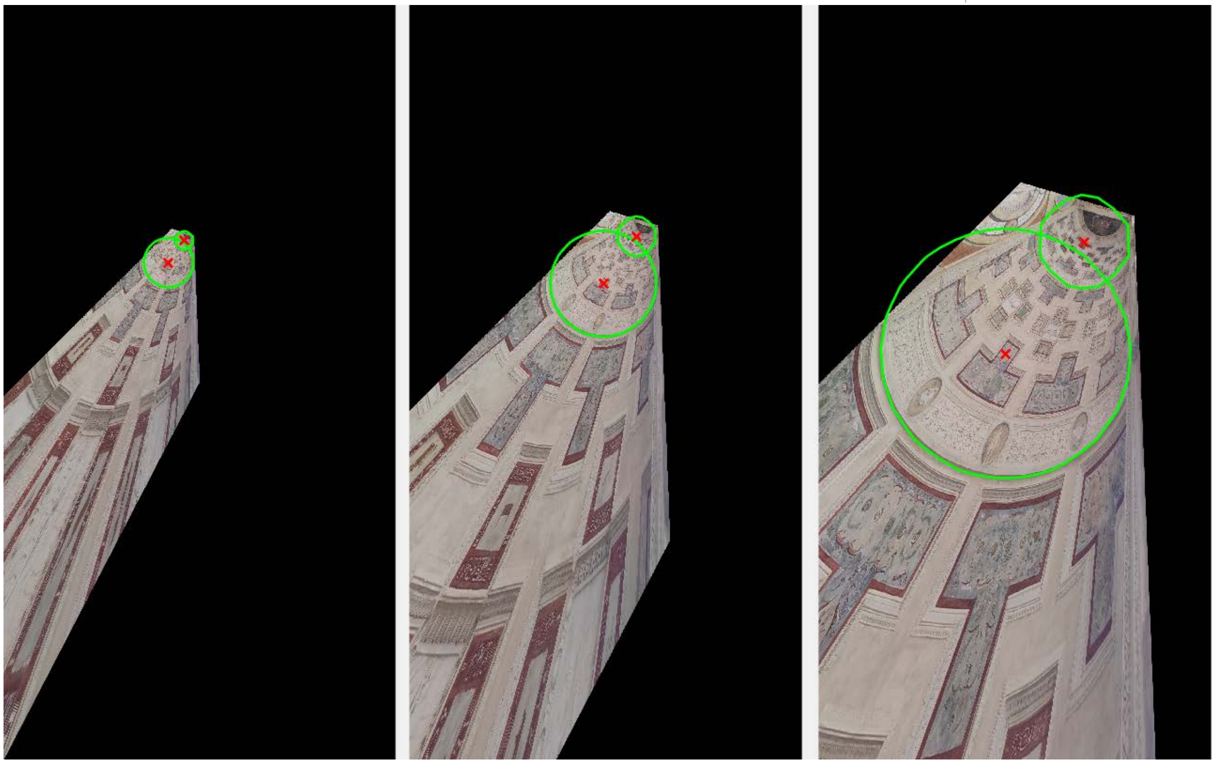


Figure 10: Rectified Image and Rectified Conics. Left to Right: More Zoomed In

Figure 10 shows the results of the rectification and we are indeed looking at the cylinder's cross sections squarely. We can also see that the conics were mapped to circumferential images as expected. (Please note that in the MatLab code, when plotting: the rectified conics had to be manually shifted to compensate for the automatic shifting of the rectified image by the "imtransform" function. This manual shift does not change the shape or size of the conics. Original parameters of the rectified conics were untouched and the shifting was applied only when displaying for the figure)

Back to our problem of trying to find the image of the absolute conic w. Image of the absolute conic contains all images of circular points for the same planar object. Then the following equations can be followed:

$$I' \epsilon\, w : I'^T w I' = 0$$

$$I' = H_{rect}^{-1} I = [h_1 \quad h_2 \quad h_3] \begin{bmatrix} 1 \\ i \\ 0 \end{bmatrix}$$

$$= h_1 + ih_2 \; where \; h_1, h_2, h_3 \; column \; vectors \; defining \; H_{rect}^{-1}$$

$$(h_1 + ih_2)^T w (h_1 + ih_2) = 0 \; which \; implies:$$

$$h_1^T w h_2 = 0 \tag{1}$$

$$h_1^T w h_1 + h_2^T w h_2 = 0 \tag{2}$$

This means that by calculating H_{rect}^{-1}, we can impose 2 linearly independent equations to w. This is a good start but we need more equations to be able to solve for w. For this we can use the following equation that defines the angle between two viewing rays (or 2 lines) $d_1 \; and \; d_2$ and the equivalent equation which includes vanishing points $V_1 \; and \; V_2$ of such viewing rays (Calculated from the intersection of parallel lines to d_1 and d_2 separately in the image) and w.

$$\cos(\theta) = \frac{d_1^T d_2}{\sqrt{(d_1^T d_1)(d_2^T d_2)}} = \frac{V_1^T w V_2}{\sqrt{(V_1^T w V_1)(V_2^T w V_2)}}$$

In case of two orthogonal lines this expression reduces to the following form:

$$\cos\left(\frac{\pi}{2}\right) = d_1^T d_2 = V_1^T w V_2 = 0$$

Now let’s use this equation to create 2 more linear equations for w. Remembering our original system and the useful lines we had defined in the “Introduction” section we have the following relations:

$$l_1' \perp l_3', \qquad l_1' \| a \,, V_1 = a' \times l_1', \qquad l_3' \| l_4' \,, V_2 = a' \times l_1', \qquad V_1^T w V_2 = 0 \tag{3}$$

$$l_2' \perp l_6', \qquad l_2' \| l_1' \,, V_1 = l_1' \times l_2', \qquad l_6' \| l_5' \,, V_3 = l_6' \times l_5', \qquad V_1^T w V_3 = 0 \tag{4}$$

We have reached our original goal of defining 4 linearly independent equations for w. w can now analytically be found by solving equations $(1), (2), (3), (4)$for w. (In the code implementation, since we are working with the homogenous coordinates, we only need to solve for 3DOF of the ratios of the w parameters)

Q3 MatLab Implementation (P1_2_3_main.m)

We must first define our new lines, points and vanishing points described above to be able to use them in calculations. All these lines are defined using the nomenclature established in the Introduction these features plotted on the original image can be seen in Figure 11 and Figure 12. Also see in these figures the graphical representations of h, a', V.

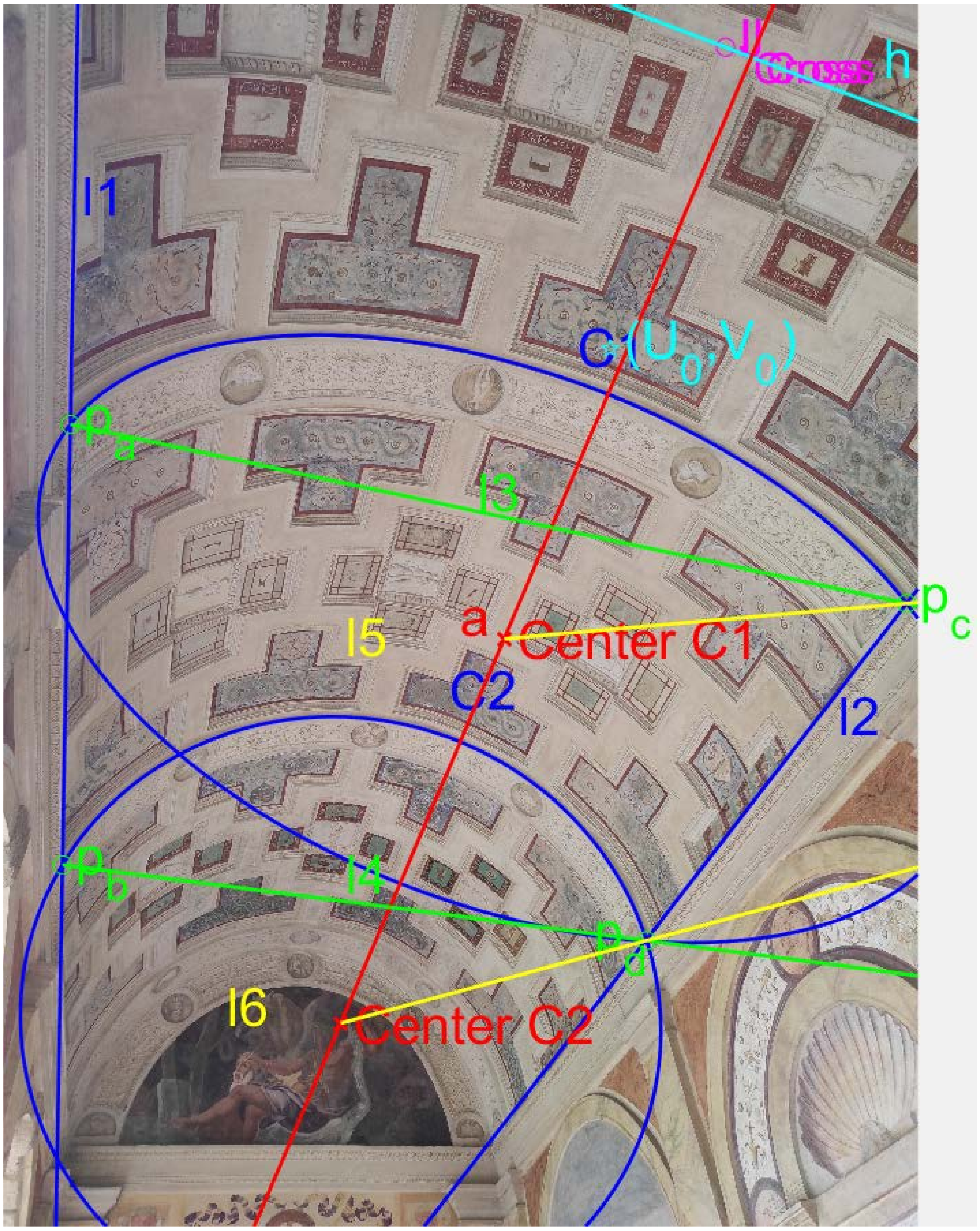


Figure 11: Original Image with the Necessary Features Marked

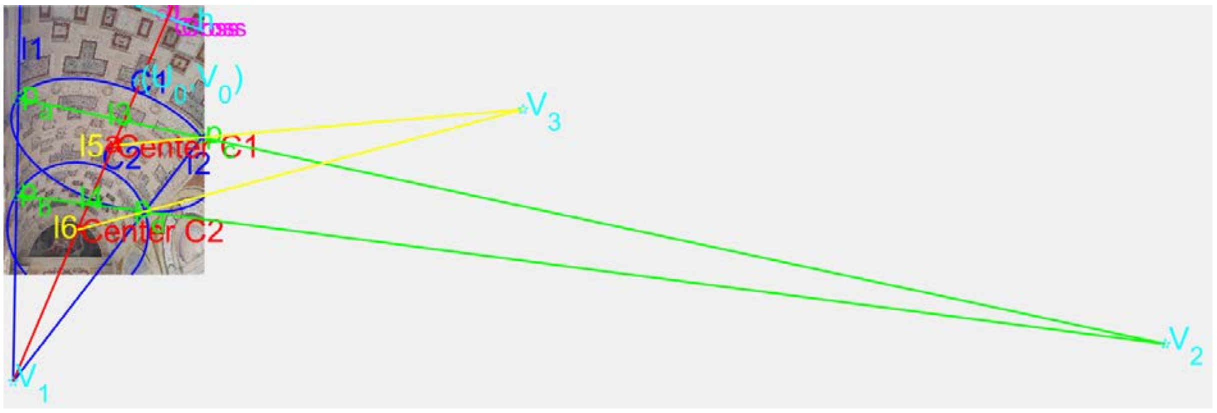
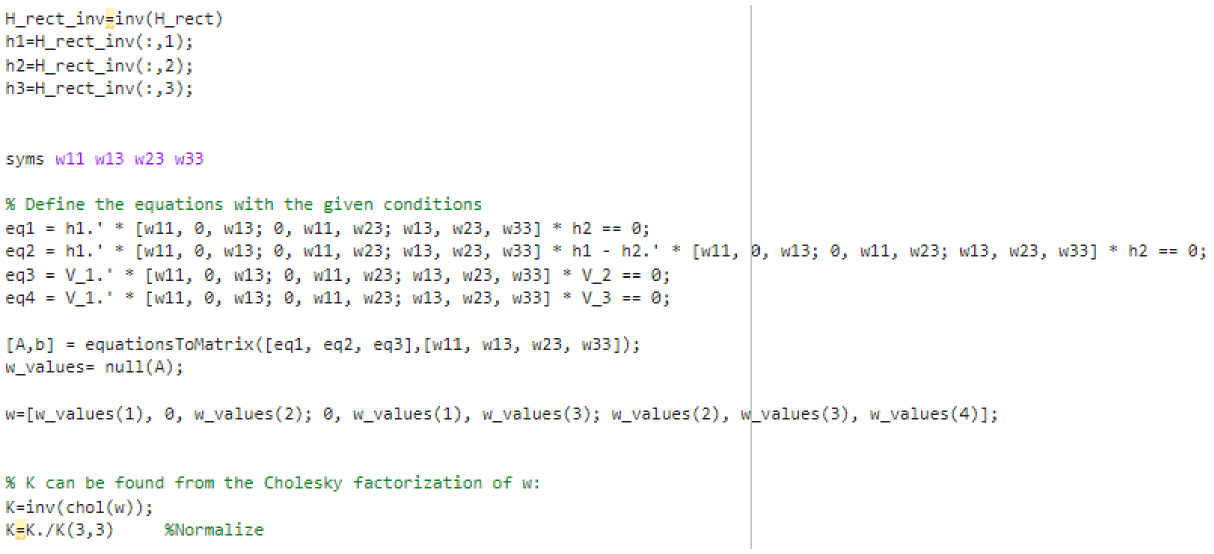
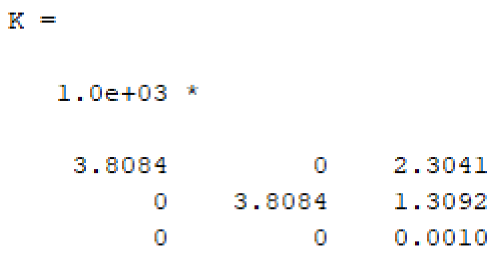


Figure 12: Original Image's Necessary Vanishing Points

As we will be solving for w in homogenous coordinates we need only 3 equations. As seen below, equations $(1), (2), (3)$ were symbolically defined and solved for the 4 w parameters. K was then simply found by Cholesky factorization and normalized with respect to f_z.



```
H_rect_inv=inv(H_rect)
h1=H_rect_inv(:,1);
h2=H_rect_inv(:,2);
h3=H_rect_inv(:,3);


syms w11 w13 w23 w33

% Define the equations with the given conditions
eq1 = h1.' * [w11, 0, w13; 0, w11, w23; w13, w23, w33] * h2 == 0;
eq2 = h1.' * [w11, 0, w13; 0, w11, w23; w13, w23, w33] * h1 - h2.' * [w11, 0, w13; 0, w11, w23; w13, w23, w33] * h2 == 0;
eq3 = V_1.' * [w11, 0, w13; 0, w11, w23; w13, w23, w33] * V_2 == 0;
eq4 = V_1.' * [w11, 0, w13; 0, w11, w23; w13, w23, w33] * V_3 == 0;

[A,b] = equationsToMatrix([eq1, eq2, eq3],[w11, w13, w23, w33]);
w_values= null(A);

w=[w_values(1), 0, w_values(2); 0, w_values(1), w_values(3); w_values(2), w_values(3), w_values(4)];


% K can be found from the Cholesky factorization of w:
K=inv(chol(w));
K=K./K(3,3)     %Normalize
```



```
K =

   1.0e+03 *

    3.8084         0    2.3041
         0    3.8084    1.3092
         0         0    0.0010
```

Preamble to 4. Pose Parameters of the Cylinder Axis

We are asked the orientation of the cylinder axis with respect to the camera reference. First let's start by defining a world reference. This is the easiest reference to understand. The world reference is an arbitrary reference in 3D space that we are given or we choose. The 3D coordinates of a point in this reference frame is given by x_{world}. The camera reference on the other hand is a coordinate frame in 3D space that puts the camera to its origin, directed with its Z axis. A 3D coordinate defined in the camera reference frame is given by x_{cam}. Coordinates in each of these frames are related by the following roto-translational relation.

$$\begin{bmatrix} x_{cam} \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{world} \\ 1 \end{bmatrix}$$

Since we have the freedom to choose our world reference we can simply choose it to be the same as the camera reference. Hence: $R = I, t = 0, x_{cam} = x_{world}$.

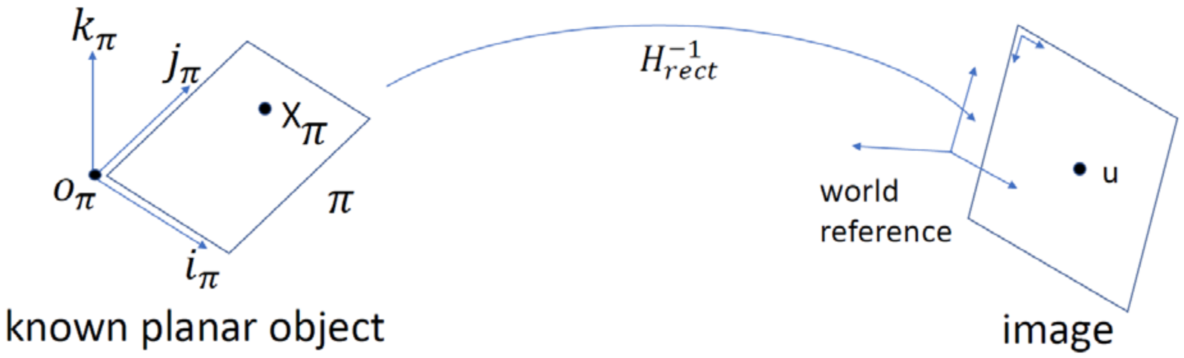


Figure 13: 3 Reference systems: π for a Known Planar Object, Arbitrary World Reference, Image Reference

Looking at Figure 13 to help us visualize, we can define 2 more coordinate systems that we are frequently working with. The first is the 2D image coordinate system. Positions are defined by pixels and the origin is on the top left of any image (+x to right, +y to down). Let's call the homogenous coordinates of a point in this reference frame u. The next coordinate system we should explore is the coordinate system used by a known planar object. Looking at Figure 9, we can see that the original scene and the similar reconstructed model both constitute as planar objects. Between these two we only know what the reconstructed planar scene looks like, let's call this plane of the planar object as π and the homogenous coordinates of a point on this plane as x_π. Having defined these, we know by our previous arguments and by theory that the following relations hold true:

$$u = H_{rect}^{-1} x_\pi$$

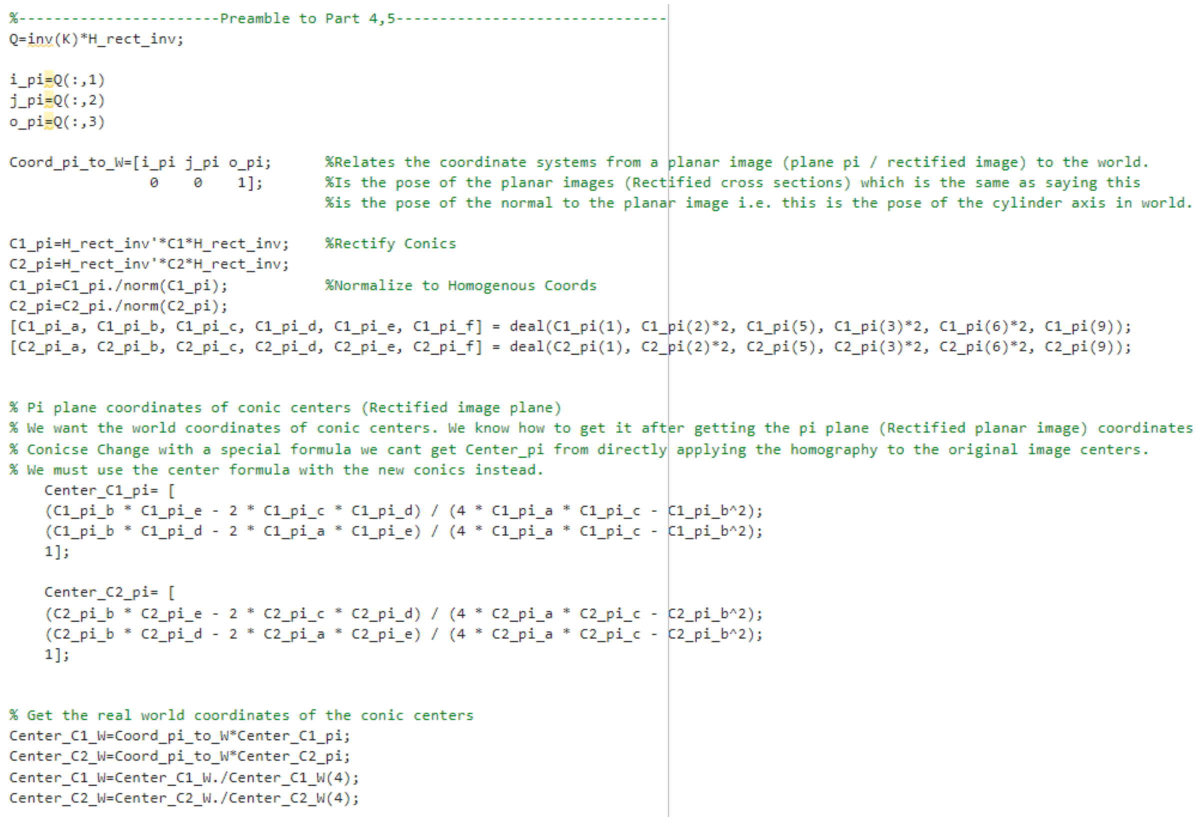
$$x_\pi = H_{rect} u$$

$$x_{world} = \begin{bmatrix} i_\pi & j_\pi & o_\pi \\ 0 & 0 & 1 \end{bmatrix} x_\pi = \begin{bmatrix} i_\pi & j_\pi & o_\pi \\ 0 & 0 & 1 \end{bmatrix} H_{rect} u$$

$$[i_\pi \quad j_\pi \quad o_\pi] = K^{-1} H_{rect}^{-1} = \boldsymbol{Pose\ Parameters\ of\ Plane\ \pi\ wrt. Camera/World\ Reference}$$

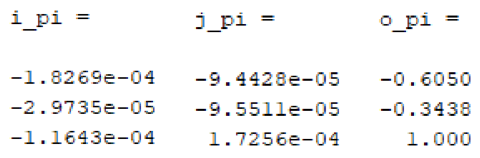
Q4 Preamble MatLab Implementation (Part1_2_3_main.m)

The described theoretical solution can be implemented in MatLab without any differences to get the parameters of the pose matrix of the rectified image plane and the conic center coordinates in 3D.



```
%----------------------Preamble to Part 4,5------------------------------
Q=inv(K)*H_rect_inv;

i_pi=Q(:,1)
j_pi=Q(:,2)
o_pi=Q(:,3)

Coord_pi_to_W=[i_pi j_pi o_pi;      %Relates the coordinate systems from a planar image (plane pi / rectified image) to the world.
               0    0    1];        %Is the pose of the planar images (Rectified cross sections) which is the same as saying this
                                    %is the pose of the normal to the planar image i.e. this is the pose of the cylinder axis in world.

C1_pi=H_rect_inv'*C1*H_rect_inv;    %Rectify Conics
C2_pi=H_rect_inv'*C2*H_rect_inv;
C1_pi=C1_pi./norm(C1_pi);           %Normalize to Homogenous Coords
C2_pi=C2_pi./norm(C2_pi);
[C1_pi_a, C1_pi_b, C1_pi_c, C1_pi_d, C1_pi_e, C1_pi_f] = deal(C1_pi(1), C1_pi(2)*2, C1_pi(5), C1_pi(3)*2, C1_pi(6)*2, C1_pi(9));
[C2_pi_a, C2_pi_b, C2_pi_c, C2_pi_d, C2_pi_e, C2_pi_f] = deal(C2_pi(1), C2_pi(2)*2, C2_pi(5), C2_pi(3)*2, C2_pi(6)*2, C2_pi(9));


% Pi plane coordinates of conic centers (Rectified image plane)
% We want the world coordinates of conic centers. We know how to get it after getting the pi plane (Rectified planar image) coordinates
% Conicse Change with a special formula we cant get Center_pi from directly applying the homography to the original image centers.
% We must use the center formula with the new conics instead.
    Center_C1_pi= [
    (C1_pi_b * C1_pi_e - 2 * C1_pi_c * C1_pi_d) / (4 * C1_pi_a * C1_pi_c - C1_pi_b^2);
    (C1_pi_b * C1_pi_d - 2 * C1_pi_a * C1_pi_e) / (4 * C1_pi_a * C1_pi_c - C1_pi_b^2);
    1];

    Center_C2_pi= [
    (C2_pi_b * C2_pi_e - 2 * C2_pi_c * C2_pi_d) / (4 * C2_pi_a * C2_pi_c - C2_pi_b^2);
    (C2_pi_b * C2_pi_d - 2 * C2_pi_a * C2_pi_e) / (4 * C2_pi_a * C2_pi_c - C2_pi_b^2);
    1];


% Get the real world coordinates of the conic centers
Center_C1_W=Coord_pi_to_W*Center_C1_pi;
Center_C2_W=Coord_pi_to_W*Center_C2_pi;
Center_C1_W=Center_C1_W./Center_C1_W(4);
Center_C2_W=Center_C2_W./Center_C2_W(4);
```

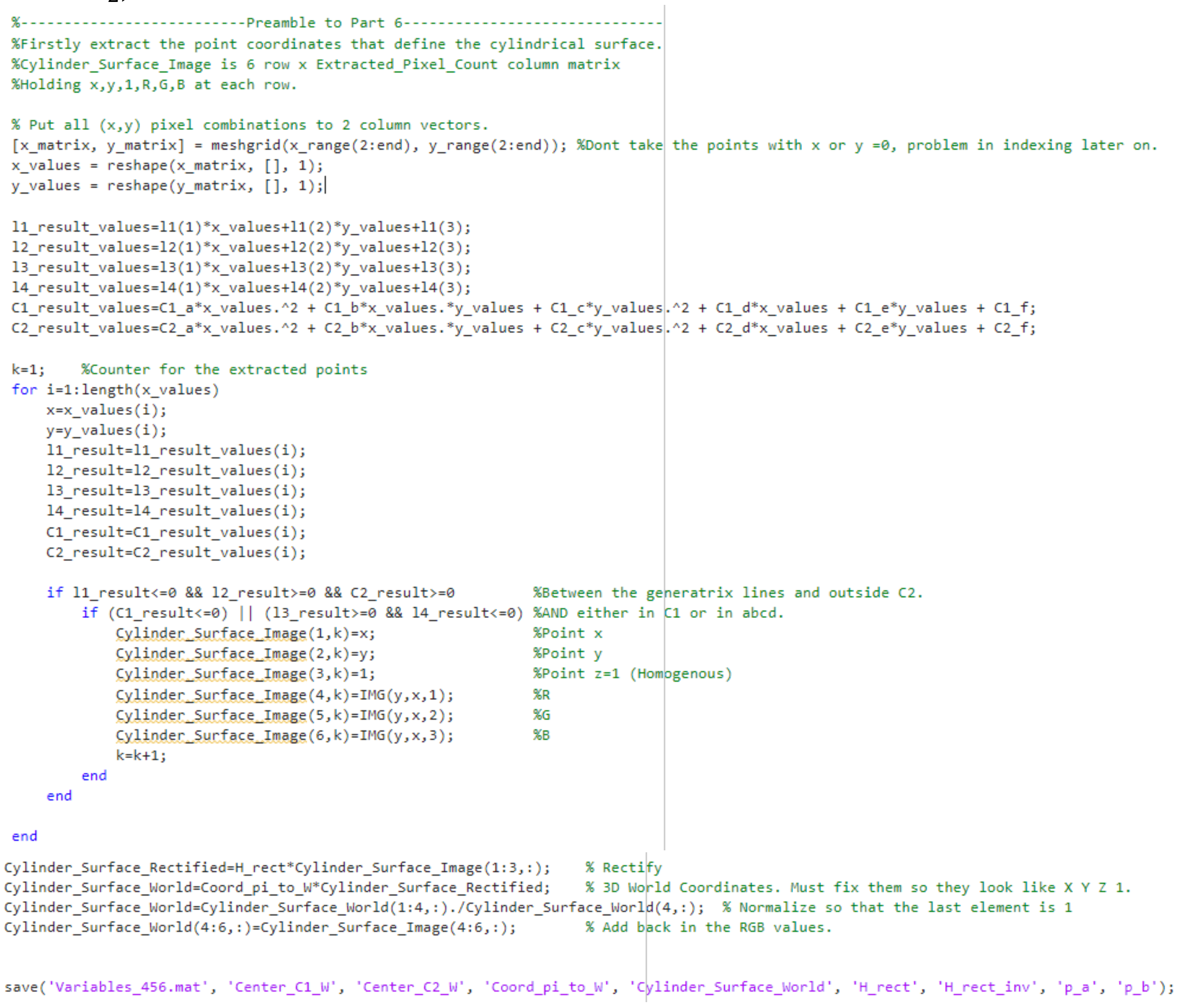
The resulting pose $[i_\pi \quad j_\pi \quad o_\pi]$ parameters can be seen as follows:



```
i_pi =          j_pi =          o_pi =

-1.8269e-04    -9.4428e-05     -0.6050
-2.9735e-05    -9.5511e-05     -0.3438
-1.1643e-04     1.7256e-04      1.000
```

At this point it is really beneficial to visualize exactly what this mapping to 3D coordinates result in. In order to help visualize this 3D mapping and also do some groundwork for Part 6, let us find the cylindrical surface left between the generatrix lines and the conics and save it. We can accomplish this by checking the inequalities given below inside the code.

- Points should be on the right of l_1'
- Points should be on the left of l_2'
- Points should be either: (Inside C_1' and Outside C_2') OR (Inside $p_a p_b p_c p_d$ and Outside C_2')



```
%-------------------------Preamble to Part 6-----------------------------
%Firstly extract the point coordinates that define the cylindrical surface.
%Cylinder_Surface_Image is 6 row x Extracted_Pixel_Count column matrix
%Holding x,y,1,R,G,B at each row.

% Put all (x,y) pixel combinations to 2 column vectors.
[x_matrix, y_matrix] = meshgrid(x_range(2:end), y_range(2:end)); %Dont take the points with x or y =0, problem in indexing later on.
x_values = reshape(x_matrix, [], 1);
y_values = reshape(y_matrix, [], 1);

l1_result_values=l1(1)*x_values+l1(2)*y_values+l1(3);
l2_result_values=l2(1)*x_values+l2(2)*y_values+l2(3);
l3_result_values=l3(1)*x_values+l3(2)*y_values+l3(3);
l4_result_values=l4(1)*x_values+l4(2)*y_values+l4(3);
C1_result_values=C1_a*x_values.^2 + C1_b*x_values.*y_values + C1_c*y_values.^2 + C1_d*x_values + C1_e*y_values + C1_f;
C2_result_values=C2_a*x_values.^2 + C2_b*x_values.*y_values + C2_c*y_values.^2 + C2_d*x_values + C2_e*y_values + C2_f;

k=1;    %Counter for the extracted points
for i=1:length(x_values)
    x=x_values(i);
    y=y_values(i);
    l1_result=l1_result_values(i);
    l2_result=l2_result_values(i);
    l3_result=l3_result_values(i);
    l4_result=l4_result_values(i);
    C1_result=C1_result_values(i);
    C2_result=C2_result_values(i);

    if l1_result<=0 && l2_result>=0 && C2_result>=0         %Between the generatrix lines and outside C2.
        if (C1_result<=0) || (l3_result>=0 && l4_result<=0) %AND either in C1 or in abcd.
            Cylinder_Surface_Image(1,k)=x;                  %Point x
            Cylinder_Surface_Image(2,k)=y;                  %Point y
            Cylinder_Surface_Image(3,k)=1;                  %Point z=1 (Homogenous)
            Cylinder_Surface_Image(4,k)=IMG(y,x,1);         %R
            Cylinder_Surface_Image(5,k)=IMG(y,x,2);         %G
            Cylinder_Surface_Image(6,k)=IMG(y,x,3);         %B
            k=k+1;
        end
    end

 end
Cylinder_Surface_Rectified=H_rect*Cylinder_Surface_Image(1:3,:);    % Rectify
Cylinder_Surface_World=Coord_pi_to_W*Cylinder_Surface_Rectified;    % 3D World Coordinates. Must fix them so they look like X Y Z 1.
Cylinder_Surface_World=Cylinder_Surface_World(1:4,:)./Cylinder_Surface_World(4,:);  % Normalize so that the last element is 1
Cylinder_Surface_World(4:6,:)=Cylinder_Surface_Image(4:6,:);        % Add back in the RGB values.


save('Variables_456.mat', 'Center_C1_W', 'Center_C2_W', 'Coord_pi_to_W', 'Cylinder_Surface_World', 'H_rect', 'H_rect_inv', 'p_a', 'p_b');
```

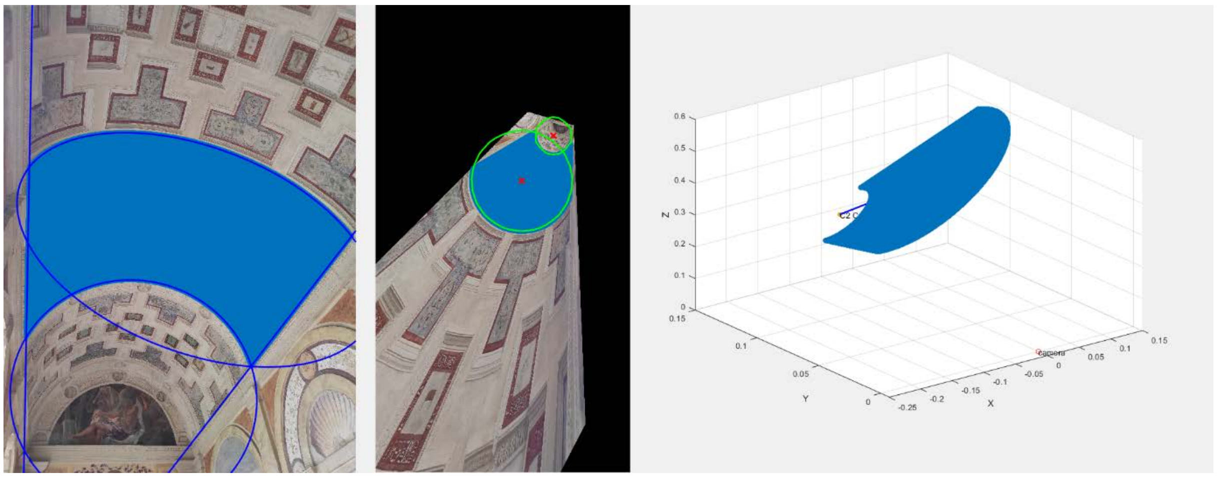


Figure 14: Target Cylindrical Surface in the Image, Rectified Image, 3D Mapping

The chosen cylindrical surface (Set of XYZ and RGB values inside Cylinder_Surface_Image, Cylinder_Surface_Rectified, Cylinder_Surface_World arrays) in the image, rectified image and with the applied 3D transform using the pose matrix are seen in Figure 14.

Now we can see that there is a problem. In the 3D coordinates there is no such a cylindrical surface, instead all the image points are on the same plane. This is because a camera can not differentiate between points on the same viewing ray. What is happening is that the camera can't differentiate between a 3D cylindrical object or the planar object that has the same viewing rays defining its points. In order to find the actually cylindrical surface in 3D we will need to follow a series of logical facts in the following parts.

For now, the code P1_2_3_main.m file is long enough, therefore the 3D coordinates with respective colors Cylinder_Surface_World, as well as the 3D coordinates of the conic centers and the transformation (Rectification and 3D mapping) matrices are exported to a file called "Variables_456.mat) Using these variables we will continue our process in the code "P4_5_main.m"

Cylinder_Surface_World holds a pixel at each column. Each column has 6 rows corresponding to the mapped X,Y,Z coordinates of a sampled pixel in the original image and the R,G,B color values of that pixel.

Q4 Preamble MatLab Implementation (P4_5_demo.m)

Let’s build a common base of logic. There is a 3D cylinder surface in 3D space. We have just 1 projection of this surface in Cylinder_Surface_World. Any of the points in this projection can be anywhere else on their respective viewing rays (vector drawn from the origin to that point) in 3D space.

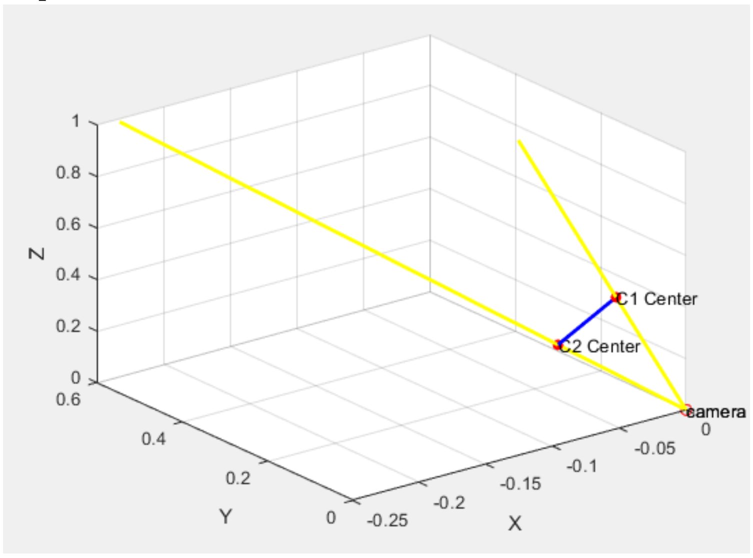


Figure 15: Conic Centers' Viewing Rays

Let’s take the conic center points. Their True 3D coordinates can be anywhere along their viewing rays shown in yellow in Figure 15. So how do we know where they must actually be in reality. To solve this let’s first understand where point A and point B should be.

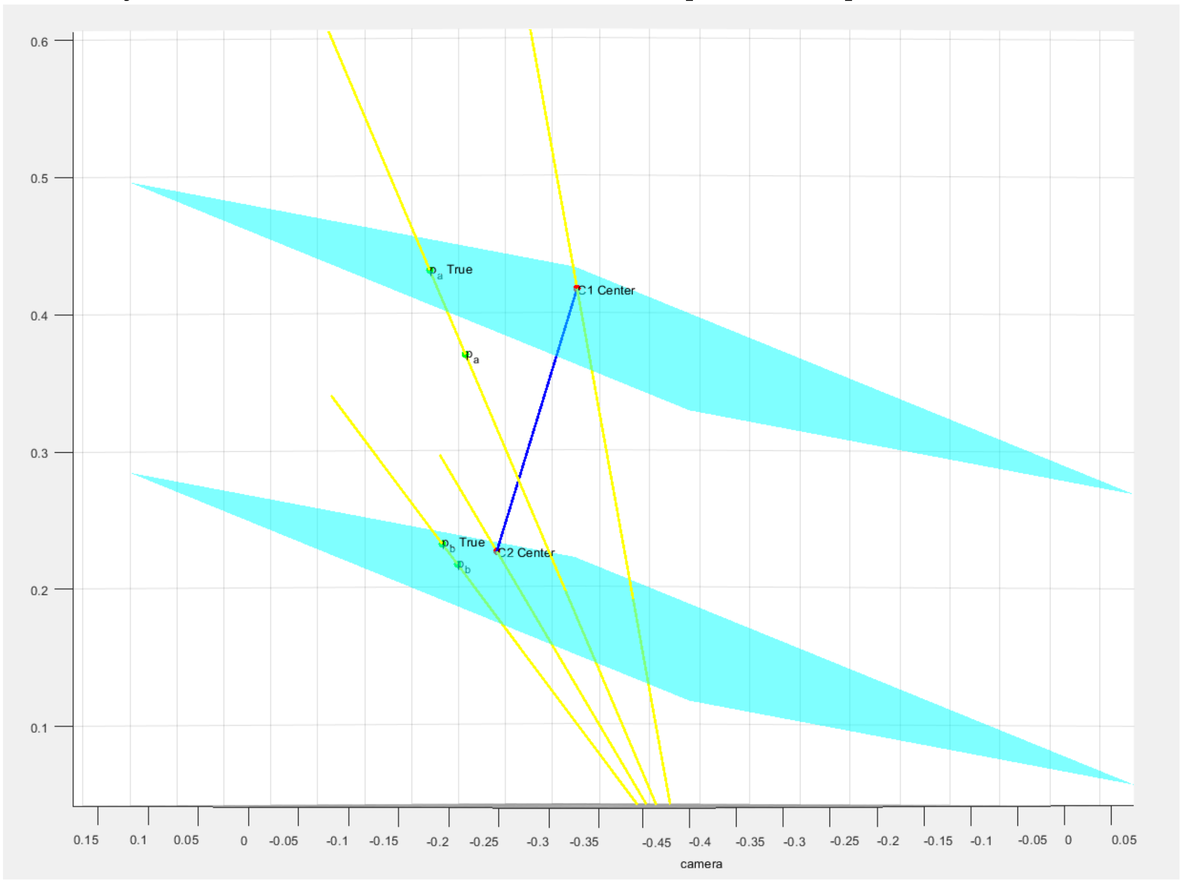


Figure 16: Points A and B 3D Coordinates and the True Coordinates Intersecting the Cross Sectional Planes

We know that Points A and B are on the cross sections of the cylinder. (At the first and the last cross section of our finite cylinder) And we know that any cross sectional plane of the cylinder is perpendicular to its axis. Then: Point A should be on a plane perpendicular to the cylinder axis passing through $Center_{C_1}$. And similarly Point B should be on a plane perpendicular to the cylinder axis passing through $Center_{C_2}$. Then drawing the viewing rays of Points A and B, their true positions are where they intersect the respective cross sectional planes (planes orthogonal to the cylinder axis passing through their respective conic centers) as seen in Figure 16.

Assuming the orientation of the cylinder axis in 3D space is correct we should have the actually true coordinates for the points A and B. It is easy to check if this is the case. We know that the distance between $Center_{C_1}$ *and* $p_{a_{true}}$ is the radius of the circumference defining the cap of the cylinder. And in theory the distance between $Center_{C_2}$ *and* $p_{b_{true}}$ defines a same size circumference. Let's plot the circumferences on the planes orthogonal to the cylinder axis with radii defined by the distance between the circumference centers and the point A or B in the same plane:

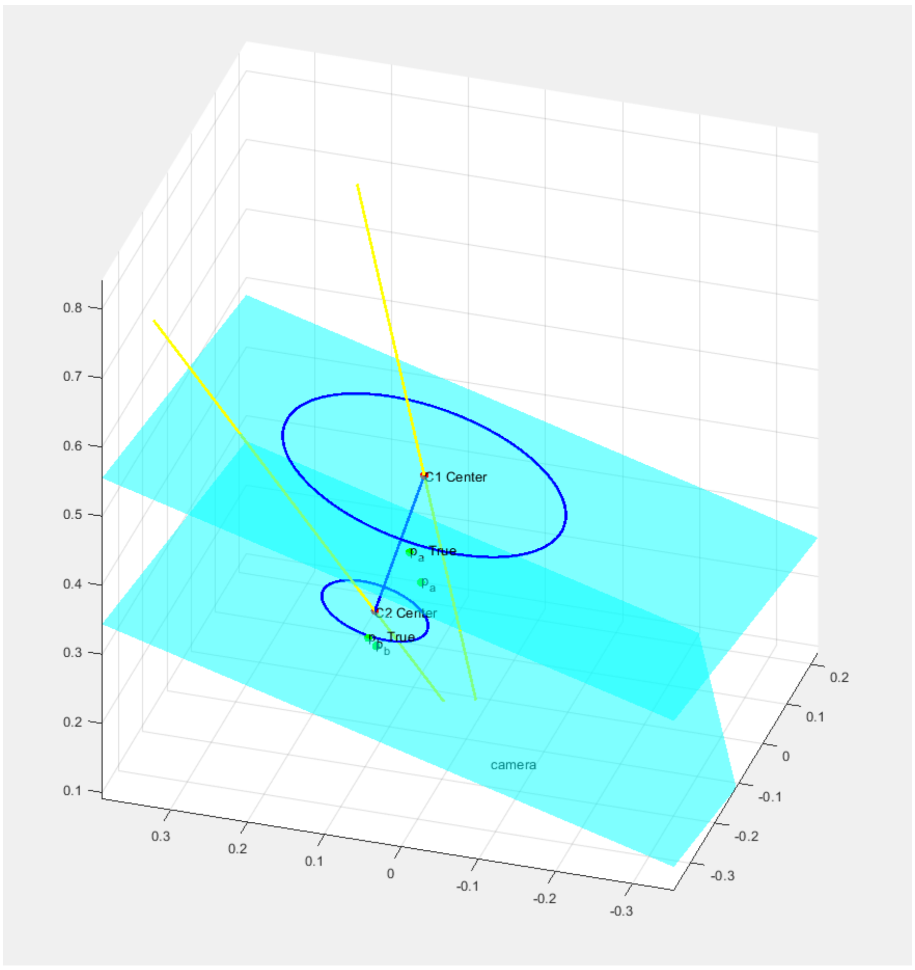


Figure 17: Cylinder (?) Defined at This Axis Orientation

We can see in Figure 17 that the correct logic followed for this cylinder orientation results in a cone not a cylinder. This means that the True positions of $Center_{C_1}$and $Center_{C_2}$ are not given by our initial mapping to 3D. The true positions of these centers are rather on somewhere else on their respective viewing rays.

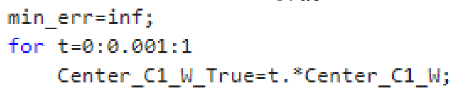
4, 5 MatLab Implementation a_{true}, $Radius/Distance$ (P4_5_main.m)

Now we know that we need to choose new 3D positions for $Center_{C_1}$ and $Center_{C_2}$ along their respective viewing rays. However, if we choose 2 new points with the axis oriented the same way as the given example case in the previous section, we would still end up with a cone not a cylinder. Therefore, we must actually choose the orientation of the cylinder axis between the two viewing rays of the centers. Hence this is a 1DOF problem. Let's approach solving this problem by keeping $Center_{C_2}$ where it is and let's only choose a new $Center_{C_1}$ on its viewing ray. We can express the new $Center_{C_1}$ on the viewing ray as:

$$Center_{C_{1True}} = t * Center_{C_1}$$

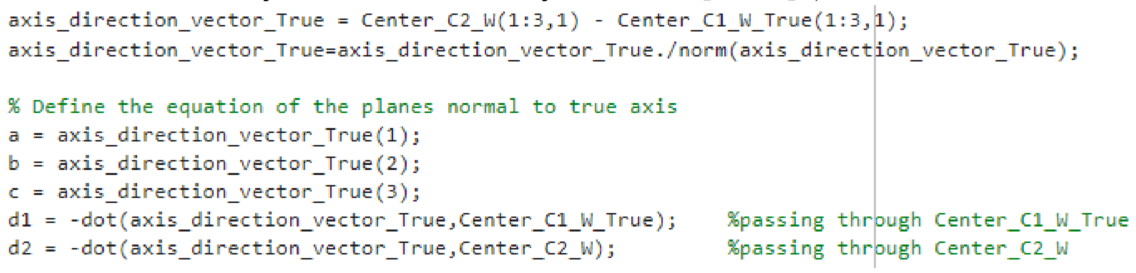
This is simple to understand as any vector on the viewing ray is just the $Center_{C_1}$ vector scaled by a scalar t. Using the following code we can find the scaling factor t.

Initializing a min_err to infinity, loop through scaling factors between 0 and 1 with step 0.001. Find $Center_{C_{1True}}$ for this t:



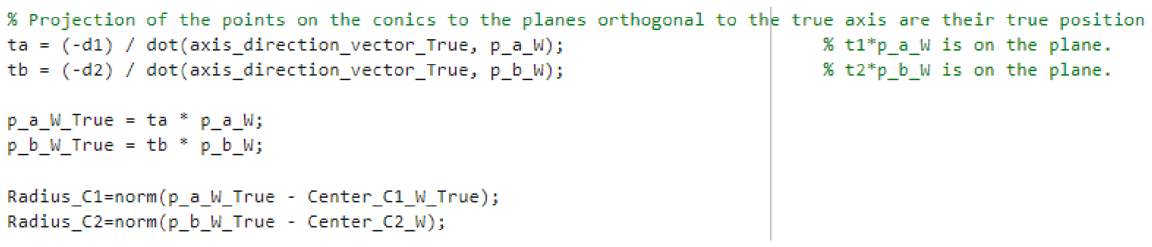
```
min_err=inf;
for t=0:0.001:1
    Center_C1_W_True=t.*Center_C1_W;
```

Using the new $Center_{C_{1True}}$ find the vector defining the axis. Following the reasoning in the previous section we can find the equations of the 2 planes normal to the axis and passing through centers 1 and 2 defined by $ax + by + cz + d_1 = 0$ and $ax + by + cz + d_2 = 0$. (Parameters a, b, c are the same for both planes since they have the same normal vector: Both planes are normal to the cylinder axis defined by the vector $[a, b, c]^T$)



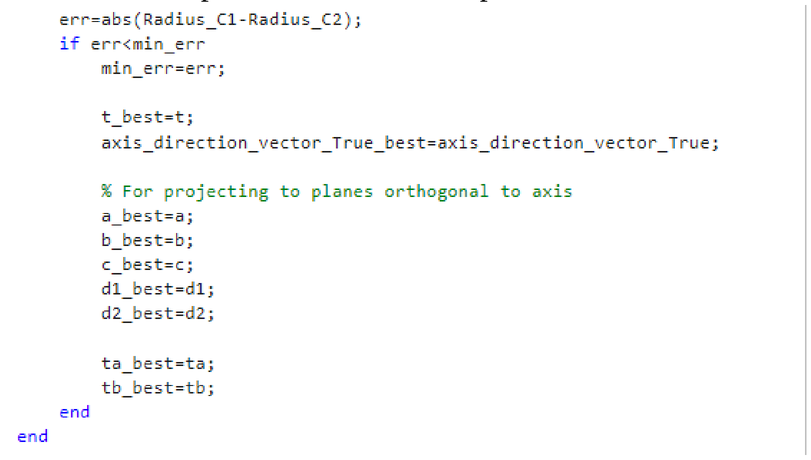
```
axis_direction_vector_True = Center_C2_W(1:3,1) - Center_C1_W_True(1:3,1);
axis_direction_vector_True=axis_direction_vector_True./norm(axis_direction_vector_True);

% Define the equation of the planes normal to true axis
a = axis_direction_vector_True(1);
b = axis_direction_vector_True(2);
c = axis_direction_vector_True(3);
d1 = -dot(axis_direction_vector_True,Center_C1_W_True);    %passing through Center_C1_W_True
d2 = -dot(axis_direction_vector_True,Center_C2_W);         %passing through Center_C2_W
```

Now that we know the planes where p_a and p_b should be we can find scaling factors t_a and t_b to move these points on their viewing rays until they are on their respective cross sectional planes. Knowing the true positions of p_a and p_b we can find the true radius of each conic in 3D.



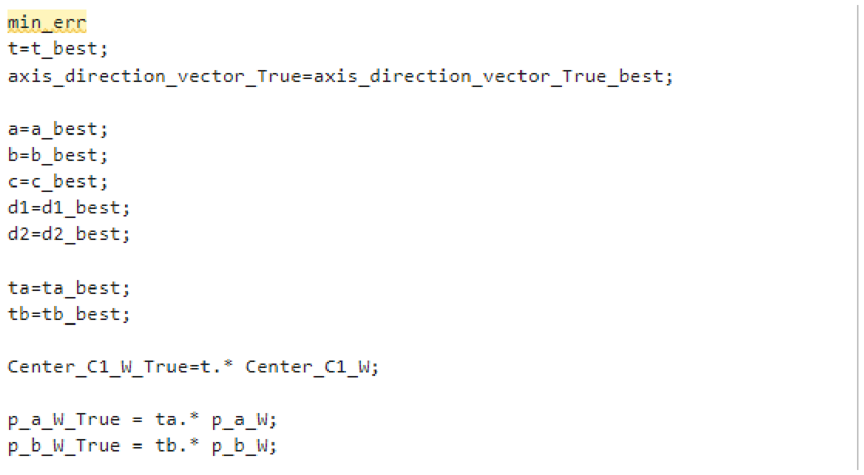
```
% Projection of the points on the conics to the planes orthogonal to the true axis are their true position
ta = (-d1) / dot(axis_direction_vector_True, p_a_W);                % t1*p_a_W is on the plane.
tb = (-d2) / dot(axis_direction_vector_True, p_b_W);                % t2*p_b_W is on the plane.

p_a_W_True = ta * p_a_W;
p_b_W_True = tb * p_b_W;

Radius_C1=norm(p_a_W_True - Center_C1_W_True);
Radius_C2=norm(p_b_W_True - Center_C2_W);
```

In this loop when the scaling factor t for $Center_{C_1}$ is correct $Radius_{C_1}$ and $Radius_{C_2}$ should be the same. (As it is in a cylinder) Therefore while we are still in this loop trying out different t values let's calculate the difference between the 2 radii and call that as an error. During our loops if for a t scaling value the error is smaller than our previous minimum error we can save the parameters in that loop.



```
        err=abs(Radius_C1-Radius_C2);
        if err<min_err
            min_err=err;

            t_best=t;
            axis_direction_vector_True_best=axis_direction_vector_True;

            % For projecting to planes orthogonal to axis
            a_best=a;
            b_best=b;
            c_best=c;
            d1_best=d1;
            d2_best=d2;

            ta_best=ta;
            tb_best=tb;
        end
    end
```

This means that when we have exited our trial and error loop for t values. The variable "t_best" will hold the t value, upon when $Center_{C_{1True}} = t * Center_{C_1}$ is calculated the resulting 3D surface as described in the previous section will indeed be a cylinder. (As we expect the min_error to exit the loop close to 0 meaning both of radii defined at each cross sectional plane is almost the same)



```
min_err
t=t_best;
axis_direction_vector_True=axis_direction_vector_True_best;

a=a_best;
b=b_best;
c=c_best;
d1=d1_best;
d2=d2_best;

ta=ta_best;
tb=tb_best;

Center_C1_W_True=t.* Center_C1_W;

p_a_W_True = ta.* p_a_W;
p_b_W_True = tb.* p_b_W;
```

The loop exits with min_err= 5.9473e-05 which means that the resulting cylinder has practically equal radii at the 2 end caps. Which confirms that we have found the correct $Center_{C_{1True}}$ such that the resulting 3D surface is a cylinder. Let's also confirm this visually by plotting our points and circumferences generated from this repositioning of $Center_{C_1}$:

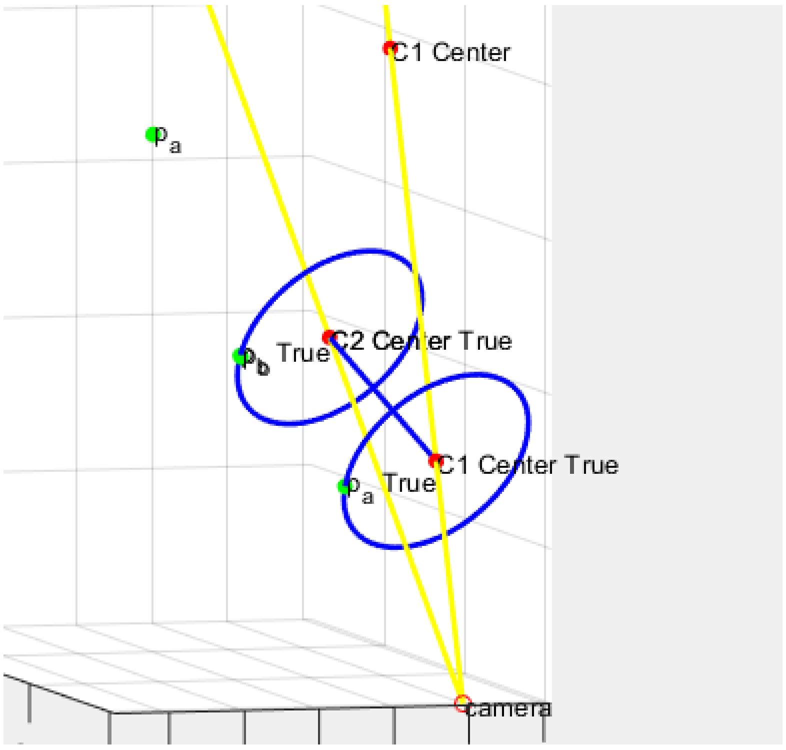


Figure 18: Cylinder Defined by the Repositioned 1st Conic Center

Figure 18 indeed shows that we have found a cylinder in 3D space on which our image points should lie. Before proceeding to visualizing the image on a 3D surface and unwrapping the cylinder we can calculate the orientation and the R_{real}/D_{real} Ratio of the cylinder.

```
%------------------------------Part 4------------------------------------
axis_direction_vector_True

%------------------------------Part 5------------------------------------
Radius=norm(p_a_W_True - Center_C1_W_True);
Distance=norm(Center_C2_W-Center_C1_W_True);
Radius_over_Distance=Radius/Distance
```

Answer to Q4: The cylinder axis is oriented with the 3D vector:

```
axis_direction_vector_True =

   -0.3214
    0.7817
    0.5344
```

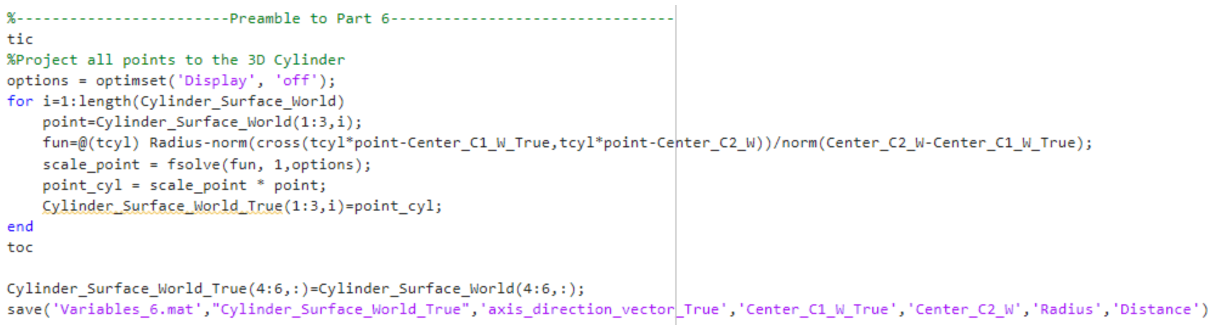
Answer to Q5: $Radius/Distance$ of the Cylinder is: (Note that this ratio won't change if both of the conic centers are moved along their viewing rays without changing the axis orientation. Since the resulting triangles defined by the viewing rays and the axis or the radii will be similar triangles.)

```
Radius_over_Distance = 0.5174
```

Now that we know where our cylinder is, we can map each of our pixel points in 3D to where it intersects the cylindrical surface along its viewing ray. This problem was formulated as: Take a pixel point p in 3D, find a scaling factor t_{cyl} such that the true position that pixel $p_{true} = t_{cyl} * p$ is on the cylinder surface. Being on the cylinder surface was formulated as: $distance\ from\ p_{true}\ to\ cylinder\ axis = Radius$. The distance from a point in 3D to a line segment between $Center_{C1_{true}}\ and\ Center_{C2}$ is given by:

$$distance\ from\ point\ to\ cylinder\ axis = \frac{\left|\left(Center_{C2} - Center_{C1_{true}}\right) \times \left(Center_{C1_{true}} - p\right)\right|}{\left|Center_{C2} - Center_{C1_{true}}\right|}$$

If $p = p_{true}$, this distance should be equal to $Radius$. So t_{cyl} is the root of the equation: $Radius - DistanceToCylinderAxis\left(p_{true}(t)\right) = 0$. This equation can be solved using the following code snippet for every point we had sampled in the desired region of the image:



```
%-----------------------Preamble to Part 6--------------------------------
tic
%Project all points to the 3D Cylinder
options = optimset('Display', 'off');
for i=1:length(Cylinder_Surface_World)
    point=Cylinder_Surface_World(1:3,i);
    fun=@(tcyl) Radius-norm(cross(tcyl*point-Center_C1_W_True,tcyl*point-Center_C2_W))/norm(Center_C2_W-Center_C1_W_True);
    scale_point = fsolve(fun, 1,options);
    point_cyl = scale_point * point;
    Cylinder_Surface_World_True(1:3,i)=point_cyl;
end
toc

Cylinder_Surface_World_True(4:6,:)=Cylinder_Surface_World(4:6,:);
save('Variables_6.mat',"Cylinder_Surface_World_True",'axis_direction_vector_True','Center_C1_W_True','Center_C2_W','Radius','Distance')
```

One important aspect in this snippet should be noted. We know that there will most probably be not one but two points where the viewing ray of a point intersects with the cylindrical surface. One point going into the surface and one point going out of it. In our case we know that we want the intersection point near the top of the cylinder. (Higher Z value corresponding to higher scaling factor t_{cyl}) To achieve this, we can simply assign the initial guess for t_{cyl} in the numerical solver "fsolve" as 1. Such that it will (most probably) find the higher t_{cyl} solution corresponding to the top intersection.

Some disclaimers:

- t_{cyl} doesn't necessarily have to be between 0 and 1. But since we tried and found a valid t for the scaling of $Center_{C1}$ between 0 and 1 we can know that t_{cyl} will also be in this range.
- When finding the scaling factor t for $Center_{C1}$ the range 0 to 1 was tried out because it could be seen by visual inspection on the 3D plot that if we pull $Center_{C1}$ closer to the origin on its viewing ray we could get a cylindrical surface. In a fully parametrized code the grid search range for variable t might have to be assigned differently.

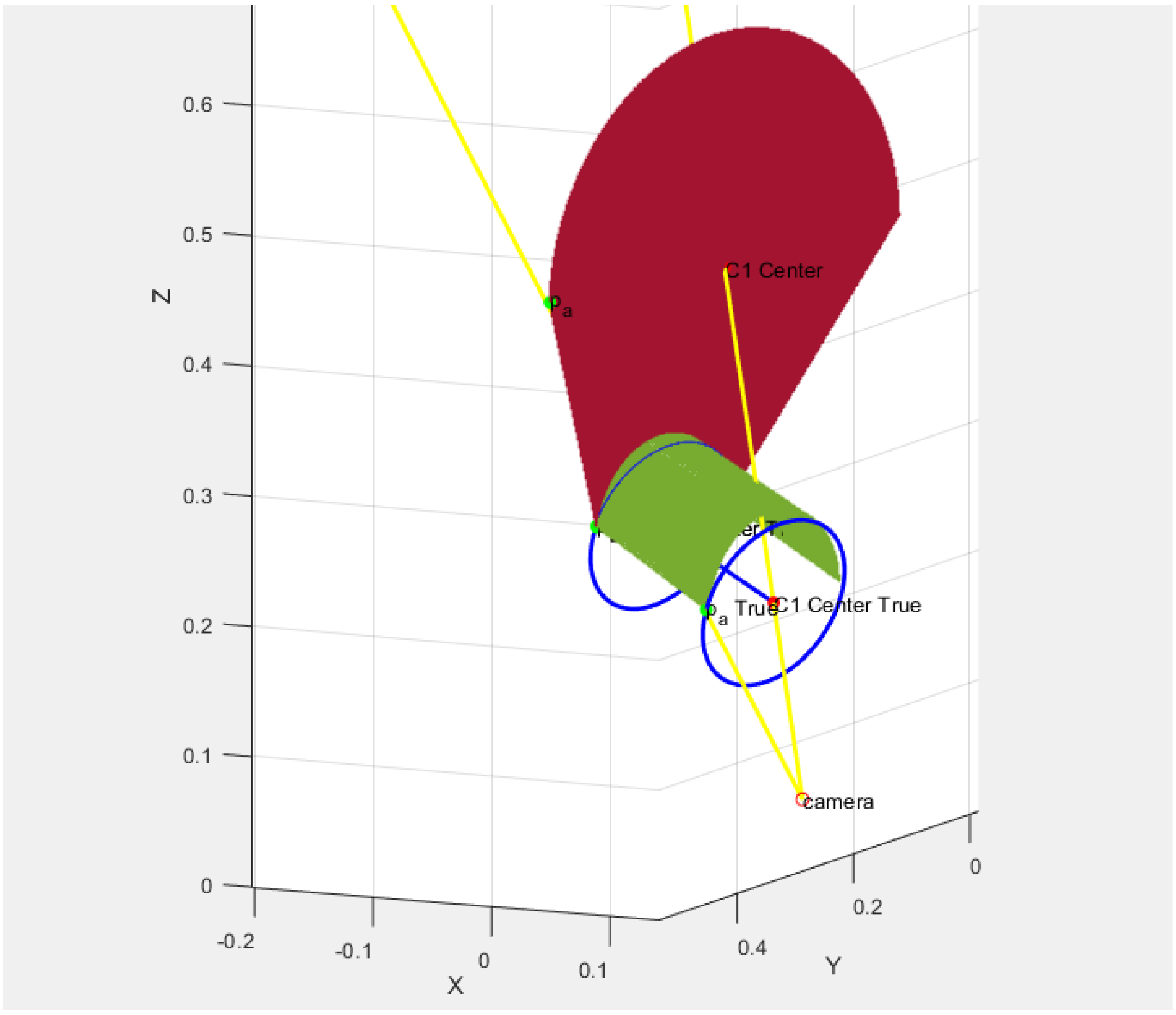


Figure 19: Points After Initial Mapping (Red), Points After Mapping on to the Cylindrical Surface (Green)

As a result of the previous code snippet we now have all our pixels' mapping on to a 3D cylindrical surface. Figure 19 shows where the initial 3D mapping showed our points to be in red. And in green, we have these points moved along their viewing rays such that they are on the surface of a cylinder. (Note that for a fixed $Center_{C2}$ there is only 1 such cylinder in 3D space.)

Note that the last given code snippet runs in about 30 minutes with a computer using Intel Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz 2.81 GHz processor and 16GB of RAM. This computation is heavy due to the numerical solution search done for about 4.5 million points. In order to not run this code every time, we export the necessary variables into a file called "Variables_6.mat" which we will process in the code: "P6_main.m"

6. Unfolding of the Cylinder Surface (P6_main.m)

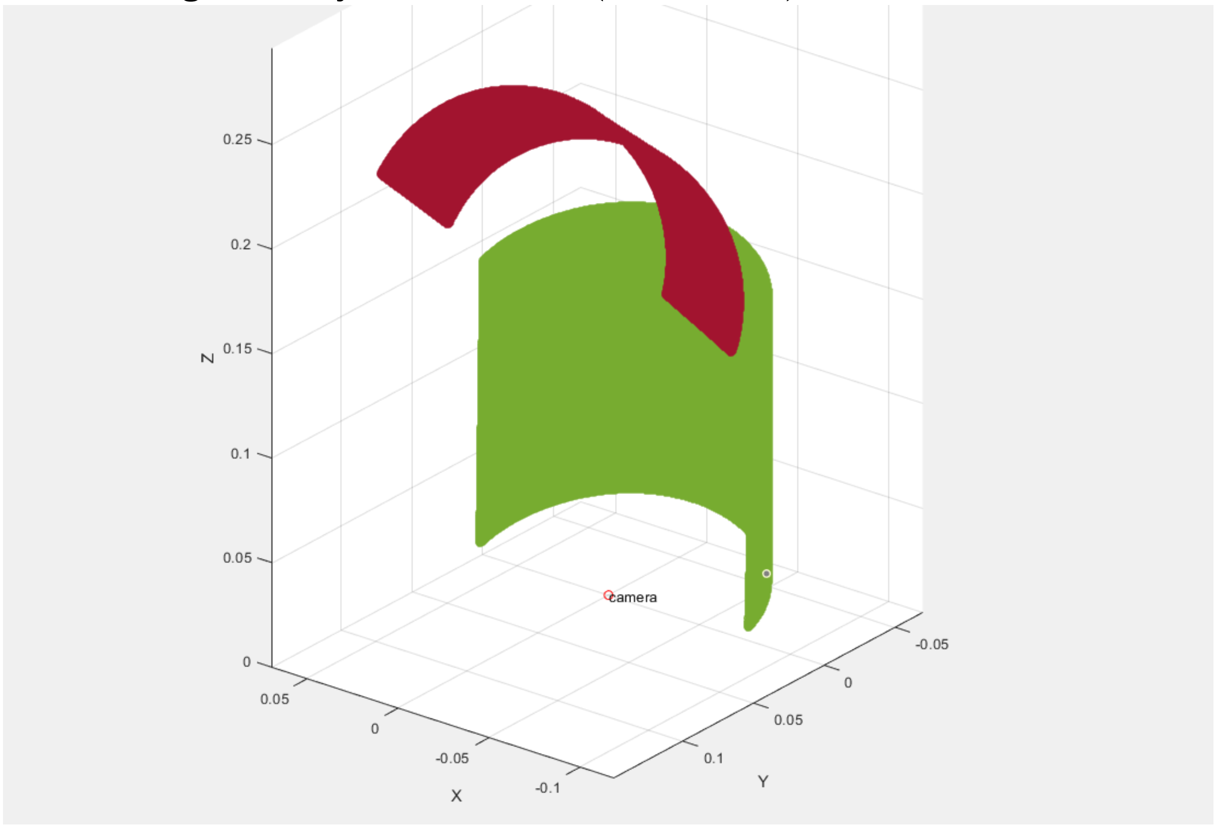
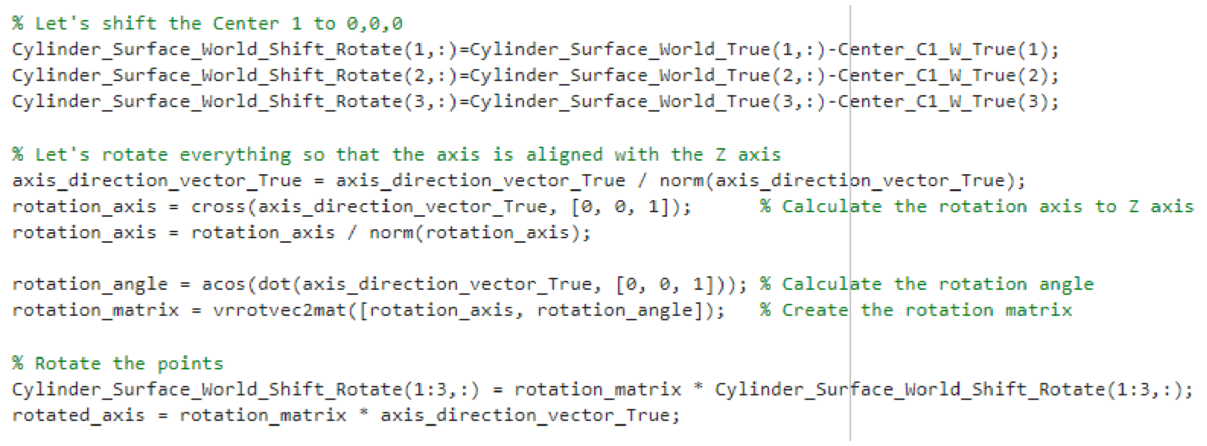


Figure 20: Original Cylindrical Surface in 3D (Red) and Rotated Cylindrical Surface in 3D (Green)

To unfold the cylinder our strategy will be to represent all points in cylindrical coordinates first. To make this step easier, all the points can be shifted and rotated such that the cylinder axis is parallel to the Z axis and $Center_{C1}$ is at the origin as seen in Figure 20. (Our cylinder is now a cylinder centered at the origin directed with the Z axis) This can be done using the following code applying simple vector geometry for shifting and rotation:



```
% Let's shift the Center 1 to 0,0,0
Cylinder_Surface_World_Shift_Rotate(1,:)=Cylinder_Surface_World_True(1,:)-Center_C1_W_True(1);
Cylinder_Surface_World_Shift_Rotate(2,:)=Cylinder_Surface_World_True(2,:)-Center_C1_W_True(2);
Cylinder_Surface_World_Shift_Rotate(3,:)=Cylinder_Surface_World_True(3,:)-Center_C1_W_True(3);

% Let's rotate everything so that the axis is aligned with the Z axis
axis_direction_vector_True = axis_direction_vector_True / norm(axis_direction_vector_True);
rotation_axis = cross(axis_direction_vector_True, [0, 0, 1]);      % Calculate the rotation axis to Z axis
rotation_axis = rotation_axis / norm(rotation_axis);

rotation_angle = acos(dot(axis_direction_vector_True, [0, 0, 1])); % Calculate the rotation angle
rotation_matrix = vrrotvec2mat([rotation_axis, rotation_angle]);   % Create the rotation matrix

% Rotate the points
Cylinder_Surface_World_Shift_Rotate(1:3,:) = rotation_matrix * Cylinder_Surface_World_Shift_Rotate(1:3,:);
rotated_axis = rotation_matrix * axis_direction_vector_True;
```

Now we can represent all the coordinates in cylindrical coordinates (r, h, θ) such that:

$$x = rcos(\theta)$$

$$y = rsin(\theta)$$

$$z = h$$

Then since we don’t really care about the radius parameter r to map this surface to a planar image, we just need to find the (h, θ) coordinates for each point:

$$h = z$$

$$\theta = \arctan\left(\frac{y}{x}\right)$$

```
%--------------------------------UNWRAP-------------------------------

% Unwrap by getting everything in cylindrical coordinates but plotting on
% to a plane.

Cylinder_Surface_h_tetha(1,:)=Cylinder_Surface_World_Shift_Rotate(3,:);
Cylinder_Surface_h_tetha(2,:)=atand(Cylinder_Surface_World_Shift_Rotate(2,:)./Cylinder_Surface_World_Shift_Rotate(1,:));


h_min=min(Cylinder_Surface_h_tetha(1,:))
h_max=max(Cylinder_Surface_h_tetha(1,:))
tetha_min=min(Cylinder_Surface_h_tetha(2,:))
tetha_max=max(Cylinder_Surface_h_tetha(2,:))
```

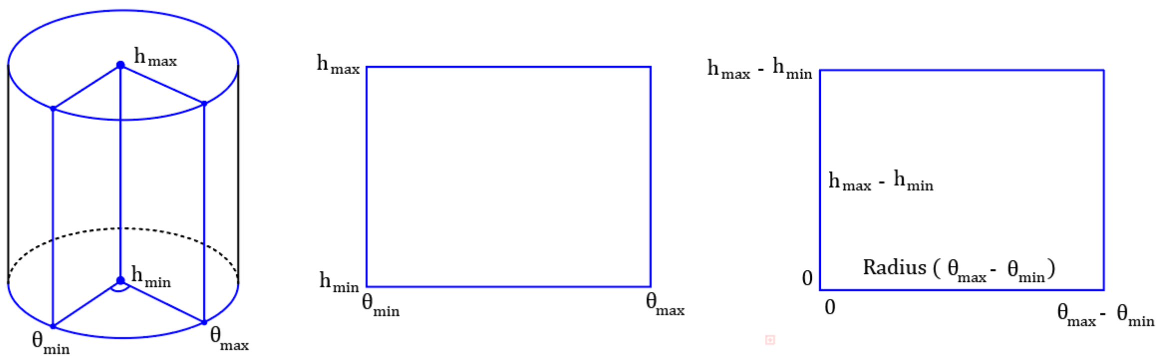


Figure 21: Cylinder Unwrapping Mapping Procedure

Now that we know each point’s cylindrical coordinates we can unwrap this cylindrical surface. Figure 21 shows how the cylindrical coordinates can simply be thought of as the linear coordinates to unwrap a cylinder (They can even be shifted by the minimum values so the smallest coordinate corresponds to (0,0)). Furthermore, we can actually calculate the aspect ratio of our unwrapped image since the height and the unwrapped length of the arc defining our surface are related. And knowing our aspect ratio we can find the width and the height of our unwrapped image since we know how many pixels the image should be. (Image should be as many pixels as we have points that are sampled in this surface) These calculations can be made as follows:

```
img_aspectratio=deg2rad(abs(tetha_max-tetha_min))*Radius/Distance;
unwrap_size_y=round(sqrt(length(Cylinder_Surface_h_tetha)/img_aspectratio));
unwrap_size_x=round(img_aspectratio*unwrap_size_y);

% Map the h and tetha values to 0->hmax-hmin, 0->tethamax-tethamin
Cylinder_Surface_h_tetha(1,:)=Cylinder_Surface_h_tetha(1,:)-h_min;
Cylinder_Surface_h_tetha(2,:)=Cylinder_Surface_h_tetha(2,:)-tetha_min;
```

We can further map the coordinates (h, θ) such that they range from $\left(0 - unwrap_{size_y}, 0 - unwrap_{size_x}\right)$ by normalizing each coordinate with $\left(\frac{unwrap_{size_y}}{h_{max}-h_{min}}, \frac{unwrap_{size_x}}{\theta_{max}-\theta_{min}}\right)$. After normalizing we can round the pixel coordinates and add 1 to it. This means that as a result all our points' coordinates will be integers ranging from $\left(1 \rightarrow unwrap_{size_y} + 1, 1 \rightarrow unwrap_{size_x} + 1\right)$. We add the 1 so that we can use these coordinate values in array indexing directly in the next step.

```
% Map the h and tetha values to 1->size_y+1, 1->size_x+1 (everything
% starting from 1,1 is easier for indexing later on.
Cylinder_Surface_h_tetha(1,:)=round(Cylinder_Surface_h_tetha(1,:).*unwrap_size_y./(h_max-h_min))+1;
Cylinder_Surface_h_tetha(2,:)=round(Cylinder_Surface_h_tetha(2,:).*unwrap_size_x./(tetha_max-tetha_min))+1;
```

Right now, we have successfully mapped all our target pixels in the desired pixel range such that they also have integer coordinates. Now we can easily construct our image to be printed by assigning the RGB value of each pixel to the image array's indexes that correspond to our (θ, h) coordinates. (Think of them as (x, y) coordinates) (Please note that the rotation and circular shift applied to the image before printing is only so we can see the image as one piece and the right way up)

```
IMG_unwrapped=zeros(unwrap_size_y+1, unwrap_size_x+1, 3, 'uint8');
for i=1:length(Cylinder_Surface_h_tetha)
    IMG_unwrapped(Cylinder_Surface_h_tetha(1,i),Cylinder_Surface_h_tetha(2,i),1)=Cylinder_Surface_World_True(4,i);  %R
    IMG_unwrapped(Cylinder_Surface_h_tetha(1,i),Cylinder_Surface_h_tetha(2,i),2)=Cylinder_Surface_World_True(5,i);  %G
    IMG_unwrapped(Cylinder_Surface_h_tetha(1,i),Cylinder_Surface_h_tetha(2,i),3)=Cylinder_Surface_World_True(6,i);  %B
end

figure(2)
imshow(imrotate(circshift(IMG_unwrapped, [0, -unwrap_size_x/2-100]), 180, 'bilinear', 'crop'))
% Manual beautification so that the image isn't cut in half and it's right way up
```

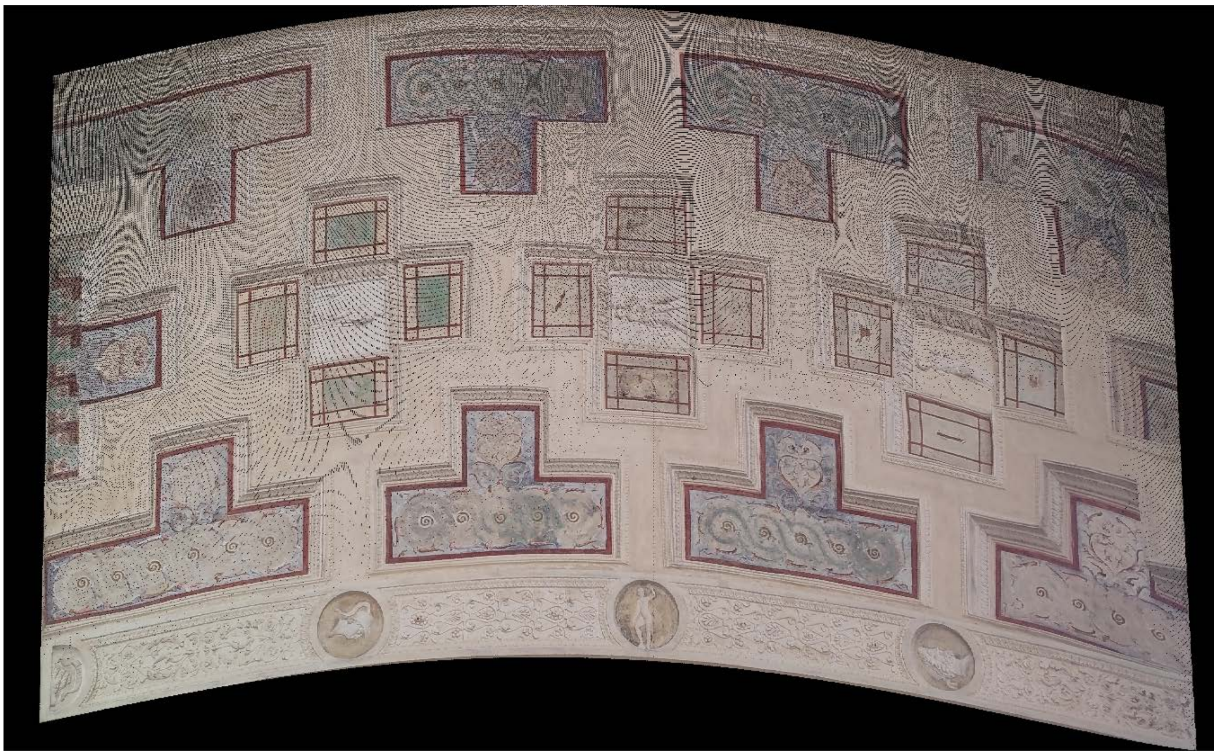
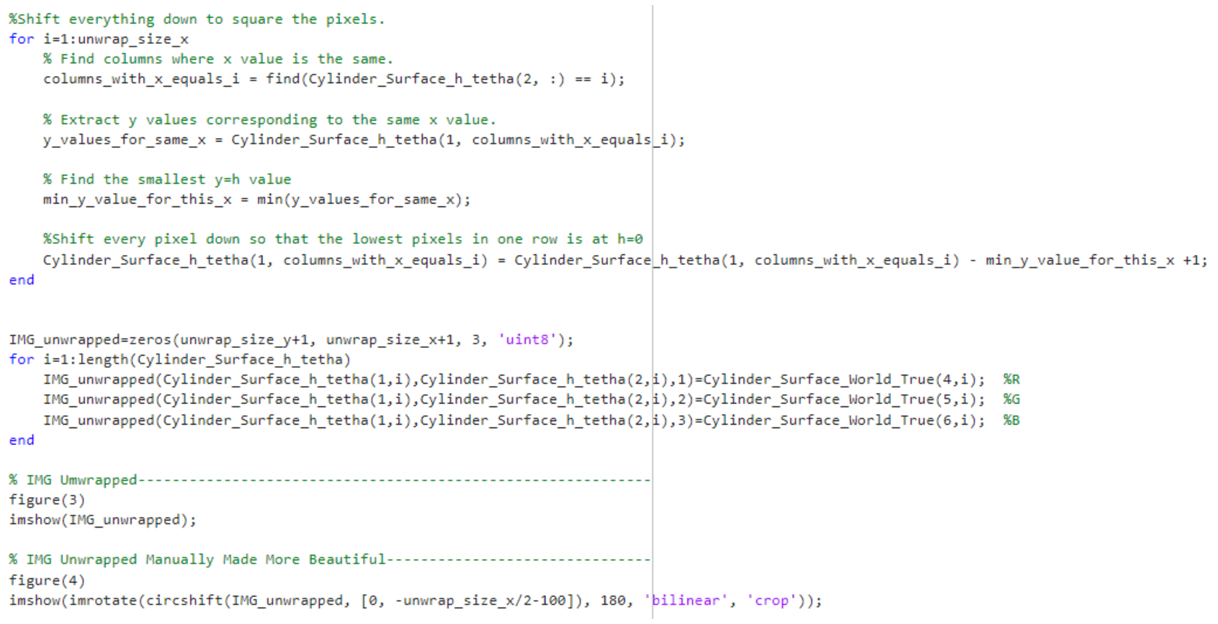


Figure 22: Initially Unwrapped Image

Now it can be seen in Figure 22 that there is still a bit of a curve in our image all though it should have been completely rectangular. We could actually see this curve while we were plotting these points in 3D as well so this is simply an imperfection inherent to the imperfect rectification and point sampling of our code. However, since at this point we know for a fact that this image should be rectangular and should not need any form of projective correction, we can apply a simple fix: At each θ value shift all points downwards such that the lowest point on that θ value is at $h = 1$. (Analogically think of this as squaring a stack of papers on a desk)



```
%Shift everything down to square the pixels.
for i=1:unwrap_size_x
    % Find columns where x value is the same.
    columns_with_x_equals_i = find(Cylinder_Surface_h_tetha(2, :) == i);

    % Extract y values corresponding to the same x value.
    y_values_for_same_x = Cylinder_Surface_h_tetha(1, columns_with_x_equals_i);

    % Find the smallest y=h value
    min_y_value_for_this_x = min(y_values_for_same_x);

    %Shift every pixel down so that the lowest pixels in one row is at h=0
    Cylinder_Surface_h_tetha(1, columns_with_x_equals_i) = Cylinder_Surface_h_tetha(1, columns_with_x_equals_i) - min_y_value_for_this_x +1;
end


IMG_unwrapped=zeros(unwrap_size_y+1, unwrap_size_x+1, 3, 'uint8');
for i=1:length(Cylinder_Surface_h_tetha)
    IMG_unwrapped(Cylinder_Surface_h_tetha(1,i),Cylinder_Surface_h_tetha(2,i),1)=Cylinder_Surface_World_True(4,i);  %R
    IMG_unwrapped(Cylinder_Surface_h_tetha(1,i),Cylinder_Surface_h_tetha(2,i),2)=Cylinder_Surface_World_True(5,i);  %G
    IMG_unwrapped(Cylinder_Surface_h_tetha(1,i),Cylinder_Surface_h_tetha(2,i),3)=Cylinder_Surface_World_True(6,i);  %B
end

% IMG Umwrapped----------------------------------------------------------
figure(3)
imshow(IMG_unwrapped);

% IMG Unwrapped Manually Made More Beautiful-----------------------------
figure(4)
imshow(imrotate(circshift(IMG_unwrapped, [0, -unwrap_size_x/2-100]), 180, 'bilinear', 'crop'));
```

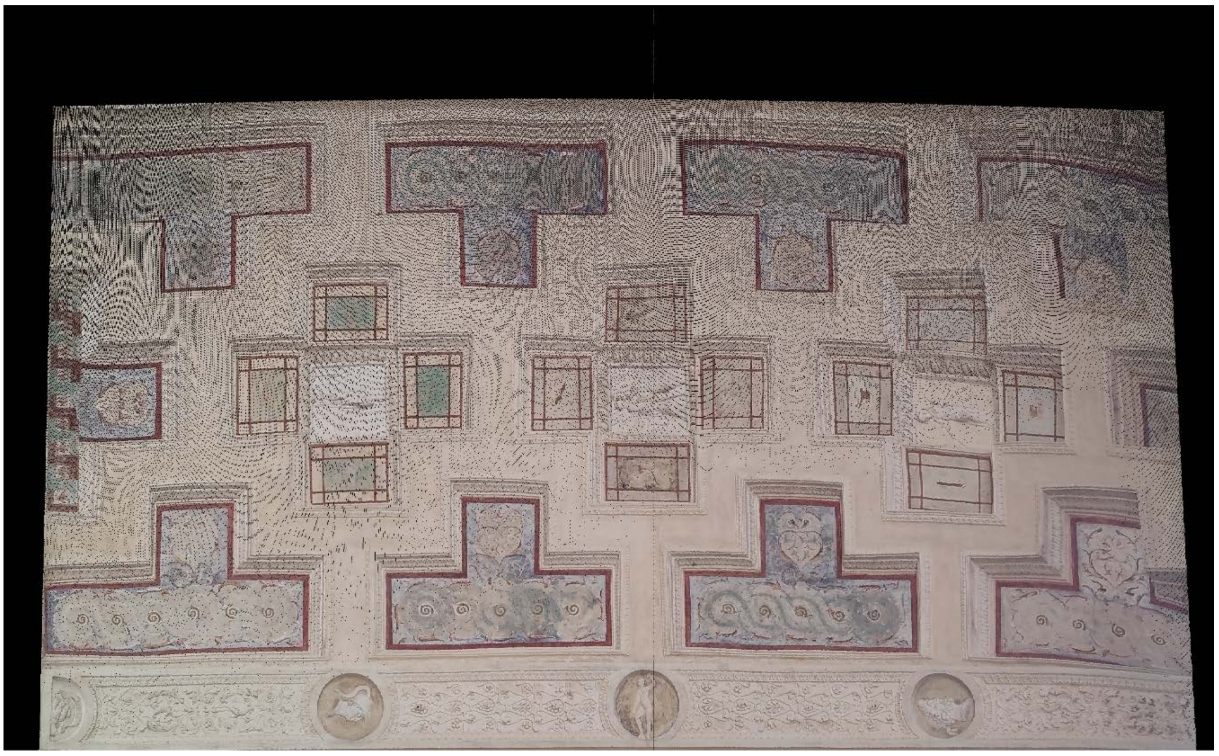


Figure 23: Final Unwrapped Cylindrical Surface

Finally, Figure 23 shows us the (mostly) rectangular unwrapped cylindrical surface image, thus also concluding this report.