IACV Homework

Sommario

Specification

In this work it has been analyzed and elaborated an image of Villa Melzi, a botanic garden situated in Bellagio, CO.

Firstly, It was requested to extract some features from the image, including edges, corners and straight lines. It was also requested to reconstruct the image, in order to compute the facades ratio, compute the calibration matrix K, reconstruct a facade from K and localize the camera that has taken the picture. Here it is shown a picture of the original image:

Each single step is executed in a different MATLAB script.

1. F1, Feature extraction

This step is devoted to feature extraction. To retrieve the features it must be called the script “Feature_extraction.m”, which contains 3 scripts for the purpose. The scripts used are “Edge_detecting.m”, “Corner_features.m”, and “Streight_lines.m”.

1.1. Edge etecting

In this step it is requested to find the edges of the image. For this purpose it is used a sobel filter. The scripts that performs this operation is “Edge_detecting.m”.

We need to smooth the image to obtain better results. To do this we can use a Prewitt smoothing filter or a Sobel smoothing filter. In this case it was chosen the Sobel one

First of all we create a differentiating filter and the smoothing Sobel filter and make the convolution between the 2, to obtain the final filters dx (for horizontal derivative) and dy (for vertical derivative).

```
disp('differentiating filters')
diffx=[1 -1]
diffy = diffx'

%smoothing filters Sobel
sx=[1 2 1 ; 1 2 1];
sy=sx';

% Build Sobel derivative filters
disp(' derivative filters Sobel')
dx=conv2(sy,diffx)
dy=conv2(sx,diffy)
```

```
dx =

     1     0    -1
     2     0    -2
     1     0    -1


dy =

     1     2     1
     0     0     0
    -1    -2    -1
```

Next we pick the original image and make it in grey scale, to let the filter works. It would have problems with an RGB image since it is a multidimensional matrix.

Now we compute the horizontal and vertical gradients, by convoluting the greyscale image and the filter computed before. We put ‘same’ in the function to return the central part of the convolution, which is the same size as y. Then the image with edges extracted is obtained by the square root of the 2 gradient components squared. Finally the image is thresholded in order to make the image binary. The threshold in this case is put to 0.25.

```
Gx=conv2(y , dx , 'same');    M=sqrt(Gx.^2 + Gy.^2);
Gy=conv2(y , dy , 'same');    M = M >= THRESHOLD;
```

Here there is the resultant image:

Edge Detected with Sobel

It is also possible to build the gradient image as a two plane image: the first plane contains horizontal derivatives, the second plane the vertical ones. So it is created Grad, that is a multidimensional matrix that has the dimension of the original image and as the first layer the horizontal derivatives and as second layer the vertical derivatives.

```
Grad=zeros(size(y,1),size(y,2),2);
Grad(:,:,1)=Gx;
Grad(:,:,2)=Gy;
```

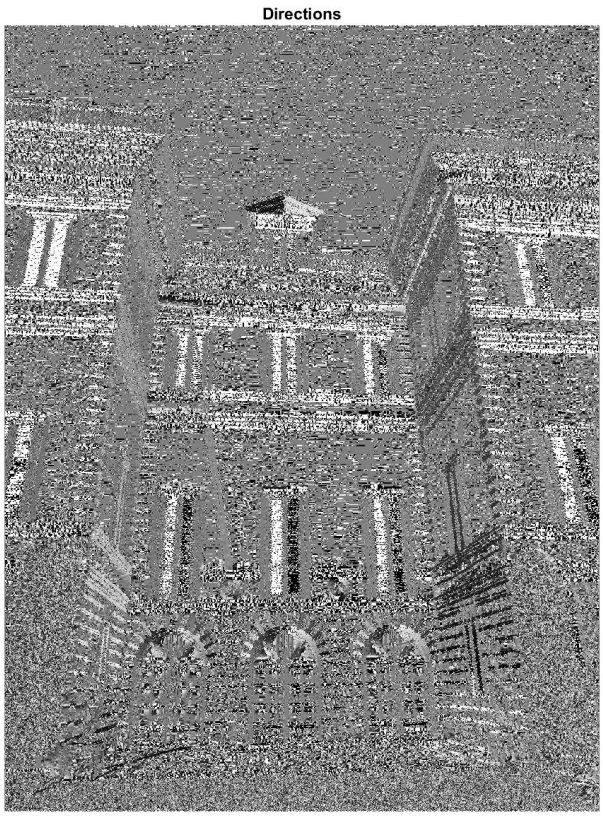
Next, it is computed the gradient norm image, with the same formula as before. In this case we need to remove the border since they are affected by zero padding.

```
Norm_Grad(1 : BORDER, :) = 0;
Norm_Grad(end - BORDER : end, :) = 0;
Norm_Grad(:, 1 : BORDER) = 0;
Norm_Grad(:, end - BORDER : end) = 0;
```

It is also computed the image that show the directions of the derivatives. In this case it is necessary to change the y sign since in MATLAB the vertical axis is directed downwards.

```
Dir_Grad=atand(- sign(Grad(:,:,1)).*Grad(:,:,2) ./(abs(Grad(:,:,1))+eps));
```

Here are the two images computed:



1.2. Corner extraction

The script for corner extraction is “Corner_features.m”. To extract corners from the original image, it was chosen the Harris method.

Firstly the derivative masks are built and it is computed the derivative of the image.

```
dx = [-1 0 1; -1 0 1; -1 0 1];
dy = dx';

Ix = conv2(y, dx, 'same');
Iy = conv2(y, dy, 'same');
```

Then it is set the Gaussian parameter. It is useful to set the window for Harris method as gaussian distributed weight, as it helps to reduce the noise.

```
SIGMA_gaussian=1
g = fspecial('gaussian',max(1,fix(3*SIGMA_gaussian)+1), SIGMA_gaussian);
```

Then the components of the matrix $M_{r,c}$ are computed.

$$M_{r,c} = \begin{bmatrix} (I_x^2 \circledast w)(r,c) & (I_x\, I_y \circledast w)(r,c) \\ (I_x\, I_y \circledast w)(r,c) & (I_y^2 \circledast w)(r,c) \end{bmatrix} \doteq \begin{bmatrix} I_x^2 \circledast w & I_x\, I_y \circledast w \\ I_x\, I_y \circledast w & I_y^2 \circledast w \end{bmatrix} (r,c)$$

```
Ix2 = conv2(Ix.^2, g, 'same');
Iy2 = conv2(Iy.^2, g, 'same');
Ixy = conv2(Ix.*Iy, g, 'same');
```

To find a corner, we must minimize the matrix $E_{x,y}(r,c)$, given the norm of [x,y] = 1. But $E_{x,y}(r,c)$ is equal to $[x;y]^T * M_{r,c} * [x;y]$. So we must maximize:

$$\min_{[x,y]} [x,y]\, M \begin{bmatrix} x \\ y \end{bmatrix}, \text{given } \|[x,y]\|_2 = 1$$

i.e. maximize smallest eigenvalue of M

But to maximize the smallest eigenvalue of M we need to use SVD method. To avoid using it, we can do something else. We know that

$$Tr(M) = \lambda_1 + \lambda_2$$
$$\det(M) = \lambda_1 \cdot \lambda_2$$

And that this formula

$$\det(M) - k\, Tr(M)$$

Is big when both eigenvalue are big and k = 0.04. We can use this result to compute the harris measure, but in this case we can obtain the trace and the determinant by the components of M.

```
k = 0.04;
cim = (Ix2.*Iy2 - Ixy.^2) - k * (Ix2 + Iy2);
```

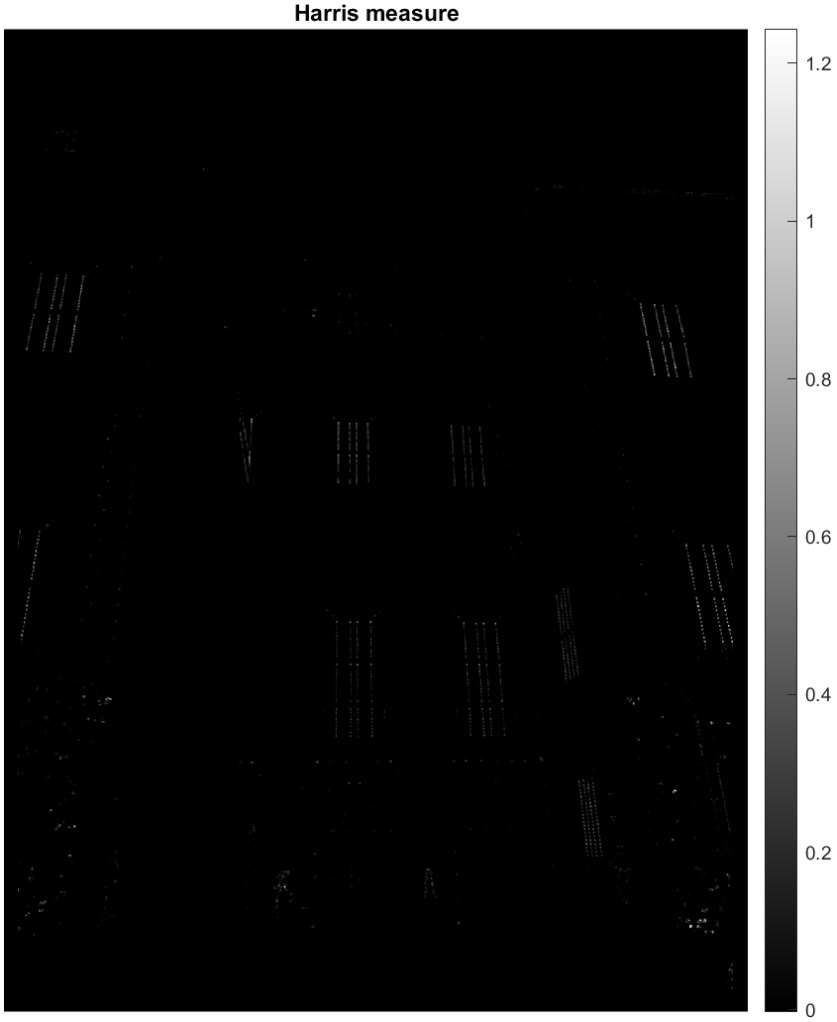
Also the borders are cut

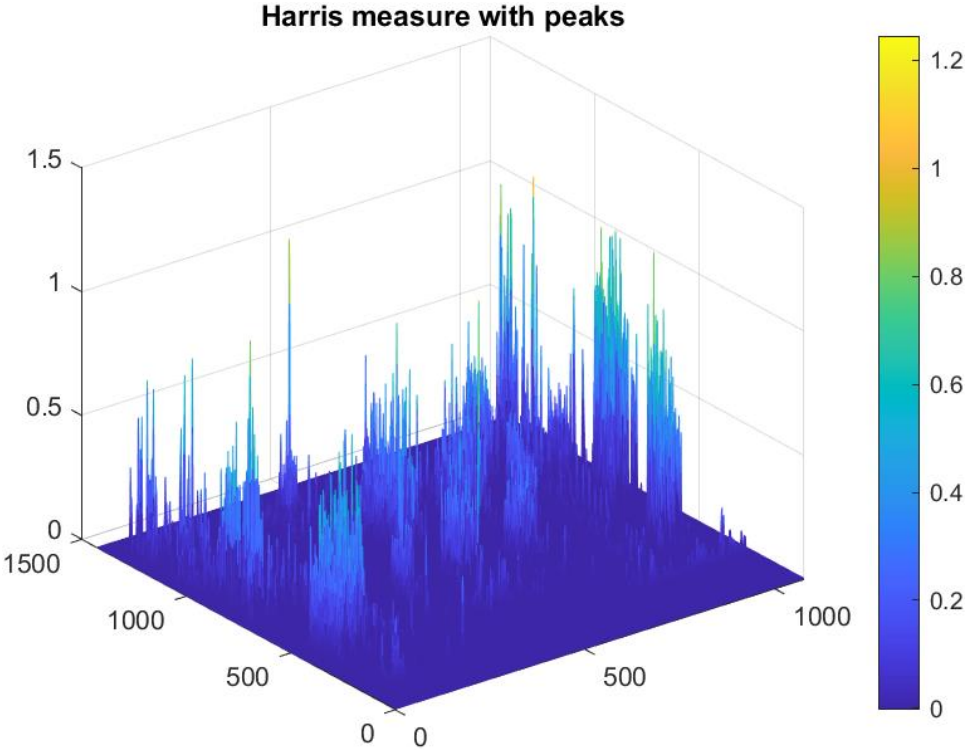
```
BORDER=20;
cim(1:BORDER,:)=0;
cim(end-BORDER:end,:)=0;
cim(:,end-BORDER:end)=0;
cim(:,1:BORDER)=0;
```

To normalize the Harris measure and better extracts the peaks, all the points below the mean value are set to 0

```
T=mean(cim(:));
CIM=cim;
CIM(find(cim<T))=0;
```

Then it is shown the Harris measure with a colorbar and a mesh of the image with the colorbar.

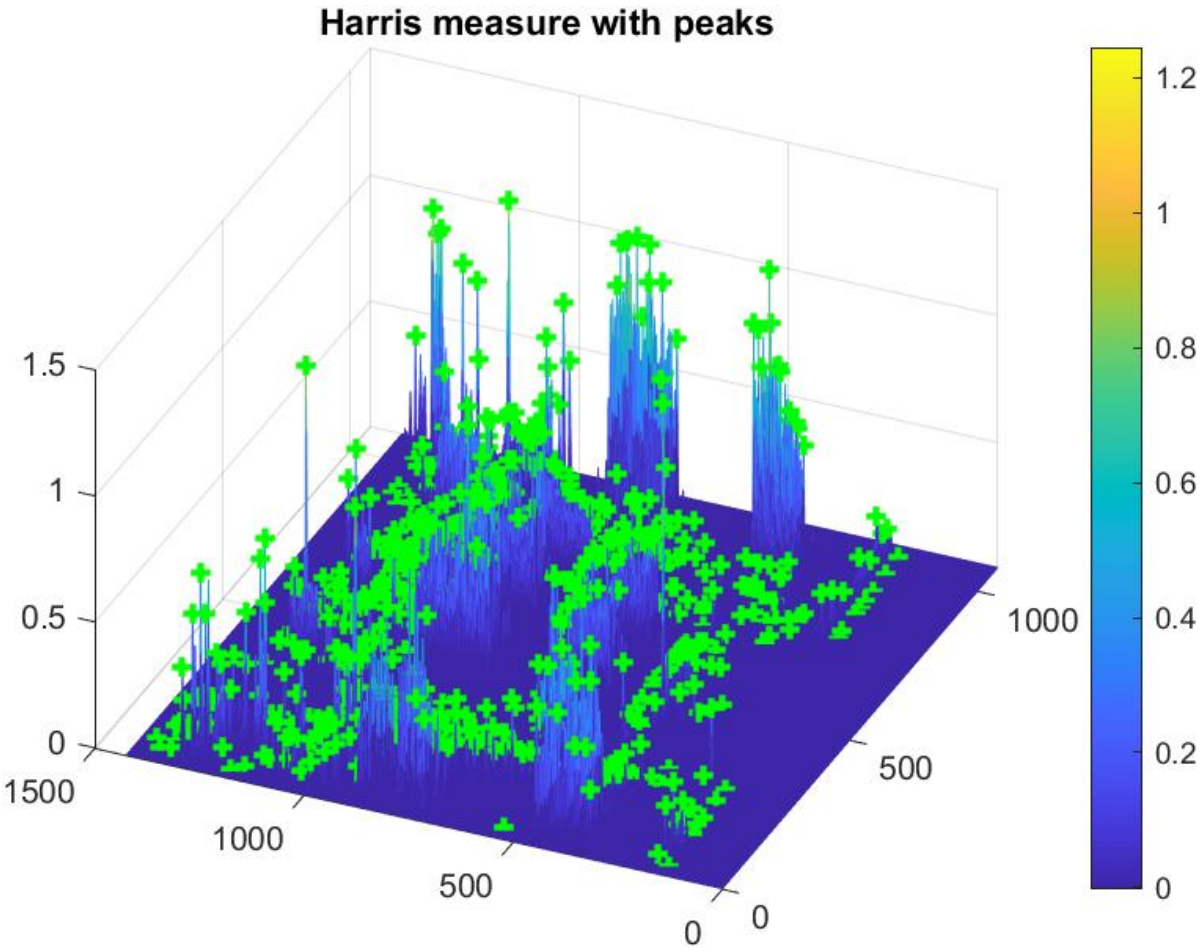




Now it is defined the moving window that will move around the Harris image. At each iteration the window is moved and the local maximum is found and the location of maximum are stored.

```
support=true(31);
% compute maximum over a square neighbor of size 31 x 31
maxima=ordfilt2(CIM,sum(support(:)),support);
% determine the locations where the max over the neigh or 31 x 31 corresponds to the cim values
[loc_x,loc_y]=find((cim==maxima).*(CIM>0));
indx = find((cim==maxima).*(CIM>0));
```

We plot the peaks onto the Harris image mesh and onto the original image, obtaining the following results:



Corners extracted

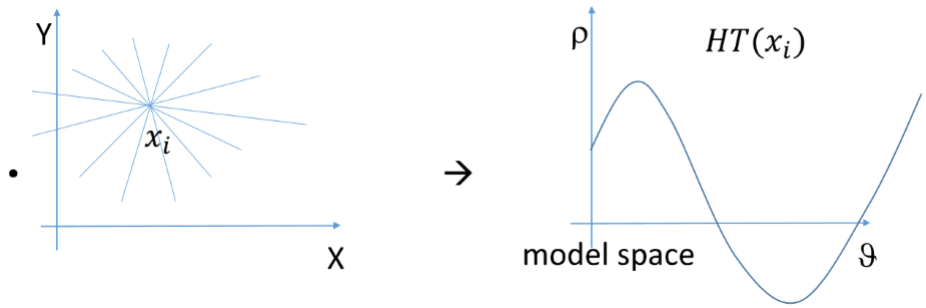
1.3. Straight lines extraction

The script that performs the lines extraction is “Straight_lines.m”. For this problem it was used the Hough transform.

To perform the Hough transform we need the image of the edges, that we retrieve from the previous point of edge extraction.

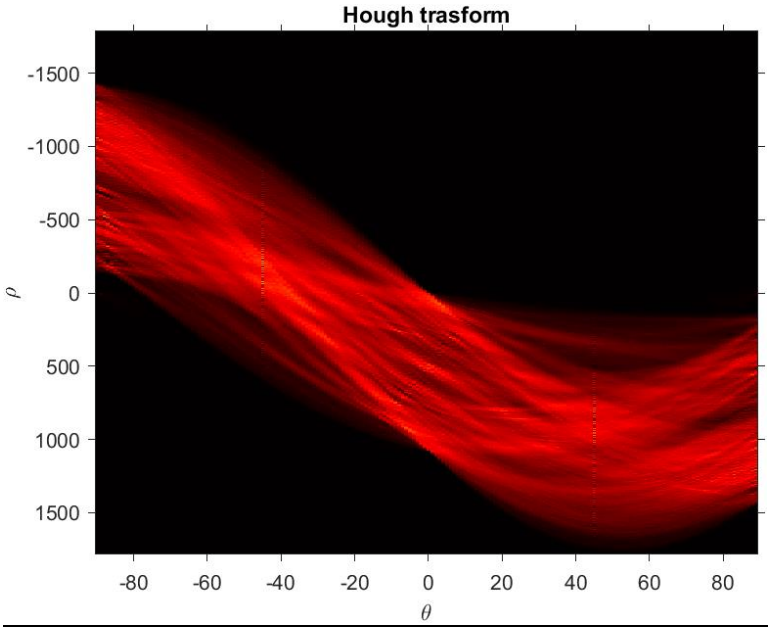
The Hough transform, is a representation of a point, through the parameter theta and rho, polar coordinates, of the set of lines crossing the point. So as Hough transform we obtain a set of lines, representing the points of the binary image.

$$\rho = X_i \cos\vartheta + Y_i \sin\vartheta$$

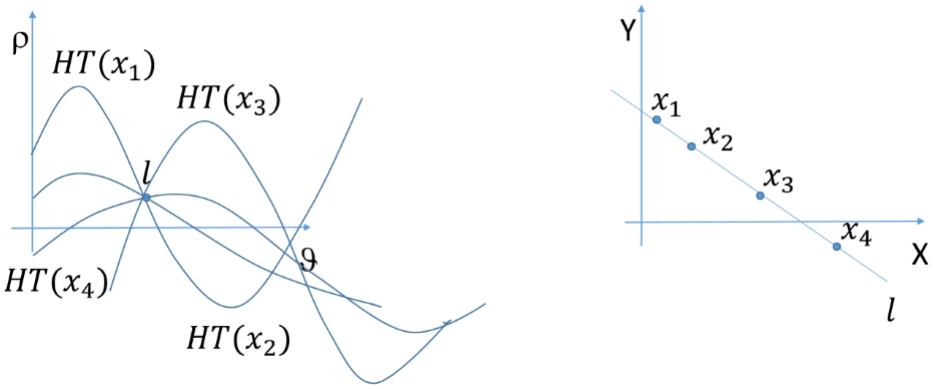


This is done through the function “hough” in MATLAB that returns R, the distance from the origin to the line along a vector perpendicular to the line, and theta, the angle in degrees between the x-axis and this vector. The function also returns the SHT (Standard Hough Transform), H, which is a parameter space matrix whose rows and columns correspond to rho and theta values respectively.

```
[H,T,R] = hough(BW);
figure(8);
imshow(H,[],'XData',T,'YData',R,...
            'InitialMagnification','fit');
xlabel('\theta'), ylabel('\rho'),title('Hough trasform')
axis on, axis normal, hold on;
colormap(gca,hot);
```



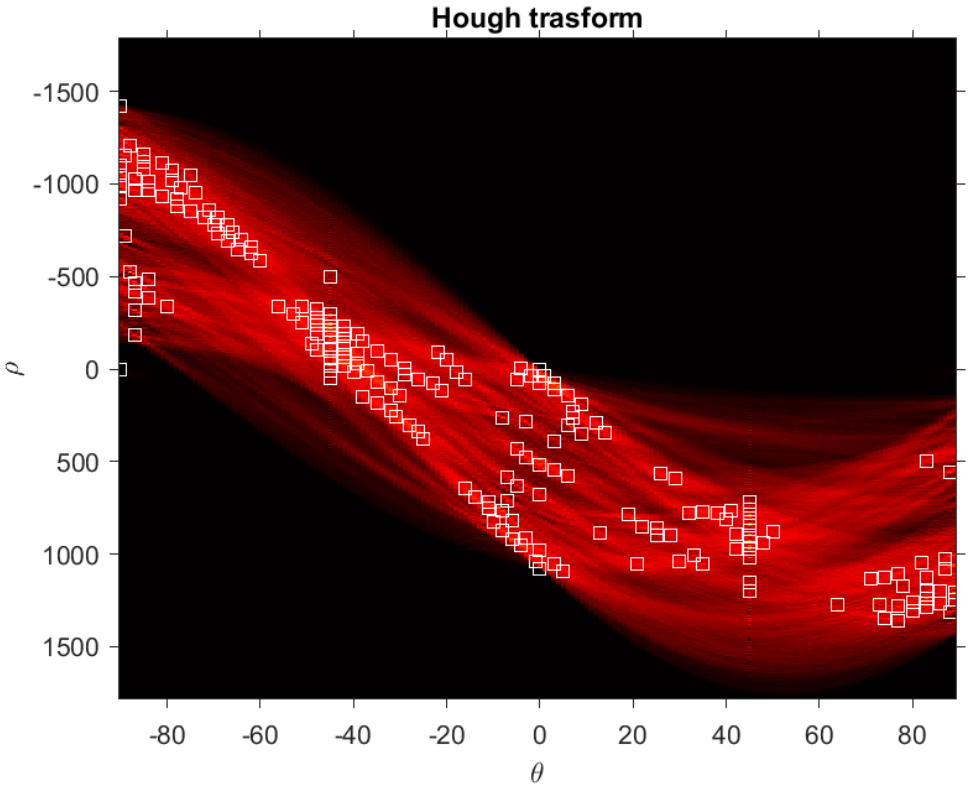
Now each points which is a local maximum in the Hough transform, is a line in the original image that contains many points.



So we need to find local maximum in the Hough transform. To do so we divide the space of the Hough transform into many cells and count the number of lines that passes through that cell. The cells with the final count greater than a threshold are chosen. So the point representing that cell can be mapped back to a line in the original image. This process can be done by a MATLAB function "houghlines", which given the Hough transform H, the maximum number of peaks that can be identified, the vector of possible thetas T, and a threshold, return a matrix P that contains the coordinates of the peaks. The peaks are then plotted onto the image of the Hough transform.

```
P = houghpeaks((H), 200, 'Theta', T, 'Threshold', 0.05 * max(H(:)));
x = T(P(:,2)); y = R(P(:,1));
plot(x,y,'s','color','white');
```

In this case the maximum number of peaks was set to 200 and the threshold to the global maximum of H multiplied by 0.05.



Next we need to transform the peaks found back to lines, and this is done by the function houghlines, already implemented in MATLAB. This function given the binary image, vector of thetas, vector of rhos, matrix of peaks coordinates P, return the lines corresponding to the peaks. Other parameters are “FillGap” (distance under with 2 lines are merged together as one) and “Minlength” (minimum line length). These parameters were set respectively to 10 and 200.

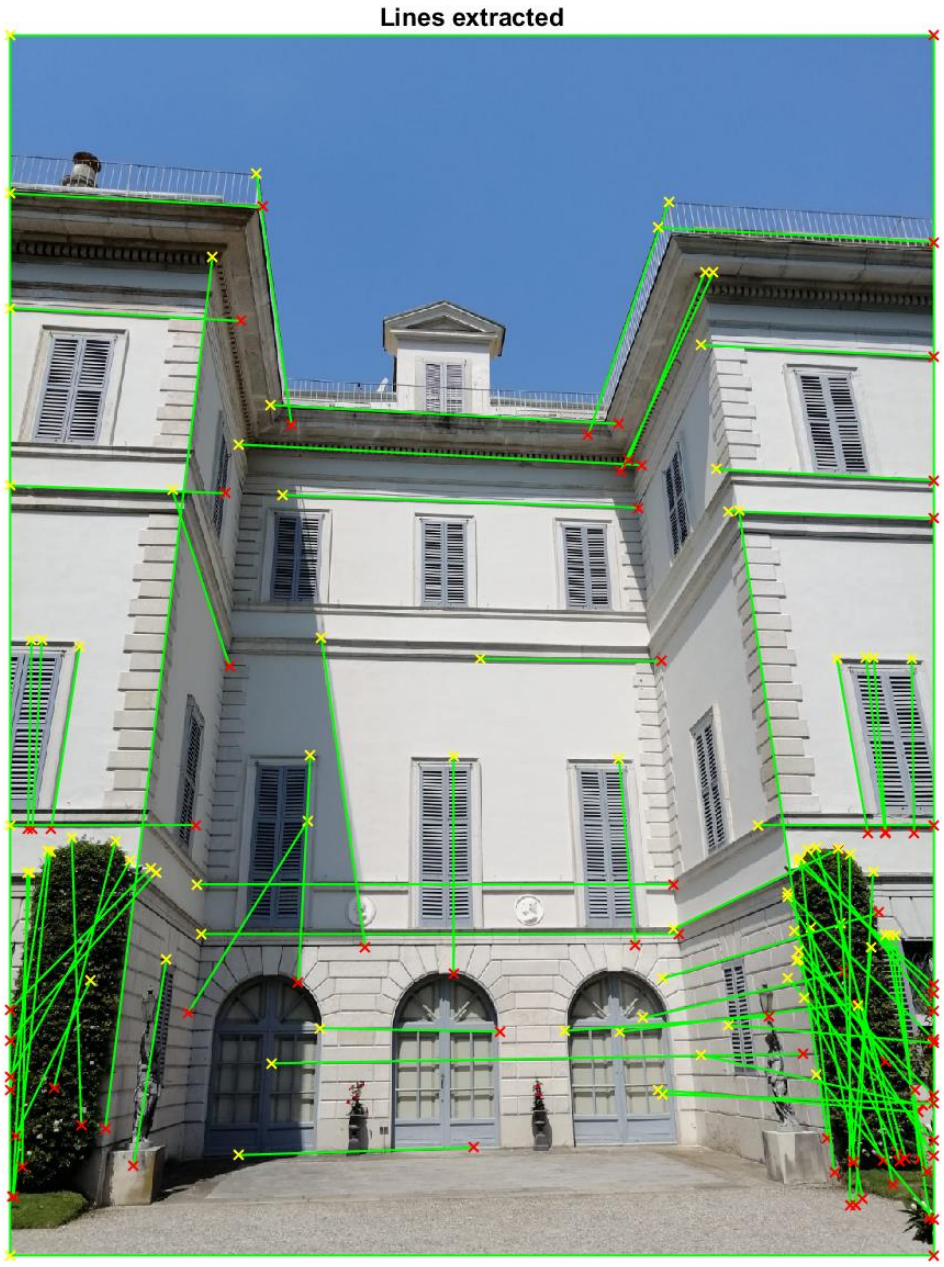
Then all the lines returned are plotted onto the original image(with their relatives end and beginning).

```
lines = houghlines(BW,T,R,P,'FillGap',10,'MinLength',200);
figure(9), imshow(I),title('Lines extracted'), hold on
for k = 1:length(lines)
   xy = [lines(k).point1; lines(k).point2];
   plot(xy(:,1),xy(:,2),'LineWidth',1,'Color','green');

   % Plot beginnings and ends of lines
   plot(xy(1,1),xy(1,2),'x','LineWidth',1,'Color','yellow');
   plot(xy(2,1),xy(2,2),'x','LineWidth',1,'Color','red');

end
```

Finally here is the final image with the lines extracted:



2. 2D Reconstruction

In this step it is reconstructed the horizontal section of the building, and then it's computed the ratio between the two orthogonal facades (2 and 3), since after the metric rectification the proportion of the facades should be equal to the one in the reality.

To perform this operation a stratified approach has been chosen. The script that execute the code is "Stratified_2D". Because of the stratified approach first we need to execute an affine transformation.

First, the image is loaded and some global variables are set.

```
I = imread("Villa image 2.png");
figure();
imshow(I), title("Original image");
hold on;

% Set global variables
FNT_SZ = 10;
PNT_SZ = 10;
x_disc = 4;
y_disc = 10;
```

Then some useful points are obtained by hand, selected from the original image. A final 1 is added as third coordinate to write the point in homogeneous coordinates. The points are also plotted onto the original image with a label with it.

```
A = [100.6698, 1.3690e+03, 1]';  plot(A(1), A(2), 'y.', 'MarkerSize', PNT_SZ);
B = [107.3153, 1.3665e+03, 1]';  plot(B(1), B(2), 'y.', 'MarkerSize', PNT_SZ);
C = [165.1699, 1.3644e+03, 1]';  plot(C(1), C(2), 'y.', 'MarkerSize', PNT_SZ);
D = [176.8972, 1.3546e+03, 1]';  plot(D(1), D(2), 'y.', 'MarkerSize', PNT_SZ);
E = [201.9849, 1.3129e+03, 1]';  plot(E(1), E(2), 'y.', 'MarkerSize', PNT_SZ);
F = [508.6777, 1.3041e+03, 1]';  plot(F(1), F(2), 'y.', 'MarkerSize', PNT_SZ);
G = [802.5925, 1.2952e+03, 1]';  plot(G(1), G(2), 'y.', 'MarkerSize', PNT_SZ);
H = [883.1107, 1.3340e+03, 1]';  plot(H(1), H(2), 'y.', 'MarkerSize', PNT_SZ);
L = [903.1667, 1.3409e+03, 1]';  plot(L(1), L(2), 'y.', 'MarkerSize', PNT_SZ);
M = [953.1747, 1.3402e+03, 1]';  plot(M(1), M(2), 'y.', 'MarkerSize', PNT_SZ);
N = [962.6749, 1.3434e+03, 1]';  plot(N(1), N(2), 'y.', 'MarkerSize', PNT_SZ);
P = [335.0041, 1.3067e+03, 1]';  plot(P(1), P(2), 'y.', 'MarkerSize', PNT_SZ);
Q = [225.0074, 1.3579e+03, 1]';  plot(Q(1), Q(2), 'y.', 'MarkerSize', PNT_SZ);
R = [269.6324, 1.3174e+03, 1]';  plot(R(1), R(2), 'y.', 'MarkerSize', PNT_SZ);
S = [746.1883, 1.3013e+03, 1]';  plot(S(1), S(2), 'y.', 'MarkerSize', PNT_SZ);
T = [841.6039, 1.3381e+03, 1]';  plot(T(1), T(2), 'y.', 'MarkerSize', PNT_SZ);
```

```
text(A(1)+x_disc, A(2)+y_disc, 'A', 'FontSize', FNT_SZ, 'Color', 'y');
text(B(1)+x_disc, B(2)+y_disc, 'B', 'FontSize', FNT_SZ, 'Color', 'y');
text(C(1)+x_disc, C(2)+y_disc, 'C', 'FontSize', FNT_SZ, 'Color', 'y');
text(D(1)+x_disc, D(2)+y_disc, 'D', 'FontSize', FNT_SZ, 'Color', 'y');
text(E(1)+x_disc, E(2)+y_disc, 'E', 'FontSize', FNT_SZ, 'Color', 'y');
text(F(1)+x_disc, F(2)+y_disc, 'F', 'FontSize', FNT_SZ, 'Color', 'y');
text(G(1)+x_disc, G(2)+y_disc, 'G', 'FontSize', FNT_SZ, 'Color', 'y');
text(H(1)+x_disc, H(2)+y_disc, 'H', 'FontSize', FNT_SZ, 'Color', 'y');
text(L(1)+x_disc, L(2)+y_disc, 'L', 'FontSize', FNT_SZ, 'Color', 'y');
text(M(1)+x_disc, M(2)+y_disc, 'M', 'FontSize', FNT_SZ, 'Color', 'y');
text(N(1)+x_disc, N(2)+y_disc, 'N', 'FontSize', FNT_SZ, 'Color', 'y');
text(P(1)+x_disc, P(2)+y_disc, 'P', 'FontSize', FNT_SZ, 'Color', 'y');
text(Q(1)+x_disc, Q(2)+y_disc, 'Q', 'FontSize', FNT_SZ, 'Color', 'y');
text(R(1)+x_disc, P(2)+y_disc, 'R', 'FontSize', FNT_SZ, 'Color', 'y');
text(S(1)+x_disc, S(2)+y_disc, 'S', 'FontSize', FNT_SZ, 'Color', 'y');
text(T(1)+x_disc, T(2)+y_disc, 'T', 'FontSize', FNT_SZ, 'Color', 'y');
```

Then are computed the main lines joining the 2 points. Since the line must be in homogenous coordinates, it is possible to compute them just by the cross product of the 2 points that the line cross.

```
lae = cross(A,E);
lan = cross(A,N);
lbc = cross(B,C);
lcd = cross(C,D);
lbe = cross(B,E);
lef = cross(E,F);
leg = cross(E,G);
lfg = cross(F,G);
lgm = cross(G,M);
lhl = cross(H,L);
llm = cross(L,M);
lnp = cross(N,P);
lgn = cross(G,N);
lbm = cross(B,M);
lqr = cross(Q,R);
lrs = cross(R,S);
lst = cross(S,T);
ltq = cross(T,Q);
lap = cross(A,P);
```

To compute the matrix for the affine reconstruction, we need the image of the line at infinity, that will be the last row of the matrix Har, the matrix for the affine reconstruction.

$$H_{AR} = \begin{bmatrix} * & * & * \\ * & * & * \\ & \boldsymbol{l'}_{\infty}{}^{T} & \end{bmatrix}$$

Since the lines at infinity contains all the vanishing points, it is possible to compute 2 vanishing points from 2 pairs of parallel lines and then compute the line crossing the two vanishing points, that will be the image of the line at the infinity.

Therefore, two pairs of parallel lines are taken. In this case the line joining AE and GN for the first vanishing point v1 and EG and AN for the second vanishing point v2. After this we need to normalize the points by dividing each value for the third coordinate (this way the third coordinate will be 1). Now it is possible to calculate the image of the line at the infinity by executing the cross product between v1 and v2.

```
v1 = cross(lae, lgn);
v1 = v1./v1(3);
v2 = cross(leg, lan);
v2 = v2./v2(3);

% Compute the image of the line at infinity
imLinfty = cross(v1,v2);
imLinfty = imLinfty./(imLinfty(3));
```

The vanishing points and the lines are also plotted onto the original image.

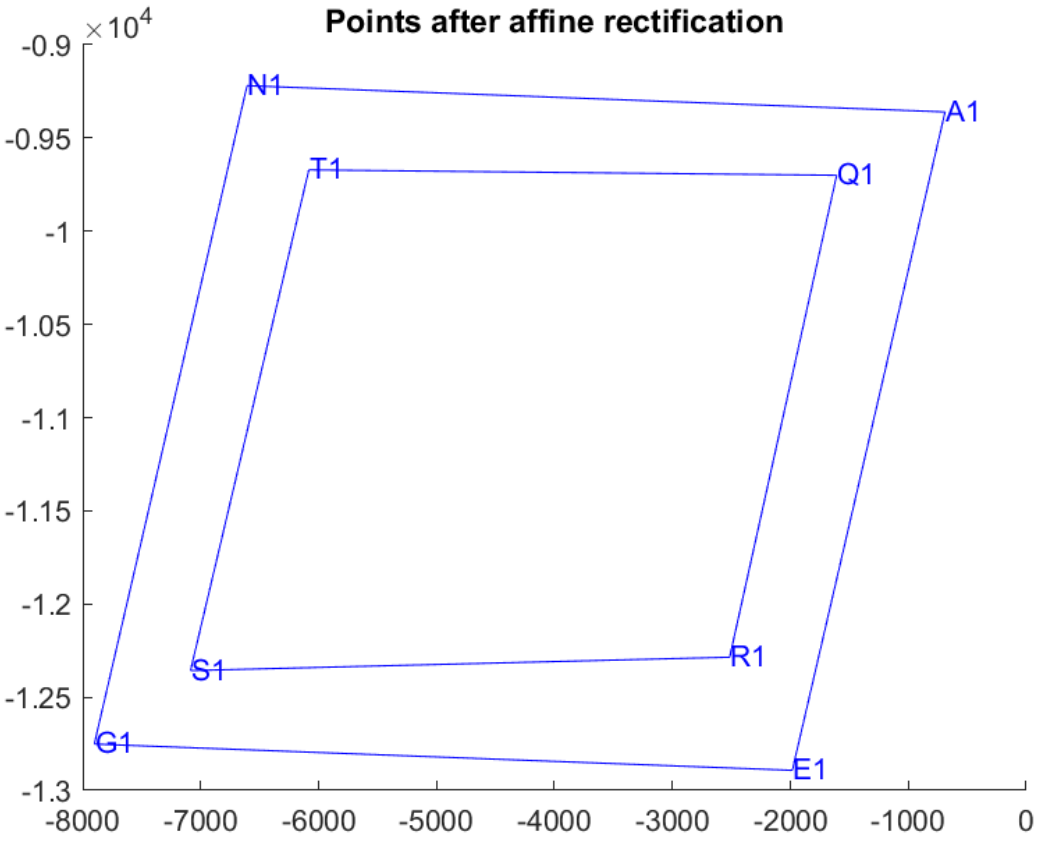
Hence the matrix for the affine rectification is build by putting 1s on the diagonal, the image of the line at infinity in the last row, and 0s elsewhere.

```
Har = [eye(2),zeros(2,1); imLinfty(:).']
```

```
Har =

    1.0000         0         0
         0    1.0000         0
   -0.0000   -0.0008    1.0000
```

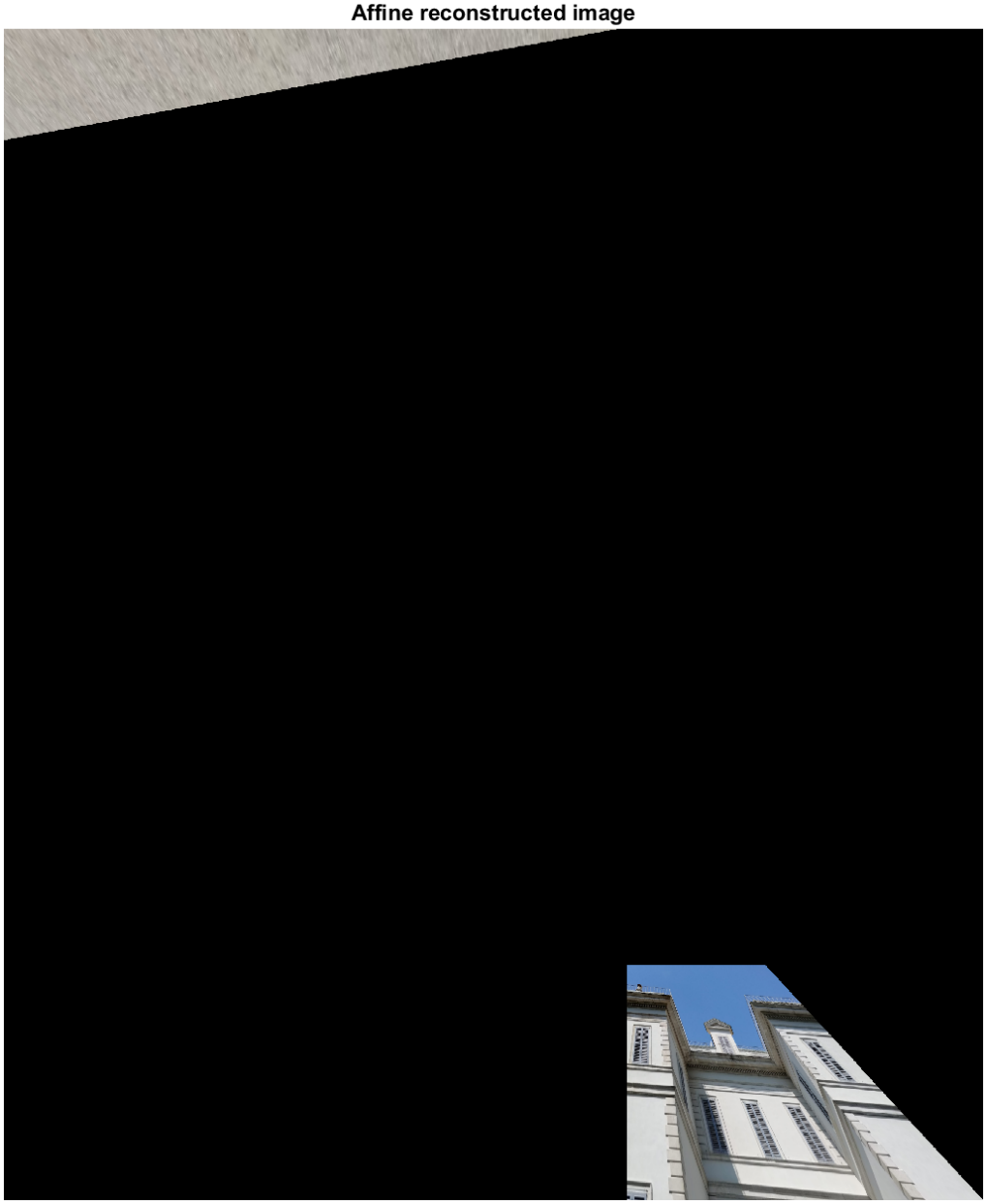
Now some of the points are multiplied by Har, to get their coordinates in the affined reconstructed image. The points plotted represent the points of the floor and its internal rectangle:



The functions projective2D and imwarp, that given an image ang a matrix, recontruct the image traformed as the matrix. The image is the shown.

```
tform = projective2d(Har.');
J = imwarp(I, tform);
figure();
title("Affine reconstructed image");
imshow(J);
hold on;
```

Affine reconstructed image



As we can see the edges of the facades are parallel to each other as an affine projection should produce.

To perform a metric rectification we need the image of the conic dual to the circular points. Then we should fine a transformation that pushes the image of the conic dual to the circular points back to the original one. The conic dual to the circular points has this form

$$C_\infty^* = \boldsymbol{IJ^T} + \boldsymbol{JI^T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

But in an affined reconstructed image, the image of the conic dual to the circular point has the form

$$C_\infty^{*\prime} = \begin{bmatrix} KK^\intercal & \mathbf{0} \\ \mathbf{0}^\intercal & 0 \end{bmatrix}$$

So we just need to find the 2x2 matrix KK^T, and to do so we need just 2 pairs of orthogonal lines. This because if in the 3D world the 2 lines are orthogonal, this constraint is valid.

$$\boldsymbol{l'}^{\mathsf{T}} C_{\infty}^{*\prime} \boldsymbol{m'} = 0$$

Therefor since we have just 2 unknowns for the image of the conic dual to the circular points, we just need 2 equations and 2 pairs of orthogonal lines.

To do so 4 points were taken from the affine reconstructed image, and the lines crossing them (orthogonal between each other), were computed as before.

```
A1 = [5342.89, 8310.39, 1].';  l1 = cross(A1,B1);  l1 = l1./l1(3);
B1 = [5587.77, 9001.45, 1].';  m1 = cross(B1,C1);  m1 = m1./m1(3);
C1 = [6746.04, 9094.59, 1].';  s1 = cross(C1,D1);  s1 = s1./s1(3);
D1 = [6507.18, 8411.05, 1].';  t1 = cross(D1,A1);  t1 = t1./t1(3);
```

Affine reconstructed image

Now, the constraints are built and the SVD is used to retrieve the matrix St, which is KK^T. In this case we take the last column of matrix v, which is the eigenvector associated with the smallest eigenvalue. This has three components which are the component of the matrix St (remember that St is symmetric).

```
St(1,:) = [l1(1)*m1(1),0.5*(l1(1)*m1(2)+l1(2)*m1(1)),l1(2)*m1(2)];
St(2,:) = [s1(1)*t1(1),0.5*(s1(1)*t1(2)+s1(2)*t1(1)),s1(2)*t1(2)];

% Execute svd to find the solutions of the matrix of the equations
[~,~,v] = svd(St); %
sol = v(:,end); %sol = (a,b,c)  [a,b/2; b/2,c];
St = [sol(1)  , sol(2)/2;
    sol(2)/2, sol(3)];
```

Then to compute K it is possible to use the function chol(St) to obtain it. Since it holds that

$$\text{From} \quad C_\infty^{*\prime} = H_A C_\infty^* H_A^T = \begin{vmatrix} K & \boldsymbol{t} \\ \mathbf{0} & 1 \end{vmatrix} \cdot \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{vmatrix} \cdot \begin{vmatrix} K^T & \mathbf{0} \\ \boldsymbol{t}^T & 1 \end{vmatrix}$$

$$\text{and} \quad H_{rect} = H_A^{-1}$$

$$H_{rect} = \begin{vmatrix} K & t \\ 0 & 1 \end{vmatrix}^{-1}$$

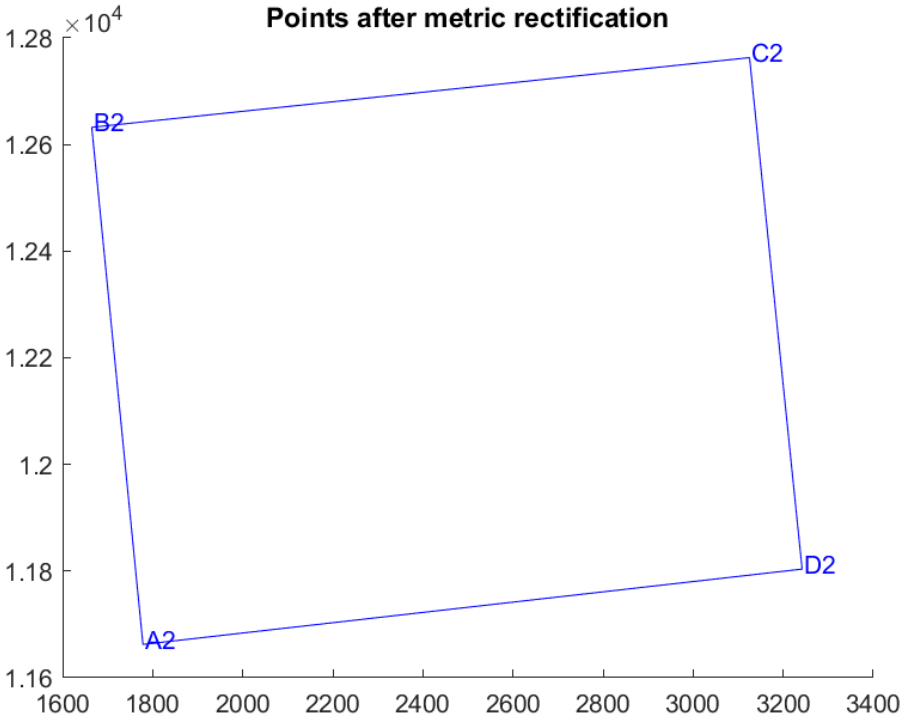
Ha is first obtained and then Hrect is set as Ha inverse. In this case we set t values to 0.

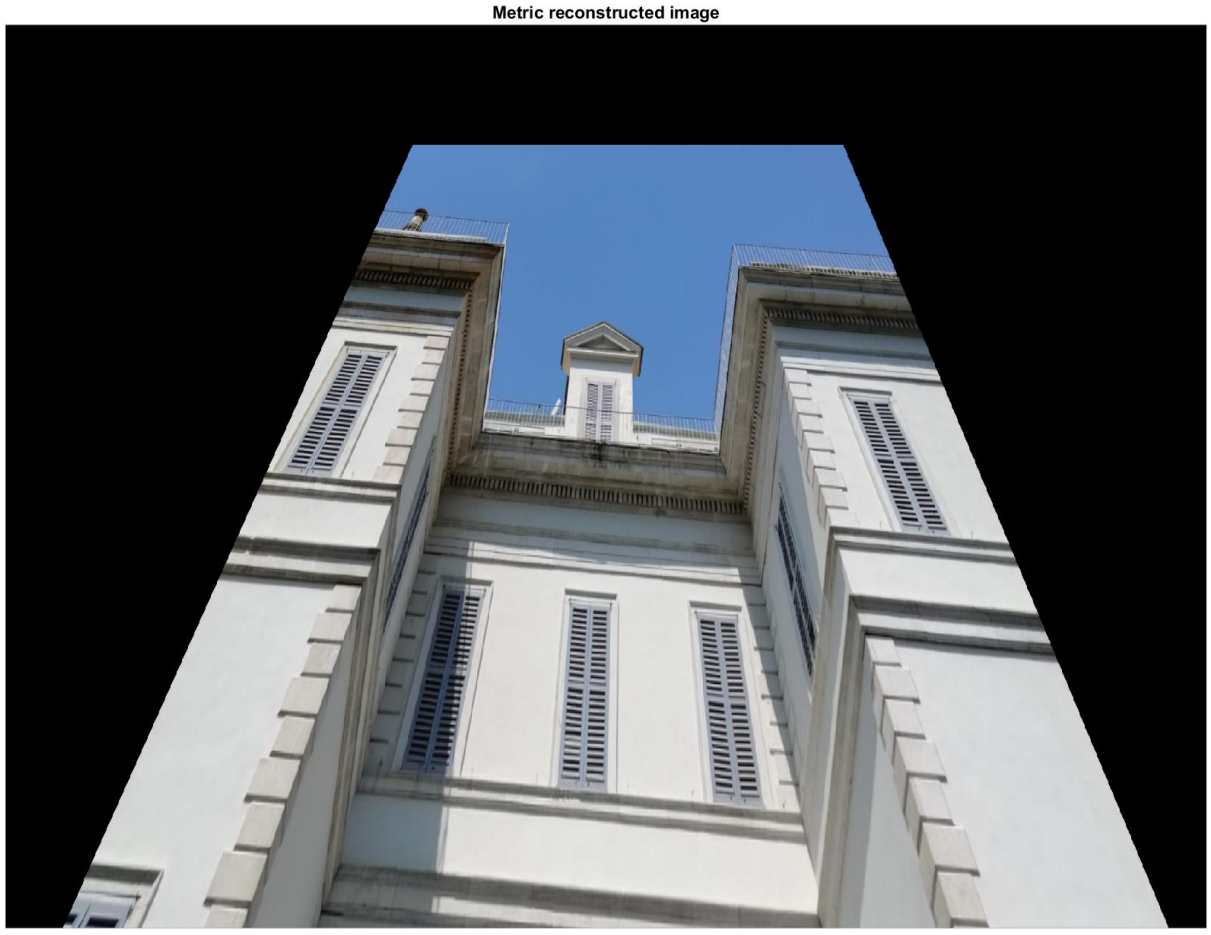
```
K = chol(St);
Ha = [K, zeros(2,1); zeros(1,2), 1];
Hsr = inv(Ha)
```

```
Hsr =

    1.3115   -0.6292         0
         0    1.4034         0
         0         0    1.0000
```

Finally the points taken from affine reconstructed image are plot after the metric rectification, and also the image is shown with the same method used before (projective2D and imwarp, using the affine image and Hsr).





As it is possible to see, now the horizontal section has a rectangular shape, as in the reality, and the proportion between the edges lengths are the same as the real ones

The ratio between facades 2 and 3 is calculated by taking the length of the segment A2B2 (Facade 2) and the segment B2C2 (Facade 3).

```
s1 = distance(A2,B2);
s2 = distance(B2,C2);
r = s1/s2;
fprintf("Facade ratio is %2.4f\n", r);
```

The length of the segment is calculated by the function distance that computes the 2D euclidean distance between 2 points.

```
function [d] = distance(a,b)
    d = sqrt((a(1)-b(1))^2+(a(2)-b(2))^2);
end
```

The calculated ratio is

```
Facade ratio is 0.6659
```

Is is also computed the Hrect = Hsr*Har, the complete rectifing matrix that will be useful for next points.

```
Hrect =

    1.3115   -0.6292         0
         0    1.4034         0
   -0.0000   -0.0008    1.0000
```

3. Camera calibration

In this step the calibration matrix of the camera K is found. The script performing this operation is "Camera_calibration.m".

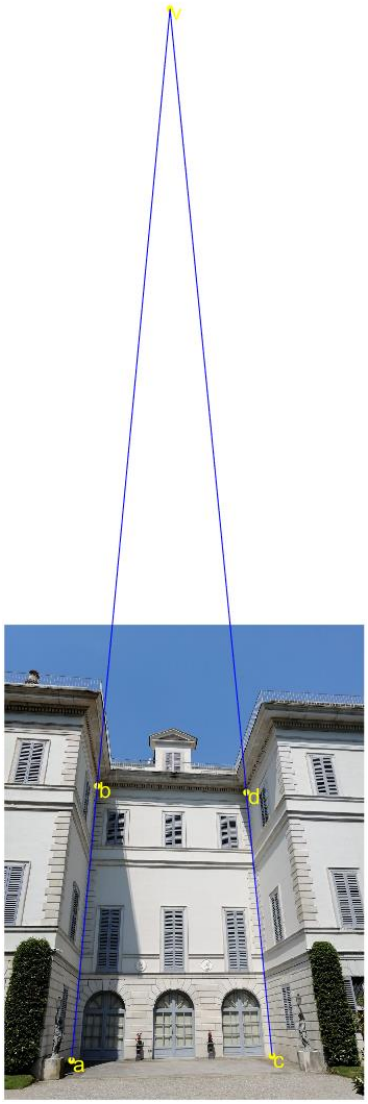
To compute the matrix K we can use the rectification matrix found in the previous point. To do so we need to find a vanishing point v along the direction orthogonal to the face (in this case the horizontal section).

Therefore the four corners of the facade 3 are taken by hand. As before the 1 as third coordinates is added to make the points in homogeneous coordinates

```
a = [202, 1.3070e+03, 1]';
b = [279, 0.4840e+03, 1]';
c = [803, 1.2930e+03, 1]';
d = [726, 0.5030e+03, 1]';
```

These points are plotted onto the original image and then the vanishing point v is computed, using the lines ab and cd which are parallel to each other

Original image




```
v = cross(cross(a,b), cross(c,d));

%Normalize v
v = v./v(3);
```

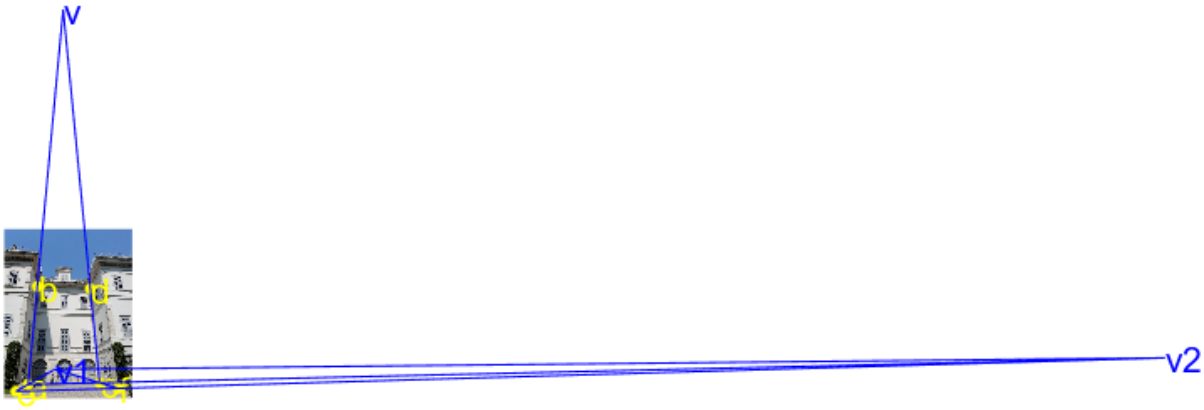
Now two other points are added in order to compute 2 vanishing points of the horizontal section.

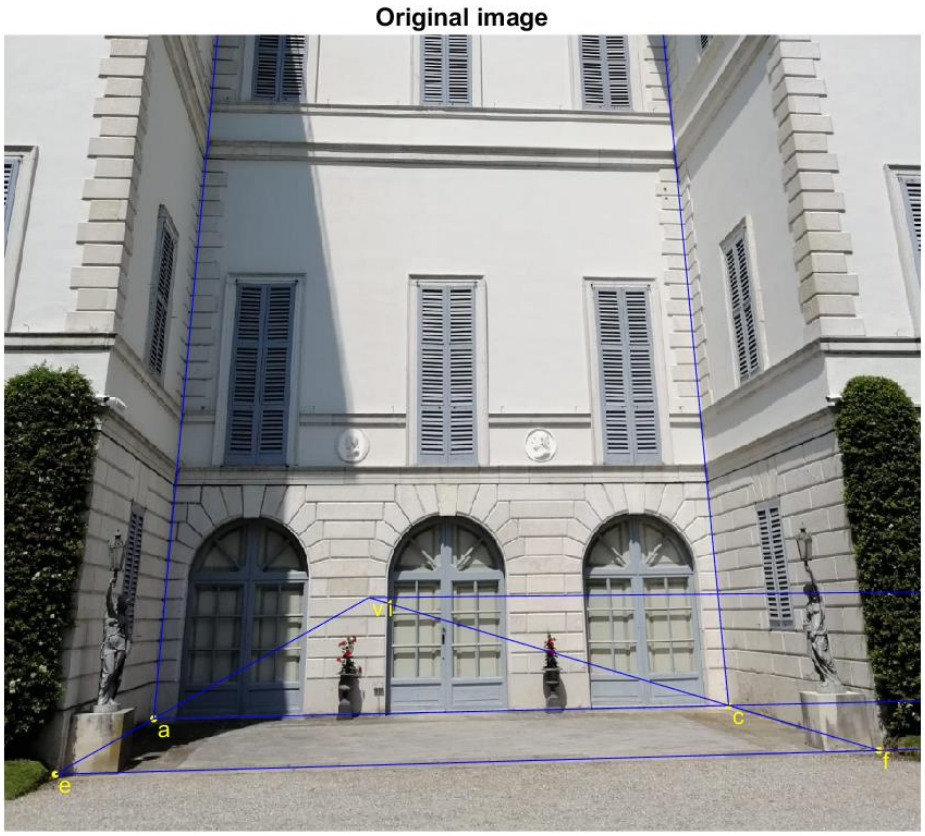
```
e = [99, 1.3650e+03, 1]';
f = [960, 1.34e+03, 1]';
```

These points are plotted and the vanishing points v1 and v2 are computed (these are the same of the previous point). Also the image of the line at the infinity crossing v1 and v2 can be computed.

```
v1 = cross(cross(e,a), cross(f,c));
v2 = cross(cross(e,f), cross(a,c));
v1 = v1./v1(3);
v2 = v2./v2(3);
```

```
linf = cross(v1,v2);
```





Then the rectification matrix is taken from the previous step

```
Hr = [1.31145040302845 -0.629152571724488  0;
                    0  1.403360600698     0;
     -2.41854176634543e-05 -0.000835517704230148  1];
```

We can obtain the first 2 equations by

from $\mathbf{v}_1^T \omega \mathbf{v} = 0$ and $\mathbf{v}_2^T \omega \mathbf{v} = 0$, and $\boldsymbol{l}'_\infty = \mathbf{v}_1 \times \mathbf{v}_2$ →

$$\boxed{\boldsymbol{l}'_\infty = \omega \mathbf{v} \text{ (2 eqns)}}$$

They are just 2 equations because the points are in homogeneous coordinates, so one equation is a linear combination of the other 2.

But it also holds that

$$\text{from } I' = H_R^{-1} I = H_R^{-1} \begin{bmatrix} 1 \\ i \\ 0 \end{bmatrix} = [h_1 \quad h_2 \quad h_3] \begin{bmatrix} 1 \\ i \\ 0 \end{bmatrix} = h_1 + i h_2,$$

$$\text{and } (h_1 + i h_2)^T \omega (h_1 + i h_2) = 0 \rightarrow$$

$$\boxed{\begin{aligned} {h_1}^T \omega h_2 &= 0 \\ {h_1}^T \omega h_1 - {h_2}^T \omega h_2 &= 0 \end{aligned}}$$

Where I' is the image of the circular point I and h1, h2 and h3 are the columns of the inverse of the rectifying matrix.

So to set up the constraints first the matrix Hr is inverted and then it is divided into columns h1, h2 and h3. Also from the line at infinity we construct a matrix to write in a simpler way the constraints.

```
Hr_inv = inv(Hr);
                    l_1 = linf(1,1);
h1 = Hr_inv(:,1);   l_2 = linf(2,1);
h2 = Hr_inv(:,2);   l_3 = linf(3,1);
h3 = Hr_inv(:,3);   lx = [0 -l_3 l_2; l_3 0 -l_1; -l_2 l_1 0];
```

We know that omega is a symmetric matrix that has this form

$$\omega = (KK^T)^{-1} = \begin{vmatrix} a^2 & 0 & -u_0 a^2 \\ * & 1 & -v_0 \\ * & * & f_Y^2 + a^2 u_0^2 + v_0^2 \end{vmatrix}$$

This has only four unknowns, so only four equations are needed. Then omega is set up as a matrix of variables, so it can be found by the MATLAB method solve

```
syms a b c d;
omega = [a 0 b; 0 1 c; b c d];
```

Then the equations are set up

```
eqn_omega = [lx(1,:)*omega*v == 0, lx(2,:)*omega*v == 0];
eqn_linf = [h1.' * omega * h2 == 0, h1.' * omega * h1 == h2.' * omega * h2];
eqns = [eqn_omega, eqn_linf];
```

Then these equations are solved through the method solve and the parameters of omega, a, b, c, d, are found.

So the matrix omega can be reconstructed from these parameters found

```
W = solve(eqns, [a,b,c,d]);
IAC = double([W.a 0 W.b; 0 1 W.c; W.b W.c W.d]);
```

```
IAC =

   1.0e+06 *

    0.0000         0   -0.0002
         0    0.0000   -0.0010
   -0.0002   -0.0010    1.6099
```

Then, because the image of the absolute conic omega is such that

$$\omega = \mathbf{M}^{-T}\Omega_{\infty}\mathbf{M}^{-1} = (\mathbf{KR})^{-T}\Omega_{\infty}(\mathbf{KR})^{-1} = \mathbf{K}^{-T}\mathbf{K}^{-1}$$

Where R is the rotation matrix of the projection. Omega is independent from R and is just dependent on K. So it is possible to apply Cholesky to the inverse of omega to obtain K.

```
K = chol(inv(IAC))
```

```
K =

   1.96710794530133   0.647934128046524   0.000638551259399212
                  0   1.66562764029174    0.00104982720240096
                  0                  0    0.000788138251943832
```

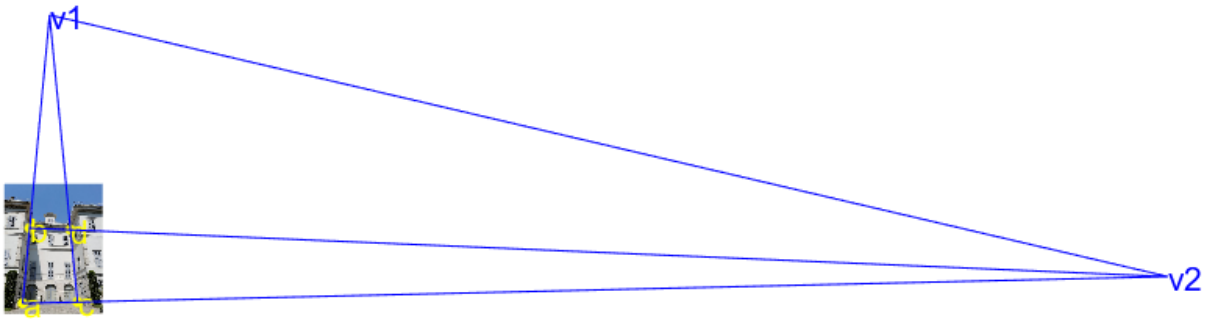
4. Reconstruction of vertical façade

In this phase it is reconstructed the facade 3, the vertical one. The script performing it is "Façade_3_Rectification.m".

Here since the calibration matrix has already been found, it is possible to use another method to reconstruct the vertical facade. In particular it is different how the image of the conic dual to the circular points is obtained

First the line at the infinity for the vertical facade is needed, so the four corners of the facade are taken by hand, and by crossing 2 pairs of parallel lines it is possible to retrieve 2 vanishing points. Then also the line at the infinity can be computed.

This is done in the same method as in the previous points.

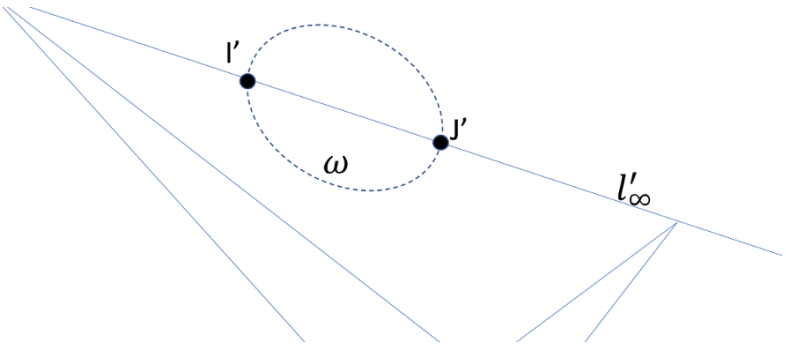


Now the calibration matrix is retrieved from previous point and the image of the absolute conic is obtained

```
K = [1.96710794530133          0.647934128046524      0.000638551259399212;
                           0           1.66562764029174       0.00104982720240096;
                           0                          0      0.000788138251943832];
% Compute IAC
IAC = inv(K'*K);
```

It is performed K^T*K since the chol method in MATLAB find K following this constraints instead of $K*K^T$. (This is exactly the inverse of the steps done in the previous point to find K).

Now it is needed to find the image of the circular points I' and J'. Since all the image of the circular points stays on the image of the absolute conic, the intersections between the image of the absolute conic and the image of the line at the infinity for that specific plane, are the image of the circular points.



So to retrieve the image of the circular points it is possible to intersect the image of the line at the infinity and the image of the absolute conic

First the two equations representing the line and the conic are built

```
syms 'x';
syms 'y';

eqn1 = [IAC(1,1)*x^2+2*IAC(1,2)*x*y+IAC(2,2)*y^2+2*IAC(1,3)*x+2*IAC(2,3)*y+IAC(3,3) == 0];
eqn2 = [linf(1)*x+linf(2)*y+linf(3) == 0];
```

Then solve is invoked to solve the equations with variables x and y, therefore it is possible to find I' and J'

```
eqns = [eqn1, eqn2];
X = solve(eqns, [x,y]);
Ii = double([X.x(1); X.y(1); 1]);
Ji = double([X.x(2); X.y(2); 1]);
```

Now that the image of the 2 circular points has been found, it is possible to find the rectifying matrix through the image of the conic dual to the circular points.

```
DCCPi = Ii*Ji.' + Ji*Ii.';
DCCPi = DCCPi ./ norm(DCCPi);
```

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

Now it is needed to push back the image of the circular points to their real coordinates [1, i, 0] and [1, -i, 0]. Since the real conic dual to the circular points is

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

It is possible to use SVD to retrieve the rectification matrix

$$\text{svd}(C_{\infty}^{*\prime}) = U \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{vmatrix} U^{T} = H_{rect}^{-1} C_{\infty}^{*} H_{rect}^{-T}$$

But in practice we don't obtain the diagonal matrix with 1s in the diagonal, this due to noise. So in practice it is used

$$\text{svd}(C_{\infty}^{*\prime}) = U \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & \varepsilon \end{bmatrix} U^{T}$$

Now the epsilon can be forced to 1 and s1 and s2 can be moved into U, such that

$$H_{rect} = (U \begin{bmatrix} \sqrt{s_1} & 0 & 0 \\ 0 & \sqrt{s_2} & 0 \\ 0 & 0 & 1 \end{bmatrix})^{-1} = \begin{bmatrix} \sqrt{s_1^{-1}} & 0 & 0 \\ 0 & \sqrt{s_2^{-1}} & 0 \\ 0 & 0 & 1 \end{bmatrix} U^{T}$$

So, in MATLAB firstly is computed the SVD of the image of the conic dual to the circular points. Then the value of D(3,3) is set to one (the epsilon in the previous image), and Hrect is obtained by U multiplied by the square root of D and then it is inverted.

```
[U,D,V] = svd(DCCPi)
D(3,3) = 1;
Hrect = U*sqrt(D);
Hrect = inv(Hrect)
```

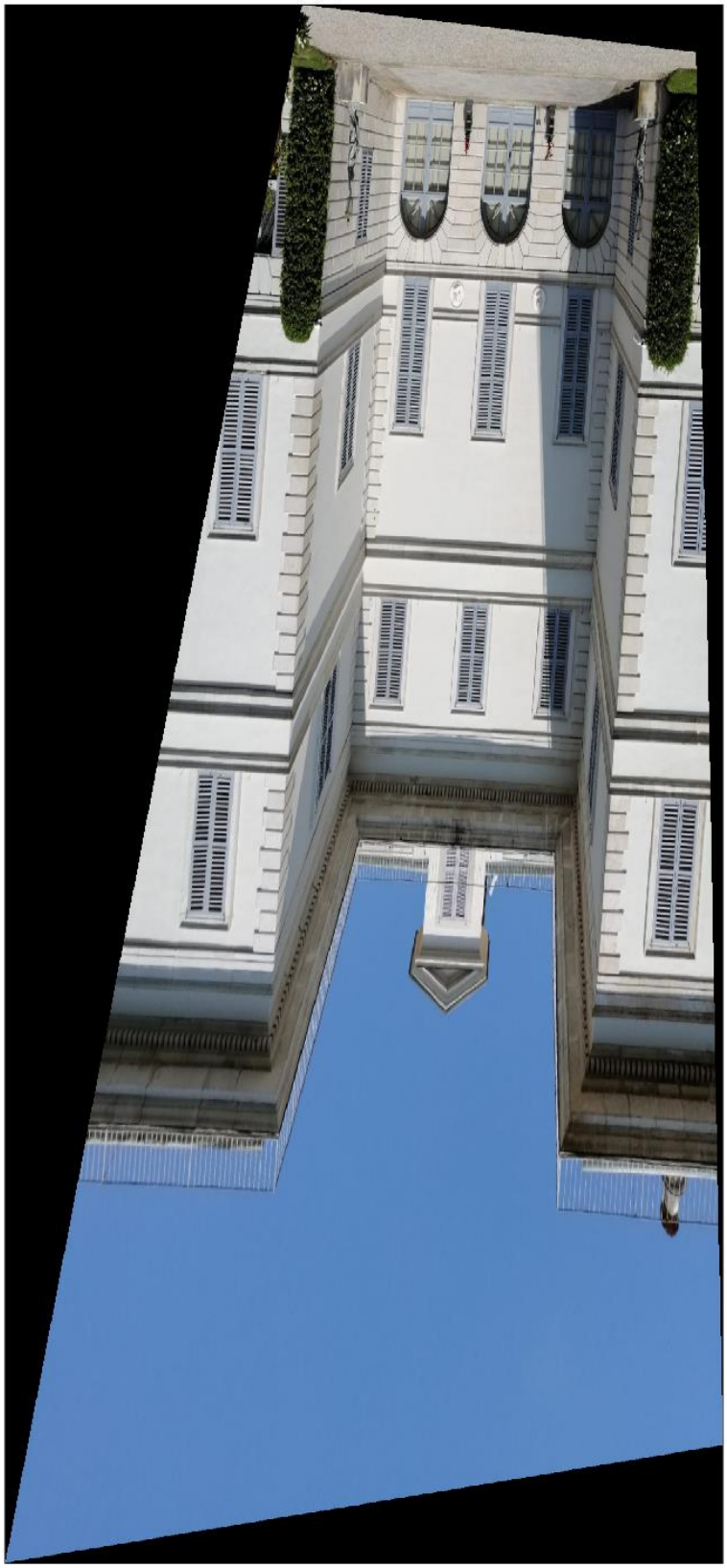
The obtained matrix Hrect is

```
Hrect =

        -0.998406312190869       -0.0564342775932574     -8.99441833227138e-05
         0.164160756552472          -2.9042503497163       0.00149959878972561
     -0.000118894013749576       0.000509625781154766         0.999999863072879
```

Finally, using this rectifying matrix, is obtained the image of the reconstructed facade (obtained, as always with projective2D and imwarp)

Facade 3 rectified



As it is seen, the vertical facade has now the shape of a rectangle (as it is in the reaity), and the proportion are as the real ones.

5. Localization

In this step the camera will be localized with respect to facade 3. This process is executed by the script “Localization.m”.

To estimate the camera position, first of all it is necessary to estimate an homography H, between some points in the planar object as reconstructed (in this case the facade 3) and their images. It is well known that to estimate an homography, 4 pairs of corresponding points are needed.

To do so the four corners from the reconstructed image were taken from the previous point.

```
a_real = [-167.738734663577 ,-2291.44104559145, 1]';
b_real = [-252.058376340953, -1120.61725401267, 1]';
c_real = [-559.452744837832, -2317.51399229086, 1]';
d_real = [-643.772386515208, -1146.69020071207, 1]';
```

Then it was taken the ratio between the 2 edges ac and ab.

```
l1 = distance(a_real,c_real);
l2 = distance(a_real,b_real);
aspect_ratio = l1/l2;
```

If we assume that the real length of segment ab is of 1 meter, we can obtain all the four corners as they are in reality. Starting from the point on the bottom left, which will have coordinates [0,0], we can take as height the length assumed, and as width length assumed divided by the aspect ratio calculated before.

```
LENGHT_AB = 1;

% Build the rectangle with the same aspect ratios
u1 = [0 0];
u2 = [0 LENGHT_AB];
u3 = [LENGHT_AB/aspect_ratio 0];
u4 = [LENGHT_AB/aspect_ratio LENGHT_AB];
```

So this are the points in the planar world. Now are needed the points taken from the original image.

```
a = [202, 1.3070e+03, 1]';
b = [279, 0.4840e+03, 1]';
c = [803, 1.2930e+03, 1]';
d = [726, 0.5030e+03, 1]';
```

Now it is used the function “fitgeotrans” of MATLAB, which given pairs of corresponding points returns the fitting homography H

```
H = fitgeotrans([u1; u2; u3; u4], [a(1:2).'; b(1:2).'; c(1:2).'; d(1:2).'], 'projective');
H = H.T.';
```

Now, it is possible to use this H and the calibration matrix K, to compute the pose of the vertical facade, relative to the camera

$$[i_\pi \quad j_\pi \quad o_\pi] = \mathbf{K}^{-1}H$$

The component relative to Z axis can be computed from i_π and j_π. Since it is a reference, the third vector must be orthogonal to the other 2. So can be estimated from the cross product of the other vectors.

$$(k_\pi = i_\pi \times j_\pi)$$

To obtain these vectors, matrix H is split into columns and matrix K from the calibration point is retrieved

```
h1 = H(:,1);
h2 = H(:,2);
h3 = H(:,3);

% Retrieve K from G2
K = [1.96710794530133         0.647934128046524      0.000638551259399212;
                          0          1.66562764029174       0.00104982720240096;
                          0                         0      0.000788138251943832];
```

To have obtain better results, it is computed a normalization factor lambda, and then i_π and j_π are calculated. As said before k_π is computed from the cross product.

```
lambda = 1 / norm(K \ h1);

% Compute i, j and k for the rotation matrix
i_pi = (K \ h1) * lambda;
j_pi = (K \ h2) * lambda;
k_pi = cross(i_pi,j_pi);
```

Up to this point, it is possible to build the rotation matrix R from these vectors. Since R must be true, but it can happen that due to noise, R is not orthogonal, it is performed an svd on R to solve this problem.

```
R = [i_pi, j_pi, k_pi];

% Use SVD to soppress noise and get an orthogonal matrix
[U, ~, V] = svd(R);
R = U * V';
```

Then it is computed o_π here called T, from the last column of H.

```
T = (K \ (lambda * h3));
```

Now, it is possible to construct the location matrix

$$\begin{bmatrix} i_\pi & j_\pi & i_\pi \times j_\pi & 0_\pi \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$

```
location_matrix = inv([R, T;
                       zeros(1,3), 1])
```

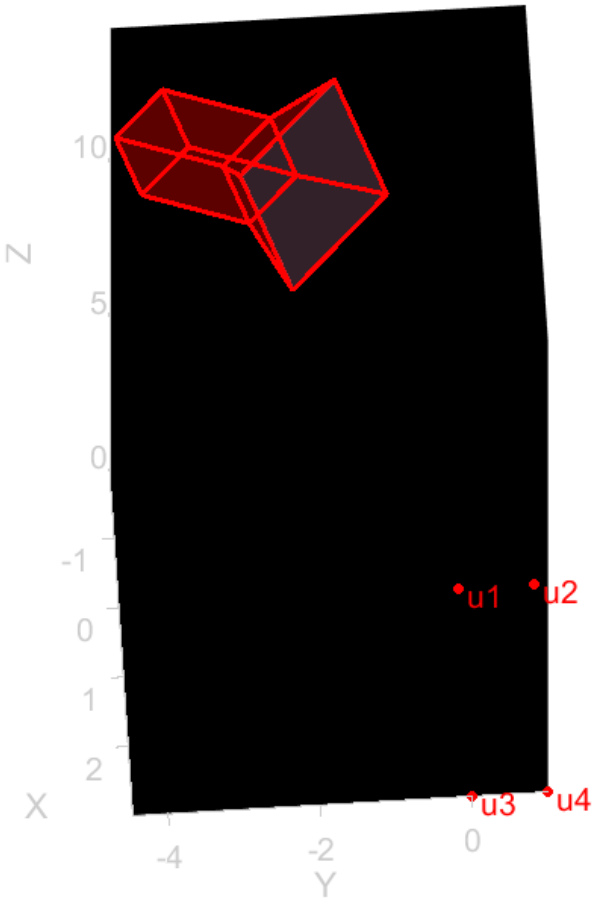
```
location_matrix =

    0.935180055399179    0.348407810293549   -0.063641666461863   -0.429871082990419
    0.287133356365642    -0.64062631105606    0.712146308875434   -3.29354333900031
     0.207346810069   -0.684258669882346   -0.699140453017169    13.4445312403979
                   0                    0                    0                   1
```

And from this matrix we obtain the camera rotation matrix and the camera position vector

```
camera_rotation = location_matrix(1:3,1:3);
camera_position = location_matrix(1:3, 4);
```

Finally, the camera and the points corresponding to the real facade are plotted, through MATLAB functions “plotCamera” and “pcshow”. Plot camera is able, given camera rotation position and size to plot the camera in a 3D space. Pcshow can plot a 3-D point cloud, given a matrix with the points.



This is the 3D image of the camere plotted with respect to the reconstructed vertical facade.