POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Homework Report

Image Analysis and Computer Vision

Christian Grasso

Student ID: 10652464
Academic Year: 2021-22

Contents

1 | Introduction

This is the report for the Image Analysis and Computer Vision Homework, A.Y. 2021-22. The assignment consisted in processing an image of Villa Melzi d'Eril in Bellagio, analyzing its geometry and performing 2D reconstruction.

1.1. Used Matlab functions

Throughout the Matlab program, various functions were used, including some custom functions. The full code for the used functions can be found in the `functions` folder in the delivered zip files, and in Appendix A.

- `segToLine(pts)` (as seen in class)
  Convert the endpoints of a line segment to a line in homogeneous coordinates (`pts = [x1 y1; x2 y2]`)
- `fitLine(XY)` (as seen in class)
  Least squares fit of a line to a set of points. `XY` is a `2xN` array of xy coordinates.
- `drawLineFamilies(title, nFamilies, nLinesPerFamily)`
  Ask the user to draw `nFamilies` families of `nLinesPerFamily` segments, and return a `nFamilies x 1` cell with lines in homogeneous coordinates.
- `drawLines(title_, nLines, color)`
  Ask the user to draw `nLinesPerFamily` segments, and return a `nLines x 3` vector with lines in homogeneous coordinates.
- `plotHCLine(line, range, nPoints, color)`
  Plot a line represented in homogeneous coordinates.
- `hpnorm(p)`
  Normalize a homogeneous coordinates vector representing a point.

- `buildHaff(lines)`
  Build an affine rectification matrix using the specified line families (`lines` is usually the output of `drawLineFamilies`)
- `buildRotationMatrix(angle)`
  Build a rotation matrix for the specified angle (in radians).

2 | F1 - Feature extraction

The objective is to find edges, corner features and straight lines in the image.

2.1. Edge detection

The **Canny** method can be used to detect edges in the image. The algorithm works by first applying a smoothing Gaussian filter to the image (to remove noise), then computing the gradient of the image, applying nonmax suppression and tracking edges via hysteresis thresholding.

Matlab already includes (in the Image Processing Toolbox) an implementation for this method in the `edge` function.

First, we need to convert the image to grayscale. This way, the resulting image matrix will contain `doubles` in the $[0, 1]$ interval, representing the "intensity" of each pixel.

```
img = im2double(imread('villa.png'));
imgGrayscale = rgb2gray(img);
```

Then we can simply use the `edge` function to apply the Canny method. A threshold can be specified to adjust the output image; in this case a value of `[0, 0.1]` seems to yield the best results.

```
edges = edge(imgGrayscale, 'canny', [0 0.1]);
```

The output image is fairly good, but it is a bit too noisy for the subsequent step of detecting straight lines. We can either try to reduce the threshold, which however seems to cut a lot of "good" lines from the output, or scale the image down to reduce detail:

```
resizeMult = 0.6;
newSize = resizeMult.*size(img, [1 2]);
imgGrayscaleRescaled = imresize(imgGrayscale, newSize, 'bilinear');
imgRescaled = imresize(img, newSize, 'bilinear');
```

(a) Canny at 100% scale (b) Canny at 50% scale

Figure 2.1: Output of Canny edge detection

2.2. Straight line detection

We can use the results of the previous step to detect straight lines in the image. This can be done using the **Hough transform**. A Hough transform of a datum (in this case a point on a detected edge) is a set of models compatible with the datum in the *model space*. For a point in the cartesian plane, the HT will be a sinusoid; collinear points will map to (almost) concurrent HTs.

The Hough transform is also included in Matlab:

```
[H,T,R] = hough(edges);
```

where `T` is θ, `R` is ρ and `H` is a matrix whose rows and columns correspond to ρ and θ values respectively.

We can then use the `houghpeaks` function to detect peaks, and pass the results to `houghlines` to extract lines.

```
P = houghpeaks(H, 100, 'threshold', 0.3*max(H(:)));
hlines = houghlines(edges, T, R, P, 'FillGap', 4, 'MinLength', 14);
```

These functions include various configuration options, namely:

- `numpeaks` is the number of peaks to detect (set to 100 here)
- `threshold` is the minimum value to be considered a peak. By default it is set to half the maximum value in the `H` matrix. For our image, we can get better results by lowering it to 30% of the max value.
- `FillGap` is, quoting the Matlab documentation, *the distance between two line segments associated with the same Hough transform bin*. In practice, this value controls the max distance between two detected segments for them to be merged into one segment.
- `MinLength` controls the minimum length for a segment. Setting this to a higher value allows to remove some bad segments in noisy areas of the image (e.g. the hedge).

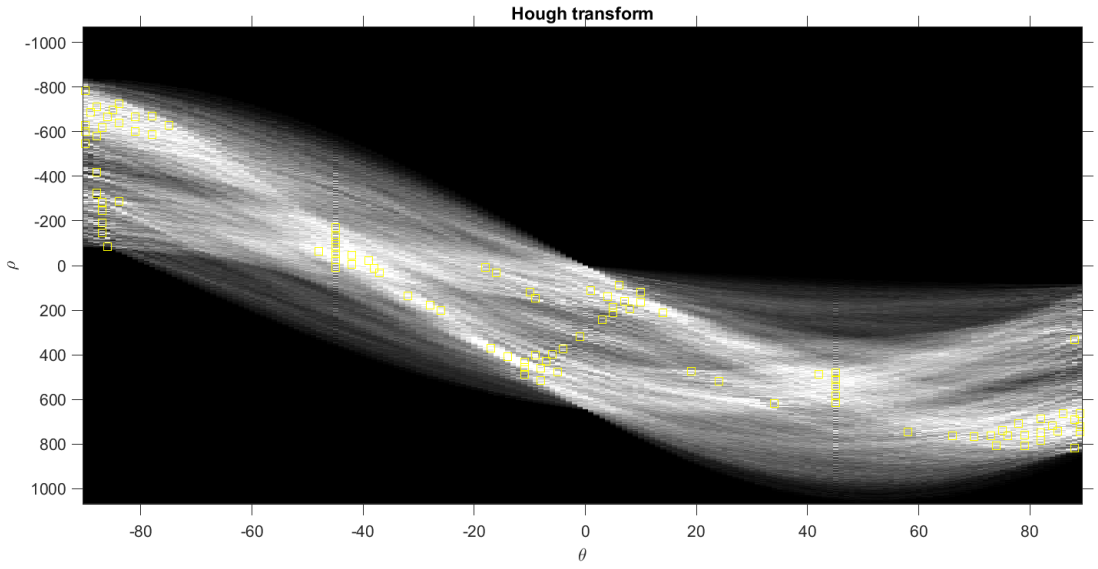


Figure 2.2: Hough Transform and detected peaks (yellow) for the image

The parameters detailed above can be adjusted based on whether we want a cleaner output or more/longer lines in the output.

(a) peaks=100, FillGaps=4, MinLength=14 (b) peaks=40, FillGaps=8, MinLength=10

Figure 2.3: Straight lines detected with the Hough Transform

2.3. Corner detection

We are going to use the **Harris-Stephens** corner detection algorithm. This algorithm is based on Moravac's corner detector, which works by splitting the image in small *patches* of pixels, and checking how much a patch differs to the surrounding patches. If significant changes are present in both directions, the patch being analyzed is possibly a corner.

Matlab implements this algorithm in the Computer Vision Toolbox, and we can easily apply it to our image:

```
points = detectHarrisFeatures(imgGrayscale);
figure; imshow(img); hold all;
plot(points.selectStrongest(5000));
```

The detection already works rather well without any adjustment, however not all corners are being detected and there are a lot of points in noisy areas like the hedge.

We can try to tackle this problem by increasing the contrast of the image, which should increase the differences for actual corners and push down the score of noisy areas. One way to do this is using the `histeq` function, which implements **histogram equalization**, a method that spreads the contrast values in the image between the minimum and maximum possible values (since our image is grayscale, this means the darkest pixels will be black and the brightest will be white).

This yields a way better result with a larger number of real corners being detected (e.g. in the railing) and fewer "wrong" points.

(a) No equalization (b) Histogram equalization applied

Figure 2.4: Harris-Stephens corner detection results

3 | G1 - 2D reconstruction of a horizontal section

The objective is to metric rectify (2D reconstruct) the horizontal section of the Villa, and to determine the ratio between the side facades (numbered 2 and 4) and the center facade (number 3). The calibration matrix is unknown at this point.

In order to compute the rectifiying homography H_rect, we need to compute the *image of the conic dual to the circular points* $C_\infty^{*\prime}$. One way to do this is using the so-called **stratified method**, which consists of first performing an **affine rectification** and then finding $C_\infty^{*\prime}$ through two more constraints (e.g. lines at a known angle).

3.1. Affine rectification

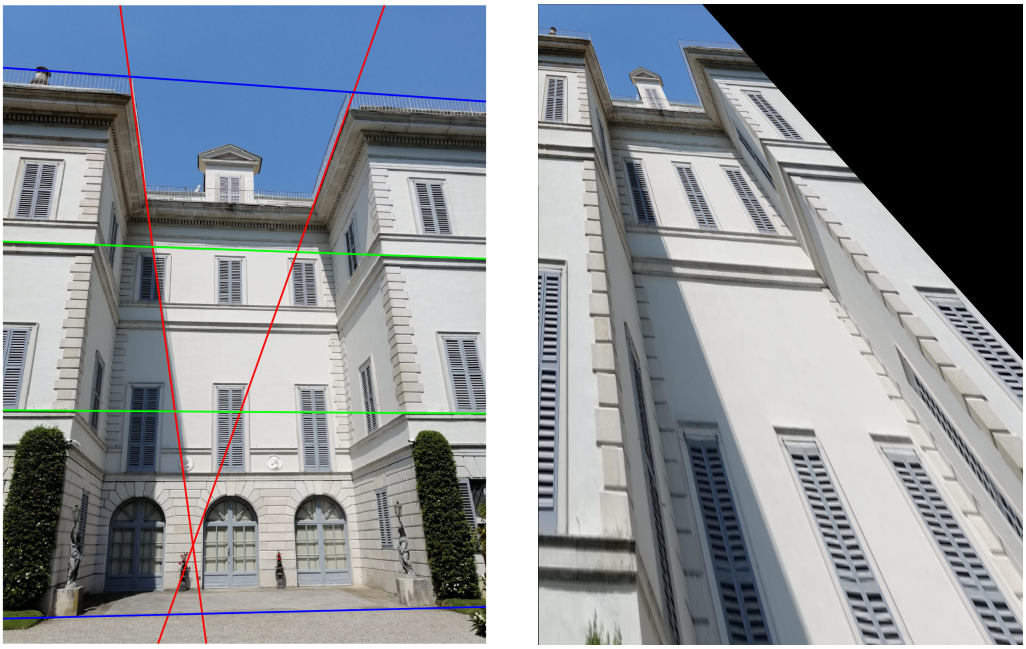
To perform affine rectification, we need at least two families of parallel lines, but we can use more to improve accuracy. In my case, three families were used. The lines are manually selected by the user on the image, as the lines detected in 2.2 were not accurate enough to obtain acceptable results.

```
figure;
hold all;
imshow(img);

nFamilies = 3;
parallelLines = drawLineFamilies('Draw parallel lines', nFamilies, 2);
```

We then build the affine rectification matrix using the `buildHaff` function, which also returns the image of the line at infinity. The function internally uses `fitLine`, which uses least squares to find the vanishing points of each direction.

```
[H_aff, imHLinf] = buildHaff(parallelLines);
```



(a) Lines used for affine rectification (b) Affine rectified image

Figure 3.1: Affine rectification of the Villa image

It is crucial to select the red lines in figure 3.1a, as we want those lines to be parallel in the output.

3.2. Metric rectification

Now that we have the affine rectified image, we need two more constraints to compute the image of $C_\infty^{*\prime}$. In particular, we can use two pairs of lines at known angles, remembering the equation:

$$\cos\theta = \frac{l'^T C_\infty^{*\prime} m'}{\sqrt{(l'^T C_\infty^{*\prime} l')\,(m'^T C_\infty^{*\prime} m')}}$$

Since the assignment specifies the **position of the sun**, we can use this information along with the shadows in the image for the first constraint, and use two orthogonal lines for the remaining unknown.

```
cosTheta = cos(atan(3.9));
orthogonalLines = drawLines('Draw orthogonal lines', 2, 'r');
angledLines = drawLines('Draw an horizontal line and the shadow', 2, 'g');
```

Finally, we can solve the system of equation with these two constraints to find the metric rectification matrix H_{metric} (or rather, its inverse, since we need to multiply it with H_{aff} and Matlab suggests to use the \ operator as it is faster):

```
% Solve the system using the selected lines
syms x1 x2;
imDCCP = [x1,x2,0;x2,1,0;0,0,0];
orthogonalConstraintEq = a.'*imDCCP*b == 0;
angledConstraintEq = cosTheta*sqrt((l.'*imDCCP*l)*(m.'*imDCCP*m)) ...
        == (l.'*imDCCP*m);
S = solve([orthogonalConstraintEq, angledConstraintEq], [x1, x2]);

% Note that, depending on the selected lines, 1 or two solutions
% could be returned. Here, the case with 1 solution is handled.
s1 = double(S.x1(1));
s2 = double(S.x2(1));

% Build the matrix via Cholesky decomposition
K = chol([s1, s2; s2, 1]);
H_metric_inv = [K.',[0;0];0,0,1];
```

Finally, we compute the rectification matrix and apply it to the image. The output size is scaled by a factor of 5 to show a larger portion of the image.

```
Hrect = H_metric_inv \ H_aff;
tform = projective2d(Hrect.');
imgRect = im2double(imwarp(img, tform, 'OutputView', imref2d(size(img)*5)));
figure; imshow(imgRect);
```

$$H_{rect} \approx \begin{pmatrix} 1.0051 & 0 & 0 \\ -0.4713 & 1.1045 & 0 \\ -1.2015 \times 10^{-5} & -8.2565 \times 10^{-4} & 1 \end{pmatrix}$$

(a) Lines used for metric rectification (b) Metric rectified image

Figure 3.2: Metric rectification lines and result

Even though the metric rectification does not show the full horizontal section, it is enough for our goal of calculating the ratio between facades.

3.3. Ratio between facades 2 and 3

To calculate the ratio, we need to find the length of each facade. This can be done by simply selecting the corners of each facade (so three points in total since they share one corner) and calculating the distance between the points.

For better precision, instead of selecting the points in the metric rectified image we can select them on the original image and multiply H_{rect} to get their images.

```
% Select three points
[x,y] = getpts();
% Convert to homogeneous coords and apply the transform
xyRect = Hrect * [x y [1;1;1]].';
% Normalize the transformed points (for accuracy and easier plotting)
xyRect = xyRect ./ xyRect(3, :);
```

(a) Points selected in the original image (b) Image of the points in the warped image

Figure 3.3: Points selected for the facade length calculation

We can then calculate the length (remember the points are normalized, so by subtracting them we effectively get rid of the w coordinate and consider them points in the cartesian plane) and print the ratio.

```
p2 = xyRect(:,1);
p23 = xyRect(:,2);
p3 = xyRect(:,3);
lengthF2 = norm(p23-p2);
lengthF3 = norm(p3-p23);
fprintf('Ratio of facade 2 over 3: %d\n', lengthF2 / lengthF3);
```

With the above points, the obtained ratio is $\approx 65\%$.

4 | G2 - Calibration

We now want to find the calibration matrix $\mathbf{K}$ containing the intrinsic parameters of the camera. From the assignment, we know the skew factor is null, thus there are four unknowns, so we will need four equations to find $\mathbf{K}$.

Since we already have H_{rect} from the previous step, we can use the direct method to calibrate from a rectified face plus orthogonal vanishing points. The equations are:

$$\begin{cases} \mathbf{l}'_{\infty} = \omega\, \mathbf{v} \\ h_1^T\, \omega\, h_2 = 0 \\ h_1^T\, \omega\, h_1 - h_2^T\, \omega\, h_2 = 0 \end{cases} \tag{4.1}$$

The Matlab code is pretty straightforward:

```
%% Get the vertical vanishing point
verticalLines = drawLines('Draw vertical parallel lines', 2, 'r');
vpV = hpnorm(cross(verticalLines(1,:), verticalLines(2,:)));
%% Find two more vanishing points
horizontalLines = drawLines('Draw two lines on the horizontal plane', 2, 'g');
vp1 = hpnorm(cross(horizontalLines(1,:), imHLinf));
vp2 = hpnorm(cross(horizontalLines(2,:), imHLinf));
%% Compute the IAC
Hrect_inv = inv(Hrect);
h1 = hpnorm(Hrect_inv(:,1));
h2 = hpnorm(Hrect_inv(:,2));

syms x1 x2 x3 x4;
x = [x1 0 x2; 0 1 x3; x2 x3 x4];
```

```
S = solve([
    vp1 * x * vpV.' == 0;
    vp2 * x * vpV.' == 0;
    h1.' * x * h2 == 0;
    h1.' * x * h1 == h2.' * x * h2
], [x1 x2 x3 x4]);
s1 = double(S.x1); s2 = double(S.x2);
s3 = double(S.x3); s4 = double(S.x4);
omega = [s1 0 s2; 0 1 s3; s2 s3 s4];
```

The result is the following (image size is normalized to 1):

$$\mathbf{K} = \begin{pmatrix} f_x & 0 & U_o \\ 0 & f_y & V_o \\ 0 & 0 & 1 \end{pmatrix} \approx \begin{pmatrix} 0.9160 & 0 & 0.4797 \\ 0 & 0.8099 & 0.4913 \\ 0 & 0 & 1 \end{pmatrix}$$

Camera intrinsic parameters:

- **Aspect ratio**: $\alpha = \frac{f_x}{f_y} \approx 1.1310$
- **Focal distance**: $f_x \approx 0.9160$, $f_y \approx 0.8099$
- **Principal point**: $(U_o, V_o) \approx (0.4797, 0.4913)$

5 | G3 - Reconstruction of a vertical facade

Now that we have the ω matrix, we can use it to rectify facade 3. By intersecting it with the line at the infinity of the vertical plane (which we can find using two families of parallel lines, as before) we obtain the image of **I** and **J** and compute the image of the dual conic. We can then use `svd` to retrieve the **U** matrix, which is the transpose of the rectifying homography. However, the following equation is used for improved accuracy:

$$H_{rect}^{-1} = U \begin{pmatrix} \sqrt{s1} & 0 & 0 \\ 0 & \sqrt{s2} & 0 \\ 0 & 0 & \epsilon \end{pmatrix}$$

Here are the most important parts of the code:

```
%% Find the intersection between the line at infinity and the IAC
syms x1 x2;
x = [x1 x2 1].';
S = solve([
    imVLinf.' * x == 0, ...
    x.' * omega * x == 0
], [x1,x2]);
s1 = double(S.x1);
s2 = double(S.x2);
%% Compute the rectification matrix for the vertical facade
I = [s1(1) s2(1) 1].'; J = [s1(2) s2(2) 1].';
imDCCP2 = I*(J.')+J*(I.');
imDCCP2 = imDCCP2 ./ norm(imDCCP2);
[U, S2, ~] = svd(imDCCP2);
S2(3,3) = 1;
Hrect2_inv = U * sqrt(S2);
```

If we try to display the image now, it will be upside down. We can try to fix this by using one of the lines we selected earlier, computing its image and calculating its rotation.

```
hLineRot = Hrect2_inv.' * parallelLines{2}(1,:).';
rotAngle = atan(-hLineRot(1)/hLineRot(2)) + pi;
Hrot = buildRotationMatrix(-rotAngle);
```

Figure 5.1: Rectified vertical facade

6 | G4 - Localization

The last part of the assignment asks for the relative pose between facade 3 and the camera reference. To accomplish this, we can use the **K** matrix and a planar surface: by fitting an homography that transforms the plane in the image to a new reference frame defined by us (which will be the camera reference frame), we can determine the position of the camera itself.

The new reference frame π can be identified with its versors $[i_\pi, j_\pi, k_\pi]$ and the origin o_π. To find these, we first need to identify the homography.

Our plane will be the bottom horizontal plane (6.1a). From G1 we know the ratio `ratio` between facades 2 and 3, which we can use to place these points in the new reference plane. Starting from the bottom left corner, points will be mapped like this:

```
points_c = [0 0; 0 ratio; 1 ratio; 1 0];
```

After selecting the points in the image (`points_img`), we use the `fitgeotrans` function to fit the homography:

```
tform_loc = fitgeotrans(points_c, points_img, 'projective');
H_loc = tform_loc.T;
```

The following relation holds:

$$[i_\pi, j_\pi, o_\pi] = \mathbf{K^{-1}} H_{loc}$$

so we use it to find all parameters (remembering $k_\pi = i_\pi \times j_\pi$):

```
c_ref = K \ H_loc;
c_ref = c_ref ./ c_ref(3,:);
R = [c_ref(:,1), c_ref(:,2), cross(c_ref(:,1), c_ref(:,2))];
[U, ~, V] = svd(R);
R = U * V';
```

Notice how each column of the `c_ref` matrix is normalized; this is to avoid numerical errors. The `svd` is needed because the **R** matrix is not exactly orthogonal, also due to numerical errors.

We can now decompose calculate the camera orientation and position like this:

```
camera_orientation = R.';
camera_pos = -R.' * c_ref(:,3);
```

The only thing left to do is solve for scale using the provided camera height of 1.5m.

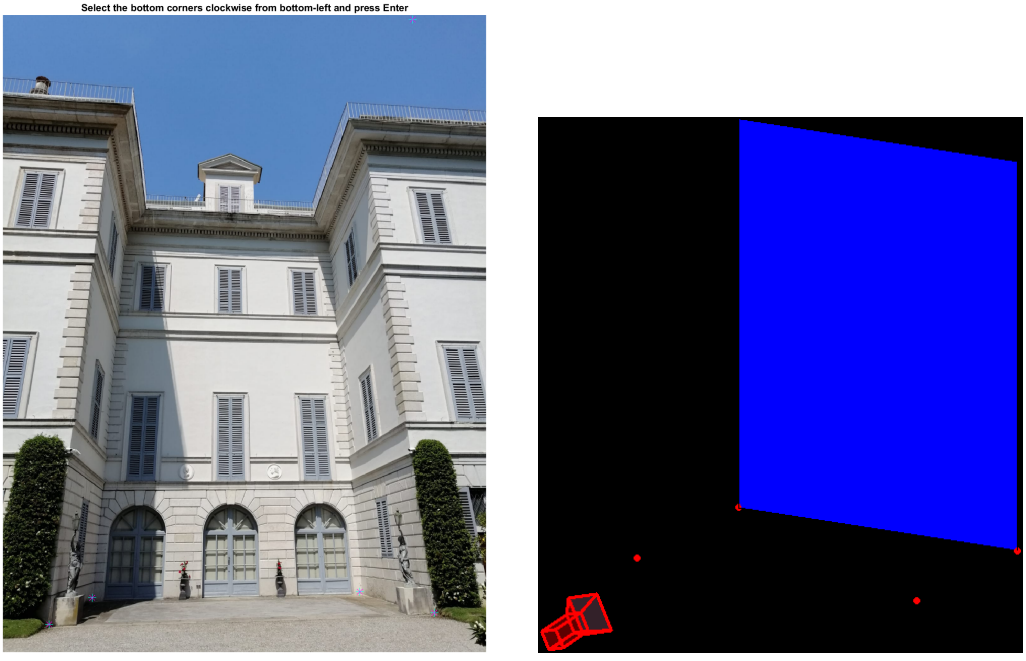
```
camera_mult = camera_pos(3) / 1.5;
camera_pos_m = camera_pos ./ camera_mult;
points_c_m = points_c ./ camera_mult;
```

The resulting locations of the points in the camera reference frame are $(0, 0)$, $(0, 10.25)$, $(15.77, 10.25)$, $(15.77, 0)$. Position of the camera itself is $(5.86, -17.8, 1.50)$.

The orientation is:

```
 0.9280    0.3304    0.1721
-0.2321    0.1513    0.9608
 0.2914   -0.9316    0.2171
```

To see the position and orientation of the camera, we can use the Matlab functions `plotCamera` and `pcshow` which render the camera itself and the points in a 3D space. I have also added a rectangle representing facade 3 to better understand the positioning of the camera (not actual height).



(a) Selected points (b) The relative camera pose

Figure 6.1: Selected lines and results of the localization process

A | Matlab functions

```
function [l] = segToLine(pts)
a = [pts(1,:)';1];
b = [pts(2,:)';1];
l = cross(a,b);
l = l./norm(l);
end

function [C, dist] = fitLine(XY)
  [rows,npts] = size(XY);
  if npts < 2
      error('Too few points to fit line');
  end
  if rows == 2
    XY = [XY; ones(1,npts)];
  end
  if npts == 2
    XY = [XY zeros(3,1)];
  end
  [XYn, T] = normalise2dpts(XY);
  [u d v] = svd(XYn',0);
  C = v(:,3);
  C = T'*C;
  if nargout==2
      dist = abs(C(1)*XY(1,:) + C(2)*XY(2,:) + C(3));
  end
end
```

```
function lines = drawLineFamilies(title, nFamilies, nLinesPerFamily)
    colors = 'rgbm';
    lines = cell(nFamilies, 1); % store lines
    for i = 1:nFamilies
        lines{i} = drawLines([title, ' - family ', num2str(i), ' of ', ...
                num2str(nFamilies)], nLinesPerFamily, colors(i));
    end
end

function lines = drawLines(title_, nLines, color)
    count = 1;
    lines = nan(nLines, 3);
    while(count <= nLines)
        figure(gcf);
        title([title_, ' - segment ', num2str(count), ' of ', num2str(nLines)]);
        segment = drawline('Color', color);
        lines(count, :) = hpnorm(segToLine(segment.Position));
        count = count + 1;
    end
end

% Normalize a point in homogeneous coordinates.
function p_norm = hpnorm(p)
    p_norm = p ./ p(3);
end
```

```
function [H, imLinf] = buildHaff(lines)
    V = nan(2, length(lines));
    for i = 1:length(lines)
        A = lines{i}(:,1:2);
        B = -lines{i}(:,3);
        V(:,i) = A\B;
    end
    imLinf = hpnorm(fitLine(V));
    H = [eye(2), zeros(2,1); imLinf(:).'];
end

function matrix = buildRotationMatrix(angle)
    ca = cos(angle);
    sa = sin(angle);
    matrix = [ca,-sa,0;sa,ca,0;0,0,1];
end
```

List of Figures