Computer Vision 252B Hw4

Hao-en Sung (wrangle1005@gmail.com)
Department of Computer Science
University of California, San Diego
San Diego, CA 92092

1 Programming: Automatic estimation of the planar projective transformation (110 points)

In this problem, I need to find out the projection matrix H, which maps 2D points in first image to 2D points in second image. It is noticeable that there is no translation between these two image points. Thus, one can only backproject these 2D image points to 2D scene points instead of 3D ones.

1.1 Feature detection (20 points)

This part is covered by the homework 1, where I convolute the original image with a designed kernel to calculate the gradient, detect all potential corners by solving a gradient matrix problem with window size 7, utilize *Non-maximum Suppression* with window size 7 to filter out too closed corner candidates, and filter out corners whose λ is below *lambda_threshold*$= 2200$. At the end, I re-calculate the center of those corners as final results.

There are 624 features detected in the first image and 649 features found in the second image. The detected corners for both images are shown in Fig. 1.

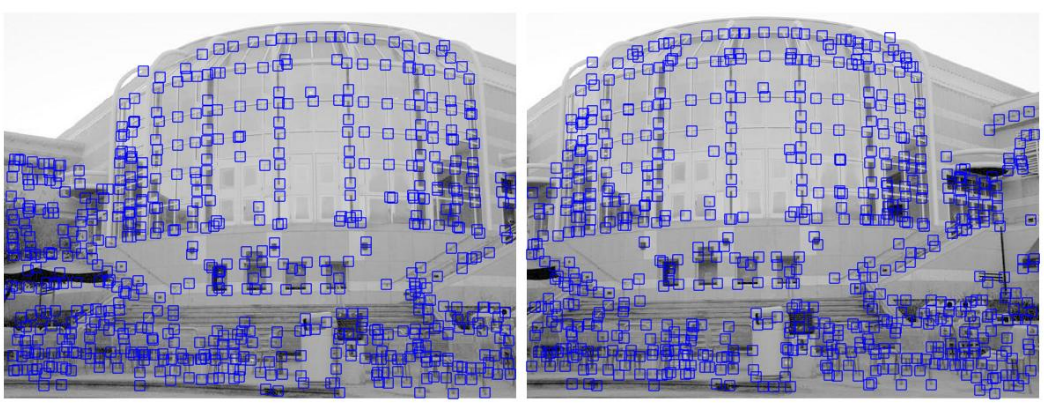

Figure 1: Picture for Detected Corners

1.2 Feature matching (15 points)

This part is also covered by homework 1, where I calculate the correlation coefficient between all pairs of corner window in image 1 and image 2, and iteratively choose the remaining pairs with largest correlation coefficient value whose absolute distance of x and y does not exceed 100. I use *similarity_threshold*$= 0.8$ and *distance_threshold*$= 0.7$ to reject those unqualified matches.

At the end, there are 229 matched feature points found between image 1 and image 2, which are shown in Fig. 2.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

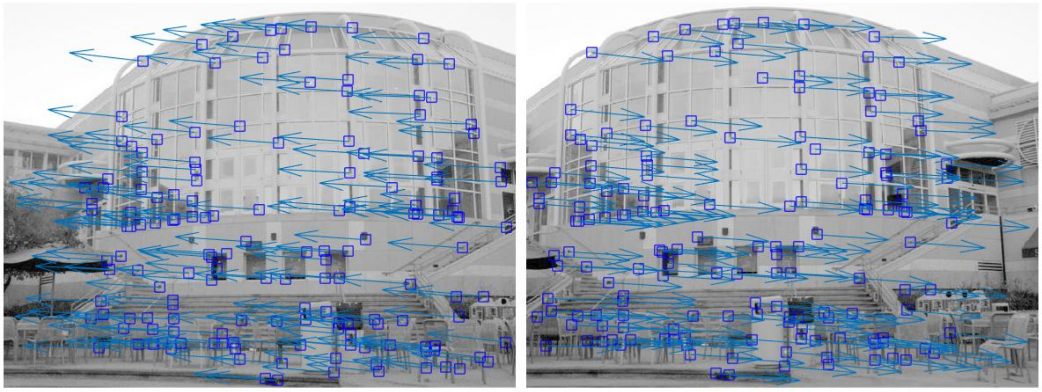

Figure 2: Picture for Matched Corners

1.3 Outlier rejection (15 points)

It is similar to what I have done for last homework, except I neither need to apply *Finsterwalder* algorithm to find out multiple solutions of 3D camera coordinate nor use *Umeyama* algorithm to find out both rotation matrix R and translation matrix t this time. Instead, I need to find out two projection matrices H' and H'', which transform points in image 1 and points in image 2 respectively to a basis $[e_1, e_2, e_3, e_4]$, where $e_1 = [0, 0, 1]^\intercal$, $e_2 = [0, 1, 0]^\intercal$, $e_3 = [1, 0, 0]^\intercal$, and $e_4 = [1, 1, 1]^\intercal$.

Take H' as an example, I first need to find out λ_1, λ_2, and λ_3, where

$$[x'_1 \quad x'_2 \quad x'_3] \cdot \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = x'_4.$$

Later, I can solve H' with equation

$$[e_1 \quad e_2 \quad e_3 \quad e_4] = H' \cdot [\lambda_1 x'_1 \quad \lambda_2 x'_2 \quad \lambda_3 x'_3 \quad x'_4] \, .$$

After retrieve H' and H'', I can calculate $H = H'' \setminus H'$ and use it to calculate δ_{x_i}.

To be more detailed, I first define ϵ_i and J as follows.

$$\epsilon_i = \begin{bmatrix} 0^\intercal & -x_i'^\intercal & y_i'' \cdot x_i'^\intercal \\ x_i'^\intercal & 0^\intercal & -x_i'' \cdot x_i'^\intercal \end{bmatrix} \cdot h,$$
$$J = \begin{bmatrix} -h_{2,1} + y_i'' h_{3,1} & -h_{2,2} + y_i'' h_{3,2} & 0 & x_i' h_{3,1} + y_i' h_{3,2} + h_{3,3} \\ h_{1,1} - x_i'' h_{3,1} & h_{2,2} - x_i'' h_{3,2} & -(x_i' h_{3,1} + y_i' h_{3,2} + h_{3,3}) & 0 \end{bmatrix},$$

Then, I can solve λ and δ_x with ϵ_i and J as below:

$$\begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{bmatrix} = (J \cdot J^\intercal) \setminus \left(- \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix} \right),$$
$$\delta_{x_i} = J^\intercal \lambda_i.$$

At the end, I use $\|\delta_{x_i}\|^2$ as the error and $\min(\|\delta_{x_i}\|^2, chi2inv(0.95, 2))$ as cost for point i.

The way to dynamically determine the number of maximum iterations are covered in homework 3 report. Here, I use the same settings that $p = 0.99$, $\alpha = 0.95$, and $\sigma^2 = 1$, and my algorithm finds out the inliers in *MAX_ITERATIONS*$= 5.7144161206$.

As the result of mSAC algorithm, it reduces matched corner pairs from 229 to 188 and achieve Rooted Mean Squared Error (RMSE) 40.8908482534, whose result is shown as Fig. 3.

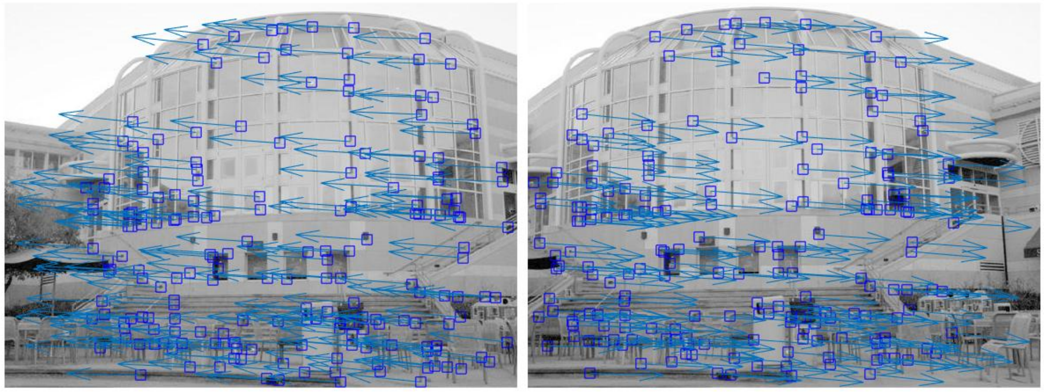

Figure 3: Picture for Robust Matched Corners

1.4 Linear estimation (15 points)

Within Direct Linear Transformation (DTL) algorithm, I firstly normalize points in both image 1 and image 2. After that, I directly solve a linear algebra problem

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_n \end{bmatrix} \cdot h = 0,$$

where

$$\begin{aligned} A_i &= \left([x_i']^{\perp} \otimes x^{\intercal}\right) \\ &= \begin{bmatrix} 0^{\intercal} & -x_i^{\intercal} & y_i' x_i^{\intercal} \\ x_i^{\intercal} & 0^{\intercal} & -x_i' x_i^{\intercal} \end{bmatrix}, \\ h &= \text{vec}(H), \end{aligned}$$

and return h as my linear estimation of H (or denoted as H_{DLT}).

My normalized H_{DLT} matrix can be expressed as

$$\begin{bmatrix} 0.0110060833521371 & -1.96489851185687e-05 & -0.984702563743917 \\ 0.000315728416207698 & 0.010723009347101 & -0.17325999820574 \\ 1.22455760910792e-06 & 7.76469095977639e-08 & 0.0102770183258898 \end{bmatrix},$$

and it successively reduces the RMSE from 40.6765940031 to 0.5941047757, which is a very significant improvement.

1.5 Nonlinear estimation (45 points)

In this part, I first use Sampson correction (details are mentioned in problem 3) to initialize scene points, where corrected point $\hat{x_i}$ can be express as

$$\hat{x_i} = \hat{x_i'} = x_i' + \delta_{x'},$$

where

$$\delta_x = \begin{bmatrix} \delta_{x'} \\ \delta_{x''} \end{bmatrix}.$$

After that, I fix projection matrix from scene plane to image 1, i.e. H', as identity matrix and iteratively update projection matrix from scene plane to image 2, i.e. H'', as well as scene points. To do so, I regard parameterized H'' and parameterized 2D homogeneous scene points x as parameters, 2D homogeneous points in both image 1 and image 2 as measurements, and solve an *Augumented Normal Equations* to find out the updates for both H'' and x.

One easiest way to obtain the *Jacobian* matrix J for this problem is to consider J as a $4n \times (2n+8)$ huge matrix, which is shown as follows.

$$\begin{bmatrix} 0 & \begin{bmatrix} B_1' & & 0 \\ & \ddots & \\ 0 & & B_n' \end{bmatrix} \\ \begin{bmatrix} A_1'' \\ \vdots \\ A_n'' \end{bmatrix} & \begin{bmatrix} B_1'' & & 0 \\ & \ddots & \\ 0 & & B_n'' \end{bmatrix} \end{bmatrix},$$

where

$$A_i'' = \frac{\partial \hat{x_i}''}{\partial \hat{h}''} = \frac{\partial \hat{x_i}''}{\partial \bar{h}''} \cdot \frac{\partial \bar{h}''}{\partial \hat{h}''},$$

$$B_i' = \frac{\partial \hat{x_i}'}{\partial \hat{x_i}} = \frac{\partial \hat{x_i}'}{\partial \bar{x_i}} \cdot \frac{\partial \bar{x_i}}{\partial \hat{x_i}},$$

$$B_i'' = \frac{\partial \hat{x_i}''}{\partial \hat{x_i}} = \frac{\partial \hat{x_i}''}{\partial \bar{x_i}} \cdot \frac{\partial \bar{x_i}}{\partial \hat{x_i}},$$

$\hat{h_i}$ is the updated projection matrix in vector form,

$\bar{h_i}$ is the updated parameterized projection matrix in vector form,

$\hat{x_i}$ is the updated 2D inhomogeneous scene points,

$\bar{x_i}$ is the updated 2D parameterized scene points,

$\hat{x_i}'$ is the estimated 2D inhomogeneous points in image 1,

$\hat{x_i}''$ is the estimated 2D inhomogeneous points in image 2.

With *Jacobian* matrix J, I then can calculate the update for parameters δ by solving a *Augmented Normal Equation* as follows.

$$(J^\intercal \Sigma_x^{-1} J + \lambda I) \cdot \delta = J^\intercal \Sigma_x^{-1} \epsilon,$$

where

$$\Sigma_x = \begin{bmatrix} \begin{bmatrix} \sqrt{\frac{2}{\mathrm{var}(x_i')}} & & \\ & \ddots & \\ & & \sqrt{\frac{2}{\mathrm{var}(x_i')}} \end{bmatrix} & 0 \\ 0 & \begin{bmatrix} \sqrt{\frac{2}{\mathrm{var}(x_i'')}} & & \\ & \ddots & \\ & & \sqrt{\frac{2}{\mathrm{var}(x_i'')}} \end{bmatrix} \end{bmatrix}, \text{ is the covariance matrix} m$$

$$\epsilon = \begin{bmatrix} x_1' - \hat{x_1}' \\ \cdots \\ x_n' - \hat{x_n}' \\ x_1'' - \hat{x_1}'' \\ \cdots \\ x_n'' - \hat{x_n}'' \end{bmatrix}, \text{ is the difference between ground truth points and estimated points,}$$

λ is a dynamic parameter, which is set as a step size.

After solving δ, I will update parameters H'' and x and perform the estimation and update for next round until the cost $\epsilon^\intercal \Sigma_x^{-1} \epsilon$ converges to a minimum. Then, I will return my non-linear estimation — projection matrix H (or said H_{LM}).

My normalized H_{LM} matrix can be expressed as

$$\begin{bmatrix} 0.0110042762280444 & -2.0915189925756e-05 & -0.984695244436302 \\ 0.0.000316277759170078 & 0.0107183324766052 & -0.173302322490435 \\ 1.22883426305626e-06 & 7.71365554271593e-08 & 0.0102714841667479 \end{bmatrix},$$

and it successively reduces the RMSE from 0.5941047757 to 0.5935540214, which is a very significant improvement.

My implementation converges in three rounds and the error log can be shown as follows.

$$\epsilon^{\mathsf{T}}\Sigma_x^{-1}\epsilon = [33.6617149584009 \quad 33.5976977320829 \quad 33.5976974848527]$$

.

Summary

From mSAC solving for P4P problem to linear estimation and non-linear estimation of projection matrix, RMSE reduces dramatically. On the other hand, one can tell that non-linear estimation further minimizes the geometric error when compared with linear estimation.

Appendix

Problem 1

Code Listing 1: Feature Detection

```
function mat = featureDetect(I, lambda_threshold, nw)

%% Parameters
hw = int32(floor(nw/2));
[n, m] = size(I);

%% Convolutional Kernel
k = [-1; 8; 0; -8; 1] / 12;

%% Calculate Gradient
Gx = conv2(double(I), k', 'same');
Gy = conv2(double(I), k, 'same');

%% Precalculate Squared Gradient
Gxx = Gx .* Gx;
Gxy = Gx .* Gy;
Gyy = Gy .* Gy;

%% Corner Detection
em = zeros(n, m);
for r = 1+nw:n-nw
    for c = 1+nw:m-nw
        vxx = sum(sum(Gxx(max(r-hw, 1):min(r+hw,n), ...
            max(c-hw, 1):min(c+hw,m))));
        vyy = sum(sum(Gyy(max(r-hw, 1):min(r+hw,n), ...
            max(c-hw, 1):min(c+hw,m))));
        vxy = sum(sum(Gxy(max(r-hw, 1):min(r+hw,n), ...
            max(c-hw, 1):min(c+hw,m))));
        ATA = [vxx, vxy; vxy, vyy];
        em(r, c) = (trace(ATA) - sqrt(trace(ATA)^2 - 4*det(ATA))) / 2;
    end
end

%% Non-maximum Suppression
mat = [];
for r = 1+nw:n-nw
    for c = 1+nw:m-nw
        vmax = max(max(em(max(r-hw, 1):min(r+hw,n), ...
            max(c-hw, 1):min(c+hw,m))));
        if em(r, c) == vmax && vmax > lambda_threshold
            mat = [mat; [c, r]];
        end
    end
```

```
end

%% Find Real Corners
[idx_x, idx_y] = meshgrid(1:m, 1:n);
xGxx = idx_x .* Gxx;
xGxy = idx_x .* Gxy;
yGxy = idx_y .* Gxy;
yGyy = idx_y .* Gyy;

for i = 1:size(mat,1)
    c = mat(i, 1);
    r = mat(i, 2);
    vxx = sum(sum(Gxx(max(r-hw, 1):min(r+hw,n), ...
        max(c-hw, 1):min(c+hw,m))));
    vyy = sum(sum(Gyy(max(r-hw, 1):min(r+hw,n), ...
        max(c-hw, 1):min(c+hw,m))));
    vxy = sum(sum(Gxy(max(r-hw, 1):min(r+hw,n), ...
        max(c-hw, 1):min(c+hw,m))));
    xVxx = sum(sum(xGxx(max(r-hw, 1):min(r+hw,n), ...
        max(c-hw, 1):min(c+hw,m))));
    xVxy = sum(sum(xGxy(max(r-hw, 1):min(r+hw,n), ...
        max(c-hw, 1):min(c+hw,m))));
    yVxy = sum(sum(yGxy(max(r-hw, 1):min(r+hw,n), ...
        max(c-hw, 1):min(c+hw,m))));
    yVyy = sum(sum(yGyy(max(r-hw, 1):min(r+hw,n), ...
        max(c-hw, 1):min(c+hw,m))));
    A = [vxx, vxy; vxy, vyy];
    b = [xVxx + yVxy; xVxy + yVyy];
    mat(i, :) = (A \ b)';
end
```

Problem 2

Code Listing 2: Feature Match

```
function [lx, rx] = featureMatch(preI, nxtI, pref, nxtf, nw)

%% Parameters
hw = floor(nw/2);
[n,m] = size(preI);
similarity_threshold = 0.8;
dist_threshold = 0.7;

preI = double(preI);
nxtI = double(nxtI);

%% Create Correlations
corrw = zeros(size(pref,1), size(nxtf,1));
for i = 1:size(pref,1)
    for j = 1:size(nxtf,1)
        % check proximality
        if abs(pref(i,1) - nxtf(j,1)) > 100 ...
                || abs(pref(i,2) - nxtf(j,2)) > 100 ...
            continue
        end

        [px, py] = ...
            meshgrid(max(pref(i,1)-hw,1):min(pref(i,1)+hw,m), ...
                     max(pref(i,2)-hw,1):min(pref(i,2)+hw,n));
        [nx, ny] = ...
            meshgrid(max(nxtf(j,1)-hw,1):min(nxtf(j,1)+hw,m), ...
                     max(nxtf(j,2)-hw,1):min(nxtf(j,2)+hw,n));

        prew = interp2(preI, px, py);
```

```
        nxtw = interp2(nxtI, nx, ny);
        corrw(i,j) = corr2(prew,nxtw);
    end
end

%% Find Largest Element Iteratively
lx = [];
rx = [];
while true
    [maxv,maxi] = max(corrw(:));
    % stop while the maximum one is not large enough
    if maxv < similarity_threshold
        break
    end

    % get the window corrdinate
    [r,c] = ind2sub(size(corrw),maxi);
    corrw(r,c) = -1;

    % get the potential two window corrdinates
    [nrv,~] = max(corrw(r,:));
    [ncv,~] = max(corrw(:,c));

    % stack them into arrays
    if nrv > ncv
        if (1-maxv) < (1-nrv) * dist_threshold
            lx = [lx; [pref(r,1), pref(r,2)]];
            rx = [rx; [nxtf(c,1), nxtf(c,2)]];
        end
    else
        if (1-maxv) < (1-ncv) * dist_threshold
            lx = [lx; [pref(r,1), pref(r,2)]];
            rx = [rx; [nxtf(c,1), nxtf(c,2)]];
        end
    end

    % reset to -1
    for j = 1:size(nxtf,1)
        corrw(r,j) = -1;
    end
    for i = 1:size(pref,1)
        corrw(i,c) = -1;
    end
end
```

Problem 3

Code Listing 3: mSAC Algorithm

```
% lx, rx: 2D homogenerous points
function [best_H, best_bmap, MAX_TRIALS, MIN_COST] = mSAC(lx, rx)
    assert(size(lx,1) == size(rx,1));
    n = size(lx,1);

    % Parameters
    MIN_COST = Inf;
    MAX_TRIALS = Inf;
    THRESHOLD = 1;
    TOLERANCE = chi2inv(0.95,2);
    PROBABILITY = 0.99;

    % Initialize return values
    best_H = zeros(3,3);
    best_bmap = false(n,1);
```

```
    trials = 0;
    s = RandStream('mt19937ar','Seed',1);
    while trials < MAX_TRIALS && MIN_COST > THRESHOLD
        trials = trials + 1;

        % Random 4 numbers
        idx = randperm(s, n);
        idx = idx(1:4);

        H1 = findH(lx(idx,:));
        H2 = findH(rx(idx,:));
        H = inv(H2) * H1;

        % Calculate delta
        delta = calDelta(lx, rx, H);

        % Calculate error
        error = calError(delta);

        % Calculate cost
        cost = calCost(error, TOLERANCE);

        if cost < MIN_COST
            MIN_COST = cost;
            best_H = H;
            best_bmap = error < TOLERANCE;
            best_n_in = sum(best_bmap);

            w = best_n_in / n;
            MAX_TRIALS = log(1-PROBABILITY) / log(1-w^4);
        end
    end
end

function H = findH(x)
    assert(size(x,1) == 4);
    A = x(1:3,:)';
    b = x(4,:)';
    lambda = A \ b;

    lambda_x = [lambda(1,1)*x(1,:)', lambda(2,1)*x(2,:)', ...
     lambda(3,1)*x(3,:)', 1*x(4,:)'];
    e = [[1,0,0]', [0,1,0]', [0,0,1]', [1,1,1]'];
    H = e / lambda_x;
end

function delta = calDelta(lx, rx, H)
    n = size(lx,1);
    vH = vector(H);
    delta = zeros(n,4);
    for i = 1:n
        Ai = [zeros(1,3), -lx(i,:), rx(i,2)*lx(i,:); ...
              lx(i,:), zeros(1,3), -rx(i,1)*lx(i,:)];
        ex = Ai * vH;
        J = [-H(2,1)+rx(i,2)*H(3,1), -H(2,2)+rx(i,2)*H(3,2), ...
             0, lx(i,1)*H(3,1)+lx(i,2)*H(3,2)+H(3,3); ...
             H(1,1)-rx(i,1)*H(3,1), H(1,2)-rx(i,1)*H(3,2), ...
             -(lx(i,1)*H(3,1)+lx(i,2)*H(3,2)+H(3,3)), 0];
        lambda = (J*J') \ (-ex);
        delta(i,:) = (J' * lambda)';
    end
end

function error = calError(delta)
```

```
    n = size(delta,1);
    error = zeros(n,1);
    for i = 1:n
        error(i,1) = delta(i,:) * delta(i,:)';
    end
end

function cost = calCost(error, TOLERANCE)
    cost = sum(min(error, TOLERANCE));
end

function vx = vector(x)
    vx = x';
    vx = vx(:);
end
```

Problem 4

Code Listing 4: Direct Linear Transformation

```
% lx, rx: 2D homogenerous points
function H = DLT_nc(lx, rx)
    assert(size(lx,1) == size(rx,1));
    n = size(lx,1);

    % Data Normalization
    lx_mean = mean(lx);
    lx_var = var(lx);
    lx_s = sqrt(2 / sum(lx_var));
    T = [lx_s, 0, -lx_mean(1)*lx_s;
         0, lx_s, -lx_mean(2)*lx_s;
         0, 0, 1];
    lx = lx * T';

    rx_mean = mean(rx);
    rx_var = var(rx);
    rx_s = sqrt(2 / sum(rx_var));
    U = [rx_s, 0, -rx_mean(1)*rx_s;
         0, rx_s, -rx_mean(2)*rx_s;
         0, 0, 1];
    rx = rx * U';

    % Left Null Space of H
    A = zeros(2*n, 9);
    for i = 1:n
        v = [rx(i,1) + sign(rx(i,1))*norm(rx(i,:)), ...
             rx(i,2), rx(i,3)]';
        Hv = eye(3) - 2 * (v * v') / (v' * v);
        A(2*i-1,:) = [Hv(2,1)*lx(i,:), Hv(2,2)*lx(i,:), ...
                      Hv(2,3)*lx(i,:)];
        A(2*i  ,:) = [Hv(3,1)*lx(i,:), Hv(3,2)*lx(i,:), ...
                      Hv(3,3)*lx(i,:)];
    end

    % Solve for P
    [~, ~, V] = svd(A, 'econ');
    H = V(:,end);
    H = reshape(H, 3, 3)';

    % Data Denormalization
    H = U \ H * T;
end
```

Problem 5

Code Listing 5: Levenberg–Marquardt Algorithm

```
% lx, rx: 2D homogenerous points
function [H, log] = LM_nc(lx, rx, H)
    assert(size(lx,1) == size(rx,1));
    n = size(lx,1);

    % Data Normalization
    lx_mean = mean(lx);
    lx_var = var(lx);
    lx_s = sqrt(2 / sum(lx_var));
    T = [lx_s, 0, -lx_mean(1)*lx_s;
         0, lx_s, -lx_mean(2)*lx_s;
         0, 0, 1];
    lx = lx * T';

    rx_mean = mean(rx);
    rx_var = var(rx);
    rx_s = sqrt(2 / sum(rx_var));
    U = [rx_s, 0, -rx_mean(1)*rx_s;
         0, rx_s, -rx_mean(2)*rx_s;
         0, 0, 1];
    rx = rx * U';
    H = U * H / T;

    % Scene Points Initialization
    delta = calDelta(lx, rx, H);
    sx = lx;
    sx(:,1:2) = sx(:,1:2) + delta(:,1:2);

    % Covariance Matrix
    Z = diag([repmat(lx_s^2, 1, 2*n), ...
        repmat(rx_s^2, 1, 2*n)]);

    % Initialization
    lambda = 0.001;
    perr = 1000000;

    [n_lx, n_rx] = estimate(sx, H);
    ex = calEpsilon(lx, rx, n_lx, n_rx);
    err = ex'*inv(Z)*ex;

    % Error Log
    log = err;

    % Parameterize H and sx
    vH = vector(H);
    pH = parameterize(vH);
    px = zeros(n,2);
    for i = 1:n
        px(i,:) = parameterize(sx(i,:));
    end

    % Angle Normalize pH and px
    if norm(pH) > pi
        pH = (1 - 2*pi/norm(pH) * ceil((norm(pH)-pi)/(2*pi))) * pH;
    end
    for i = 1:n
        if norm(px(i,:)) > pi
            px(i,:) = (1 - 2*pi/norm(px(i,:)) * ...
                ceil((norm(px(i,:))-pi)/(2*pi))) * px(i,:);
        end
    end
```

```matlab
% Deparameterize pH and px
vH = deparameterize(pH);
H = reshape(vH, 3, 3)';
for i = 1:n
    sx(i,:) = deparameterize(px(i,:)')';
end

while abs(perr-err) > 0.0001
    % Estimate Homogeneous 2D Points in
    % Image 1 and 2 from Scene Points
    [n_lx, n_rx] = estimate(sx, H);

    % Calculate J
    J = zeros(4*n,8+2*n);

    % Fill up J with A_i''
    partial_hh = [-0.5 * vH(2:end)'; ...
                  _sinc(norm(pH)/2)/2 * eye(8) + ...
                  1/(4*norm(pH)) * _dsinc(norm(pH)/2) * (pH*pH')];
    for i = 1:n
        partial_x2h = 1/n_rx(i,3) * ...
            [sx(i,:), zeros(1,3), -n_rx(i,1)/n_rx(i,3)*sx(i,:);
             zeros(1,3), sx(i,:), -n_rx(i,2)/n_rx(i,3)*sx(i,:)];
        J(2*n+2*i-1:2*n+2*i,1:8) = partial_x2h * partial_hh;
    end

    % Fill up J with B_i'
    for i = 1:n
        partial_x1x = 1/n_lx(i,3) * ...
            [1, 0, -n_lx(i,1)/n_lx(i,3);
             0, 1, -n_lx(i,2)/n_lx(i,3)];
        partial_xx = [-0.5 * sx(i,2:end); ...
              _sinc(norm(px(i,:))/2)/2 * eye(2) ...
              + 1/(4*norm(px(i,:))) * _dsinc(norm(px(i,:))/2) ...
              * (px(i,:)'*px(i,:))];
        J(2*i-1:2*i,8+2*i-1:8+2*i) = partial_x1x * partial_xx;
    end

    % Fill up J with B_i''
    for i = 1:n
        partial_x2x = 1/n_rx(i,3) * ...
            [H(1,:) - n_rx(i,1)/n_rx(i,3) * H(3,:);
             H(2,:) - n_rx(i,2)/n_rx(i,3) * H(3,:)];
        partial_xx = [-0.5 * sx(i,2:end); ...
              _sinc(norm(px(i,:))/2)/2 * eye(2) ...
              + 1/(4*norm(px(i,:))) * _dsinc(norm(px(i,:))/2) ...
              * (px(i,:)'*px(i,:))];
        J(2*n+2*i-1:2*n+2*i,8+2*i-1:8+2*i) ...
            = partial_x2x * partial_xx;
    end

    while true
        % Solve delta and update pH and px
        d = (J'*inv(Z)*J + lambda*eye(2*n+8)) \ (J'*inv(Z)*ex);
        n_pH = pH + d(1:8,1);
        n_px = px;
        for i = 1:n
            n_px(i,:) = px(i,:) + d(8+2*i-1:8+2*i)';
        end

        % Angle Normalize n_pH and n_px
        if norm(n_pH) > pi
            n_pH = (1 - 2*pi/norm(n_pH) ...
                * ceil((norm(n_pH)-pi)/(2*pi))) * n_pH;
```

```
            end
            for i = 1:n
                if norm(n_px(i,:)) > pi
                    n_px(i,:) = (1 - 2*pi/norm(n_px(i,:)) ...
                        * ceil((norm(n_px(i,:))-pi)/(2*pi))) ...
                        * n_px(i,:);
                end
            end

            % Deparameterize pH and px
            n_vH = deparameterize(n_pH);
            n_H = reshape(n_vH, 3, 3)';
            n_sx = sx;
            for i = 1:n
                n_sx(i,:) = deparameterize(n_px(i,:)')';
            end

            [n_lx, n_rx] = estimate(n_sx, n_H);
            nex = calEpsilon(lx, rx, n_lx, n_rx);
            nerr = nex'*inv(Z)*nex;

            if nerr < err
                H = n_H;
                sx = n_sx;
                lambda = 0.1 * lambda;
                break;
            else
                lambda = 10 * lambda;
            end
        end

        % Update error
        perr = err;
        err = nerr;
        log = [log, err];
    end

    % Data Denormalization
    H = U \ H * T;
end

function [n_lx, n_rx] = estimate(sx, H)
    n_lx = (eye(3) * sx')';
    n_rx = (H * sx')';
end

% n_lx, n_rx have been normalized by their third columns
function ex = calEpsilon(lx, rx, n_lx, n_rx)
    n_lx = n_lx ./ n_lx(:,3);
    n_rx = n_rx ./ n_rx(:,3);
    ex = [vector(lx(:,1:2)) - vector(n_lx(:,1:2)); ...
          vector(rx(:,1:2)) - vector(n_rx(:,1:2))];
end

function vx = vector(x)
    vx = x';
    vx = vx(:);
end

function p = parameterize(v)
    v = v / norm(v);
    a = v(1);
    b = v(2:end);
    p = 2 / _sinc(acos(a)) * b;
end
```

```
function v = deparameterize(p)
    v = [cos(norm(p)/2), _sinc(norm(p)/2)/2 * p']';
end

function ret = _sinc(x)
    if x == 0
        ret = 1;
    else
        ret = sin(x) / x;
    end
end

function ret = _dsinc(x)
    if x == 0
        ret = 0;
    else
        ret = cos(x) / x - sin(x) / x^2;
    end
end

function delta = calDelta(lx, rx, H)
    n = size(lx,1);
    vH = vector(H);
    delta = zeros(n,4);
    for i = 1:n
        Ai = [zeros(1,3), -lx(i,:), rx(i,2)*lx(i,:); ...
              lx(i,:), zeros(1,3), -rx(i,1)*lx(i,:)];
        ex = Ai * vH;
        J = [-H(2,1)+rx(i,2)*H(3,1), -H(2,2)+rx(i,2)*H(3,2), ...
             0, lx(i,1)*H(3,1)+lx(i,2)*H(3,2)+H(3,3); ...
             H(1,1)-rx(i,1)*H(3,1), H(1,2)-rx(i,1)*H(3,2), ...
             -(lx(i,1)*H(3,1)+lx(i,2)*H(3,2)+H(3,3)), 0];
        lambda = (J*J') \ (-ex);
        delta(i,:) = (J' * lambda)';
    end
end
```

Main Function

Code Listing 6: Main Function

```
%% Read Files
preI = imread('../dat/price_center20.JPG');
preI = rgb2gray(preI);
nxtI = imread('../dat/price_center21.JPG');
nxtI = rgb2gray(nxtI);

%% Parameters
lambda_threshold = 2200;
nw = 7;

%% Problem 1: Extract Features
pref = featureDetect(preI, lambda_threshold, nw);
fprintf('Number of extracted features: %d\n', size(pref, 1));
nxtf = featureDetect(nxtI, lambda_threshold, nw);
fprintf('Number of extracted features: %d\n', size(nxtf, 1));

% Draw Feature-Deteced Figures
res = figure('visible','off');
res.PaperPosition = [0 0 8 3];

subaxis(1, 2, 1, 'sh', 0.01, 'sv', 0, 'padding', 0, 'margin', 0.01);
imshow(preI);
```

```
hold on
plot(pref(:,1),pref(:,2),'bs', 'MarkerSize', nw);
hold off

subaxis(1, 2, 2, 'sh', 0.01, 'sv', 0, 'padding', 0, 'margin', 0.01);
imshow(nxtI);
hold on
plot(nxtf(:,1),nxtf(:,2),'bs', 'MarkerSize', nw);
hold off

saveas(res, '../res/pc_detect.jpg');

%% Problem 2: Match Features
[lx, rx] = featureMatch(preI, nxtI, pref, nxtf, nw);
dx = rx-lx;
fprintf('Number of matches: %d\n', size(lx, 1));

% Draw Figures
res = figure('visible','off');
res.PaperPosition = [0 0 8 3];

subaxis(1, 2, 1, 'sh', 0.01, 'sv', 0, 'padding', 0, 'margin', 0.01);
imshow(preI);
hold on
plot(lx(:,1), lx(:,2), 'bs', 'MarkerSize', nw);
quiver(lx(:,1), lx(:,2), dx(:,1), dx(:,2), 0);
hold off

subaxis(1, 2, 2, 'sh', 0.01, 'sv', 0, 'padding', 0, 'margin', 0.01);
imshow(nxtI);
hold on
plot(rx(:,1), rx(:,2), 'bs', 'MarkerSize', nw);
quiver(rx(:,1), rx(:,2), -dx(:,1), -dx(:,2), 0);
hold off

saveas(res, '../res/pc_match.jpg');

%% Problem 3: Outlier Reection
% Homogenize
lx(:,3) = ones(size(lx,1),1);
rx(:,3) = ones(size(rx,1),1);
[H, bmap, MAX_TRIALS, cost] = mSAC(lx, rx);
fprintf('\n\nProblem 3\n');
fprintf('Number of Inliers: %d\n', sum(bmap));
fprintf('Number of MaxTrials: %.10f\n', MAX_TRIALS);
fprintf('Final Cost: %.10f\n', cost);
RMSE = calRMSE(lx, rx, H);
fprintf('RMSE: %.10f\n', RMSE);

% Reject Outliers
lx = lx(bmap,:);
rx = rx(bmap,:);
dx = rx-lx;

% Draw Figures
res = figure('visible','off');
res.PaperPosition = [0 0 8 3];

subaxis(1, 2, 1, 'sh', 0.01, 'sv', 0, 'padding', 0, 'margin', 0.01);
imshow(preI);
hold on
plot(lx(:,1), lx(:,2), 'bs', 'MarkerSize', nw);
quiver(lx(:,1), lx(:,2), dx(:,1), dx(:,2), 0);
hold off
```

```
subaxis(1, 2, 2, 'sh', 0.01, 'sv', 0, 'padding', 0, 'margin', 0.01);
imshow(nxtI);
hold on
plot(rx(:,1), rx(:,2), 'bs', 'MarkerSize', nw);
quiver(rx(:,1), rx(:,2), -dx(:,1), -dx(:,2), 0);
hold off

saveas(res, '../res/pc_robust_match.jpg');

%% Problem 4: DLT Algorithm (Linear Estimation)
H = DLT_nc(lx, rx);
fprintf('\n\nProblem 4\n');
fprintf('H_DLT:\n'); disp(H ./ norm(H,'fro'));
RMSE = calRMSE(lx, rx, H);
fprintf('RMSE: %.10f\n', RMSE);

%% Problem 5: Levenberg-Marquardt Algorithm (NonLinear Estimation)
[H, logs] = LM_nc(lx, rx, H);
fprintf('\n\nProblem 5\n');
fprintf('P_LM:\n'); disp(H ./ norm(H,'fro'));
fprintf('Error log:\n'); disp(logs);
RMSE = calRMSE(lx, rx, H);
fprintf('RMSE: %.10f\n', RMSE);
```

Utility Function

Code Listing 7: Calculate Rooted Mean Squared Error

```
function RMSE = calRMSE(lx, rx, H)
    px = (H * lx')';
    px = px ./ px(:,3);
    diff = rx(:,1:2) - px(:,1:2);
    RMSE = sqrt(mean(sum(diff.^2,2)));
end
```