Computer Vision – Programming Project 4

王順興 0210814

1. **ImageStitchingBrightnessMathcing**.m: The main program, contains RANSAC and stitching.

For Image Stitching:

2. **Preprocessing**.m: Preprocessing for input images (Resize/Histogram equalization)
3. **surfFindMatchPoints**.m: Feature point detection /extraction/matching
4. **findHomography**.m: Non-linear homography finding

For Brightness Matching:

5. **IntensityMatchingMulti**.m: Main process of Brightness Matching
6. **PolynomialRegression**.m: Helper function for IntensityMatchingMulti. Find the parameter for brightness adjustment.
7. **PolynomialInference**.m: Helper function for IntensityMatchingMulti. Find the Intensity mapping for brightness adjustment.

Files for testing are omitted.

Full project: https://github.com/vicodin1123/CV-project/tree/master/p4

ImageStitchingBrightnessMathcing.m

Image stitching with brightness matching

```
clear; close all;
```

Read and resize the images

```
disp('Read images...');
EnableHEQ = false;
% Target
ImgT = imread('./data/a3.jpg');
ImgT = Preprocessing(ImgT, EnableHEQ);
% Photo
ImgW = imread('./data/a4.jpg');
ImgW = Preprocessing(ImgW, EnableHEQ);
```

Find matched SURF feature points

```
disp('Find SURF matching points...');
[T, W] = surfFindMatchPoints(histeq(rgb2gray(ImgT)), histeq(rgb2gray(ImgW)));

NumOfMPs = size(W, 1);

disp(sprintf('Num of MPs: %d', NumOfMPs));
```

RANSAC

```
disp('RANSAC...');
HomographyIterations = 300; % # of maximum iterations for non-linear optimization
RANSACiteration = min(max(500, NumOfMPs*10), 1000); % # of maximum iterations for
RANSAC
InlierThreshold = 1.0;    % Thershold for projection error in pixels

maxInliers = zeros(1,1);    % Store the largest set of inliers
maxInlierCount = -1;

for i = 1 : RANSACiteration
% Randomly pick four points
Indices = randperm(NumOfMPs, 4);
WorldCoord = W(Indices, :);
```

```
TargetCoord = T(Indices, :);

% Find homography using the four points
phi = findHomography(WorldCoord, TargetCoord, HomographyIterations);
H = double(reshape(phi, [3 3]));

% Find point-wise error - psi
exW = [W ones(NumOfMPs, 1)];
H = [phi(1:8);1];
denom = exW*H(7:9);
x = exW*H(1:3)./denom;
y = exW*H(4:6)./denom;
X = [x y];
psi = T - X;

% Find squared root error for each point
sqE = sqrt(psi(:, 1).^2 + psi(:, 2).^2);

% Count the number of inliers
Inliers = find(sqE<InlierThreshold);
InlierCount = numel(Inliers);

% If a better set of inliers is found, store it
if InlierCount > maxInlierCount
         maxInliers = Inliers;
    maxInlierCount = InlierCount;
    disp(sprintf('Itr: %d InlierCount: %d', i, maxInlierCount));

    if double(maxInlierCount)/NumOfMPs >= 0.7
        break;
    end
end

end
```

Intensity matching

```
WorldCoordInliers = W(maxInliers, :);

TargetCoordInliers = T(maxInliers, :);



% Match the intensity of two images using matched points
normImgW = IntensityMatchingMulti(ImgT, ImgW, TargetCoordInliers,
WorldCoordInliers);
```

Use the best set of inliers to find homography

```
disp('Find the final homography...');
disp(sprintf('NumOfMPs: %d Inliers: %d', NumOfMPs, maxInlierCount));

WorldCoordInliers = W(maxInliers, :);
TargetCoordInliers = T(maxInliers, :);

phi = findHomography(WorldCoordInliers, TargetCoordInliers, 3000);

% Here it is
H = double(reshape(phi, [3 3]));
```

Append the pattern onto the image

```
RNe = imref2d(size(normImgW));

t = projective2d(H);

[transformedImgW RNex] = imwarp(normImgW, RNe, t);

alpha = imwarp(255*ones([size(normImgW, 1) size(normImgW, 2)]), RNe, t);



% Fix the pivot of the stitched images for the four configurations
if floor(RNex.YWorldLimits(1)) < 0
    coord1Y = max(1, -floor(RNex.YWorldLimits(1)));
    coord2Y = 1;
    Ylim = max(-floor(RNex.YWorldLimits(1))+size(ImgT, 1), ...
```

```
        RNex.ImageSize(1));
else
    coord1Y = 1;
    coord2Y = max(1, floor(RNex.YWorldLimits(1)));
    Ylim = RNex.ImageSize(1) + floor(RNex.YWorldLimits(1));
end
if floor(RNex.XWorldLimits(1)) < 0
    coord1X = max(1, -floor(RNex.XWorldLimits(1)));
    coord2X = 1;
    Xlim = max(-floor(RNex.XWorldLimits(1))+size(ImgT, 2), ...
        RNex.ImageSize(2));
else
          coord1X = 1;
    coord2X = max(1, floor(RNex.XWorldLimits(1)));
    Xlim = RNex.ImageSize(2) +  floor(RNex.XWorldLimits(1));
end

% Project images according to the new coordinate
x2 = uint8(zeros([Ylim Xlim 3]));
x2( ...
          coord2Y:coord2Y + size(transformedImgW, 1) - 1, ...
    coord2X:coord2X + size(transformedImgW, 2) - 1, :)...
    = transformedImgW;

xAlpha = uint8(zeros([Ylim Xlim 1]));
xAlpha(...
          coord2Y:coord2Y + size(transformedImgW, 1) - 1, ...
    coord2X:coord2X + size(transformedImgW, 2) - 1, :)...
    = alpha;

x1 = uint8(zeros([Ylim Xlim 3]));
x1( ...
          coord1Y:coord1Y + size(ImgT, 1) - 1,...
    coord1X:coord1X + size(ImgT, 2) - 1, :)...
    = ImgT;
```

```
% Stack the images
image = uint8(zeros([Ylim Xlim 3]));
for i = 1 : 3
    image(:, :, i) = x1(:, :, i).*(1-xAlpha/255) + x2(:, :, i).*(xAlpha/255);
end

figure; imshow(image);
imwrite(image, 'result.jpg');
```

Preprocessing.m

```
function img = Preprocessing(oImg, EnableHEQ)
% Resize and apply Histogram Equalization acoording to setting
% INPT: oImgT: RGB image.
% OUPT: img: Processed RGB image.

% Resize
img = imresize(oImg, [NaN 1280]);

% RGB
if EnableHEQ
    img = rgb2hsv(img);
    img = cat(3, img(:, :, 1:2), histeq(img(:, :, 3)));
    img = uint8(hsv2rgb(img).*255);
end

end
```

surfFindMatchPoints.m

```
function [matchedPoints1 matchedPoints2] = surfFindMatchPoints(Img1, Img2)
% Find Corresponding Points Using SURF Features
% INPT: Img1, Img2: grayscale image of any size
% OUPT: matchedPoints1, matchedPoints2:
%       Nx2 matrix. [x y] coordinates of matched N FPs corresponding to Img1,Img2

% Find the SURF features
points1 = detectSURFFeatures(Img1);
points2 = detectSURFFeatures(Img2);

% Extract the features
[f1,vpts1] = extractFeatures(Img1,points1);
[f2,vpts2] = extractFeatures(Img2,points2);

% Retrieve the locations of matched points
indexPairs = matchFeatures(f1,f2) ;
matchedPoints1 = vpts1(indexPairs(:,1));
matchedPoints2 = vpts2(indexPairs(:,2));

% Remove points that are too close (mainly duplicates)
radius = 1;
p1BadIndices = zeros(size(matchedPoints1));
for i = 1 : size(matchedPoints1, 1)
    pts = matchedPoints1.Location(i, :);
    xDupe = abs(pts(1)-matchedPoints1.Location(:, 1)) < radius;
    yDupe = abs(pts(2)-matchedPoints1.Location(:, 2)) < radius;
    xDupe(i) = 0; yDupe(i) = 0;
    p1BadIndices = p1BadIndices | (xDupe&yDupe);
end
```

```matlab
p2BadIndices = zeros(size(matchedPoints2));
for i = 1 : size(matchedPoints2, 1)
    pts = matchedPoints2.Location(i, :);
    xDupe = abs(pts(1)-matchedPoints2.Location(:, 1)) < radius;
    yDupe = abs(pts(2)-matchedPoints2.Location(:, 2)) < radius;
    xDupe(i) = 0; yDupe(i) = 0;
    p2BadIndices = p2BadIndices | (xDupe&yDupe);
end

GoodIndices = ~(p1BadIndices | p2BadIndices);
matchedPoints1 = matchedPoints1(GoodIndices);
matchedPoints2 = matchedPoints2(GoodIndices);

% Return the coordinate
matchedPoints1 = matchedPoints1.Location;
matchedPoints2 = matchedPoints2.Location;


end
```

findHomography.m

```
function phi = findHomography(W, T, NumOfIterations)
% Find Corresponding Points Using SURF Features
% INPT: W: Nx2 matrix. The [x y] coordinates of world FPs
%       T: Nx2 matrix. The [x y] coordinates of target FPs
%       NumOfIterations: The # of maximum iterations for non-linear optimization
% OUPT: phi: 1x9 vector. The vectorized homography parameters

NumOfMPs = size(W, 1);
Lambda = 0.001;

% Append 1 for homogenous coordinate
W = [W ones(NumOfMPs, 1)];
T = [T ones(NumOfMPs, 1)];

A = zeros(2*NumOfMPs, 9);
for i = 1 : NumOfMPs
    w = W(i, :);
    x = T(i, :);
    a = [0 0 0 -w x(2)*w; ...
         w 0 0 0 -x(1)*w];
    A((i-1)*2+1:i*2, :) = a;
end
[U,S,V] = svd(A);
phi = V(:, 9);

% Reparameterize Phi
phi(1) = phi(1) + 1;
phi(5) = phi(5) + 1;
phi = phi(1:8);
exphi = [phi(1:8);1];

T = T(:, 1:2);
X = zeros(NumOfMPs, 2);

for iteration = 1 : NumOfIterations
```

```
% Find Psi
exphi = [phi(1:8);1];
denom = W*exphi(7:9);
x = W*exphi(1:3)./denom;
y = W*exphi(4:6)./denom;
X = [x y];
psi = T - X;

% Construct J
J = zeros(2*NumOfMPs, 8);
for i = 1 : NumOfMPs
    w = W(i, :);
    x = T(i, :);
    j = [w 0 0 0 -x(1)*w(1) -x(1)*w(2); ...
         0 0 0 w -x(2)*w(1) -x(2)*w(2)];
    J((i-1)*2+1:i*2, :) = j./(w*exphi(7:9));
end

A = zeros(8, 8);
b = zeros(8, 1);
for i = 1 : NumOfMPs
    j = J((i-1)*2+1:i*2, :);
    A = A + j'*j;
    b = b + j'*psi(i, :)';
end

if numel(find(isinf(A))) ~= 0 || numel(find(isnan(A)))
    break;
end

% Find gradient
dPhi = pinv(A)*b;
phi = phi + dPhi;

if mean(dPhi) < 1e-6
    phi = [phi(1:8);1];
    return;
```

```
end


% Stop when the projection error is 0
AvgError = (sum(abs(psi(:, 1))) + sum(abs(psi(:, 2))))/2/size(psi, 1);
if AvgError == 0
    break;
end


end
phi = [phi(1:8);1];
% disp(sprintf('for i=%d: %.2f %.2f', iteration, sum(abs(psi(:, 1))),
sum(abs(psi(:, 2)))));
end
```

IntensityMatchingMulti.m

```
function normImgW = IntensityMatchingMulti(ImgT, ImgW, TargetCoordInliers,
WorldCoordInliers)
% Find the modified ImgW that has the brightness matched with ImgT using
% given matched points
% INPT: ImgT: The image that has the targeted brightness.
%       ImgW: The image that we wish to modify for the matching.
%       TargetCoordInliers: Nx2 matrix. The [x y] coordinates of ImgT's
%       matched points.
%       WorldCoordInliers: Nx2 matrix. The [x y] coordinates of ImgW's
%       matched points.
% OUPT: normImgW: Modified ImgW that has the brightness matched with ImgT
```

Constants

```
MaxInlierCount = size(WorldCoordInliers, 1);    % # of the MPs
PolyRegressionOrder = 2;    % Order of Polynomial Regression (1~3)
Neighbors = 1;              % The # of neighbors to consider for brightness counting
BlockSize = (Neighbors*2+1)^2;  % The size of the region considered for brightness
counting
MidGain = 4;                % The weight for the inlier itself
```

Brightness value counting

Use HSV images so that the brightness can be seperated

```
HSVImgT = rgb2hsv(ImgT);

HSVImgW = rgb2hsv(ImgW);
```

```
% Count the brightness of a region centered by given MPs
TargetInt = zeros(MaxInlierCount, 1);
WorldInt = zeros(MaxInlierCount, 1);
for i = 1 : MaxInlierCount
    Wcoord = floor(WorldCoordInliers(i, :));
    Tcoord = floor(TargetCoordInliers(i, :));
    for r = -Neighbors : Neighbors
        for c = -Neighbors : Neighbors
            if r == 0 && c == 0
                k = MidGain;
            else
                k = 1;
            end
            TargetInt(i) = TargetInt(i) + ...
                k*HSVImgT(Tcoord(2) + r, Tcoord(1) + c, 3);
            WorldInt(i) = WorldInt(i) + ...
                k*HSVImgW(Wcoord(2) + r, Wcoord(1) + c, 3);
        end
    end
end

% Normalize according to the size of considered region and the weight of the
% inlier itself
TargetInt = TargetInt./(BlockSize-1+MidGain);
WorldInt = WorldInt./(BlockSize-1+MidGain);
```

DEBUG ONLY - Show the RMS error of brightness after adjustment

Learning

```
Param = PolynomialRegression(WorldInt, TargetInt, 1, PolyRegressionOrder);

% Inference
Pred = PolynomialInference(WorldInt, Param, 1, PolyRegressionOrder);

Erms = sqrt( sum((WorldInt-TargetInt).^2)/size(TargetInt, 1) );
disp(sprintf('Original Erms: %.3f', Erms));
Erms = sqrt( sum((Pred-TargetInt).^2)/size(TargetInt, 1) );
disp(sprintf('Order:%d Erms: %.3f', PolyRegressionOrder, Erms));
```

Construct brightness mapping

Only consider the unique brightness values in ImgW

```
IntTable = unique(HSVImgW(:, :, 3));



% Find the mapping
IntTarget = PolynomialInference(IntTable, Param, 1, PolyRegressionOrder);

% Sometimes the value exceeds the boundary [0 1]
IntTarget(IntTarget>1) = 1; IntTarget(IntTarget<0) = 0;
```

Adjust the brightness and return the brightness matched image

```
for i = 1 : numel(IntTarget)
    HSVImgW(HSVImgW==IntTable(i)) = IntTarget(i);
end
normImgW = uint8(hsv2rgb(HSVImgW).*255);
end
```

PolynomialRegression.m

```
function W = PolynomialRegression(X, T, Dimension, Order)
% Learn the polynomial parameter W using data X and target T
% INPT: X: NxD matrix. N samples with D dimension.
%       T: Nx1 vector. The traget value of N samples
%       Dimension: scalar. Dimension of the sample.
%       Order: scalar. Order of the polynomial.
% OUPT: W: Kx1 vector. The parameters of the polynomial y = W'x.
```

Constants and data reading %%

```
cTrainSetSize = size(X, 1);

cDimension = Dimension;

cOrder = Order;

TrainX = X;

TrainT = T;
```

2/3nd order linear regression by partial derivation Build the Normal Equation: AW = Y

```
cOrder = min(cOrder, 3);

if cDimension > 1
    S = (1 - cDimension^(cOrder+1))/(1 - cDimension);
else
    S = Order + 1;
end

W = zeros(S);
Y = zeros(length(W), 1);
A = zeros(length(W), length(W));
```

```
% Build the Weight from partial derivative
Weight = ones(cTrainSetSize, 1);
for i = 1 : cDimension
    Weight = cat(2, Weight, TrainX(:, i));
end
if cOrder > 1
for i = 1 : cDimension
    for r = 1 : cDimension
        Weight = cat(2, Weight, TrainX(:, i).*TrainX(:, r));
    end
end
end
if cOrder > 2
for i = 1 : cDimension
    for r = 1 : cDimension
        for p = 1 : cDimension
        Weight = cat(2, Weight, TrainX(:, i).*TrainX(:, r).*TrainX(:, p));
        end
    end
end
end

% Build the A and Y matrix
for i = 1 : length(W)
        A(i, :) = sum(Weight.*repmat(Weight(:, i), 1, S));
        Y(i) = sum(TrainT.*Weight(:, i));
end
```

Find W using W = pinv(A)Y

```
W = pinv(A)*Y;

end
```

PolynomialInference.m

```
function Predictions = PolynomialInference(X, W, Dimension, Order)
% Calculate the inferred value of input X using polynomial paramter W
% INPT: X: NxD matrix. N samples with D dimension.
%       W: Kx1 vector. The parameters of the polynomial y = W'x.
%       Dimension: scalar. Dimension of the sample.
%       Order: scalar. Order of the polynomial.
% OUPT: Predictions: Nx1 vector. Inferred value of X.
```

Constants and data reading %%

```
cTestSetSize = size(X, 1);

cDimension = Dimension;

cOrder = Order;

TestX = X;
```

Construct X for test set

```
X = ones(cTestSetSize, 1);

for i = 1 : cDimension
    X = cat(2, X, TestX(:, i));
end
if cOrder > 1
for i = 1 : cDimension
    for r = 1 : cDimension
        X = cat(2, X, TestX(:, i).*TestX(:, r));
    end
end
end
if cOrder > 2
for i = 1 : cDimension
    for r = 1 : cDimension
        for p = 1 : cDimension
        X = cat(2, X, TestX(:, i).*TestX(:, r).*TestX(:, p));
        end
    end
end
end
```

Predict

```
Predictions = X*W;
```