```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Project_6_Blending.m
% Associated Files: Prep_Candidate.m, DoG_Pyramid.m, G_Pyramid.m,
%                   Thresholding.m, Cost_Band.m, Cost_Seam.m,
%                   Cost_Intensity.m, Calc_Distance.m, Calc_Error.m
%                   Insert_Min.m, Calssification.m
%
% EE368 "What Could Have Been" Project
% Huimin Huang
% Tian Kai Woon
% Created 5/16/2010
% Last Modified 6/04/2010
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Perform Laplacian blending(Approximated by DoG) of input image with
% minimum cost candidate images.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% close all
clear all
t = cputime;
% Declare parameters
    fname = 'Mask\Road_mask.bmp'; %Input image filename
    s = [512 512]; %Output image size
    hsize = round(min(s)/64); % Gaussian filter size
    sigma = 1; % Gaussian filter std dev
    t_max = log2(min(s)); % Pyramid height
    num_class = 3;
    num_min = 5;

% Read in input image
disp(['Input image: ' fname])
input = double(imread(fname));

% Process input image: Create mask
input_gray = mean(input,3);
input_mask = Thresholding(input_gray, 255);
input_mask = imresize(input_mask,s,'nearest');
imwrite(input_mask, [fname(1:end-4) 'BW.bmp']);
disp(['Pause to edit ' fname(1:end-4) 'BW.bmp'])
figure
imshow(input_mask); title('Input Mask')
keyboard
input_mask = logical(imread([fname(1:end-4) 'BW.bmp']));

dist_mat = Calc_Distance(input_mask);

% Process input image: Resize
input = imresize(input,s);

% Process input image: Classify
[folder_class num_pic_class] = Classification(input, num_class);
disp(['Classification: ' folder_class{1:end} ])
```

```
% Find minimum cost candidate images
min_cost = inf(1,num_min); % Store min. cost values
min_cost_candidate = cell(1,num_min); % Store min. cost images
min_fname_candidate = cell(1,num_min); % Store min. cost file names
min_bitmask = cell(1,num_min); % Store min. cost bitmask
for index_folder = 1 : length(folder_class)
    for index_pic = 1 : num_pic_class{index_folder}
        % Pre-process candidate image
        fname_candidate = ['Training\' folder_class{index_folder} '\' num2str(index_pic) ↙
];
        disp(['Candidate image: ' fname_candidate])
        [colour_match candidate] = Prep_Candidate(fname_candidate, input, input_mask);
        if colour_match %Check for rgb and grayscale discrepencies
            [cost bitmask] = Cost_Seam(input, candidate, dist_mat, input_mask);  %↙
Calculate cost and create bitmask
            [min_cost, min_cost_candidate, min_fname_candidate, min_bitmask] = Insert_Min ↙
(min_cost, min_cost_candidate, min_fname_candidate, min_bitmask, cost, candidate, ↙
fname_candidate, bitmask); %Store minimum cost candidates
            disp(['Cost: ' num2str(cost)])
        end
    end
end


% Perform Laplacian blending approximated by DoG Pyramids
figure;
filt = fspecial('gaussian',hsize,sigma); % Create Gaussian filter
for index = 1 : num_min
    candidate = min_cost_candidate{index};
    fname_candidate = min_fname_candidate{index};

    %Build Laplacian(DoG) Pyramid of 2 images and Gaussian Pyramid of bitmask
    disp(['Performing blending of ' fname_candidate])
    bitmask_scale_G = G_Pyramid(min_bitmask{index},filt,t_max);  % Gaussian Pyramid of↙
bitmask
    im_recon = zeros(size(input));

    for rgb = 1 : size(input,3)
        % Create DoG of input and candidate images
        input_t = input(:,:,rgb);
        candidate_t = candidate(:,:,rgb);
        input_t(input_mask) = candidate_t(input_mask);
        input_scale_DoG = DoG_Pyramid(input_t, filt, t_max);
        candidate_scale_DoG = DoG_Pyramid(candidate_t, filt, t_max);

        % Perform blending and reconstruction of image
        for t = 1 : t_max
            output_scale = bitmask_scale_G{t}.*candidate_scale_DoG{t} + (1-bitmask_scale_G ↙
{t}).*input_scale_DoG{t};
            upsamp = imresize(output_scale, s);
            im_recon(:,:,rgb) = im_recon(:,:,rgb) + upsamp;
        end
```

```
    end
    subplot(3,2,index);
    imshow(uint8(im_recon))
    title(['Blending with ' fname_candidate ', Cost = ' num2str(min_cost(index))])
    imwrite(uint8(im_recon),[fname(6:end-4) '_' num2str(index) '.bmp'], 'bmp');
end
disp('Complete')
disp(['Elapsed time: ' num2str(cputime-t)])
disp(' ')
```

```
function [colour_match candidate] = Prep_Candidate(fname_candidate,data, data_mask)
% function candidate = PrepCandidate(fname_candidate,data)
% Takes in file name of candidate image, fname_candidate, and resizes the
% candidate image to the size of data and check for rgb and grayscale
% mismatches

candidate = double(imread(fname_candidate));
colour_match = true;
z_data = size(data,3);
z_candidate = size(candidate,3);

% Resize Candidate to size of data and check for rgb and gray scale
% discrepencies
if z_data > z_candidate
    colour_match = false;
else
    if z_data == z_candidate
        candidate = imresize(candidate,[size(data,1) size(data,2)]);
    elseif z_data < z_candidate
        candidate = imresize(mean(candidate,3),[size(data,1) size(data,2)]);
        candidate = repmat(candidate,[1 1 z_data]);
    end
end
```

```
function scale_DoG = DoG_Pyramid(data, filt, t_max)
% function scale_DoG = DoG_Pyramid(data, filt, t_max)
% Takes in 2-D Image, data, and filter, filt, to create a DoG Pyramid
% of height t_max.

% Calculate Gaussian Pyramid
scale_G = G_Pyramid(data, filt, t_max);

% Calculate DoG Pyramid
scale_DoG = cell(1,t_max);
scale_DoG{t_max} = scale_G{t_max};
for t = t_max-1 : -1 : 1
    upsamp = imresize(scale_G{t+1},size(scale_G{t}));
    scale_DoG{t} = scale_G{t} - upsamp;
end
```

```
function scale_G = G_Pyramid(data, filt, t_max)
% function scale_G = G_Pyramid(data, filt, t_max)
% Takes in 2-D image data and builds a Gaussian pyramid of height t_max
% using filt

scale_G = cell(1,t_max);
scale_G{1} = data;
for t = 2 : t_max
    im_filt = filter2(filt,scale_G{t-1});
    scale_G{t} = imresize(im_filt,0.5);
end
```

```
function data_mask = Thresholding(data_BW, thres)
% function Thresholding(data_BW, thres)
% Takes in a gray scale image and sets all values below thres to false and all
% values equal and above to thres to true

data_mask = zeros(size(data_BW));
data_mask(data_BW >= thres) = 1;
data_mask = logical(data_mask);
```

```
function data_band = Cost_Band(data_mask, r)
% function data_band = Cost_Band(data_mask, r)
% Takes in a binary image data_thres and builds a band of radius r around
% the white regions.

% Build blending and cost-function radius
se = strel('disk',r+1);
data_dil = imdilate(data_mask, se, 'same');
data_band = logical(data_dil - data_mask);
```

```
function [cost bitmask_seam] = Cost_Seam(input, candidate, dist_mat, input_mask)
% function [cost bitmask_seam] = Cost_Seam(input, candidate, input_mask, input_band)
% Takes in input image and candidate image and calculates the local minimum cost
% seam to connect the 2 images along the bitmask
% Cost(seam line) = Distance from input_mask + Pixel colour gradient + Intensity
% difference of input_band
err_mat = Calc_Error(input, candidate, dist_mat, input_mask);  % Calculate error matrix

% Find local minimum cost seam
cost = inf;
cost_temp = 0;
bitmask_seam = input_mask;
boundary = logical(bitmask_seam - imerode(bitmask_seam,ones(3)));
r_max = size(input_mask,1);
c_max = size(input_mask,2);
count = 0;
while cost ~= cost_temp && count < 100
    cost_temp = cost;
    [r c] = find(boundary);
    for index = 1 : sum(boundary(:))
        % Expand bitmask_seam to local minimums
        rr = max([r(index)-1 1]):min([r(index)+1 r_max]);
        cc = max([c(index)-1 1]):min([c(index)+1 c_max]);
        err_temp = err_mat(rr,cc);
        err = err_temp <= err_mat(r(index),c(index));
        bitmask_seam(rr,cc) = bitmask_seam(rr,cc) | err;
    end
    boundary = logical(bitmask_seam - imerode(bitmask_seam,ones(3)));
    cost = mean(mean(err_mat(boundary)));
    count = count + 1;
end
bitmask_seam = imclose(bitmask_seam,ones(3));
bitmask_band = Cost_Band(bitmask_seam, 10);
if count >= 99
    disp('Infinite loop')
end
% Weight cost
w = 0.00005;
cost = cost + w * Cost_Intensity(input, candidate, bitmask_band);
```

```
function cost = Cost_Intensity(data, candidate, data_band)
% function cost = Cost_Intensity(data, candidate, data_band)
% Takes in 2 images, data and candidate, and calculates the mean square
% error between the two images across data_band. The MSE is output as cost.
% candidate must have the same dimensions as data. data_band must have the
% same dimensions in DIM-1 and DIM-2 as data.

data_band_rep = repmat(data_band,[1 1 size(data,3)]);
SqErr = (data(data_band_rep) - candidate(data_band_rep)).^2;
cost = mean(SqErr(:));
```

```
function dist_mat = Calc_Distance(input_mask)
% function dist_mat = Distance(size(input,1), size(input,2), input_mask);
% Calculates the minimum distance of each pixel to the hole of 1's in
% input_mask. Sets distance within input_mask = 0

[num_r num_c] = size(input_mask);
x = (-num_c + 1 : num_c - 1);
y = (-num_r + 1 : num_r - 1);
x2 = x.^2;
y2 = y.^2;
[xx2 yy2] = meshgrid(x2,y2);
dist_template = (xx2 + yy2).^0.5;

boundary = input_mask - imerode(input_mask,ones(3));
[r c] = find(boundary);
dist_mat = inf(num_r,num_c);
for index = 1 : sum(boundary(:))
    dist_mat = min(cat(3,dist_mat,dist_template(num_r-r(index)+1:end-r(index)+1,num_c-c ↙
(index)+1:end-c(index)+1)),[],3);
end
dist_mat(input_mask) = 0;
```

```
function err_mat = Calc_Error(input,candidate,dist_mat, input_mask)
% function err_mat = Calc_Error(input,candidate, dist_mat, input_mask)
% Creates the error associated to each pixel based on distance to the input
% mask and pixel colour gradient. Cost within input_mask = inf

k = 0.02; % Weight on distance cost
l = 0.001; % Weight on gradient cost
SD = (input-candidate).^2;
SSD = sum(SD,3);
[fx fy] = gradient(SSD);
grad = (fx.^2 + fy.^2).^0.5;
err_mat = (k * dist_mat).^3 + l * grad;
err_mat(input_mask) = inf;
```

```
function [min_cost, min_cost_candidate, min_fname_candidate, min_bitmask] = Insert_Min ↙
(min_cost, min_cost_candidate, min_fname_candidate, min_bitmask, cost, candidate, ↙
fname_candidate, bitmask)
% function [min_cost, min_cost_candidate, min_fname, min_bitmask] = Insert_Min(min_cost, ↙
min_cost_candidate, min_fname, min_bitmask, cost, candidate, fname_candidate, bitmask)
% Inserts values of cost, candidate, fname and bitmask to keep track of the
% minimum cost candidates

if cost ~= min_cost
    index_insert = find(cost < min_cost, 1, 'first');
    if ~isempty(index_insert)
        min_cost(index_insert+1:end) = min_cost(index_insert:end-1);
        min_cost(index_insert) = cost;
        min_cost_candidate(index_insert+1:end) = min_cost_candidate(index_insert:end-1);
        min_cost_candidate{index_insert} = candidate;
        min_fname_candidate(index_insert+1:end) = min_fname_candidate(index_insert:end-1);
        min_fname_candidate{index_insert} = fname_candidate;
        min_bitmask(index_insert+1:end) = min_bitmask(index_insert:end-1);
        min_bitmask{index_insert} = bitmask;
    end
end
```

```
function [folder_class num_pic_class] = Classification(input, num_matches)
% function [folder_class num_pic_class] = Classification(input, num_matches)
% Classification of input using eigen images to match candidate image to
% similar classes of images. Projection of input onto eigen image space
% used as measure of error
load 'Training\training.mat'

input_s = reshape(input, [], 1);
proj_err = zeros(1,length(folder));

% Calculate error in projection of input onto eigen image space using least
% squares approximation
for index_folder = 1 : length(folder)
    alpha = (V{index_folder}.'*V{index_folder})^-1*V{index_folder}.'*(input_s-A_mean ↙
{index_folder});
    input_proj = A_mean{index_folder} + V{index_folder}*alpha;
    proj_err(index_folder) = norm(input_s - input_proj);
end

% Pick num_matches classications with least error
[proj_err_sort I] = sort(proj_err,2,'ascend');
folder_class = folder(I(1:num_matches));
num_pic_class = num_pic(I(1:num_matches));
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Project_6_Training.m
% Associated Files:
%
% EE368 "What Could Have Been" Project
% Huimin Huang
% Tian Kai Woon
% Created 6/01/2010
% Last Modified 6/01/2010
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create eigen images representative of the respective classifications
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all

folder = {'Building' 'Portrait' 'Road' 'Sky' 'Water'}; % Available classifications
num_pic = {20 20 20 36 20}; % Number of training pics in each class
V = cell(1, length(folder)); % Store eigen images
A_mean = cell(1, length(folder)); % Store mean images
s = [512 512]; %Image size
num_eig = 5; % Number of eigen images

% Calculate eigen images for each classification
for index_folder = 1 : length(folder)
    disp(['Current Folder: ' folder{index_folder}]);
    A = zeros(s(1)*s(2)*3, num_pic{index_folder});

    % Populate A with training pics
    for index_pic = 1 : num_pic{index_folder}
        disp(['Training\' folder{index_folder} '\' num2str(index_pic)]);
        data = imread(['Training\' folder{index_folder} '\' num2str(index_pic)]);
        data = imresize(data,s);
        if size(data,3) ~= 3
            data = repmat(mean(data,3),[1 1 3]);
        end
        A(:,index_pic) = reshape(data,[],1);
    end

    A_mean{index_folder} = mean(A,2);
    A = A - repmat(A_mean{index_folder},[1 num_pic{index_folder}]);
    [V_unsort D_unsort] = eig(A.'*A);
    [Y I] = sort(diag(D_unsort), 1, 'descend');
    V_sort = V_unsort(:,I(1:num_eig));
    V{index_folder} = A * V_sort; % Store r most significant eigen images
end

save 'Training\training.mat' V A_mean folder num_pic;
clear all
```