Lund University
Faculty of Engineering
Name: ZHANG Yifei
Email: yi4840zh-s@lu.se
Date: October 23, 2023

FMAN20 - Image Analysis, Autumn 2023

Exam

Answer to Task 1

1. Calculating the Number of Parameters for Each Model:

   (a) Nave Bayes Classifier: For each class, you need to estimate the mean and variance for each pixel (feature) independently. In a 40x40 image, you have 1600 pixels. For each class, you need to estimate 2 parameters per pixel (mean and variance).

   If you have 10 classes, you will need 10 times the number of parameters for one class.

   Total parameters = 10 * 2 * 1600 = 32,000 parameters.

   (b) K-Nearest Neighbors (KNN): KNN doesn't learn parameters during training. It stores all the training data points. No trainable parameters in the model.

   (c) Convolutional Neural Network (CNN): Convolutional Layer: Each kernel (filter) requires 3x3 weights (parameters), and you have 32 output channels. Parameters in Convolution Layer = 3x3x32 = 288 parameters.

   Fully Connected Layer: This layer connects each of the 20x20x32 input nodes to 10 output nodes, requiring 20x20x32x10 weights and 10 biases.

   Parameters in Fully Connected Layer = 20x20x32x10 + 10 = 128,010 parameters.

   Total parameters in CNN = Parameters in Convolution Layer + Parameters in Fully Connected Layer = 288 + 128,010 = 128,298 parameters.

2. Training Process for Each Model:

   (a) Nave Bayes Classifier: Estimate the mean and variance of each pixel for each class using the training data. Use the training data to compute class priors (prior probabilities) based on the number of training samples in each class. The model is now trained and ready for inference.

   (b) K-Nearest Neighbors (KNN): In KNN, there is no training phase. The model simply stores all the training data points.

   (c) Convolutional Neural Network (CNN): Convolutional Layer: The weights (kernels) in the convolution layer are learned through backpropagation and gradient descent during training. You pass the training images through the network, calculate the gradients, and update the weights.

   Fully Connected Layer: Similar to the convolutional layer, the weights in the fully connected layer are learned during training.

   The network's parameters are adjusted until the loss function converges to a minimum.

3. Inference Phase for Each Model:

   (a) Nave Bayes Classifier: Given a new image, calculate the likelihood of observing the pixel values under each class's Gaussian distribution.

   Multiply the likelihood by the class priors (prior probabilities).

   The class with the highest posterior probability is the predicted class.

   (b) K-Nearest Neighbors (KNN): Given a new image, find the K nearest neighbors from the training data based on some similarity metric (e.g., Euclidean distance) using the pixel values.

   Assign the class label that is most common among the K nearest neighbors as the predicted class.

(c) Convolutional Neural Network (CNN): Pass the new image through the trained network, layer by layer.

In the convolutional layers, apply convolutions with learned filters and apply ReLU activations.

In the fully connected layers, compute weighted sums and apply non-linear activations. The softmax layer gives the probability distribution over the classes, and the class with the highest probability is the predicted class.

In all cases, the model is used to make predictions on new data points during the inference phase based on the parameters learned during training.

Answer to Task 2

For this task, my primary approach is to recognize that the overall structure of the maze is divided by several walls, so once I can isolate the individual parts of the maze walls, the areas in between are the pathways.

I start by using an eight-neighbor algorithm to perform image segmentation on the black pixels (representing the maze walls), resulting in two components, shown as 1. I then apply dilation and erosion operations to each of these two components, and subtract the results of the dilation and erosion operations from one another, taking the absolute value. This process yields the paths within the maze, shown as Figure 2.

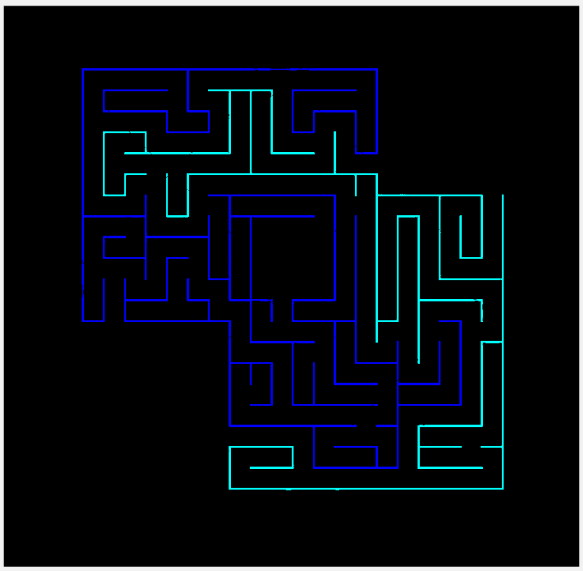

Figure 1: Task2 1

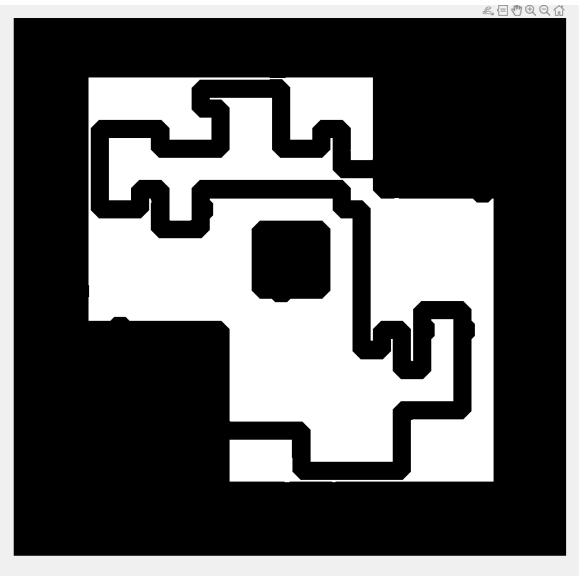

Figure 2: Task2 2

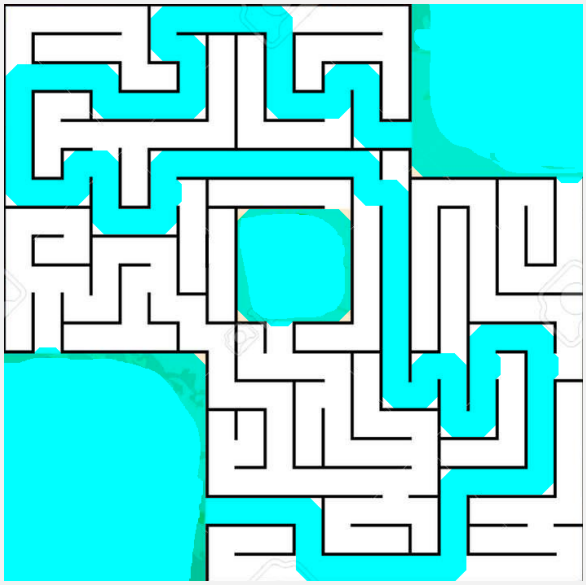

Figure 3: Task2 3

The code is as follow.

```
clc
clear
close all

% Read the image
image = imread('pirate1.png');
originalImage = image;
% Convert the image to grayscale (if it's not already grayscale)
if size(image, 3) == 3
    grayImage = rgb2gray(image);
else
    grayImage = image;
end

```

```
% Binarize the image, setting the black background to 0 and objects to 1
binaryImage = imbinarize(grayImage, 0.35);
invertedImage = imcomplement(binaryImage);

% Extract connected components using 8-connectivity
cc = bwconncomp(invertedImage, 8);
numObjects = cc.NumObjects;

% Assign different colors to each connected component
labeledImage = labelmatrix(cc);
labelImage = labeledImage;
rgbImage = label2rgb(labeledImage, 'jet', 'k');

% % Display the resulting image
% imshow(rgbImage);
% title(['Found ', num2str(numObjects), ' objects']);

% Create a binary mask for the current connected component
currentRegion = labelImage == 1;
se = strel('disk', 25);  % Adjust the dilation radius
dilatedRegion1 = imdilate(currentRegion, se);

erodedRegion1 = imerode(dilatedRegion1, se);

currentRegion = labelImage == 2;
se = strel('disk', 25);  % Adjust the dilation radius
dilatedRegion2 = imdilate(currentRegion, se);

erodedRegion2 = imerode(dilatedRegion2, se);

mask = abs(erodedRegion1 - erodedRegion2);
mask = imcomplement(mask);

% Apply the mask to the original image
resultImage = originalImage;
resultImage(:, :, 1) = uint8(resultImage(:, :, 1)) - uint8(mask * 255);

% Display the resulting image
imshow(resultImage);
```

Answer to Task 3

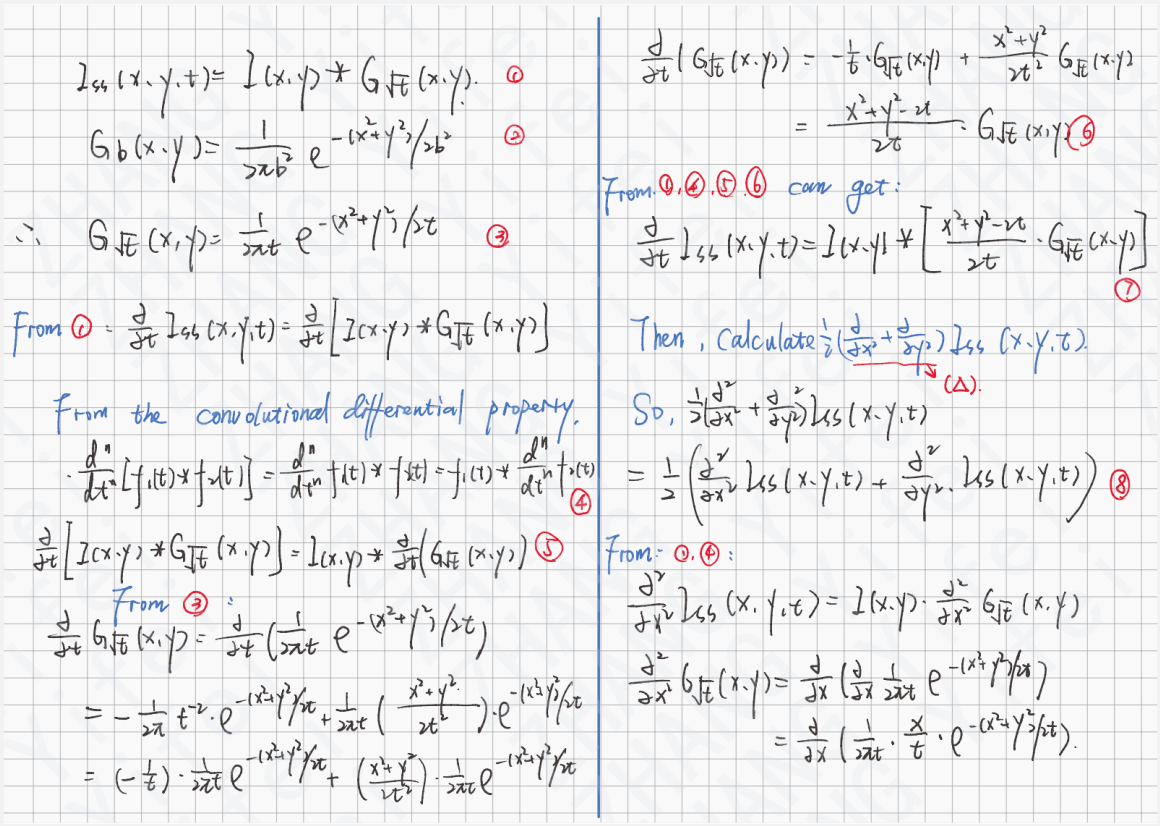

Figure 4: Task3 1

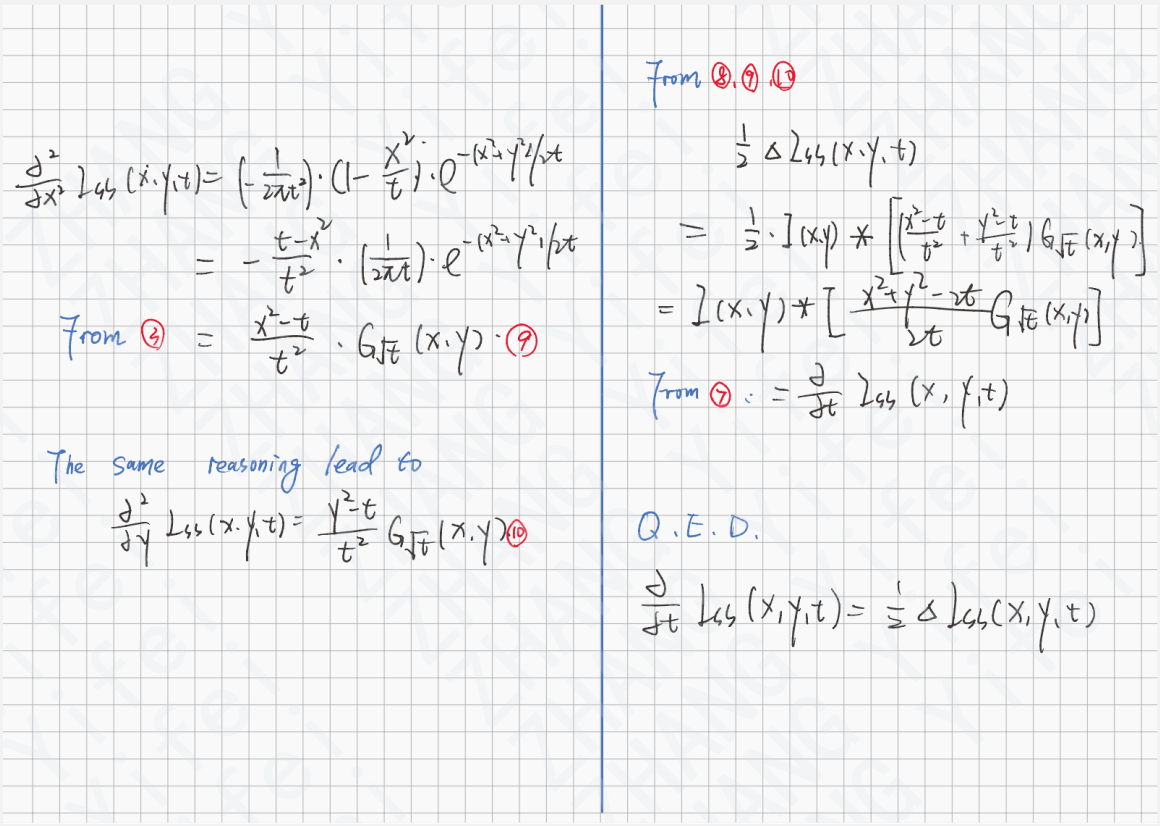

Figure 5: Task3 2

Answer to Task 4

I opted for HSV color thresholding and the Hough circle detection method to identify the coins. Initially, I read the HSV colors of each type of coin on 'coins.jpg' and selected the color thresholds. Then, using OpenCV, I obtained a mask and applied it over the grayscale version of the original image.

Afterward, I employed the Hough transform to detect circles. This method performs with high accuracy in scenarios with simple backgrounds, but it may not work well in complex backgrounds. I also experimented with methods like template matching, but the results were not as satisfactory.

Code:

```
# -*- coding: utf-8 -*-
"""
@Time: 2023/10/21 18:40
@Author: Dexter ZHANG
@File: task4.py
@IDE: PyCharm
"""

#!/usr/bin/env python3
import cv2
import cv2 as cv
import numpy as np

img_file = r'coin4.jpg'  # Name of the image
img = cv2.imread(img_file)

height, width = img.shape[:2]

# Specify new width and height (1/4 of the original)
new_width = width // 4
new_height = height // 4

# Resize the image using the resize function
img = cv2.resize(img, (new_width, new_height))
ori_img = img
#print('Image dimensions: {}'.format(img.shape)) # Print the dimensions of the image

# gray_img = cv2.cvtColor(img, cv2.COLOR_BGRA2GRAY)  # Convert to grayscale
kernel_size = (5, 5)  # Adjust kernel size as needed
kernel_size = 5
# Specify the standard deviation of the Gaussian kernel
sigma = 0  # Adjust the standard deviation as needed, 0 means auto calculation

# Apply Gaussian blur
# img = cv2.GaussianBlur(img, kernel_size, sigma)  # Median blur
img = cv2.medianBlur(img, kernel_size)

gray_img = cv2.cvtColor(img, code=cv2.COLOR_BGRA2GRAY)
hsv_img = cv2.cvtColor(img, code=cv2.COLOR_BGR2HSV)  # Color space transformation

lower = np.array([0, 0, 100])
upper = np.array([100, 200, 255])

# lower = np.array([0, 20, 80])
# upper = np.array([200, 120, 170])

mask = cv2.inRange(hsv_img, lower, upper)

# kernel_size = (5, 5)
kernel_size = (20, 20)

# Create an elliptical kernel
```

```
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, kernel_size)

# Apply the closing operation to the image
closed_image = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
opened_image = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)

res = cv2.bitwise_and(gray_img, gray_img, mask=closed_image)


circles = cv2.HoughCircles(
                          res,                    # Input image (can be directly ...
                              grayscale)
                          cv2.HOUGH_GRADIENT,       # 3-element output vector (x, ...
                              y, and radius)
                          1,                       # Accumulator has the same ...
                              resolution as the input image
                          80,                       # Minimum distance between the ...
                              centers of detected circles (discard circles too close)
                          param1=25,
                          param2=67,
                          minRadius=45,             # Minimum circle radius
                          maxRadius=100)             # Maximum circle radius

if circles is None:
    print("None")
else:
    circles = np.uint16(np.around(circles))
    print(circles)  # Print circle position information (x-coordinate, ...
        y-coordinate, and radius)
    for i in circles[0, :]:
        cv2.circle(ori_img, (i[0], i[1]), i[2], (0, 255, 0), 2)
        cv2.circle(ori_img, (i[0], i[1]), 2, (0, 0, 255), 3)

cv2.imshow('circle', ori_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Result:

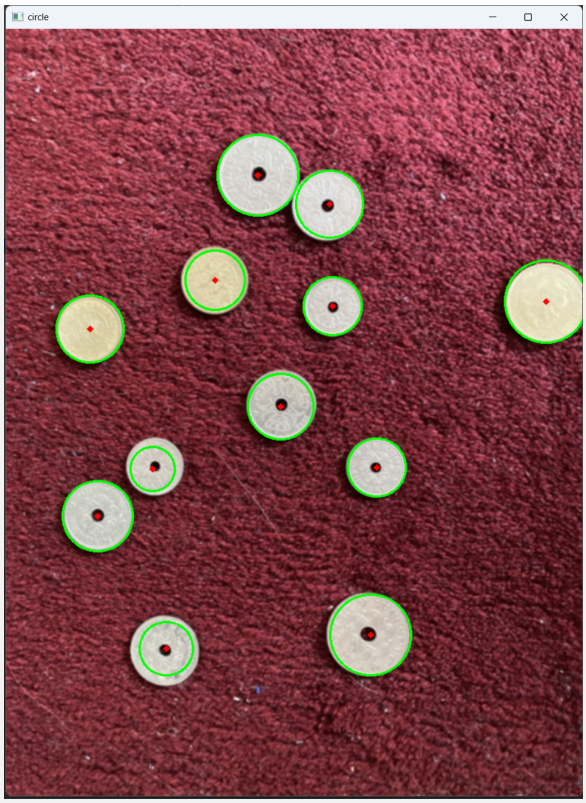

Figure 6: Task4 0

Figure 7: Task4 1

Figure 8: Task4 2

Figure 9: Task4 3

Figure 10: Task4 4

Answer to Task 5

a) The problem specifies a linear transformation for RGB values, and I attempted to solve for the parameters using both the least-squares method and RANSAC.

However, the results were not very satisfactory with either approach.

Least-squares :

```
clc;
clear;
close all;
load('imdata_pineapple.mat')
% Step 1: Load two images
image1 = imread('pineapple2.jpg'); % Load the first image (gt)
image2 = imread('pineapple_t1.jpg'); % Load the second image

xy2 = u1;
xy1 = u2;

rgb1 = impixel(image1, xy1(:, 1), xy1(:, 2)); % (gt)
rgb2 = impixel(image2, xy2(:, 1), xy2(:, 2));

% Extract RGB values
% Data preprocessing
rgb1 = double(rgb1);
rgb2 = double(rgb2);

% For R channel
R1 = rgb1(:, 1);
G1 = rgb1(:, 2);
B1 = rgb1(:, 3);

R2 = rgb2(:, 1);
G2 = rgb2(:, 2);
```

```
B2 = rgb2(:, 3);

%% R Channel
% Fit using least squares
X = [R1, ones(size(R1))];
paramsR = X \ R2;
% Extract fit results
A_R = paramsR(1);
B_R = paramsR(2);

%% G Channel
% Fit using least squares
X = [G1, ones(size(G1));
paramsG = X \ G2;
% Extract fit results
A_G = paramsG(1);
B_G = paramsG(2);

%% B Channel
% Fit using least squares
X = [B1, ones(size(B1));
paramsB = X \ B2;
% Extract fit results
A_B = paramsB(1);
B_B = paramsB(2);

%%
% Apply the best-fit models
% Restore the color of image2

restored_R = (image2(:,:,1) - B_R) / A_R;
restored_G = (image2(:,:,2) - B_G) / A_G;
restored_B = (image2(:,:,3) - B_B) / A_B;

% Convert the restored RGB values to 8-bit integers
restored_rgb2 = uint8(cat(3, restored_R, restored_G, restored_B));

% Display the restoration results
figure;
subplot(1, 2, 1);
imshow(image1);
title('Original Image 1');
subplot(1, 2, 2);
imshow(restored_rgb2);
title('Image 2');
```

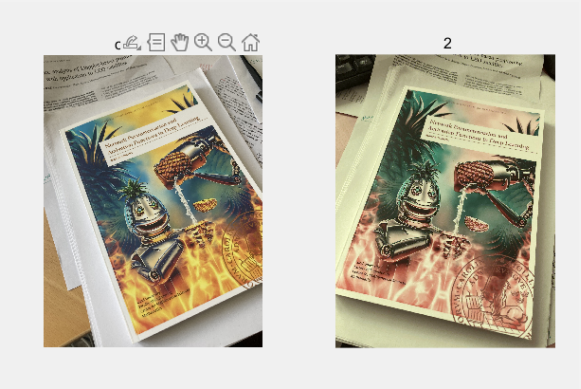

Figure 11: LS

Figure 12: RANSAC

b) I used a multivariate linear regression model to fit the data.

```
clc;
clear;
close all;
load('imdata_pineapple.mat')

% Step 1: Load two images
image1 = imread('pineapple2.jpg'); % Load the first image (gt)
image2 = imread('pineapple_t2.jpg'); % Load the second image

xy1 = u2;
xy2 = u1;

x = xy2(:, 1);
y = xy2(:, 2);

rgb1 = impixel(image1, xy1(:, 1), xy1(:, 2)); % (gt)
rgb2 = impixel(image2, xy2(:, 1), xy2(:, 2));

% Extract RGB values
R1 = rgb1(:, 1);
G1 = rgb1(:, 2);
B1 = rgb1(:, 3);

R2 = rgb2(:, 1);
G2 = rgb2(:, 2);
B2 = rgb2(:, 3);

% Data preprocessing
rgb1 = double(rgb1);
rgb2 = double(rgb2);

%% Red Channel

% Combine data into a table
T_R = table(R2, x, y, R1, 'VariableNames', {'R2', 'x', 'y', 'R1'});

% Perform polynomial fitting
degree = 1; % Choose the degree of the polynomial (1 for linear)
coeff = polyfit([T_R.R2, T_R.x, T_R.y], T_R.R1, degree);

% Extract coefficients
A_R = double(coeff.Coefficients(1));
B_R = double(coeff.Coefficients(2));
C_R = double(coeff.Coefficients(3));
D_R = double(coeff.Coefficients(4));

%% Green Channel

% Combine data into a table
T_G = table(G1, G2, x, y, 'VariableNames', {'R1', 'R2', 'x', 'y'});
```

```
51
52  % Use multivariate linear regression to fit the data
53  lm_G = fitlm(T_G, 'R1 ~ R2 + x + y');
54
55  % Get regression coefficients
56  Coefficients = lm_G.Coefficients.Estimate;
57
58  % Coefficients contain the estimated values for A, B, C, and D
59  A_G = double(Coefficients(2));
60  B_G = double(Coefficients(3));
61  C_G = double(Coefficients(4));
62  D_G = double(Coefficients(1));
63
64  %% Blue Channel
65
66  % Combine data into a table
67  T_B = table(B1, B2, x, y, 'VariableNames', {'R1', 'R2', 'x', 'y'});
68
69  % Use multivariate linear regression to fit the data
70  lm_B = fitlm(T_B, 'R1 ~ R2 + x + y');
71
72  % Get regression coefficients
73  Coefficients = lm_B.Coefficients.Estimate;
74
75  % Coefficients contain the estimated values for A, B, C, and D
76  A_B = double(Coefficients(2));
77  B_B = double(Coefficients(3));
78  C_B = double(Coefficients(4));
79  D_B = double(Coefficients(1));
80
81  % Apply the best models to restore the color of image2
82  image_width = 3024;
83  image_height = 4032;
84
85  X = zeros(image_height, image_width);
86  Y = zeros(image_height, image_width);
87
88  % Fill the X and Y matrices
89  for i = 1:image_height
90      for j = 1:image_width
91          X(i, j) = double(j); % Horizontal coordinate
92          Y(i, j) = double(i); % Vertical coordinate
93      end
94  end
95
96  restored_R = (double(image2(:,:,1)) - B_R * X - C_R * Y - D_R) / A_R;
97  restored_G = (double(image2(:,:,2)) - B_G * X - C_G * Y - D_G) / A_G;
98  restored_B = (double(image2(:,:,3)) - B_B * X - C_B * Y - D_B) / A_B;
99
100 % Convert the restored RGB values to 8-bit integers
101 restored_rgb2 = uint8(cat(3, restored_R, restored_G, restored_B));
102
103 % Display the restored image
104 figure;
105 subplot(1, 2, 1);
106 imshow(image1);
107 title('Original Image 1');
108 subplot(1, 2, 2);
109 imshow(restored_rgb2);
110 title('Restored Image 2');
```

The result show as follow.

Figure 13: Multivariate Linear Regression

Table 1: Least Square.

	λ	γ
R	1.5793	-107.98
G	1.0853	-31.43
B	1.3765	-45.64

Table 2: RANSAC.

	λ	γ
R	4.8667	86.7237
G	3.5744	87.4564
B	18.4537	78.3745

Table 3: Multivariate Linear Regression.

	λ	α	β	γ
R	4.65487	-0.0435	-0.1875	0
G	0.5845	0.0056	0.0146	0
B	1.7642	-0.0934	0.0278	0

Answer to Task 6

For this task, I used SIFT to extract feature points and then employed the FLANN (Fast Library for Approximate Nearest Neighbors) matcher for feature point matching.

After filtering the suitable matches, I used these matches with the RANSAC algorithm to estimate the transformation matrix. RANSAC algorithm is effective in removing outliers, which helps eliminate some of the less accurate data in feature matching.

By using the transformation matrix to map the ground truth facial feature coordinates from 'cat_train,' I could obtain the facial feature coordinates in 'cat_test.'

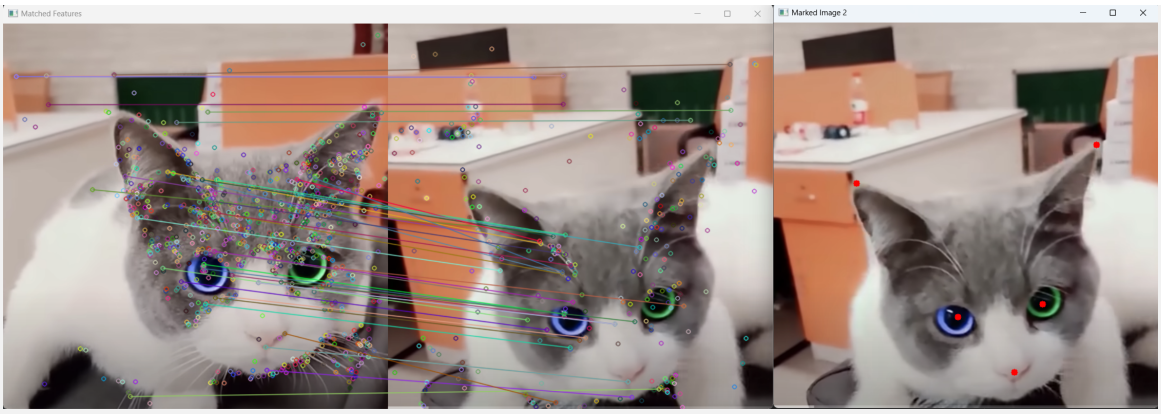

Figure 14: cat test1

```
Point 1 in Image 1: (184.00, 144.00)
Transformed Point in Image 2: (129.75, 250.71)
Point 2 in Image 1: (586.00, 130.00)
Transformed Point in Image 2: (503.34, 190.26)
Point 3 in Image 1: (326.00, 386.00)
Transformed Point in Image 2: (287.84, 458.61)
Point 4 in Image 1: (474.00, 376.00)
Transformed Point in Image 2: (419.10, 438.59)
Point 5 in Image 1: (412.00, 492.00)
Transformed Point in Image 2: (375.21, 544.68)
```

Figure 15: cat test1 point

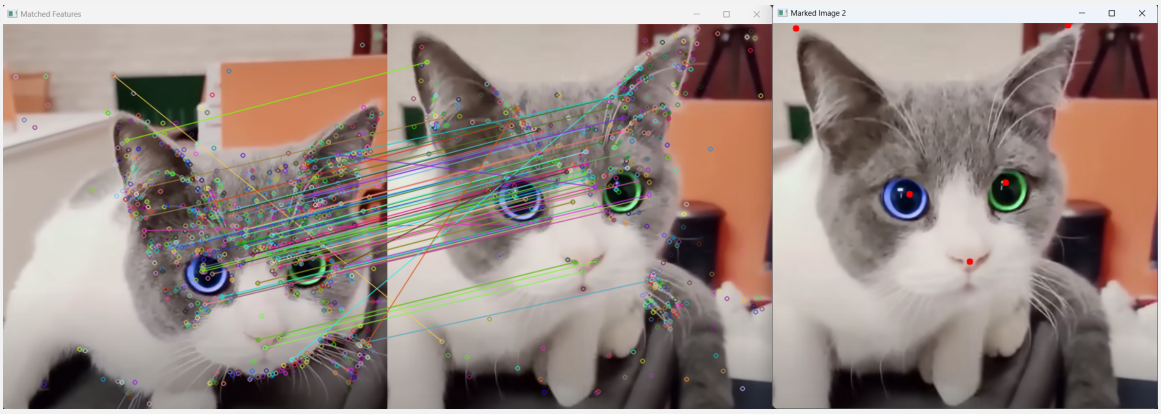

Figure 16: cat test2

```
Point 1 in Image 1: (184.00, 144.00)
Transformed Point in Image 2: (36.06, 8.85)
Point 2 in Image 1: (586.00, 130.00)
Transformed Point in Image 2: (460.19, 3.95)
Point 3 in Image 1: (326.00, 386.00)
Transformed Point in Image 2: (213.90, 266.06)
Point 4 in Image 1: (474.00, 376.00)
Transformed Point in Image 2: (363.43, 248.58)
Point 5 in Image 1: (412.00, 492.00)
Transformed Point in Image 2: (307.46, 371.75)
```

Figure 17: cat test2 point

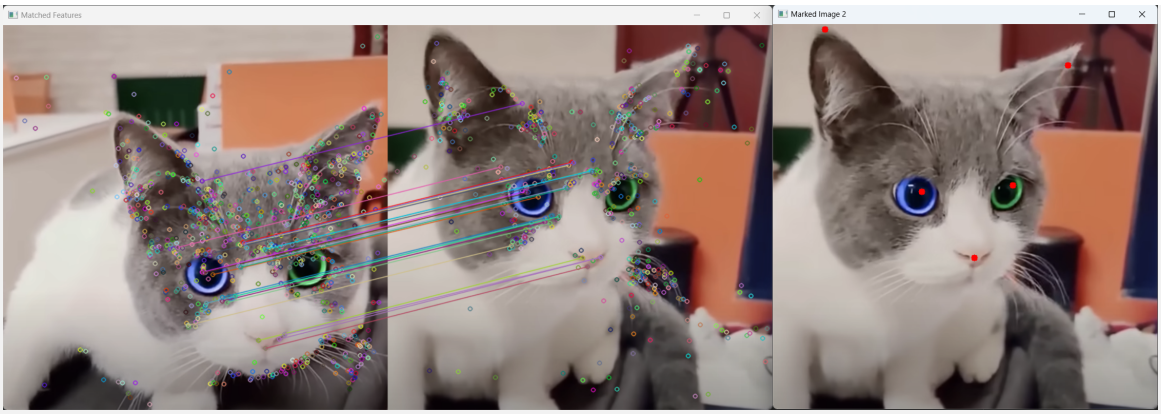

Figure 18: cat test3

```
Point 1 in Image 1: (184.00, 144.00)
Transformed Point in Image 2: (81.75, 8.49)
Point 2 in Image 1: (586.00, 130.00)
Transformed Point in Image 2: (460.54, 64.05)
Point 3 in Image 1: (326.00, 386.00)
Transformed Point in Image 2: (232.23, 261.92)
Point 4 in Image 1: (474.00, 376.00)
Transformed Point in Image 2: (374.42, 251.67)
Point 5 in Image 1: (412.00, 492.00)
Transformed Point in Image 2: (314.34, 364.47)
```

Figure 19: cat test3 point

```python
# -*- coding: utf-8 -*-
"""
@Time: 2023/10/23 14:39
@Author: Dexter ZHANG
@File: sift.py
@IDE: PyCharm
"""

import cv2
import numpy as np

# Read two consecutive images
image1 = cv2.imread('cat_train.png')
image2 = cv2.imread('cat_test3.png')

grayImage1 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
grayImage2 = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)

# Initialize the SIFT detector
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors on both images
keypoints1, descriptors1 = sift.detectAndCompute(grayImage1, None)
keypoints2, descriptors2 = sift.detectAndCompute(grayImage2, None)

# Use the FLANN (Fast Library for Approximate Nearest Neighbors) matcher for ...
    feature point matching
index_params = dict(algorithm=0, trees=10)
search_params = dict(checks=50, crossCheck=True)
flann = cv2.FlannBasedMatcher(index_params, search_params)
matches = flann.knnMatch(descriptors1, descriptors2, k=2)

# Select good matches
```

```
good_matches = []
for m, n in matches:
    if m.distance < 0.7 * n.distance:
        good_matches.append(m)

# Extract the coordinates of matching points
src_pts = np.float32([keypoints1[m.queryIdx].pt for m in ...
    good_matches]).reshape(-1, 1, 2)
dst_pts = np.float32([keypoints2[m.trainIdx].pt for m in ...
    good_matches]).reshape(-1, 1, 2)

# Connect the matching feature points
matched_image = cv2.drawMatches(image1, keypoints1, image2, keypoints2, ...
    good_matches, None)

# Estimate the transformation matrix using the RANSAC algorithm
M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

# Input coordinates on Image 1
input_points = np.array([[184, 144], [586, 130], [326, 386], [474, 376], [412, ...
    492]], dtype=np.float32).reshape(-1, 1, 2)

# Transform the coordinates from Image 1 to Image 2 using the transformation matrix
output_points = cv2.perspectiveTransform(input_points, M)
output_points = np.abs(output_points)

# Print coordinate values
for i, point in enumerate(output_points):
    x, y = point[0]
    print(f"Point {i + 1} in Image 1: ({input_points[i][0][0]:.2f}, ...
        {input_points[i][0][1]:.2f})")
    print(f"Transformed Point in Image 2: ({x:.2f}, {y:.2f}")

# Mark the coordinates on Image 2
for point in output_points:
    x, y = point[0]
    cv2.circle(image2, (int(x), int(y)), 5, (0, 0, 255), -1)

# Save the marked image
cv2.imwrite('marked_image2.jpg', image2)

# Display the matched features and the marked Image 2
cv2.imshow('Matched Features', matched_image)
cv2.imshow('Marked Image 2', image2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```