IVR Coursework 1
Report

Weiting Goh (S1450710) and Tomas Markevicius(S1452595)

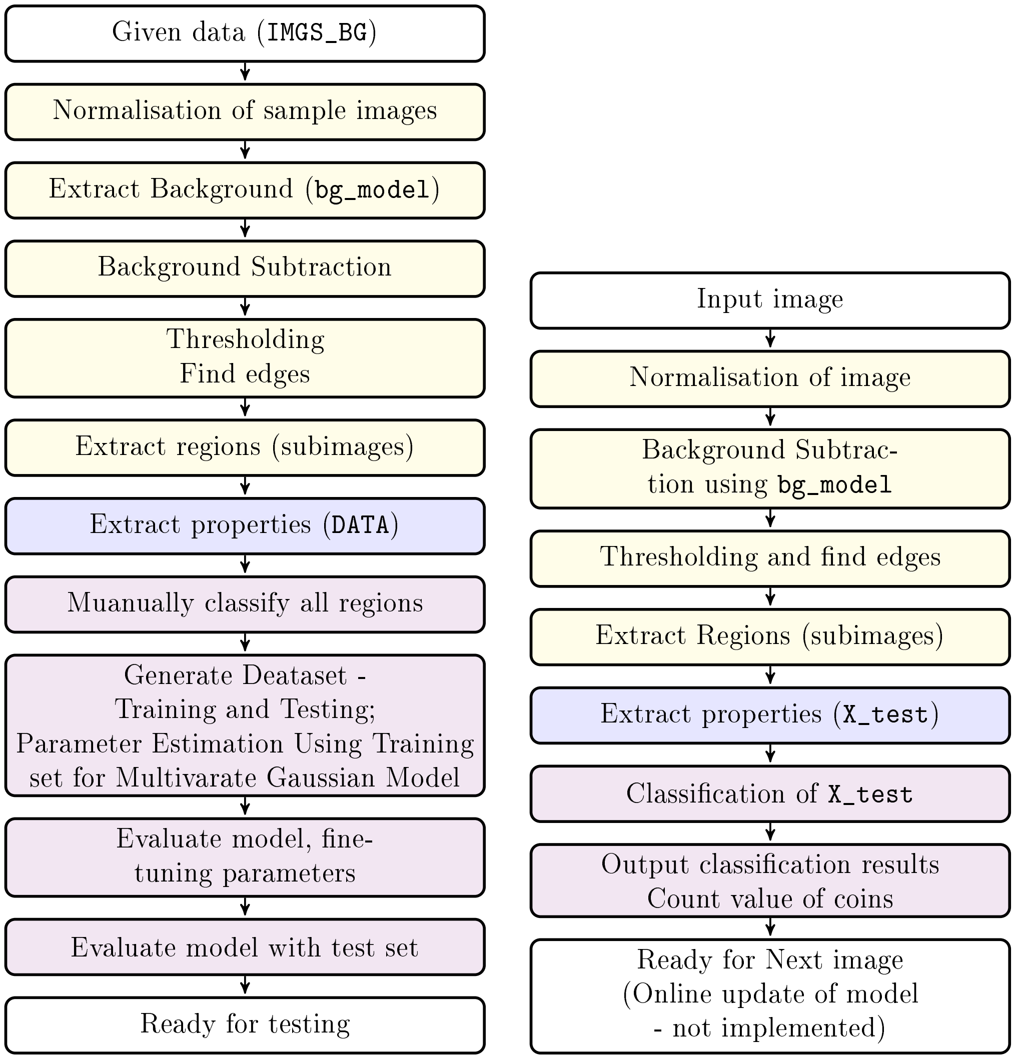
October 27, 2016

1 Introduction

To recognise and count the coins in an image, Coinsy, have three subtasks: 1) Image processing (includes image segmentation), 2) Features extraction, and 3) Classifiation (and then counting the coins).

For Coinsy to be a proficient detector, classifier and counter, we have to first train it. The training processes differs from evaluation as described in figure Figure 1 - the operation pipeline for training and evaluation. We will describe the methods that we use for each of these subtasks in the next section - methodology, and her evaluation results after. Lastly, we conclude with a discussion on Coinsy performance. The code for the project is listed in the appendix.

In the following subsections, we give an brief description of the operation pipeline for each subtasks from training to testing. The approach here is an abstract idea of what we did for our base model. Additional techniques were explored and will be discussed in later section. We start by understanding the data.



(a) Operation pipeline to train Coinsy

(b) Pipeline for (trained) Coinsy to count coins

Figure 1: Differences in Pipeline; Yellow boxes indicates the image processing procedures; Blue indicates the feature extraction procedures; Violet indicates the classification procedures.

Data

The data we were given are 14 images (consisting of 5 `harder` and 9 `simpler images`). Each consists of objects in the foreground for us to classify and calculate the values. The class labels and its respective object and value are as follow:

Class	Object	Value
1	1 Pound coin	1 pound
2	2 Pound coin	2 pound
3	50 Pence coin	50 pence
4	20 Pence coin	20 pence
5	5 Pence coin	5 pence
6	Washer with small hole	75 pence
7	Washer with large hole	25 pence
8	Angle bracket	2 pence
9	AAA Battery	-
10	Nut	-
11	unknown	-

The objects differ in colors, size and shapes; some are very similar to the background - such as 1 pound coins (see Figure 2). Hence, the features extracted must be invariant to rotation, and importantly to detect the objects will the images to be processed, such that the objects are salient to the computer vision. On top of the images itself output from `imshow` or `imagesc`, we can understand an image from the distribution of the pixel intensities (such as a histogram) and its gradient magnitude in each channel.

Figure 2: Sample images from given data.

Figure 3: Sample images after their background is subtracted.

Image Processing

The image processing step aims to 1) make all images (for training the model or for evaluation) comparable, 2) extract objects in the foreground (also known as image segmentation). The outcome of this stage is an array of subimages ready for feature extraction.

The central idea is to, first, model the background before subtracting it from all images; second - make the edges of the objects 'obvious' by finding a suitable threshold to binarise the image; third - crop out the objects to obtain the subimages, which will be our data points.

Feaure Extraction

Given the subimages, this stage aims to represent each subimages with a feature vector that contains properties to adequately describe the class it belongs to (or shape). Notaby, we have many circular objects varying in size. This renders global descriptors such as convexity and elongation less useful for these classes.

Classification

The task here is to label an unknown (sub)image given its set of feature vector. The model we consider is a multivarate gaussian classifier that classify an image based on the parameters of each class (class mean and standard deviation (or variance)). This multivarate gaussian classifier outputs the posterior probability of a given feature vector x ($P(C_k|X)$) for each class k, with the class giving the highest probability being the label. That is $x.class = argmax_k P(C_k|x)$.

A slight tweak for Coinsy is her ability to reject the class label output by the classifier if the highest probability falls out of her confidence interval. In such cases, Coinsy will intervene and change the class to unknown (class 11).

The last step of Coinsy is to sum the value of all the objects she managed to identify from a given image.

Code

The following directory trees will provide an overview of the code utilitsed for the project. Codes presented in the appendix are hyperlinked, although some may depend on the code repository given in `http://www.inf.ed.ac.uk/teaching/courses/ivr/matlab/flatpartrecog/`.

```
src
├── imgs ................................ Store images from function
├── dataset ............................. Datasets from experiments
├── setup
├── training
├── main
├── image_processing
├── gradmag_edge
├── extract_features
├── manual_classification
├── trainclf_loglikelihood
└── filters
    ├── median_filter
    ├── median_filter_iter
    ├── gaussian_filter_1d
    └── gaussian_filter_2d
```

- .2 image_processing
    - normalise_RGB
    - bg_extract
    - bg_subtraction
    - dothresh
    - edg_morph_disk
- features
    - getFeatures
    - rawmmoment
    - centralmoment
    - SI_moment
    - humomentinvariants
    - complexmoment
- classification
    - split_data
    - user_classify
    - findConfusion
    - gaussianDistr
    - multivarate

2 Methodology

In this section we describe the techniques we considered, brainstormed during the project and those that are implemented. Table 1 is an effort to summarise all these techniques, but they are not exhausive. This is because we realised that the order of applying these techniques will have varying impact on subsequent stages of the task. To further complicate the matter, filtering the images with a gaussian filter, median filter, or 'morphing' the images with a structuring element (such as a disk with 15 pixels) before or after each stage in the pipeline will have varying impact to the outcome. The number of permutation in these steps is too large for us to consider all. What we have implemented is not the optimum, but one of the many options. We rationalise the choice of parameters along the course of this section.

The reader should refer to training.m, image_processing.m, extract_features.m, manual_classification.m, and trainclf_loglikelihood.m.

Normalisation and Background Modelling

In this coursework, since a background image is not readily available, we have to model it. Noting that images varies in illumination, we have to make the images comparable by normalising it first, using the following formula.

$$P_{r,c}(R', G', B') = (\frac{R}{\sqrt{(R^2+G^2+B^2)}}, \frac{G}{\sqrt{(R^2+G^2+B^2)}}, \frac{B}{\sqrt{(R^2+G^2+B^2)}})$$

This is executed by normalise_RGB.m.The outcome of the background with and without normalisation is shown in Figure 4.

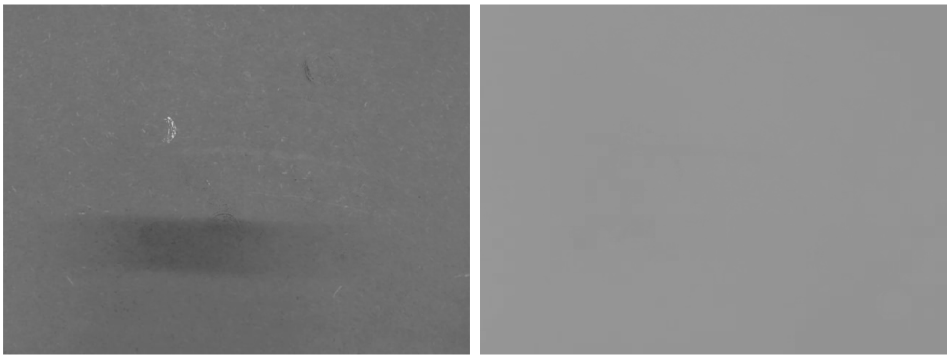
Next, with objects scattered around randomly in the images, we find the median of all image pixels for each channel separately in order to reconstruct the background. Our approach uses a neighbourhood of pixels for each pixel in the backgrund model. Hence, for a window of size 3, we have $bg\_model_{r,c} = median(i_{r+1,c+1}, i_{r+1,c}, i_{r+1,c-1}, i_{r,c+1}, i_{r,c}, i_{r,c-1}, i_{r-1,c+1}, i_{r-1,c}, i_{r-1,c-1})$.

Task	Subtask	Techniques	Implmentation
Image Processing	Background Model	Finding median for each pixel and each channel	✓
		Finding median of neighbourhood for each pixel and each channel	✓
		Probabilistic Modeling of background	
	Thresholding	Finding minimum of bimodal distribution	✓
		Otsu's method for thresholding	
		Using Gradient magnitude of image	
Feature Extraction	Global Descriptors	• Area • Perimeter • Compactness • Rectangularity • Elongation	✓
	Moments	• Hu's Invariant moments (7 features) • Complex moments (6 features)	✓
Classification	-	Multivarate Gaussian Model	✓
		Linear Discriminant	

Table 1: Summary of techniques in Coinsy.

The outcomes of the background (with and without normalisation) with different neighbourhood size = 1, 3, 5 are shown in Figure 5. This appraoch requires large amount of memory and time to compute the background model. We find that although the images have subtle differences, the subimages we derieved in the later part of the pipeline is actually better.

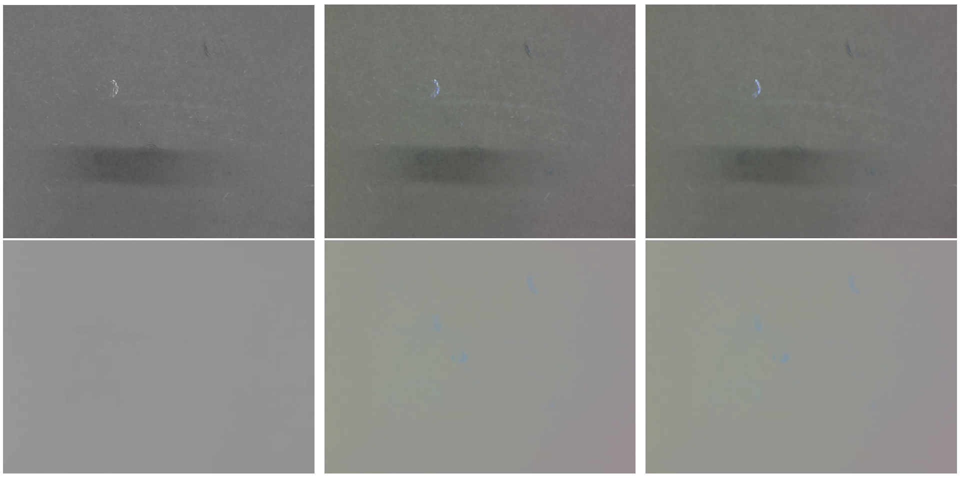
The sample images with their background removed is shown in Figure 3. This removal process: $img\_bg\_removed_{r,c}(R, G, B) = img_{r,c}(R, G, B) - bg\_model_{r,c}(R, G, B)$, however, also inevitably reduce the intensity for the



(a) Background model without normalisation

(b) Background model after normalisation

Figure 4: Background model generated from all 14 images.



(a) Neighbouhood=1

(b) Neighbourhood=3

(c) Neighbourhood=5

Figure 5: Background extraction with different neighbourhood size.

bottom half of each images, such that the objects are no longer salient. This is because the background we modelled have a lower intensity at the bottom, possibly due to presence of shadow in all 14 images.

An alternative approach we considered is to find the average of the neighbourhood. However, we did not materialise this, as the presence of high pixel intensity (such as in the presence of an object in the foreground) will distort the mean, giving an inaccurate representation of the background.

Segmentation

After the background is removed, the objects are left in the image. Our next task is to extract these regions where the objects exists. We used dothresh.m on each image to:

1. Find the histogram of the pixel intensity
2. Find the threshold of the histogram - `thresh`
3. Apply this threshold value to the image (i.e. $if\ IMG_{i,j} \geq thresh,\ IMG_{r,c,chn} = 1,\ else\ IMG_{r,c,chn} = 0$
4. Produce a binary image, `BW`, using:

$$BW_{r,c} = IMG_{r,c,R} || IMR_{r,c,G} || IMG_{r,c,B}$$

The binary image indicates the existence of objects as 1 (white) and the background as 0 (black). Figure 10 are the binary images for the sample images.

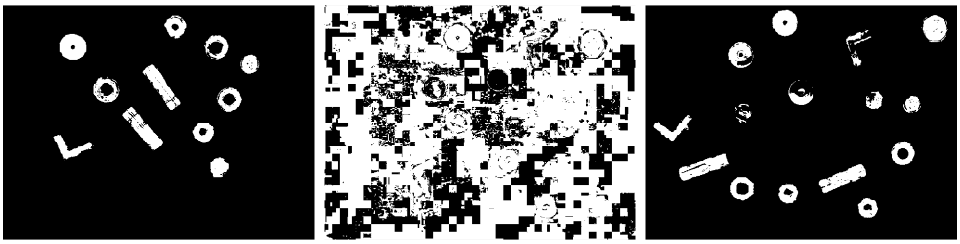


Figure 6: Black white images of normalised sample images with background removed.

Morphological Gradient Edge Detection

Another model that we tried out, which performed better, is the morphological gradient edge detector (gradmag_edge.m). Using a structuring element B (such as a disk with 5 pixel radius), we first find the image A dilated with

B and image A eroded with B separately. Then we subtract the two. Since the background is largely the same for both outcomes, the subtraction of each other will remove the background and retain the edges of each objects in the foreground.

We used `matalb` built in function to createa structuring element with disk radius of 3 pixels, and dilate and erode a given image. After the subtraction, we use the same thresholding function to get the binary image. The outcome of this operation is shown in Figure 7.

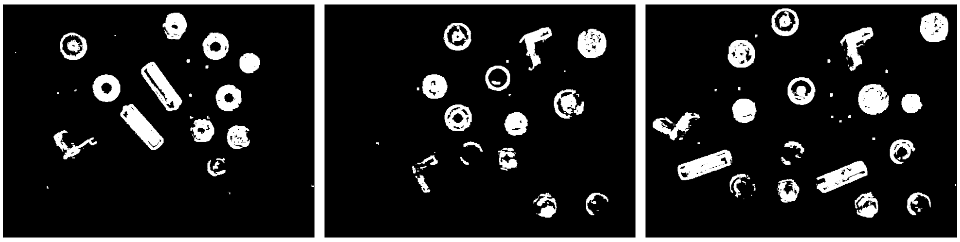


Figure 7: Black white images of sample images after morphological gradient edge detection and thresholding. As the operator is background independent and does not require us to model the background first, the objects are more obvious now. And the centre sample image does not appaer to be a mess.

The difference in result of extractiing images for this two different segmentation methods will be revealed in result section later.

Filtering

There are numerous filtering technique to make images better. One that we considered during the project is median filtering, and iterative median filtering. The aim is to preserve the edges while removing the noise in the background by find the median of the neighbourhood of the pixel. With the latter, the image undergoes any number of iteration until no visible change is observed. In our function, we utilised the `matlab` function `medfilt2`.

However, we realised that despite the changes in the edges, normalising an image is the main cause of a bad threshold. We are unable to find the threshold, as the gaussian smoothing operator in `findthresh` causes the bimodal peaks in the histogram of to become a unimodal peak. In this case, finding a useful threshold is futile.

2.1 Feature Extraction

The end product of the previous stage leaves us with an array of subimages. These subimages are black and white, such as those in Figure 8. The output from this section is to have an array of features that uniquely defined each classes of objects defined above. We chose the following features:

1. Area
2. Perimeter
3. Compactness
4. Rectangularity
5. Elongation
6. Hu's invariant moments
7. Complex invariant moments

We later realised that the number of features is too large for our classifier, and casuse the covariance matrix to be almost singular. Although we regularised the matrix, we decided to remove Hu's invariant moments from the features, leaving us with 11 features to describe an object. Hence, for each subimages, there will be a feature vector of size 11.

2.2 Classification

After extracting the features, split_data.m will be called to split the dataset into the training set (`X_train` and `y_train`) and test set (`X_test`) and

Figure 8: Some subimages of the objects detected by `matlab` function `bwlabel`. It first find region where 1s are and give them a numeric label. Since pixel belonging to an object will come together, the numeric clsas will uniquely identify the object.

`y_test`). Then, trainmultivarate.m will estimate the parameters based on the `X_train` - the `prior`, `covariance` and `mean` for each class will be estimated.

With these class parameters, the gaussianDistr.m and gaussian_clf.m will be called to estimate the posterior probability for each object in `X_test`.

Linear Discriminant - average covariance for all classes

We consider the use case of a Linear Discriminant function. Howwever, on second though, the idea that all objects (classes) having the same covariance does not quite make sense. Since the features describe the data, the covariance of the feature set should be unique to all the object.

The classification step outputs the confusion matrix for the data, such as this one when the multivarate classifier is trained with 75 percent of the data and 25 percent of it for the test data.

```
  ========================================================
>> trainmultivarate
Done!

The confusion matrix is:
(rows = actual class; columns = predicted class)
```

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	3	0	0	0	0	0	0	0	0
0	0	0	2	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	1	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	2	2	0
0	0	0	0	0	0	0	0	1	0
0	1	0	0	1	0	0	0	0	2

```
The classification results for each class are:
   (FN    FP    TP   TN)
     0     0     0   19
     0     4     0   15
     3     0     0   16
     1     0     2   16
     0     2     1   16
     1     0     1   17
     0     0     1   18
     2     1     2   14
     0     2     1   16
     2     0     2   15
=======================================================
Summary:
Classification using full gaussian model
Number Incorrect = 18
Number Correct = 10
Number of classes = 10
Accuracy = 0.526316 percent
=======================================================
```

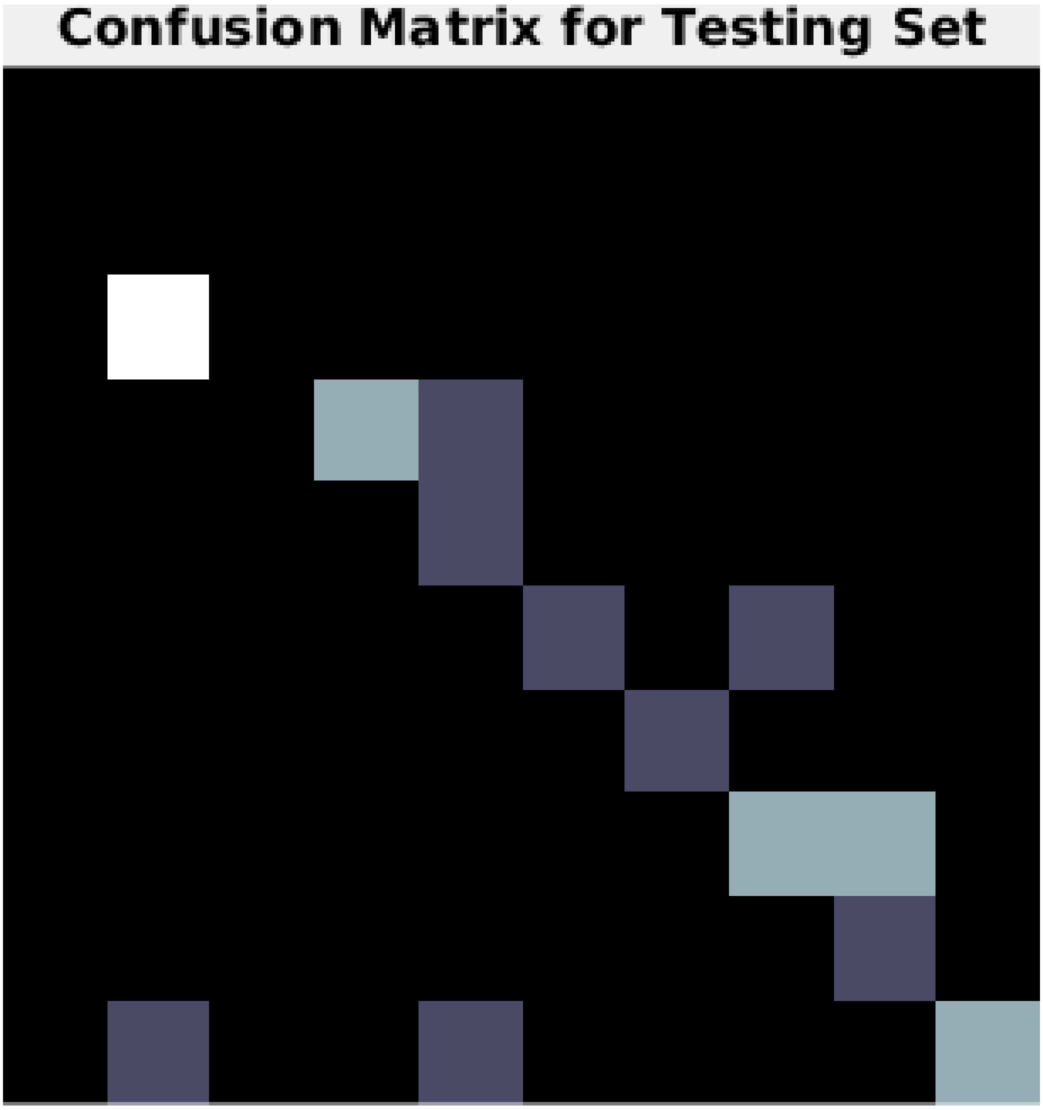


Figure 9: Confusion Matrix, with rows representing actual class and columns representing predicted class. A brighter white indicates a higher number of true positive. The class 1 (1 pound coin) have no boxes lit up, signifying that this test set does not have any 1 pound coin in it

3 Result

In this section, we describe the training and test result for our multivarate classifier, using the gradient magnitude segmentation method in the previous section. When we manually classify the images, we output a colored border signifying the classes of the object.

Figure 10: There difference in classification using normalised background (right) and gradient magnitude segmentation (left) is that one pound coins are almost always gone in the former, while found in the later .

documentclass[main.tex]article

A Scripts

In this section, all the scripts used to call other scripts and/or functions are presented. Some fuunctions are matlab's native functions.

training.m

```
%% MASTER SCRIPT USE FOR TRAINING
% Steps:
%   1.  setup
%       a. load images and IMGS_BG (for bg modelling), IMGS (all simpler images)
%
%   2.  image_processing
%       a.  normalisation          (normalise_rgb)
%       b.  background model       (bg_extract)
%       c.  background subtraction (bg_subtraction)
%       d.  thresholding           (dothresh)
%
%   3.  extract_features
%       a.  regionprops
%       b.  getFeatures
%           —   rawmoment,
%           —   centralmoment,
%           —   complexmoment,
%           —   SI_momment,
%           —   humomentinvariant
%
%   4.  Classification
%       a.  manual_classifcation
%           —   user_classify
%       b.  trainclf_loglikelihood
%           —   split_data
```

```
%           —   gaussianDistr, gaussian_clf, logdet
%           —   findConfusion
%%
clear all; close all; clc;
setup % import images
% START TRAINING:
image_processing;
extract_features;  % OUTPUT DATA!

%% Init Manual Classification
man_class = input('do you want to manually classify these images now? [0/1]');
if man_class
    manual_classification;
end

%%
```

setup.m

```
%% START CODE:
clc,clf,clear all; close all;

% add all relevant folders && misc stuff
addpath('filters/', 'image_processing/', 'classification/', ...
            'dataset/', 'imgs/', 'features');
addpath('../misc/export_fig.package/');

bar = '=========================================================';
barbar = '_________________________________________________________';

```

```
disp(bar);
fprintf('\t\tIMPORTING IMAGES\n');
% add all given images for traiing
% ? SHOULD we add the harder ones too?
img2    = imread('../practice/simpler/02.jpg');
img3    = imread('../practice/simpler/03.jpg');
img4    = imread('../practice/simpler/04.jpg');
img5    = imread('../practice/simpler/05.jpg');
img6    = imread('../practice/simpler/06.jpg');
img7    = imread('../practice/simpler/07.jpg');
img8    = imread('../practice/simpler/08.jpg');
img9    = imread('../practice/simpler/09.jpg');
img10   = imread('../practice/simpler/10.jpg');
IMGS    = {img2, img3, img4, img5, img6, img7, img8, img9, img10};

img11   = imread('../practice/harder/17.jpg');
img12   = imread('../practice/harder/18.jpg');
img13   = imread('../practice/harder/19.jpg');
img14   = imread('../practice/harder/20.jpg');
img15   = imread('../practice/harder/21.jpg');

IMGS_BG = {img2, img3, img4, img5, img6, img7, img8, img9, img10, ...
            img11, img12, img13, img14, img15 };
% IMGS    = {img2, img3, img4, img5, img6, img7, img8, img9, img10, ...
%             img11, img12, img13, img14, img15 };
[~, num_img_bg]  = size(IMGS_BG);
fprintf('\t\t\t\t\t\t    done\n');
%%
tmp = input('Continue? [1/0] ');
if ~tmp
   return
```

```
end
disp(bar);
```

extract_features.m

```
%% Script for feature extraction
%
% This script follows naturally from the segmentation script where the
% images are segmented and edges are found. The next step in the operation
% pipeline is then to find the obejects in the picture, then extract
% the features from the objects
%
% assume you have done called segmentation and the following are in
% the workspace:
% 1) bg_model       : the background model we generated
% 2) IMGS           : the original images (in cell array)
% 3) IMG_BGREMOVE   : the original iamges bg recmoved
% 4) IMG_THRESH     : the BW images thresholded. The objects are in white/1

%%
[~, num_imgs] = size(IMGS_THRESH);
PROP ={}; % define an array to hold the structs for each images
DATA = struct(); % struct to hold all the subimages
num_instance = 1; % counter for number of instances

% iterate through all the images to extract the subimages and its properties
for i=1:num_imgs
```

```
    fprintf('image %d ',i);

    % here, get the label from the threshold image, and extract information
    % about each region
    [L, ~]      = bwlabel(IMGS_THRESH{i}, 4); %% THIS IS A PARAMETER TO PLAY WITH
    imagery     = regionprops(L, 'BoundingBox','Image'); % this is the BW image!
    scalar      = regionprops(L, 'MajorAxisLength', 'MinorAxisLength', 'Area');

    % remove regions with small pixel area, which may be blobs:
    bad = [scalar.Area] <= 300;
    scalar(bad)     = []; % remove these instances
    imagery(bad)    = [];
    disp('prune - Area<=300'); %% DEBUG

    [num_subimages , ~] = size(imagery); % update the number of instances left!

    % grab the colored subimages, and calculate the complex moments,..etc,
    % for ease of classification:
    for n=1:num_subimages

        org_img     = IMGS{i}; % get the original image
        boundary    = imagery(n).BoundingBox; % find the boundary
        subImg      = imcrop(org_img, boundary); % crop the original image according to boundary

```

```
        % calculate the moments by calling classification/getProperties
        DATA(num_instance).Features         = getFeatures(imagery(n), scalar(n));
        DATA(num_instance).ColoredImage     = subImg;
        DATA(num_instance).BoundingBox      = imagery(n).BoundingBox;
        DATA(num_instance).Image            = imagery(n).Image;
        DATA(num_instance).MajorAxisLength  = scalar(n).MajorAxisLength;
        DATA(num_instance).MinorAxisLength  = scalar(n).MinorAxisLength;
        DATA(num_instance).ParentID         = i;
        DATA(num_instance).Class            = 0; % set to 0 = unclassified

        fprintf('%d  ',num_instance);
        num_instance = num_instance + 1;
    end

    % store in struct
    PROP{i} = struct('label', L, ...
                'num_of_obj', num_subimages, ...
                  'ORIGINAL', IMGS{i},...
                    'THRESH', IMGS_THRESH{i},...
                 'SubImages', imagery,...
                'Properties', scalar);

    fprintf('\t\tDone\n');
end

% clear boundary;
% clear imagery;
% clear scalar;
```

```
%%
```

image_processing.m

```
%% START CODE:
clc,clf,clear all; close all;

% add all relevant folders && misc stuff
addpath('filters/', 'image_processing/', 'classification/', ...
            'dataset/', 'imgs/', 'features');
addpath('../misc/export_fig.package/');

bar = '
   =========================================================';
barbar = '
   _________________________________________________________';

disp(bar);
fprintf('\t\tIMPORTING IMAGES\n');
% add all given images for traiing
% ? SHOULD we add the harder ones too?
img2    = imread('../practice/simpler/02.jpg');
img3    = imread('../practice/simpler/03.jpg');
img4    = imread('../practice/simpler/04.jpg');
img5    = imread('../practice/simpler/05.jpg');
img6    = imread('../practice/simpler/06.jpg');
img7    = imread('../practice/simpler/07.jpg');
img8    = imread('../practice/simpler/08.jpg');
img9    = imread('../practice/simpler/09.jpg');
img10   = imread('../practice/simpler/10.jpg');
IMGS    = {img2, img3, img4, img5, img6, img7, img8, img9,
   img10};

```

```
img11   = imread('../practice/harder/17.jpg');
img12   = imread('../practice/harder/18.jpg');
img13   = imread('../practice/harder/19.jpg');
img14   = imread('../practice/harder/20.jpg');
img15   = imread('../practice/harder/21.jpg');

IMGS_BG = {img2, img3, img4, img5, img6, img7, img8, img9,
    img10, ...
            img11, img12, img13, img14, img15 };
% IMGS    = {img2, img3, img4, img5, img6, img7, img8, img9,
    img10, ...
%              img11, img12, img13, img14, img15 };
[~, num_img_bg]  = size(IMGS_BG);
fprintf('\t\t\t\t\t\t    done\n');
%%
tmp = input('Continue? [1/0] ');
if ~tmp
   return
end
disp(bar);
```

gradmag_edge.m

```
%% MORPHOLOGICAL GRADIENT EDGE DETECTION
% source : http://www.vlsi.uwindsor.ca/presentations/2007/13-
    Neil.pdf#15
%   do Edge finding by subtracting opened img with the closed
    image
%   following thresholding to detect edges
%%
clear all;
setup ; % load all the images
for i=1:9
```

```
    % performs edge detection using morphology with size of 3;
    edges_morph{i} = edge_morph_disk(IMGS{i});

    s = sprintf('./imgs/morph_thresh/edges_morph.%d.png',i);
    close all;
    figure;
    imshow(edges_morph{i})
    export_fig(s);
    close all;
end

% use standard thresholding technique to threhsold images
[edges_morph_BW, ~ ] = dothresh(edges_morph, 16);


for i=1:9
    % store images
    s = sprintf('./imgs/morph_thresh/edges_morph_thresh.%d.png',i);
    close all;
    figure;
    imshow(edges_morph_BW{i})
    export_fig(s);
    close all;
end

fprintf('Completed thresh holding edge morphed images');

%% Extract Features
%
IMGS_THRESH = edges_morph_BW;
extract_features;
```

manual_classification.m

```
%% Script for classification of subimages
%   USER CLASSIFY THE SUBIMAGES

%% Param
% Color for each class
cmap = [0.80369089,  0.61814689,  0.46674357;
        0.81411766,  0.58274512,  0.54901962;
        0.58339103,  0.62000771,  0.79337179;
        0.83529413,  0.5584314 ,  0.77098041;
        0.77493273,  0.69831605,  0.54108421;
        0.72078433,  0.84784315,  0.30039217;
        0.96988851,  0.85064207,  0.19683199;
        0.93882353,  0.80156864,  0.4219608 ;
        0.83652442,  0.74771243,  0.61853136;
        0.7019608 ,  0.7019608 ,  0.7019608
        244/255, 66/255, 66/255]; % Class 11
total_instance = 0;
total_relevant = 0;
t = datetime('now'); % for image title


%%
[~, num_instance] = size(DATA);
for i=1:num_instance % for each datapoint:

    img_num     = DATA(i).ParentID;
    img_BIG     = PROP{img_num}.ORIGINAL; % original big image
    subimg      = DATA(i).ColoredImage;
    bw_subimg   = DATA(i).Image;

    fprintf('\n\n\n\nObject %d/%d\n', i , num_instance);
    close all; figure; % Close all opened windows
```

```

    % Plot the images
    subplot(1,2,1);
    imshow(subimg);
    subplot(1,2,2);
    imshow(bw_subimg);

    % call function to for classification
    [relevance, class] = user_classify();
    close all;
    % if user need help, display the bigger image with a bounding box for object:
    while class == 0
        fig = figure;
        imshow(img_BIG);
        hold on;
        rectangle('Position', DATA(i).BoundingBox,... % draw rectangle around img
            'EdgeColor', 'r', 'LineWidth',3);
        [relevance, class] = user_classify();
        close all;
    end

    DATA(i).Class = class; % store the class; irrelevant ones at 11

    % SAVE THE IMAGE
    imshow(subimg);
    s = sprintf('./imgs/CLASS_%d/%s_%d.png', class, t, num_instance);
    export_fig(s);
    close;

end
```

```
%% DISPLAY and drawings
close all; figure;
imshow(img_BIG);
titl = sprintf('Classification for picture %d (%s)',i,t);
title(titl);
hold on;
[~,num_imgs] = size(PROP); % num of images

ID = [DATA.ParentID];
for i=1:num_imgs % draw the boundary box with differernt color for each image
    close all;

    list_ = ID == i; % logical
    data_class = DATA(list_);
    img_BIG = PROP{i}.ORIGINAL;
    img_BW = PROP{i}.THRESH;
    imshow(img_BW); hold on;
    for  n=1:sum(list_)  % draw the boundary on BW image
        boundary    = data_class(n).BoundingBox;
        class       = data_class(n).Class;
%         disp(cmap(class,:));
        rectangle('Position', boundary, 'EdgeColor', cmap(class,:), 'LineWidth', 2);
    end
    s = sprintf('./imgs/manual_classy/manual_clas_pic#%d_BW.(%s).png',i,t);
    export_fig(s);

    close all; % repeat for colored images
    imshow(img_BIG);
    for  n=1:sum(list_)  % draw the boundary on BW image
        boundary    = data_class(n).BoundingBox;
```

```
        class       = data_class(n).Class;
%         disp(cmap(class,:));
        rectangle('Position', boundary, 'EdgeColor', cmap(class
            ,:), 'LineWidth', 2);
        s = sprintf('./imgs/manual_classy/manual_clas_pic#%d_BW
            .(%s).png',i,t);
        export_fig(s);
    end
end


%% Delete Class 11 instances
class_list  = [DATA.Class];
logica_     = [class_list == 11];
DATA(logica_) = [];
[~,init_size]   = size(class_list);
[~,after_size]  = size(DATA);
fprintf('Number of datapoints removed (class 11) = %d\n',
    init_size - after_size);
```

trainmultivarate.m

```
%% SCRIPT FOR TRAINING MULTIVARATE GAUSSIAN CLASSIFIER
%   Assume you have done extract_features and
    manual_classification
%   PROP must be in your workspace

%% COMPACT ALL YOUR DATA:
[~, num_instance] = size(DATA);
[~, num_feature] = size(DATA(1).Features);

num_data = 0;
X = []; % Feature
```

```
y = []; % classes


% first, put all images together matrix
for im=1:num_instance
    X = [X; DATA(im).Features];
    y = [y; DATA(im).Class];
end
% y = reshape(y, [],1); % convert into col vector


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% % CLASS 1 is missing (NO POUND COIN DETECTED!)
% % CREATE BOGUS DATA:
% for w=1:4
%     y(num_instance+w) = 1;
%     X(num_instance+w,:) = [rand(1,num_feature)]; % randomly give some data!
% end
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Do hold-out validation:
% 50% for training, 25% for validation 25% for test
[X_train, X_valid, X_test, y_train, y_valid, y_test] = ...
```

```
    split_data(X, y, .80, 0, .20);

%% TRAINING THE CLASSIFIER:
% GROUP IN CLASS AND PARAMETER ESTIMATION:
classes      = unique(y);
num_class    = length(classes);
num_instance = length(X);

% Sort data into struct:
DATA_CLASS = {};

for i = 1:num_class % create a DATA_CLASS for each class
%     disp(i); %% DEBUG
    logica_     = [y_train == classes(i)];
    prior_      = sum(logica_)/num_instance;
    data        = X_train(logica_, :);
    mean_       = mean(data,1); % take the mean along the cols
    cov_        = cov(data,0); % number of observations -1; Maximmum posterior

    % Regularise COV:
    reg = exp(-10);
    reg_term = eye(length(cov_)) * reg;
    cov_ = cov_ + reg_term; % add regularisation
%     disp(cov_);

    % store the parameters
    DATA_CLASS{i} = struct('Data', data, 'Prior', prior_, ...
                            'Mean', mean_, 'Cov', cov_);
end

%% VALIDATION DATASET:
% [y_vali_pred, ~] = gaussian_clf(X_valid, DATA_CLASS);
%
```

```
% % Generate Statistics:
% [cm_valid, per] = findConfusion(y_vali_pred, y_valid);
% imshow(cm_valid, [], 'InitialMagnification', 1600); colormap(bone);
% title('Confusion Matrix for Validation Set');

%% Testing
% p_limit = 0;
[y_test_pred, prob] = gaussian_clf(X_test, DATA_CLASS);

% Generate Statistics:
[cm_test, per] = findConfusion(y_test_pred, y_test, 10);
%%
```

main.m

```
%% This is the main code for the assignment:
clc;
start = 1;
bar = '=========================================================';
barbar = '_________________________________________________________';

while start
%  Part 1: Reading the image, query from the user

    disp(bar); disp(barbar);
    fprintf('This is the coinsy counter!\nYour current work directory is: \n\t');
    disp(pwd); disp(barbar);
    fprintf('To END: enter cltr + c\n');
```

```
    prompt_start = 'To START: enter your image file (rel/abs
       dir) below:\n';

    filename = input(prompt_start, 's');
    if isempty(filename)
        disp('Using trial image: practice/simpler/05.jpg');
        filename = '../practice/simpler/05.jpg';
    end

    % load the image into original_image
    original_image = imread(filename);

    % display image
    imshow(original_image);
    s = sprintf('is this the correct image? [0/1] \n');
    yes = input(s);
    if yes
        tmp = input('Continue? [1/0] ');
    else
        fprintf('Lets try again...\n');
        disp(bar);
        return
    end
    fprintf('Continuing...\n');

    disp(barbar); disp(bar); fprintf('\n\n')

%%  Part 2: Image segmentation.... ?

    disp(bar); disp(barbar);
    disp('NOW: Segmenting the image...');
    fprintf('\n Using Morphological Gradient Edge Detection...\
       n');

```

```
    % Apply morphological gradient edge detection to it
    edges_morph_TEST{1} = edge_morph_disk(original_image);

    % save input
    s = sprintf('./imgs/testing/edges_morph.TEST.png');
    close all;
    figure;
    imshow(edges_morph_TEST{1})
    export_fig(s);
%     close all; % user will close!

    disp('NOW: Thresholding the image...');
    fprintf('\n Using findThresh...\n');

    % dothresholding on the image
    [IMGS_THRESH, thresh_vals ] = dothresh(edges_morph_TEST{1}, 16);
    disp(thresh_vals);

    s = sprintf('./imgs/testing/edges_morph_thresh.TEST.png');
    close all;
    figure;
    imshow(IMGS_THRESH)
    export_fig(s);
%     close all;

    tmp = input('Continue? [1/0] ');
    if ~tmp
        disp(bar);
        return
    end

    disp(barbar); disp(bar); fprintf('\n\n');

```

```
%% Part 3: Feature Extraction...?

    disp(bar); disp(barbar);
    disp('NOW: Extracting the features...');

    % call extract_features
    extract_features; % will output all the features

    tmp = input('Continue? [1/0] ');
    if ~tmp
        disp(bar);
        return
    end

    disp(barbar); disp(bar); fprintf('\n\n');
%% Part 4: Classification

    disp(bar); disp(barbar);
    disp('NOW: Classifying the image...');

    % Since the DATA_CLASS is already trained (from previous
       samples)
    %     We are ready to classify

    %       gather all the features in X_test:
    [~, num_instance] = size(DATA);
    for im=1:num_instance
        X_test = [X; DATA(im).Features];
    end

    % y_test_pred is a vector of classes predicted for each
       token
    [y_test_pred, prob] = gaussian_clf(X_test, DATA_CLASS);

```

```
    % display the classifcation of each features detected
    imshow(original_image);
    hold on;
    for i=1:length(y_test_pred);
        boundary = DATA.BoundingBox;
        class    =  y_test_pred;
        rectangle('Position', boundary, 'EdgeColor', cmap(class,:), 'LineWidth', 2);
    end

    % Save the figure
    s = sprintf('./imgs/testing/prediction.TEST.png',i,t);
    export_fig(s);

    % continue?
    fprintf('Classification Done!\n')
    tmp = input('Continue? [1/0] ');
    if ~tmp
        disp(bar);
        return
    end


    disp(barbar); disp(bar); fprintf('\n\n');

%% Part 5: Coinsy Counter:

    disp(bar); disp(barbar);
    disp('NOW: Initialising the counter...');
    % counter starts at 0
    counter = 0;
    values = [1,2,.5,.2,.05,.75,.25,.2,0,0,0];
    for i=1:length(y_test_pred)
        class = y_test_pred(i);
```

```
        counter = counter + values(class);
    end

    fprintf('Total Amount in image = %f', counter);


    disp(barbar); disp(bar); fprintf('\n\n');

%%  Part 5: Summary Statistics:
    disp(bar); disp(barbar);
    disp('SUMMARY STATISTICS');

% Expect something like:
fprintf('number of 1 pound = %d\n', sum(y_test_pred == 1));
fprintf('number of 2 pound = %d\n', sum(y_test_pred == 2));
fprintf('number of 50 pence = %d\n', sum(y_test_pred == 3));
fprintf('number of 20 pence = %d\n', sum(y_test_pred == 4));
fprintf('number of 5 pence = %d\n', sum(y_test_pred == 5));
fprintf('number of 75 pence = %d\n', sum(y_test_pred == 6));
fprintf('number of 25 pence = %d\n', sum(y_test_pred == 7));
fprintf('number of 2 pence = %d\n', sum(y_test_pred == 8));
fprintf('number of AAA Battery = %d\n', sum(y_test_pred == 9));
fprintf('number of Nut = %d\n', sum(y_test_pred == 10));
    disp(barbar); disp(bar); fprintf('\n\n');

%% Next image?
    % single loop for now:
    prompt_end = ('Do you want to load another image? [y/n]');
    x = input(prompt_end, 's');
    switch x
        case 'y'
            start = 1;
        case 'n'
            start = 0;
```

```
        case 'Y'
            start = 1;
        case 'N'
            start = 0;
        otherwise
            start = 1;
    end
end
```

B Image Processing

normalise_RGB.m

```
function [img_out, gray_out] = normalise_RGB(RGB, SHOW)
%% NORMALISE_INPUT_RGB(RGB, SHOW)
%   Normalise the RGB values for each pixel in the image RGB
%   Also, output the gray normalised output of RGB (i.e. normalised RGB +
%   rgb2gray();
%   The algorithm for normalisation is the root sum of channels squared.

%%
RGB = double(RGB); % cast into double
RED_Channel     = RGB(:,:,1);
GREEN_channel   = RGB(:,:,2);
BLUE_channel    = RGB(:,:,3);

[row,col,chn] = size(RGB);
img_out = zeros(row,col,chn);

for i = 1:row
    for j = 1:col
        r = RED_Channel(i,j);
        g = GREEN_channel(i,j);
        b = BLUE_channel(i,j);

        sum_sq = sqrt(r^2 + g^2 + b^2);
%         sum_sq = r + g + b;

        img_out(i,j,1) = r/sum_sq;
        img_out(i,j,2) = g/sum_sq;
        img_out(i,j,3) = b/sum_sq;
```

```
    end
end

% CAST IT BACK TO INT!
RGB = uint8(RGB);
img_out = uint8(img_out*255); % IMPORTANT TO MULTIPLY BY 255!!!

%% GRAY OUT STRATEGY:
%   simple for now..!
gray_out = rgb2gray(img_out);


%% DISPLAY RESULT:
if SHOW
    display_stats(RGB, img_out);
    figure;
    display_stats(rgb2gray(RGB),gray_out);
end

end
```

bg_extract.m

```
function [ bg_model ] = bg_extract( IMGS, WINDOW_SIZE )
%% BACKGROUND_MODEL(IMG, WINDOW_SIZE
%   Given a series of image, we find the common background
    using median
%   filtering. For each pixel in the bg_model, we take the
    median of all
%   the pixels in the WINDOW_SIZE for all the images. If
    WINDOW_SIZE = 1,
```

```
%   it is equivalent to taking the median of pixel intensity of
     all the
%   images.
%   If input image is RGB, then this is carried out for all
    channel.

%   INPUT:
%   — IMGS : A cell array of images. Images must be of the same
     size. IMGS
%   have size of (1,num_imgs)
%   — WINDOW_SIZE : the window of median_filter.
%       If undefined, WINDOW_SIZE = 1

%   OUTPUT:
%   — bg_model — an image of the same size as IMGS with the
    background
%   extracted.

%% Setting parameters
if nargin == 1
    WINDOW_SIZE = 1;
end

[~, num_imgs] = size(IMGS);
sample      = IMGS{1};
bg_model    = sample; % preallocation of memory

% Given a WINDOW_SIZE, find the number of cell to compensate:
% Window_Size    1 3 5 7 9...
% offset     ==  1 2 3 4 5
% ==>  offset = (WS + 1) / 2

% !! prevent even number WINDOW_SIZE
if mod(WINDOW_SIZE, 2) ~= 1
```

```
    error('Window_size must be an odd number!');
end

offset = uint64((WINDOW_SIZE + 1)/2);

% if offset == 1
%     offset = 0; % no need to offset if Window_size = 1
% end

disp('Extracting background from images....');
fprintf('\tWINDOW_SIZE = %d\n', WINDOW_SIZE);
fprintf('\tOffset = %d\n', uint8(offset));


%% iterate through all the images and set the
if ndims(sample) == 3

    [rows, cols, ~] = size(sample);

    % iterate each cell, neglecting offset cause of WINDOW_SIZE
    for i = offset:rows-offset+1
        for j = offset:cols-offset+1
%             disp([i,j]); %DEBUG

            % store all the values from each img in IMGS
%             median_RED      = zeros(1, num_imgs);
%             median_GREEN    = zeros(1, num_imgs);
%             median_BLUE     = zeros(1, num_imgs);
            median_RGB = zeros(num_imgs,1,3);

            % find bounding box of pixels
            x_low   = i - offset + 1;
            x_high  = i + offset - 1;
            y_low   = j - offset + 1;
```

```
            y_high  = j + offset - 1;
%             disp([x_low,x_high,y_low,y_high]); % DEBUG

            % iterate through all the pixels for the image
            for k=1:num_imgs
                temp = IMGS{k};
                segment = temp(x_low:x_high, y_low:y_high, :);
                med = median(median(segment)); % median along the color axis
                median_RGB(k,1,:) = med;
%                 % get the pixels belonging in the image:
%                 pixels_RED      = IMGS{k}(x_low:x_high, y_low:y_high, 1);
%                 pixels_RED      = reshape(pixels_RED, [], 1);
%                 median_RED(k)   = median(pixels_RED); % get the median of the nieghbood!
%
%                 pixels_GREEN    = IMGS{k}(x_low:x_high, y_low:y_high, 2);
%                 pixels_GREEN    = reshape(pixels_GREEN, [], 1);
%                 median_GREEN(k) = median(pixels_GREEN); % get the median of the nieghbood!
%
%                 pixels_BLUE     = IMGS{k}(x_low:x_high, y_low:y_high, 3);
%                 pixels_BLUE     = reshape(pixels_BLUE, [], 1);
%                 median_BLUE(k)  = median(pixels_BLUE); % get the median of the nieghbood!
            end

            % Set the meidan for respecitve color channel to the bg_model
```

```
%               bg_model(i,j,1) = median(median_RED);
%               bg_model(i,j,2) = median(median_GREEN);
%               bg_model(i,j,3) = median(median_BLUE);
            bg_model(i,j,:) = median(median_RGB);
        end
        fprintf('.');
    end

else
%% 2D images:
    [rows, cols] = size(sample);

    % iterate each cell
    for i = offset : rows-offset
        for j = offset : cols-offset

            median_val = zeros(1,num_imgs);

            % finding bounding box:
            x_low   = i - offset + 1;
            x_high  = i + offset - 1;
            y_low   = j - offset + 1;
            y_high  = j + offset - 1;

            % iterate through all the pixels for the image
            for k=1:num_imgs
            % find bounding box of pixels
                pixels  = IMGS{k}(x_low:x_high, y_low:y_high);
                pixels  = reshape(pixels, [], 1);
                median_val(k) = median(pixels); % get the
                    median of the nieghbood!
            end
            % Set the meidan for respecitve color channel to
                the bg_model
```

```
                bg_model(i,j) = median(median_val);

            end
            fprintf('.');
        end


end

bg_model = uint8(bg_model); % cast back to int
% imshow(bg_model); % DEBUG
disp('done');

end
```

bg_subtraction.m

```
function [img_bgremove, bg_model] = bg_subtraction(img,
    bg_model)
%% BG_SUBTRACTION(IMG, BG_MODEL)
%   returns a new image after subtracting it with the bg_model
%   Given a cell array of img, bg_model models after these img
    and return a
%   cell array of img with their background removed using the
    inferred
%   bg_model

%   INPUT:
%   - IMG:  a cell array or just an image. If it is just an
    image, a
%   bg_model must be given
%   - bg_model : optional if you want the algorithm to infer
    the bg model
```

```matlab
%   from the img. in this case, img must be a cell array of image
%   N.B, if bg_model is given and img is cell, no bg_model is inferred, and
%   this will be just a simple straightforward bg subtraction algorithm.

%   OUTPUT:
%   - new_img : a cell array of image if input is cell array
%   - bg_model : if bg_model is inferred, otherwise just the bg_model

%%
% No bg_model given; img is cell array of images
if iscell(img)

    [~, num_img] = size(img);
    img_bgremove = img; % memory allocation

    switch nargin
        case 1
            disp('extracting bg_model from cell array of images');
            bg_model = bg_extract(img);

            % carry out subtraction:
            for i=1:num_img
                img_bgremove{i} = abs(img{i} - bg_model); % take abs, avoid negative
            end

        case 2
            disp('subtracting all images with given bg_model')
            % carry out subtraction:
```

```
            for i=1:num_img
                img_bgremove{i} = abs(img{i} - bg_model);
            end
    end
else
    % subtract bg from img;
    img_bgremove = abs(img - bg_model);
end

disp('done');

end
```

dothresh.m

```
function [img_thresh, thresh_vals] = dothresh(IMGS, sizeparam)
%% DOTHRESH(IMGS, SIZEPARAM)
%   Function that find the threshold for an image then apply
    thresholding
%   to get a binary image.
%
%   INPUT:
%   - IMGS : a cell array of images or just an image of
    interest
%   - sizeparam : thte
%
%   OUTPUT:
%   - thresh_imgs : if IMGS is a cell array of images, so it
    thresh_imgs.
%       the images are thresholded with its corresponding
    threshold in
%       thresh_vals
```

```matlab
%   — thresh_vals : if the imgage is RGB, then thresh_vals is a veector of
%       threshold for each RGB channel
%
%   Dependencies:
%   — findthresh.m — from rbf's ivr repository; Standard filterlen = 50,
%       alpha = sizeparam.
%       N.B.    if filterlen is large, then curve is smoother!
%               if alpha is large, then width of the window is smaller!

%%

% % In case sizeparam is not passed
% try sizeparam
% catch
%     sizeparam = 16;
% end

if iscell(IMGS)
    [~, num_imgs] = size(IMGS); % find number of images
    thresh_vals = {}; % for storing all the threshold values

    for k = 1:num_imgs % iterate through all the images

        img             = IMGS{k}; % this image
        imgX            = zeros(size(img)); % the output image

        if ndims(img) == 3 % RGB Channel

            thresh_vals{k}  = zeros(1,3); % pre—allocation
            for i=1:ndims(img) % iterate through each dim to get the BW pic
```

```
                % call itself to get the threshold value (see `else` below)
                [imgX(:,:,i), thresh_vals{k}(i)] = ...
                    dothresh(img(:,:,i), sizeparam);
            end

            % Now, `OR` the values together
            img_thresh{k} = imgX(:,:,1) | imgX(:,:,2) | imgX(:,:,3);

        else % for 2D case
            [img_thresh{k}, thresh_vals{k}] = dothresh(img, sizeparam);
        end

    end
%% 2D image input:
%   For 2D array, use findthresh to get the threshold for the image and
%   then get the bw representation of it!

%   TODO: MAY need to toggle the bw = ~bw, depending if you want objects to be
%   white or black
else
    hist = dohist(IMGS); % get the histogram of 2D image
    thresh_vals = findthresh(hist, sizeparam, 0); % find the threshold of the iamge
    [n,m] = size(IMGS);

    % now, get the binary representation
    for i=1:n
        for j=1:m
```

```
            if IMGS(i,j) >= thresh_vals % this is the objects!
               we want it!
                img_thresh(i,j) = 1;
            else
                img_thresh(i,j) = 0; % set background to 0
            end
        end
    end

end

end
```

C Feature Extraction

getFeatures.m

```
function vec = getFeatures(image, prop)
%% getproperties(Image)
%   gets property vector for a binary shape in an image
%   properties extracted:
%       1) Area
%       2) Perimeter
%       3) compactness
%       4) rectangularity
%       5) elongation
%       6) hu moment invariant
%       7) complex invariant

Image = image.Image;

[H,W] = size(Image);
area = bwarea(Image);
perim = bwarea(bwperim(Image,8));

% compactness
compactness = perim*perim/(4*pi*area);

% rescale properties so all have size proportional
% to image size
area_ = 4*sqrt(area);
compactness_ = H*compactness;

% rectangularity
bb_width = image.BoundingBox(3);
bb_height = image.BoundingBox(4);
area_bb = bb_width * bb_height;
```

```
rectangularity = area / area_bb;

% Elongation - ratio of principal axis
elongation = prop.MajorAxisLength / prop.MinorAxisLength;

hu_invariant = humomentinvariants(Image);

% get scale-normalized complex central moments
c11 = complexmoment(Image,1,1) / (area^2);
c20 = complexmoment(Image,2,0) / (area^2);
c30 = complexmoment(Image,3,0) / (area^2.5);
c21 = complexmoment(Image,2,1) / (area^2.5);
c12 = complexmoment(Image,1,2) / (area^2.5);
%c=[c11,c20,c30,c21,c12]

% get invariants, scaled to [-1,1] range
ci1 = real(c11);
ci2 = real(1000*c21*c12);
tmp = c20*c12*c12;
ci3 = 10000*real(tmp);
ci4 = 10000*imag(tmp);
tmp = c30*c12*c12*c12;
ci5 = 1000000*real(tmp);
ci6 = 1000000*imag(tmp);

%ci=[ci1,ci2,ci3,ci4,ci5,ci6]

vec = [area_, perim, compactness_ , rectangularity, elongation, ...
        ci1, ci2, ci3, ci4, ci5, ci6]; % 18 features


end
```

rawmoment.m

```
function M_ij = rawmoment(img,p,q)
%% rawmoment(img,p,q) calculates the (p+q)th raw moment of img
%   image is a BW img

%%
[m,n] = size(img);
M_ij = 0;
for i=1:m
    for j=1:n
        x=i; y=j;
        I_xy = img(i,j);
        M_ij =  M_ij + (x^p * y^q * I_xy);
    end
end


end
```

centralmoment.m

```
function miu_pq = centralmoment(img,p,q)
%% centralmoment(img,p,q) calculates the (p+q)th central moment of img
%   image is a BW img
%   covariance = miu_11; variance = miu_02 or miu_20;

%%
[m, n] = size(img);
M_00 = rawmoment(img,0,0);
M_10 = rawmoment(img,1,0); % mean x
M_01 = rawmoment(img,0,1); % mean y
```

```
centroid_x = M_10/M_00;
centroid_y = M_01/M_00;

miu_pq = 0;
for i=1:m
    for j=1:n
        diff_x = (i-centroid_x) ^ p;
        diff_y = (j-centroid_y) ^ q;
        I_xy = img(i,j);
        miu_pq = miu_pq + (diff_x * diff_y * I_xy);
    end
end


end
```

SI_moment.m

```
function pi_pq = SI_moment(img,p,q)
%% Calculates the Scale invariant moment given the (p+q) moment

    miu_00 = centralmoment(img,0,0); % the area
    miu_pq = centralmoment(img,p,q);

    pi_pq = miu_pq / (miu_00^(1+(p+q)/2));
```

complexmoment.m

```
% gets a given complex central moment value
function c_uv = complexmoment(Image,u,v)

```

```
    [r,c] = find(Image==1);            % get (r,c) of region's pixels
    rbar = mean(r);
    cbar = mean(c);
    n = length(r);
    momlist = zeros(n,1);

    for i = 1 : n
      c1 = complex(r(i) - rbar, c(i) - cbar);
      c2 = complex(r(i) - rbar, cbar - c(i));
      momlist(i) = c1^u * c2^v;
    end

    c_uv = sum(momlist);
```

D Classification

split_data.m

```
function [X_train, X_vali, X_test, y_train, y_vali, y_test] = ...
    split_data(X, y, train, vali, test)
%% SPLIT_DATA(X, y, train, vali, test);
% Use hold out validation technique to randomly generate the
% training, validation and testing set
% Since the images are input, we will use create psuedo
% samples from the  subimages

% INPUT:
% âĹŠ train,vali,test : are double from [0,1] that indicate the size of each
% sets. Hence they must sum up to 1;

%%
num_instances = length(X);
if length(X) ~= length(y)
    error('X and y does not have the same number of instances');
end
if (train + vali + test) ~= 1;
    error('train + vali + test ~= 1!!');
end

num_train = floor(num_instances * train);
num_vali = floor(num_instances * vali);
num_test = num_instances - num_train - num_vali;

X_train_idx = randperm(num_instances,num_train);
X_train = X(X_train_idx, :);
```

```
y_train = y(X_train_idx);
% remove these instances
X(X_train_idx,:) = [];
y(X_train_idx) = [];

X_vali_idx = randperm(num_instances-num_train, num_vali);
X_vali = X(X_vali_idx,:);
y_vali = y(X_vali_idx,:);
% remove these instances
X(X_vali_idx,:) = [];
y(X_vali_idx) = [];
% the rest for test:
X_test = X;
y_test = y;

end
```

user_classify.m

```
function [relevance, class] = user_classify()
%% USER_CLASSIFY(IMG)
%   Given an img, ask the user which class it belongs to

%%
% fprintf('\n\nplease enter the two class for this img\n');
prompt = 'Is this relevant? [0/1]';
relevance = input(prompt); % to count the class or not!

if relevance
    fprintf('\n====\nWhats the value?\n');
    fprintf('[1] 1 POUND  [2] 2 POUND  [3] 50 P  [4] 20 P  [5]
       5 P\n')
```

```
    fprintf('[6] 75 P (washer w small hole)  [7] 25 P (washer w large hole)\n');
    fprintf('[8] 2 P (angle bracket)\n[9] AAA battery (no val) [10] nut (no value)\n');
%     fprintf('[11] HELP!! (will display the bigger picture)\n\n');
    fprintf('[0] HELP!! (will display the bigger picture)\n\n');
    prompt = '>>  ';
    class = input(prompt);

    % Reject error in class input
    while (class < 0 || class > 10)
        fprintf('Classes ranges from 1 to 11 only\n');
        class = input(prompt);
    end

    fprintf('\n===\n')
else
    class = 11;
end


end
```

findConfusion.m

```
function[ CM, Per ] = findConfusion(result, test_class, num_class, p_limit)
%% findConfusion
% INPUT: [targets, output]
%   S = number of features ( in this case, 10)
%   Q = number of test data
```

```
%   result      :   Q-by-1 data each (i,j) indicates the ith
    input's class,
%   test_class  :   Q-by-1 data each (i,j) indicates the class
    given to ith
%               input.
%   targets and output must be ordered the same way.

% OUTPUT: [c. cm, ind, per]
%   cm  :   S-by-S confusion matrix, where (i,j) is the number
    of samples
%           whose target is the ith class that was classified
    as j
%   per :   S-by-4 matrix, where each row summarises four
    percentages
%           associated with the ith class:


%% setup:
[Q,S] = size(result); % Q = number of observation
[~,S1] = size(test_class);

% check for number of test-case
if S ~= S1
    error('test_class and results doesnt match in size');
end

%% create the confusion matrix
% Row = actual
% column = predicted
cm = zeros(num_class, num_class); % dont need to show cm for
    class 11

% iterate through all the test data to add data into the
    confusion matrix
```

```
for q=(1:Q)
    predictedClass = test_class(q,1);
    actualClass = result(q,1);

    if predictedClass == actualClass
        % if the classifier successfully classsfied the
            datapoint
       cm(actualClass,actualClass) = ...
           cm(actualClass,actualClass) + 1;
    else
        % classifier classifies the point wrongly.
        cm(actualClass, predictedClass) = ...
            cm(actualClass,predictedClass) + 1;
    end
end

%% manipulate the cm to get per:
per = zeros(num_class,4); % ignore the unclassified class here
% each row corresponds to each class
%         per(i,1) false negative rate
%                   = (false negatives)
%         per(i,2) false positive rate
%                   = (false positives)
%         per(i,3) true positive rate
%                   = (true positives)
%         per(i,4) true negative rate
%                   = (true negatives)

% for each class find the FN, FP, TP, TN respectively.
for s=(1:num_class)
    % generate the data;
    TP = cm(s,s);
    FP = sum(cm(:,s)) - TP;
    FN = sum(cm(s,:)) - TP;
```

```matlab
    TN = sum(sum(cm)) - TP - FN - FP;

    % store the values
    per(s,1) = FN;
    per(s,2) = FP;
    per(s,3) = TP;
    per(s,4) = TN;
end

CM = cm;
Per = per;

%%
figure; imshow(CM, [], 'InitialMagnification', 1600); colormap(bone);
title('Confusion Matrix for Testing Set');

fprintf('Done!\n\nThe confusion matrix is:\n(rows = actual class; columns = predicted class)\n');
disp(CM);
fprintf('\nThe classification results for each class are:\n   (FN    FP    TP   TN)\n');
disp(per);

disp('========================================================');
fprintf('Summary:\nClassification using full gaussian model\n');
FN = sum(per(:,1));
FP = sum(per(:,2));
TP = sum(per(:,3));
TN = sum(per(:,4));
incorrect = FP + FN;
correct = TP;
```

```
acc_score = TP/ Q; % Q = number of obervation

fprintf('Number Incorrect = %d\n', incorrect);
fprintf('Number Correct = %d\n', correct);
fprintf('Number of classes = %d\n', num_class);
% fprintf('Number Unclassified (lesser than p = %.2f) = %d\n',
    p_limit, Per(11,1) );
fprintf('Accuracy = %3f percent\n\n', acc_score);
disp('========================================================'
    );

end
```

gaussianDistr.m

```
function p = gaussianDistr(mean_, cov_, prior, data)
%% GAUSSIANDISTR(MEAN,COV,X)
%   using log posterior probability:
%   ln P(C|x) = (-.5)(x-mu)'(inv(cov))(x-mu) - .5(ln(det(cov))
    + ln(P(C))
%
%   INPUT:
%       mean = scalar; mean of gaussian distribution
%       cov  = D-by-D matrix; covariance of ditribution
%       x    = D dimension vector to calculate the pr.
%
%   OUTPUT:
%       p    = probability of x being classified using this
    gaussian model

%% generate Probability;

% diff = data - mean_;
```

```
% dist = diff*cov_*diff';
% n = length(data);
% wgt = 1/sqrt(det(cov_));
% p = prior * ( 1 / (2*pi)^(n/2) ) * wgt * exp(-0.5*dist);
% disp(p); %% DEBUG
% %
%
[A,D] = size(cov_);
mean_ = mean_'; % assume data and mean is presented as row vector
data  = data';

logDet = (-.5) * logdet(cov_);
firstPart = (-.5) * ((data - mean_)' / cov_) * (data - mean_);
prior = log(prior);

% calculate the probability using the formula:
p = firstPart + logDet + prior;

end
```

gaussian_clf.m

```
function [prediction, prob_all] = gaussian_clf(X_test, DATA_CLASS, p_limit)
%% GAUSSIAN_CLF(X_TEST, MEAN, COVARIANCE)
%   Given a the features of some images (X_test), we use a gaussian model
%     to find the most probable class (i.e. highest probability);
%
%   INPUT:
```

```
%   - DATA_CLASS is a cell array of struct where each cell gives us the
%    information of the class. The number of class = length of DATA_CLASS
%
%    OUTPUT:
%   - predictions : a list of classes for each instances in X_test

%%

num_class   = length(DATA_CLASS);
num_sample  = length(X_test);

prediction  = zeros(num_sample,1);
prob_all    = zeros(num_sample, num_class);

for n=1:num_sample
    for i=1:num_class
%         fprintf('%d,%d',n,i); %%DEBUG
        prior_  = DATA_CLASS{i}.Prior;
        mean_   = DATA_CLASS{i}.Mean;
        cov_    = DATA_CLASS{i}.Cov;
        data    = X_test(n,:);
        p = gaussianDistr(mean_, cov_, prior_, data);
%         disp(p);
        prob_all(n,i) = p;

    end

    [probs, prediction(n)] = max(prob_all(n,:));

    % toggle the use of p_limit
    if nargin == 3
```

```
        % If probability is larger than the confidence interval, thrash it!
        for i = 1:num_sample
            if probs < p_limit
                prediction(n) = 11; % set to unclassified
%                 fprintf('x');
            end
        end
    end


end

end
```

E Basic Filters

median_filter.m

```
function img_filtered = median_filter(img, show, SIZE )
%% MEDIAN_FILTER
%   Use median filter to reduce the impulse noise in the image base on the
%   local intensity distribution. The distribution being conisdered by the
%   filter is determined by SIZE.
%
%   If there are more than 2 dims in img (such as a HSV or RGB) image,
%   median filter is passed through each dimension independently. The
%   resulting image is then put together as img_filtered.

% INPUT:
%   SIZE - either a scalar or a vector representing the row and col. If not
%   defined, the default value of 3x3 is used.
%   img - image to be filtered
%   show - 0/1 to imshow the images

%% Do Median Filtering for each channel (can be HSV/RGB)
if ndims(img) == 3
    RGB = img;
%     [r, c, channel] = size(img);
    red_org     = RGB(:, :, 1);
    green_org   = RGB(:, :, 2);
    blue_org    = RGB(:, :, 3);

    if nargin == 3
```

```
        red_medfilt     = medfilt2(red_org, SIZE, 'symmetric');
        green_medfilt   = medfilt2(green_org, SIZE, 'symmetric');
        blue_medfilt    = medfilt2(blue_org, SIZE, 'symmetric');
    else
        red_medfilt     = medfilt2(red_org, 'symmetric');
        green_medfilt   = medfilt2(green_org, 'symmetric');
        blue_medfilt    = medfilt2(blue_org, 'symmetric');
    end

    img_filtered = cat(3, red_medfilt, green_medfilt, blue_medfilt);

else
    %% DO 2D Median Filtering
    if nargin == 3
        img_filtered = medfilt2(img, SIZE);
    else
        img_filtered = medfilt2(img);
    end

end

if show
    display_stats(img, img_filtered);
%     figure; imshow([img, img_filtered]);
end

end
```

median_filter_iter.m

```
function img_filtered = median_filter_iter(img, ITER, show,
    SIZE)
%% MEDIAN_FILTER_ITER
%   Use median filtering for a definite number of times.
% INPUT:
%   img     : initial image
%   ITER    : Number of iteration
%   show    : to display the images after
%   SIZE    : SIZE of the filter window (OPTIONAL)

%%

img_temp = img;

try
    for i = 1:ITER
        img_temp = median_filter(img_temp, 0, SIZE);
    end
catch
    for i = 1:ITER
        img_temp = median_filter(img_temp, 0);
    end
end

if show
    display_stats(img, img_temp);
end

img_filtered = img_temp;
end
```

gaussian_filter_1d.m

```
function smoothed_1d = gaussian_filter_1d(hist, show, window_size, alpha)
%% GAUSSIAN_FILTER_1D(HIST, SHOW, WINDOW_SIZE, ALPHA)
%   Uses the gausswin function to produce a gaussian window,
%   then apply a conv to the 1d-hist.
%   If hist is not 1d, coerce it into 1d

%   Note:
%   As alpha increase, width of window will decrease. Default = 2.5
%   As window_size increase, the curve will be smoother.
%   Use dohist to get the histogram!

%%
% first check for the size of the image, if not 1d, coerce it
if ndims(hist) == 3 % a color image is input,
    % convert to grayscale first:
    hist = rgb2gray(hist);
end

% Use dohist to get the histogram of intensity
hist = dohist(hist);

% window_size and alpha not defined, use default
if nargin == 1 || nargin == 2
    gauss_filter = gausswin(50, 6);
else
    gauss_filter = gausswin(window_size, alpha);
end

filter = gauss_filter/ sum(gauss_filter);
smoothed_1d = conv(filter, hist);

try
```

```
    if show
        subplot(2,2,1); plot(hist); title('Original Image');
        subplot(2,2,3); plot(filter); title('Filter');
        subplot(2,2,2); plot(smoothed_1d); title('Smoothed
            Image');
    end
catch
    subplot(2,2,1); imshow(hist); title('Original Image');
    subplot(2,2,3); plot(filter); title('Filter');
    subplot(2,2,2); imshow(smoothed_1d); title('Smoothed Image'
        );
end
```

gaussian_filter_2d.m

```
function smoothed_2d = gaussian_filter_2d(img, show, HSIZE,
    SIGMA )
%% gaussian_filter_2d(img, show, HSIZE, SIGMA )
% Smooth an image using Gaussian lowpass filter and imfilter
% INPUT:
% - img : can be an RGB/HSV or GRAYSCALE image
% - HSIZE : corresponds to fspecial requirements, can be a
    vector
%   specifying the number of rows and columns or a scalar (
    infered to be a
%   squared matrix
% - SIGMA : the spreaed of the Gaussian
% N.B. Default HSIZE = [3,3], SIGMA = .5

if (nargin == 1 || nargin == 2)
    H = fspecial('gaussian');
else
    H = fspecial('gaussian', HSIZE, SIGMA);
```

```
end


% ensure the output is the same size as img
% use conv instead of filter function
smoothed_2d = imfilter(img, H, 'conv', 'same');

try
    if show
        subplot(2,2,1); imshow(img); title('Original Image');
        subplot(2,2,3); surfc(H); title('Filter');
        subplot(2,2,2); imshow(smoothed_2d); title('Smoothed Image');
    end
catch
    % if fail to input show, will just output the images!
    subplot(2,2,1); imshow(img); title('Original Image');
    subplot(2,2,3); surf(H); title('Filter');
    subplot(2,2,2); imshow(smoothed_2d); title('Smoothed Image');
end
```

References

[1] E.R. Davies. *Computer and Machine Vision (Fourth Edition)*. Academic Press, Boston, fourth edition edition, 2012. ISBN 978-0-12-386908-1. doi: http://dx.doi.org/10.1016/B978-0-12-386908-1.00001-X. URL `http://www.sciencedirect.com/science/article/pii/B978012386908100001X`.

[2] Noah Snavely. Lecture 2: Image filtering. URL `http://www.cs.cornell.edu/courses/cs6670/2011sp/lectures/lec02_filter.pdf`.

[3] A. Walker R. Fisher, S. Perkins and E. Wolfart. Hipr2 - image processing learning resources. URL `http://homepages.inf.ed.ac.uk/rbf/HIPR2/index.htm`.

[4] Andrew Ng. Stanford machine learning class notes. URL `http://www.holehouse.org/mlclass/index.html`.