This course has been developed in the framework of the ITEMS master program (Techniques for Analysis, Modeling and Simulation for Imaging, Bioinformatics and Complex Systems) financed through project POSDRU/86/1.2/S/61756.

Computer Vision I
C. Rasche
compvis12 [at] gmail [dot] com

Abstract
We go straight to the heart of modern computer vision and firstly introduce the techniques for feature extraction and matching (of histograms of gradients), which is at the core of many tasks such as object recognition, image retrieval, image stitching, etc. Then we introduce object detection based on the sliding window technique (e.g. suitable for face and pedestrian detection). It follows a light but sufficient treatment of image processing techniques - segmentation and morphological processing; as well as other features such as contours, blobs, regions etc.; and the introduction of shape recognition techniques. We overview the essential tracking methods (for regions and moving objects). We close with a survey of video surveillance and in-vehicle vision system. The techniques are explored in Matlab and sufficient code snippets are provided to facilitate exploration of all those concepts. It is possible to follow the notes without any particular prerequisites, but basic knowledge of linear algebra, signal processing and pattern recognition is of great advantage.

Prerequisites: basic programming skills; enthusiasm to write a lot of code
Recommended: basic statistical pattern recognition, basic linear algebra, basic signal processing

Contents

1 Introduction ................................................. 5
  1.1 Related Fields ........................................ 5
  1.2 Recognition - An Overview .......................... 5
  1.3 Areas of Application (Examples) .................... 6
  1.4 Organization of a Computer Vision System  ......... 7
  1.5 Historical Note ........................................ 7
  1.6 Prototyping in Matlab, Implementation in C ....... 8
  1.7 Exercises ............................................. 8
    1.7.1 Localizing Face Parts ............................ 8
    1.7.2 Understanding Recognition ....................... 9

2 Image Processing I: Scale Space and Gradient Image .... 10
  2.1 Scale Space, Pyramid ............................... 10
    wiki: Scale_space -------------------------- Sze p17, s3.5, p144 FoPo p164, s4.7, p134 SHB p106, s4.3 .... 10
  2.2 Gradient Image ...................................... 11
    wiki: Image_gradient ................................. 11
  2.3 Exercises ........................................... 12

3 Feature Extraction I: Patches ......................... 13
  3.1 Detection ........................................... 13
    FoPo p179, pdf 149 Dav p158, s6.7, 7 SHB p156, s5.3.10 Pnc p281, s13.2.2 .... 13
  3.2 Description ........................................ 15
    FoPo p179, s5.4.1, pdf157 Dav p173, s6.7.3 Pnc p284, s3.3.3.2 ........ 15
  3.3 Matching ........................................... 16
  3.4 Notes .............................................. 16
  3.5 Exercises ........................................... 17

4 Feature Quantization .................................. 19
  4.1 Building a Dictionary ................................ 19
    4.1.1 Vector-Quantization using the k-Means Algorithm ........ 20
  4.2 Applying the Dictionary to an Image ............... 20
  4.3 Classification ...................................... 21
  4.4 Principal Component Analysis (PCA) ............... 21
  4.5 Exercises ........................................... 22
# 5 Object Detection  
FoPo p549, ch 17, pdf 519  

## 5.1 Face Detection  
Sze p578, ch 14.1.1, pdf 58  

### 5.1.1 Rectangles  
Sze p106, pdf 120  
Pnc p275  
Dav p175  
SHB p101, alg 4.2  

## 5.2 Pedestrian Detection  

## 5.3 Improvement by Knowing Context  

## 5.4 Notes  

# 6 Image Processing II: Segmentation & Morphology  
27  

## 6.1 Segmentation  
SHB p176, ch 6  
FoPo p285, ch 9, pdf 255  
Sze p235, ch 5, pdf 267  

### 6.1.1 Thresholding (Global, Local, Band, Multi-Level)  
SHB p177, s 6.1  
Dav p82, ch 4, pdf 119  

### 6.1.2 Region Growing: The Watershed Algorithm  
SHB p233, s 6.3.4  
Sze p251, s 5.2.1, pdf 283  

### 6.1.3 Clustering with k-Means  

### 6.1.4 Region Labeling and Statistics  

## 6.2 Morphological Processing  
Sze p112, s 3.3.2, p 127  

### 6.2.1 Binary Morphology  

### 6.2.2 Grayscale Morphology  

## 6.3 Exercises  

# 7 Feature Extraction II: Contours, Regions, Blobs & Texture  
33  

## 7.1 Contour Extraction  

### 7.1.1 Edge Detection  

### 7.1.2 Edge Following  

## 7.2 Dots, Blobs, Regions - Bandpass Filtering  

## 7.3 Texture  
Dav p209, ch 8  
FoPo p194, pdf 164  

### 7.3.1 Statistical  
ThKo p412  

### 7.3.2 Spectral [Structural]  
FoPo p202  
Pnc p277, s 13.1.5  

## 7.4 Special Features  

### 7.4.1 Straight Line Segments, Circles  

## 7.5 Exercises  

## 7.5.1 Edge Detection & Following  

## 7.5.2 Blobs (Band-Bass Filtering)  

# 8 Shape  
38  

## 8.1 Compact Description  

### 8.1.1 Simple Measures  

### 8.1.2 Radial Description  

## 8.2 Point-Wise  

### 8.2.1 Boundaries  

### 8.2.2 Sets of Points  

## 8.3 Toward Parts: Distance Transform & Skeleton  

### 8.3.1 Distance Transform  
Sze p113, s 3.3.3, pdf 129  
Dav p240, s 9.5  
SHB p19, s 2.3.1  

### 8.3.2 Symmetric Axes, Skeleton  

## 8.4 Classification  

## 8.5 Exercises  

# 9 Image Classification/Search/Retrieval  
43  

## 9.1 Image Classification  
FoPo p612, ch 16, pdf 462  

## 9.2 Image Search and Retrieval  
FoPo p657, ch 21, pdf 627  

### 9.2.1 Applications  

### 9.2.2 Document Retrieval  
Sze p604, s 14.3.2, pdf 687  
FoPo p682, s 21.2, pdf 632  

---
1 Introduction

Computer vision is the field of interpreting image content: it is concerned with any type of recognition of parts (regions or structure) of the image, or with the classification of the entire image. Often one seeks a comparison to the human visual system - your eyes and most of your brain -, which can easily interpret any scene with little effort: it perfectly discriminates between thousands of categories and it can find objects in scenes within a duration of several hundred milliseconds only. This is a flexibility, which has not been well understood yet and which therefore is difficult to implement. In computer vision in contrast, one resorts to specific tasks, namely to determine whether an image (or image sequence) contains some specific object, feature, or activity; for example to detect all faces in an image. Often, such specific tasks can be implemented in different ways, with each implementation having certain advantages and disadvantages. Some of the implemented tasks can outperform a human observer.

1.1 Related Fields

Several fields are related to computer vision, of which two fields are particularly closely related, namely image processing and machine vision; in fact, those two fields overlap with computer vision to some extent. Here is an attempt to outline their scope, although there exist no agreed definitions and distinctions:

**Image Processing** is concerned with the transformation or other manipulation of the image with the goal to emphasize certain image aspects, e.g. contrast enhancement, or extraction of level features such as edges, blobs, etc; in comparison, computer vision is rather concerned with higher-level feature extraction and their interpretation for recognition purposes.

**Machine Vision** is concerned with applying a range of technologies and methods to provide imaging-based automatic inspection, process control and robot guidance in industrial applications. A machine vision system has typically 3 characteristics:

1) objects are seen against an uniform background, which represents a 'controlled situation'.
2) objects possess limited structural variability, sometimes only one objects needs to be identified.
3) the exact orientation in 3D is of interest.

In comparison, computer vision often deals with objects of larger variability and objects that are situated in their typical background, which in turn can be rather complex.

Another related field is pattern recognition (or machine learning), which is the art of classification (categorization). To build a good computer vision system, it requires substantial knowledge of classification methodology - sometimes it is even the more significant part of the computer vision system. Clearly, we cannot treat classification in depth in this course and we will merely point out how to use some of the classifiers, see appendix A - later we will refer to classifiers more specifically.

The field of computer graphics is sometimes considered as part of computer vision. The objective in computer graphics is to represent objects and scenes as compactly and efficiently as possible; however there is no recognition of any kind involved.

1.2 Recognition - An Overview

We firstly explain the three principal recognition processes and their challenges. Then we mention other recognition objectives.

**Classification (Categorization):** an object or scene is assigned to a class (category), such as 'car', 'apple', 'beach scene', etc. In praxis, the discrimination between a few classes is manageable, however the larger the number of classes we desire to discriminate, the more challenging it is to deal with the intra-class variability. It is difficult to express the structural variability between instances within the same class - think of how differently chairs can look like.

**Identification:** an individual instance of an object is recognized. It is essentially a specific type of classification (as introduced just above), whereby the challenge is to discriminate between endlessly subtle structural variability. Examples: face identification, fingerprint identification, identification of a specific vehicle.
Detection (Localization): the image is searched for either a specific object class, or for an object instance or it is tested for a specific condition; the number of object occurrences is counted. The challenge is to create an efficient search that can find the object irrespective of its size: does the object cover the entire image?, or is it small and therefore difficult to detect? Examples: face detection, vehicle detection in an automatic road toll system, detection of possible abnormal cells or tissues in medical images.

The term object recognition often implies some combination of those processes - sometimes it stands for only one of those three processes. The term general object recognition refers to the recognition of any object in arbitrary scenes and implies the combination of the processes of detecting and classifying an object: it is of course difficult and represents essentially the process of human recognition. The following are other frequent recognition tasks:

Motion Analysis: the movement of an object is investigated. One can be interested in merely detecting the movement, which is called tracking. Or one can identify specific movements, in which case it is an identification task.

Retrieval: here we order images according to certain criteria. For instance we pass an image to the system (e.g. white rose), and the system returns us the 20 most similar images (rose, tulip, sunflower etc). The order is determined based on some sort of 'comparison', similar to the comparison made during classification or identification processes.

Pose Estimation: determines the exact position or orientation of a specific object relative to the camera. Example: assisting a robot arm in retrieving objects from a conveyor belt in an assembly line or picking parts from a bin.

1.3 Areas of Application (Examples)

The following list of areas merely gives an overview of where computer vision techniques have been applied so far; the list also contains applications of image processing and machine vision, as those fields are related:

Medical imaging: registering pre-operative and intra-operative imagery; performing long-term studies of people’s brain morphology as they age; tumor detection, measurement of size and shape of internal organs; chromosome analysis; blood cell count.

Automotive safety: traffic sign recognition, detecting unexpected obstacles such as pedestrians on the street, under conditions where active vision techniques such as radar or lidar do not work well.

Surveillance: monitoring for intruders, analyzing highway traffic, monitoring pools for drowning victims.

Gesture recognition: identifying hand postures of sign level speech, identifying gestures for human-computer interaction or teleconferencing.

Fingerprint recognition and biometrics: automatic access authentication as well as forensic applications.

Visual authentication: automatically logging family members onto your home computer as they sit down in front of the webcam.

Robotics: recognition and interpretation of objects in a scene, motion control and execution through visual feedback.

Cartography: map making from photographs, synthesis of weather maps.

Radar imaging: target detection and identification, guidance of helicopters and aircraft in landing, guidance of remotely piloted vehicles (RPV), missiles and satellites from visual cues.

Remote sensing: multispectral image analysis, weather prediction, classification and monitoring of urban, agricultural, and marine environments from satellite images.

Machine inspection: detect and fault inspection of parts: rapid parts inspection for quality assurance using stereo vision with specialized illumination to measure tolerances on aircraft wings or auto body parts; or looking for defects in steel castings using X-ray vision; parts identification on assembly lines.

The following are specific tasks which can be often solved with image processing techniques and pattern recognition methods and that is why they are often marginally treated only in computer vision textbooks - if at all:
Optical character recognition (OCR): identifying characters in images of printed or handwritten text, usually with a view to encoding the text in a format more amenable to editing or indexing (e.g. ASCII). Examples: mail sorting (reading handwritten postal codes on letters), automatic number plate recognition (ANPR), label reading, supermarket-product billing, bank-check processing.

2D Code reading reading of 2D codes such as data matrix and QR codes.

1.4 Organization of a Computer Vision System

The organization of a computer vision system is highly application dependent and consequently there does not exist a general recognition scheme. The following list introduces terminology that is used to describe the stages (phases) in a recognition process, which do not strictly appear in that order and which can not always be clearly distinguished. The first two stages clearly belong to the field of image processing; the following three fields represent the ‘meat’ of computer vision (feature extraction, detection/segmentation and high-level processing); the final stage corresponds essentially to pattern recognition.

Image acquisition: is the analog-to-digital conversion of the ‘outer’ signal to a number by one or several image sensors (cameras). Besides the typical types of light-sensitive cameras, there are also range (depth) sensors, tomography devices, radar, ultra-sonic cameras, etc. Depending on the type of sensor, the resulting image data is an ordinary 2D image, a 3D volume, or an image sequence. The pixel values typically correspond to light intensity in one or several spectral bands (gray images or color images), but can also be related to various physical measures, such as depth, absorption or reflectance of sonic or electromagnetic waves, or nuclear magnetic resonance.

Image processing: is the manipulation of the ‘raw’ image to assure that it satisfies certain assumptions implied by the method. Examples are: a) re-sampling in order to assure that the image coordinate system is correct; b) noise reduction in order to assure that sensor noise does not introduce false information. c) contrast enhancement to assure that relevant information can be detected. d) scale-space representation to enhance image structures at locally appropriate scales.

Feature extraction: is the process of extracting specific types of ‘information’ (features) from the image in order to facilitate later classification processes. Typical examples of such features are lines, edges, blobs, corners, etc.; more complex features may be related to texture, shape or motion.

Detection/segmentation: is the process of deciding which image points or regions of the image are relevant for further processing. Examples are: selection of a specific set of interest points; segmentation of one or multiple image regions which contain a specific object of interest.

High-level processing: At this step the input is typically a small set of data, for example a set of points or an image region which is assumed to contain a specific object. The remaining processing deals with, for example:
- Verification that the data satisfy model-based and application specific assumptions.
- Estimation of application specific parameters, such as object pose or object size.
- Image recognition: classifying a detected object into different categories.
- Image registration: comparing and combining two different views of the same object.

Decision making: Making the final decision required for the application, for example:
- Pass/fail on automatic inspection applications
- Match / no-match in recognition applications
- Flag for further human review in medical, military, security and recognition applications

1.5 Historical Note

In the early years of computer vision, the paradigm for recognition was formulated as a process, which gradually and meticulously reconstructs the spatial 3D layout of the scene from the 2D image. This 3D reconstruction process was often divided into low-level, mid-level and high-level vision process. Over the years, it has become clear that this paradigm is too elaborate and complicated. Presently, the focus lies on solving recognition tasks with ‘brute-force’ approaches, e.g. based on extensive matching of image patches, for which classical techniques such as edge detection or image segmentation hardly play a role (if at all). Some of the classical techniques have therefore moved a bit into the background. This is also reflected in
recent text books. For instance, Forsyth and Ponce’s book follows the structure of the classical paradigm (low/mid/high-level vision), but the treatment of edge detection and image segmentation is rather marginal; Szeliski’s book organization is centered around the recent feature-matching approaches, but still contains substantial material on image segmentation for instance.

1.6 Prototyping in Matlab, Implementation in C

It is most convenient to test the algorithms in Matlab first, or some other higher-level language like Python, GNU Octave, R, Scilab, etc. Then, you would ‘translate’ your system in C++ for instance, to make it a real-time application.

Matlab is expensive unfortunately, but has an image processing toolbox that is very useful and hard to beat; but one can manage without the toolbox - we give plenty of code examples to do so. Matlab also features a computer vision toolbox since several years, which however is not as elaborate yet. Use doc or help to read about the functions and commands it provides. It is useful to familiarize yourself with the image processing toolbox by starting with doc images. The syntax of other software packages (R, Octave,...) is often very similar.

For implementation into C, we merely point out that there exist C libraries on the web, see for example wiki: OpenCV. Or browse through the links provided in appendix D.1. Or simply google for some expression such as ‘C code for computer vision applications’ - maybe there has appeared a new toolbox somewhere recently.

1.7 Exercises

1.7.1 Localizing Face Parts

To warm up to Matlab and some of its essential commands we construct a primitive but fairly effective algorithm for localizing facial features (face parts). Such algorithms are often used in human-computer interaction to determine precisely the location of a user’s face. Download an image database containing faces, where the face is placed approximately in the image center: google for ‘face database’; a few tens of images suffice. We will create so-called profile to localize the facial features (see figure 1 for the idea).

Figure 1: Intensity profiles for a face image. The vertical profile is generated by summing the image pixel values along the y-axis (vertical; column-wise); the horizontal profile is generated by summing along the x-axis (horizontal; row-wise). To what face parts do the extrema in the profiles correspond to? Code in appendix E.1.

1. Load and display an image (see also Appendix E.1): load an image with the command imread and assign it to variable Icol (if it is color) or I if it is a gray-scale image already. If it is a color image, then convert it to grayscale using the command rgb2gray. Display the figure, for instance figure(1); clf; imagesc(I);. Add the command colorbar if you want to check the intensity range. Add the command hold on, which allows you to continue plotting on the image.

2. Understand the coordinate system in the displayed figure: plot something, e.g. an asterisk at plot(20,40,’*’). Observe that the image is displayed in matrix mode, that is on the y-axis we have rows, on the x-axis we have columns. If you prefer to see your plots on top of the image in Cartesian coordinates you need to switch axes, either by using the axis command (e.g. axis xy) or by swapping the x and y dimensions of your plotting coordinates.
3. Generate a vertical and a horizontal intensity profile by summing up the intensity values for each dimension, \( \text{Ver} = \text{sum}(I,2) \); and \( \text{Hor} = \text{sum}(I,1) \) respectively. Plot the two profiles and you will immediately see how certain local extrema correspond to specific facial features. With the command `subplot` you can place several images or plots into the same figure; or use the command `axes` to place axes individually in a figure.

4. It is difficult to operate on the raw density profiles, because they are typically noisy and that makes local extrema detection difficult. Use the command `convn` to smoothen (low-pass filter) the profiles and `pdf` to generate a filter:

\[
\text{LowFlt} = \text{pdf('norm', -nPf:nPf, 0, round(nPf/2))}; \quad \%\text{ creates a Gaussian}
\]

\[
\text{Verf} = \text{convn(Ver, LowFlt, 'same')}; \quad \%\text{ convolution with low-pass filter}
\]

\( nPf \) is the number of points (use some proportion of the image size); note that the orientation of the vectors matters. Plot the filtered profiles into the same figure to verify visually that the filtering has worked.

5. For extrema detection you need the derivatives which you obtain using the command `diff`. Then you locate the sign changes by firstly applying the function `sign`, followed by convolving with a two-element filter \([-1\ 1] \). For instance:

\[
\text{Ex} = \text{convn}(\text{sign(Dv1)}, [-1\ 1]); \quad \%\text{ mark extrema: requires row vector!}
\]

\[
\text{bMax} = \text{Ex}==2; \quad \%\text{ maxima}
\]

\[
\text{bMin} = \text{Ex}==-2; \quad \%\text{ minima}
\]

The vector \( \text{Ex} \) holds the value of 2 for those elements where a local maximum exists, and it holds the value -2 where a minimum exists. In principle you needed to check for the boundary values as well and as for the case when two neighboring values are equal, in which case \( \text{Ex} \) will hold 1 or -1. In the case of latter, use a broader low-pass filter (step above).

6. Now you can manually construct some rules that identify facial features. There are certain appearances of users, which make identification of certain facial features difficult. Which ones are they?

As pointed out, this type of face part localization is somewhat simple, but its complexity is low and that is why this approach is often used as a first phase in a more elaborate facial feature tracking system.

### 1.7.2 Understanding Recognition

The objective of the following two study problems is to help out comprehending the differences between the recognition processes as introduced in subsection 1.2.

1. Google provides with the stand-alone program ‘Google Goggles’ an illustration of object recognition. Which exact recognition processes does it emulate?

2. There exist systems that perform object recognition for automated checkout lanes in retail. What do you think how the individual recognition processes are addressed in such a system?
### 2 Image Processing I: Scale Space and Gradient Image

For many computer vision tasks, it is useful to analyze an image at different resolutions - roughly speaking. More precisely, we intend to observe the image at different degrees of blur - squint your eyes to understand this effect. Generating those different blurs and resolutions is done with the scale space, treated in the following subsection 2.1.

For many computations it is also useful to know how the 'intensity' landscape is oriented at each pixel. This is expressed with the gradient image, which describes the 'surface slope' for a small pixel neighborhood at each image pixel. That will be treated in subsection 2.2.

#### 2.1 Scale Space, Pyramid

The scale space is a stack of images where the first image corresponds to the original image and subsequent images are increasingly blurred; the images become increasingly coarser - the corresponding intensity landscape becomes smoother. The axis representing the stack height corresponds to the smoothing parameter. To generate those coarse images, the original image is low-pass filtered. We already used the process of low-pass filtering in the previous exercise, namely to smoothen the profiles; there the process was performed in one dimension only; here the low-pass filter works along both dimensions. This blurring gives us smoother information with the following benefits:

1) Verification of structures across the 'blur' scale (axis): for instance, if a specific corner feature is present at different scales, then it is less likely to be accidental.
2) Finding more 'coherence': for instance, contours appear more 'coherent' at coarser scales.

First we introduce the typical formalism. An image $I_o(x,y)$ is blurred by low-pass filtering it with a Gaussian filter $g(x,y,\sigma)$:

$$I_c(x,y) = I_o(x,y) * g(x,y,\sigma),$$

where $\sigma$ is the standard deviation and $*$ is the convolution operator. One typically generates a fine-to-coarse space, the scale space $S(x,y,\sigma)$, which we obtain by filtering the (original) image with a filter of increasing standard deviation $\sigma$. Typically, sigma values are $\sigma = 1, 2, \ldots, 5$.

In Matlab we can create a Gaussian filter with the function fspecial and convolve the image with the command conv2:

```matlab
Fsc1 = fspecial('gaussian', [3 3], 1); % 2D gaussian with sigma=1
Fsc2 = fspecial('gaussian', [5 5], 2); % 2D " " sigma=2
Isc1 = conv2(Io, Fsc1, 'same'); % filtering at scale=1
```

The image processing toolbox command imfilter may be more convenient (if available).

#### Pyramid

Operating with the entire scale space $S$ is computationally expensive, as it is rather large and to some extent redundant. Because at coarser scales fine changes are not present, it therefore makes sense to subsample coarser scales. For instance for each new, coarser scale $I_c$, only every 2nd pixel (along the horizontal and vertical axis) is taken. We so arrive at the (octave) pyramid, e.g. $P_1=200\times300$ pixels ($I_o$), $P_2=100\times150$ (subsample of $I_{c1}$), $P_3=50\times75$ (subsample of $I_{c2}$), etc.

To downsample in Matlab we can simply select every second row and column, e.g. $I_{d2} = Isc1(1:2:end,:);$ (for rows) followed by $I_{d2} = I_{d2}(:,1:2:end);$ (for columns). In Matlab there exists also the function downsample which however works only for rows only if the input is a matrix. If other subsampling steps are required then the function impyramid may be more convenient.

#### Application

In many matching tasks, it is more efficient to search for a pattern starting with the top of the pyramid, the smallest level, and then to work downward to the larger levels, a strategy also called coarse-to-fine matching. More specifically, if the pattern we search for does not look promising at a coarse level, we keep on searching only at a coarse level which is time efficient. Only when a potential detection was made at the coarse level, then one starts to verify by moving towards finer levels, which is more time consuming.
Implementation \(\text{wiki: Scale\_space\_implementation}\)

2.2 Gradient Image \(\text{wiki: Image\_gradient}\)

The gradient image describes the steepness at each point in the intensity landscape, more specifically how the local ‘surface’ of the landscape is inclined. In a first step, one determines the gradient (derivative) in both dimensions, that is the difference between neighboring pixels along both axes, \(\frac{\partial I}{\partial x}\) and \(\frac{\partial I}{\partial y}\) respectively. This operation is typically expressed with the nabla sign \(\nabla\), the gradient operator:

\[
\nabla I = \left( \frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right)^T.
\]

(2)

whereby the gradient information is expressed as a vector; for an entire this is visually illustrated as a vector field, see figure 3. Given this vector (at each pixel), we can now compute the direction and the magnitude of the local intensity ‘surface’. We give examples: a point in a plane has no inclination, hence no magnitude and an irrelevant direction - the gradient is zero; a point in a slope has a certain direction - an angle value out of a range of \(2\pi\) - and a certain magnitude representing the steepness. The direction is computed with the arctangent function using two arguments \(\text{atan2}\), returning a value \(\in [-\pi, \pi]\) in most software implementations. The magnitude, \(\| \nabla I \|\), is computed using the Pythagorean formula.
imgradientxy to obtain two matrices corresponding to the individual gradients. If we lack the toolbox, we can use the following piece of code, whereby the function gradient returns two matrices - of the size of the image - which represent the gradients along the two dimensions:

```matlab
Fg = fspecial('gaussian',[3 3],1); % 2D gaussian of size 3x3 with sigma=1
Isc1 = conv2(I, Fg, 'same');
[Dx Dy] = gradient(single(Isc1)); % gradient along both dimensions
Dir = atan2(-Dy, Dx) + pi; % [-pi,pi]
bk0 = Dir<0; % negative values
Dir(bk0) = Dir(bk0)+2*pi; % [0,2*pi]
Mag = sqrt(Dx.^2+Dy.^2); % gradient magnitude
```

In the example code, the angle values are shifted into the positive range $[0,..,2\pi]$. One can plot the gradient field using the function quiver:

```matlab
% --- Plotting:
X = 1:ISize(2); Y = 1:ISize(1);
figure(1); clf; colormap(gray);
imagesc(I, [0 255]); hold on;
quiver(X, Y, Dx, Dy);
```

The gradient image is used for a variety of tasks such as edge detection, feature detection and feature description (coming up).

### 2.3 Exercises

We are going to write a few functions, but instead of placing the code immediately into separate function scripts, we firstly develop our code in a testing script, which we call t_NameOfScript (t for test), because debugging is easier in a script than in a function script. Once we have developed our code sufficiently well in a script, then we copy the commands into a function script called f_NameOfFunction (f for function). Do not erase the code in your testing script - leave it there! You then apply the function f_NameOfFunction at the end of your testing script to verify its output, e.g. by taking the differences of the output in the testing script and the output of the function script.

1. Write a script t_Pyramid, which generates a pyramid for an image (see appendix E.2 for an example). Store the levels in a cell or struct. Display the pyramid with subplot. What (intensity) range do the different levels have? - use colorbar again to illustrate. A value of 0 is what color? What is the maximal value? Place the code into a function f_Pyramid.

2. Write a script t_GradImg which generates a gradient image, see subsection 2.2 for hints. Display the maps for directions and magnitudes in separate images (imagesc). Use colorbar to verify the ranges. Where are the magnitude values the largest?

Now display the gradients - direction and magnitude - with the command quiver and place the output on top of the image (don’t forget hold on). For large images it may take a long time - several seconds - to display all the arrows. Zoom into the results to verify the output. In what direction do the arrows point? Upward, downward in the intensity image? To what measure corresponds the length of the arrows?

Finally, write the function f_GradImg, which takes as input the image I and returns two maps, one with the direction values and one with the magnitudes. Return the maps in a structure GM: GM.Dir = Dir, GM.Mag = Mag, it is handier for later use. Now verify the function at the end of your testing script with any(GM.Dir(:)-Dir(:)) if 0 then it is ok, if not, then we may have an error - though there may be differences in rounding; verify with any(sum(abs(GM.Dir(:)-Dir(:)))>0.1).
3 Feature Extraction I: Patches

Feature extraction is the process of finding suitable features, that facilitate recognition. Features can be dots, blobs (small undefined shapes outlining a region), edges, corner, boundaries, regions and (square) image patches and transformations thereof. In this section we only deal with the latter, as those are the most frequently used ones.

Patches are local features, typically small square-sized pixel arrays of the image; they are taken at points where there appears to be a corner or other 'interesting' structure (figure 4), hence they are also called 'corners', 'interest points' or 'keypoint features', hereafter simply called 'features'. The patches are often described by their appearance, e.g. a histogram of the local gradients, which makes them relatively unique and distinct and thus suitable for matching between images. Patches are applied in feature-based correspondence techniques such as stereo matching, image stitching, fully automated 3D modeling, object instance and category recognition as well as video stabilization. A key advantage of using matching with sets of keypoints is that it permits finding correspondences even in the presence of clutter (occlusion) and large scale and orientation changes.

The process of finding and matching keypoints consists of three stages:

1. Feature detection: search for unique patch locations, that are likely to match well in other images.
2. Feature description: conversion of the patch into a more compact and stable (invariant) descriptor that can be matched against other descriptors.
3. Feature matching: weighting of feature descriptors and matching with descriptors of other images.

3.1 Detection

One way to find corners is to find edges, and then walk the edges looking for a corner. This approach can work poorly, because edge detectors often fail at corners. Also, at very sharp corners or unfortunately oriented corners, gradient estimates are poor, because the smoothing region covers the corner. At a 'regular' corner, we expect two important effects. First, there should be large gradients. Second, in a small neighborhood, the gradient orientation should swing sharply. We can identify corners by looking at variations in orientation within a window, which can be done by autocorrelating the gradients:

\[
\mathcal{H} = \sum_{\text{window}} \{ (\nabla I)(\nabla I)^T \} \approx \sum_{\text{window}} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}
\]

whereby \( I_x = I_o \ast \frac{\partial g}{\partial x} \) and \( I_y = I_o \ast \frac{\partial g}{\partial y} \) (\( g \) is a Gaussian). In a window of constant gray level (that is without any strong gradient), both eigenvalues of this matrix are small because all the terms are small. In a window containing an edge, we expect to see one large eigenvalue associated with gradients at the edge and one small eigenvalue because few gradients run in other directions. But in a window containing a corner, both eigenvalues should be large. The Harris corner detector looks for local maxima of

\[
C = \det(\mathcal{H}) - k \left( \frac{\text{trace}(\mathcal{H})}{2} \right)^2
\]
where \( k \) is some constant, typically set between 0.04 and 0.06. The detector tests whether the product of the eigenvalues (which is \( \det(\mathcal{H}) \)) is larger than the square of the average (which is \( (\text{trace}(\mathcal{H})/2)^2 \)). Large, locally maximal values of this test function imply the eigenvalues are both big, which is what we want. These local maxima are then tested against a threshold. This detector is unaffected by translation and rotation.

**Algorithm 1 Feature detection.**

1. Compute the horizontal and vertical derivatives \( I_x \) and \( I_y \) of the original image by convolving it with derivatives of Gaussians.
2. Compute the three images corresponding to the outer products of these gradients.
3. Convolve each of these images with a larger Gaussian.
4. Compute a scalar interest measure using for instance equation 5.
5. Find local maxima above a certain threshold and report them as detected feature point locations.

Matlab’s image processing toolbox provides the command `corner`; this is how we would code it ourselves:

```matlab
%% ----------- Step 1
gx = repmat([-1 0 1],3,1); % derivative of Gaussian (approximation)
gy = gx';
Ix = conv2(I, gx, 'same');
Iy = conv2(I, gy, 'same');
%% ----------- Step 2 & 3
Glrg = fspecial('gaussian', max(1,fix(6*sigma)), sigma); % Gaussian Filter
Ix2 = conv2(Ix.^2, Glrg, 'same');
Iy2 = conv2(Iy.^2, Glrg, 'same');
Ixy = conv2(Ix.*Iy, Glrg, 'same');
%% ----------- Step 4
k = 0.04;
HRS = (Ix2.*Iy2 - Ixy.^2) - k*(Ix2 + Iy2).^2;
```

As noted above already, we have presented the working principles of the Harris corner detector. There are many types of feature detectors but they are all based on some manipulation of the gradient image (see for instance Dav p177, s6.7.6). See also website links in FoPo p190,s5.5 for code examples.

To now select corners in the ‘corner image’ (\( \text{HRS} \)) (step 5), we select maxima and suppress their neighborhood to avoid the selection of very near-by values. Here’s a very primitive selection mechanism:

```matlab
%% ----------- Step 5
% Extract local maxima by performing a grey scale morphological % dilation and then finding points in the corner strength image that % match the dilated image and are also greater than the threshold.
sze = 2*radius+1; % size of dilation mask.
Hmx = ordfilt2(HRS,sze^2,ones(sze)); % grayscale dilate.
% Make mask to exclude points within radius of the image boundary.
bordermask = zeros(size(HRS));
bordermask(radius+1:end-radius, radius+1:end-radius) = 1;
% Find maxima, threshold, and apply bordermask bHRS = (HRS==Hmx) & (HRS>thresh) & bordermask;
[r,c] = find(bHRS); % find row,col coords.
PtsIts = [r c]; % list of interest points [nPts 2]
```
Feature Tracking: Most features found at coarse levels of smoothing are associated with large, high-contrast image events because for a feature to be marked at a coarse scale, a large pool of pixels need to agree that it is there. Typically, finding coarse-scale phenomena misestimates both the size and location of a feature. At fine scales, there are many features, some of which are associated with smaller, low-contrast events. One strategy for improving a set of features obtained at a fine scale is to track features across scales to a coarser scale and accept only the fine-scale features that have identifiable parents at a coarser scale. This strategy, known as feature tracking in principle, can suppress features resulting from textured regions (often referred to as noise) and features resulting from real noise.

3.2 Description

The most popular descriptor is the scale invariant feature transform (SIFT), which is formed as follows:

a) take the gradient (0-360 deg) at each pixel (from $\nabla I$) in a $16 \times 16$ window around the detected keypoint (subsection 2.2), using the appropriate level of the Gaussian pyramid at which the keypoint was detected.

b) the gradient magnitudes are downweighted by a Gaussian fall-off function (shown as a circle in figure 5) in order to reduce the influence of gradients far from the center, as these are more affected by small misregistrations.

c) in each $4 \times 4$ quadrant, a gradient orientation histogram is formed by (conceptually) adding the weighted gradient value to one of 8 orientation histogram bins. To reduce the effects of location and dominant orientation misestimation, each of the original 256 weighted gradient magnitudes is softly added to $2 \times 2 \times 2$ histogram bins using trilinear interpolation. (Softly distributing values to adjacent histogram bins is generally a good idea in any application where histograms are being computed).

d) form an $4 \cdot 4 \cdot 8$ component vector $v$ by concatenating the histograms: the resulting 128 non-negative values form a raw version of the SIFT descriptor vector. To reduce the effects of contrast or gain (additive variations are already removed by the gradient), the 128-D vector is normalized to unit length: $u = v / \sqrt{v \cdot v}$.

e) to further make the descriptor robust to other photometric variations, values are clipped to $t = 0.2$: form $w$ whose $i$th element $w_i$ is $\min(u_i, t)$. The resulting vector is once again renormalized to unit length: $d = w / \sqrt{w \cdot w}$.

Figure 5: Forming SIFT features. Left: Gaussian weighting (for a 8x8 field in this illustration). Right: formation of histograms demonstrated on 2x2 quadrants. [Source: Szeliski 2011; Fig 4.18]

The following code fragments give an idea of how to implement steps a-c:

```matlab
EdGrad = linspace(0,2*pi,8); % edges to create 8 bins
[yo xo] = deal(pt(1),pt(2)); % coordinates of an interest point
Pdir = Gbv.Dir(yo-7:yo+8,xo-7:xo+8); % 16 x 16 array from the dir map
Pmag = Gbv.Mag(yo-7:yo+8,xo-7:xo+8); % 16 x 16 array from the mag map
Pw = Pmag .* fspecial('gaussian',16, 4); % weighting center
BLKmag = im2col(Pw, [4 4], 'distinct'); % quadrants columnwise for magnitude
BLKdir = im2col(Pdir, [4 4], 'distinct'); % quadrants columnwise for direction
Gh = [];
for k = 1:16
    [HDir Bin] = histc(BLKdir(:,k), EdGrad);
    HDw = accumarray(Bin, round(BLKmag(:,k)*10000), [8 1]);
```
\[ G_h = [G_h; HDw]; \]

end

3.3 Matching

To compare two descriptor lists (originating from two different images for instance), \( d_i \) and \( d_j \) \((i = 1, \ldots, k, j = 1, \ldots, l)\), we take the pairwise distances and form a \( k \times l \) distance matrix \( D_{ij} \). Then we take the minimum in relation to one descriptor list, e.g. \( \min_i D_{ij} \), and obtain the closest descriptor from the other descriptor list. That would be a simple correspondence and may suffice if we compare a list of image descriptors and a list of category descriptors, as we will do for image classification (section 4). If we intend to establish correspondences for registration (section 11), we want to find the mutual matches.

\[
\begin{align*}
L1, L2: & \text{ the 2 descriptor lists, [nD1 x nDim] and [nD2 x nDim]} \\
% \text{----- Compact version:} \\
DM &= pdist2(L1,L2); \\
% \text{----- Explicit version: (building DM ourselves)} \\
DM &= zeros(nD1,nD2); \\
\text{for } i = 1 : nD1 \\
& \quad iL1rep = repmat(L1(i,:), nD2, 1); \quad \% \text{replicate individual vector of } L1 \text{ to size of } L2 \\
& \quad Di = sqrt(sum((iL1rep - L2).^2,2)); \quad \% \text{Euclidean distance} \\
& \quad DM(i,:) = Di; \quad \% \text{assign to distance matrix} \\
\end{align*}
\]

% \text{----- Correspondence with respect to one list} \\
[MX2 IX1] = min(DM,[],1); \quad \% [1 x nD2] \\
[MX1 IX2] = min(DM,[],2); \quad \% [nD1 x 1] \\
% \text{----- Correspondence mutual} \\
IXP1to2 = [(1:nD1)', IX2]; \quad \% [nD1 x 2] \text{ pairs with indices of 1st list and minima of 2nd list} \\
IXP2to1 = [(1:nD2)', IX1']; \quad \% [nD2 x 2] \text{ pairs with indices of 2nd list and minima of 1st list} \\
BMut1 = ismember(IXP1to2, IXP2to1(:,[2 1]), 'rows'); \quad \% \text{binary array of mutual matches in list 1} \\
IXMut1 = find(BMut1); \quad \% \text{mutually corresponding pairs with indexing to list 1} \\
IXPMut1 = IXP1to2(IXMut1,:); \quad \% [nMut1 x 2] \text{ mutual pairs of list 1} \\
\]

One may also want to use a for-loop for the maximum operation, that is to take the maximum row-wise (to avoid costly memory allocation).

**Note** For large databases with thousands of vectors, these “explicit but precise” distance measurements are too slow anyway. Instead, faster but slightly inaccurate methods are used, as for instance hashing functions or kd-trees (see advanced course).

3.4 Notes

The advantage of the previously introduced histograms of gradients are their versatility, that is they can be used in many computer vision tasks. Their downside is that they cannot be used for finding actual object outlines (nor are they considered a credible approach to a more capable recognition system).

C implementations with Matlab interface can be found here for instance:

http://www.vlfeat.org/
http://www.aishack.in/2010/05/sift-scale-invariant-feature-transform/

See also links in Appendix D.1.
3.5 Exercises

1. Write a script t_CornerDet, which detects interest points in an image, that is implement algorithm 1, see code hints in subsection 3.1. Display the corner map HRS: what is its range? How do you find the maxima? To find maxima you could apply different methods, for instance using morphological operations (bwmorph), or the statistical solution as given under step 5 in subsection 3.1. Name the variable for detected points PtsInt (no. points x 2 [row/column]). Display the detected points on the gray-scale image - be aware of axes differences (Cartesian and matrix axes). Use the plotting option 'markersize' to draw markers whose size corresponds to the degree of interest, the corresponding value in the corner map HRS.

Exclude interest points, that lie too close to the image bounds by either applying another bordermask (see step 5 in 3.1), or by operating on the list of interest points:

   [nRow nCol] = size(Image);                     % no. of rows and columns of the images
   ms = 8;                                        % margin size
   b_NearHorz = PtsInt(:,1)<ms+1 & PtsInt(:,1)>nRow-ms;  % near horizontal borders
   b_NearVert = PtsInt(:,2)<ms+1 & PtsInt(:,2)>nCol-ms;  % near vertical borders
   b_Near  = b_NearHorz | b_NearVert;              % near any border
   PtsInt = PtsInt(~b_Near,:);                    % keeping inside border

Finally copy the code into a function f_CornerDet, whose input is the image I and the parameters k, radius and thres, and whose output is a list of interest points PtsInt. Play with the parameters and observe the changes, e.g. process the image with three different values for radius and plot the output in different images. Do the same for k and thres.

2. Write a script t_DescGradHist, which generates the gradient descriptors as treated under subsection 3.2. For each interest point of PtsInt, we take its 16 x 16 pixel neighborhood and generate the 1 x 128-dimensional gradient histogram. Display the histograms using the command imagesc. What is the maximum amplitude of all histograms? If all looks good, create the corresponding function f_DescGradHist, whose input are the image and the list of interest points, and the output is the list (matrix) of vectors DSC; choose the format number of points (rows) times number of dimensions (columns).

3. Write a script t_DescMatch, which matches the descriptor lists of a pair of similar images, DSC1 and DSC2. Use the pdist2 command to find the pairwise distances. Caution! this step is memory consuming: make sure you obtain fewer than 2000 interest points per image, i.e. use imresize to scale the image. To save further on memory, use only single precision instead of the default double precision, because we do not really require double precision in our application. Plot the distance matrix, check its data range.

To find the mutual correspondences, it is useful to firstly study a mock example. Open a script t_indexing and take two short, random lists to clarify the assignments, e.g. nD1=5; nD2=8; L1 = rand(nD1,3); L2 = rand(nD2,3);, where dimensionality is only 3 to keep everything simple. Generate the distance matrix and determine the mutual correspondences as exemplified in subsection 3.3). Study the method exactly. Then apply to your real-word example. Sort the minimum distances for the mutual matches in increasing order and plot them.

Verification I: To verify visually, plot some of the best matching, corresponding pairs. Place the two images into one figure, then plot the first best 30 matches and denote the corresponding point pairs in the two images with a text label using text. We expect that some will match and some won't due to variability. Pay attention to the axes (x/y versus row/column, see also note in subsection 1.6).

Verification II: if descriptors do not match, then take the same image twice - that should return exact correspondence.

If some points match, copy the code into a function f_DescMatch: its input is the two list of descriptors DSC1 and DSC2, the output are arrays with corresponding point indices for the best matches.
4. It is worthwhile documenting the above functions. Write for each one an explanation of what it does (1-3 sentences) into its header. Specify the dimensions of the input and output variables.

It is always useful to display some text output in the command window. Use tic and toc to measure how long the functions take to process an image. Display how many interest points are detected with fprintf. Determine and display the minimum and maximum distances between features. And so on.

5. Take two slightly different images, e.g. room scene photographed from 2 slightly different angles, for instance take a step sideways for the second shot (angle change of few degrees is sufficient). Plot the results in two different figures and observe again how the contour output differs amongst the two images. Notice the variability - it is one of the reasons that makes recognition in images so difficult.
We now move towards representations for objects and scenes using the patches as obtained in the previous section 3. A generic way to do this is to collect a large number of patches for a category and to find clusters within them, that are representative for that category. To illustrate the idea, we look at an example from image compression, specifically color encoding:

**Example Quantization** An image is stored with 24 bits/pixel and can have up to 16 million colors. Assume we have a color screen with 8 bits/pixel that can display only 256 colors. We want to find the best 256 colors among all 16 million colors such that the image using only the 256 colors in the palette looks as close as possible to the original image. This is color quantization where we map from high to lower resolution. In the general case, the aim is to map from a continuous space to a discrete space; this process is called vector quantization. Of course we can always quantize uniformly, but this wastes the colormap by assigning entries to colors not existing in the image, or would not assign extra entries to colors frequently used in the image. For example, if the image is a seascape, we expect to see many shades of blue and maybe no red. So the distribution of the colormap entries should reflect the original density as close as possible placing many entries in high-density regions, discarding regions where there is no data. Color quantization is typically done with the k-Means clustering technique.

**The Principal Applied to Features** In our case, we aim to find clusters amongst our features that represent typical ‘parts’ of objects, or typical ‘objects’ of scenes. In the domain of image classification and object recognition, these clusters are sometimes also called (visual) ‘words’, as their presence or absences in an image, corresponds to the presence or absence of words in a document; in texture recognition they are also called ‘textons’. The list of words represents a ‘pooled’ representation or a ‘dictionary’ (aka ‘bag of words’), with which we attempt to recognize objects and scenes. Thus, in order to apply this principal, there are two phases: one is building a dictionary, and the other is applying it; which would correspond to training and testing in machine learning terminology. Figure 6 summarizes the approach. We will merely point out how to use the machine learning techniques and omit lengthy explanations in order to progress with the concepts in computer vision.

### 4.1 Building a Dictionary

We quantize features as follows:

1. Collect patches $x_i(d)$ from images (or image patches for objects) of the same category, e.g. $n_p$ in total ($i = 1..n_p$). These patches can be represented in various ways, e.g. only by pixel values, in which case the dimensionality $n_d$ corresponds to the number of pixels ($d = 1..n_d$); or by a SIFT histogram, in which case the dimensionality corresponds to the histogram length ($n_d = 128$ as for original SIFT features). Normalization can sometimes improve performance - try different normalization schemes.

2. This step is optional. You may want to try a principal component analysis (PCA) to reduce the dimensionality of your features. This maybe in particularly useful if you use pixel intensities as dimensions only, but can also be tried for SIFT features. We now denote the reduced dimensionality with $n_r$ and the reduced vectors as $x_i^r(d)$ with $d = 1..n_r$.

3. Quantize vectors with a clustering technique and obtain a list of clusters $c_j(d)$ with $j = 1..n_c$. In the simplest case, a cluster is represented by the mean of its members. Determine a threshold that decides when a tested feature is close enough to the center.

**Algorithm 2** Building a dictionary for a category with $x_i \in D^l$. Compare with upper half of figure 6.

1) Collect many training patches $x_i(d)$ ($i = 1..n_p$; $d = 1..n_d$)

2) Optional: apply the PCA: $x_i(d) \rightarrow x_i^r(d)$ ($d = 1..n_r$)

3) Find $k$ ($n_c$) cluster centers $c_j(d)$ ($j = 1..n_c$; $d = 1..n_r$)
4.1.1 Vector-Quantization using the k-Means Algorithm

The most frequent clustering technique used for vector quantization is the k-Means algorithm, a widely used machine learning technique. For this technique we need to provide the number of expected cluster \( k \), hence the name k-Means, which we here however denote as \( n_c \). In other words, we have to estimate the number of words of our dictionary prior to its usage. This is a bit odd as we rather prefer an algorithm that choses the optimal number of clusters automatically, but unfortunately we have to determine that number ourselves by systematic testing.

K-means is a ‘quick-and-dirty’ method to cluster - in comparison to computationally more expensive and slower algorithms. It is an iterative procedure in which the cluster centers and sizes are gradually evolved by comparing the individual data points (vectors) sequentially. The procedure starts by randomly selecting \( n_c \) data points (from \( n_t \) total data points), which are taken as initial cluster centers. Then, the remaining data points are assigned to the nearest cluster centers and the new cluster centers are determined. Because the new cluster centers will be in slightly different locations, a new assignment labeling is carried out and the new centers determined, etc. The most beautiful illustration for this process is in Forsyth/Ponce 2010; Fig 6.8.

To apply this technique in Matlab we organize our patches (features) in a \( n_F \times n_D \) matrix \( FTS \). We then cluster with the command \( \text{kmeans} \): \( \text{Ixs} = \text{kmeans}(FTS, k) \), whereby \( \text{Ixs} \) is a vector of length \( n_F \) containing the cluster indices (\( \in 1..k|n_c| \)). See also appendix A for more details. There are different ways how to calculate the clusters during evolution, see the options of \( \text{kmeans} \).

4.2 Applying the Dictionary to an Image

One collects patches from a testing image, vector quantizes them by identifying the index of the closest cluster center, then computes a histogram with bins corresponding to the cluster indices that occur within
the image. A bit more elaborate and step by step:

1. For a given testing image (or object), find interest points and describe the patches around them: \(v_m(d)\) with \(m = 1..n_q\). Apply the PCA (if it was used before): \(\to v_m^r(d)\) with \(d = 1..n_r\).

2. For each patch \(v_m\) (or \(v_m^r\)) find the nearest cluster center \(c_j\) \((j = 1..n_c)\) - if there is one (thresholding!).

3. Create a histogram \(H(j)\), which counts the occurrences of (quantized) features, that is the cluster centers. The total histogram count is less equals \(n_q\) as some features may not have exceeded the threshold (ideally none for the features of a different category).

Histograms can now be used for classification or retrieval.

Algorithm 3 Applying the dictionary (for one category), \(x_i \in D^T\). Compare with lower half of figure 6.

1) For each relevant pixel \(m\) in the image: compute the vector representation \(v_m\) around that pixel
2) Obtain \(j\), the index of the cluster center \(c_j\) closest to that feature
3) Add a value of one to the corresponding bin the histogram \(H(j)\).

4.3 Classification

A classifier is trained (or learned) with a so-called training dataset. To estimate its prediction performance it is applied to a so-called testing (or sampling) set. It requires a training and a testing set: the classifier is learned on the training set and then its performance is verified on the testing set. See also appendix A for implementation details. When working with a dictionary, we need to partition the training set as well: one partition is used for building the dictionary, the other partition is used for generating histograms as training ‘material’ for the classifier.

Example: We have 30 images per category. For each category we use 25 instances for training and 5 for testing. Of the 25 training instances, we use 5 for building the dictionary (algorithm 2), the other 20 are used for generating histograms (algorithm 3). The actual classifier is then trained with those 20 histogram vectors and tested on the 5 training histograms, which were also generated with algorithm 3. We do this for 3-5 folds (see appendix).

As you may have noticed, there are many parameters that influence performance. The optimization of such a system is equally challenging (if not even more) as developing just the system - hence enthusiasm to deal with much code is of benefit. For the moment, we attempt to get the classification system going with a moderate performance and leave fine tuning to experts in classification. We mention here only that applying the principal component analysis may result in the largest performance improvement as well as the tuning of the feature thresholds.

4.4 Principal Component Analysis (PCA)

The PCA transform can provide us with a more compact representation, that is we transform the data such that we can omit seemingly irrelevant dimensions. In some cases, we have to apply the PCA, otherwise the classifier will not work at all. But using the PCA, we possibly loose somewhat of the discriminatory power of the higher dimensional space. See appendix for implementation details.

To build a good classifier, the PCA is applied to the training set only. The resulting coefficients are then used to transform the testing set.
4.5 Exercises

Let us apply a dictionary (‘bag of words’) to image/object classification. Download a database with at least 5 classes and at least 25 testing images per class (with fewer training images it is difficult to train a classifier).

Examples:
- The urban and natural scene collection: http://cvcl.mit.edu/database.htm. 60 percent can be easily reached with 25 training images and 5 testing images (6-fold crossvalidation).
- Satellite images: http://vision.ucmerced.edu/datasets/landuse.html. 70 percent can be easily reached using all 100 images per category. Try with 25 training and 5 testing images per category (6-fold crossvalidation).
- The Caltech 101 collection: http://en.wikipedia.org/wiki/Caltech_101. Choose a subset of categories if you prefer. (This one’s hard even though the categories are objects only and are centered in the image).
- Or you may even photograph your own collection, which is more exciting but also time-consuming.

To preprocess, match and classify the images we break down the recognition sequence into several scripts:

1. Write a script `img_desc`, which extracts the interest points and corresponding descriptors for each image. Loop over the images and save the descriptors as `Desc` with the command `save`. Create a directory `DatDesc` where you place the individual files. To save on memory, save the descriptors in single format by converting them by calling `Desc = single(Desc)`.

2. Write a script `img_concat`, which loads the individual descriptor lists and concatenates them into a single array of size: total no. descriptors for collection x no. dimensions. Speed up this procedure by initializing with `DESC = zeros(nDscTotEst,nDim,'single');` where `nDscTotEst` is an overestimation of the total number of descriptors for the collection. You trim the list `DESC` to the correct size at the end of the loop. Make sure you save as data type `single`.

   Along with concatenating all image descriptor lists, also create an array of indices `IXD` of size ‘total no. descriptors for collection x 3’, where the first column corresponds to the image index, the 2nd column to the feature index and the 3rd is the category label.

3. Write a script `desc_cluster`, which loads the entire descriptor list `DESC` and perform clustering on it, see appendix A. Use `kmeans` to find clusters. Generate three clusterings for `k` equal 25, 50 and 100 as the number of clusters, which correspond to three different sets of words. Create an average vector for each cluster (word).

4. Write a script `img_match`, which matches the average cluster vectors (words) against the descriptor list of each image. For each image determine the word count.

5. Write a script `img_classif`, which classifies the images using the word vectors. Try first without principal component analysis.
5 Object Detection

Object detection is the (repeated) localization of an object category and the counting of the number of its occurrences. Examples: face detection, pedestrian detection. The principal technique is to apply a classifier to individual windows of an image, whereby windows typically overlap and are taken from a grid; coarsely speaking, it is a template-matching technique. For the training phase, we collect two datasets of image windows: one set contains the object of (relatively) fixed size (say, $n \times m$) and is centered within the window; the other set contains non-object (‘distractor’) images. We then train a classifier to discriminate between these two sets of windows (classes). In the testing phase, we pass $n \times m$ windows of a new image to the classifier; the window is moved by a step size of few pixels to speed up the search (e.g. $\Delta x$ and $\Delta y = 3$ pixels). There are 3 challenges with this technique:

1) Size invariance: the detection system should be invariant to object size. This can be achieved by a search over scale, meaning by using the pyramid (subsection 2.1): to find large objects, we search on coarser scales (layers), to find small objects we search on a finer scales. Put differently, we apply the $n \times m$ window in each layer of the pyramid.

2) Avoiding multiple counts: the very same object instance in an image should not be counted multiple times, which may happen due to the sliding search: the smaller the step sizes, the higher the chance for repeated detection. To avoid multiple counts, the neighboring windows are suppressed, when a local maximum was detected, also called nonmaximum suppression.

3) Accelerating spatial search: searching for a match in the highest image resolution is time consuming and it is more efficient to search for a match in the top pyramidal layers first and then to verify on lower layers (finer scales), that means by working top-down through the pyramid, e.g. first $P_3$, then $P_2$, etc. This strategy is also known as coarse-to-fine matching.

The technique in summary:

**Algorithm 4** Sliding window technique for object detection.

---

**TRAINING:** Train a (binary) classifier on $n \times m$ image windows with positive (object) examples and windows with negative (non-object) examples.

---

**TESTING:**

**Parameters** detection threshold $t$, step sizes $\Delta x$ and $\Delta y$

1) Construct an image pyramid.

2) For each level of the pyramid:
   - apply the classifier to each $n \times m$ window (moving by $\Delta x$ and $\Delta y$) and obtain strength $c$.
   - if $c > t$, then add window to a list $L$ including response value $c$.

3) Rank list $L$ in decreasing order of $c$ values $\rightarrow L^{eq}$.

4) For each window $W$ in sequence $L^{eq}$ (starting with maximal $c$):
   - remove all windows $U \neq W$ that overlap $W$ significantly where the overlap is computed in the original image by expanding windows in coarser scales $\rightarrow L^{red}$.

$L^{red}$ is the list of detected objects.

---

There are obviously tradeoffs between search parameters (e.g. step sizes) and system performance (e.g. detection and localization accuracy). For example, if we work with training windows that tightly surround the object, then we might be able to improve object/distractor discrimination, but we will have to use smaller step sizes for an actual performance improvement. Vice versa, if we use windows that surround the object only loosely, then we can use smaller steps sizes but our discrimination and localization performance suffers.

Matlab: see ‘Neighborhood and Block Operations’ under the image processing toolbox.

5.1 Face Detection

A typical face detection system uses the following learning tricks to improve performance:

a) non-face images are collected from aerial images or vegetation for instance (figure 7b).
b) the set of collected face images is augmented by artificially mirroring, rotating, scaling, and translating the images by small amounts to make the face detectors less sensitive to such effects (figure 7a).

c) after an initial set of training images has been collected, some optional pre-processing can be performed, such as subtracting an average gradient (linear function) from the image to compensate for global shading effects and using histogram equalization to compensate for varying camera contrast (figure 7c).

Figure 7: Pre-processing stages for face detector training (Rowley, Baluja, and Kanade 1998a): a) artificially mirroring, rotating, scaling, and translating training images to generate a training set with larger variability; b) using images without faces (looking up at a tree) to generate non-face examples; c) pre-processing the patches by subtracting a best fit linear function (constant gradient) and histogram equalizing. [Source: Szeliski 2011; Fig 14.3]

Viola-Jones Algorithm The most frequently used face detection algorithm is probably the one by Viola and Jones. It uses features consisting of 2-4 rectangular patches of different polarity, see upper row of figure 8. The pixels inside the white rectangles are subtracted from the pixels inside the black pixels; rectangles are computed with the integral image (subsection 5.1.1). To find out which combinations of rectangles (orientations and size) are representative for a category, it is necessary to try out all combinations, which is a very time-intensive procedure (despite the rapid computation of rectangles). This feature selection can be done with a ‘boosting’ classifier. The most significant 2 features are shown in figure 8; there exist also a number of other less significant features. Testing an image occurs very rapidly by searching for the most significant features first; if they are present, the search continues; if they are not present, the search is stopped.

The primary advantage of this detection system is that it is extremely fast and runs in real time. The downside of the system is that it detects only vertically oriented faces (the majority of faces is vertically oriented anyway), and the long learning duration as just mentioned.

Figure 8: Face detection with groupings of rectangular patches. Top row: the 2 most significant rectangle-based combinations (in isolation). The horizontally oriented feature represents the eyes and the cheekbones; the vertically oriented ones represent the region covering left eye-nose bridge-right eye. [Source: Szeliski 2011; Fig 14.6]

Applications Face detectors are built into most of today’s digital cameras to enhance auto-focus and into video conferencing systems to center on the speaker. They are also used in consumer-level photo organization packages, such as iPhoto, Picasa, and Windows Live Photo Gallery. Finding faces and allowing users to tag them makes it easier to find photos of selected people at a later date or to automatically share them with friends. In fact, the ability to tag friends in photos is one of the more popular features on Facebook.
5.1.1 Rectangles

Rectangular regions can be detected rapidly by use of the integral image, aka summed area table, which is computed as the running sum of all the pixel values from the origin:

$$I_i(i,j) = \sum_{k=0}^{i} \sum_{l=0}^{j} I_o(k,l).$$

(6)

To find now the summed area (integral) inside a rectangle $[i_0, i_1] \times [j_0, j_1]$, we simply combine four samples from the summed area table:

$$R_s(i_0..i_1,j_0..j_1) = I_s(i_1,j_1) - I_s(i_1,j_0) - I_s(i_0,j_1) + I_s(i_0,j_0)$$

(7)

Matlab:

```
Is = cumsum(cumsum(Io,1),2); % integral image
Rp = Is(i1,j1)-Is(i1,j0)-Is(i0,j1)+Is(i0,j0); % a rectangular pixel patch
```

5.2 Pedestrian Detection

According to Dalal and Triggs, one can typify the structure of pedestrians into 'standing' and 'walking':
- standing pedestrians look like lollipops (wider upper body and narrower legs).
- walking pedestrians have a quite characteristic scissors appearance.

Dalal and Triggs use histograms of gradients (HOG) descriptors, taken from a regular grid of overlapping windows (figure 9). Windows accumulate magnitude-weighted votes for gradients at particular orientations, just as in the SIFT descriptors (see previous section). Unlike SIFT, however, which is only evaluated at interest point locations, HOGs are taken from a regular grid and their descriptor magnitudes are normalized using an even coarser grid; they are only computed at a single scale and a fixed orientation. In order to capture the subtle variations in orientation around a person's outline, a large number of orientation bins is used and no smoothing is performed in the central difference gradient computation.

![Figure 9: Left: typical pedestrian window. Center left: HOG descriptor. Each of the orientation buckets in each window is a feature, and so has a corresponding weight in the linear SVM. Center right: HOG descriptor weighted by positive weights, then visualized (so that an important feature is light). Notice how the head and shoulders curve and the lollipop shape gets strong positive weights. Right: HOG descriptor weighted by the absolute value of negative weights, which means a feature that strongly suggests a person is not present is light. Notice how a strong vertical line in the center of the window is deprecated (because it suggests the window is not centered on a person). [Source: Forsyth/Ponce 2010; Fig 17.7]](image)

Figure 9 left shows a sample input image, while Figure 9 center left shows the associated HOG descriptors. Once the descriptors have been computed, a support vector machine (SVM) is trained on the resulting high-dimensional continuous descriptor vectors. Figures 9 center right and right show the corresponding weighted HOG responses. As you can see, there are a fair number of positive responses around the head, torso, and feet of the person, and relatively few negative responses (mainly around the middle and the neck of the sweater).
Applications   Needless to say, that pedestrian detectors can be used in automotive safety applications.

5.3  Improvement by Knowing Context

The sliding window technique is obviously a bit simple. The technique works with objects that exhibit limited variability in appearance (gradients) and structure, meaning that do not deform too much. Some improvement could be made if we knew more about the scene. Let's take pedestrian detection as an example. Pedestrians (and like most objects) appear in a typical context: pedestrians are all about the same absolute size, have their feet on or close to the ground, and are usually seen outdoors, where the ground is a plane. Thus, if we knew the horizon of the ground plane and the height of the camera above that ground plane, we could exclude many windows immediately. For instance, windows whose base is above the horizon would be suspect because they would imply pedestrians in the air; windows whose base is closer to the horizon should be smaller (otherwise, we would be dealing with gigantic pedestrians). The height of the camera above the ground plane matters because in this problem there is an absolute scale, given by the average height of a pedestrian. Assume the horizon is in the center of the image. Then, for cameras that are higher above the ground plane, legitimate pedestrian windows get smaller more quickly as their base approaches the horizon. There are two strong sources of information about the horizon and the camera height. First, the textures of the ground, buildings, and sky are all different, and these can be used to make a rough decomposition of the image that suggests the horizon. Second, observing some reliable detection responses should give us clues to where the horizon lies, and how high the focal point is above the ground plane. Hoiem et al. (2008) show that these global geometric cues can be used to improve the behavior of pedestrian and car detectors (see also Hoiem et al. (2006)).

5.4  Notes

Of course, one could also try to build object representations with SIFT features (or other histograms at interest point detectors), e.g. by determining the spatial alignment (configuration) between interest points but then our system grows very large and becomes slower. We leave that for course II, where also some matching optimization methods are introduced.

Matlab  Creating windows with \texttt{im2col}:

\begin{verbatim}
Iext = zeros(size(I)+2, 'single'); % init extended image
Iext(2:end-1, 2:end-1) = I; % place I into it
Iext(:, [1 end]) = Iext(:, [2 end-1]); % extend by 1 row on each side
Iext([1 end], :) = Iext([2 end-1], :); % extend by 1 col on each side
IBLK = im2col(Iext, [3 3], 'sliding'); % [nPixperBlk x nPixperImg]
\end{verbatim}


6 Image Processing II: Segmentation & Morphology

Image segmentation is the task of delineating (partitioning) the objects or meaningful regions in an image. For a scene, image segmentation aims at isolating its objects and identifying 'background' regions, thus performing a foreground/background segregation; for an object in face of some background, image segmentation aims at finding the exact silhouette and possibly locating the object's parts. For other applications, the exact goal may differ as well. Practically speaking, segmentation is the search for groups of pixels of a certain 'coherence'. We elaborate on this in subsection 6.1.

After segmentation, we often have a binary (logical) map, also called black-white image, whereby on-pixels correspond to foreground or regions of interest and off-pixels correspond to background (values zero and one, respectively; or true and false). That map is then analyzed or manipulated, which in Matlab is carried out with commands starting with `bw` for black-white.

Morphological processing is the manipulation of images toward a specific goal; it can be used in a variety of ways. Often it is used to clean up binary images obtained from image segmentation or contour extraction (latter to be explained in subsection 7.1); thus it is often used as the final step in image segmentation. Morphological processing can also carry out filtering processes. We shall discuss this in subsection 6.2.

6.1 Segmentation

Image segmentation is one of the oldest and most widely studied problems in computer vision. It was once thought to be an essential, early step in a systematic reconstruction of the semantic image content. But due to the difficulties of obtaining consistent segmentation across different image types (scenes, objects, texture,...), which would correspond to human interpretation -, it has lost its significance for recognition. Nowadays much recognition is performed without any segmentation - as was presented in the first few sections -, but segmentation algorithms are used for instance in medical image analysis (e.g. in X-rays) and in consumer applications, where users initiate segmentation by pointing out which regions are to be segmented.

If we do not require precise segmentation but merely a coarse localization of objects, thresholding as mentioned in subsection 6.1.1 can be enough. If we desire reasonably precise outlines, then region growing as introduced in subsection 6.1.2 could be more suitable. If we desire a precise segregation between a large foreground object and its background, then a statistical classifier can be of benefit (subsection 6.1.3).

6.1.1 Thresholding (Global, Local, Band, Multi-Level)

For many machine vision applications, it suffices to apply a simple threshold to the intensity image. A suitable threshold can be found by looking at the intensity distribution of the image: if the objects and background have two distinct colors, then the distribution is bimodal and we chose the minimum between the two modes as the threshold. In Matlab one can look at the intensity histogram with the function `imhist`, see also SHB p.24, s.2.3.2. Matlab provides the function `graythresh`, which finds an optimal threshold between two peaks.

Applying a single threshold to the entire image is also called global thresholding. One can also apply a local threshold, whose value depends on a window of the image around the thresholded pixel. A band threshold sets pixel intensity values outside a range (band) to zero.

For more complex scenes, thresholding is of limited use, but can be exploited for locating objects (or regions), that possess a relatively distinct gray-level. For instance, thresholding has been applied for road segmentation and for vehicle location in in-vehicle vision systems. The intensity distribution for such images is often multi-modal and methods to identify the correct threshold or range of values are sometimes called multi-level thresholding. One possible step toward that goal would be distribution filtering and extrema detection as we did for face part detection (exercise 1.7.1).

Matlab `imhist, graythresh, im2bw`

Advantages speed (real time)

Disadvantages overly simple for complex scenes; only a coarse segmentation
6.1.2 Region Growing: The Watershed Algorithm

A technique related to thresholding, since it operates on a grayscale image, is watershed computation (Vincent and Soille 1991). This technique segments an image into several catchment basins, which are the regions of an image (interpreted as a height field or landscape) where rain would flow into the same lake. An efficient way to compute such regions is to start flooding the landscape at all of the local minima and...
to label ridges, the watersheds, wherever differently evolving components meet. Watershed segmentation is usually applied to a smoothed version of the gradient magnitude image ($\| \nabla I \|$), subsection 2.2), thus finding smooth regions separated by visible (higher gradient) boundaries (which also makes it usable with color images).

Figure 11: One-dimensional example of watershed segmentation: local minima of gray-level (altitude) yield catchment basins; local maxima define the watershed lines. [Source: Sonka/Hlavac/Boyle 2008; Fig 6.48b]

Watershed segmentation often leads to over-segmentation (see lower right in figure 10), that is a segmentation into too many regions. Watershed segmentation is therefore often used as part of an interactive system, where the user first marks seed locations (with a click or a short stroke) that correspond to the centers of different desired components.

Matlab watershed

Advantages no specification of the number of clusters necessary or any other parameter.

Disadvantages oversegmentation, slow

6.1.3 Clustering with k-Means

One can exploit machine learning techniques for cluster analysis to segment an image. In cluster analysis one groups data points - so-called feature vectors - according to a distance measurement. In image segmentation these data points are the pixels, whereby we can use any information for the feature vector: the pixel's intensity value, its color values, its coordinates (x and y value), a local measurement or combinations thereof.

We have already met one type of cluster analysis, namely the k-Means algorithm when we searched for clusters (groups) in a list of patches (subsection 4.1.1). Thus instead of feeding a 128-dimensional vector, it will be here for instance a three-dimensional vector consisting of the dimensions (gray-level) intensity, x-coordinate and y-coordinate. Each pixel will then be assigned to one of the k clusters.

The advantage of using the k-Means algorithm is that it works relatively fast; its downside is that one needs to specify a number of clusters, meaning one needs to know what objects and background one expects. Of course, there are more sophisticated segmentation methods, such as the normalized cuts, which work well for foreground and background segmentation, but they also require much more computation.

Matlab kmeans

Advantages relatively fast (faster than region growing), slower than thresholding

Disadvantages specification of k (number of expected clusters)

6.1.4 Region Labeling and Statistics

If the segmentation algorithm has returned a binary map, then the next step would be to localize the regions of interest in it. If we are not interested in small regions, we may eliminate them immediately with the function bwareaopen. To find connected pixels that form regions of interest (objects), we can pursue two directions:

1. bwlabel: returns a map with connected pixels set to a certain integer value for the same object. The objects are numbered $1...n_{\text{objects}}$. We then write a loop to find the objects' indices using find.

2. bwconncomp: returns a structure with fields corresponding to the objects indices, that is indices have already been extracted, as opposed to bwlabel. If one desires a labeled matrix as it is created with bwlabel, then we apply the function labelmatrix.
The advantage of the use of `bwconncomp` is, that it requires less memory than `bwlable`.

### 6.2 Morphological Processing

Morphological processing is a set of local manipulations of images, that serve to modify the image toward a desired outcome in order to improve measurements of regions, contours, shapes, blobs etc. There exists binary and gray-scale morphology. Binary morphology manipulates only binary images where the results is again a binary image (subsection 6.2.1); grayscale morphology manipulates in particular grayscale images (subsection 6.2.2). The basic two operations of those local manipulations are the erosion and dilation operations, which - as the name implies - make the object(s) shrink or grow in some specified way. Those two basic operations are then combined to form more complex operations.

![Figure 12: Binary morphological processing. From left to right: original; erosion of original; dilation of original; closing of original (first dilation, then erosion); opening of original (first erosion, then dilation). (The effects of closing are not really viewable in this case)](image)

#### 6.2.1 Binary Morphology

Morphological operations with binary images are carried out with the function `bwmorph` in Matlab. The two basic operations cause the following on an original image as shown on the leftmost plot in figure 12:

- **Erosion**: the object will loose one 'layer' of pixels at its boundary (see second graph in figure 12).
- **Dilation**: the object will add one layer of pixels at its boundary (third graph in figure 12).

We can add more than one layer by specifying the parameter \( n \) in Matlab, e.g. `bwmorph(I, 'erosion', 2)` for two pixel layers.

If one applies the above two operations in sequence, then that tends to leave large regions and smooth boundaries unaffected, while removing small objects or holes and smoothing boundaries:

- **Closing**: is the dilation of an image followed by an erosion; it fuses narrow breaks and fills small holes and gaps (4th graph in figure 12).
- **Opening**: is an erosion followed by a dilation; it eliminates small objects and sharpens peaks in an object (5th graph in figure 12).

Other combinations of those basic operations are possible, the Matlab document pages explain those well. When developing an application, one simply has to try out a lot of combinations of such morphological operations and observe carefully the output.

#### 6.2.2 Grayscale Morphology

The above introduced operations also exist in grayscale morphology, but here the operation is not the change of a bit value, yet the selection of an extremum value in the neighborhood under investigation. Here we talk of the *structural function* which takes the maximum or minimum in the simplest case. To carry out grayscale morphology in matlab, there exist the functions `imdilate`, `imerode`, `imopen` and `imclose`.
6.3 Exercises

6.3.1 Segmentation

Study the example given in appendix E.3 (figure 10).

1. The use of the median threshold is motivated by the assumption that the majority of background values is clustered around a frequent value. This is indeed the case as we can observe in the histogram, but the problem is that the background intensity values are rather widely distributed and that inevitably will assign many background pixels to the object. A better threshold is the algorithm by Otsu, which takes the spread of fore- and background pixels into account.

2. If you run the script multiple times, then you observe that the k-Means algorithm changes its foreground/background assignment. Why?

3. On which aspect was the k-Means algorithm tested? You could try a 3-component vector per pixel (intensity and x/y coordinates). Create the x and y coordinates with `repmat`:

```matlab
[m n] = size(I); % size of image
Xco = repmat(1:n,m,1);
Yco = repmat((1:m)',1,n);
FET = [I(:) Xco(:) Yco(:)];
Ix = kmeans(FET,2);
LbMx = reshape(Ix,[m n]);
```

Display the label matrix `LbMx` with `imagesc`. Explain the odd results? Increase the number of clusters (k) one by one. What happens now? Obviously, coordinates are of little use in this example. When then could they be of benefit?

4. Take some other images and compare the method.

5. Include color information (RGB values).

6. Take several images of the same type, e.g. one type of flower images; or street scenes of certain perspectives. Make a comparison between the instances of the same class.

6.3.2 Morphological Operations

The output of the segmentation methods above is reasonable but not sufficient if we intend for instance to count the number coins: one coin was detected only partially and may be counted multiple times. We thus perform some morphological operations to achieve this goal.

1. Use the following lines to arrive and the correct coin count starting with the logical map as obtained by the Otsu-thresholding method. The function `bwlabel` labels the connected components (regions or objects) in the logical map. The opening and closing calls will solidify somewhat the coin with lower intensity.

```matlab
LbOts = bwlabel(BWots);
fprintf('# objects after segm: %d
', max(LbOts(:)));
BWmod = bwmorph(BWots,'open');
BWmed = bwmorph(BWmod,'close');
Lbmod = bwlabel(BWmod);
fprintf('# objects after morph: %d
', max(Lbmod(:)));
figure(3);clf;imagesc(BWmod);
```

2. Alternatively, use the following erosion operation on the median-threshold image.

```matlab
BWmod = bwmorph(BWmed,'erode',7);
```
This is certainly the more elegant solution, as we have only one type of morphological operation to arrive at the correct coin count - instead of two as above.

Both examples are merely improvised solutions. Whether they would be useful also for other images with coins would have to be demonstrated of course. But it is clear that there is not necessarily one solution. That the output of the median filter can be used to obtain the correct coin count is not immediately clear from the segmentation output. The application of morphological processing requires experience.

Try the entire processing - segmentation and morphology - on Matlab's 'rice' image (imread('rice.png')).
7 Feature Extraction II: Contours, Regions, Blobs & Texture

The structure of objects in images is outlined by contours; scenes often consist of a specific alignment of regions, blobs or dots. We now move toward the detection and description of such features. To some extend such features are already detected with the image segmentation methods introduced earlier (subsection 6.1), but in some application we would like to be more precise. For example to detect the precise outline of an object, we better detect edges in the intensity landscape directly (subsection 7.1). Or if we know the approximate size of a type of object we search for, for instance nuclei detection in medical images of cell tissues, then we might as well build a filter that directly favors such structures (subsection 7.2).

7.1 Contour Extraction

Contours are extracted in two phases, namely edge detection and edge following. In edge detection, the image is filtered with a local neighborhood filter that detects local 'edges' (subsection 7.1.1). The output of an edge detector is typically only a logical (binary) map with those pixels set to 1, where edges are present. We therefore need to trace neighboring edges to arrive at an actual contour, a process called edge following (subsection 7.1.2).

7.1.1 Edge Detection

An edge is an abrupt change in contrast with a certain orientation or direction; it is observed locally only and at each pixel. Edges can be detected using different techniques. The principal technique is to convolve the image with local 2D filters, which filter for a 'step' in intensity by using masks such as

\[
\begin{bmatrix}
-1 & 0 & 1 \\
-1 & 0 & 1 \\
-1 & 0 & 1
\end{bmatrix}
\quad \text{and} \quad
\begin{bmatrix}
0 & 1 & 1 \\
-1 & 0 & 1 \\
-1 & -1 & 0
\end{bmatrix}.
\]

Those two filters represent primitive examples for a vertical and diagonal orientation filter respectively. Thus, the image is convolved multiple times with different orientations to detect all oriented edges. Matlab provides the function \texttt{edge}, which returns . The code example in appendix E.4.1 displays the output of different edge detection techniques. The Roberts detector is a rather primitive detector and is rather outdated by now, but it is still used in some industrial applications: its advantage is its low processing duration, its downside is that it does not detect all edges. The Prewitt and Sobel detector detect more edges, but at the price of more computation. The most elaborate detection technique is the Canny algorithm, \texttt{Medg = edge(I, ’canny’, [], 1)}, whereby \[] stands for no particular choice of threshold parameters (and thus the threshold is determined automatically) and 1 is the scale value. Typically, scale values up to 5 are specified. Hence, with this technique one can easily perform low-pass-filtering and edge detection using one function only.

The binary edge maps may contain contours whose width may exceed one pixel. This can be problematic when doing edge following and some morphological processing may facilitate the process, in which case we apply the morphological thinning operation. Or we may want to remove isolated pixels immediately:

\begin{verbatim}
Medg = bwmorph(Medg,'clean');  \% removes isolated pixels
Medg = bwmorph(Medg,'thin');  \% turns 'thick' contours into 1-pixel-wide contours
\end{verbatim}

7.1.2 Edge Following

Edge following is also called contour or boundary tracing, whereby boundary stands strictly speaking for finding the boundaries of regions. There are several ways to do this: 1) we use the Matlab function \texttt{bwboundaries}; 2) we use the Matlab function \texttt{bwtraceboundary}; 3) write our own routine.

1. \texttt{bwboundaries}: this is rather useful for finding the boundaries of regions; for contours it will coil up the contour pixels in both directions - that is twice - because function treats contours as regions as well: it quasi treats a contour (of width equal one pixel) as a region with width equal 0 and return a circular
contour whose pixels of 'one side' sit on the pixels of the 'other side'. Thus, one needed to eliminate one half of the contour coordinates, if one preferred the pixel coordinates only once.

2. \texttt{bwtraceboundary}: here one specifies a starting point and the function will then trace until it coincides with its starting point. In contrast to the function \texttt{bwboundaries} above, it does not treat a contour as a region and pixels are 'recorded' only once. Hence, we write a loop which detects starting points and trace contours individually. We need to take care of when tracing should stop for an individual contour.

3. Own routine: it is possibly easiest to start tracing from the endpoints of contours. With the command \texttt{bwmorph(Medg,'endpoints')} we can easily identify them. From those endpoints one can trace toward the intersection point of contours, which in turn can be identified with the morphological operation 'branchpoints'.

### 7.2 Dots, Blobs, Regions - Bandpass Filtering

Dots are considered to be very small features, one or several pixels in size and round in shape. Blobs are considered to be larger and can be elongated (e.g. elliptically shaped) and may also represent regions. Regions are considered to be of any size and shape and their detection and extraction is typically treated under the topic of image segmentation (section 6). One type of rectangular filters was mentioned already, namely the rectangle detection for face detection (subsection 5.1.1).

Dots and blobs stand out either as a bright(er) feature in a dark(er) surround (or context), or vice versa as a dark(er) feature in a bright(er) surround. If one has generated a scale space, then one can easily search for such 'stand-out' features by subtracting the adjacent images of the scale space. For example we subtract an image low-pass filtered with sigma equal 2 from the one filtered with sigma equal 1: 

$$I_{b12} = I_{\sigma=1} - I_{\sigma=2}.$$ 

We then apply a suitable threshold \(t\) to obtain our desired features, for instance \(M_{\text{brighter}} = I_{b12} > t\), to obtain a map with brighter dots, blobs and regions.

This subtraction of two low-pass filtered images corresponds to bandpass filtering, meaning a center range (band) of values is favored, as opposed to the low-pass filtering process, which favors a lower range of values only. If the low-pass filtering occurred with Gaussian functions, then the shape of the bandpass filter is that of a Mexican hat approximately, as it corresponds to the difference of two Gaussian functions of different width (sigma).

![Figure 13: Detecting cell nuclei in cell tissue. Nuclei typically appear as dark blobs, so one may try to segment them by starting with a bandpass filter whose size is adjusted to the typical nucleus size. Or one could try to solve the task by using morphological processing (subsection 6.2). Or by a combination of both methodologies.](image)

If one intends to search for structures of specific size then one can design a specific filter. For example, if the objective is to find nuclei in microscopic images of cell tissue - they typically appear as a dark blob -, then one designs a bandpass filter whose size corresponds to the typical size of a nucleus. Thus, one would low-pass filter the image for two suitable sigmas, subtract the two images and then apply a threshold to the bandpass filtered image. And then continue the segmentation of nuclei.

If one searches for more complex structures, then one needs to apply more complex filters, such as the Gabor function, which is the product of a Gaussian function and a sinus wave. Examples of such complex filters are given in the subsequent section on texture.

### 7.3 Texture

Texture is observed in the structural patterns of surfaces of objects such as wood, grain, sand, grass, and cloth. But even scenes, that contain many objects, can be regarded as texture.
The term texture generally refers to repetition of basic texture elements called textons (or texels in older literature). Natural textures are generally random, whereas artificial textures are often deterministic or periodic. Texture may be coarse, fine, smooth, granulated, rippled, regular, irregular, or linear.

One can divide texture-analysis methods into three broad categories:

**Statistical**: these methods are based on describing the statistics of individual pixel intensity values. We only marginally mention these methods (subsection 7.3.1), as they have been outperformed by methods of spectral analysis (subsection 7.3.2).

**Structural**: in structural analysis, primitives are identified first, such as circles or squares, which then are grouped into more ‘global’ symmetries. We also do not treat these methods as no dominant method has evolved so far (see SHB pch 15 for discussion).

**Spectral**: in those methods, the image is firstly filtered with a variety of filters such as blob and orientation filters, followed by a statistical description of the filtered output. Caution: these methods are sometimes also called 'structural'.

### 7.3.1 Statistical

One can distinguish between first-order statistics and second-order statistics. First-order statistics take merely measures from the distribution of gray-scale values. In second-order statistics, one attempts to express also spatial relationships between pixel values and coordinates.

#### First-Order Statistics

Let $v$ be the random variable representing the gray levels in the region of interest. The first-order histogram $P(v)$ is defined as

$$P(v) = \frac{n_v}{n_{tot}}$$

with $n_v$ the number of pixels with gray-level $v$ and $n_{tot}$ the total number of pixels in the region ($\text{imhist}$ in Matlab for an entire image). Based on the histogram (equ. 9), quantities such as moments, entropy, etc. are defined. Matlab: $\text{imhist}$, $\text{rangefilt}$, $\text{stdfilt}$, $\text{entropyfilt}$

#### Second-Order Statistics

The features resulting from the first-order statistics provide information related to the gray-level distribution of the image, but they do not give any information about the relative positions of the various gray levels within the image. Second-order methods correlate these values. There exist different schemes to do that, the most used one is the gray-level cooccurrence matrix ($\text{graycomatrix}$ in Matlab), see $\text{Dav p213, s8.3 SHB p723, s15.1.2}$ for more.

The use of the cooccurrence matrix is memory intensive but useful for categories with low intensity variability and limited structural variability - or textural variability in this case. For ‘larger’ applications, the use of a spectral approach may return better performance.

### 7.3.2 Spectral [Structural]

In the spectral approach the texture is analyzed at different scales and described as if it represented a spectrum, hence the name. One possibility to perform such a systematic analysis is the use of wavelets, which has found great use in image compression with the jepg format. Wavelets are however based on a single mother wavelet only, that is on a single 'filter’ function. And instead of compression, we aim at classification. To describe texture it is therefore more meaningful to search for more complex filters. Such filters were already introduced in the sections image processing (2) and feature extraction (3). Here they are illustrated more explicitly.

Figure 14 shows an example set of such filters (see Appendix E.4.4 for code). The first four filters are merely Gaussian functions for different sigmas as we used them to generate the scale space (subsection 2.1). The following 8 filters are bandpass filters as we mentioned them in subsection 7.2. In this case the blob filter is much larger and is a Laplacian-of-Gaussian (LoG) filter. The filters in rows three to five are oriented filters that respond well to a step or edge in an image - it corresponds to edge detection as introduced for contour extraction in subsection 7.1. In this case, the first derivative of the Gaussian function
is used. The filters in the last three rows correspond to a bar filter. It is generated with the second derivative of the Gaussian.

Figure 14: A bank of filters for texture analysis (the Leung-Malik filter bank): blob and orientation filters for different scales (numbering rowwise).

**Filters 1-4:** Gaussian function (low-pass filter) at 4 different scales (sigmas).

**Filters 5-12:** a blob filter - a LoG in this case - at 8 different scales.

**Filter 13-30:** an edge filter - 1st derivative of the Gaussian in this case - for 6 different orientations and 3 different scales.

**Filter 31-48:** a bar filter - generated analogously to the edge filter.

To apply the filter bank, an image is filtered with each filter, thus resulting in a corresponding response map for each filter. The filtering output is thus a large collection of response maps and in order to find textons in this large output, one applies a quantization scheme as discussed in section 4. Algorithm 5 is a summary of the filtering procedure. In a first step, each filter is applied to obtain a response map $R_i$ ($i = 1..n$). $R_i$ can have positive and negative values and in order to find all extrema, we rectify the maps obtaining so $2n$ maps $R_j$ ($j = 1..2n$) - as a second step. In a third step, we try to find the 'hot spots' (maxima) in the image, that potentially correspond to a texton, by aggregating large responses. We can do this for example by convolving $R_j$ with a larger Gaussian; or by applying a max operation to local neighborhoods. Finally, we locate the maxima using an 'inhibit-of-return' mechanism that starts by selecting the global maximum followed by selecting (the remaining) local maxima, whereby we ensure that we do not return to the previous maxima, by inhibiting (suppressing) the neighborhood of the detected maximum. Thus, for each image we find a number of 'hot spots', each one described by a vector $x_i$. With those we then build a dictionary as described in section 4.

Note 1: In principle, one can add such texture descriptors to the interest-point descriptors and so enrich the description. Or one may work only with a specific set of textures.

Note 2: Textons are also be used for 'semantic segmentation', see the work by Shotton et al.

### 7.4 Special Features

#### 7.4.1 Straight Line Segments, Circles

If an application requires the extraction of straight lines only, then there are some specialized methods to find them. The most used one is the Hough transform, which places straight line equations on each point of the diagonal toward all directions. The transform returns a matrix whose axes correspond to two variables:
Algorithm 5 Local filtering for texture.

**Input**: image $I$, set of $n$ filters $F_i$ (see figure 14) ($i = 1..10$)

**Parameters**: radius for suppression in maximum search

1) Apply each filter $F_i$ to the image to obtain a response map $R_i = I * F_i$
2) Rectification: for each $R_i$ compute $\max(0, R_i)$ and $\max(0, -R_i)$, resulting in $2n$ maps $R_j$ ($j = 1..2n$)
3) For each $R_j$, compute local summaries $R_s^j$ by either
   a) convolving with a Gaussian of scale approximately twice the scale of the base filter
   b) taking the maximum value over a certain radius $r$. 
4) Locate the maxima (in each map?) using an inhibition-of-return mechanism: $l = 1..n_{max}$ with coordinates $[x_l, y_l]$

**Output**: set of 'hot spot' vectors $x_l(j)$ whose components $j$ correspond to the filter response $R_j$ at location $(x_l, y_l)$

one is the distance $\rho$ of the straight line from the image origin; and the other is the angle $\theta$. The matrix is a 2D histogram, called an accumulator here, with the maximum corresponding to the longest, straightest line in the image. For both variables, the bin size needs to be specified, which requires some tuning by the user. Example: the image contains a single line running from point $(x = 0, y = 1)$ to point $(x = 1, y = 0)$, then the transform has a single point at $(\rho = \sqrt{2}/2, \theta = 45^\circ)$.

Matlab: see the example under houghlines

Straight lines are used to find vanishing points and hence internal and external camera parameters.

The Hough transform can be modified to search also for circles, see `imfindcircles`.

7.5 Exercises

7.5.1 Edge Detection & Following

1. The code in appendix E.4.2 should help clarify the differences between the different edge/boundary detection functions. Observe carefully how many contours are extracted with each method. Why are three contours extracted with `bwboundaries`?
2. Contour Extraction: Write a script `ContExtr` (contour extraction), which performs edge detection with `edge` and contour tracing using `bwboundaries`. The output of `bwboundaries` is a list of contours, whose coordinates we simply draw with the command `plot`, see `doc bwboundaries` for further examples. Plot the contours in two different images: once on top of the edge map $EM$ and once on top of the gray-scale image $I$. Use color yellow, it contrasts well to the gray-scale image. Mark the starting point of a contour with a star marker ("*").
3. Perform edge detection in the scale space, meaning for each scale separately. Display all levels including contours into a single figure. Observe how the edges change across scale: how do they differ across scale?

7.5.2 Blobs (Band-Bass Filtering)

The example in appendix E.4.3 shows how we can filter for specific structures, in this case nuclei. Two types of band-pass filters are used: the difference-of-Gaussians (DoG) and the Laplacian-of-Gaussian (LoG). The filter sizes and threshold were chosen somewhat arbitrarily.

1. Is the filtering optimal? What is the size of a typical nucleus? What filter size would you suggest?
2. Why is the intensity range different between the DoG and the LoG output?
Shape

Shape means the geometry of an object or form. In many applications it is merely a closed curve representing a silhouette. The following applications are examples where shape matching is used:

- medical imaging: to detect shape changes related to illness (tumor detection) or to aid surgical planning
- archeology: to find similar objects or missing parts
- architecture: to identify objects that spatially fit into a specific
- computer-aided design, computer-aided manufacturing: to process and to compare designs of mechanical parts or design objects.
- entertainment industry (movies, games): to construct and process geometric models or animations

In many shape applications, the task to be solved is retrieval and not classification as we pursued it in our exercises before (section 4), see subsection 1.2 again to understand the difference. Thus, for shape recognition a major concern is often the matching duration of two shapes, because when we compare one shape with thousands of other shapes, we wish to solve this matching process in reasonable time.

The number of shape matching techniques is almost innumerous. Each technique has its advantages and disadvantages and works often for a specific task and merely under certain conditions. Textbooks are typically shy of elaborating on this topic, because there is no agreed useful method and it is somewhat unsatisfactory and endless to present all techniques. The most significant criterion in choosing a technique is then the speed-accuracy tradeoff: the more complex a description is, the better is its matching accuracy, but the slower is its matching duration. If our application deals with millions of shapes, then we prefer a compact description, that is one with few parameters and hence short matching duration; we introduce such descriptions first (subsection 8.1). Then we introduce point-wise matching, which is more accurate but also slower (subsection 8.2). For completion, we mention how one would move toward a part description (subsection 8.3). Finally, we mention successful shape classification systems (subsection 8.4).

Before diving into shape matter, we introduce the terminology that is used to characterize shape descriptions. Ideally, a shape description would be invariant to the following modifications:

<table>
<thead>
<tr>
<th>Modification</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) scaling</td>
<td>changes in size</td>
</tr>
<tr>
<td>2) rotation</td>
<td>changes in orientation</td>
</tr>
<tr>
<td>3) translation</td>
<td>changes in position</td>
</tr>
<tr>
<td>4) reflection</td>
<td>changes in laterality (mirroring)</td>
</tr>
<tr>
<td>5) articulation</td>
<td>changes in alignment of parts (structural variability; intra-class variability)</td>
</tr>
<tr>
<td>6) deformation</td>
<td>changes in variability such as blur, cracks, noise</td>
</tr>
<tr>
<td>7) occlusion</td>
<td>presence of clutter</td>
</tr>
</tbody>
</table>

Practically, it is hardly possible to account for all these invariances and one focuses on only a subset for a given task.

8.1 Compact Description

The following descriptions can not discriminate large sets of shapes, but their use may serve as a triage for a more complex description and matching. Typically, one determines a few shape parameters and forms a vector with them, a feature vector. In subsection 8.1.1, we mention ‘simple’ measures based on the boundary or interior of the shape. In subsection 8.1.2 we introduce the radial description, which is probably the most efficient description that uses feature vectors.

8.1.1 Simple Measures

It is not difficult to come up with a few simple boundary and region measurements. One can also apply the statistics of moments to the shape region (interior). Here are examples of measures and their suggested definitions:
<table>
<thead>
<tr>
<th>Area</th>
<th>size of (shape) region (bwarea)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circularity</td>
<td>e.g. area shape / area circle (circle with equal diameter)</td>
</tr>
<tr>
<td>Principle Axis Ratio</td>
<td>(also called eccentricity or aspect ratio) ratio between axis of elongation and its orthogonal axis (or length of major axis / length of minor axis)</td>
</tr>
<tr>
<td>Euler number</td>
<td>( = S - N ): # contiguous parts - # holes. Example shape '3': 1=1-0; shape 'B': -1=1-2; shape '9': 0=1-1. Matlab: (bweuler)</td>
</tr>
<tr>
<td>Bending Energy</td>
<td>degree of curvature ( \sum \kappa(s) )</td>
</tr>
</tbody>
</table>

Matlab  
`regionsprops`  
**Advantages** compact, useful if very large number of shapes are to be matched, can serve as a triage  
**Disadvantages** not very discriminative

### 8.1.2 Radial Description

If the shape is a continuous curve, that is a single, closed curve, then we can extract many more useful parameters by determining its radial signature. The radial signature is the sequence of distances \( R(s) \) from the shape's center point to each silhouette (curve) point \( s \). To obtain the center point, one simply averages the curve points.

There are two relatively simple manipulations we can do with the radial signature. One is a Fourier analysis, meaning we express the signature as a spectrum (we omit an introduction of this complex but powerful analysis - it is topic of a signal processing lecture). The other is, we observe the extrema of the signature and determine so the number of corners. The former is more discriminative than the latter, but combining both is even more powerful.

**Fourier Analysis** The Fourier analysis transforms a signal into a spectrum, which in digital implementation is a sequence of so-called Fourier descriptors (FD). We apply this discrete Fourier transform to the (unmodified) radial signature \( R(s) \), normalize it by its first value:

\[
\text{FDabs} = \text{abs(fft(Rad))}; \quad \% \text{fast Fourier} \\
\text{FDn} = \text{FDabs(2:end)}/\text{FDabs(1)}; \quad \% \text{normalization by 1st FD}
\]

Typically, the first 5 to 10 Fourier descriptors are sufficient for discriminating the shapes.

**Derivative Analysis** Finding extrema in the signature is easier if we low-pass filter the signature and that may now remind of the facial profile analysis we carried out in our introductory exercise - yes, it is the same principle. The number of extrema corresponds to the number of corners in a shape, whereby here corner means a curvature higher than its context. Two corners would correspond to ellipse or bicorn shapes, three corners to triangle or trident shapes, etc.

### 8.2 Point-Wise

There are two cases of shape 'formats' one can distinguish. One is, a shape is expressed as a single boundary, a sequence of points, in which case one matches list of points (subsection 8.2.1). Or a shape is considered as a set of points (subsection 8.2.2). In both cases, one prefers too know the approximate alignment between the two shapes before an accurate similarity is determined. This is also known as the correspondence problem, meaning which points in one shape correspond to which others in another shape, at least approximately.

#### 8.2.1 Boundaries

If two shapes are expressed as a single boundary, then the simplest way to compare two shapes is to take the pairwise distances between the two lists of points and to sum the corresponding minima to arrive at a measure of similarity. The pairwise point matching is computationally costly and the computational complexity is said to be square, expressed also as \( O(N^2) \), where \( N \) is the number of pixels and \( O \) is the symbol for complexity. And to solve the correspondence problem once could simply one could shift the two
shapes against each other to find a minimum for the correspondence. This increases the complexity to $O(N^3)$, that is it is cubic now and thus rather impractical.

Of course, the complexity were greatly reduced if one used so-called landmarks or key-points only, namely points on the shape which represent either very high curvature or they are straight. The problem is that such key-points are difficult to determine consistently. The search for corners in the radial description (subsection 8.1.2) may serve as key-points to find the approximate correspondence.

The most efficient boundary matching technique is based on observing the local orientations along the boundary and including them in the matching process. The detailed steps are as follows:

1. Sample an equal number of points $i = 1, \ldots, N$ from each shape, equally spaced along the boundary.

2. Determine the local orientation $o$ at each point, for instance the angle of the segment spanning several pixels on both sides of the center pixel. Thus the shape is described by a list of $N$ points with three values per point: x- and y-coordinate, as well as orientation $o$.

3. Determine the farthest point using the radial description. This will serve as a correspondence. When matching two shapes, one would take the point-wise distances (including orientation $o$) using the farthest point as reference. The point-wise distances are also taken in reverse order to account for asymmetric shapes. Thus, the matching complexity is $O(2N)$ only.

### 8.2.2 Sets of Points

Here again we can distinguish between two cases: 1) shapes consist of multiple boundaries - and not only of one as assumed above; 2) shapes have limited intra-class variability and consist of few points.

**Multiple Boundaries** The most successful approach is called Shape Context (Belongie, Malik & Puzicha, 2002) and is based on taking local radial histograms at selected points of the shape. The selection of such key-points may not be completely consistent, but that would be compensated by a flexible matching procedure. At each key-point, a circular neighborhood of points is selected and a one-dimensional histogram is generated counting the number of on-pixels as a function of radial distance.

**Limited Variability; Few Points** This case is rather useful for localization an less for retrieval (or classification). We assume that we know the shape’s key-points and that its articulation is limited (see again property list given in the introduction). The goal is then to find the shape in an image. We are then faced with two tasks: the correspondence problem and the transformation problem. This will be treated in more detail in course II.

### 8.3 Toward Parts: Distance Transform & Skeleton

Intuition says that a description by segments or ‘parts’ should be the obvious solution to shape recognition. Decades of praxis has taught computer vision scientists that such a description is difficult to achieve. What exactly is supposed to be a part and how it should be represented are unsolved problems. But if one intended to work toward that direction, then the distance transform could be a piece of the puzzle, because it appears a ‘natural’ step toward extracting parts. After we introduced the distance (subsection 8.3.1), we mention the difficulties with obtaining the appropriate skeletons.

#### 8.3.1 Distance Transform

The distance transform calculates at each off-pixel in a binary image $b(i,j)$ the distance to the nearest on-pixel. The result is thus a scalar field called distance map $D(i,j)$. The distance map looks like a landscape observed in 3D, which is illustrated in figure 15. The distance values inside a rectangular shape form a roof-like shape, a chamfer (shapes used in woodworking and industrial design); the interior of a circle looks like a cone. The distance transform is also sometimes known as the grassfire transform or symmetric-axis transform, since it can also be thought of a propagation process, namely a fire front that marches forward until it is canceled out by an oncoming fire front. Wherever such fronts meet, that is where they form symmetric points, which correspond to the ridges in the distance map: it is a roof-like skeleton for the rectangle
and a single symmetric point for a circle - the peak of the cone.

Matlab `bwdist`

Applications: fast chamfer matching (binary image alignment), feathering in image stitching and blending, nearest point alignment for range data merging, level sets (course II)

8.3.2 Symmetric Axes, Skeleton

The symmetric axes are the ridges in the distance map, the contours that look like veins (upper right in figure 15). Extracting those is relatively difficult - at least no one has succeeded so far. Instead, they can be approximated by various other algorithms. One is a thinning algorithm, similar to the erosion operation for morphological processing (subsection 6.2.1, which when carried out infinitely results in a skeleton resembling the sym-axes, see lower right in figure 15 whereas the starting point are the filled shapes (lower left in figure). Such skeletons are then fragmented and a shape is expressed as a structural description, a description by parts in a certain alignment. No powerful method exists so far.

8.4 Classification

When it comes to classification, the most successful systems are not computer vision algorithms, but deep neural networks (DNNs). Deep neural networks are merely elaborate ‘traditional’ artificial neural networks (ANNs). For instance in handwritten digit identification, the best-performing methodology is a network
of so-called restricted Boltzmann machines (RBMs). And traffic signs are best recognized with so-called convolutional neural networks (CNNs); as the name implies, it involves a systematic convolution-type feature extraction.

The power of those networks comes from their robustness to local changes: small changes in the boundary do have less consequences in networks. In contrast, for any of the shape description techniques introduced above, such small changes can result in relatively different features and thus tendentially more wrong classifications than with DNNs. Take for instance the two rectangular shapes in figure 15: the corresponding skeletons in the lower right graph, show sufficient differences that make a robust comparison difficult.

There are two disadvantages with DNNs for shape recognition. One is they require the shape to be fairly well centered in the image; thus, a search algorithm is necessary. The other disadvantage is that they require fairly long to learn the features.

8.5 Exercises

First we create our own set of stimuli, see appendix E.5.1. In our subsequent testing scripts, we load that image, threshold it to obtain a logical map, then extract the shapes and finally compare them.

Appendix E.5.2 shows an example of how to extract and describe shapes, conveniently done with the Matlab function \texttt{regionprops}. In that example, we create a three-dimensional vector for each shape using the shape measurements ‘area’, ‘eccentricity’ and ‘equivalent diameter’. Then we compare each shape with each other one using the Euclidean distance measure, resulting in the distance matrix $DM$. This matrix is then sorted and we observe the first few most similar shapes, row-wise.

1. The sorting does not look very impressive. Hint: there is no normalization. By what aspect is the sorting dominated?
2. Add normalization:
   
   $$MxV = \max(Vec,[],1);$$
   
   $$Vec = \text{bsxfun(@ldivide,Vec,MxV)};$$

3. To what modifications is the shape description robust? See the properties again given in the introduction.
4. The distance measure in the example is not normalized. Normalize and observe whether there is improvement.

Appendix E.5.3 shows to do the same with a Fourier-transformed radial signature.

1. The shape was already normalized. Exclude normalization and observe the sorting. What invariance has been lost?
9 Image Classification/Search/Retrieval

Now that we have learned about segmentation, features, shapes and classifiers, we can tackle more on the topic of image classification as well as on search and retrieval systems. This section is partly review and partly the introduction of information theoretic concepts to improve the ‘recognition’ performance. Under subsection 9.1, we explain what tasks are considered image classification. In subsection 9.2 we introduce the idea of search and retrieval systems, and the methodology to carry it out efficiently.

9.1 Image Classification

Image classification is the detection of a specific scene or the assignment of a single label to a scene. Three examples:

Detecting explicit images  Example: detecting nudity or sexual content. The principal method is to detect a large peak for skin color in the image color histogram.

Material classification  Example: labeling material as ‘wood’, ‘glass’, ‘rubber’, ‘leather’, etc. Generally, different materials produce different image textures, so natural features to use for this multiclass classification problem will be texture features. However, materials tend to have some surface relief (for example, the little pores on the surface of an orange; the bumps on stucco; the grooves in tree bark), and these generate quite complex shading behavior. Changes in the illumination direction can cause the relief’s shadows to change quite sharply, and so the overall texture changes.

Scene classification  Example: labeling pictures as ‘bedroom’, ‘kitchen’, ‘beach’, etc. This type of labeling is of course very difficult because the structural variability in scenes is larger than in a single object only. Because scene objects can be found in very different (but not random) positions, scene representations require a substantial degree of spatial looseness. As a result, exact feature locations tend to be irrelevant.

For some time, the most successful scene classification systems were based on the selection of a number of characteristic local patches (for an image class) and on the creation of a code book (sections 3 and 4). Meanwhile Deep Neural Networks have shown better performance.

9.2 Image Search and Retrieval

An image retrieval system is a computer system for browsing, searching and retrieving images from a large database of digital images. Browsing, searching and retrieving are search processes of increasing specificity:
- browsing: user looks through a set of images to see what is interesting.
- searching: user describes what he wants, called a query, and then receives a set of images.
- retrieval: user enters an image and obtains the most similar images in return, ranked by some similarity measure.

Most traditional and common methods of image retrieval utilize some method of adding metadata such as captioning, keywords, or descriptions to the images so that retrieval can be performed over the annotation words. Manual image annotation is time-consuming, laborious and expensive; to address this, there has been a large amount of research done on automatic image annotation. Additionally, the increase in social web applications and the semantic web have inspired the development of several web-based image annotation tools.

To search for images, a user may provide query terms such as keyword, image file/link, or click on some image or image features, and the system will return images “similar” to the query. The similarity used for search criteria could be meta tags, color distribution in images, region/shape attributes, descriptors as above (see scene classification in above subsection), etc.

Content-based image retrieval (CBIR) is the application of computer vision to the domain of image retrieval. CBIR aims at avoiding the use of textual descriptions and instead retrieves images based on similarities in their contents (textures, colors, shapes etc.).
### 9.2.1 Applications

#### Finding Near Duplicates

1) Trademark registration: A trademark needs to be unique, and a user who is trying to register a trademark can search for other similar trademarks that are already registered (see figure 21.1 below).

2) Copyright protection.

![Figure 21.1](image.png)

**FIGURE 21.1:** A trademark identifies a brand; customers should find it unique and special. This means that, when one registers a trademark, it is a good idea to know what other similar trademarks exist. The appropriate notion of similarity is a near duplicate. Here we show results from Belongie et al. (2002), who used a shape-based similarity system to identify trademarks in a collection of 300 that were similar to a query. The figure shown below each response is a distance (i.e., smaller is more similar). *This figure was originally published as Figure 12 of “Shape matching and object recognition using shape contexts,” by S. Belongie, J. Malik, and J. Puzicha, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2002, © IEEE, 2002.*

#### Semantic Searches

Other applications require more complex search criteria. For example, a stock photo library is a commercial library that survives by selling the rights to use particular images. An automatic method for conducting such searches will need quite a deep understanding of the query and of the images in the collection. Internet image search shows one can build useful image searches without using deep object recognition methods (it is a safe bet that commercial service providers do not understand object recognition much better than the published literature). These systems seem to be useful, though it is hard to know how much or to whom.

#### Trends and Browsing

In data mining, one uses simple statistical analyses on large datasets to spot trends. Such explorations can suggest genuinely useful or novel hypotheses that can be checked by domain experts. Good methods for exposing the contents of images to data mining methods would find many applications. For example, we might data mine satellite imagery of the earth to answer questions like: how far does urban sprawl extend?; what acreage is under crops?; how large will the maize crop be?; how much rain forest is left?; and so on. Similarly, we might data mine medical imagery to try and find visual cues to long-term treatment outcomes.

### 9.2.2 Document Retrieval

We now review techniques from document retrieval, that found their ways into image retrieval. Typical text information retrieval systems expect a set of query words. They use these to query some form of index, producing a list of putative matches. From this list they chose documents with a large enough similarity measure between document and query. These are ranked by a measure of significance, and returned.

Much of text information retrieval is shaped by the fact that a few words are common, but most words are rare. The most common words - typically including ‘the’, ‘and’, ‘but’, ‘it’ - are sometimes called *stop words* and are ignored because they occur frequently in most documents. Other words tend to be rare, which means that their frequencies can be quite distinctive. Example: documents containing the words ‘stereo’, ‘fundamental’, ‘trifocal’ and ‘match’ are likely to be about 3D reconstruction.

#### Indexing Documents

A $N_w \times N_d$ table $D_{ij}$ is generated in which an entry represents the frequency $f$ for a word $t$ in a document $j$ ($N_w$ total number of words, $N_d$ total number of documents). The table is sparse as...
most (non-stop) words occur in few documents only. We could regard the table as an array of lists. There is one list for each word, and the list entries are the documents that contain that word. This object is referred to as an inverted index, and can be used to find all documents that contain a logical combination of some set of words. For example, to find all documents that contain any one of a set of words, we would take each word in the query, look up all documents containing that word in the inverted index, and take the union of the resulting sets of documents. Similarly, we could find documents containing all of the words by taking an intersection, and so on. This represents a coarse search only, as the measure $f$ is used as a binary value only (and not as an actual frequency). A more refined measure would be the word count, or even better, a frequency-weighted word count. A popular method is the following:

**tf-idf** stands for ‘term frequency-inverse document frequency’ and consists of two mathematical terms, one for ‘term frequency’, the other for ‘inverse document frequency’. With $N_t$ as the number of documents that contain a particular term, the idf is

$$\text{idf} = \frac{N_d}{N_t}.$$  \hspace{1cm} (10)

Practical tip: add a value of one to the denominator to avoid division by zero. With $n_t(j)$ for the number of times the term appears in document $j$ and $n_w(j)$ for the total number of words that appear in that document, the tf-idf measure for term $t$ in document $j$ is

$$f_{t,j} = \frac{n_t(j)}{n_w(j)} \log \left( \frac{N_d}{N_t} \right).$$ \hspace{1cm} (11)

We divide by the log of the inverse document frequency because we do not want very uncommon words to have excessive weight. The measure aims at giving most weight to terms that appear often in a particular document, but seldom in all other documents.

**Comparing Documents** Assume we have a fixed set of terms that we will work with. We represent each document by a vector $f$, with one entry for each term. Put differently, we select a subset of dimension $t$ of the table. Two document vectors, e.g. $f_1$, $f_2$, can then be compared using the dot product and normalization by their respective lengths:

$$\text{sim} = \frac{f_1 \cdot f_2}{\|f_1\| \cdot \|f_2\|}.$$ \hspace{1cm} (12)

The cosine similarity weights uncommon words that are shared more highly than common words that are shared. Expressed differently, two documents that both use an uncommon word are most likely more similar than two documents that both use a common word.

**Ranking Documents** In response to a query, a set of $N_b$ documents is returned, where $N_b$ needs to be specified. To evaluate the query, two measures are determined:

- **Recall**: the percentage of relevant items that are actually recovered
  $$R = \frac{n_r}{N_r},$$ \hspace{1cm} (13)
  where $n_r$ is the number of recovered relevant items and $N_r$ is the total number of relevant items for this query.

- **Precision**: the percentage of recovered items that are actually relevant
  $$P = \frac{n_r}{N_b},$$ \hspace{1cm} (14)
As $N_b$ increases, $R$ will increase as well, but $P$ will decrease. Thus, when plotted as Recall against Precision, the curve typically decreases.

It is tempting to believe that good systems should have high recall and high precision, but is actually application dependent, as the following examples illustrate.

- Patent searches: Patents can be invalidated by finding prior art (material that predates the patent and contains similar ideas). A lot of money can depend on the result of a prior art search. This means that it is usually much cheaper to pay someone to wade through irrelevant material than it is to miss relevant material, so very high recall is essential, even at the cost of low precision.

- Web and email filtering: Some companies worry that internal email containing sexually explicit pictures might create legal or public relations problems. One could have a program that searched email traffic for problem pictures and warned a manager if it found anything. Low recall is fine in an application like this; even if the program has only 10% recall, it will still be difficult to get more than a small number of pictures past it. High precision is very important, because people tend to ignore systems that generate large numbers of false alarms.

**F measure**: To summarize the recall and precision values, different formulas can be used. A popular one is the $F_1$-measure, which is a weighted harmonic mean of precision and recall:

$$F_1 = 2 \frac{PR}{P + R}. \tag{15}$$

**Average Precision** An important way to summarize a precision-recall curve is the average precision, which is computed for a ranking of the entire collection. This statistic averages the precision at which each new relevant document appears as we move down the list. $P(r)$ is the precision of the first $r$ documents in the ranked list, whereby $r$ corresponds to $N_b$ in equation 14; $N_r$ the total number of documents in the collection. Then, the average precision is given by

$$A = \frac{1}{N_r} \sum_{r=1}^{N_r} P(r). \tag{16}$$

which corresponds to the area under the curve.

**Example** In response to a query the (total of) 3 relevant items are found at positions 2, 13 and 36:

$$A = \frac{1}{3} \left( \frac{1}{2} + \frac{2}{13} + \frac{3}{36} \right) = 0.2457$$

**Implementation**

Given an index vector with sorted retrieval positions, $I_x$, the measure is:

```matlab
Ix = sort(Ix); % sort in increasing order
nItm = length(Ix); % number of relevant items (N_r)
A = sum((1:nItm)./Ix)./nItm; % average precision
```

**Closing Note** In order to apply all these techniques to CBIR, we simply use ‘visual words’ (obtained as explained previously) instead of text words, and then use exactly the above equations. Or one can use both textual and visual information to create measures that weigh both types of information.

When we will discuss texture recognition (section 7.3), they will be called textons, but in the domains of object recognition and image classification they tend to be called visual words. Because these word histogram distributions are generally different from the histogram distributions for gradients (as in SIFT), alternate distance measures such as the $\chi^2$-squared kernel may perform better.
Tracking is the pursuit of one or multiple moving objects and possibly the interpretation of their dynamics. It has many applications:

**Motion Capture**: is the recording of the 3D configuration of a moving person using markers, e.g., white patches placed on the joints of a person dressed in a black suit. Such recordings are used for animation, e.g., rendering a cartoon character, thousands of virtual extras in a crowd scene, or a virtual stunt avatar.

**Recognition from Motion**: is the object identification by recognizing its (motion) dynamics. In some sense it is motion capture with discrimination.

**Surveillance**: is the monitoring of activities and the warning when a problem case is detected. Example: airport traffic surveillance: different kinds of trucks should move in different, fixed patterns - if they do not, it is suspicious; similarly, there are combinations of places and patterns of motions that should never occur (e.g., no truck should ever stop on an active runway).

**Targeting**: A significant fraction of the tracking literature is oriented toward (a) decision what to shoot, and (b) hitting it. Typically, this literature describes tracking using radar or infrared signals (rather than vision), but the basic issues are the same: What do we infer about an object’s future position from a sequence of measurements? Where should we aim?

There exist two simple but effective methods for tracking:

1. **Tracking by detection**: we have a strong model of the object, strong enough to identify it in each frame. We localize it, link up the instances, and that would be our track.

2. **Tracking by matching**: we have a model of how the object moves. We have a domain in the \( n \)th frame in which the object sits, and then use this model to search for a domain in the \((n + 1)\)th frame that matches it.

### 10.1 Tracking by Detection

- **Case '1 object in each frame'**: If there is only one object in each frame, we try to build a reliable detector (for that object) and observe its position in each frame (figure 16a). If the detector is not reliable we treat the problem as if there were multiple objects, see next.

  **Example**: Tracking a red ball on a green background: the detector merely needs to look for red pixels. In other cases, we might need to use a more sophisticated detector, e.g., tracking a frontal face.

- **Case 'Multiple objects (or unreliable detector)’**: If objects enter or leave the frame (or the detector occasionally fails), then it is not enough to just report the object’s position. We then must account for the fact that some frames have too many (or too few) objects in them. Hence, we maintain a track, which represents a timeline for a single object (figure 16b).

  Typically, the tracks from the previous frame are copied to the next frame, and then object detector responses are allocated to the tracks. How we allocate depends on the application (we give some examples below). Each track will get at most one detector response, and each detector response will get at most one track. However, some tracks may not receive a detector response, and some detector responses may not be allocated a track. Finally, we deal with tracks that have no response and with responses that have no track. For every detector response that is not allocated to a track, we create a new track (because a new object might have appeared). For every track that has not received a response for several frames, we prune that track (because the object might have disappeared). Finally, we may postprocess the set of tracks to insert links where justified by the application. Algorithm 6 breaks out this approach.

  The main issue in allocation is the cost model, which will vary from application to application. We need a charge for allocating detects to tracks. For slow-moving objects, this charge could be the image distance between the detect in the current frame and the detect allocated to the track in the previous frame. For objects with slowly changing appearance, the cost could be an appearance distance (e.g., a \( \chi^2 \)-squared distance between color histograms). How we use the distance again depends on the application. In cases where the detector is very reliable and the objects are few, well-spaced, and slow-moving, then a greedy
Algorithm 6 Tracking multiple objects (or tracking with unreliable object detector). \(i=\text{time}; \ t=\text{track}\).

**Notation:**
- Write \(x_k(i)\) for the \(k\)'th response of the detector in the \(i\)th frame.
- Write \(t(k, i)\) for the \(k\)'th track in the \(i\)th frame.
- Write \(*t(k, i)\) for the detector response attached to the \(k\)'th track in the \(i\)th frame.
  (Think C pointer notation)

**Assumptions:** Detector is reasonably reliable; we know some distance \(d\) such that \(d(*t(k, i-1), *t(k, i))\) is always small.

**First frame:** Create a track for each detector response.

**N'th frame:**
- **Link** tracks and detector responses by solving a bipartite matching problem.
- **Spawn** a new track for each detector response not allocated to a track.
- **Reap** any track that has not received a detector response for some number of frames.

**Cleanup:** We now have trajectories in space time. Link anywhere this is justified (perhaps by a more sophisticated dynamical or appearance model, derived from the candidates for linking).

Algorithm (allocate the closest detect to each track) is sufficient. This algorithm might attach one detector response to two tracks; whether this is a problem or not depends on the application.

The more general algorithm solves a bipartite matching problem, meaning tracks on one side of the graph are assigned to the detector responses on the other side of the graph. The edges are weighted by matching costs, and we must solve a maximum weighted bipartite matching problem (figure 16c), which could be solved exactly with the Hungarian algorithm, but the approximation of a greedy algorithm is often sufficient. In some cases, we know where objects can appear and disappear, so that tracks can be created only for detects that occur in some region, and tracks can be reaped only if the last detect occurs in a disappear region.

**Background subtraction** is often a simple-but-sufficient detector in applications where the background is known and all trackable objects look different from the background. In such cases, the background-subtracted objects appear as blobs and those are taken as detector responses. It is the simplest form of foreground/background segmentation.

**Example:** People tracking on a fixed background, such as a corridor or a parking lot. If the application does not require a detailed report of the body configuration, and if we expect people to be reasonably...
large in view, we can reason that large blobs produced by background subtraction are individual people. Weaknesses: if people stand still for a long time, they might disappear; it would require more work to split up the large blob of foreground pixels that occurs when two people are close together; and so on - many applications require only approximate reports of the traffic density, or alarms when a person appears in a particular view.

In many tracking tasks, nothing more complex is required. The trick of creating tracks promiscuously and then pruning any track that has not received a measurement for some time is quite general and extremely effective.

10.2 Tracking Translations by Matching

Example: Tracking soccer players on a television screen, with players of height 10-30 pixels. Detailed body-part dynamics can not be tracked due to low resolution and high frame rate (30Hz). Instead, we assume that the domain translates and we thus track the player as a box. We can model a player’s motion with two components. The first is the absolute motion of a box fixed around the player and the second is the player’s movement relative to that box. To do so, we need to track the box, a process known as image stabilization. As another example of how useful image stabilization is, one might stabilize a box around an aerial view of a moving vehicle; now the box contains all visual information about the vehicle’s identity.

In each example, the box translates. If we have a rectangle in frame \( n \), we can search for the rectangle of the same size in frame \( n + 1 \) that is most like the original, e.g. using the sum-of-squared differences (or SSD) of pixel values as a test for similarity and search for its minimum over a small neighborhood.

In many applications the distance the rectangle can move in an inter-frame interval is bounded because there are velocity constraints. If this distance is small enough, we could simply evaluate the sum of squared differences to every rectangle of the appropriate shape within that bound, or we might consider a search across the scale space (or even better the pyramid) for the matching rectangle.

Matching Principle  The simplest way to establish an alignment between two images or image patches is to shift one image relative to the other. Given a template image \( I_0(x) \) sampled at a set of discrete pixel locations \( \{x_i = (x_i; y_i)\} \), we wish to find where it is located in image \( I_1(x) \). A least squares solution to this problem is to find the minimum of the sum of squared differences (SSD) function

\[
E_{SSD}(u) = \sum_i |I_1(x_i + u) - I_0(x_i)|^2 = \sum_i e_i^2, \tag{17}
\]

where \( u = (u; v) \) is the displacement and \( e_i = I_1(x_i + u) - I_0(x_i) \) is called the residual error (or the displaced frame difference in the video coding literature). (We ignore for the moment the possibility that parts of \( I_0 \) may lie outside the boundaries of \( I_1 \) or be otherwise not visible.) The assumption that corresponding pixel values remain the same in the two images is often called the brightness constancy constraint.

Tracking Initiation  We can start tracks using an interest point operator. In frame 1, we find all interest points. We then find the location of each of these in the next frame, and check whether the patch matches the original one: if so, it belongs to a track; if not, the track has already ended. We now look for interest points that do not belong to tracks and create new tracks there. Again, we advance tracks to the next frame, check each against their original patch, reap tracks whose patch does not match well enough, and create tracks at new interest points.

10.3 Exercise

Understand the problems of motion detection by trying to detect, localize and track a motion in a short video. To start with, use the example in appendix E.6. Try other videos as well.
- Can you automatically find a threshold for eliminating ‘noise’?
- Try with background subtraction, that is take the first frame as background and subtract it from the following frames.
- Attempt to track an object, that is, select a suitable object patch that can be followed robustly. Can you improve by using color, namely exploiting the 3 components red/green/blue?
11 Alignment/Registration

In certain tasks we are given an object and we try to detect it somewhere else and we would like to know how exactly it moved, that is how exactly it changed its orientation, shape and perspective - or mathematically speaking how it transformed. In the simplest case, the object has moved straight from one location to another and otherwise it did not change - that would be a mere translation, see figure 17. But the object may have also been rotated slightly during the motion, in which case this would correspond to a so-called Euclidean transformation. The object may have also shrunk or enlarged during motion, in which case the transformation is expressed with similarity. The task to reconstruct the transformation that the object has undergone during a motion is called alignment or registration. The object does not need to have actually moved but it is convenient to imagine the reconstruction as a ‘motion estimation’. Put differently, alignment tries to find a transformation that takes one configuration of points to another configuration.

![Figure 17: Basic set of 2D planar transformations.](Source: Szeliski 2011; Fig 3.45)

In the most straightforward form, the two datasets have the same dimensionality, for instance we are registering 2D data to 2D data or 3D data to 3D data, and the transformation is rotation, translation, and perhaps scale. Here are two examples:

Medical Support: We have an MRI image (which is a 3D dataset) of a patient’s interior that we wish to superimpose on a view of the real patient to help guide a surgeon. In this case, we need to know the rotation, translation, and scale that will put one image on top of the other.

Cartography: We have a 2D image template of a building that we want to find in an overhead aerial image. Again, we need to know the rotation, translation, and scale that will put one on top of the other; we might also use a match quality score to tell whether we have found the right building.

We treat only 2D feature-based alignment for the beginning. First we explain how some of the 2D motions of figure 17 are expressed mathematically (subsection 11.1). Then we learn how to estimate them (subsection 11.2). Finally, we learn how to robustly estimate them, that is even in the presence of noise and clutter (subsection 11.4). The culminating exercise will be to estimate the motion of the two slightly rotated photographs taken in a previous exercise.

11.1 2D Transforms

Figure 17 showed the global parametric transformations for rigid 2D shapes. Here we introduce the formulas that express those transformations with matrix multiplications whereby two frequent notations are given in the table below: notation 1 is more explicit with respect to the individual motions; notation 2 concatenates the individual motions into a single matrix such that the entire motion can be expressed as a single matrix multiplication, which can be convenient sometimes.

**Translation**: simplest type of transform - merely a vector \( t \) is added to the points in \( x \). To express this as a single matrix the identity matrix is used (unit matrix; square matrix with ones on the main diagonal and zeros elsewhere) and the translation vector is appended resulting in a \( 2 \times 3 \) matrix, see also figure 18.
2D Euclidean Transform: consist of a rotation and translation. The rotation is achieved with the a so-called orthonormal rotation matrix $R$ whose values are calculated by specifying the rotation angle $\theta$. In notation 2, the single matrix is concatenated as before and it remains a $2 \times 3$ matrix.

Scaled Rotation: merely a scale factor is included to the previous transform. The size of the single matrix in notation 2 does not change.

Affine: Here a certain degree of distortion is allowed, but we do not elaborate on this transformation any further here. The single matrix for notation 2 still remains of size $2 \times 3$.

<table>
<thead>
<tr>
<th>Transform</th>
<th>Notation 1</th>
<th>Notation 2</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation</td>
<td>$x' = x + t$</td>
<td>$x' = [ I \ t]x$</td>
<td>$I$ is the $2 \times 2$ identity matrix</td>
</tr>
<tr>
<td>Rotation + Translation = 2D rigid body motion = 2D Euclidean transformation</td>
<td>$x' = Rx + t$</td>
<td>$x' = [R \ t]x$</td>
<td>$R = \begin{bmatrix} \cos(\theta) &amp; -\sin(\theta) \ \sin(\theta) &amp; \cos(\theta) \end{bmatrix}$ is orthonormal rotation matrix: $RR^T = I$ and $</td>
</tr>
<tr>
<td>Scaled Rotation = Similarity Transform</td>
<td>$x' = sRx + t$</td>
<td>$x' = [sR \ t]x$</td>
<td>$x' = \begin{bmatrix} a \ -b \ b \ a \ t_x \ t_y \end{bmatrix}x$</td>
</tr>
<tr>
<td>Affine</td>
<td>$x' = Ax$</td>
<td>$x' = \begin{bmatrix} a_{00} &amp; a_{01} &amp; a_{02} \ a_{10} &amp; a_{11} &amp; a_{12} \end{bmatrix}x$</td>
<td>Parallel lines remain parallel</td>
</tr>
</tbody>
</table>

11.2 Motion Estimation with Linear-Least Squares

To estimate the motion between two objects we require two types of information. One is the correspondence between the two sets of points, which is not so trivial to establish in real-word scenes (see feature detection before), but for the moment we assume that the correspondence is known. The other type of information is the kind of transformation we expect. To address this we could simply work with complex transformations in order to be prepared for any type of transformation (i.e. affine transformation), which however do not return the individual motion parameters explicitly; in scaled rotation in contrast we have explicit parameters for rotation, scale and translation.

Formulated mathematically, given a set of matched feature points $\{x_i, x'_i\}$ and a chosen planar transformation $f$ with parameters $p$ (e.g. $t, R...$),

$$x' = f(x_i; p),$$

how can we reconstruct the best estimate of the motion parameters values? The usual way to do this is to use least squares, i.e., to minimize the sum of squared residuals

$$E_{LS} = \sum_i \|r_i\|^2 = \sum_i \|f(x_i; p) - x'_i\|^2,$$

where

$$r_i = x'_i - f(x_i; p) = \hat{x}_i - x'_i$$

is the residual between the measured location $x'_i$ and its corresponding current predicted location $\hat{x}_i = f(x_i; p)$. See also appendix for more explanation on linear least squares.

For simplicity we assume now a linear relationship between the amount of motion $\Delta x = x' - x$ and the unknown parameters $p$:

$$\Delta x = x' - x = J(x)p,$$

where $J = \partial f/\partial p$ is the Jacobian of the transformation $f$ with respect to the motion parameters $p$. $J$ is shown in figure 18 and has a particular form for each transformation. In this case, a simple linear regression (linear-least-squares problem) can be formulated as

$$E_{LLS} = \sum_i \|J(x_i)p - \Delta x_i\|^2,$$
which - after some reformulation - equates to

\[ p^T Ap - 2p^T b + c. \]  \hspace{0.5cm} (23)

The minimum can be found by solving the symmetric positive definite (SPD) system of normal equations:

\[ Ap = b, \]  \hspace{0.5cm} (24)

where

\[ A = \sum_i J^T(x_i)J(x_i) \]  \hspace{0.5cm} (25)

is called the Hessian and

\[ b = \sum_i J^T(x_i)\Delta x_i. \]  \hspace{0.5cm} (26)

<table>
<thead>
<tr>
<th>Transform</th>
<th>Matrix</th>
<th>Parameters ( p )</th>
<th>Jacobian ( J )</th>
</tr>
</thead>
</table>
| translation | \[
\begin{bmatrix}
1 & 0 & tx \\
0 & 1 & ty
\end{bmatrix}
\] | \((tx, ty)\) | \[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\] |
| Euclidean | \[
\begin{bmatrix}
c_\theta & -s_\theta & tx \\
s_\theta & c_\theta & ty
\end{bmatrix}
\] | \((tx, ty, \theta)\) | \[
\begin{bmatrix}
1 & 0 & -s_\theta x - c_\theta y \\
0 & 1 & c_\theta x - s_\theta y
\end{bmatrix}
\] |
| similarity | \[
\begin{bmatrix}
1 + a & -b & tx \\
b & 1 + a & ty
\end{bmatrix}
\] | \((tx, ty, a, b)\) | \[
\begin{bmatrix}
1 & 0 & x & -y \\
0 & 1 & y & x
\end{bmatrix}
\] |
| affine | \[
\begin{bmatrix}
1 + a_{00} & a_{01} & tx \\
a_{10} & 1 + a_{11} & ty
\end{bmatrix}
\] | \((tx, ty, a_{00}, a_{01}, a_{10}, a_{11})\) | \[
\begin{bmatrix}
1 & 0 & x & y & 0 & 0 \\
0 & 1 & 0 & 0 & x & y
\end{bmatrix}
\] |

Figure 18: Jacobians of the 2D coordinate transformations \( x' = f(x; p) \) (see table before), where we have reparameterized the motions so that they are identity for \( p = 0 \). [Source: Szeliski 2011; Tab 2.1]

### 11.3 Exercises

#### 11.3.1 Preparing Transformations

Place the code in appendix E.7 into a script called \( t_{2D} \text{Transforms} \). In this script, a simple shape was specified with coordinates given as \( Co \). It was then transformed in various ways, followed by estimating the corresponding motions - we expect a small error of course. The estimation procedure is done with \( \text{lsqlin} \) or \( \text{lsqnonneg} \), whereby \( A \) and \( b \) are passed as arguments.

Now we are going to write separate functions for the individual transformations and motion estimations and we verify them at the end of the testing script \( t_{2D} \text{Transforms} \) by simply plotting their output into another figure for visual comparison.

1. Write a function \( f_{\text{TransRot}} \), which carries out a transformation for a given list of coordinates, meaning the function input is \( Co \) and the corresponding parameters for rotation; the function output is the set of transformed coordinates. Then write a function \( f_{\text{TransEuc}} \), which carries out a Euclidean transformation, etc. Verify by plotting the functions’ output into a separate figure.

(Or more elegantly: write a single function \( f_{\text{TransLin}} \), which performs a (linear) transformation according to the user’s specification: the input is a list of coordinates and a string specifying the type of transformation. In the function we would use \( \text{if}-\text{then} \) or \( \text{switch} \) statements to carry out the desired transformation. Use \( \text{varargin} \) for variable input arguments.)
2. Write functions for motion estimation: one called \texttt{f.EstMotSim} for similarity, whose input are two list of coordinates and whose output are the estimated parameters with estimation errors. Always verify the functions’ output in your testing script. Then write another function for motion estimation with affinity \texttt{f.EstMotAff}, etc.

3. Take a more complex shape now, e.g. with 10 coordinates.

11.3.2 Motion Estimation

We now do motion estimation with ‘noisy’ input, once with artificial noise and once using a real application:

1. Add some random offset to the transformed coordinates, e.g. \( \text{CoTN} = \text{CoT} + \text{rand}(\text{nP}, 2) \times 0.1 \), where \( \text{CoT} \) are the transformed coordinates, \( \text{nP} \) are the number of points (coordinates) and 0.1 scales the amount of noise. Use \( \text{Co} \) and \( \text{CoTN} \) as input - and not \( \text{CoT} \). Now we will have estimation errors. Increase the scale factor.

2. Try to estimate the motion between the two photographs you made in the previous exercise. Do this in another script called \texttt{t.Panoram}. To match the two photographs, what transformations could suffice? It actually depends how you moved your hand exactly between the two shots. Translation may suffice, but including rotation - which would account for slight rotations - may be beneficial.

11.4 Robust Alignment with RANSAC

You may have experienced some difficulties in the previous motion-estimation exercise. For instance, some points may not have had a corresponding point in the other image; other points were matched to the wrong points in the other image. One can summarize such ‘misses’ as outliers and noise, and they make motion estimation in real world applications difficult; in other words we have a correspondence problem. It therefore requires more robust estimation techniques, for instance an iterative process consisting of 2 steps:

1) selection of a random subset of points and motion estimation with them.
2) verification of the estimate with the remaining set of points.

Example: We are fitting a line to an elongated point cloud which contains about 50% outliers. If we draw pairs of points uniformly and at random, then about a quarter of these pairs will consist of ‘inlier’ data points. We can identify these inlier pairs by noticing that a large proportion of other points lie close to the fitted line. Of course, a better estimate of the line could then be obtained by fitting a line to the points that lie close to our current line.

Fischler and Bolles (1981) formalized this approach into an algorithm called RANSAC, for RANdom SAmple Consensus (algorithm 7). It starts by finding an initial set of inlier correspondences, i.e., points that are consistent with a dominant motion estimate: it selects (at random) a subset of \( n \) correspondences, which is then used to compute an initial estimate for \( p \). The residuals of the full set of correspondences are then computed as

\[
\text{r}_i = \hat{x}'_i(x_i; p) - \hat{x}'_i,
\]

where \( \hat{x}'_i \) are the estimated (mapped) locations and \( \hat{x}'_i \) are the sensed (detected) feature point locations.

The RANSAC technique then counts the number of inliers that are within \( \epsilon \) of their predicted location, i.e., whose \( \| \text{r}_i \| \leq \epsilon \). The \( \epsilon \) value is application dependent but is often around 1-3 pixels. The random selection process is repeated \( k \) times and the sample set with the largest number of inliers is kept as the final solution of this fitting stage. Either the initial parameter guess \( p \) or the full set of computed inliers is then passed on to the next data fitting stage.

11.5 Closing Notes

For range images, the most widely used 3D registration technique is the iterated closest point (ICP) algorithm (Sze p332, s 12.2.1, pdf 302). The most complex registration problem treats objects that can deform. In this case, the family of transformations that could register the two datasets is large, and the search for a particular transformation is
Algorithm 7 RANSAC: Fitting structures using Random Sample Consensus.

Input: \( D, D^* \)

Parameters:
- \( n \) the smallest number of points required (e.g., for lines, \( n = 2 \), for circles, \( n = 3 \))
- \( k \) the number of iterations required
- \( t \) the threshold used to identify a point that fits well
- \( d \) the number of nearby points required to assert a model fits well

Until \( k \) iterations have occurred:
- Draw a sample of \( n \) points from the data \( D \) uniformly and at random \( \rightarrow D^*; D^c = D \setminus D^* \)
- Fit to \( D^* \) and obtain estimates \( p \)
- For each data point \( x \in D^c \): if point close, that is smaller than a threshold: \( \|x_i\|^2 < t \), then \( x_i \rightarrow D^{good} \)
- If there are \( d \) or more points close to the structure (\( |D^{good}| \geq d \)), then \( p \) is kept as a good fit.
  - refit the structure using all these points.
  - add the result to a collection of good fits \( \rightarrow P^{good} \)

Choose the best fit from \( P^{good} \), using the fitting error as a criterion

correspondingly more difficult. Registering deformable objects is a core technology for medical image analysis, because human organs deform and because it is quite usual to image the same body component using different imaging modes.

11.6 Exercises

1) Appendix E.8 contains a function and a testing script. Copy the function code into a script called \( f_{RanSaC} \) and the testing code into a script called \( t_{Ransac} \). Study the example; compare with previous exercise script \( t_{2DTransforms} \). Play with the parameters to understand what is happening. Which parameter names in the script correspond to which parameter names in algorithm 7. With what type of transformation does the given function \( f_{RanSac} \) work? Note: the function script assumes ‘corresponding’ points already!

To verify the function script, you can also test it by appending the following lines to \( t_{2DTransforms} \), where Co and Caff are from that testing script:

```matlab
Opt.nMinPts = 7;
Opt.nMaxIter = 3;
Opt.thrNear = 0.2;
Opt.nNear = 5;
Opt.xlb = ''; 
Opt.Match = ''; 
b_plot.dist = 1;
RSprm = f_RanSaC(Co, Caff(:,[1 2]), Opt, b_plot);
RSprm'
```

We now optimize the function script for readability. First make a copy of the function and call it \( f_{RanSaCRaw} \), which we use for verification. Now optimize the function script by using the transformation functions which you have written in the previous exercise, that is replace the inline function with those function scripts.

2) Now apply RanSac to motion estimation of your two images from the previous exercise. Now you should be able to determine the degree of motion more reliable than before, even under noisy conditions.

3) Try to create a panorama of your room, or any other viewpoint. Translation may suffice for minimization, that is a 'simpler' \( J \) can be used.
12 Systems

12.1 Video Surveillance

Surveillance is useful for monitoring road traffic, monitoring pedestrians, assisting riot control, monitoring of crowds on football pitches, checking for overcrowding on underground stations, and generally is exploited in helping with safety as well as crime; and of course surveillance is used in military applications. Traditional video surveillance heavily relies on simple matching operations and on background subtraction, but recent systems, that use the principles of feature detection and matching (as introduced in section 3), have often outperform the traditional systems easily. It may thus be somewhat futile to dwell too long in old methods and rather apply the new methods, although a combination of old and new methods may always be worth trying. We already mentioned some aspects of surveillance (e.g. in sections 10 and 6), but we round the picture by adding some aspects and by giving some examples.

The Geometry The ideal camera position is above the pedestrian, at some height $H_c$, and the camera’s optical axis has a declination (angle $\delta$) from the horizontal axis (see figure 19). This is simply the most suitable way to estimate the distance and height $H_t$ of the pedestrian, thereby exploiting triangulation and the knowledge of where the position of the pedestrian’s feet.

Figure 19: 3-D monitoring: camera tilted downwards. $\delta$ is the angle of declination of the camera optical axis.

[Source: Davies 2012; Fig 22.2]

Foreground/Background Separation

The idea of background subtraction is to eliminate the 'static' background to obtain only the (moving) foreground objects. Although that appears to be a straightforward idea in principal, it is a challenging task because the background also changes continuously, e.g. the illumination changes throughout the day, vegetation flutters, shadows wander (of fixed objects and of clouds), etc. Furthermore, some of these background changes aggravate the challenge of detecting initial object motion. As mentioned before, using interest point features, much more stable motion detection and tracking can be provided (subsection 10.2).

Vehicle License Plate Detection License plate recognition is a challenging task due to the diversity of plate formats and the nonuniform outdoor illumination conditions during image acquisition. Therefore, most approaches work only under restricted conditions such as fixed illumination, limited vehicle speed, designated routes, and stationary backgrounds. Algorithms (in images or videos) are generally composed of the following three processing phases:

1) Extraction of a license plate region. An effective method is described in figure 20.
2) Segmentation of the plate characters. Thresholding as described previously (subsection 6.1.1)
3) Recognition of each character. A typical optical-character recognition system will do, see Pattern Recognition.

Scholar google "Anagnostopoulos" for a review.
Vehicle Recognition  Depending on the purpose, vehicles are categorized into types (e.g., car, truck, van,...) or more specifically, their make and model is identified (e.g. Volkswagen, Golf). A type classification has been approached with an analysis of depth images. For make/model identification, many systems have focused on analyzing the car’s front, which can be regarded as its face. A typical system locates the license plates first and then measures other front features in relation to it, such as head lights, radiator grill, etc.:

So far, (exploratory) studies have used up to 100 auto model tested on still images. Scholar google ‘Pearce and Pears’ for a recent publication.

12.2 In-Vehicle Vision Systems

A few years ago, a fully automated vehicle had been considered impossible by many experts, but Google has meanwhile automated (driverless) cars on (American) roads. We review some of the methods that are (most likely) used by such vehicles (subsection 12.2.1).

12.2.1 Automated Vehicle (AV)

An AV contains 4 interacting processes: environment perception, localization, planning and control. Localization typically relies on GPS. An AV has ca. 2/3 of its computers assigned for perception and the rest assigned to the other processes - interestingly enough the proportion is about the same for the human brain.

Vision processing (environment perception) relies mostly on range cameras, because a range image can be easier segmented than an intensity image (see Kinect). A set of different range cameras (radar and lidar [Light Detection and Ranging, also ladar]) is used covering different depth ranges with resolution down to 0.1 degree and depth up to 300m.
Perception consists of the detection of pedestrians, bicyclists, cars, traffic signs, traffic lights, telephone poles, curbs, etc. The algorithms all appear to be based on techniques as we have introduced so far, meaning there is no particular magic involved; it rather requires an elaborate coordination of the different detection processes. One example was given already with car license plate recognition, we give more examples below. The recognition tasks are typically solved using multiple techniques complementing each other, as a single technique is often insufficient to reliably solve the problem.

**Roadway/marker Location** multilevel thresholding (subsection 6.1.1); vanishing point detection using RANSAC on edge points (see figure 22).

![Figure 22](source: Davies 2012; Fig 23.2)

**Location of Vehicles**

a) shadow induced by vehicle: Importantly, the strongest shadows are those appearing beneath the vehicle, not least because these are present even when the sky is overcast and no other shadows are visible. Such shadows are again identified by the multilevel thresholding approach (subsection 6.1.1).

b) symmetry: The approach used is the 1-D Hough transform, taking the form of a histogram in which the bisector positions from pairs of edge points along horizontal lines through the image are accumulated. When applied to face detection, the technique is so sensitive that it will locate not only the centerlines of faces but also those of the eyes. Symmetry works also for plant leaves and possibly other symmetric shapes.

![Figure 23](source: Davies 2012; Fig 23.6)

**Locating Pedestrians**

a) detection of body parts, arms, head, legs. The region between legs often forms an (upside-down) U feature.

b) Harris detector (subsection 3.1) for feet localization.

c) skin color (see also subsection 9.1).
A Appendix - Reduction/Classification/Clustering/Retrieval

Given are the data DAT and possibly some labels Lbl:
- DAT: \([n_{\text{Obs}} \times n_{\text{Dim}}]\) matrix with number of features times number of dimensions
- Lbl: \([n_{\text{Obs}} \times 1]\) vector with component values corresponding to category membership

A.1 Dimensionality Reduction with Principal Component Analysis

\[
\text{coeff} = \text{princomp}(\text{DAT}); \\
nPco = \text{round}(\min(\text{size}(\text{DAT}))*0.7); \\
\text{PCO} = \text{coeff}(:,1:nPco); \\
\text{DATRed} = \text{zeros}(n_{\text{Obs}},nPco); \\
\text{for } i = 1 : n_{\text{Obs}}, \\
\quad \text{DATRed}(i,:) = \text{DAT}(i,:) * \text{PCO}; \\
\text{end}
\]

A.2 Classification

We generate indices for training and testing set with crossvalind, then loop through the folds using classify or other classifiers:

\[
\text{Ixs} = \text{crossvalind(‘Kfold’, Lbl, 3);} \\
\text{Pf} = []; \\
\text{for } i = 1 : 3 \\
\quad \text{IxTst} = \text{Ixs}==i; \\
\quad \text{IxTrn} = \text{Ixs}‘=i; \\
\quad \text{LbOut} = \text{classify(DAT(IxTst,:), DAT(IxTrn,:), Lbl(IxTrn));} \\
\quad \text{nTst} = \text{nnz(IxTrn)}; \\
\quad \text{Pf(i) = nnz(LbOut==Lbl(IxTst))/nTst*100;} \\
\text{end}
\]

\[
\text{fprintf(‘Pc correct: %4.2f\n’, mean(Pf));}
\]

If the bioinfo toolbox is available, then the command classperf can be used to determine performance slightly more convenient.

K Nearest Neighbor (kNN) Training: uses the samples as memory without building a model for discrimination. Testing: a new sample is compared to all others. Matlab: knnclassify (bioinfo toolbox)

Linear Classifiers 2 variants: either the traditional and robust LDA (linear discriminant analysis) with classify in Matlab: it also allows multi-class categorization. Or the better performing but also trickier support vector machines svmclassify (bioinfo toolbox), which however is only a binary classifier.

Decision Trees classregtree

A.2.1 Performance Measures

For 3 or more classes, one usually determines the percentage of correct classification. For 2 classes (binary discrimination), one applies performance measures from information retrieval theory, namely precision, recall and F measure (see also subsection 9.2.2):

\[
\text{CM} = \text{accumarray}([\text{LbOut IxTst}], 1, [n_{\text{Cat}} n_{\text{Cat}}]); \\
\text{TP} = \text{diag(CM)}; \\
\text{FP} = \text{sum(CM,2)} - \text{TP};
\]

59
FN = sum(CM,1)' - TP;  % false negatives
Prec.all = TP ./ (TP + FP + eps);
Reca.all = TP ./ (TP + FN + eps);
TPoth = sum(TP(:))-TP;
Prec.oth = TPosth ./ (TPoth + FP + eps);
Reca.oth = TPosth ./ (TPoth + FN + eps);

%% F measure: 2 * Prec * Reca / (Prec + Reca)
Fmsr.all = 2 * Prec.all .* Reca.all ./ (Prec.all + Reca.all + eps);
Fmsr.oth = 2 * Prec.oth .* Reca.oth ./ (Prec.oth + Reca.oth + eps);

A.3 Clustering

Given are the data DAT in the format as above, but there are no labels available: we have to find them.

**K-Means** To apply this clustering technique we need to provide the number of assumed groups $k$ (denoted as $n_c$ called in our case). In Matlab: $Ixs = kmeans(DAT, nc)$. $Ixs$ is a vector of length $nObs$ containing numbers ($\in 1..n_c$) which represent the cluster labels.

**Hierarchical Clustering** Creates a hierarchy of the datapoints, which can be displayed as a tree. You need to provide a cut level between 0 and 2 that cuts the tree.
B Appendix - Color Spaces

A (digital) color image typically comes with a red-green-blue (RGB) encoding of the color spectrum. If one is interested only in the gray levels, then we can convert it to a grayscale image by adding the 3 components as follows (in Matlab rgb2gray):

$$0.2989 \times R + 0.5870 \times G + 0.1140 \times B$$

(Note: A grayscale image is also called a gray-scale, gray scale, or gray-level image).

If one intends to exploit color for segmentation or representation purposes for instance, then one applies the operations for each component, that is for each of the 3 color layers separately and combined the resulting 3 new maps somehow.

There are also other color spaces than the RGB space, also often expressed with 3 components, which may be worth looking at, because certain types of images are better segmented in those alternate color spaces. For instance, segmentation of natural objects (leaves, fruits, landscapes) is sometimes done in the Hue-Saturation-Value (HSV) space (matlab: rgb2hsv), which corresponds rather to human perception. (The space comes in variations and you also find HSI for instance)

Figure 24: The HSV color space/labeling. Hue: reflects our color ability to discriminate color. Brightness loosely reflects intensity. Saturation is a type of strength.
C  Coding

C.1 Programming Hints

To write fast-running code in Matlab, one should exploit Matlab’s matrix manipulating commands in order
to avoid the slower for loops (see for instance repmat or accumarray; see also Matlab’s documentation on
‘vectorization’). Writing a kNN classifier can be conveniently done using the repmat command. However,
when dealing with high dimensionality and large number of samples, exploiting this command can in fact
slow down computation because the machine will spend a significant amount of time allocating the required
memory for the large matrices. It may in fact be faster to maintain one for loop, and to use repmat only
limitedly in that case. Thus there is a speed/memory tradeoff.

C.2 Organizing Your Code

Images and Data Paths  Create separate folders for images, data obtained from preprocessing, data from
matching images, etc. Create a global structure variable, which contains the path names to those folders,
e.g.:

    FOLD.ImgSatImg = 'C:/CollectionSatelliteImages/';
    FOLD.ImgPanorama = 'D:/Photos/Panorama/';
    FOLD.DatSatImg = 'D:/Data/ImgPreProc/SatImg/';

With the command dir you can obtain a list of all images, e.g. ImgNames = dir(FOLD.ImgSatImg) and
then access the filenames FileNames(1).name. The first two entries will contain ‘.’ and ‘..’, thus start with
FileNames(3). Use fileparts to separate the path into its components. Create a script called init_img,
where you specify such paths and obtain the filenames. You will call that script at the beginning of each of
your scripts.

Time-Consuming Preprocessing  When you process hundreds of images - or even a few large ones -
and you tune an algorithm that operates only on the final steps of the recognition sequence, then you may
want to save some data from preprocessing first, and then load those data only, instead of carrying out the
time-consuming preprocessing every time. Use save and load for saving and loading data.

Overviewing Scripts  Create a file called README_SatImg (or some other appropriate ending), where
you list all the essential scripts line by line, with a few keywords on the right-hand side. It will help you
maintaining overview of your collection of scripts. From there you can easily access every script by moving
the mouse cursor onto the script name and right-clicking your mouse.
D Appendix Websites & Reading

D.1 Useful Websites

Many algorithms can be found on MathWork's file exchange website:
http://www.mathworks.com/matlabcentral/fileexchange

Peter Kovesi's website with lots of Matlab and Octave scripts:

Coding in Python:
http://programmingcomputervision.com/

Collection of sites for C/C++/... implementation
http://en.wikipedia.org/wiki/Opencv

Another site with computer vision code (link to tutorials)
http://www.aishack.in/topics/tutorials/vision/

On-line compendium:
http://homepages.inf.ed.ac.uk/rbf/CVonline/

Dataset, job offers, conference information, etc.:
http://www.computervisiononline.com/

D.2 Reading

See references (below) for publication details.

(Sonka et al., 2008): introductions to topics are broad yet the method sections are concise. Contains many, precisely formulated algorithms. Exhaustive on texture representation. Oriented a bit towards classical methods, thus, not all newer methods can be found. Written by three authors, but reads like if authored by one person only.

(Szeliski, 2011): meticulous and visually beautiful exposure of many topics, including on graphics and image processing; Strong at explaining feature-based recognition and alignment, as well as complex image segmentation methods with the essential equations only. Compact yet still understandable appendices explaining matrix manipulations and optimization methods.

(Forsyth and Ponce, 2010): exhaustive on topics on object, image and texture classification and retrieval, with many practical tips in dealing with classifiers. Equally exhaustive on tracking. Strong at explaining object detection and simpler image segmentation methods. Slightly more praxis oriented than Szeliski. Only book to explain image retrieval and image classification.

(Davies, 2012): rather machine vision oriented (than computer vision oriented). Contains extensive summaries explaining advantages and disadvantages of each method. Summarizes the different interest points detectors better than any other book. Treats video surveillance and automotive vision very thoroughly. Only book to contain automotive vision.

(Prince, 2012): also a beautiful exposure of some computer vision topics; very statistically oriented, starting like a pattern recognition book. Contains up-to-date reviews of some topics.

Wikipedia: Always good for looking up definitions, formulations and different viewpoints. Even textbooks sometimes point out wikipedia pages. But wikipedia's 'variety' - originating from the contribution of different authors - is also its shortcoming: it is hard to comprehend the topic as a whole from the individual articles (websites). Wikipedia is what it was designed for after all: an encyclopedia. Hence, textbooks remain irreplaceable.

Furthermore, because different authors are at work at wikipedia, it can happen that an intuitive and clear illustration by one author is being replaced by a different illustration (by another author), possibly less intuitive than the first illustration. I therefore recommend to copy/paste a well illustrated problem into a word editor (e.g. winword) in order to keep it.
References

E  Code Examples

E.1  Introduction

```matlab
clear;
Iorg = imread('KlausIohannisCrop.jpg');  % image is color [m n 3]
Ig = rgb2gray(Iorg);  % turn into graylevel image
[h w] = size(Ig);  % image height and width
Pver = sum(Ig,1);  % vertical intensity profile
Phor = sum(Ig,2);  % horizontal intensity profile

%% ---- Plotting
figure(1);clf;
subplot(1,2,1);
imagesc(Ig); colormap(gray);
subplot(2,2,2);
plot(Pver);
set(gca,'xlim',[1 w]);
title('Vertical Profile');
subplot(2,2,4);
plot(Phor);
set(gca,'xlim',[1 h]);
title('Horizontal Profile');

set(gcf,'paperposition',[0 0 8 4]);  % adjust plot size for printing
% print('FaceProfile','-djpeg','-r300');  % printing jpeg, resolution 300dpi
```

E.2  Image Processing I: Scale Space and Pyramid

```matlab
clear;
Icol = imread('autumn.tif');  % uint8 type; color
Ig = single(rgb2gray(Icol));  % turn into single type

%% ------ Initialize
nLev = 5;
[SS PY aFlt] = deal(cell(nLev,1));
SS{1} = Ig;  % scale space: make original image first level
PY{1} = Ig;  % pyramid: " " " "

%% ===== Scale Space and Pyramid
for i = 1:nLev-1
    Flt = fspecial('gaussian', [2 2]+i*3, i);  % 2D Gaussian
    aFlt{i} = Flt;  % store for plotting
    Ilpf = conv2(Ig, Flt, 'same');  % low-pass filtered image
    SS{i+1} = Ilpf;
    PY{i+1} = Ilpf;
    stp = 2*i;
    Idwn = downsample(Ilpf,stp);  % first along rows
    PY{i+1} = downsample(Idwn',stp)';  % then along columns
end

%% ----- Plotting
figure(1);clf;
[nr nc] = deal(nLev,3);
for i = 1:nLev
    if i<nLev,
        subplot(nr,nc,i*nc-2);
        imagesc(aFlt{i});
    end
    subplot(nr,nc,i*nc-1);
    imagesc(SS{i});
    subplot(nr,nc,i*nc);
    imagesc(PY{i});
end
```

65
### E.3 Image Processing II: Segmentation

```
clear;
I   = imread('coins.png');

%% ----- Thresholding with Otsu and Median
  tOts = graythresh(I);
  BWots = im2bw(I,tOts);
  I   = single(I); % the following computation is simpler in single
  tMed = median(I(:))/max(I(:));
  BWmed = im2bw(uint8(I),tMed);

%% ----- K-means
  Ix = kmeans(I(:),2);
  BWkmen = false(size(I));
  BWkmen(Ix==2) = true;

%% ----- Watershed
  Ilpf = conv2(I,fspecial('Gaussian',[5 5],2));
  W   = watershed(Ilpf);
```

### E.4 Feature Extraction II

#### E.4.1 Edge Detection

```
clear; format compact;
sgm = 1; % scale, typically 1-5

%% ----- Load an Image
  I   = double(imread('cameraman.tif'));

%% ----- Blurring
    Nb = [2 2]+sgm;
    Fsc = fspecial('gaussian', Nb, sgm); % 2D gaussian
    Iblr = conv2(I, Fsc, 'same');

%% ----- Edge Detection
    BWrob = edge(Iblr, 'roberts');
    BWsob = edge(Iblr, 'sobel');
    BWpre = edge(Iblr, 'prewitt'); % similar to Sobel
    BWlog = edge(Iblr, 'log', [], sgm); % laplacian of Gaussian
    BWzex = edge(Iblr, 'zerocross'); % For Canny we apply to original image I (because it will perform the % blurring
    BWcny = edge(I, 'canny', [], sgm);

%% ----- Plotting
  figure(1); clf; colormap(gray); [nr nc] = deal(3,2);
  subplot(nr,nc,1); imagesc(BWrob); title('Roberts');
E.4.2 Edge Following

clear;
\%
\% ------ Stimulus
M = zeros(15,15); \% an empty map
M([5 10], 5:10) = 1; \% upper & lower sides of a rectangle
M(5:10, [5 10]) = 1; \% left & right sides of a rectangle
M(3:8, 3) = 1; \% straight line
\%
\% ------ Labeling, Tracing
ML = bwlabel(M);
CC = bwconncomp(M);
aBon = bwboundaries(M);
BonR = btraceboundary(M, [5 5], 'N');

\%
\% ------ Plotting
figure(1); clf;
imagesc(M); hold on;
nB = length(aBon);
for i = 1:nB
   Bon = aBon{i};
   plot(Bon(:,2), Bon(:,1), 'c');
   plot(BonR(:,2)+.2, BonR(:,1)+.2, 'g');
end

E.4.3 Blobs

clear;
Icol = imread('tissue.png');
I = single(rgb2gray(Icol));

\%
\% ------ Diff-of-Gaussians (DOG)
Fs1 = fspecial('gaussian', [11 11], 3); \% fine low-pass filter
Fs2 = fspecial('gaussian', [21 21], 6); \% coarse low-pass filter
Is1 = conv2(I, Fs1, 'same'); \% convolution with image
Is2 = conv2(I, Fs2, 'same');
Idog = Is2 - Is1; \% diff-of-Gaussians
BWblobs = Idog > 15; \% thresholding for blobs

\%
\% ------ Laplacian-of-Gaussian (LoG)
Flog = fspecial('log', [15 15], 5);
Ilog = conv2(I, Flog, 'same');
BWlog = Ilog > .6;

\%
\% ------ Plotting
figure(1); clf;

[nc, nr] = deal(3, 2);
subplot(nr, nc, 1), imagesc(Icol);
subplot(nr, nc, 2), imagesc(I);
subplot(nr, nc, 3), imagesc(Idog); colorbar;
subplot(nr, nc, 4), imagesc(BWblobs);
subplot(nr, nc, 5), imagesc(Ilog); colorbar;
subplot(nr, nc, 6), imagesc(BWlog);
E.4.4 Texture Filters

% A filter bank for texture. (The Leung-Malik filter bank)
% see also: http://www.robots.ox.ac.uk/~vgg/research/texclass/filters.html

function F = f_GenTxtFilt(figNo)

sz = 49; % filter size
SclBlb = sqrt(2).^(1:4); % sigma for blob filters
SclOri = sqrt(2).^(1:3); % sigma for oriented filters
nSclBlb = length(SclBlb);
nSclOri = length(SclOri);

nBlb = 12; % # of blob filters (first 12 filters)
nOri = 6; % # of orientations

ntFlt = nSclOri*nOri*2 + nBlb; % # of total filters

%% ----- Init Memory & Points
F = zeros(sz,sz,ntFlt);
rd = (sz-1)/2;
[X Y] = meshgrid(-rd:rd, rd:-1:-rd);
PTS = [X(:) Y(:)]';

%% ----- Blob Filters (first 12 filters)
for i = 1:nSclBlb
    F(:,:,i) = ff_Norm(fspecial('gaussian', sz, SclBlb(i)));
    F(:,:,4+i) = ff_Norm(fspecial('log', sz, SclBlb(i)));
    F(:,:,8+i) = ff_Norm(fspecial('log', sz, SclBlb(i)*4));
end

%% ----- Edge & Bar Filters (filters 13-48)
cc = 1;
for s = 1:nSclOri,
    for o = 0:nOri-1,
        ang = pi*o/nOri; % Not 2pi as filters have symmetry
        ca = cos(ang);
        sa = sin(ang);
        PTSrot = [ca -sa; sa ca] * PTS;
        F(:,:,12+cc) = ff_Edg(PTSrot, SclOri(s), sz); % edge
        F(:,:,30+cc) = ff_Bar(PTSrot, SclOri(s), sz); % bar
        cc = cc+1;
    end
end

%% ----- Plotting
figure(figNo); clf; colormap(gray);
for i = 1:ntFlt
    subplot(8,6,i)
    I = F(:,:,i);
    imagesc(I);
    set(gca,'fontsize',4);
    title(num2str(i),'fontsize',6,'fontweight','bold');
end
end % MAIN

%% ============== SUB FUNCTIONS ============
function F = ff_Bar(PTS, sgm, sz)
    Gx = normpdf(PTS(1,:), 0, sgm*3);
    Gy = normpdf(PTS(2,:), 0, sgm);
    vec = sgm*2;
    Gy = Gy .* ((PTS(2,:).^2 - vec.^2)/(vec.^2)); % 2nd derivative
    F = ff_Norm(reshape(Gx.*Gy, sz, sz));
end % SUB

function F = ff_Edg(PTS, sgm, sz)
    Gx = normpdf(PTS(1,:), 0, sgm*3);
    Gy = normpdf(PTS(2,:), 0, sgm);
    Gy = Gy .* ((PTS(2,:)./(sgm^2))); % 1st derivative
    F = ff_Norm(reshape(Gx.*Gy, sz, sz));
end % SUB

function D = ff_Bure(D)
    D = D - mean(D(:));
    D = D / max(abs(D(:)));
end % SUB
E.5 Shape

E.5.1 Generating Shapes

clear;
I = ones(150,760)*255; % empty image
sz = 28;
Ix = 65:90; % indices for A to Z
nShp = length(Ix);
figure(1);clf;
imagesc(I); hold on; axis off;
for i = 1:nShp
   ix = Ix(i);
   text(i*sz,20,char(ix),'fontweight','bold');
   text(i*sz,50,char(ix),'fontweight','bold','fontsize',12);
   text(i*sz,110,char(ix),'fontweight','bold','fontsize',12,'rotation',45);
   text(i*sz,80,char(ix),'fontweight','bold','fontsize',14);
end
print('ShapeLetters','-djpeg','-r300');

E.5.2 Simple Measures

clear;
BW = rgb2gray(imread('ShapeLetters.jpg')) < 80;
%% ===== Shape Properties
RG = regionprops(BW, 'all');
Ara = cat(1,RG.Area);
Ecc = cat(1,RG.Eccentricity);
EqD = cat(1,RG.EquivDiameter);
BBx = cat(1,RG.BoundingBox);
Vec = [Ara Ecc EqD]; % [nShp 3] three dimensions
nShp = length(Ara);
fprintf('# Shapes %d
', nShp);
%% ===== Pairwise Distance Measurements
DM = squareform(pdist(Vec)); % distance matrix [nShp nShp]
DM(diag(true(nShp,1))) = nan; % inactivate own shape
[DO O] = sort(DM,2,'ascend'); % sort along rows
%% ----- Plotting First nSim Similar Shapes for Each Found Shape
% Bounding box
UL = floor(BBx(:,1:2)); % upper left corner
Wth = BBx(:,3); % width
Hgt = BBx(:,4); % height
mxWth = max(Wth)+2;
mxHgt = max(Hgt)+2;
nSim = 20; % # similar ones we plot
nShp2 = ceil(nShp/2);
[ID1 ID2] = deal(zeros(nShp2*mxWth,nSim*mxHgt));
for i = 1:nShp
   % --- given/selected shape in 1st row
   Row = (1:Hgt(i))+UL(i,2); % rows
   Col = (1:Wth(i))+UL(i,1); % columns
   Sbw = BW(Row,Col);
   if i<=nShp2, ID1((1:Sbw)+i*mxHgt, 1:Sbw*2) = Sbw*2;
   else ID2((1:Sbw)+i-nShp2*mxHgt,1:Sbw*2) = Sbw*2;
end

% --- similar shapes in rows 2nd to nShp+1
for k = 1:nSim
   ix = 0(1,k);
   Row = (1:Hgt(ix))+UL(ix,2); % rows
   Col = (1:Wth(ix))+UL(ix,1); % columns
   Sbw = BW(Row,Col);
end
SZS = size(SBW);
if i<=nShp2, ID1((1:SZS(1))+(i-1)*mxHGT, (1:SZS(2))+(k*mxWTH)) = SBW;
else ID2((1:SZS(1))+(i-nShp2)*mxHGT, (1:SZS(2))+(k*mxWTH)) = SBW;
end
end
end

figure(1);clf;
subplot(1,2,1); imagesc(ID1); title('First Half of Letters');
subplot(1,2,2); imagesc(ID2); title('Second Half of Letters');

E.5.3 Shape: Radial Signature

clear;
BW = imread('text.png'); % the stimulus
aBonImg = bwboundaries(BW);
nShp = length(aBonImg);
fprintf('# Shapes %d
', nShp);

%% ===== Shape Properties
Vec = zeros(nShp,4);
aBon = cell(nShp,1);
for i = 1:nShp
    Bon = aBonImg{i};
    nPix = length(Bon);
    cenPt = mean(Bon,1);
    Rsig = sqrt(sum(bsxfun(@minus,Bon,cenPt).^2,2));
    Fdsc = abs(fft(Rsig)); % fast Fourier
    Fn = Fdsc(2:end)/Fdsc(1); % normalization by 1st FD
    fprintf('%3d #pix %2d FFD %2d
', i, nPix, length(Fn));
    Vec(i,:) = Fn(1:4);
    aBon{i} = bsxfun(@minus,Bon,cenPt-[10 10]); % move into [1..20 1..20]
end
clear aBonImg

%% ===== Pairwise Distance Measurements
DM = squareform(pdist(Vec)); % distance matrix [nShp nShp]
DM(diag(true(nShp,1))) = nan; % inactivate own shape
[DO O] = sort(DM,2,'ascend'); % sort along rows

%% ----- Plotting First nSim Similar Shapes for Each Found Shape
sz = 19;
nSim = 20; % # similar ones we plot
nShp2 = ceil(nShp/2); % [ID1 ID2] = deal(zeros(nShp2*sz,nSim*sz));
figure(1);clf;
subplot(1,2,1); imagesc(ID1); title('First Half of Letters'); hold on;
subplot(1,2,2); imagesc(ID2); title('Second Half of Letters'); hold on;
for i = 1:nShp
    % --- given/selected shape in 1st row
    Bon = aBon{i};
    if i<=60,
        subplot(1,2,1);
        plot(Bon(:,2), Bon(:,1)+(i-1)*sz,'b');
    else
        subplot(1,2,2);
        plot(Bon(:,2), Bon(:,1)+(i-61)*sz,'b');
    end
    % --- similar shapes in rows 2nd to nShp+1
    for k = 1:nSim
        ...
ix = 0(i,k);
Bon = aBon{ix};
if i<=60
    subplot(1,2,1);
    plot(Bon(:,2)+k*sz,Bon(:,1)+(i-1)*sz,'k');
    % ID1((1:Szs(1))+(i-1)*mxHgt, (1:Szs(2))+(i-1)*sz) = Sbw;
else
    subplot(1,2,2);
    plot(Bon(:,2)+k*sz,Bon(:,1)+(i-61)*sz,'k');
    %ID2((1:Szs(1))+(i-45)*mxHgt,(1:Szs(2))+(i-45)*sz) = Sbw;
end
end
end
E.6 Tracking

To play the movie for illustration, uncomment the line starting with 'movie(Moc,...' (line 14 approximately).

clear;
sR = 7; % radius of search region (block matching)
wR = sR+2+1; % diameter of search region

%% Load Movie
ObjVid = VideoReader('xylophone.mpg'); % movie ‘handler’
FRAMES = read(ObjVid); % actual movie data
[nRow nCol dmy nFrm] = size(FRAMES); % rows/columns/colors/frames
MOC(1:nFrm) = struct('cdata', zeros(nRow, nCol, 3, 'uint8'), 'colormap', []);
MOG = zeros(nRow,nCol,nFrm,'uint8'); % movie in grayscale

for k = 1:nFrm
    MOC(k).cdata = FRAMES(:,:,k); % movie in color
    MOG(:,:,k) = rgb2gray(MOC(k).cdata); % movie in grayscale
end

%% LOOPING FRAMES
b_pause = 1; figure(1); clf;

%% Difference Frames
Fprv = MOG(:,:,i-1); % previous frame
Fnow = MOG(:,:,i); % current frame
FDf = abs(Fprv-Fnow); % difference image [nrow ncol]
FDforig = FDf; % keep original for plotting
FDf(FDf<20) = 0; % threshold to eliminate 'noise'

%% Localize patches that change
[LF nMot] = bwlabel(FDf); % find connected components
[IXPCH Sz] = deal({}, []); % pixel indices and sizes
for k = 1:nMot % loop thru detected motion patches
    bR = LF==k;
    IXPCH{k} = find(bR); % pixel indices
    Sz(k) = nnz(bR); % patch size
end
fprintf('Frm %d # of motions %3d,	 patch sizes %2d-%4d,
        %d,%d ', i, nMot, min(Sz), max(Sz),
        %d,%d );

%% Select Large Patches/Changes
[Szs Ixs] = sort(Sz, 'descend'); % sort patch sizes
bLrg = Sz>10; % detect large ones
nLrg = nnz(bLrg); % # of large ones
IXPCH = IXPCH(Ixs(bLrg)); % reduce to detected large changes
fprintf('
reduced to %d large patches
', nLrg);

%% Bounding Box of patches
PchBB = [];
cP = 0;
for k = 1:nLrg
    IXP = IXPCH{k};
    [Row Col] = ind2sub([nRow nCol], IXP);
    [leb rib] = deal(min(Col), max(Col)); % left and right boundary
    [upb lob] = deal(min(Row), max(Row)); % upper and lower boundary
    [leb rib] = deal(max(leb,sR+1), min(rib,nCol-sR-1));
    % --- check width and height (maybe 0 or even negative)
    width = rib-leb;
    height = lob-upb;
    if width<1 || height<1, continue; end % if any 0, then move on
    % --- patch bounding box (coordinates)
    cP = cP+1;
    PchBB(cP,:) = [upb lob leb rib]; % upper/lower/left/right
end

%% Match each motion patch with its neighborhood
MchBB = [];
for k = 1:cP
    Co = PchBB(k,:); % coordinates [1 4]
rRow = Co(1):Co(2); % range rows
rCol = Co(3):Co(4); % range cols
Fprv = Fprv(rRow,rCol); % patch in previous frame
ps = size(Fprv);
% --- correlation with neiboring patches
CorrPtch = zeros(wR,wR);
for m = -sR:1:sR
    for n = -sR:1:sR
        Pnow = Fnow(rRow+m, rCol+n);
        CorrPtch(m+sR+1,n+sR+1) = corr2(Pprv,Pnow);
    end
end
CorrPtch(sR+1,sR+1) = 0; % set own match to 0
% --- selection
[v1 ix] = max(CorrPtch(:)); % select highest correlation
[rr cc] = ind2sub([wR wR], ix); % linear index to subindices
MchBB(k,1:2) = Co(1:2)+rr-sR-1; % store bounding box of best match
MchBB(k,3:4) = Co(3:4)+cc-sR-1;
if 1
    figure(10); clf
    imagesc(CorrPtch); colormap(gray); hold on;
    plot(cc,rr,'*')
    pause();
end
end

%% ------------------------ Plotting
MotPresFrm2(i) = sum(FDf(:));
if b_pause,
    figure(1); [rr cc] = deal(2,2);
    subplot(rr,cc,1), imagesc(Fprv); colormap(gray); title(['Frame ' num2str(i)]);
    subplot(rr,cc,2), imagesc(255-FDforig); title('Difference Image');
    for k = 1:cP
        Ix = PchBB(k,:); Lo1 = [Ix(3) Ix(1) Ix(4)-Ix(3) Ix(2)-Ix(1)];
        Ix = MchBB(k,:); Lo2 = [Ix(3) Ix(1) Ix(4)-Ix(3) Ix(2)-Ix(1)];
        rectangle('position', Lo1, 'edgecolor', 'b');
        rectangle('position', Lo2, 'edgecolor', 'r');
    end
    subplot(rr,cc,3), imagesc(255-FDf); title('Thresholded Diff Img');
    subplot(4,2,6), hist(FDforig(:),1:255); title('Histogram of Differences');
    subplot(4,2,8), hist(Sz,1:10:1000); title('Histogram of Patch Sizes');
    pause();
end
end
E.7 2D Transformations

% Examples of 2D transforms and their motion estimation. See pdf 36, 312.

clear;

%% Simple Pattern
Co = [0 0; 0.6 0.75; 1.2 0.8; 1.0 0.8; 1.2 0; 0 0]*4+2; % distorted rectangle
np = size(Co,1); % # of points
cpt = mean(Co,1); % center point
Cpt = repmat(cpt,np,1); % replicated center point

%% ================== Transformations ==================

% -------------- Transformation Matrices
TRot = [cos(r) -sin(r); sin(r) cos(r)]; % clockwise rotation
TEuc = [TRot; [tx ty]']; % Euclidean (rot&trans) aka rigid body motion
TEuc0 = [TEuc; [0 0 1]]; % with row extended
TShr = [1 0; 0.5 1]; % shear
TScl = [r 0; 0 r]; % scaling
TSim = [1+a -b tx; b 1+a ty]; % similarity
TSim0 = [TSim; 0 0 1]; % with row extended
TAff = [1+a00 a01 tx; a10 1+a11 ty; 0 0 1]; % affinity (with row extended)

% ------------- perform transformations
% transf. mx are transposed to match column-wise coordinates:
Coc = Co-Cpt; % subtract cenpt: coordinates 0,0 centered
% --- rot, shear, scale:
Crot = Coc * TRot' + Cpt; % rot transformation and add cenpt
Cshr = Coc * TShr' + Cpt; % shear 
Cscl = Coc * TScl' + Cpt; % scale 
% For aff/euc/sim: add row of 1s, then transform
Caff = [Coc ones(np,1)] * TAff'; % affinity " here we add a col of 1s
Ceuc = [Coc ones(np,1)] * TEuc0'; % Euclidean " here we add a col of 1s
Csim = [Coc ones(np,1)] * TSim0'; % similarity " here we add a col of 1s
Caff(:,[1 2]) = Caff(:,[1 2])+Cpt; % THEN add cenpt
Ceuc(:,[1 2]) = Ceuc(:,[1 2])+Cpt;
Csim(:,[1 2]) = Csim(:,[1 2])+Cpt;

%% ================== Estimating Motion ==================

% Building A and b
JSim = inline('[1 0 x -y; 0 1 y x]');
JAff = inline('[1 0 x y 0 0; 0 1 0 0 x y]');
% JEu = inline('[1 0 -sin(r)*x -cos(r)*y; 0 1 -cos(r)*x -sin(r)*y]');
Asim = zeros(4,4); bsim = 0; % init A and b for similarity case
Aaff = zeros(6,6); baff = 0; % init A and b for affinity case

for i = 1 : np
    pt = Coc(i,:); % original 0,0 centered
    Jp = JSim(pt(1),pt(2));
    Asim = Asim + Jp'*Jp;
    bsim = bsim + DltSim(i,:)*Jp;
end

% Least-Square for A and b for Similarity
disp('Similarity');
[Prm1 resn1] = lsqnonneg(Asim, bsim'); % Prm1(3:4) contain estimates for a and b
[Prm2 resn2] = lsqlin(Asim, bsim');
Prm1'
Prm2'
(Prm1(1:2)-cpt')' % translation parameters (tx, ty)

% Least-Square for A and b for Affinity
disp('Affinity');
[Prm1 resn1] = lsqlin(Aaff, baff'); % Prm1(3:6) contain estimates for a00, a01, a10, all baff
[Prm2 resn1] = lsqlin(Aaff, baff');
Prm1'
Prm2'
(Prm1(1:2)-cpt')' % tx and ty
E.8 RanSAC

% Random Sampling Consensus for affine transformation.
% No correspondence determined - assumes list entries correspond already.
% IN: - Pt1 list of original points [np1,2]
% - Pt2 list of transformed points [np2,2]
% - Opt options
% OUT: - TP struct with estimates
% - PrmEst [nGoodFits x 6] parameters from lsqlin
% - ErrEst [nGoodFits x 1] error
function TP = f_RanSaC(Pt1, Pt2, Opt, b_plot)

TP.nGoodFits = 0;
if isempty(Pt1) || isempty(Pt2), return; end

JAff = inline('[1 0 x y 0 0; 0 1 0 0 x y]);
np1 = size(Pt1,1);
np2 = size(Pt2,1);

fprintf('Original has %d points, transformed has %d points
', np1, np2);
nMinPt = Opt.nMinPts; % # of minimum pts required for transformation
nCom = np1-nMinPt; % # of complement pts

cpt = mean(Pt1); % center point of original
Pt1Cen = Pt1 - repmat(cpt,np1,1); % original 0,0 centered

%% Iterating

while ~isempty(Pt1) && ~isempty(Pt2) && nGoodFits < Opt.nMaxIter && b_plot

%% Random Subset (Samples)

Ixrs = randsample(np1, nMinPt); % random sample indices
Ixcm = setdiff(1:np1, Ixrs); % complement

%% Estimation with Random Samples

Dlt = Pt2(Ixrs,:)-Pt1Cen(Ixrs,:); % delta (transformed-original 0,0 centered)
Atot = zeros(6,6); b = 0; % init Atot and b
for i = 1:nMinPt
pt = Pt1Cen(Ixrs(i),:);
Jp = JAff(pt(1),pt(2));
Aot = Aot + Jp'*Jp; % building A
b = b + Dlt(i,:)*Jp; % building b
end

[prm err] = lsqlin(double(Atot), double(b')); % prm(3:6) contain a00,..

[tx ty] = deal(prm(1)-cpt(1), prm(2)-cpt(2)); % tx and ty
eqclose = 'tx %1.4f ty %1.4f ';
fprintf(eqclose, tx, ty);

%% Transform Complement

[a00 a10 a01 a11] = deal(prm(3), prm(4), prm(5), prm(6));
Taff = [1+a00 a01 tx; a10 1+a11 ty; 0 0 1]; % affinity (row extended)
Pt1Com = Pt1Cen(Ixcm,:); % the complement pts

Caff = [Pt1Com ones(nCom,1)] * Taff'; % we add a col of 1s

cff = Caff(:,1:2); % close enough?

err = cff - Pt2(Ixcm,:); % err = cff - complement

if sum(sum(abs(err))) < Opt.thrNear*2, nNear = nnz(bNear);

fprintf('#GoodFits %d out of %d iterations
', nGoodFits, Opt.nMaxIter);

end

end % function

%% TESTING SCRIPT t_Ransac

clear;

nP = 40;

%% Simple Pattern
CoOrig = rand(nP,2).*3+5;

np = size(CoOrig,1); % # of points
cpt = mean(CoOrig,1); % center point
Cpt = repmat(cpt,np,1); % replicated center point

%% ================== Transform Original ==================

tx = 4; % x-translation
ty = 1.5; % y-translation
a = 0.25; % scale (for similarity)
b = 0.35; % rotation (for similarity)

% -------------- Transformation Matrices
TSim = [1+a -b tx; b 1+a ty]; % similarity
TSim0 = [TSim; 0 0 1]; % with row extended

% -------------- Perform Transformations
Coc = CoOrig-Cpt; % subtract centpt: coordinates 0,0 centered
Csim = [Coc ones(np,1)] * TSim0'; % similarity " here we add a col of 1s
Csim(:,[1 2]) = Csim(:,[1 2])+Cpt; % adding some noise

%% ================== Plotting ==================

gonfig(1);clf;hold on;
plot(CoOrig(:,1), CoOrig(:,2), 'g.'); % original set of points
plot(CoTrns(:,1), CoTrns(:,2), 'r.'); % transformed set of points
axis equal
legend('original', 'transformed', 'location', 'northwest');

%% ================== RanSaC ==================

disp('Verifying f_RanSaC');
Opt.nMinPts = 6;
Opt.nMaxIter = 10;
Opt.thrNear = 0.2;
Opt.nNear = 3;
Opt.xlb = ' ';
Opt.Match = ' ';  
b_plot.dist = 1;
RSprm = f_RanSaC(CoOrig, CoTrns, Opt, b_plot);
RSprm'
RSprm.PrmEst
76
F Example Questions

F.1 Questions

1. What is an interest point detector? What does it look for? How is it implemented?
2. What is the dominant principle of recognizing image content (in computer vision) recently?
3. You have extracted thousands of image patches from a collection of images. How do you continue to build a recognition system, let's say an image classification system.
4. How are image patches (obtained around interest points) compared with each other?
5. What is an (image) pyramid? What is it good for?
6. What is the image gradient? How is it formed?
7. What type of texture representations do you know?
8. How can you align two images (that have overlapping scene content)? What are the challenges?
9. How can you track objects (in a video)?
10. How does the linear least-squares optimization method work?
11. What image segmentation methods do you know? Advantages, disadvantages?
12. What is 'RanSaC'? What is it good for? How does it work?
13. What is image classification? What examples do you know?
14. Where do you place a camera for video surveillance? And why? Why is foreground/background separation difficult?
15. What is optic flow? What can it be used for? How is it computed in principal?
16. What does the OF for a moving object look like (observer is stationary)?
17. What is a frequently used principle to build an object detection algorithm (for objects with limited structural variability)? How does it work? When does it not work?
18. What are active contours? What are they good for? How does their principle work?
19. How do you collect your training set for object recognition?
20. What is image retrieval? How is it implemented (with modern methods)?
21. With what kind of recognition task is an automated vehicle equipped (to drive through traffic)? How are these tasks achieved? With what methods?

F.2 Answers (as hints)

1. An interest point detector finds 'salient' points in an image, that can be used for matching. It looks for sharp corners or intersections of contours or anything 'unusual'. It is implemented by cross-correlating the pixel values of a local patch. See script for more details.
2. It is the detection of interest points and the matching of their corresponding local neighborhood (patches) with each other.
3. Identify the most important patches by clustering the set of image patches, that is building the equivalent of a dictionary (visual words).
4. Typically a gradient histogram is formed. The most popular one is the SIFT feature, where gradients are take blockwise from a square-shaped patch...see script for details. For specific objects, such as pedestrians, the gradients are collected from more complicated spatial arrangements.

5. Is a collection of subsampled, filtered images (of the original image). It provides more (structural) information (than just the original image alone), and can be used to speed up the feature detection process, exploiting a coarse-to-fine search.

6. Describes the direction and magnitude of change in the intensity 'landscape' at a given point. It is formed by firstly low-pass filtering the original image (to eliminate 'noise') and then by taking the derivative in each direction...see script for details.

7. Statistical and structural. Statistical: e.g. gray-level occurrence matrix; structural: processing with various kinds of filters and building a dictionary. See script for details.

8. Find interest points and match their corresponding matches. The challenge is to find the corresponding matches due to the presence of 'noise'. A transformation combined with RandSAC is one method to find the exact alignment.

9. By tracking detection or by tracking by matching - as simplest methods.

10. It takes the distance between points and the estimated straight line and minimizes the sum of squared distances, hence the name 'least-squares'.

11. Thresholding, watershed, clustering,...see script for details.

12. Random sampling consensus: it is an estimation procedure that can particularly deal well with outliers. It takes a subsample of the set and makes and estimation, which then is confirmed by the complement of the subset. Repeated application of this will find the best 'match'. see script for more details.


14. Above pedestrian's height to detect the feet and to perform a triangulation. See script for details.

15. Optic flow describes the direction and velocity of individual points (or patches) of a moving object or scene (but not the object as a whole!). It can be used for (refined) tracking and image segmentation.

16. In the simplest case it is a set of equal sized motion vectors pointing into the same direction.

17. Sliding window technique, see script for details.

18. It is a segmentation method, in particular object segmentation. It is a precise segmentation and can be used for tracking for instance. The principle is to optimize a sum of constraints (edge, contrast, ...).

19. Target images containing the object; distractor images containing 'noise', e.g. natural scenes.

20. It is the search for images given a query image as provided by a user. It is implemented using a visual vocabulary (patch based matching) and evaluated with retrieval measures form information theory.

21. Roadmarker location, vehicle location, pedestrian location; see script for details.