Computer Vision

C. Rasche
compvis12 [at] gmail [dot] com
February 21, 2019

This is a dense introduction to the field of computer vision. It covers all topics including Deep Neural Networks (TensorFlow and PyTorch). It provides plenty of code snippets and copy-paste examples for Matlab, Python and OpenCV (accessed through Python).

We firstly sketch some of the basic feature extraction methods as those help us to understand the architecture of Deep Neural Networks. After we have introduced DeepNets, we sketch object detection and recognition techniques. Then we turn toward the more classical techniques, such as feature extraction and matching based on histograms of gradients. It follows a treatment of image processing techniques - segmentation and morphological processing - and of shape and contour recognition techniques. We overview the essential tracking methods. We close with a survey of video surveillance, in-vehicle vision system, text recognition and remote sensing.

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

Computer Vision is the field of interpreting image content. It is concerned with the classification of the entire image, such as in a system classifying photos uploaded to the internet (Facebook, Instagram). Or Computer Vision is concerned with the recognition of objects in an image, such as detecting faces or car license plates (Facebook, GoogleStreetView). Or it is concerned with the detection of aspects of an image, such as cancer detection in biomedical images.

Origin Computer Vision was originally founded as a sub-discipline of the field of Artificial Intelligence in the 1970s. The founding goal was to create a system that has the same perceptual capabilities as the human visual system has - your eyes and most of your brain. The human visual system can easily interpret any scene with little effort: it perfectly discriminates between thousands of categories, and it can find objects in scenes within a time span of several hundred milliseconds only; it easily switches between several types of recognition processes with a *flexibility* and swiftness, whose complexity and dynamics have not been well understood yet. It quickly turned out, that that goal was rather ambitious.

Instead, Computer Vision has focused on a set of specific recognition challenges, to be introduced in Section 1.2. Those challenges can be often implemented in different ways, with each implementation having advantages and disadvantages. Throughout the decades, many applications have been created (Section 1.3), and some of those implemented tasks begin now to outperform a human observer - such as face identification, letter recognition, or the ability to maneuver through traffic (autonomous vehicles). And that itself is astounding, even though the original goal of an omni-vision system has not been achieved yet. Today, Computer Vision is considered its own field.

Frontier Computer Vision is still considered a frontier, despite its evolvement over almost 50 years. The success of modern Computer Vision is less the result of truly novel algorithms, but rather a result of increasing computer speed and memory. In particular shape recognition is - despite its simple sounding task - still not properly understood. And even though there exist neural networks that can recognize thousands of classes, the system occasionally fails so bluntly, that one may wonder what other algorithms need to be invented in order to achieve a flawless recognition process. If those algorithms will not be invented, then household robots may always make some nerve-wrecking errors, such as mistaking the laundry basket for the trash bin, confusing the microwave with the glass cabinet, etc. Thus, despite all the progress that has been made, it still requires innovative algorithms.

In particular in the past several years, Computer Vision has received new impetus by the use of so-called Deep Learning algorithms, with which one can classify decently large image collections. Google and Facebook are competing to provide the best interface to those learning algorithms: Goolge offers *Tensorflow*, Facebook provides *PyTorch*, both of which are Python-based libraries to run classification algorithms. And that is the reason why we treat that topic relatively early (Section 5), after a quick warm up with the classical methods. We then proceed with a method that had been popular just before the arrival of Deep Learning algorithms, namely feature extraction and matching (Sections 7 and 8). Later on, we continue with traditional techniques (Section 9), and we also mention approaches to the most enigmatic challenge of Computer Vision, namely shape recognition (Section 11) and contour description (Section 12).

1.1 Related Fields

Several fields are related to Computer Vision, two of which are closely related, namely *Image Processing* and *Machine Vision*; in fact, those two fields overlap with Computer Vision to such an extent, that their names are sometimes used synonymously. Here is an attempt to discriminate between them, 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 low-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 object needs to be identified.
3) the exact orientation in 3D is of interest.

An example is car license plate detection and reading at toll gates, which is a relatively controlled situation. In comparison, computer vision systems often deal with objects of larger variability and objects that are situated in varying backgrounds. Car license plate detection in GoogleStreetView is an example of an object with limited variability but varying context.

There exist two other fields that overlap with Computer Vision:

Pattern Recognition (Machine Learning) is the art of classification (or 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, as in case of image classification, for which so-called Deep Neural Networks have produced the best classification accuracy so far (Section 5). Clearly, we cannot treat classification in depth in this course and we will merely point out how to use some of the classifiers (see also Appendices F and G).

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 introduce the three principal recognition processes and their challenges. Then we explain what those processes are called in motion analysis.

Classification (Categorization): an object or scene is assigned to a class (category), such as 'car', 'apple', 'beach scene', etc. The challenge here is to deal with the so-called *intra-class variability*, the variability among class instances. In some classes the variability is small, e.g. bananas look relatively the same; in some classes, the variability is near endless - think of how differently chairs can look like, including designer chairs.

Identification: an individual instance of an object class is recognized, such as in face identification, fingerprint identification, identification of a specific airplane, etc. In principle, this process can be regarded as classification at a much finer level.

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.

In the literature, the term *object recognition* often implies some combination of those three processes - occasionally it stands for only one of them.

Now we introduce recognition processes in *motion analysis*. Motion analysis is a general term for reconstructing where and how something has moved between two moments in time. There are two principal type of movements: either the objects in the scene have moved; or the observer has moved - in which case the entire scene appears as moving.

Tracking: is the pursuit of one or several (moving) objects in an image - it can be regarded as frame-by-frame object detection. Examples: tracking a user's face for human-computer interaction; tracking pedestrians in surveillance; etc. Tracking answers *where* the object has moved.

Motion Estimation: is the reconstruction of the precise movement between successive frames or between frames that are separated by a short duration. It answers *how* the object or scene has moved. It helps to reconstruct the precise pose of objects or an observer's viewpoint.

Motion Classification: is the challenge of recognizing entire, completed motions as in gesture recognition or human body movements. It answers *what* type of movement the object has carried.

1.3 Areas of Application (Examples)

Sze p7

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.

Retrieval: as in image search on Google for instance.

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: defect 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.

For more application examples one can visit Google's sites that offer various products:

`https://cloud.google.com/vision/`

`https://js.tensorflow.org/`: recognition for web-browers

1.4 Styles of Methodology

There exist two types of methodology to solve a vision task, see the two pathways in Figure 1. The left path represents the traditional methodology and is sometimes called *feature engineering*. The right path stands for the modern methodology and is sometimes called *feature learning* - it can be considered the

more successful avenue these days, as it has proven to outperform many 'engineered' systems in many tasks. We elaborate a bit.

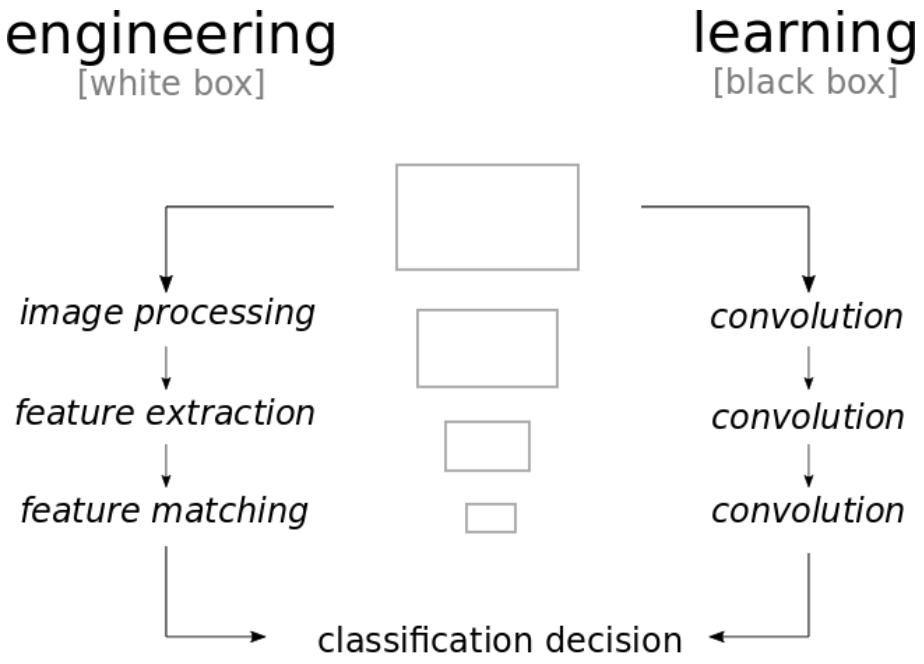

Figure 1: Two styles of solving recognition tasks: *feature engineering* (left) versus *feature learning* (right). Feature engineering is the more traditional approach (Sections 7, 8, 16); feature learning the modern approach (Section 5). Both have advantages and disadvantages.

The **feature engineering** approach typically starts with an *image processing* phase whose aim is to prepare the image for the subsequent processing phases, as mentioned under 'Image Processing' in Section 1.1. In a second phase, the *feature extraction* phase, the image is searched for features that are robust under changes in light, for instance contour corners, so-called blobs, etc. Those are then matched in a third phase, the *feature matching* phase, to some sort of representation in order to arrive at a classification decision. This approach is sometimes labeled a white box system, as the individual techniques are individual, the processing steps are more explicit and as there are few parameters. In case of failure of a (single) recognition run, it is possible to pinpoint to the insufficient processing step.

The **feature learning** approach is pursued by developers of the neural network methodology, nowadays simply called *deep learning*. The image is processed by a series of *convolution* phases, by which features are gradually extracted and then integrated to a 'global' image map, from which the decision is made. The learning procedure involves finding the 'weights' for the innumerous convolutions in an automatic manner. In comparison to the feature engineering approach, the techniques here are less individual, the processing steps are less explicit, and the system has innumerous parameters. In case of failure, it is difficult to pinpoint an exact location of weakness.

The two depictions are exagerated formulations - created by the proponers of the respective philosophies. In the engineering approach, not everything is as fully transparent as the label 'white box' would suggest. And in the learning approach, not everything is completely automatically learned - there is substantial tuning involved sometimes. Thus it would be more appropriate to use the terms brighter and darker, or more and less transparent. And because the deep learning approach has drawn inspiration from some of the engineered architectures, one should not be surprised if combined approaches will appear in the near future. In fact, first DeepNets have started to appear that include also traditional methodology.

1.4.1 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, starting from the 2D image layout. This 3D reconstruction process was often divided into low-level, mid-level and high-level vision process, a division partly reflected in the above list of stages. It was inspired by the fact that we humans perceive the world as a 3D space. Over the years, it has become clear that this paradigm is too elaborate and too complicated. Presently, the focus lies on solving recognition tasks with 'brute-force' approaches, the two pathways depicted in Fig. 1, for which classical techniques such as edge detection or image segmentation hardly play a role. 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 feature-matching approache (left side in Fig. 1), but still contains substantial material on image segmentation for instance. But no book contains the latest, breath-taking developments, namely the use of Deep Neural Networks for image classification. We therefore will start with that topic relatively early (Section 5).

1.5 From Development to Implementation

Usually one develops a system first in a higher-level language, such as Matlab, Python, GNU Octave, R, Scilab, etc. Once this testing phase is completed, then one would 'translate' the system into a lower-level language such as Cython, C++ or even C for instance, to make the application run in real-time if that is necessary.

Matlab: (`http://www.mathworks.com/`) Is extremely convenient for prototyping (research) because its 'formulation' is very compact and because it probably has the largest set of functions and commands. It offers an image processing toolbox that is very rich in functionality, use `doc images` do get the overview. And it offers a computer vision toolbox with a lot of tutorials for complex systems; use `doc vision` for the overview.

Octave, R: (`https://www.gnu.org/software/octave/`, `https://www.r-project.org/`) For training purposes one can certainly also use software packages such as R and Octave, in which most functions have the same name as in Matlab.

Python: (`https://www.python.org/`) Is perhaps the most popular language by now. Coding in Python is slightly more elaborate than in Matlab and does not offer the flexibility in image display that Matlab has. Python's advantage is, that it can be relatively easily interfaced to other programming languages that are suitable for mobile app development for instance, whereas for Matlab this is very difficult. In Python, the initialization process is a bit more explicit and the handling of data-types (integer, float, etc.) is also a bit more elaborate, issues that make the Python code a bit lengthier than in Matlab.

If one switches between any of those high-level languages, then the following summary is useful:
`http://mathesaurus.sourceforge.net/matlab-python-xref.pdf`.

In the following we mention programming languages that are considered rather lower-level and that require more care in initializing and maintaining variables. If you process videos, then you probably need to implement your time-consuming routines into one of those languages.

Cython: (`https://www.cython.org/`) Is essentially the same code as Python, but offers to specify certain variables and procedures in more detail with a notation similar to C (C++). That additional notation can speed up the code by several factors. Cython is included in the Anaconda distribution. (Not to be confused with *CPython*, which is the canonical implementation of the Python syntax).

C++, C: For implementation into C or into one of its variants (i.e. C++), we merely point out that there exist C libraries on the web with implemented computer vision routines. The most prominent one is called Open CV, see wiki OpenCV or `https://opencv.org/`. Many of the routines offered by those libraries can also be easily accessed through Python by importing them.

For more information on training material and coding we refer to Appendix I.

1.6 Reading

Here I list in particular books introducing the concepts. There are many more books providing details on implementations, in particular on OpenCV.

Sonka, M., Hlavac, V., and Boyle, R. (2008). *Image Processing, Analysis, and Machine Vision*. Thomson, Toronto, CA. 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, R. (2011). *Computer Vision: Algorithms and Applications*. Springer. 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, D. and Ponce, J. (2010). *Computer Vision - A Modern Approach*. Pearson, 2nd edition. Exhaustive on topics about 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 with feature methods.

Davies, E. R. (2012). *Computer and Machine Vision*. Elsevier Academic Press, Oxford. 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, S. (2012). *Computer Vision: Models, Learning, and Inference*. Computer Vision: Models, Learning, and Inference. Cambridge University Press. 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 less intuitive one by another author. I therefore recommend to copy/paste a well illustrated problem into a word editor (e.g. winword) in order to keep it.

With regard to image processing I recommend the following book as it does not treat topics in too much mathematical detail, and which therefore is sufficient for our purposes:

Gonzalez, R. C. and Woods R.E. (2017). *Digital Image Processing*: an introduction oriented toward Matlab.

2 Simple Image Manipulations (First Steps)

To get acquainted with some of the basics, we perform a few simple image manipulations in this section. Firstly, we learn about the image format and some basic operations such as thresholding and data type conversions (Section 2.1). Then we mention some of the image enhancement techniques that are occasionally used for computer vision systems (Section 2.2). Finally, we introduce a simple detector for face part localization to understand the complexity of the intensity landscape (Section 2.3).

2.1 Image Format, Thresholding, Conversion

A typical digital image, for instance a jpeg image, comes as a three-dimensional array. The first two dimensions correspond to the spatial axes x and y. The third dimension holds color information in three chromatic channels, namely red, green and blue (RGB). The color values generally range from 0 to 255 and are stored as unsigned integers, specifically as `uint8`, the number 8 standing for 8 bits, a data-type designed to hold exactly that range ($255 = 2^8 - 1$). For each pixel then, there exist 24 bits (3*8 bits), which allows to store 256 x 256 x 256 $\approx$ 16.7 million colors. Despite that rich color information, it is often more convenient to computer only with a gray-scale version of that image, thus reducing the information back to 8 bits per pixel. To convert a RGB image into gray-scale image, one converts the color values to a gray value L by adding the three components according to a specific ratio, for instance:

$$L = 0.2989 \cdot R + 0.5870 \cdot G + 0.1140 \cdot B \tag{1}$$

In Matlab one can load an image with the command `imread`. To convert it into a gray-level image there exists the function `rgb2gray`. To display the image we use the function `imagesc` (image scale) and for that purpose we initialize a figure with the function `figure`. The function `clf` clears the figure. With the command `subplot` we can pack several images into the same figure. The following code shows how to use those commands, its output is shown in Figure 2.

```
clear;                              % clear memory
Irgb    = imread('yellowlily.jpg'); % load jpg image
Igry    = rgb2gray(Irgb);           % convert it to gray-scale
IrgbCen = Irgb(400:1200,300:900,:); % zoom into center
Igreen  = Irgb(:,:,2);              % green channel only
BWflw   = Igry>100;                 % thresholded (black-white image)
Iblur   = conv2(single(Igry),ones(25,25)/625);    % blurring the image

%% -----    Plotting    ------
figure(1); clf; [nr nc] = deal(3,2);
subplot(nr,nc,1); imagesc(Irgb); title('Original');
subplot(nr,nc,2); imagesc(Igry); colormap(gray); title('Gray-Scale');
subplot(nr,nc,3); imagesc(IrgbCen); title('Sub-Selection (Zoom)');
subplot(nr,nc,4); imhist(Igreen);   title('Histogram of Green Channel');
subplot(nr,nc,5); imagesc(BWflw);   title('Black-White (Logical) Image');
subplot(nr,nc,6); imagesc(Iblur);   title('Blurred Image');
```

To select a part of the image - see comment 'zoom into center' -, we specify the row numbers first - vertical axis first -, followed by specifying the column numbers - horizontal axis. That is, one specifies the indices as in matrices in mathematics.

Black-White Image We can threshold an image by applying a relational operator, see line `BWflw = Igry>100`, in which case the image is automatically converted into a *logical* data-type, that is `true` or `false`, namely one bit (value one and zero respectively). An image of that data-type is also called a *black-white* image sometimes, hence the variable's name `BW`. In the code example above we attempted to separate the flower from its background, a foreground/background segregation as it is also called. We have chosen the threshold somewhat arbitrarily and of course it would make sense to choose a threshold based on a

histogram, a histogram of intensity values for instance, as shown in the figure. We elaborate on that in Section 9. After we have segmented, one often manipulates the black-white image by so-called *morphological* operations, to be introduced in Section 10.

Blurred Image Sometimes it is useful to blur an image because a blurred image helps analyzing the 'coarse' structures of an image, which otherwise are difficult to detect in the original image with all its details. We can blur an image by averaging over a local neighborhood at each pixel in the image, in Matlab done with the function `conv2`. In our example we take a 25x25 pixel neighborhood, generated with `ones(25,25)` and merely sum up its 625 pixel values: this summation operation is done for each pixel in the image. Normally one divides the sum by the filter size in which case one speaks of a *box filter*. We will come back to more blurring filters in Section 3.

Data-Type Conversion (Casting) The function `imread` returns a jpeg image as data-type `uint8`, which is not very practical for certain computations. For many image-processing functions we need to cast (convert) the image into a floating-number data-type. In Matlab that can be done with the functions `single` or `double`, returning lower and higher precision respectively. In the above code, we did that casting for the function `conv2`, but we could also write a separate line if desired, `Irgb = single(Irgb)`.

Many functions are flexible and will produce an output of the same data-type; others expect a specific data-type as input; some functions produce a specific data-type as output such as the thresholding operation. It is best to be always aware of what data type an image is - or any variable -, and what type the functions expect and produce.

In Python the code looks very similar, but we need to 'import' those functions from modules. In particular the module `skimage` holds a lot of functions for computer vision and image processing, see also Appendix K:

```
from numpy           import arange, ones, histogram
from skimage.io      import imread
from skimage.color   import rgb2gray
from scipy.signal    import convolve2d

Irgb    = imread('someImage.jpg')   # loading the image
Igry    = rgb2gray(Irgb)            # convert to gray-scale
IrgbCen = Irgb[100:500,400:700,:]       # zoom into center
Igreen  = Irgb[:,:,2]                   # green channel only
BWflw   = Igry>0.3                      # thresholding (black-white image)
Iblur   = convolve2d(Igry,ones((25,25))/625)  # blurring the image

#%% ---- Plotting
from matplotlib.pyplot import figure, subplot, imshow, plot, hist, title, cm
figure(figsize=(10,6)); (nr,nc) = (3,2)
subplot(nr,nc,1); imshow(Irgb); title('Original')
subplot(nr,nc,2); imshow(Igry, cmap=cm.gray); title('Gray-Scale')
subplot(nr,nc,3); imshow(IrgbCen); title('Sub-Selection (Zoom)')
subplot(nr,nc,4); hist(Igreen.flatten(),bins=arange(256));   title('Histogram of Green Channel')
subplot(nr,nc,5); imshow(BWflw);   title('Black-White (Logical) Image')
subplot(nr,nc,6); imshow(Iblur, cmap=cm.gray);   title('Blurred Image')
```

Note that in Matlab the function `rgb2gray` returns a map of data-type `uint8` - if the input was of type `uint8`. In Python however there exist many functions to read an image and some of them perform scaling to an interval $\in$ [0 1.0].

Topology To obtain an impression what type of stimulus an image represents, we recommend to observe the image with the function `mesh`, as given in the last line in the Matlab code block. This illustrates better

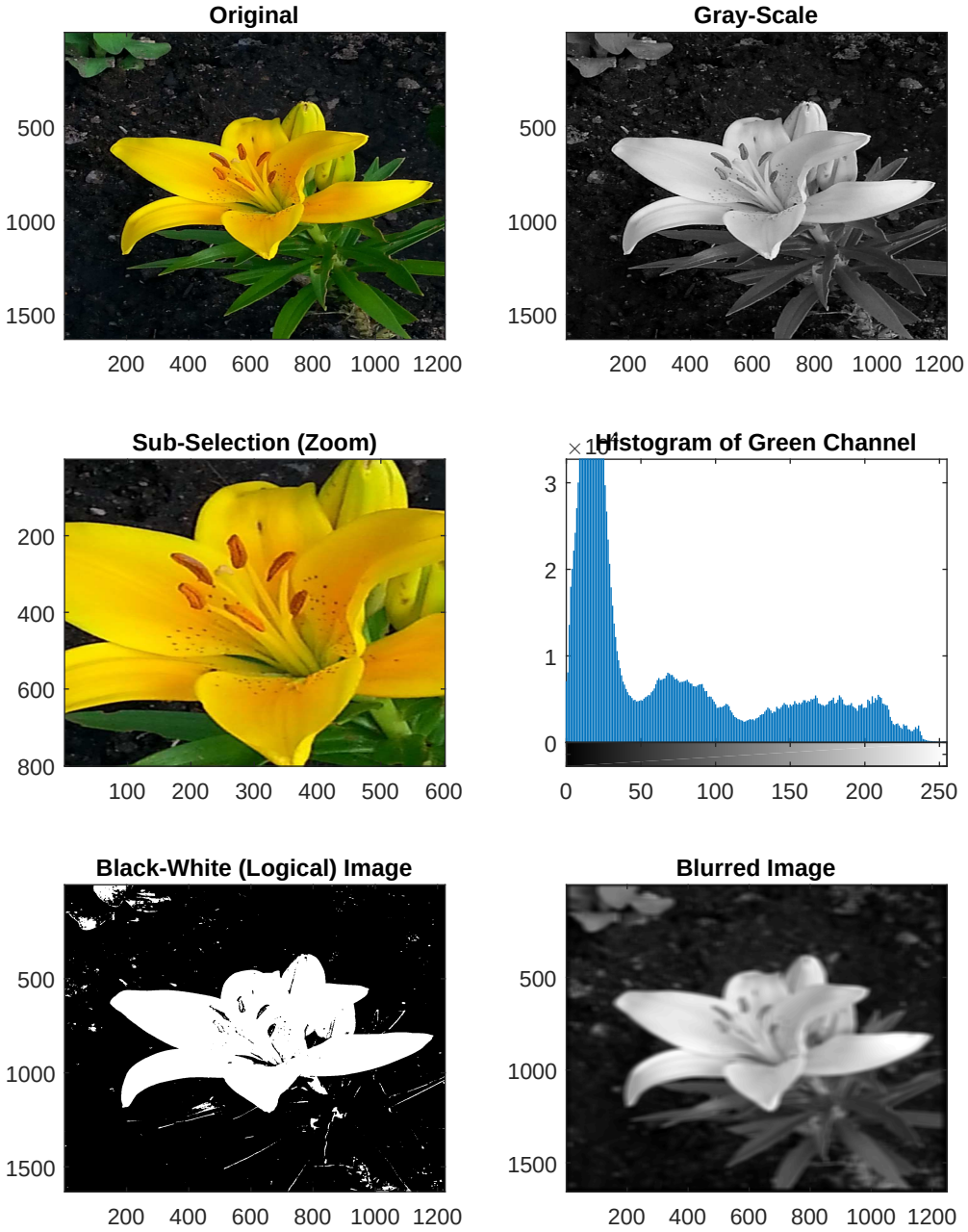

Figure 2: Some simple image manipulations.
Upper Left: original RGB image; each pixel is represented by 24 bits (3 chromatic channels $\times$ 8 bits).
Upper Right: gray-level: RGB values combined to a single luminance value according to Eq. 1; each pixel is now represented by 8 bits only (values from 0 to 255).
Center Left: zoomed image; selection occurs with matrix-style indexing (rows/columns).
Center Right: image histogram of the green channel: the x-axis represents the chromatic value, ranging from 0 to 255; the y-axis represents the pixel-count.
Lower Left: result of thresholding the gray-level image at value equal 100: white represents on-pixel (true), black off-pixel (false).
Lower Right: gray-level image blurred with an average filter: at each pixel the average over its 25x25 pixel-neighborhood is taken.

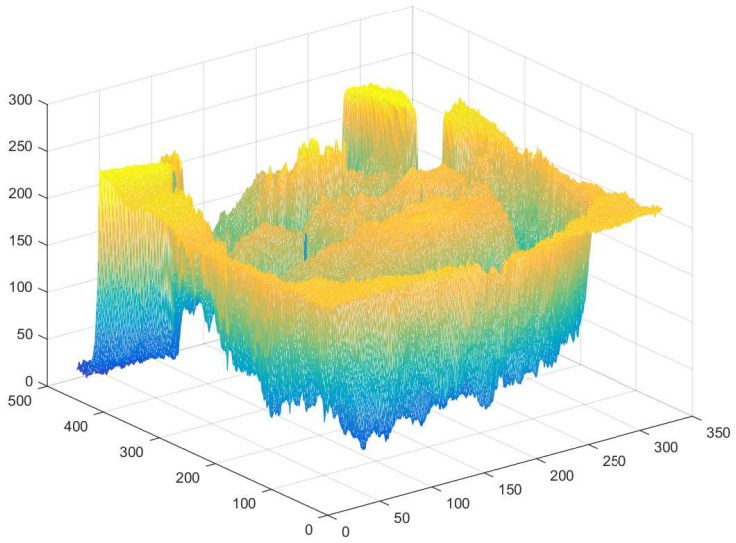

Figure 3: An image as observed from a three-dimensional perspective (command `mesh` in Matlab). The image appears as a landscape whose elevation - the vertical axis - represents intensity with values ranging typically from zero to 255 (for images coded with 8 bits). It is not easy for a human to understand the semantic content of a scene from this perspective, because the human visual system is trained to interpret frontal views of images. But this perspective illustrates better what the computer vision system receives as input.

that the image array holds an 'intensity landscape' (Fig. 3). For that reason computer vision scientists often borrow terminology from the field of topology to describe the operation of their algorithms, such as the term 'watershed' in case of a segmentation algorithm.

2.2 Image Enhancement

If an image shows low contrast, then it can be manipulated to show higher contrast by shifting its pixel values according to some distribution. The manipulations are typically based on the image histogram (Fig. 2 center right). In an image with low contrast, the histogram shows a peak at either side, see Fig. 4 left side. During the equalization process, that histogram peak is moved more toward the center (right side in Figure), based on an *input histogram* distribution. The result is often visually appealing and is also used occasionally when training recognition systems.

In Matlab the functions `imadjust`, `histeq` and `adapthisteq` carry out such image enhancements. For instance when using `histeq` without any parameters,

```
Iheq    = histeq(Igry);          % histogram equalization with a flat 64-bin histogram
```

it will automatically assume a flat input histogram made of 64 bins.

In Python those functions are within module `skimage.exposure`. For instance with `equalize_hist` the histogram equalization is carried, whereby there the default input histogram consists of 256 bins.

Sometimes, images happen to be noisy. For example, digitization of old photographs leads often to images in which single pixels stand out of their immediate neighborhood, a type of 'noise' that is also called *salt-and-pepper* noise, because those stand-out pixels appear like grains of salt (bright) or pepper (dark) on an image. In that case we can take a median filter, a filter that replaces a pixel value with the median value of its neighborhood. Typically, one chooses a 3x3 neighborhood and that is the default if one uses Matlab's function `medfilt2`:

```
Igry    = medfilt2(Igry);          % median filter for a gray-level map
```

If one intends to denoise a color image, then one would apply the median filter to the individual chromatic channels.

In Python the function `medfilt2d` can be found in module `scipy.signal`.

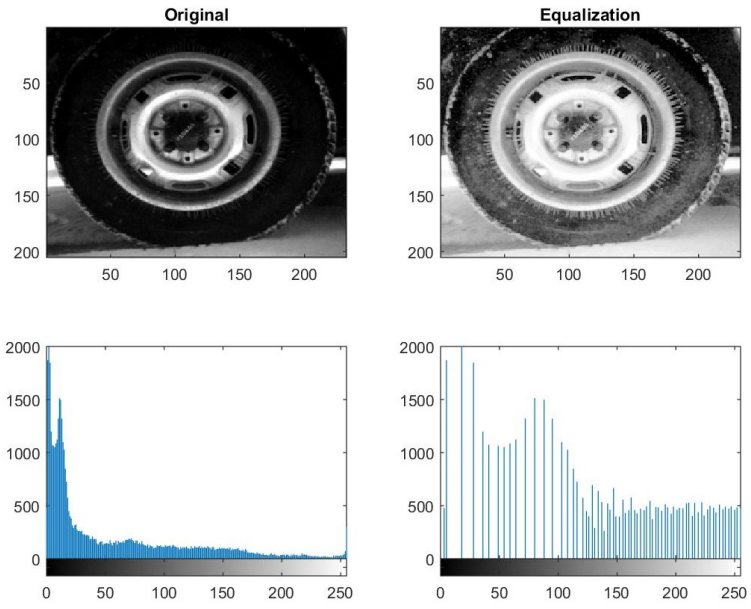

Figure 4: Image enhancement with simple manipulations using its histogram for redistribution. This has visual appeal primarily, but is also carried out for learning classification tasks sometimes.

2.3 Face Part Detection

With some simple image manipulations we can sometimes gain a lot of information. For instance to detect facial features in a portrait, it can suffice in many cases to look at the sum of pixel intensity values along each dimension, see Fig. 5. We generate two such *profiles*, one by integrating the pixel values along the horizontal axis (dimension), and one by integrating along the vertical axis. In Matlab this is easily done as follows:

```
Pver    = sum(Ig,1);                    % vertical intensity profile
Phor    = sum(Ig,2)';                   % horizontal intensity profile
```

Those profiles are shown in black in the Fig. 5. In such profiles it is - in principle - relatively easy to locate facial features by observing local maxima, minima, etc. The 'raw' profile sums are a bit 'noisy' however - like most raw signals: they contain too many 'erratic' extrema that make detection of the facial features difficult. We therefore smoothen the profiles a bit by low-pass filtering them. The smoothened versions are shown in magenta in the Figure; now it is easier to locate facial features. Smoothening can be done by averaging over a local neighborhood in the signal at each pixel in the profile. We have done this above already to obtain a blurred image in two dimensions with the function `conv2`. Here we do it in one dimension only. And we do so with a so-called Gaussian filter, a function whose shape is a 'bump': it is an elegant way to filter a signal, see Appendix C.2 for its exact shape. For the blurring of the image above we had used a flat filter, which is a rather crude way.

The size of the filter matters: a small size does not smoothen very much, a large size flattens the signal to much. We therefore make the filter size dependent on the image size by choosing a fraction of it. The Gaussian values are generated with function `pdf`.

```
nPf     = round(w*0.05);                % # points: fraction of image width
LowFlt  = pdf('norm', -nPf:nPf, 0, round(nPf/2)); % generate a Gaussian
Pverf   = conv(Pver, LowFlt, 'valid');  % filter vertical profile
Phorf   = conv(Phor, LowFlt, 'valid');  % filter horizontal profile
```

If you are not familiar with the convolution process you should consult Appendix B - for the moment the Appendix B.1 on one-dimensional convolution is sufficient. Those concepts are part of the field of signal

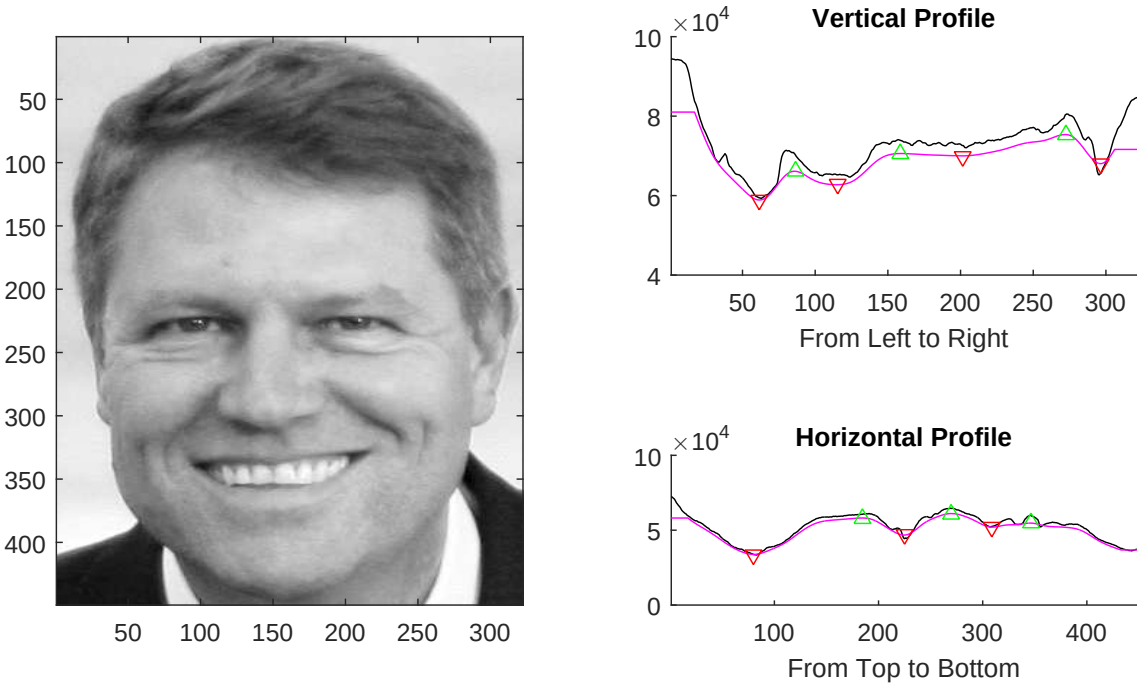

Figure 5: Intensity profiles for a face image. The vertical profile is generated by summing the pixel intensity values along the y-axis (vertical; column-wise); the horizontal profile is generated by summing along the x-axis (horizontal; row-wise). To what face parts do the extrema in the profiles correspond to? Code in Appendix **L.2**.

processing and we cannot give lengthy introductions to those. But with Matlab you can conveniently explore them and obtain a quick understanding of those operations, that is why we give a lot of code examples in the Appendix.
For extrema detection we can use the function `findpeaks`, which returns the local maxima as well as their location in the signal. In order to find the local minima we invert the distribution. The code in Appendix **L.2** shows how to apply those functions.

This type of face part localization is somewhat simple, but its temporal complexity is low and that is why this approach is often used as a first phase in a more elaborate facial feature tracking system. Many applied computer vision systems consist of cascades of sub-systems that progressively carry out a task with increasing precision.

3 Image Processing I: Scale Space, Pyramid and Gradient Image

For many computer vision tasks, it is useful to blur the image and to analyze those different blurs separately. We have introduced the idea of a blurred image already in the previous section, but here we introduce a more systematic approach that generates this blur repeatedly with increasing filter sizes arriving at a so-called *scale space* as shown in Fig. 6. This space will be introduced in the first Section 3.1. Then we introduce the image pyramid, Section 3.2, which is - coarsely speaking - a reduction of the scale space.

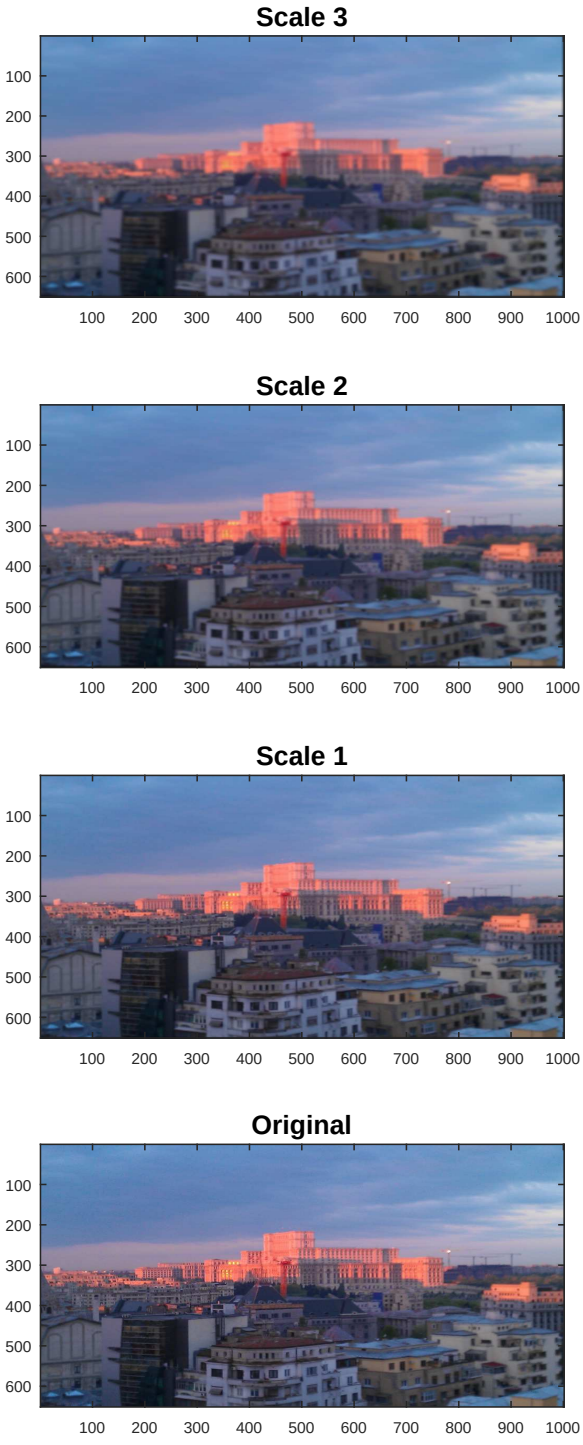

Figure 6: Scale space of an image. Observe the increasing blur from bottom to top.

Original: unfiltered image I_o.
Scale 1: I_o was smoothened with a Gaussian filter of sigma equal one, $\sigma = 1$.
Scale 2: I_o was smoothened with $\sigma = 2$.
Scale 3: I_o was smoothened with $\sigma = 3$.

It is called a space because one can regard it as a three-dimensional space, with the third dimension corresponding to a fine-to-coarse axis with variable σ.
At coarser scales (larger values of σ) it is easier to find contours and regions as a whole, but structures are sometimes smeared with other different structures. Coarser scales are often down-sampled to obtain a more compact representation of the scale space, which so forms a pyramid, see Fig. 7. Code in Appendix **L.3**.

(In this case, the filtering was performed for each color channel separately, once for the red, once for the green and once for the blue image.)

For many computations it is also useful to know how the intensity landscape is 'oriented' at each pixel. Specifically, we would like to know the 'surface slope' at each image pixel for a small pixel neighborhood. This is expressed with the gradient image, to be explained in Section 3.3.

3.1 Scale Space

wiki Scale_space

Sze p127, s3.5, p144
FoPo p164, s4.7, p134
SHB p106, s4.3

An image is blurred by convolving it with a two-dimensional filter that averages across a small neighborhood. In our introductory example of the previous Section we merely used a summation function, but typically image blurring is done with a Gaussian filter as we did for filtering the face profiles (Section 2.3). Here the Gaussian filter is a two-dimensional function and looks like shown in the first four patches of Fig. 11. It is expressed as $g(x, y, \sigma)$, where x and y are the image axes and where σ is the standard deviation regulating the amount of blur - also called the smoothing parameter. In the language of signal processing one expresses the blurring process as a convolution, indicated by the asterisk $*$. One says, the image $I_o(x, y)$ is convolved by the filter $g(x, y, \sigma)$

$$I_c(x, y) = I_o(x, y) * g(x, y, \sigma), \tag{2}$$

resulting in a coarser image I_c. If you are unfamiliar with the 2D-convolution process, then consult now Appendix B.2 to familiarize yourself with it.

If this blurring is done repeatedly, then the image becomes increasingly coarser, the corresponding intensity landscape becomes smoother, illustrated in Fig. 6. Practically, there are different ways to arrive at the scale space. The most straightforward implementation is to low-pass filter the original image repeatedly with 2D Gaussians of increasing size, that is first with $\sigma = 1$, then with $\sigma = 2$, etc. That approach is however a bit slow and there exist fast implementation, for which we refer to the Appendix B.2.

The resulting stack of images, $\{I_o, I_1, I_2, \dots\}$, is called a *scale space*. In a typical illustration of the scale space, the bottom image corresponds to the original image; subsequent, higher images correspond to the coarser images. In other words, the images are aligned vertically from bottom to top and one can regard that alignment as a *fine-to-coarse* axis. That axis is now labeled σ. The resulting space is then expressed as

$$S(x, y, \sigma). \tag{3}$$

The axis σ can be understood as a smoothing variable: a σ-value equal zero corresponds to the original image, that is, there is no smoothing; a σ-value equal one corresponds to the first coarse image, etc. In applications, sigma values typically range from one to five, $\sigma = 1, 2, .., 5$.

Application The scale space is used for the following feature detection processes in particular:

1. Region finding: we can easily determine brighter and darker regions by subtracting the scales from each other, a technique we will return to in Section 4.1.
2. Verification of structures across the scale (axis): for instance, if a specific feature can be found at different scales, then it is less likely to be accidental, a technique to be used in feature detection in general (Section 7).
3. Finding more 'coherence': for instance, contours appear more continuous at coarser scales (Sections 4.2 and 12).

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

```
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 also offers commands such as `imgaussfilt` and `imfilter` to generate blurs of images and are probably the preferred functions, because those are optimized for speed.

In Python we can use the submodule `scipy.ndimage` that contains the function `gaussian_filter` to blur images:

```
scipy.ndimage.gaussian_filter(I, sigma=1.0)
```

Generating such a scale space gives us more information that we intend to exploit to improve our features extraction. This gain in information comes however at the price of more memory usage, as we now have multiple images. And multiple images means more computation when searching for features. But we can reduce this scale space, without much loss of information by 'compressing' the coarse scales, which leads to the *pyramid* treated in the next section.

Figure 7: Multi-resolution pyramid of an image. The images of the scale space (Figure 6) are down-sampled by every second pixel: the resolution is halved with each scale along the fine-to-coarse axis.

Original: P_0 = original image I_o
Level 1: P_1 = sub-sampled of I_{c1} ($I_{c1} = I_o * g(1)$)
Level 2: P_2 = sub-sampled of I_{c2} ($I_{c2} = I_o * g(2)$)
Level 3: P_3 = sub-sampled of I_{c3} ($I_{c3} = I_o * g(3)$).

It is called a pyramid, because one could align the images such that they form a pyramid - for simplicity we show the images frontally. Code in Appendix **L.3**.

3.2 Pyramid

wiki Pyramid_(im

Operating with the entire scale space S is computationally expensive, as it is rather large and to some extent redundant. Because coarser scales do not possess any fine structure anymore, it makes sense to subsample them: for each new, coarser scale I_c, only every second pixel is taken, once along the horizontal axis and once along the vertical axis. We so arrive at the (octave) pyramid with levels P_0 to P_3 for instance, see Fig. 7. In higher-level languages we can down-sample 'manually' by selecting every second row and column, i.e. sub-sampling rows with:

```
Idwn = Isc1(1:2:end,:); % sub-sampling rows
```

followed by sub-sampling columns by

```
Idwn = Idwn(:,1:2:end); % sub-sampling columns
```

In Matlab there exists also the function `downsample`, which however works for rows only - if the input is a matrix; we then need to flip the matrix to down-sample it along the columns. Matlab also offers the function `impyramid`, which carries out both the Gaussian filtering as well as the down-sampling. But we can also operate the other direction and up-sample:

```
I1  = impyramid(I0,'reduce');  % filter and downsampling every 2nd
I0  = impyramid(I1,'expand');  % upsampling every 2nd
```

If other down- or up-sampling steps - than halving and doubling- are required, then the function `imresize` may be more convenient.

In Python we find those functions in the module `skimage.transform`, namely as `pyramid_reduce` and `.pyramid_expand`. See also Appendix D for an overview.

Application In many matching tasks, it is more efficient to search for a pattern starting with the top level of the pyramid, the smallest resolution, and then to work downward toward the bottom levels, a strategy also called coarse-to-fine search (or matching). More specifically, only after a potential detection was made at the coarse level, then one starts to verify by moving towards finer levels, where the search is more time consuming due to the higher resolution.

3.3 Gradient Image

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. At each pixel, two measures are determined: the direction of the slope, also called the gradient; and the magnitude - the steepness - of the slope. Thus the gradient image consists of two maps of values, the direction and the magnitude. That information is most conveniently illustrated with arrows, namely as a vector field, see Figure 8: the direction of the arrow corresponds to the gradient; the length of the arrow corresponds to the magnitude.

To determine the gradient, one takes the 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 ∇, the gradient operator:

$$\nabla I = (\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y})^T. \tag{4}$$

whereby the gradient information is expressed as a vector. 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 $[0, 2\pi]$ - and a certain magnitude representing the steepness. The direction is computed with the arctangent function using two arguments (atan2), returning a value $\in [-\pi, \pi]$ in most software implementations. The magnitude, $\parallel \nabla I \parallel$, is computed using the Pythagorean formula.
Determining the gradient field directly on the original image does not return 'useful' results typically, because the original image is often too noisy (irregular). Hence the image is typically firstly low-pass filtered with a small Gaussian function, e.g. $\sigma = 1$, before determining its gradient field.

In Matlab there exists the routine `imgradient` which returns the magnitude and direction immediately (Image Processing Toolbox); if one needs also the derivatives later again in the code, then one can use `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:

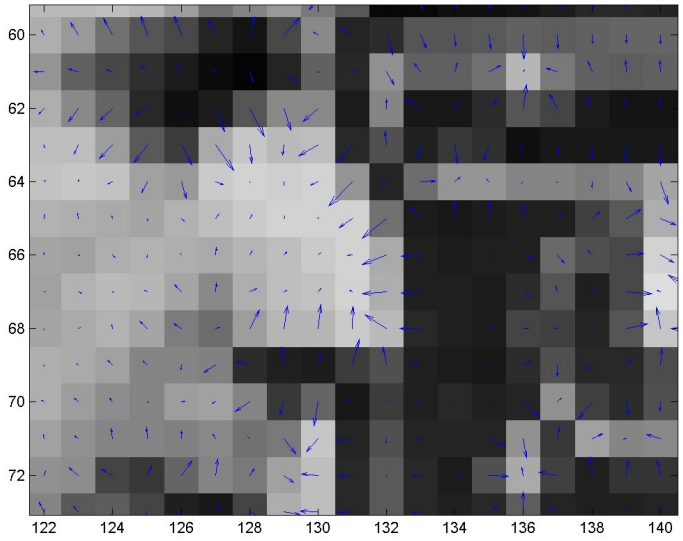

Figure 8: Gradient image. The arrows point into the direction of the gradient - the local slope of the neighborhood; the length of the arrows corresponds to the magnitude of the gradient. The gradient direction points toward high values (white=255; black=0).
(What does the picture depict? Hint: generate a coarse scale by squinting your eyes.)

```
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`:

```
% --- 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);
```

In Python we can use the function `gradient` of the `numpy` module:

```
Dx, Dy = numpy.gradient(I)
```

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

4 Feature Extraction I: Regions, Edges & Texture

Now we make a first step toward reading out the structure in images. We will try to find edges in the intensity landscape, dots, regions, repeating elements called texture, etc. This feature extraction is useful for locating objects, finding their outlines and - before the arrival of Deep Learning - were used for classification.

Regions can be relatively easily obtained by subtracting the images of the scale space from each other. This is an operation called band-pass filtering and will be introduced in Section 4.1. Edges can be obtained by observing where a steep drop in the intensity landscape occurs, to be treated in Section 4.2. And for texture detection, the techniques are essentially a mixture of region and edge detection to be highlighted in Section 4.3.

4.1 Regions (Blobs, Dots) with Bandpass Filtering

Many regions are often either darker or brighter in comparison to their surround. One way to identify them would be by applying a threshold to the intensity values. That would be a technique that works well if the image consist of a homogenous background and foreground objects that stand out clearly from the background; we elaborate on that a bit later (Section 10). Here we use a technique that is slightly more flexible, but also more complicated. Specifically, we intend to exploit the scale space that we have developed previously. We subtract the images of the scale space S (Equation 3). That is for two images of S, one fine I_{fine} (e.g. $\sigma = 1$) and one coarse I_{coarse} (e.g. $\sigma = 2$), we take their difference

$$I_{\text{band}} = I_{\text{fine}} - I_{\text{coarse}} \tag{5}$$

in which case positive values correspond to brighter regions, and negative values correspond to darker regions. Figure 9 shows an example where those differences are taken between levels of the scale space.

As the fine and coarse image correspond to low-pass filtered images, their subtraction corresponds to the process of *band-pass filtering*. During band-pass filtering a certain 'center' range ('band') of values is selected, as opposed to the low-pass filtering for which the lower range of values is preferred. See also Appendix C for more explanations.

If the low-pass filtering occurred with Gaussian functions, that is g in Equation 2 is a Gaussian filter, then the corresponding band-pass filter is a difference-of-Gaussian filter, a DOG filter. It is perhaps the most common band-pass filter in image processing.

In Matlab we can generate this 'band space' by applying merely the difference operator to the scale space `SS`, in which case however the signs for brighter and darker regions are switched due to the implementation of `diff`:

```
%% =========    DOG     ==========
DOG         = diff(SS,1,3);     % brighter is negative, darker is positive
% --- set borders to 0:
for i=1:size(DOG,3), p=i*3;
    DOG(:,:,i) = padarray(DOG(p:end-p+1,p:end-p+1,i),[p p]-1);
end
%% =========    BRIGHTER/DARKER   ========
[BGT DRK]  = deal(DOG);    clear DOG; % clear DOG to save memory
BGT(BGT>0)  = 0;                % all positive values to 0
DRK(DRK<0) = 0;                % all negative values to 0
BGT        = -BGT;             % switch sign to make brighter positive
```

The band-pass space `DOG` contains one level less than the scale space. In this piece of code we also set the border values to zero for reason of simplicity. The variables `BGT` and `DRK` hold then only positive and negative values (left and right column in Figure 9). Of course one could also take other threshold values than zero to select perhaps only very bright or very dark regions.

Python provides functions to perform blob detection, that can be found in submodule `skimage.feature`, for example function `blob_dog`.

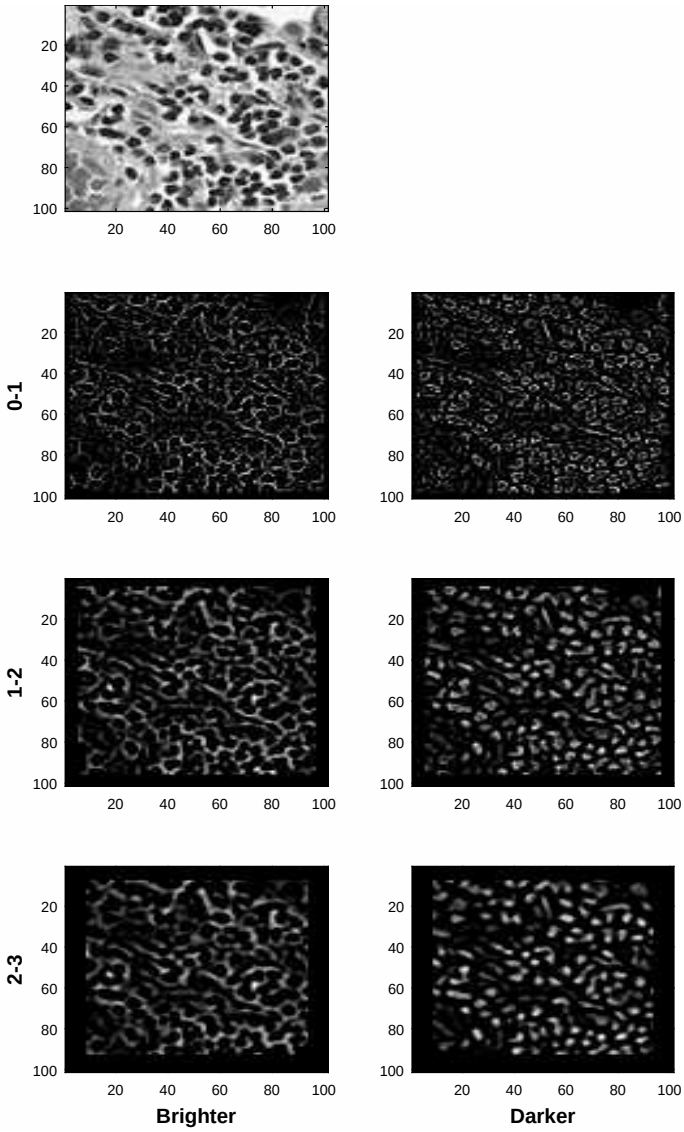

Figure 9: Example of a band-pass space, in this case a Difference-of-Gaussian (DOG) space: it is suitable for detecting regions (example code in L.4.1).
Left column represents brighter regions (`BGT` in code): negative values were set to zero.
Right column represents darker regions (`DRK` in code): positive values were set to zero and the absolute value of the negative values is displayed.

The second row labeled **0-1** is the subtraction of the input image and its low-pass filtered version. The third row labeled **1-2** is the subtraction of the corresponding two adjacent levels in the scale space; etc.

If one intended to select only the nuclei in this image of cell tissue, then the lower right image (2-3, darker) might be a good starting point and one would continue with morphological processing as will be presented in Section 10.

Now that we have regions, we are interested in manipulating them. For example we wish to eliminate small regions and would like to know the approximate shape of the remaining larger regions. We continue with that in Section 10.

Generating and dealing with the entire band-pass space is time consuming. If we develop a very specific task it might suffice to chose a few 'dedicated' scales. For instance to select nuclei from histological images that have image sizes of several thousands pixel (for either row or column), then generating the entire space might be too expensive. Rather one would design a bandpass filter whose size matches approximately the size of a typical nucleus. The original image would then be low-pass filtered with the corresponding two suitable sigmas, the resulting two filtered images subtracted from each other, and then a threshold is applied to the difference image.

4.2 Edge Detection

An edge is a steep drop in the intensity image, see again Figure 3 where we can clearly recognize huge cliffs. More precisely, an edge is an abrupt change in contrast with a certain orientation or direction; it is observed locally only, that is in a neighborhood of limited size. One could simply exploit the gradient image, see again 8, and apply a threshold to select the large gradients - the large arrows. Indeed, that represents

a first processing step in more sophisticated edge detectors. But thresholding the gradient image does not always find the precise location of the edge, as the edge could be very large and that would result in the detection of neighboring pixels as well.

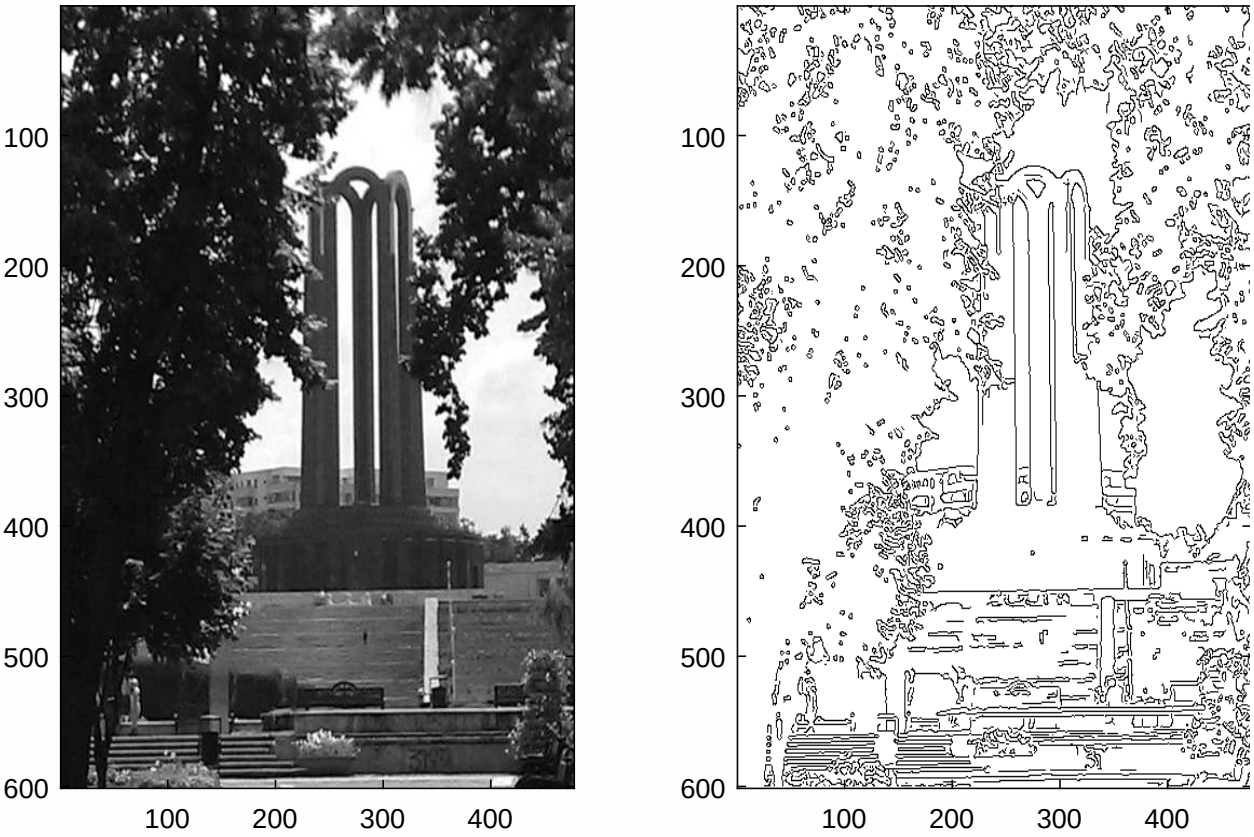

Figure 10: **Left**: input image. **Right**: binary map with on-pixels representing edges as detected by an edge-detection algorithm. There are different algorithms that can do that, the most elegant one is the Canny algorithm which is based on the gradient image as introduced in Section 3. An edge following algorithm traces the contours in such a binary map.

To find edges more precisely, the principal technique is to convolve the image with two-dimensional filters that emphasize such edges in the intensity landscape. The filter masks of such oriented filters therefore exhibit an 'orientation' in their spatial alignment. Here are two primitive examples for a vertical and diagonal orientation filter respectively.

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}. \tag{6}$$

Thus, the image is convolved multiple times with different orientation masks to detect all edge orientations, resulting in a corrsponding number of different output maps. Those maps are then combined to a single output map by a mere logical AND operation. More sophisticated techniques interpolate between those principal orientations.

Matlab provides the function `edge` to perform this process and it provides different techniques. Python offers pretty much the same set of techniques, but in different modules, `skimage.feature` and `skimage.filters`. Those functions return as output a binary map where on-pixels correspond to locations of pixels. The code example in Appendix **L.4.2** displays the output of different edge detection techniques.

- Robert detector: this detector is rather primitive and is 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.

- Prewitt and Sobel detector: those detectors find more edges, but at the price of more computation.
- Canny detector: this is most elaborate detection technique.

```
Medg = edge(I, 'canny', [], 1);
```

whereby the empty brackets [] stands for no particular choice of threshold parameters, and thus the threshold is determined automatically; the value 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.

Matlab `edge` (one function for all edge detectors)
Python `skimage.feature.canny`, `skimage.filters.roberts/sobel/prewitt`
Code Examps Appendix **L.4.2**

4.3 Texture

Dav p209, ch8
FoPo p194, pdf 164

Texture is observed in the structural patterns of object surfaces such as wood, grain, sand, grass, and cloth. But even scenes, that consists of many objects, can be regarded as texture.

The term texture generally refers to the 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 four broad categories:

Statistical: these methods are based on describing the statistics of individual pixel intensity values. We only marginally mention these methods (Section 4.3.1), as they have been outperformed by other methods.

Structural: in structural analysis, primitives are identified first, such as circles or squares, which then are grouped into more 'global' symmetries (see SHB pch 15 for discussion). We omit that approach in favor of better performing approaches.

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'.

Local Binary Patterns: turns statistical information into a code. It is perhaps the most successful texture description to date (Section 4.3.3). Fore some databases - in particular those containing textures - the description performs almost as well as Deep Nets. In comparison to Deep Nets, it has a smaller representation and a negligible learning duration.

4.3.1 Statistical

ThKo p412

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

$$P(v) = \frac{n_v}{n_{tot}} \tag{7}$$

with n_v the number of pixels with gray-level v and n_{tot} the total number of pixels in the region (`imhist` in Matlab for an entire image). Based on the histogram (equ. 7), quantities such as moments, entropy, etc. are defined. Matlab: `imhist, rangefilt, stdfilt, entropyfilt`

ThKo p414
Dav p213, s8.3
SHB p723, s15.1.2

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 (cor)relate these values. There exist different schemes to do that, the most used one is the gray-level cooccurrence matrix (`graycomatrix` in Matlab), see Dav p213, s8.3 SHB p723, s15.1.2 for more.

The use of the cooccurence 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.

4.3.2 Spectral [Structural]

FoPo p202
Pnc p277, s.13.1.5

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 (wiki Wavelet), which has found great use in image compression with the jpeg format. There are two reasons why wavelets are not optimal for texture representations. First, they are based on a so-called mother wavelet only, that is they are based on a single filter function. Second, they are useful for compression, but less so for classification. To represent texture, it is therefore more meaningful to generate more complex filters.

In the following we introduce a bank of filters that has been successful for texture description, see Figure 11 (see Appendix **L.4.3** for code). It is one specific bank of filters, namely the one by authors Leung and Malik. But other spectral filter banks look very similar. All these filters are essentially a mixture of the filtering processes we have introduced before. There are four principally different types of filters in that bank:

Filter no. 1-4: those filters are merely Gaussian functions for different sigmas as we used them to generate the scale space (Section 3.1).

Filter no. 5-12: those eight filters are bandpass filters as we mentioned them in Section 4.1. In this case the blob filter is much larger and is a Laplacian-of-Gaussian (LoG) filter.

Filter no. 13-30 (rows 3, 4 and 5): these are oriented filters that respond well to a step or edge in an image - it corresponds to edge detection as introduced above in Section 4.2. In this case, the first derivative of the Gaussian function is used.

Filter n. 31-48 (bottom three rows): those filters correspond to a bar filter. It is generated with the second derivative of the Gaussian.

To apply this filter bank, an image is convolved with each filter separately, each one returning a corresponding response map. In order to detect textons in this large output, one applies a quantization scheme as discussed in Section 8. Algorithm 1 is a summary of the filtering procedure.

1. 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.
2. In order to find all extrema, we rectify the maps obtaining so $2n$ maps R_j $(j = 1..2n)$.
3. 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.
4. Finally, we locate the maxima using an 'inhibiton-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 $\mathbf{x}_l$. With those we then build a dictionary as will be described in Section 8.

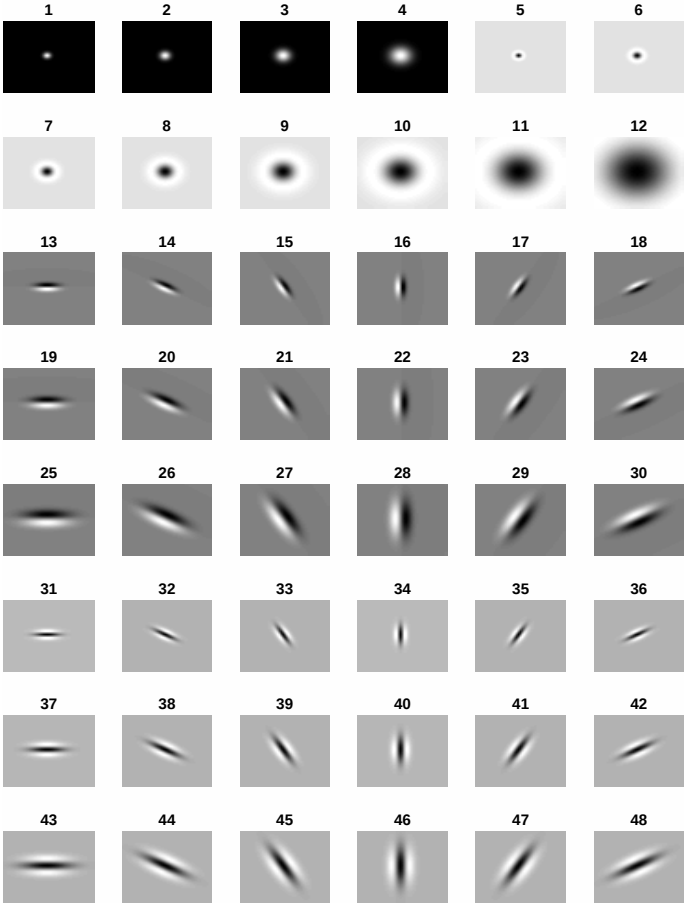

Figure 11: A bank of filters for texture analysis (the Leung-Malik filter bank): blob and orientation filters for different scales (see Appendix **L.4.3** for code). Filter patches are generated for a size of 49 x 49 pixels.
Filters 1-4: Gaussian function (low-pass filter) at four different scales (sigmas).
Filters 5-12: a blob filter - a LoG in this case - at eight different scales.
Filters 13-30 (rows 3, 4 and 5): an edge filter - first derivative of the Gaussian in this case - for six different orientations and three different scales.
Filter 31-48 (bottom three rows): a bar filter - generated analogously to the edge filter.

Algorithm 1 Local filtering for texture.

Input : image I, set of n filters F_i (see Figure 11) ($i = 1..10$)
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 $\min(0, -R_i)$, resulting in $2n$ maps R_j ($j = 1..2n$)
3) For each R_j, compute local summaries R_j^s 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 $\mathbf{x}_l(j)$ whose components j correspond to the filter response R_j at location (x_l, y_l)

4.3.3 Local Binary Patterns

wiki Local_binary_patterns

This texture descriptor observes the neibhouring pixels, converts them into a binary code and that code is transformed into a decimal number for further manipulation. Let us take the following 3x3 neighborhood. The center pixel - with value equal 5 here - is taken as a reference and compared to its eight neighboring pixels:

$$\begin{bmatrix} 3 & 4 & 6 \\ 1 & \mathbf{5} & 4 \\ 7 & 6 & 2 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 0 & 0 & 1 \\ 0 & - & 0 \\ 1 & 1 & 0 \end{bmatrix} \tag{8}$$

Neighboring values that are larger than 5, are set to 1, neighboring values that are smaller, are set to 0. This results in a 8-bit code, which in turn can be converted into a decimal code. This descriptor is constructed for each pixel of some window, sometimes even for the entire image. The descriptor numbers are then histogrammed: in case of the 8-bit code turned into a decimal code that would result in a 256-dimensional vector. That vector is the classified with traditional machine learning methods. There exist many variations of this texture descriptor.

Matlab `extractLBPFeatures`
Python `skimage.feature.local_binary_pattern`

5 Image Classification with Deep Neural Networks

Image classification is the process of assigning a scene or an object label to an entire image. One would think that this can be done with the feature extraction methods introduced in the previous section, and indeed there are innumerous attempts to do so. That approach is also called *feature engineering*. It has however lost its appeal - at least for the moment - with the arrival of the approach of *Deep Learning*, also called *feature learning*. With Deep Learning we let a so-called Deep Neural Network (DNN) find out the features that are necessary for getting the image classes discriminated. Not only are DNNs (or Deep Nets) more convenient - as we do not need to write elaborate feature extraction algorithms - but they also perform better, sometimes much better.

Deep Neural Networks are elaborations of traditional Artificial Neural Networks (ANNs), a methodology that exists already since decades. For a primer on ANNs we refer to Appendix **F.1**. In this section we immediately start with Deep Neural Networks. It most popular type is the so-called Convolutional Neural Network (CNN), which we introduce in Section 5.1. For this, we will switch to the programming language Python, as it has become the dominant language to explore neural network architectures. The big tech companies provide libraries to run such networks. Those libraries use similar terminology, but have slightly different ways to set up, initialize and run networks. One term that is often used is *tensor*, which is a function manipulating vectorial functions. In the context of Deep Nets, a tensor is often understood as an object representing batches of images, which makes the tensor three-dimensional or four-dimensional: two dimensions for the spatial dimension x and y, one dimension for the number of images in the batch, and another dimension if the image contains color information (typically 3: red, green, blue). A tensor can also be two- or even one-dimensional, in which case they are sometimes called simple tensors. Practically we do not need to learn novel algebra with this term, it is merely a different label for certain processes.

Some of the most popular Deep Net packages are:

PyTorch, https://pytorch.org/: This package is provided by Facebook. It is perhaps the most trending package. In particular *transfer learning* can be conveniently done with those libraries, coming up in Section 5.2.

Tensorflow, https://www.tensorflow.org/: This package is provided by Google. It is considered somewhat intricate to understand and to apply, and that is why a simpler API was developed for it, called **Keras**. We use Keras for introducing MLPs (Appendix **F.1**) and for introducing a simple CNN (Section 5.1 next).

Caffe, http://caffe.berkeleyvision.org/, https://github.com/BVLC: Provided by UC Berkely. It was one of the early packages made public and has lost in appeal after the emergence of PyTorch and Tensorflow, but is still a reference. Many implementations of tasks are based on it.

Darknet, https://pjreddie.com/darknet: A package suitable for embedded systems, because it has a smaller memory footprint. It runs a very efficient object detection system.

FastAI, https://www.fast.ai/: A package that promises to deliver state-of-the art classification accuracies. It uses in particular ensemble of Deep Nets, which provide better results than individual Deep Nets. It is based on PyTorch.

For other packages we refer to the wiki pages: wiki Comparison_of_deep_learning_software.

A fundamental downside of Deep Learning is that it takes very long to train a Deep Net. One solution to the problem is to apply massive computing power, available however mostly to big tech companies and universities. Another way to mitigate the downside is to exploit the so-called CUDA cores of the graphic cards in a PC. Software packages (PyTorch, Tensorflow) provide routines to exploit the available CUDA cores of a computer. A third way to counteract the problem is to use the trick of *transfer learning*. The latter two tricks will be addressed in particular in Section 5.2.

5.1 A Convolutional Neural Network (CNN)

wiki Convolutional_neural_network

Roughly speaking, a Convolutional Neural Network (CNN) can be regarded as an elaboration of a MLP (Section **F.1**). It is elaborated by the following two principal tricks in particular, see also Figure 12:

1. CNNs make actual use of the two-dimensionality of images, namely by applying quasi-convolutions to find discriminative, local features, similar to the feature extraction process discussed in Section 4 (Section 4.3.2 in particular). The difference is that in a 'traditional' convolution there is only a single kernel mask (K in equation 40), but in a CNN there are many individually learned kernel masks K_i, one for each local neighborhood i. The corresponding layers are also called *feature layers* or *feature maps*.

2. CNNs subsample the feature layers to so-called *pooling layers* akin to building a spatial pyramid discussed in Section 3.1.

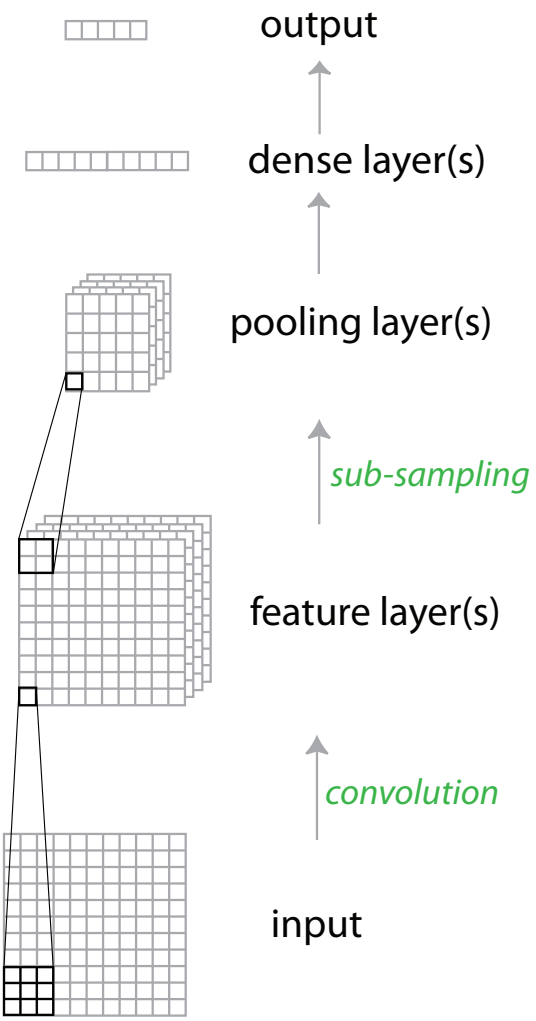

Figure 12: The typical (simplified) architecture of a Convolutional Neural Network (CNN) used for classifying images. It is essentially an MLP elaborated by convolution and subsampling processes as introduced in the previous sections. The architecture has in particular several alternating feature and pooling layers. The term 'hidden' layer is not really used anymore in a CNN, as there are so many different types of hidden layers, that more specific names are needed.

The **feature layer(s)** is sometimes also called *feature map(s)*: it is the result of a convolution of the image, however not one with a single, fixed kernel, but one with many individual, learned kernels: each unit observes a local neighborhood in the input image and learns the appropriate weights.

The **pooling layer** is merely a lower resolution of the feature layer and is obtained by the process of sub-sampling. This sub-sampling helps to arrive at a more global 'percept'.

In a typical CNN, there are several alternations between feature and pooling layer. The learning process tries to find optimal convolution kernels K_i for each neighborhood i that help to separate the image classes.

Between the last pooling layer and the output layer, there lies typically a dense layer: it is a flat (linear) layer, all-to-all connected with its previous pooling layer and its subsequent output layer, again as a complete bipartite graph. This is sufficient introduction for the moment and we now start looking at some code as written in Keras. The code is merely an extension of the MLP code example of Appendix **F.1**, extended by convolution and pooling layers:

```
# https://github.com/fchollet/keras/blob/master/examples/mnist_cnn.py
# Trains a simple convnet on the MNIST dataset.
# Achieves 99.25% test accuracy after 12 epochs
from __future__ import print_function
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from LoadMNIST import LoadMNIST
batchSz     = 128
nEpoch      = 12

#%% -----   Load Database & Labels  ------
Isz         = (28, 28, 1)
TREN, TEST, Lbl = LoadMNIST()    # load the digits
# extending by one dimension
TREN    = TREN.reshape(60000, Isz[0], Isz[1], 1) # [60000 28 28 1]
TEST    = TEST.reshape(10000, Isz[0], Isz[1], 1) # [10000 28 28 1]

#%% =====   Build Network     =======
N       = Sequential()
N.add( Conv2D(32, kernel_size = (3,3), activation = 'relu', input_shape = Isz))
N.add( Conv2D(64, (3,3), activation = 'relu') )
N.add( MaxPooling2D( pool_size = (2,2) ) )
N.add( Dropout(0.25) )
N.add( Flatten() )
N.add( Dense(128, activation = 'relu') )
N.add( Dropout(0.5) )
N.add( Dense(10, activation = 'softmax') )
N.summary()
N.compile( loss      = keras.losses.categorical_crossentropy,
           optimizer = keras.optimizers.Adadelta(),
           metrics   = ['accuracy'])

#%% =====   Learning     =======
N.fit(TREN, Lbl.TrenMx,
      batch_size = batchSz,
      epochs     = nEpoch,
      verbose    = 1,
      validation_data = (TEST, Lbl.TestMx) )

#%% =====   Evaluation     =======
score = N.evaluate(TEST, Lbl.TestMx, verbose=0)
print('Test loss:    ', score[0])
print('Test accuracy:', score[1])
```

A typical CNN has many alternations between feature and pooling layers: for small tasks CNNs frequently have 3 or more such alternations, resulting in a 8-layer network or larger networks; for large tasks, CNNs can consist of several hundreds of layers.

5.2 Transfer Learning (PyTorch)

So far we have played with an image collection, whose image sizes were merely 28x28 pixels. But now we intend to classify images of larger size and that increases learning duration significantly, because we convolve larger arrays. To train a full network, it can take weeks. Fortunately, there exists a trick called *transfer learning*, which allows us to re-use a fully trained network for other tasks. That seems to work because the lower layers of a fully trained network appear to be general feature extractors. In transfer learning, one takes such a network, cuts off its top weight layer, and replaces it with a weight layer for the new tasks that one intends to solve. The fully trained networks are then called *pre-trained* networks in this context, see Appendix F.3 for some examples. A pre-trained network is typically trained with thousands or even millions of images for hundreds of different classes, mostly general categories such as dogs, cars, lamps, flowers, etc.; an often-used collection is the *ImageNet* data set.

We introduce how to exploit transfer learning in PyTorch, because that is (presently) the most convenient

package. PyTorch downloads those networks automatically and also provides routines for data preparation, in particular for data augmentation.

There are two modes of exploiting a pre-trained network. In one mode, we merely exchange the top weight layer and re-train only that top-layer with our images at hand. This is called fixed feature extraction, because the rest of the network is not re-trained anymore; one would use the rest of the network *as is*. That would be the quickest and simplest way of doing transfer learning and will be introduced in Section 5.2.1.

In another mode, we go a step further by re-training the whole network, in addition to retraining the top weight layer. This takes more time, but we gain a bit of prediction accuracy. This is also called *fine-tuning* a network and requires only few changes to the code of fixed feature extraction (Section 5.2.2).

What follows first are explanations for the code of either mode. A full example is given in Appendix **L.6.3**. Here we explain excerpts of it.

One specifies a desired pre-trained model with a single line, in this case we select the 'resnet18' model:

```
from torchvision import models
MOD         = models.resnet18(pretrained=True)
```

There exist other models, e.g. alexnet, densenet, inception, etc. (Appendix F.3). Often they come as a range of versions learned on different resolutions. For instance 'resnet34' provides higher resolution, but it also takes longer to train on it.

Now we perform *data augmentation*, a process to artificially enrich the labeled data set, see also Appendix H. This carried out with the module `transforms`. The various types of transforms can be lumped together to a single object, called `AUGNtrain` in the following code snippet:

```
from torchvision import transforms
AUGNtrain    = transforms.Compose([
        transforms.RandomResizedCrop(szImgTarg),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(ImgMean, ImgStdv)  ])
```

Furthermore, PyTorch provides a routine `ImageFolder` in module `torchvision.datasets` to extract images and class labels from your collected data set automatically. You need to pass only the path of the folder that contains your classes, labeled accordingly, ie. all dog images placed into a folder called 'dog'. The images can be of varying size. A routine called `DataLoader` in module `torch.utils.data` will automatically prepare the data set.

```
FOLDStrain  = torchvision.datasets.ImageFolder(dirImagesTrain, AUGNtrain)
LOADERtrain = torch.utils.data.DataLoader(FOLDStrain, batch_size=szBtch, shuffle=True)
```

5.2.1 Fixed Feature Extraction

As introduced already, the fastest way to arrive at some results is to use the pre-trained network as is - without making further adjustments to it. To do that we proceed with two steps. One is to freeze the layers by preventing it to calculate gradients during the learning process: `requires_grad = False`. The second step is to replace the top layer with our problem set, for instance for a 5-class problem we need to insert a final weight layer that connects from the last dense layer to the final 5-unit output layer; function `nn.Linear` does that (`nClss` is the number of classes):

```
# freezing gradient calculation
for param in MOD.parameters():
    param.requires_grad = False
# replaces last fully-connected layer with new random weights:
MOD.fc  = nn.Linear(MOD.fc.in_features, nClss)
```

Now we initialize the training parameters. With the function `SGD` we select an optimization procedure called *stochastic gradients*. Note that we specify the training only for the final layer by selecting `.fc`.

By calling `lr_scheduler.StepLR` we specify more learning parameters. Then we choose a loss function, a function that calculates the classification accuracy.

```
# only parameters of final layer are being optimized
optim   = SGD(MOD.fc.parameters(), lr=lernRate, momentum=momFact)

# Decrease learning rate every szStep epochs by a factor of gamma
sched   = lr_scheduler.StepLR(optim, step_size=szStep, gamma=gam)

crit    = nn.CrossEntropyLoss()
```

5.2.2 Fine Tuning

To adjust the weights of the entire networks, we modify the steps of the previous section in two ways. One is, we refrain from freezing the parameters, so we drop the `for param` loop. Another step, is to omit the attribute `.fc`, which in turn instructs that the weights of the entire net are being re-learned:

```
# .fc is omitted - as opposed to above:
optim   = SGD(MOD.parameters(), lr=lernRate, momentum=momFact)
```

Adjusting the parameters of the entire net will take more time, but we gain classification accuracy. Other than that we can use the same code example as in **L.6.3**.

6 Object Detection, Object Recognition/Localization

FoPo p549, ch 17

Object *detection* is the localization and count of a specific object category in a scene. For example, we would like to determine how many faces there are in a group photo; or how many pedestrians there are in a street scene. It is a two-class classifier discriminating between target and non-targets, or object and non-object; the non-targets or non-objects are also called distracters sometimes.

Object *localization* is also called generic or general object recognition sometimes; it is the process of object detection for multiple classes. It is a multi-class classifier discriminating between a set of target objects and the distractor. Obviously, for such multi-class systems, the overall complexity multiplies with the number of classes one would like to discriminate.

Such systems - whether mere two-class detection or generic multi-class detection - consist of two processes, a search algorithm and a classification process. The classification process is essentially the same as introduced previously for image classification. The search algorithm scans the image for potential object candidates. The challenge is to make that search algorithm as efficient as possible, because scanning the entire image pixel by pixel would be the most accurate but also the costliest approach; one therefore deals with a speed-accuracy tradeoff, whose terminology will be introduced in Section 6.2. But before that, we start with face detection, for which there exist a particularly efficient search algorithm (Section 6.1). Then we introduce a popular pedestrian detector (Section 6.3). Finally we mention a DeepNet that performs very efficient search for object recognition (Section 6.4).

6.1 Face Detection

Sze p578, s14.1.1, pdf658

FoPo p550, s17.1.1, pdf520

Face detection is ubiquitous nowadays: it is run in most of today's digital cameras to enhance auto-focus; on social-media sites to tag persons; in Google street view to blur persons, etc. There exist many algorithms each one with advantages and disadvantages. A good face detection system combines different algorithms, but most of them will run the Viola-Jones algorithm (Section 6.1.2). But first we mention general tricks used for training a face detection system.

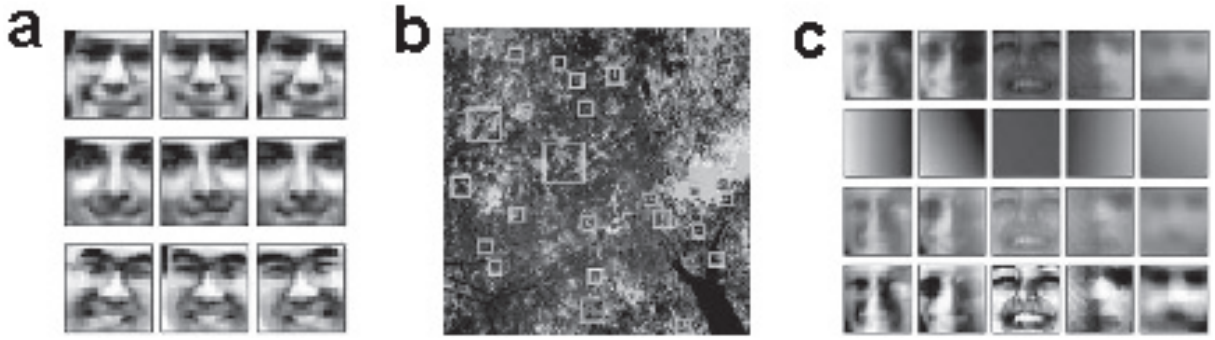

Figure 13: Training a face detector (Rowley, Baluja, and Kanade 1998a):
a) Data augmentation: artificially mirroring, rotating, scaling, and translating training images to generate a training set with including larger variability.
b) Hard negative mining: using images without faces (looking up at a tree) to generate non-face examples.
c) Image enhancment: pre-processing the patches by subtracting a best fit linear function (constant gradient) and histogram equalizing.
[Source: Szeliski 2011; Fig 14.3]

6.1.1 Optimizing Face/Non-Face Discrimination

A typical face detection system uses the following tricks to improve performance. For the first two tricks see also Appendix H.

a) Hard Negative Mining: non-face images are collected from aerial images or vegetation for instance (Figure 13b).

b) Data Augmentation: the set of collected face images is expanded artificially by mirroring, rotating, scaling, and translating the images by small amounts to make the face detectors less sensitive to such effects (Figure 13a).

c) Image Enhancement: after an initial set of training images has been collected, some optional preprocessing 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 (Fig. 13c), see again Section 2.2.

6.1.2 Viola-Jones Algorithm

The most frequently used face detection algorithm is probably the one by Viola and Jones. It uses features consisting of two to four rectangular patches of different polarity, see upper row of figure 14. The pixels inside the white rectangles are subtracted from the pixels inside the black pixels. Computing the values for one rectangle can be done extremely efficiently with the integral image (Section 6.1.3). 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 rectangle intensities. This feature selection can be done with a 'boosting' classifier. The two most significant features are shown in Figure 14; 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.

Figure 14: 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. Python code example in **L.8.1** (accessing OpenCV). [Source: Szeliski 2011; Fig 14.6]

In Python can the algorithm be applied through OpenCV. It comes in several variants and one specifies the desired variant with function `CascadeClassifier`, here the default variant `haarcascade_frontalface_default.xml`. It is run with the method `detectMultiScale`:

```
FaceDet = cv2.CascadeClassifier(cascPth + 'haarcascade_frontalface_default.xml')
aFaces  = FaceDet.detectMultiScale(Igry, 1.3, 5)
```

A full code example is given in **L.8.1**.

Other Applications Face detectors are built 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.

Further Leads

`http://vis-www.cs.umass.edu/lfw/` Database with 13k faces.
`https://facedetection.com/`
`https://www.mathworks.com/matlabcentral/fileexchange/39627-cascade-trainer-specify-ground-truth-train-a-detecto`

6.1.3 Rectangles

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

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

$$I_s(i,j) = \sum_{k=0}^{i} \sum_{l=0}^{j} I_o(k,l). \tag{9}$$

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:

$$R_s(i_0..i_1, j_0..j_1) = I_s(i_1, j_1) - I_s(i_1, j_0) - I_s(i_0, j_1) + I_s(i_0, j_0) \tag{10}$$

Matlab

```
Is = cumsum(cumsum(Io,1),2);                % integral image (same size as original image)
Rs = Is(i1,j1)-Is(i1,j0)-Is(i0,j1)+Is(i0,j0); % summed intensity values for rectangular patch (scalar)
```

Python skimage.transform.integral_image

6.2 Sliding Window Technique

The term window stands for a rectangular subset of the image, a large neighborhood in some sense. This window is moved across the image at selected column and row positions, a search called *sliding*. Together that forms the expression *sliding window* technique. The selected column and row positions are typically taken from a grid.

After we have trained a classifier, we pass $n \times m$ windows of a new (testing) image to the classifier; the window is moved along the grid, by a step size of few pixels e.g. Δx and Δy=3 pixels. There are three 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 (Section 3.1): 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 2 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 Δx and Δy
1) Construct an image pyramid.
2) For each level of the pyramid:
- apply the classifier to each $n \times m$ window (moving by Δx and Δy) and obtain strength c.
- if $c > t$, then add window to a list $\mathcal{L}$ including response value c.
3) Rank list $\mathcal{L}$ in decreasing order of c values $\rightarrow \mathcal{L}^{seq}$.
4) For each window $\mathcal{W}$ in sequence $\mathcal{L}^{seq}$ (starting with maximal c):
- remove all windows $\mathcal{U} \neq \mathcal{W}$ that overlap $\mathcal{W}$ significantly where the overlap is computed in the original image by expanding windows in coarser scales $\rightarrow \mathcal{L}^{red}$.
$\mathcal{L}^{red}$ 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.

In software packages, there are some functions to facilitate the search processe:

In Matlab this can be found in particular under 'Neighborhood and Block Operations' in the image processing toolbox. In particular function `blockproc` and `nlfilter` are useful here.

In Python one would have to look into module `numpy.lib.stride_tricks`, but that does not appear to be a much pursued direction.

6.3 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 used histograms of gradients (HOG) descriptors, taken from a regular grid of overlapping windows (Fig. 15). 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 15 left shows a sample input image, while Figure 15 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 15 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).

Matlab `extractHOGfeatures`
Python `skimage.feature.hog`

Applications Needless to say, that pedestrian detectors can be used in automotive safety applications. Matlab even has a code example for pedestrian detection.

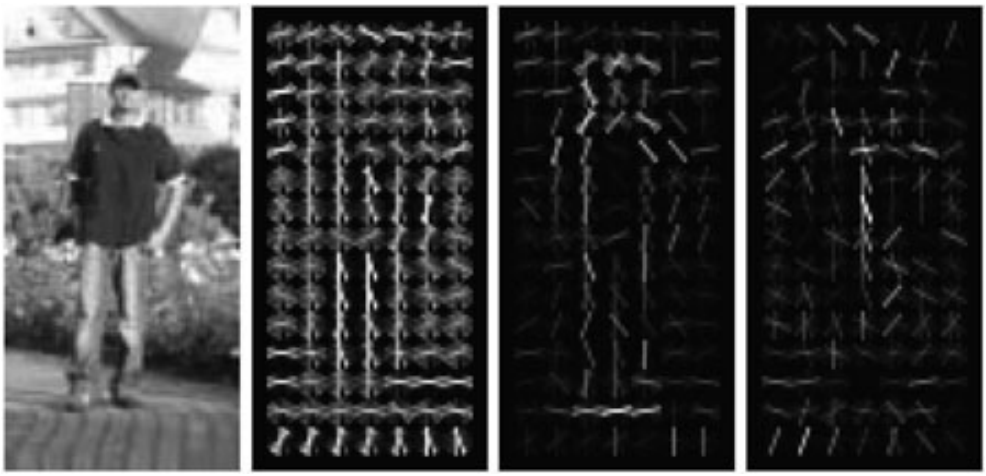

Figure 15: **Left**: typical pedestrian window.
Center left: HOG descriptor. Each of the orientation buckets in each window is a feature, and so has a corresponding weight in the linear SVM.
Center right: HOG descriptor weighted by positive weights, then visualized (so that an important feature is light). 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). Python code example in **L.8.2** (accessing OpenCV).
[Source: Forsyth/Ponce 2010; Fig 17.7]

6.4 Object Localization/Recognition

Deep Nets are - not surprisingly - the most succesful multi-class object detection systems. Many of them use the sliding window technique as introduced above. A recent DeepNet however manages to find objects by an extremely efficient search: the classifier is applied only to an array of cells placed at a very coarse grid at multiple resolutions. Run on the appropriate hardware with CUDA cores, the system can classify at a rate of several images per second. The network's name is YOLO: you only look once. It is available from the same provider of the DeepLearning software called DarkNet: https://pjreddie.com/darknet/yolo/.

To apply the network for probing purposes, one needs to download only three files:

- coco.names: the class labels. The network was trained on 80 classes.
- yolov3.cfg: a text file specifying the network architecture.
- yolov3.weights: the network weights. ca. 230 MB.

We can run the network with OpenCV access through Python using the DeepNet module dnn. We firstly load the network:

```
NET         = cv.dnn.readNetFromDarknet(fileConfig, fileWeights)
```

An image is transformed into a blob image as follows:

```
Iblob    = cv.dnn.blobFromImage(Irgb, 1/255, (wthInput, hgtInput), \
                        [0,0,0], 1, crop=False) # [1 3 hgtInp wthInp]
```

Then we send the image through the network in Python itself with functions as introduced under image classification. The output of the network are three sets of arrays representing the class confidence values of every cell. One then performs a non-maxima suppression to find the most promising candidates:

```
IxSel   = cv.dnn.NMSBoxes(aCanBbox, aCanConf, thrConf, thrNonMaxSupp)
```

The full code is available in Appendix **L.8.3**.

7 Feature Extraction II: Patches and Transformations

Sze p181, ch4, p205

Dav p149, ch6

We turn toward a feature that is based on an image *patch* - it is the basis for the feature engineering approach as depicted in Fig. 1, left half. A patch is typically a small, square-sized pixel array of the image. Patches are taken in particular at points where there appears to be a corner or other 'interesting' structure (Figure 16). That process is called *detection*. The detected patch is then transformed, a process called *extraction* and its result is then a vector called *descriptor*. The most frequent transformation is the generation of a histogram of the local gradients of the patch. Such transformations make the patch relatively unique and distinct and thus suitable for matching between images; they are therefore also called *keypoint features*, *interest points* and if they express angles in particular, then they are also called *corners*.

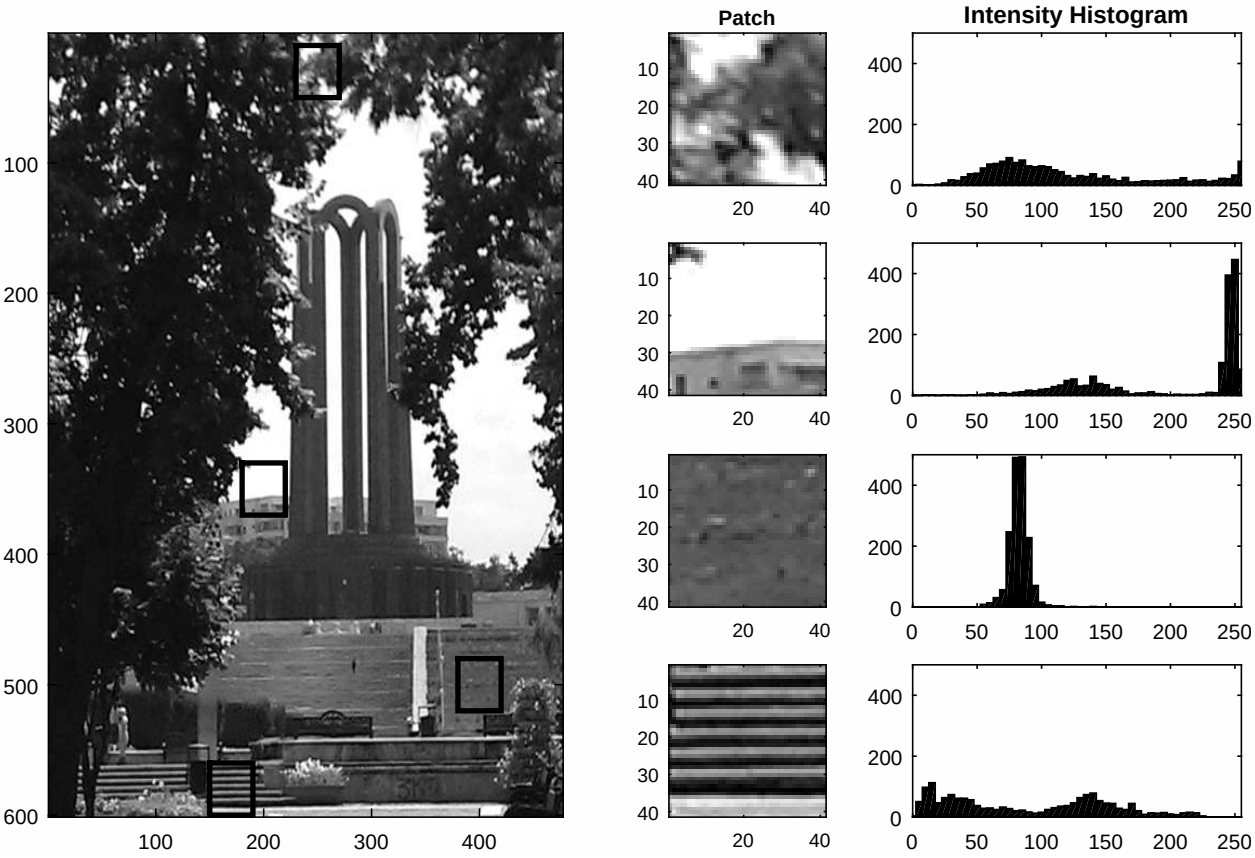

Figure 16: Examples of image patches and their corresponding description, in this case a mere intensity histogram.
Center column: four patches detected in the picture. In this case the patches were selected randomly to illustrate the potential information they can offer.
Right column: histogram of intensity values for each patch. This extraction and description process is rather simple but shown for illustration. In this section, more complex transformations will be introduced, for example based on the intensity gradients as introduced in Section 3.3.

Patches are applied in feature-based correspondence techniques such as stereo matching, image stitching, fully automated 3D modeling, object instance detection 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. Patches used to be employed also for image classification, but were meanwhile surpassed by Deep Neural Networks.

Summarizing, 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 extraction (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.

In Matlab, functions carrying out these processes start with a corresponding keyword, e.g. `detectFeatures`, `extractFeatures` and `matchFeatures`. In Python, those processes are found in `skimage.feature`, ie. `corner_xxx` and `match_descriptors`.

7.1 Detection

Sze p185
FoPo p179, pdf 149
Dav p158, s6.5 s6.7
SHB p156, s5.3.10
Pnc p281, s13.2.2

One way to attempt to find corners is to find edges - as introduced in Section 4.2 - , and then along 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:

$$\mathcal{H} = \sum_{window} \{(\nabla I)(\nabla I)^T\} \tag{11}$$

$$\approx \sum_{window} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \tag{12}$$

whereby $I_x = I_o * \frac{\partial g}{\partial x}$ and $I_y = I_o * \frac{\partial g}{\partial y}$ (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

$$C = \det(\mathcal{H}) - k\Big(\frac{\mathrm{trace}(\mathcal{H})}{2}\Big)^2 \tag{13}$$

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 $(\mathrm{trace}(\mathcal{H})/2)^2$). 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 3 Feature detection. Sze p190, pdf 214

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 13.
5. Find local maxima above a certain threshold and report them as detected feature point locations.

Here is how a simple feature detector would look like:

```
%% ----------- 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' (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:

```
%% ----------- 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]
```

In Matlab the Harris detector is available with the function `detectHarrisFeatures`,

```
FHar    = detectHarrisFeatures(I);
```

where `FHar` is a structure that contains the locations and the significance of the detected features. See Appendix **L.7.1** for an example. An example of the detection output is shown in Fig. 17, along with the output of three other feature detectors. As one can see there are substantial differences in location as well as in detection count. Both, location and count, depend on the default parameters settings, but the differences between algorithms remain.

Feature Tracking (in Scale Space): 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.

7.2 Extraction and Description

FoPo p187, s5.4.1, pdf157
Dav p173, s6.7.3
Pnc p284, s13.3.2

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

a) take the gradient (0-360 deg) at each pixel (from ∇I) in a 16×16 window around the detected keypoint (Section 3.3), using the appropriate level of the Gaussian pyramid at which the keypoint was detected.

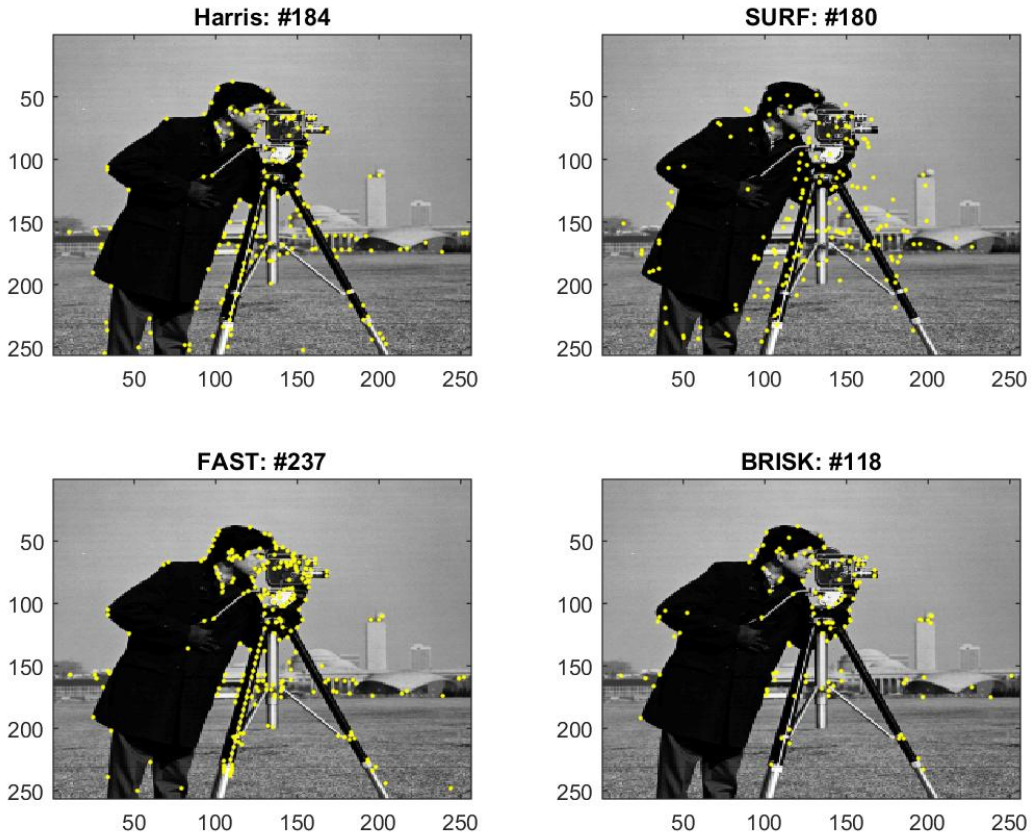

Figure 17: Output of various feature detector algorithms. Code in Appendix **L.7.1**

b) the gradient magnitudes are downweighted by a Gaussian fall-off function (shown as a circle in Figure 18) in order to reduce the influence of gradients far from the center, as these are more affected by small misregistrations.

c) in each 4×4 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 $\mathbf{v}$ 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: $\mathbf{u} = \mathbf{v}/\sqrt{\mathbf{v} \cdot \mathbf{v}}$.

e) to further make the descriptor robust to other photometric variations, values are clipped to $t = 0.2$: form $\mathbf{w}$ whose i'th element w_i is $\min(u_i, t)$. The resulting vector is once again renormalized to unit length: $\mathbf{d} = \mathbf{w}/\sqrt{\mathbf{w} \cdot \mathbf{w}}$.

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

```
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
```

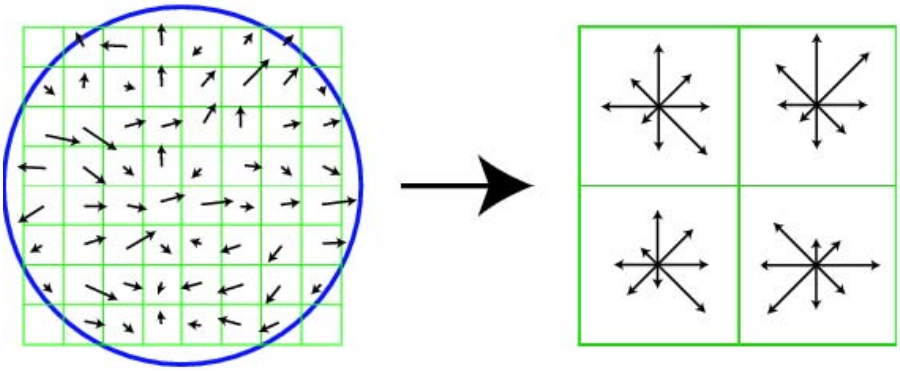

Figure 18: Forming SIFT features.
Left: Gaussian weighting, shown for only a 8x8 field in this illustration.
Right: formation of histograms demonstrated on 2x2 quadrants.
[Source: Szeliski 2011; Fig 4.18]

```
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]);
    Gh  = [Gh;  HDw];
end
```

In Matlab one uses the function `extractFeatures` and feeds both the image and the detected points as parameters to it:

```
Dhar    = extractFeatures(I, FHar);
```

The output variable `Dhar` is a structure containing the selected features and their descriptions, namely vectors.

7.3 Matching

To compare two two descriptor lists (originating from two different images for instance), $\mathbf{d}_i$ and $\mathbf{d}_j$ ($i = 1,..k, j = 1,..l$), 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 8). If we intend to establish correspondences for registration (section 16), we want to find the mutual matches.

```
L1, L2: the 2 descriptor lists, [nD1 x nDim] and [nD2 x nDim]
% ----- Compact version:
DM = pdist2(L1,L2);
% ----- Explicit version: (building DM ourselves)
DM  = zeros(nD1,nD2);
for i = 1 : nD1
   iL1rep  = repmat(L1(i,:), nD2, 1);         % replicate individual vector of L1 to size of L2
```

```
   Di      = sqrt(sum((iL1rep - L2).^2,2)); % Euclidean distance
   DM(i,:) = Di;                            % assign to distance matrix
end
% ----- Correspondence with respect to one list
[Mx2 Ix1] = min(DM,[],1);      % [1 x nD2]
[Mx1 Ix2] = min(DM,[],2);      % [nD1 x 1]
% ----- Correspondence mutual
IxP1to2 = [(1:nD1)' Ix2];      % [nD1 x 2] pairs with indices of 1st list and minima of 2nd list
IxP2to1 = [(1:nD2)' Ix1'];     % [nD2 x 2] pairs with indices of 2nd list and minima of 1st list
bMut1   = ismember(IxP1to2, IxP2to1(:,[2 1]), 'rows'); % binary array of mutual matches in list 1
IxMut1  = find(bMut1);         % mutually corresponding pairs with indexing to list 1
IxPMut1 = IxP1to2(IxMut1,:);   % [nMut1 x 2] mutal 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).

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. In Matlab that is implemented with function `matchFeatures`.

7.4 Summarizing

Now that we have seen feature detection, feature extraction and feature matching, we can find corresponding matches between two (similar) images. Appendix **L.7.2** gives a full code example. This type of establishing correspondence is the starting point for many tasks and was the leading approach to recognition before Deep Neural Networks toke over the scene, see again Fig. 1.

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/

8 Feature Quantization

FoPo p203, s6.2, pdf 164

Sze p612, s14.4.1, pdf 718

We now move towards representations for objects and scenes using the patches as obtained in the previous Section 7. For some time, that was the most successful method for scene classification system, but has lost its appeal since Deep Neural Networks took over. Nevertheless, it is useful to understand the method, because it has its advantages too: it does not need as many training samples as a DNN and it learns much, much quicker.

A generic way to exploit such patches 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 19 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.

8.1 Building a Dictionary

We quantize features as follows:

1. Collect patches $\mathbf{x}_i(d)$ 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 $\mathbf{x}_i^r(d)$ with $d = 1..n_r$.
3. Quantize vectors with a clustering technique and obtain a list of clusters $\mathbf{c}_j(d)$ 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 4 Building a dictionary for a category with $\mathbf{x}_i \in \mathcal{D}^L$. Compare with upper half of Figure 19.

1) Collect many training patches $\mathbf{x}_i(d)$	$(i = 1..n_p; d = 1..n_d)$
2) Optional: apply the PCA: $\mathbf{x}_i(d) \rightarrow \mathbf{x}_i^r(d)$	$(d = 1..n_r)$
3) Find k (n_c) cluster centers $\mathbf{c}_j(d)$	$(j = 1..n_c; d = 1..n_r)$

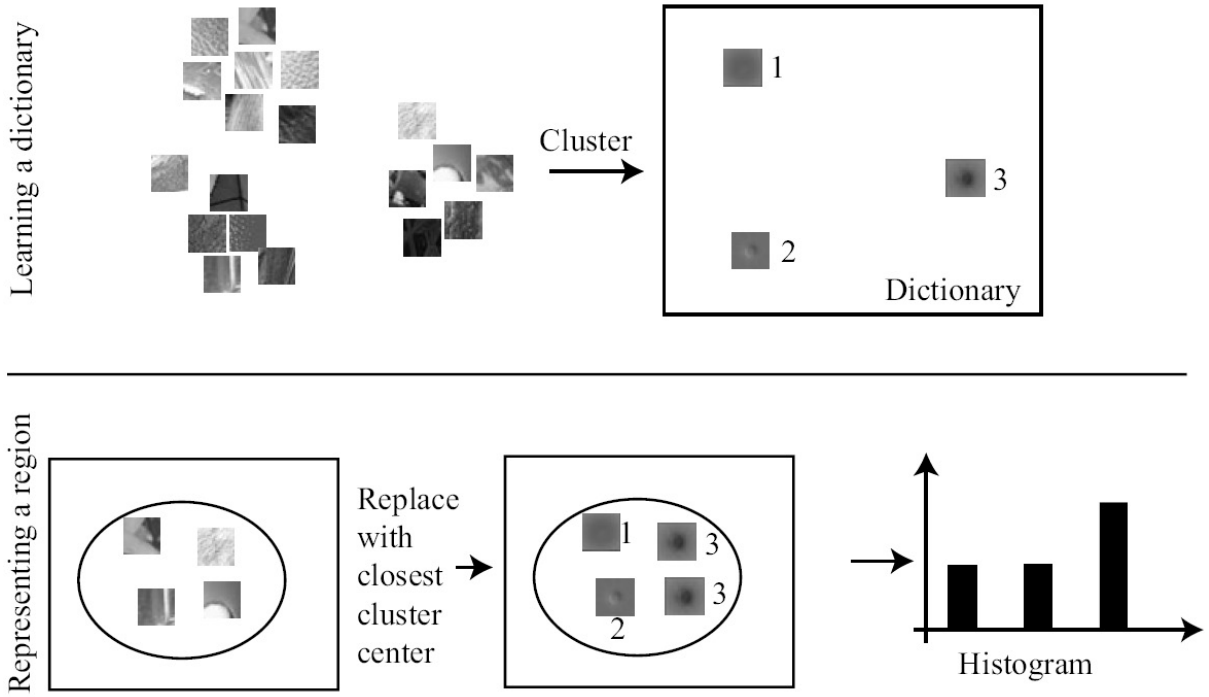

Figure 19: Using image patches for classification.
Upper part Learning a dictionary: 3 clusters were found in the training set (using some clustering technique), whose average values are shown as 'Dictionary'.
Lower part Representing a region: describing an object or scene. For a given image, replace detected texture patches with the closest cluster label and create a histogram. [Source: Forsyth/Ponce 2010; Fig 6.8]

8.1.1 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 Pnc p291, s.13.4.4.

To apply this technique in Matlab we organize our patches (features) in a `nF`×`nD` matrix `FTS`. We then cluster with the command `kmeans`: `Ixs = kmeans(FTS, k)`, whereby `Ixs` is a vector of length `nF` containing the cluster indices ($\in 1..k[n_c]$). See also Appendix G for more details. There are different ways how to

calculate the clusters during evolvement, see the options of `kmeans`.

8.2 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: $\mathbf{v}_m(d)$ with $m = 1..n_q$. Apply the PCA (if it was used before): $\rightarrow v_m^r(d)$ with $d = 1..n_r$.
2. For each patch $\mathbf{v}_m$ (or $\mathbf{v}_m^r$) find the nearest cluster center $\mathbf{c}_j$ $(j = 1..n_c)$ - 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 5 Applying the dictionary (for one category), $\mathbf{x}_i \in \mathcal{D}^T$. Compare with lower half of Figure 19.

1) For each relevant pixel m in the image: compute the vector representation $\mathbf{v}_m$ around that pixel
2) Obtain j, the index of the cluster center $\mathbf{c}_j$ closest to that feature
3) Add a value of one to the corresponding bin the histogram $H(j)$.

8.3 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 G 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 4), the other 20 are used for generating histograms (algorithm 5). The actual classifier is then trained with those 20 histogram vectors and tested on the 5 training histograms, which were also generated with algorithm 5. 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.

9 Segmentation (Image Processing II)

SHB p176, ch6
FoPo p285, ch9, pdf255
Sze p235, ch5,pdf267

Image segmentation is the task of partitioning a scene into its constituent regions. In the early days of computer vision, the process was once considered to be an essential, early step in a systematic and deterministic reconstruction of a scene, see historical note in Section 1.4.1 again. This is sometimes referred to as *semantic* or *general* segmentation and is still pursued in particular in Content-Based Image Retrieval (CBIR) and video analysis, where one is often faced with such a large number of scenes. But in general, image segmentation nowadays is pursued for more specific tasks. If the task is to separate an object in the center of the image from its background, then this is also called *figure-ground segregation*. If there are multiple objects in the scene, one speaks rather of *foreground-background* segmentation.

Segmentation is normally carried out completely automatically, in which case one speaks of *unsupervised* segmentation (as in 'unsupervised learning', a term from the field of machine learning). But sometimes segmentation is also initiated by a user in an interactive system: that of course generates more precise segmentation results. In video analysis, segmentation is often considered the task of segregating the moving objects from the stationary background.

Functionally speaking, segmentation is often defined as a search for groups of pixels of a certain coherence. This loose definition makes certainly sense if regions are homogeneous, but regions themselves can also show certain patterns sometimes, which makes that definition already a bit fuzzy.

In the following the segmentation methods are introduced ordered along their degree of complexity. Simple segmentation processes carry out mere thresholding of images, which does not always provide satisfactory results, but that is completed quickly and is therefore suitable for applications with time constraints, such as video processing (Section 9.1). Then there exist segmentation processes that regard the image as a landscape (Section 9.2). Then there is a set of useful clustering algorithms, which essentially are statistical methods (Section 9.3). Finally, there are DeepNets methods that carry out segmentation very accurately, after one has trained the network for a specific class of scenes (Section 9.4).

9.1 Thresholding (Histogramming)

SHB p177, s6.1
Dav p82, ch4, pdf119

Segmentation by thresholding is often also called histogram segmentation, because one typically looks at the intensity distribution of the individual image pixels by generating a histogram, see for instance the histogram in the upper right of Fig. 20 (see also SHB p24, s.2.3.2). In many scenes, the modes (maxima) of such a histogram correspond to objects and backgrounds and one therefore searches for optimal thresholds by analyzing those modes. As the typical image is encoded in 8 bits, the histogram is generated with 256 bins, with values ranging from 0 to 255.

Global If the objects and background are of distinct intensity, then the distribution in the histogram will appear bimodal, as in Fig. 20. The most straightforward way to determine a threshold would then be to take the minimum between the two maxima. But there exists more clever analysis that investigates the cumulative distribution function (CDF) of the histogram, such as Otsu's method, see the (vertcial) blue stippled line in the histogram. Applying a single threshold to the image is also called *global* thresholding.

Band, Multi-Level Instead of splitting values in two halves as in global thresholding, it may also make sense to specify ranges. If we specify a range that constitutes one class, and the values outside that range another class, then that would be called a *band* threshold.

If a histogram shows multiple distinct modes, as is the case for many complex scenes, then we can attempt to specify several thresholds, in which case we talk of *multi-level* thresholding. That is unlikely to give us a good segmentation result, but it may work if we search for specific regions parts or objects 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. Such techniques often extend the techniques for bimodal thresholding - as mentioned under global thresholding - to the individual modes of the multi-modal distribution. Another way would be the smoothening of the distribution followed by extrema detection as we did for face part detection (Section 2.3).

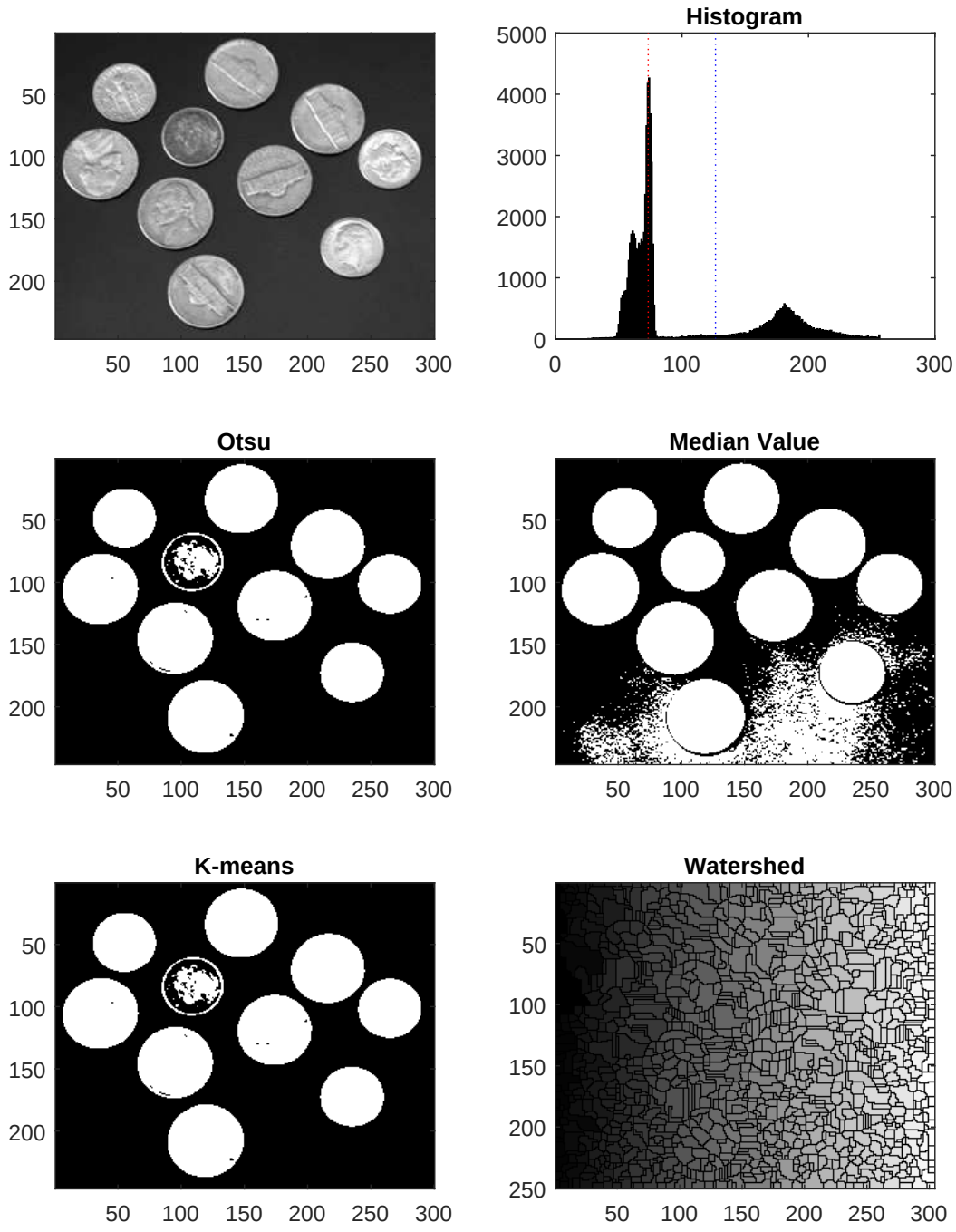

Figure 20: Segmentation methods in comparison (code in **L.9.1**).
Upper Left: image 'coins' from Matlab, `imread('coins.png')`.
Upper Right: intensity histogram distribution; the vertical blue stippled line is the threshold value of Otsu's method (in Matlab `graythresh`); the vertical red stippled lline is the median value.
Center Left: binary map as thresholded with Otsu's method.
Center Right: output as produced with a median threshold (shown for comparison).
Lower Left: clustering result with K-Means algorithm (`kmeans`).
Lower Right: output of the watershed algorithm. The watershed algorithm typically over-segments. In this case, the outline of the coins can be recognized nevertheless (with some effort).

Adaptive/Local There exists scenes that exhibit a slowly changing background color, which - seen as an intensity landscape - constitutes a shallow slope (gradient) across the entire image. The image in Fig. 20 shows signs of such a gradient, which is evident when we apply a median threshold, see center right graph. It then would make sense to search for a threshold value that is based on only a neighborhood and not the entire image, hence also called *local* or *adaptive* thresholding. This has proven to be particularly useful in document analysis, where photographed documents can exhibit a strong gradient due to light shining from one direction (Section 20.2.1).

In Matlab one can look at the intensity histogram with the function `imhist`; the function `graythresh` finds an optimal threshold between two peaks; the function `im2bw` thresholds the image using the level as specified by `graythres`. The function `multithres` is based on the method used in function `graythres`.

> **Caution** `graythresh/im2bw`**.** The function `graythresh` always generates the level as a scalar between 0 (black) and 1 (white) irrespective of the image's class type ('uint8', 'single', etc.). Correspondingly, the function `im2bw` expects a level specified between 0 and 1 and the original image class type. If you have converted the original image to class 'single' already - as recommended in previous exercises - then this may produce wrong results. Here it is better to work with the original image class type.

In Python those functions can be found in particular in sub-module `skimage.filters`, e.g. `threshold_otsu`. Python offers more methods to determine global thresholds than Matlab does.

In OpenCV [py] there exist the function `calcHist` for calculating the histogram, for thresholding one would use `threshold`.

```
H       = cv.calcHist([Igry.flatten()],[0],None,[256],[0,256])
thr, BW = cv.threshold(Igry,0,255,cv.THRESH_BINARY + cv.THRESH_OTSU)
```

Some of this is very well illustrated on the OpenCV documentation pages:
https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

Application Thresholding is particularly appealing for tasks that require fast segmentation, as in video processing for example.

9.2 Region Growing: The Watershed Algorithm

SHB p233, s6.3.4

Sze p251, s5.2.1, pdf283

In region growing one starts with several, selected pixels and then keeps expanding from those pixels until some stopping condition is fulfilled. The most popular algorithm is the watershed algorithm and as its name implies, one floods the intensity landscape until the rising water level meets the watersheds. To imagine that, we observe the image as an intensity landscape, see Fig. 3 again. In a first step, the algorithm determines the landscape's (local) minima and then proceeds by growing from those minima until the flood front encounters another growing flood front. The points of encounter form lines that correspond to watersheds. The resulting regions can be regarded as catchment basins, where rain would flow into the same lake.

Watershed segmentation is usually applied to a smoothed version of the gradient magnitude image ($\| \nabla I \|$, Section 3.3), thus finding smooth regions separated by visible (higher gradient) boundaries.

Watershed segmentation often leads to over-segmentation (see lower right in Figure 20), 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`
Python `skimage.segmentation.watershed`
Advantages no specification of the number of clusters necessary or any other parameter.
Disadvantages over-segmentation, slow

9.3 Clustering: Statistical Methods

Segmentation can also be carried out with statistical methods, in particular with *clustering* methods. Clustering is sometimes considered as part of the topic of machine learning, known there as *unsupervised* classification methods, because we have no actual labels/supervision. In principle, any clustering method can be used for image segmentation, but practically we prefer the ones that operate reasonable fast. The fastest method is the ubiquitous *K-Means* algorithm (Section 9.3.1), a method already used for feature quantization (Section 8). Another fast method is the *Mean-Shift* and its variants (Section 9.3.2). A slower method is the *Normalized-Cut* method, which is often used for foreground-background segregation (Section 9.3.3).

All those statistical methods treat the image as a matrix, namely number of image pixels times number of 'features'. In the simplest case, the features are the three chromatic channels RGB, thus a $n_{pixels} \times 3$ matrix, meaning we try to find clusters in three dimensions. We can also add the pixel coordinates as x and y values, thus generating a $n_{pixels} \times 5$ matrix; that would then be clustering in five dimensions. We can also add local texture measurements, as discussed in texture features (Section 4.3), in which case we move toward a high-dimensional space. In short, we select the features we think that are suitable to obtain our preferred segmentation results and then run it with our preferred clustering algorithm. The more features we take however, the slower becomes the clustering process as the dimensionality grows correspondingly.

9.3.1 K-Means

The K-Means algorithm is perhaps the most used clustering procedure for segmentation. We used the K-Means procedure already for feature quantization (Section 8.1.1), namely on a 128-dimensional problem. Here we use it to segment much lower-dimensional spaces, for instance in Fig. 20 we used merely one dimension, namely gray-scale intensity. If we have a color image, then it would be obvious to start clustering with only the three chromatic channles (RGB); later we would add more dimensions (features) to obtain our desired segmentation outcome.

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 k, meaning one needs to know how many different objects one expects. Suppose we have an image of a flower with a homogeneous background (leaves of other plants). Then perhaps we could start the segmentation with $k = 3$: one cluster for the flower leaves, one cluster for the background, and one for the remaining parts of the flower. For that purpose we reshape to image to an array of number of pixels times number of chromatic channels. Then we perform the clustering and obtain a label vector of length number of pixels. To visualize the clusters we can reshape the label vector to a matrix, also called the *label matrix*.

In Matlab we would write:

```
szI   = size(Irgb);                        % [rows columns 3]
ICol  = reshape(Irgb,[szI(1)*szI(2) 3]);  % turn image into data matrix for clustering [nPix 3]
Lb    = kmeans(single(ICol),3, 'MaxIter',50, 'online','off'); % Labels E [1,2,3]
Lmx   = reshape(Lb, szI(1:2));    % turn into label matrix
```

It is recommended to experiment first with the online phase turned `off`; the default is set to `on` and that takes very long time, for relatively little improvement.

In Python we can use the function `KMeans` from the module `sklearn.cluster`:

```
from sklearn.cluster import KMeans
...
ICol    = Irgb.reshape((nPix,3)).astype(float)
RKM     = KMeans(n_clusters=nK, max_iter=20, n_init=1).fit(ICol)
```

The variable `n_init` determines how many times the algorithm is run. We set it to one for simplicity - and because it most likely produces acceptable results. `RKM` is a structure (class) that contains variables for the labels. See Appendix **L.9.2** for full examples.

Advantages relatively fast: faster than region growing, but slower than thresholding
Disadvantages specification of k: number of expected clusters needs to be specified beforehand

9.3.2 Mean-Shift, Quick-Shift

The *Mean-Shift* procedure is similar to the K-Means in its use, but here one specifies a bandwidth value h and not an expected number of clusters. That has the obvious advantage that we do not need to provide a particular `k`, yet if we do not specify an adequate bandwidth value, then the results can be unsatisfying. Thus the challenge here is to estimate a reasonable bandwidth, but in some cases that is better than providing a fixed `k`. The mean-shift has the down-side that it is slower than the K-Means algorithm and for that reason a fast variant was developed named *Quick-Shift*.

Another down-side is that the algorithm works only for limited dimensionality, up to 6 or 7 dimensions in practice. The reason for that limitation is that the algorithm does not use distance measurements between 'pixels', but it computes gradients.

This type of segmentation is often used for tracking moving objects. An object in a scene often has a characteristic color histogram 'signature' that is distinct from its context and that is exploited with this type of algorithm (Section 14.3).

9.3.3 Normalized Cut

The normalized-cut algorithm is a clustering algorithm that is used in particular for generating two clusters. It is often used for a segregation between foreground and background, for example in a graphics programs to extract an object from its background. To initiate the procedure it requires two samples, one from each cluster, which practically are often provided by the user, for instance manually by mouse-clicks, one pixel from the background and one from the foreground.

The advantage of this clustering procedure is that it can arrive at better results than the other clustering algorithms. Its downside is that it takes considerably more time to arrive at the segmentation result and that it requires two samples as initial input. It is therefore rather used for individual images. The algorithm starts typically by relating the features at each pixel with all its neighboring pixels, which results in much more computation than the above k-Means and mean-shift procedures.

An example of how to apply normalized cuts is given in Section 9.4 of the SciKit-Learn documentation.

9.4 Deep Nets

The methodology for segmentation with Deep Nets is similar to the one for classification. A good starting point might be:

https://github.com/mrgloom/awesome-semantic-segmentation

10 Morphology and Regions (Image Processing III)

Sze p112, s3.3.2, p127

Dav p185, ch7

Morphological processing is the local manipulation of the structure in an image - its morphology - toward a desired goal. Often the manipulations aim at facilitating the measurement of features such as regions, contours, shapes, etc. Morphological processing can also carry out filtering processes.

We have already given two use cases for morphological processing. In one example, morphological processing 'polished' the map of edge pixels that one has obtained with some edge-detection algorithm (Section 12.2). And we have mentioned several times that after application of a segmentation algorithm one would like to continue modifying the black-white image toward a specific goal. In both cases one would apply so-called *binary* morphology, coming up in Section 10.1. But one can also apply such manipulations to gray-scale images, called *gray-scale* morphology, which we will quickly introduce in Section 10.2.

After we have completed (segmentation and) morphological processing, we often want to count the objects or regions in the black-white image. Or if we know their count already, then perhaps we intend to localize the regions and describe them by a few parameters. This is called region finding or labeling. This is a fairly straightforward process and will be explained in Section 10.3.

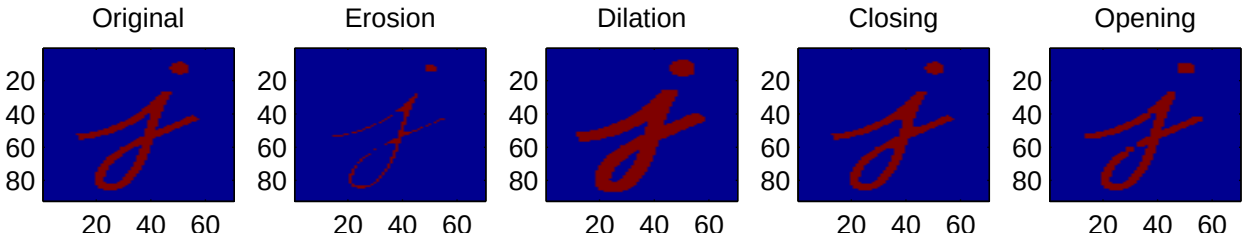

Figure 21: Binary morphological processing on a hand-written letter 'j'. From left to right:
Original: the object to be manipulated.
Erosion: a slimmed version of the original object: note that the resulting object is now ruptured.
Dilation: a thickened version of the original.
Closing: this process consists of two operations: first dilation of the original, followed by erosion. (the effects of any actual closing are not really viewable in this case)
Opening: first erosion of the original, followed by dilation. Note that some of the ruptures are now more evident.

10.1 Binary Morphology

Binary morphology manipulates only binary (black-white) images whereby the result is again a binary image. There exist two basic operations, the erosion and the dilation operation, which - as the names imply - make the *object(s)* shrink or grow in some specified way. Those two basic operations are then combined to form more complex operations. We introduce those operations with Figure 21:

Erosion: the object looses one or several 'layers' of pixels along its boundary.
Dilation: the object fattens by one or several layers of pixels along its boundary.

If one applies the above two operations in sequence, then that causes the following modifications:

Closing: is the dilation followed by an erosion, then that fuses narrow breaks and fills small holes and gaps.
Opening: is the erosion followed by a dilation, then that eliminates small objects and sharpens peaks in an object.

The last two sequential operations can be summarized as follows: they tend to leave large regions and smooth boundaries unaffected, while removing small objects or holes and smoothing boundaries.

Other combinations of those basic operations are possible, leading to relatively complex filtering operations such as the **tophat** and **bothat** operations, which however can also become increasingly time-

consuming. When developing an application, it is difficult to foresee which morphological operations are optimal for the task: one simply has to try out a lot of combinations of such operations and observe carefully the output.

More algorithmically speaking, the image is modified by use of a *structuring element*, which is moved through the image - very much like in the convolution process. The structuring element can be any shape in principal: it can be a simple 3×3 box filter as in the examples above; or it can be a more complicated structure, for instance some simple shape that one attempts to find in the image, leading so essentially to a filter process.

If one deals with large images, then one of the first steps toward object characterization or region finding might be to eliminate very small regions, which in software programs we can do with a single function. This is of benefit because morphological operations are carried out much faster after one has eliminated any unlikely object candidates.

In Matlab , binary manipulations are carried out with commands starting with the letters `bw` standing for black-white. Here are examples of some essential manipulations, whereby most manipulations can be found in function `bwmorph`:

```
BW      = imclearborder(BW);    % deletes regions touching image border
BW      = bwareaopen(BW,4);     % deletes regions < 4 pixels (4-pix remain!)
areaObjs    = bwarea(BW);       % summed areas of all objects

BW      = bwmorph(BW,'dilate',1);   % dilates by one
BW      = bwmorph(BW,'erode',2);    % erodes by two
BWper   = bwperim(BW);              % returns boundary pixels of objects
BWrem   = bwmorph(BW,'remove');     % removes interior of pixels
BWthn   = bwmorph(BW,'thin',inf);   % thinned
```

The operation 'thin' creates skeleton-like structures. It is particularly useful for contour tracing: one would apply first a thinning operation before trying to trace contours in a map.

More sophisticated manipulations can be achieved using a structural element defined with `strel`. There also exist functions such as `bwareafilt` and `bwpropfilt` to carry out complex filtering operations.

In Python the functions are found in different sub-modules, in `scipy.ndimage.binary_xxx` as well as in `skimage.morphology`. In examples:

```
from skimage.morphology import remove_small_objects, remove_small_holes, \
                               binary_erosion, binary_dilation, thin, disk
BWlrg   = remove_small_objects(BW, 4)
BWnoho  = remove_small_holes(BW, 4)
BWdil   = binary_dilation(BW)           # dilation by one (=disk(1)=5-pix cross)
BWero   = binary_erosion(BW,disk(2))    # erosion by two
BWthin  = thin(BW)                      # thins automatically to inf
```

In OpenCV [py] we need to define the structuring element first:

```
StcElm   = cv.getStructuringElement(cv.MORPH_ELLIPSE,(3,3))
BWero    = cv.morphologyEx(BW, cv.MORPH_ERODE, StcElm, iterations=2)
BWdil    = cv.morphologyEx(BW, cv.MORPH_DILATE, StcElm, iterations=2)
BWopn    = cv.morphologyEx(BW, cv.MORPH_OPEN, StcElm, iterations=2)

# in-place
cv.morphologyEx(BWero, cv.MORPH_ERODE, StcElm, dst=BWero, iterations=2)
```

10.2 Grayscale Morphology

The above introduced operations also exist in gray-scale morphology, but here the operation is not the change of a bit value, yet the selection of an extrema value in the neighborhood under investigation. In gray-scale morphology one talks of the *structural function*: in the simplest case, the function takes the maximum or minimum.

Matlab The operations are found under the initial letters `im`, for instance `imdilate`, `imerode`, `imopen` and `imclose`.

Python The operations can be found in `scipy.ndimage.grey_xxx` or some of them in `skimage.morphology`. For instance `skimage.morphology.dilation` carries out dilation for a gray-scale image and can also be applied to a black-white image, but for the latter the above functions starting with `binary_` are faster.

10.3 Region Finding, Description (Properties) and Boundary Detection

After segmentation and morphological processing, we need algorithms that search those regions that consist of connected pixels. In that context, one talks of *connected components*, which are defined as regions of adjacent pixels that have the same input value (or label). Connected components can be your objects of interest (value true or other labels), whereby the background may consist of one or more regions (value false); or the connected components may be some other 'targets'.

Label Matrix To find connected components, one runs an algorithm that labels the regions with integers: 1, 2, 3, ...$n_{regions}$. The labeled regions remain in the map whereby the value 0 signifies background. Labeling can occur with two principal different types of *connectivity*: the 'conservative' type uses only a so-called *4-connectivity* or 4-connected neighborhood and uses only the 4 neighbors along the vertical and horizontal axes. The 'liberal' type uses the so-called *8-connectivity* or 8-connected neighborhood and uses all 8 neighbors; it typically results in fewer and larger regions than the 4-connectivity.

Region Properties After we have located the regions, we may want to describe them. Software packages typically provide a function that describes regions with a number of simple measures based on geometry or statistics, e.g. various measures of the region's extent in the image, statistical moments, etc.

In Matlab we can find and characterize connected components in different ways:

1. `bwlabel`: returns a map with connected pixels set to a certain integer value for a given object. The objects are numbered 1,..$n_{objects}$. We then write a loop to find the objects' indices using `find`.
2. `bwconncomp`: returns a structure with fields corresponding to the objects' indices, information that the function `bwlabel` does not provide. 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 `bwlabel`.

Matlab's function `regionprops` can be applied in a variety of ways. Here are explained three ways. We can apply the function `regionprops` directly to the black-white image `BW`, in which case we have skipped the function `bwconncomp` - it is carried out by `regionprops` in that case. Or we apply the function `bwconncomp` first and then feed its output to `regionprops`. Or we use the function `bwboundaries`, which is useful if we intend to describe the silhouette of shapes, for example with a radial description as will be introduced in Section 11 on the topic of shape.

```
I       = imread('cameraman.tif');
BW      = im2bw(I,128/255);

%% =====    RegionProps directly from BW   ========
```

```
RPbw  = regionprops('table',BW,I,'maxintensity','area');

%% =====    RegionProps via bwconncomp   ========
CC      = bwconncomp(BW);
RPcon  = regionprops('table',CC,I,'maxintensity','area');

%% =====    RegionProps via bwboundaries   ========
[aBon MLbon]    = bwboundaries(BW,8,'noholes');
RPbon  = regionprops('table',MLbon,I,'maxintensity','area');

%% -----    Verification  --------
assert(all(RPbw.Area==RPcon.Area));     % compare direct and bwconncomp
% via bwboundaries: only same for bwboundaries(BW,8,'noholes');
assert(all(RPcon.Area==RPbon.Area));    % compare bwconncomp and bwboundaries
```

The function `regionprops` provides only simple region descriptions. For more complex descriptions, one inevitably moves toward the topic shape description and that will be introduced in the upcoming section.

In Python there exists slightly less flexibility than in Matlab. The sub-module `skimage.measure` provides the functions `label` and `regionprops`. The label function requires type `int` as input, and the function `regionprops` takes only a label matrix as input:

```
from skimage.measure import label, regionprops
...
LB = label(BW)        # we call 'label' first, because...
RG = regionprops(LB)  # ...input to 'regionprops' must be of type int! (not logical)
nShp = len(RG)
print('# Shapes', nShp)
```

To obtain the list values as a table - as a single array -, we write a loop as follows, a formulation that is called *list comprehension* in Python:

```
Ara = asarray([r.area for r in RG])        # obtaining all area values
```

In OpenCV [py] there are several functions. The bare function script `connectedComponents` returns only the number of detected regions and the label matrix `Lmx`. With `connectedComponentsWithAlgorithm` we can specify a different labeling algorithm. With `connectedComponentsWithStats` we obtain minimal region properties. In OpenCV, the background is included in the region count as well as a first entry in the region properties and centers.

```
BW        = BW.astype(uint8)    # source must be uint
nC1, Lmx1 = cv2.connectedComponents(BW, connectivity=8)
nC2, Lmx2 = cv2.connectedComponentsWithAlgorithm(BW, connectivity=8, \
                ltype=cv2.CV_16U, ccltype=cv2.CCL_DEFAULT)
# Prop is array [nReg 5]: left, top, width, height, area
nC3, Lmx3, Prop, Cen = cv2.connectedComponentsWithStats(BW)
# exclude background (1st entry)
nC3     -= 1                # now we have number of conn comps
Prop    = Prop[1:,:]
Cen     = Cen[1:,:]
# sort by decreasing area
Odec    = (Prop[:,-1]).argsort()[::-1]
Prop    = Prop[Odec,:]      # region properties ordered
Cen     = Cen[Odec,:]       # region centers ordered
```

11 Shape

Dav p227, ch 9 and 10

Shape means the geometry of an object or its form. It is a 'structure' consisting of a few segments; sometimes it is a set of points with; but typically without any texture. Shape description techniques are used in the following applications for example:

- 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 structure
- 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 applications, the task to be solved is the process of *retrieval*, namely the ordering (sorting) of shapes, and less so the process of classification, see Section 1.2 again to understand the difference. In a retrieval process, one shape is compared to all other shapes and that is computationally much more intensive than just classification. Thus, one major concern in retrieval is the speed of the entire matching process.

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 - with the exception of Davies' book - , because there is no dominating method and it is somewhat unsatisfactory and endless to present all techniques. Perhaps the two most important aspects in choosing a shape matching technique are its matching duration and its robustness to shape variability.

Shape Variability Depending on the task or the collection, shapes can vary in size or in spatial orientation; they can be at different positions in the image; they can appear mirrored; their parts may be aligned slightly differently amongst class instances; their context may differ. These different types of 'condition' are also called *variability*. Ideally, a shape matching technique would be invariant to all those variabilities. The table below shows the terminology used with respect to those desired shape matching properties:

Variability	**Invariance**
size	scaling
orientation	rotation
position	translation
laterality (mirroring)	reflection
alignment of parts	articulation
blur, cracks, noise	deformation
presence of clutter	occlusion

Practically, it is impossible to account for all these invariances and therefore one needs to observe what type of variability is present in the shape database and make a choice of the most suitable technique. This choice is also important for classification techniques.

The first three sections introduce shape descriptions of increasing complexity. Section 11.1 introduces simple shape descriptions suitable for rapid retrieval. Section 11.2 introduces techniques based on point comparisons: those techniques have a longer matching duration but show better retrieval accuracy. Section 11.3 introduces part-based descriptions: they can be even more accurate, but the matching techniques are rather complicated. Thus, the choice of a matching technique is also a matter of dealing with a speed-accuracy tradeoff. In the final section 11.4, we have a word on shape classification systems.

11.1 Compact Description

In compact descriptions, the shape is expressed by a number of parameters and those are used to form a feature vector. By using vectors one can conveniently apply traditional classification methodology. In Section 11.1.1, we mention simple measures based on the boundary or interior of the shape. In Section 11.1.2 we introduce the radial description, which is probably the most efficient description that uses feature vectors.

11.1.1 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:

Area	size of (shape) region (`bwarea`)
Circularity	e.g. area shape / area circle (circle with equal diameter)
Principle Axis Ratio	(also called eccentricity or aspect ratio) ratio between axis of elongation and its orthogonal axis (or length of major axis / length of minor axis)
Euler number	$= S - N$: # contiguous parts - # holes. Example shape '3': 1=1-0; shape 'B': -1=1-2; shape '9': 0=1-1. Matlab: (`bweuler`)
Bending Energy	degree of curvature ($\sum \kappa(s)$)

Matlab `regionprops`, introduced in Section 10.3 already
Python `regionprops` in submodule `skimage.measure`

The code example in **L.10.1** and **L.10.2** show an example of how to generate and manipulate simple shape representations.

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

11.1.2 Radial Description (Centroidal Profiles)

Dav p269, s 10.3

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*, see Figure 22, also called *centroidal profiles*. For that we need the silhouette points of the shape - its boundary. In Matlab the boundary can be obtained with the function `bwboundary`, see Section 10.3. In Python there exists the function `find_contours`.

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. Sometimes one uses angle Ω as the dependent variable instead of curve point s.

For a circle, the radial signature would be a constant value. For an ellipse, the signature would be undulating with two 'mounds'. For a triangle, there would be three sharp peaks. For a square, there would be four peaks. And so on. For a complex shape, such as the the pigeon shape in the figure, the signature is relatively complex.

There are two relatively straight-forward analyses we can do with the radial signature. One is a Fourier analysis, meaning we express the signature as a spectrum of frequencies. This is an enormously powerful analysis, which merits its own lectures, but is rather the topic of a signal processing course. The other analysis is an investigation of the extrema present in the signature. The Fourier analysis is more discriminative than the extrema analysis, but combining both can yield even better results.

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 ($R(s)$) and normalize it by its first value:

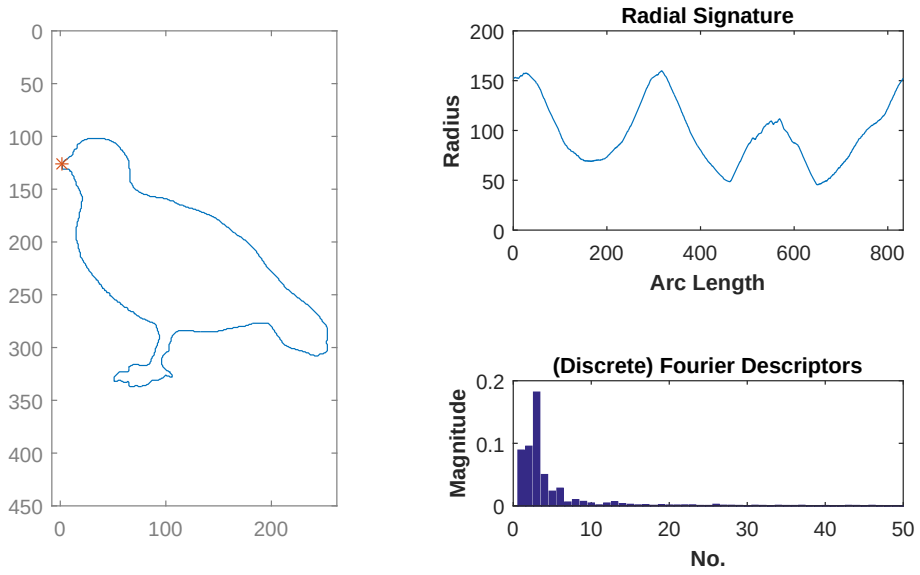

Figure 22: **Left:** a pigeon shape; the asterisk marks the beginning of the signature. **Upper Right:** radial signature: the distances between pole (shape center) and the individual shape points. **Lower Right:** discrete Fourier descriptors of the radial signature.

```
FDabs    = abs(fft(Rad));         % fast Fourier
FDn      = FDabs(2:end)/FDabs(1); % normalization by 1st FD
```

In the lower right of Fig. 22, the first 50 Fourier descriptors are shown. But 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 first low-pass filter the signature - very much as in the analysis of facial profiles, see introductory exercise. 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 an ellipse or bicorn shape, three corners to a triangle or trident shape, etc.

The presented descriptions can not discriminate large sets of shapes, but their use may serve as a triage for a more complex description and matching. Appendix **L.10.3** shows an example of how to generate the radial and Fourier descriptors.

11.2 Point-Wise

There are two cases of point-wise formats one can distinguish. In one format, a shape is expressed as a single boundary, a sequence of points, which typically corresponds to its silhouette (Section 11.2.1). In the other format, a shape is considered as a set of points (Section 11.2.2). Because the shape comparison with such formats is relatively time-consuming, 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 in an approximate sense.

11.2.1 Boundaries

In this case, the list of points of one shape, are compared to the point list of another shape, by somehow determining a distance (or similarity) measure between the two lists. The simplest way would be 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 one could simply shift the two shapes against each other to find a minimum for the correspondence. This would increase 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 that are at locations of high curvature in a shape. We can do this using the radial, the curvature or the amplitude signatures, see Sections 11.1.2 and 12.3. The problem is that such key-points are difficult to determine consistently.

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 $i = 1, .., N$ 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.

11.2.2 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.

1) 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.

2) Limited Variability; Few Points This case is rather useful for localization and 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 target shape in another image. We are then faced with two tasks: the correspondence problem and the transformation problem.

11.3 Toward Parts: Distance Transform & Skeleton

Because we humans interpret a shape as an alignment of segments or parts, it was assumed from the early days of computer vision, that a shape description should also consist of segments or parts somehow. Decades of trial-and-error has taught computer vision scientists, that such a description is difficult to achieve. What exactly is supposed to be a segment or a part and how those should be represented, are still 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 that distance transform (Section 11.3.1), we mention the difficulties with obtaining segments and parts (Section 11.3.2).

11.3.1 Distance Transform

Sze p113, s 3.3.3, pdf 129
Dav p240, s 9.5
SHB p19, s 2.3.1

The distance transform is typically determined for a binary image. The transform calculates at each background (off) pixel the distance to the nearest object (on) pixel. This results in a scalar field called *distance map* $D(i, j)$. The distance map looks like a landscape observed in 3D, which is illustrated in Figure 23. 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

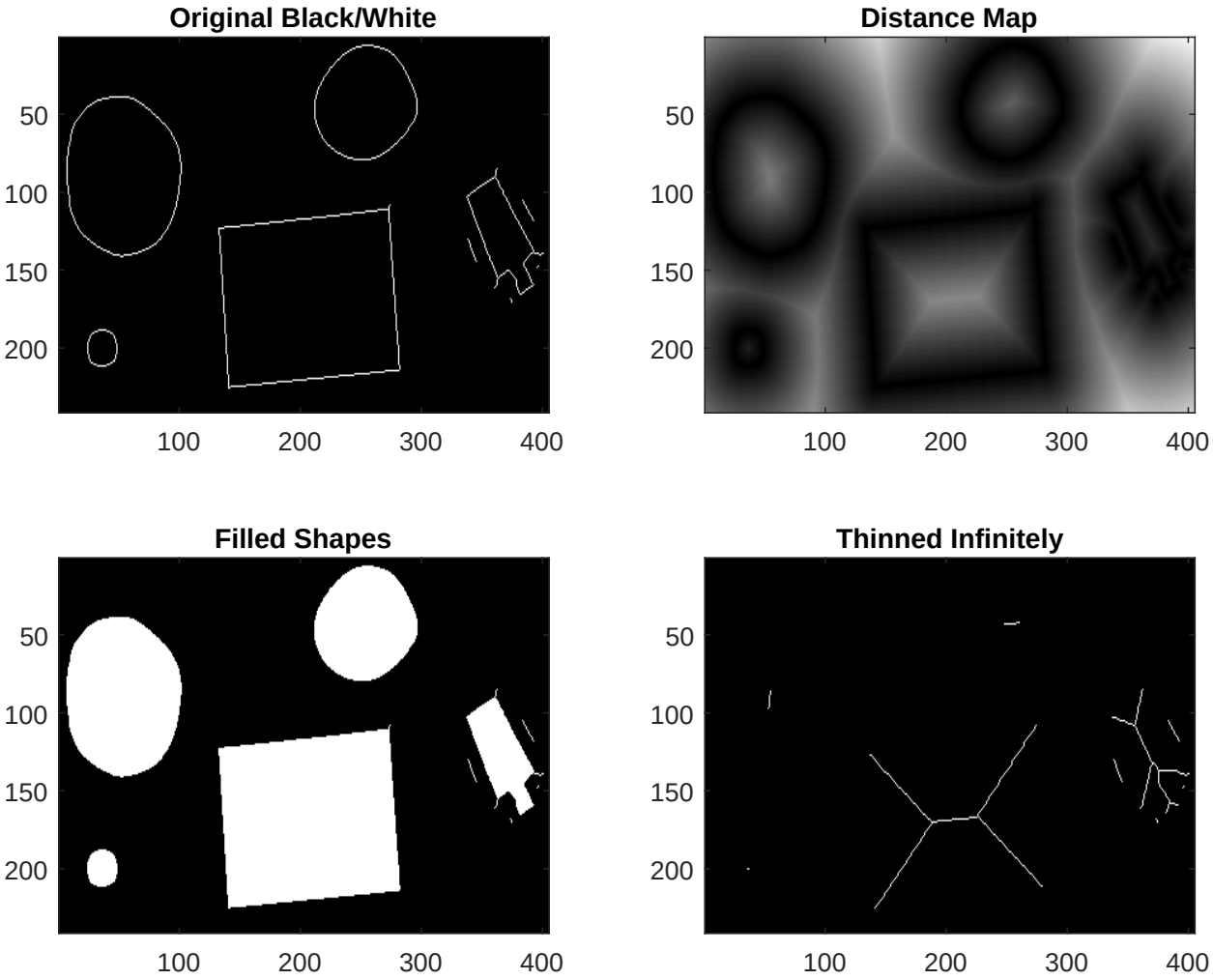

Figure 23: Distance map and skeleton.
Upper Left: input image.
Upper Right: its distance map obtained with `bwdist`.
Lower Left: Filled shapes of the original image.
Lower Right: quasi skeletons of the shapes, obtained using the 'thin' option of `bwmorph`.

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. Sometimes that skeleton is also called *medial axis*.

The distance transform can be calculated with different degrees of precision. Simpler implementations provide less precision but calculate quicker an imprecise distance value; they are based on the Manhattan distance for instance. Precise implementations use the Euclidean distance.

In Matlab the function `bwdist` calculates the distance values for off-pixels, meaning on-pixels are understood as boundaries (as introduced above). It calculates Euclidean distances by default, but a simpler implementation can be specified as option.

```
DM  = bwdist(BW);                    % distance calculated at OFF pixels
```

In Python the distance transform can be found in the `scipy` module, specifically `scipy.ndimage.distance_transform_edt` for Euclidean precision. It calculates the distances at on-pixels (in contrast to Matlab). Simpler implementations are provided by separate function scripts, ie. `distance_transform_cdt`.

```
from scipy.ndimage import distance_transform_edt
DM  = distance_transform_edt(BW)        # distance calculated at ON pixels
```

In OpenCV [py] the behavior is as in Python: the distance is calculated at on-pixels.

```
DM = cv2.distanceTransform(BW.astype(uint8), distanceType=cv2.DIST_L2, maskSize=cv2.DIST_MASK_PRECISE)
```

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

11.3.2 Symmetric Axes (Medial Axes), Skeleton

The symmetric axes are the ridges in the distance map, the contours that look like veins (upper right in Figure 23). 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 (Section 10.1), which when carried out infinitely results in a skeleton resembling the sym-axes, see lower right in Figure 23 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.

In Matlab those implementations are accessed throught the function script `bwmorph`:

```
BWs  = bwmorph(BW,'skel',Inf);
BWs  = bwmorph(BW,'thin',Inf);
```

In Python there exist separate function scripts in module `skimage.morphology`

```
BWs  = medial_axis(BW)
BWs  = skeletonize(BW)
```

The differences between those implementations is subtle and it is diffult to foresee when what implementation is more appropriate for a specific task. One general difficulty with any of those implementations is that they are limitedly robust to subtle variability (see table 11 again). In particular, they are limitedly invariant to articulation and deformation, but even for rotation of the same shape, the output of an algorithm may change.

11.4 Classification

When it comes to classification, the most successful systems are not the above introduced techniques, but Deep Neural Networks as introduced in Section 5. Even for the digit database MNIST, the use of symmetric axes for instance, has not provided better prediction accuracies than machine learning algorithms.

There are two disadvantages with CNNs for shape recognition. One is, that they require the shape to be fairly well centered in the image; thus, a search algorithm is necessary that finds the exact shape center. The other disadvantage is, that they require fairly long to learn the features, as pointed out already in Section 5, though with transfer learning that short coming is almost eliminated.

The power of those networks comes from their robustness to local changes: small changes in the boundary do have little 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 23: the corresponding skeletons in the lower right graph, show sufficient differences that make a robust comparison difficult.

12 Contour

Contours outline the objects and parts in a scene. Contours can be detected by edge detection as introduced in Section 4.2. But to understand the object's shape (or scene part) we need a description. As contours are often fragmented due to various types of noise and illumination effects, it requires a description that can deal with open contours - as opposed to the radial description for closed contours (Section 11.1.2). In Section 12.3 we introduce the challenges toward that goal. But before we arrive there, we need to coil up the individual edge pixels of the edge map, a process called *edge following* or *edge tracing* (Section 12.2).

If one needs to detect straight lines or circles only - as in case of some specific applications - then we can use statistical methods to find them (Section 12.1). Those methods are also relatively robust to contour fragmentation.

Finally, it is worth pointing out that there exist also other contour types, used in some particular scenarios (Section 12.4).

12.1 Straight Lines, Circles

If an application requires the extraction of straight lines only, then there exist some dedicated techniques to find them. The most popular 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: one is the distance ρ of the straight line from the image origin; and the other is the angle θ. 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 $(\rho = \sqrt{2}/2, \theta = 45°)$.

The Hough transform can also be modified to detect circles. There are also modification that extend it to ellipse detection.

Application: Straight lines are used to find vanishing points in road scenes for instance (Figure 44). Or are used for calibration of internal and external camera parameters.

In Matlab those detectors can be found under `houghlines` and `imfindcircles`.

In Python the functions reside in module `skimage.transform` as `hough_xxx`.

12.2 Edge Following (Curve Tracing)

Dav p257, s 9.8

Edge following is also called *edge tracing* or *edge tracking*. Sometimes it is also called *boundary tracing*, which makes sense when edges represent a region boundary.

There are two issues to resolve before we start tracing. One is that contours in edge maps are occasionally broader than one pixel, which can make tracing difficult, because such clusters represent ambiguous situations; they occur in particular when the contour runs along the diagonal axes. To avoid such thickened contours, one can employ the operations of morphological processing as introduced previously, binary morphology in particular (Section 10.1). With the so-called *thinning* operation we can 'slim' those thick contour locations. Or perhaps we wish to remove isolated pixels immediately by using an operation *clean*:

```
Medg = edge(I, 'canny', [], 1);
Medg = bwmorph(Medg,'clean');   % removes isoloated pixels
Medg = bwmorph(Medg,'thin');    % turns 'thick' contours into 1-pixel-wide contours
```

Another issue is that we need to decide how to deal with junctions. Let us take a T-junction as an example: should we break it up at its branching point and trace the three segments individually? Or should trace around it - an obtain a boundary - which we later partition into appropriate segments? This is an issue of representation in principle.

Matlab There are several ways to do this: two involve the use of Matlab functions, one way would be to write our own routine, see also example in Section **L.11.1**.

1. `bwboundaries`: this routine is useful for finding the boundaries of regions. In case of a contour, the routine will trace around the contour and half the contour pixels match with the opposite pixels: if one preferred the pixel coordinates only, then one had to eliminate the duplicate coordinates.
2. `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 `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: the easiest approach would be to trace contours using their end- and branchpoints. The endpoints can be found with the command `Mept = bwmorph(Medg,'endpoints')`; the branchpoints can be found with the option `'branchpoints'`.

Python There exists no routine designed specifically for that task. Python however offers a routine to measure iso-contours, called `skimage.measure.find_contours`. One specifies a height value at which the map is thresholded and then the corresponding region boundaries are taken. This can be exploited to emulate boundary detection in binary images. If one specifies a value of 0.8 (between 0.5 and 1.0), then the contours lie closer to the region pixels; if one specifies a value of 0.2 (between zero and 0.5), then the contours lie closer to the adjacent exteriors pixels of the region. The example in Section **L.11.1** demonstrates that.

12.3 Description

To properly describe an open or closed contour it is necessary to move a window through the contour and take some measure of the window's subsegment. This results in a signature analogous to the radial signature introduced for shapes 11.1.2. There are two types of signatures that have been pursued so far: curvature and amplitude.

12.3.1 Curvature

The curvature measure is calculated by simply taking the derivatives of the contour. The second derivative is our signature. Here is an improvised curvature measure, where `Rf` and `Cf` are our coordinates (row and column indices):

```
%% =====    1st Derivative      =======
Y1          = [0; diff(Rf)];
X1          = [0; diff(Cf)];
%% =====    Curvature Improvised    =====
Dv1   = Y1+X1;                    % adding 1st derivatives
Drv1    = conv(Dv1,Lpf,'same');     % low-passfilter again (to smoothen)
Drv2     = abs([0; diff(Drv1)]);    % 2nd dervivative: that is our curvature
```

Wherever a peak in the signature occurs, that is where the contour exhibits sharp curve, a *point of highest curvature*. This is sometimes used for shape finding and identification. If there is no peak, then we deal with a perfect, smooth arc or with a circle.

If the coordinates are integer values, then it is better to firstly smoothen the coordinate values. Appendix **L.11.2** demonstrates a complete example.

The shortcoming with this signature is that it is suitable only for a certain scale: we are not able to detect all points of highest curvature in arbitrarily sized contours. Broad curvatures are easier detected when the signature has been smoothened with a correspondigly large lowpass filter. In order to cover all curvatures in an arbitrary contour, one therefore generates a space, the *curvature scale space* (CSS) by low-pass filtering the curve for a range of sigmas, analogous to the image scale space as introduced in Section 3.1.

12.3.2 Amplitude

The curvature measure has the downside that it modifies the signal by low-pass filtering it. It would be better to leave the signal as is and take an alternate measure. That alternate measure would be the amplitude of the subsegment, the distance between chord (equation) and curve pixels (of the subsegment). The resulting signature does not look much different than a curvature signature, but in a space it allows a more reliable detection of points of highest curvature. The downside is that calculating the amplitude is a relatively costly procedure.

12.4 Other Contour Types

In Fig. 10 we had shown the output of a contour-detection algorithm that focused on edges - it is the most informative type of contour. But there exist three other types of contours that can be informative as well:

Ridge contours: describe the ridges in the intensity topology. Ridge contours have been used for finger print identification.

River contours: describe the bottom of the valleys. The latter are shown on the left in Fig. 24.

Iso-contours: correspond to the line of equal height in a topological map: they have the same intensity value. Examples of those are shown on the right in Fig. 24. Iso-contours have been used for nuclei detection and classification in histological images.

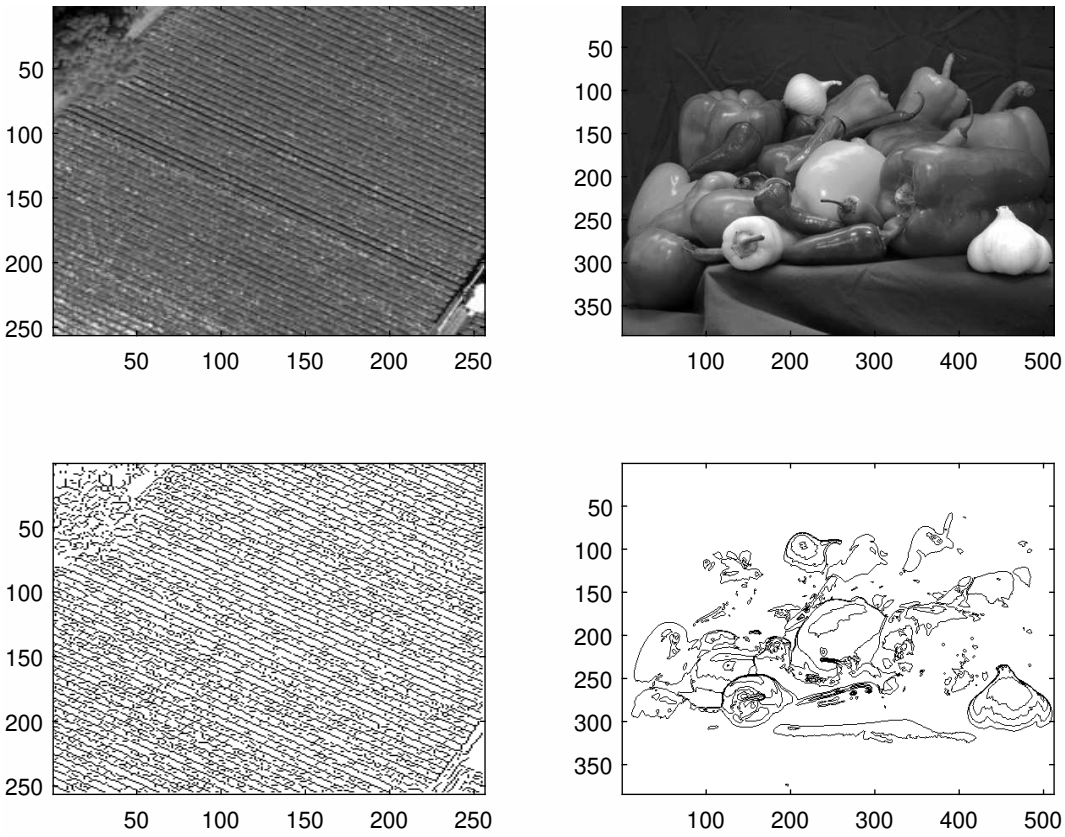

Figure 24: **Left Column:** river contours detected in a satellite image of an agricultural field. Those contours are difficult to detect with edge detectors due to their tight spacing. **Right Column:** iso-contours taken at various intensity levels; some of them represent well surfaces and also regions reflecting the light source.

13 Image Search & 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: the user looks through a set of images to see what is interesting.
- searching: the user describes what he wants, called a *query*, and then receives a set of images in response.
- retrieval: the user uploads an image and in return obtains the most similar images, ranked by some similarity measure.

Traditionally, methods of image retrieval utilized metadata such as captioning, keywords, or (textual) descriptions to find similar images, that is they would not use any computer vision. Then, with increasingly powerful computer vision techniques, search started to be carried out also with the actual 'pixel content' using simple image histogramming in the early days. This new approach was then called *content-based image retrieval* (CBIR). For some time, CBIR was based on the (engineered) features as introduced in early sections (Section 4), but modern CBIR uses also Deep Nets of course.

Note that images are selected by some measure of similiarity - they are not classified as in image classification, see again Section 1.2 to understand the difference or the introduction on shapes in Section 11. For a query image, a similiarity to all other images is calculated and then the images are ranked according to that similarity measure.

The following section does therefore not offer novel computer vision methods, but introduces terminology and methods from the field of information retrieval, that optimize the outcome of a search. Although that terminology was developed in particular for feature engineering techniques, it is equally useful for feature learning techniques.

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 25 below).
2) Copyright protection.

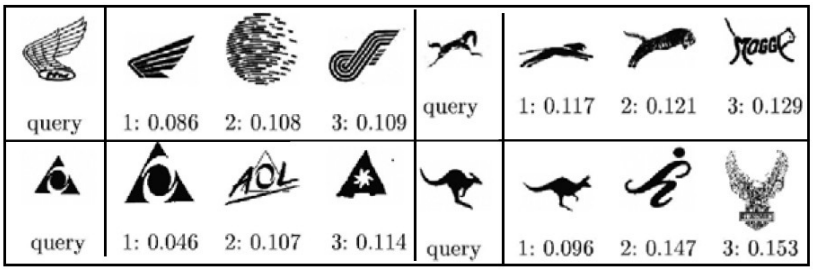

Figure 25: A trademark identifies a brand; customers should find it unique and special. This means that, when one registers a trademark, it is a good idea to know what other similar trademarks exist. The appropriate notion of similarity is a near duplicate. Here we show results from Belongie et al. (2002), who used a shape-based similarity system to identify trademarks in a collection of 300 that were similar to a query (the system mentioned in Section 11.2.2). 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.

13.1 Retrieval Methods

Sze p604, s 14.3.2, pdf687

FoPo p662, s 21.2, pdf632

We now review techniques from document retrieval, that found their ways into image retrieval. A typical text information retrieval systems expects a set of query words. With those query words an initial set of putative matches is selected from an index (Section 13.1.1). From this list they chose documents with a large enough similarity measure between document and query (Section 13.1.2). These are ranked by a measure of significance, and returned (Section 13.1.3). For the purpose of image retrieval we can think of the words as being the 'visual words' as in developed in Section 8, and of documents as being the images. Some more comparison is given in the final section 13.1.4.

13.1.1 Indexing Documents

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.

Assume now that the total number of non-stop words is N_w; and we work with N_d documents. For each document j we determine the frequency f with which a word t occurs. This will generate a $N_w \times N_d$ table $\mathcal{D}_{tj}$ in which an entry represents the frequency f for a word t in a document j:

$$\{f_{t,j}\} = D(t, j) \tag{14}$$

The table is sparse as most 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

$$\text{idf} = \frac{N_d}{N_t}. \qquad \frac{\text{total}}{\text{containing term}} \tag{15}$$

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

$$f_{t,j} = \Big(\frac{n_t(j)}{n_w(j)}\Big) / \log\Big(\frac{N_d}{N_t}\Big). \tag{16}$$

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.

13.1.2 Comparing Documents

Assume we have a fixed set of terms that we will work with. We represent each document by a vector $\mathbf{f}$, with one entry for each term. Put differently, we select a subset of dimension t of the table. Two document vectors, e.g. $\mathbf{f}_1$, $\mathbf{f}_2$, can then be compared using the dot product and normalization by their respective lengths:

$$\text{sim} = \frac{\mathbf{f}_1 \cdot \mathbf{f}_2}{\| \mathbf{f}_1 \| \| \mathbf{f}_2 \|}. \tag{17}$$

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.

13.1.3 Ranking Documents

In response to a user query, a set of N_b documents is returned, where N_b needs to be specified, i.e. 100 items are returned to the user. To determine how fitting the selected documents are, one calculates two measures:

Recall: the percentage of relevant items that are actually recovered

$$R = \frac{n_r}{N_r}, \qquad \frac{\text{retrieved}}{\text{total}} \tag{18}$$

where n_r is the number of recovered relevant items and N_r is the total number of relevant items for this query. Example: if the database has a total of 200 relevant items for the user query and the selection returned 60 relevant items, then the recall value is 60/200.

The larger we make our selection N_b, the larger will be R and if we return all documents then R will be one.

Precision: the percentage of recovered items that are actually relevant

$$P = \frac{n_r}{N_b}. \qquad \frac{\text{retrieved}}{\text{query set}} \tag{19}$$

Example (continued): the precision value would be 60/100. The larger we make the selection N_b, the lower will be the precision value typically: if N_b is set to be 1000 and if this happen to return us all 200 relevant items, the precision is 200/1000.

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:

$$F_1 = 2\frac{PR}{P+R}. \tag{20}$$

Precision-Recall Curve To obtain a more comprehensive description of the system, one calculates the precision values for increasing recall by systematically increasing N_b. One plots recall on the x-axis against

precision on the y-axis and this curve is called the precision-recall curve .

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 19; N_r the total number of documents in the collection. Then, the average precision is given by

$$A = \frac{1}{N_r} \sum_{r=1}^{N_r} P(r). \tag{21}$$

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, `Ix`, the measure is computed as follows:

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

13.1.4 Application to Image Retrieval

In order to apply the above measures to CBIR, we simply use 'visual words' - as in developed in Section 8 - 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.

In the domain of texture recognition (Section 4.3.2), those words are also called textons, but in object recognition and image classification they tend to be called visual words. Because the histogram distributions for those words are generally different from those for gradients (as in SIFT), alternate distance measures such as the χ-squared kernel may perform better.

It is tempting to believe that good systems should have high recall and high precision values, but obtaining high values can also be costly. The following examples illustrate that trade-offs are sometimes made:

- 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.

14 Tracking

Tracking is the close pursuit of one or multiple moving objects in a scene. Tracking is used in many applications:

Surveillance: traffic analysis: counting traffic participants, such as cars, pedestrians, cyclists, etc.; monitoring objects to report when something is suspicious, trucks at airports with unusual movement patterns.

Human Computer Interaction (HCI): continuously localizing the precise position of a user's head and orientation to facilitate the recognition of an user's emotions or the direction of his gaze.

Automotive Vision: anticipating when a moving object might enter the driving direction, i.e. a pedestrian.

Tracking is computationally intensive because we deal with a rapid series of images, called *frames* in this context. Image analysis has to be very rapid in order to achieve real-time tracking; algorithms need to be optimized for speed; complex algorithms can only be applied intermittently. The more objects we track, the more computing power is required. This multi-object tracking is manageable, if the camera is stationary; however when it is moving, as in an autonomous vehicle, then tracking becomes enormously challenging, because in addition to the moving objects, we have also the motion of the camera (egomotion). And if one attempts to maximize tracking accuracy with DeepNets, then we require also a lot of memory and power, something that is still a hindrance for embedded systems.

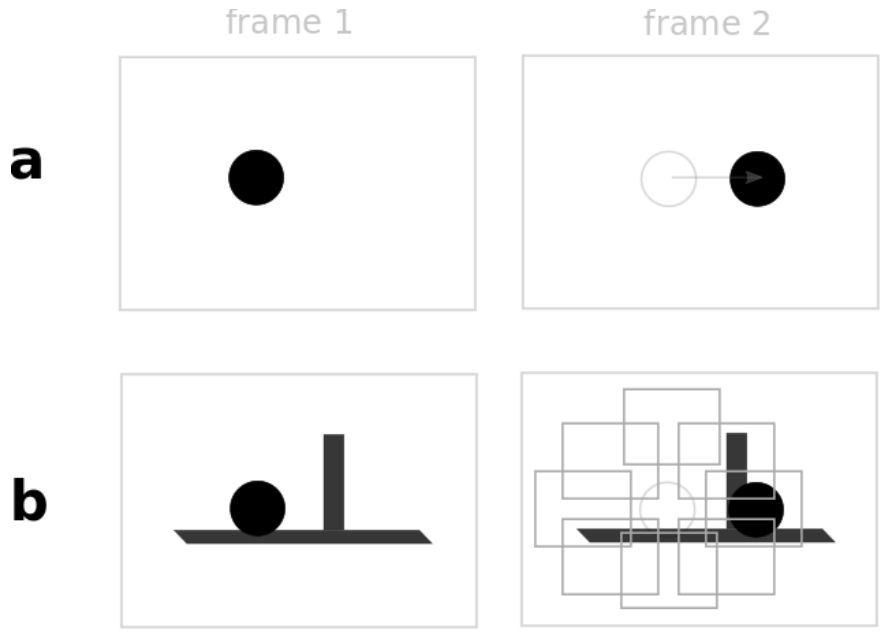

Figure 26: Tracking an object, a black disk moving to the right; two frames are shown.
a. In simple scenarios, such as surveillance, tracking can be done by firstly detecting regions that have moved, for instance by comparing frames pixel-wise. We then merely link up the detected, moving objects. This is also referred to as *tracking-by-detection*.
b. In complicated scenarios, when the background is a distractor, then tracking is done by knowing already what we intend to track and then searching over a small neighborhood to ensure we find the object. This search is illustrated here by 8 grey neighborhoods, radially placed around the object's location of the first frame. In that case one also speaks of *tracking-by-matching*, because one uses the object's properties to search for matching neighborhoods in the next frame.

For surveillance, there exists a relatively controlled situation, partly by design. The camera is often placed high above the zone to be observed, which results in small objects on an often homogeneous background. The size of the moving region corresponds to the size of the object. As the camera is stationary, so

is the scene, and we can therefore use so-called *background subtraction* to easily detect moving objects. Tracking in that situation is sometimes described as tracking-by-detection; Figure 26a depicts the situation. We firstly introduce background subtraction (Section 14.1), then the techniques for pursuing multiple objects (Section 14.2).

Tracking becomes a bit more challenging in the scenario of face tracking for Human-Computer Interaction (HCI) for instance. Not only does the target move - the user's head - but also the target's context, namely the user's throat and shoulders, resulting in one large contiguous region moving across the display. It now requires a more elaborate tracking scheme, namely one in which we pursue the object's properties. This is done by searching in a small neighborhood around the tracking target, depicted in Fig. 26b. This is also referred to as *tracking-by-matching* sometimes, because we try to match object properties to small neighborhoods in the next frame. Tracking-by-matching also means that we now need to find the object first - for which we apply object detection algorithms as introduced in Section 6. In case of face-tracking, there exists a particularly succesful tracking technique that uses a clustering algorithm, the so-called mean-shift algorithm (Section 14.3).

Tracking becomes even more challenging, when the target object also drastically changes its appearance during the motion. For instance, seen from an autonomous vehicle, pedestrians increase in size very quickly. In that case, we wish to have a tracker that also adapts to the appearance changes during target pursuit (Section 14.4). That makes tracking computationally even more expensive. To reduce the search over a small neighborhood, it is useful to make predictions of where the objects might go next. For that purpose we carry a number of observations with each tracked object, for example motion direction and magnitude (speed). This would also help us resolve ambiguities when two objects cross their paths, a situation which easily leads to confusions of tracks. Two popular prediction schemes are mentioned in Section 14.5.

14.1 Background Subtraction

FoPo p291, s 9.2.1

SHB p776, s 16.5.1

Background subtraction is a simple method to determine which parts of the scene are stationary. It makes sense to use it when the camera is stationary. Background subtraction is challenging for a number of reasons: changing daylight affects the overall scene luminance; clouds introduce temporary luminance changes; vegetation flutter is a source of noise; wandering object shadows cause easily confusions, etc. Background subtraction therefore requires continuous updating of the background parameters and is for that reason not quite as trivial as it sounds. One therefore needs to specify a set of parameters, such as the number of frames across which the background is calculated or a threshold to suppress noise. The most common technique to actually determine the background image is to take the median value for a pixel, taken a cross a number of frames.

In Matlab this process of background monitoring is called `ForegroundDetector` - the inverse of the background. It is initialized as follows for instance - before any tracking:

```
ForeGnd  = vision.ForegroundDetector('NumGaussians', 3, 'NumTrainingFrames', 40, ...
                                     'MinimumBackgroundRatio', 0.7);
```

During tracking, when we inspect each frame, we update the detector by providing the frame `Frm` itself:

```
Msk     = ForeGnd.step(Frm); % foreground [m n nCh]
```

where the output is called a mask, `Msk`, and is of the same size as the frame (number of pixels, number of channels).

In Python we can access methods for background subtraction through OpenCV. They are initialized with function names called `createBackgroundsubtractorXXX`:

```
import cv2 as cv
BckSub  = cv.createBackgroundSubtractorKNN()
```

When looping frames we call the method `.apply` to generate the mask:

```
Msk      = BckSub.apply(Frm)    # generates a mask
```

When one observes the output of the mask for an outdoor scene, one will notice that the mask can be very noisy. For that reason, the mask is often post-processed with morphological operations as introduced in Section 10. Two examples are given in the code section **L.12.1**. The parameters for the morphological operations are chosen such, that we avoid the elimination of potential target objects. The remaining regions are then our objects, our detections. That result is also called the *detector response*. To obtain the actual object outlines, we label and measure the actual region pixels as introduced in Section 10.3. Matlab provides the function `vision.BlobAnalysis` for this situation.

The Matlab movie `atrium.avi` shows well that this type of mere motion detection works well. It suffices however only for a coarse estimation of events: it is sufficient to estimate the number of people and follow their paths coarsely. Looking more closely, we observe that there are potential confusions, for instance when two persons cross their paths. Or if a person stands still, then the corresonding track might disappear.

14.2 Maintaining Tracks

FoPo p357, s 11.1.1

After we have found moving regions in individual frames, then we try to link them up to obtain the trajectory. If there is only one object to pursue, then that linking is not very difficult. Even if the object is not detected in each frame - due to noise or occlusion for example -, then we can manage the pursuit by keeping a record of previous movements in a so-called *track*. In the first frame, we initiate the track by assigning the detected object. In the second frame, we match the new detection with the track of the previous frame. To facilitate continued tracking, it helps to store for how long the object had appeared in previous frames.

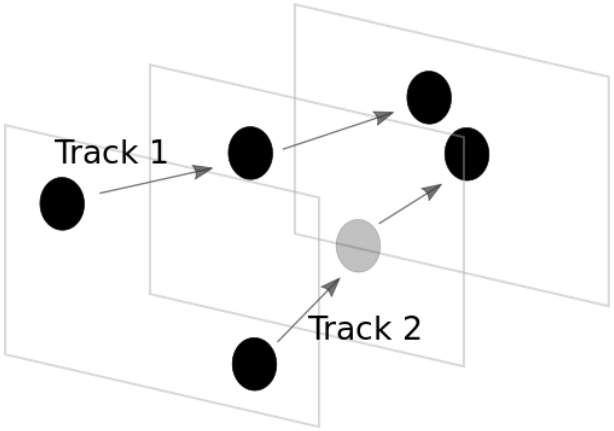

Figure 27: Pursuing multiple objects using *tracks*. Three frames are shown with two dots converging; one dot is not detected in the 2nd frame (shown as gray). In such situations is necessary to work with records, called tracks, and find the corresponding matches from frame to frame, also known as a bipartite graph matching problem.

Tracking becomes more challenging with multiple objects (Fig. 27). Now we need to sort out, which ones are most likely to match and the obvious way is to take the spatial distance as a measure. We create a distance matrix in which we measure the distance between the last locations of the previous tracks with the locations of the new object detections of the present frame. Then we take the minimum. That assignment is complicated by the occasional lack or sometimes abundance of detections. In that case we require a more sophisticated scheme: the tracks from the previous frame are copied to the next frame, and then the new object detector responses are allocated to the tracks. Each track will be assigned 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.

Algorithm 6 Tracking multiple objects (or tracking with unreliable object detector). i=time; t=track.

Notation:

Write $\mathbf{x}_k(i)$ for the k'th response of the detector in the ith frame
Write $t(k,i)$ for the k'th track in the ith frame
Write $*t(k,i)$ for the detector response attached to the k'th track in the ith 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).

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 χ-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 (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, 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.

A full Matlab example is given in Appendix **L.12.2**. That example includes a Kalman filter to predict a track's location for the next frame.

14.3 Face Tracking / CAM-Shift

We now introduce the first example of tracking-by-matching. As explained, in face tracking detection of moving regions alone is not sufficient: it requires something more specific. First, one performs face localization with an algorithm as introduced in Section 6.1: this is done in the first frame. In a subsequent frame, we want to track only the face region. We could therefore apply the face detector again just in a small neighborhood around the location of the detected face of the first frame. With todays computing power that is certainly doable and is in fact the idea of one type of tracking algorithm (next section), but here we introduce a succesful, venerable technique, to illustrate its elegance. The idea is to generate an intensity histogram of the face and to track just that histogram in subsequent frames. This is computationally simpler, than trying to match entire neighborhoods. The histograms show characteristic peaks which we match across frames with the principle of mean-shift. The mean-shift is technique for clustering data points. It works similar to

the K-Means algorithm, which uses distance measurements between points to find clusters. The mean-shift use gradients instead and that has proven to be quite effective in tracking faces.

Because a user might change his seating position - in particular if sitting in front of a home PC - tracking the initial histogram only is not reliable. One therefore adapts the entire procedure to the changes throughout the entire recording, analogous to the monitoring of the background for background subtraction (Section 14.1). This leads us to the CAM-Shift algorithm, the continuously adapting mean-shift algorithm. An example is shown in **L.12.3**. Matlab calls that tracker `HistogramBasedTracker`.

Face tracking nowadays is done with more capable detectors of course, introduced next.

14.4 Tracking by Matching and Beyond

FoPo p360, s11.1.2 or s11.2

Sze p337, s8.1, pdf384

Tracking by matching intensity histograms fails when we are faced with 'wilder' scenarios, for example, when objects are too small on the screen, or when their size changes substantially, for instance if the object is oncoming as in case of an autonomous vehicle tracking a pedestrian. There exists a large range of more sophisticated matching methods, but the essence remains the same: one searches over a small neighborhood to keep the computations at a possible minimum. Examples: some methods use Local Binary Patterns (LBP) as mentioned in Section 4.3.3; some use HOG features as introduced in Section 6.3; or they use other features as introduced in 7.1.

More ambitious trackers include learning. The object itself is learned at the beginng, such as in the boosting tracker; and/or the object itself is relearned during tracking as it might change its appearance during the motion. The most capable trackers are most likely the ones based on DeepNets - they come at the price of heavy resources; for a DeepNet one needs to store easily hundreds of MB of weights. Thus there is always a speed-accuracy tradeoff: the more robust a tracker the slower is its operation.

In Python we can initiate various trackers in OpenCV with functions called `TrackerXXX_create()` and then calling its method `init`, for example:

```
Trkr    = cv.TrackerKCF_create()
bOk     = Trkr.init(Frm, bbox)
```

where variable `bbox` is the bounding box of our starting position. In the loop cycling through the frames, we update the tracker by calling the method `update`:

```
bOk, bbox = Trkr.update(Frm)
```

A full example is given in **L.12.4**.

14.5 Optimization and Increasing Precision

Tracking can be optimized by using the vector field measurements to predict where the object is likely going to be in the next frame and to exploit this prediction to reduce the search effort. This optimization can be regarded as a two-step process (Fig. 29). In the prediction step, the filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement (necessarily corrupted with some amount of error, including random noise) is observed, these estimates are updated using a weighted average, with more weight being given to estimates with higher certainty. Because of the algorithm's recursive nature, it can run in real time using only the present input measurements and the previously calculated state; no additional past information is required.

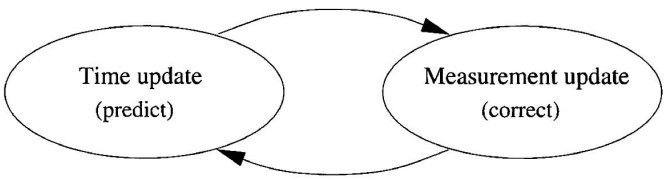

Figure 28: The predictor-corrector iterative cycle; the time update predicts events at the next step, and the measurement update adjusts estimates in the light of observation. [Source: Forsyth/Ponce 2010; Fig 16.27]

There are different methods to pursue this optimization specifically:

Kalman Filters : theoretically well elaborated, but not applicable to all real-world situations because it assumes a unimodal density distribution. The main assumption of the Kalman filter is that the underlying system is a linear dynamical system and that all error terms and measurements have a Gaussian distribution (often a multivariate Gaussian distribution).

Particle Filters : allows dealing with multi-modal density distributions as no restrictive assumption about the dynamics of the state-space or the density function to be estimated are made; in computer vision known as condensation algorithm (in AI: survival of the fittest).

14.5.1 Kalman Filters

SHB p793, s 16.6.1

FoPo p369, s 11.3, pdf 339

Kalman filters are based on a linear update, e.g.

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{w}_t \tag{22}$$

where $\mathbf{x}_t$ and $\mathbf{x}_{t-1}$ are the current and previous state variables, $\mathbf{A}$ is the linear transition matrix, and $\mathbf{w}$ is a noise (perturbation) vector, which is often modeled as a Gaussian (Gelb 1974). The matrices $\mathbf{A}$ and the noise covariance can be learned ahead of time by observing typical sequences of the object being tracked (Blake and Isard 1998). We here summarize the entire model following the book by SHB, whereby k now represents time.

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}_k\mathbf{x}_k + \mathbf{w}_k, \\ \mathbf{z}_k &= H_k\mathbf{x}_k + \mathbf{v}_k. \end{aligned} \tag{23}$$

$\mathbf{A}_k$ describes the evolution of the underlying model state
$\mathbf{w}_k$ is zero mean Gaussian noise with assumed covariance $Q_k = E[\mathbf{w}_k\mathbf{w}_k^T]$
H_k is the measurement matrix, describing how the observations are related to the model
$\mathbf{v}_k$ is another zero mean Gaussian noise factor, with covariance $R_k = E[\mathbf{v}_k\mathbf{v}_k^T]$

The Kalman gain matrix:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + R_k)^{-1} \tag{24}$$

Covariances:

$$P_k^- = A_k P_{k-1}^+ A_k^T + Q_{k-1} \tag{25}$$

$$P_k^+ = (I - K_k H_k) P_k^-. \tag{26}$$

14.5.2 Particle Filters

Sze p243, s 5.1.2, pdf 276

SHB p798, s 16.6.2

FoPo p380, s 11.5, pdf 350

Strictly, a particle filter is a sampling method that approximates distributions by exploiting their temporal structure; in computer vision they were popularized mainly by Isard and Blake in their CONditional DENSity propagATION algorithm, short CONDENSATION.

Particle filtering techniques represent a probability distribution using a collection of weighted point samples, see upper graph in Fig. 29. To update the locations of the samples according to the linear dynamics (deterministic drift), the centers of the samples are updated according to and multiple samples are generated for each point (lower graph in Figure). These are then perturbed to account for the stochastic diffusion, i.e., their locations are moved by random vectors taken from the distribution of $\mathbf{w}$. Finally, the weights of these samples are multiplied by the measurement probability density, i.e., we take each sample and measure its likelihood given the current (new) measurements.

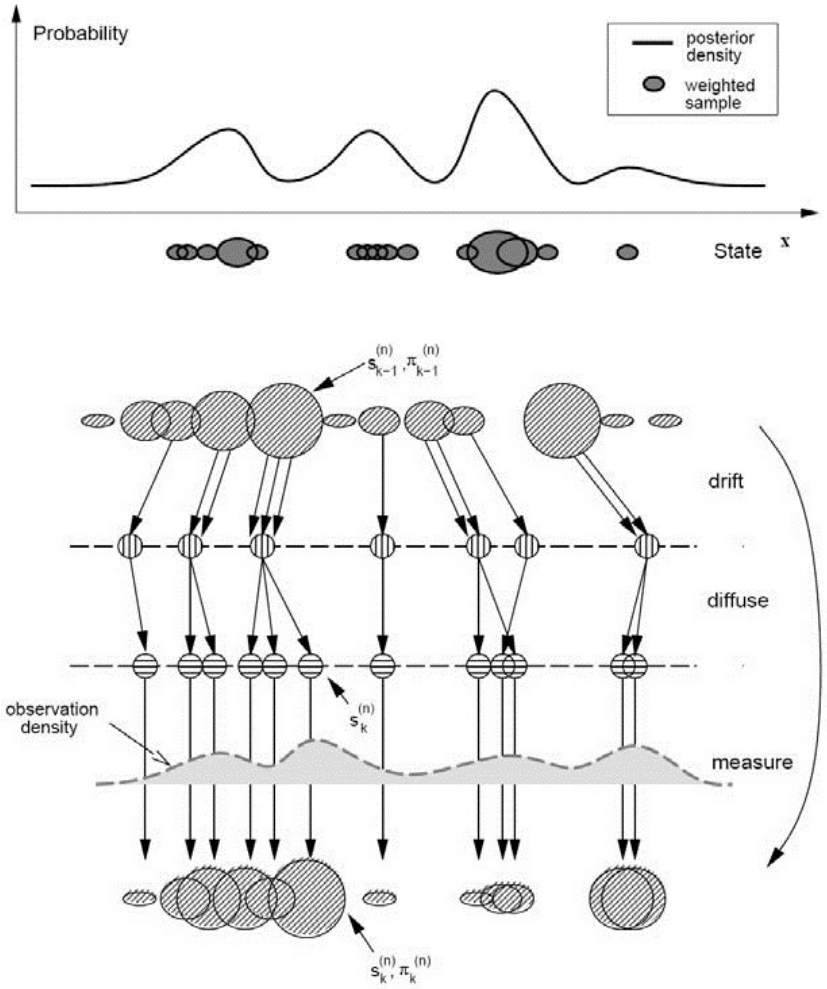

Figure 29: Top: each density distribution is represented using a superposition of weighted particles. Bottom: the drift-diffusion-measurement cycle implemented using random sampling, perturbation, and re-weighting stages. [Source: Szeliski 2011; Fig 5.7]

15 Optic Flow (Motion Estimation I)

Dav p506, s19.2
Sze p360, s8.4
FoPo p343, s10.6, pdf 313
SHB p757, s16.2

Optic flow is the estimation of motion between two successive frames, resulting in a pattern of motion vectors that is also called the *vector flow-field*. This is a first step toward understanding *how* an object or scene has moved, see again Section 1.2. In some sense, optical flow is tracking of the whole scene or of an object's individual parts or regions; and for that reason it is computationally very intensive. Optical flow is computed for video compression, in micro air vehicles, in robots and sometimes it is exploited for tracking.

Optical flow is particularly interesting for *egomotion*, when the camera (observer) moves. Figure 30 shows two examples:

Forward Looking (a) the observer moves forward and looks forward. This creates a radial flow pattern as objects will move to the periphery of the visual field, away from the vanishing point. Near objects move faster than far objects - if they are fixed in position (or slowly moving) - and result in longer vectors.

Side-Ways Looking (b) the observer moves forward but looks toward one side, i.e. turning the visual sensors 90 degrees. This creates a horizontal flow pattern with near objects again generating longer vectors than far objects.

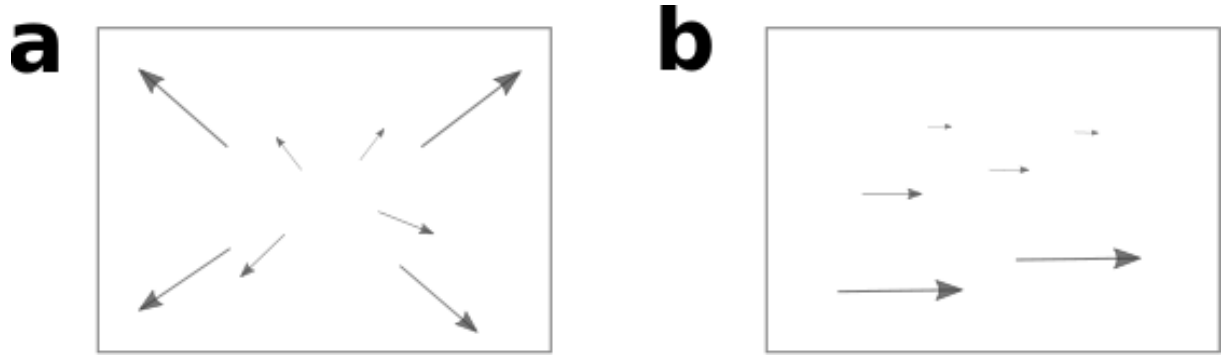

Figure 30: Optical flow fields are velocity flow fields typically measured for the entire scene for egomotion (camera moves), but can be equally interesting for objects. Long vectors correspond typically to objects near the observer.
(a) Optic flow as generated when moving forward (and looking forward), i.e. driving a car.
(b) Optic flow as generated when looking to one side while moving forward, i.e. observing the landscape from a train.

Such flow patterns are also created if an object moves and the observer is stationary, one may consider them to be local flow fields. For example, if an object moves toward the observer, then this motion will create a radial flow pattern as in **a**, but one that is confined to the object's silhouette. If an object moves side-ways, then this will create a flow pattern as in **b**, again only at points of the object.

Optical flow can be calculated with a variety of methods and - one may have guessed it already - those methods are similar to the ones for tracking. Optical flow can be calculated pixel-by-pixel, which is computationally intensive but precise; it can be calculated feature-by-feature, which is much faster; etc. We sketch some prominent, distinct approaches:

Block-Based: here one searches in a small neighborhood, analogous to the search for the object in tracking, see again Fig.26b. One minimizes the sum of squared differences or sum of absolute differences (Appendix E), or one maximizes the normalized cross-correlation. It is perhaps the simplest approach, but also a costly one.

Differential, Global: this method calculates optical flow for the entire image, in some sense at once, hence the term global. It does so by using the two derivatives of an image, in each direction, akin to calculating gradients in Section 3.3. Its most famous representative method is the Horn-Schunck algorithm, abbreviated HS in software packages: it yields a high density of flow vectors, i.e. the flow information missing in inner parts of homogeneous objects is filled in from the motion boundaries. On the downside, the algorithm is sensitive to noise. wiki Horn-Schunck_method

Differential, Local: is a method also based on the image derivatives, and hence the same term 'differential'. But unlike the global method, it partitions the image to avoid the sensitivity to noise. The best known method for this approach is the Lucas-Kanade method, abbreviated LK in software packages. The method cannot provide flow information in the interior of uniform regions. wiki Lucas-Kanade_method

And there exist of course combinations of local and global methods that attempt to combine the advantages of both methods. The OpenCV library contains a large set of implementations. Optical flow algorithms can be compared with the following sequences for instance:

`http://vision.middlebury.edu/flow/`
`http://of-eval.sourceforge.net/`

In Matlab we can calculate optical flow with two functions: one for initiating a structure whose name starts with `opticalFlow`; and one function for updating the flow-field computation during looping, called `estimateFlow`. A full example is given in Appendix **L.13**.

In Python we can access some of the methods through OpenCV, whose function names start with `calcOptialFlow`, for instance:

```
import cv2 as cv
Pts1, St, err = cv.calcOpticalFlowPyrLK(Iold, Inew, Pts0, None, **PrmOptFlo)
```

where `Iold` is the previous frame, `Inew` is the next frame, `Pts0` are a list of feature coordinates from the previous frame and `PrmOptFlo` is a list of parameters for that algorithm (full example in Appendix **L.13**).

16 Alignment (Motion Estimation II)

FoPo p397, ch 12, pdf 446

Sze p273, ch 6, pdf 311

Alignment is the process of estimating object or camera motion, which is typically carried out after the object or observer has completed its movement. It is therefore rather a reconstruction process, as opposed to tracking or calculating optic flow that can be considered continuous processes designed for monitoring (following) in some sense. In alignment, one is interested in how the object or observer has moved precisely between two distant moments, in some sense between two frames separated by some duration.

The motion can be estimated with various methods. In this section, we pursue an estimation that uses the analytical, geometric approach, namely by finding a geometric transformation for the motion. In mathematics one speaks of how the object *transformed*: we seek the transform that describes the motion as informative and precise as possible. Figure 31 shows the common 2D transformations.

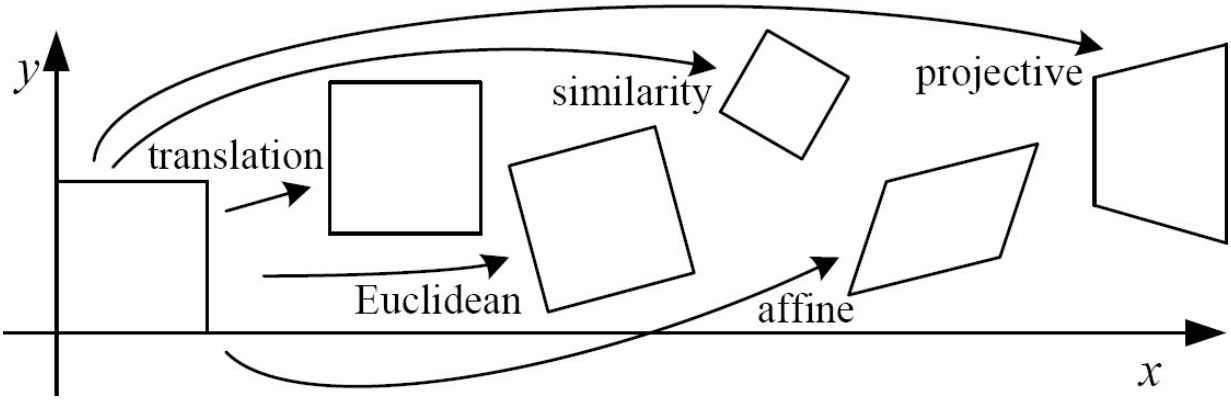

Figure 31: Basic set of 2D planar transformations. Used for estimating object motion (fixed observer) or camera motion (observer moved). The square shape at the origin is also called *moving* image/points or source; the transformed shapes are the target, *fixed* or senses image/points.
translation: a shift in position.
Euclidean: translation and rotation, also known as rigid body motion.
similarity: translation, rotation and scaling.
affine: including distortion.
projective: including change of viewpoint.
[Source: Szeliski 2011; Fig 3.45]

As implied in the introduction already, there are two principal scenarios where this motion estimation is calculated:

Object Motion: is the reconstruction of how precisely an object has moved, we attempt to match the two poses by some transformation.

Camera Motion: is the reconstruction of how the camera (observer) has moved. In other words, we are given two images of some scene or object, but taken from different viewpoints. We estimate the observer (camera) motion; we estimate the observer's new viewpoint.

Both scenarios are sometimes more generally referred to as pose estimation, which however should not be confused with posture estimation for humans (Section 18.1).

The methods to estimate the transforms are analogous to the methods for estimating optical flow. We operate with a vector flow-field and we can attempt to make the estimation with all pixels, or with a subset of pixels selected using feature detectors. If done with features, then we also need to determine which points or features correspond to each other, an issue also know as the *correspondence problem*. Once we have established that correspondence, then we can determine the exact motion parameters by using geometric transformations. This will be introduced later; for the moment we continue the introduction of the geometric transformations.

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 Fig. 31. But the object may have also rotated during the *motion*, in which case the motion corresponds 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*. There are more complex transformations such as *affine* and *projective*, which reconstruct distortion and change in perspective, respectively. However from those we cannot easily infer translation, rotation and scaling anymore, although those three are very informative and intuitive.

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 some example applications:

Shape Analysis: A simple 2D application is to locate shapes, for instance finding one shape in another image as employed in biology, archeology or robotics. If the transformations involve translation, rotation, scaling and reflection, then it is also called *Procrustes* analysis wiki Procrustes_analysis.

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.

In the following section we explain how some of the 2D motions of Figure 31 are expressed mathematically (Section 16.1). Then we learn how to estimate them (Section 16.2), assuming that the correspondence problem was solved. Finally, we learn how to robustly estimate them, that is even in the presence of noise and clutter when the correspondence problem is a challenge (Section 16.3). At the very end, we have a few words on the topic of image registration (Section 16.4).

16.1 2D Geometric Transforms

Sze p33, s 2.1.2, pdf 36

Figure 31 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 $\mathbf{t}$ is added to the points in $\mathbf{x}$. 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 2×3 matrix, see also Figure 32.

2D Euclidean Transform: consist of a rotation and translation. The rotation is achieved with the a so-called orthonormal rotation matrix $\mathbf{R}$ whose values are calculated by specifying the rotation angle θ. In notation 2, the single matrix is concatenated as before and it remains a 2×3 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 2×3.

In Appendix **L.14** we show how the transforms are carried on a set of points (the first section in that script). The transformations can also be applied to entire images. That is what is frequently done for data augmentation when training neural networks (Section 5).

In Matlab there exists single functions to carry out individual transformations such as `imtranslate`, `imrotate` and `imresize`; `imcrop` is useful too for data augmentation. More complex transformations can be generated with functions `imtransform`, for which we prepare the transformation matrices in `maketform` for instance. The function script `imwarp` is a more general form for generating transformations.

In Python all those functions are found in `skimage.transform`.

Transform	Notation 1	Notation 2	Comments
Translation	$\mathbf{x}' = \mathbf{x} + \mathbf{t}$	$\mathbf{x}' = \begin{bmatrix}\mathbf{I} & \mathbf{t}\end{bmatrix}\mathbf{x}$	$\mathbf{I}$ is the 2×2 identity matrix
Rotation + Translation = *2D rigid body motion* = *2D Euclidean transformation*	$\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$	$\mathbf{x}' = \begin{bmatrix}\mathbf{R} & \mathbf{t}\end{bmatrix}\mathbf{x}$	$\mathbf{R} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$ = orthonormal rotation matrix: $\mathbf{R}\mathbf{R}^T = \mathbf{I}$ and $\lvert\mathbf{R}\rvert = 1$ Euclidean distances are preserved.
Scaled Rotation =*Similarity Transform*	$\mathbf{x}' = s\mathbf{R}\mathbf{x} + \mathbf{t}$	$\mathbf{x}' = \begin{bmatrix}s\mathbf{R} & \mathbf{t}\end{bmatrix}\mathbf{x}$ $\mathbf{x}' = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \end{bmatrix}\mathbf{x}$	no longer requires $a^2 + b^2 = 1$
Affine	$\mathbf{x}' = \mathbf{A}\mathbf{x}$	$\mathbf{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}\mathbf{x}$	Parallel lines remain parallel

In PyTorch all those functions are packed into `torchvision.transforms`.

16.2 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 us 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 $\{\mathbf{x}_i, \mathbf{x}'_i\}$ and a chosen planar transformation $\mathbf{f}$ with parameters $\mathbf{p}$ (e.g. $\mathbf{t}, \mathbf{R}$...),

$$\mathbf{x}' = \mathbf{f}(\mathbf{x}; \mathbf{p}), \tag{27}$$

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

$$E_{LS} = \sum_i \|\mathbf{r}_i\|^2 = \sum_i \|\mathbf{f}(\mathbf{x}_i; \mathbf{p}) - \mathbf{x}'_i\|^2, \tag{28}$$

where

$$\mathbf{r}_i = \mathbf{x}'_i - \mathbf{f}(\mathbf{x}_i; \mathbf{p}) = \hat{\mathbf{x}}'_i - \tilde{\mathbf{x}}'_i \tag{29}$$

is the residual between the measured location $\hat{\mathbf{x}}'_i$ and its corresponding current predicted location $\tilde{\mathbf{x}}'_i = \mathbf{f}(\mathbf{x}_i; \mathbf{p})$.

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

$$\Delta\mathbf{x} = \mathbf{x}' - \mathbf{x} = \mathbf{J}(\mathbf{x})\mathbf{p}. \tag{30}$$

where $\mathbf{J} = \partial\mathbf{f}/\partial\mathbf{p}$ is the Jacobian of the transformation $\mathbf{f}$ with respect to the motion parameters $\mathbf{p}$. $\mathbf{J}$ is shown in figure 32 and has a particular form for each transformation. In this case, a simple *linear* regression (linear-least-squares problem) can be formulated as

$$E_{LLS} = \sum_i \|\mathbf{J}(\mathbf{x}_i)\mathbf{p} - \Delta\mathbf{x}_i\|^2, \tag{31}$$

which - after some reformulation - equates to

$$\mathbf{p}^T\mathbf{A}\mathbf{p} - 2\mathbf{p}^T\mathbf{b} + c. \tag{32}$$

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

$$\mathbf{A}\mathbf{p} = \mathbf{b}, \tag{33}$$

where

$$\mathbf{A} = \sum_i \mathbf{J}^T(\mathbf{x}_i)\mathbf{J}(\mathbf{x}_i) \tag{34}$$

is called the Hessian and

$$\mathbf{b} = \sum_i \mathbf{J}^T(\mathbf{x}_i)\Delta\mathbf{x}_i. \tag{35}$$

Transform	Matrix	Parameters p	Jacobian J
translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix}$	(t_x, t_y)	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Euclidean	$\begin{bmatrix} c_\theta & -s_\theta & t_x \\ s_\theta & c_\theta & t_y \end{bmatrix}$	(t_x, t_y, θ)	$\begin{bmatrix} 1 & 0 & -s_\theta x - c_\theta y \\ 0 & 1 & c_\theta x - s_\theta y \end{bmatrix}$
similarity	$\begin{bmatrix} 1+a & -b & t_x \\ b & 1+a & t_y \end{bmatrix}$	(t_x, t_y, a, b)	$\begin{bmatrix} 1 & 0 & x & -y \\ 0 & 1 & y & x \end{bmatrix}$
affine	$\begin{bmatrix} 1+a_{00} & a_{01} & t_x \\ a_{10} & 1+a_{11} & t_y \end{bmatrix}$	$(t_x, t_y, a_{00}, a_{01}, a_{10}, a_{11})$	$\begin{bmatrix} 1 & 0 & x & y & 0 & 0 \\ 0 & 1 & 0 & 0 & x & y \end{bmatrix}$

Figure 32: Jacobians of the 2D coordinate transformations $\mathbf{x}' = \mathbf{f}(\mathbf{x}; \mathbf{p})$ (see table before), where we have re-parameterized the motions so that they are identity for $\mathbf{p} = 0$. [Source: Szeliski 2011; Tab 2.1]

Practically, we write a loop that takes each point and calculates the dot product and accumulates them to build $\mathbf{A}$. Then we call a script that does the least-square estimation. This is shown in the second part in the code of **L.14**. The function `lsqlin` carries out the least-square estimation. We also show how to use Matlab's function scripts `fitgeotrans` that does both in a single script - building $\mathbf{A}$ and $\mathbf{b}$, as well the estimation of the motion parameters. In that code we also show how to use the function script `procrustes`, which does also everthing at once.

When using functions in software packages the preferred terminology is as follows: the first image is referred to as the *moving* image (or points), or the *source*; the second image is referred to as the *target*, *fixed* or *sensed* image (points).

In Python , the procrustes analysis can be found in module `scipy.spatial`. I am not aware of a single script in Python, but OpenCV provides functions that do the entire alignment process.

16.3 Robust Alignment with RANSAC

FoPo p332, s 10.4.2, pdf 30
Sze p281, s 6.1.4, pdf 318
Pnc p342, s 15.6
SHB p461,s.10.2

As pointed out already, in complex situations it is not straightforward to establish the correspondence between pairs of points. Often, when we compare two images - or two shapes in different images - we find only part of the shape points and sometimes wrong points or features were determined. One can summarize such 'misses' as outliers and noise, and they make motion estimation in real world applications difficult. It therefore requires more robust estimation techniques, for instance an iterative process consisting of the following two steps:

1) selection of a random subset of points and estimation of their motion.

2) verification of the estimate with the remaining set of points.

Example: We are fitting a line to an elliptical point cloud that contains about 50% outliers - or that is embedded in random noise. 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.

Algorithm 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 $\mathbf{p}$. The *residuals* of the full set of correspondences are then computed as

$$\mathbf{r}_i = \tilde{\mathbf{x}}'_i(\mathbf{x}_i; \mathbf{p}) - \hat{\mathbf{x}}'_i, \tag{36}$$

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

Algorithm 7 RANSAC: Fitting structures using Random Sample Consensus. FoPo p332, pdf 305

Input : $\mathcal{D}$, $\mathcal{D}^*$
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 $\mathcal{D}$ uniformly and at random $\rightarrow \mathcal{D}^s$; $\mathcal{D}^c = \mathcal{D} \setminus \mathcal{D}^s$
- Fit to $\mathcal{D}^s$ and obtain estimates $\mathbf{p}$
- **For** each data point $\mathbf{x} \in \mathcal{D}^c$: if point close,
that is smaller than a threshold: $\|\mathbf{r}_i\|^2 < t$, then $\mathbf{x}_i \rightarrow \mathcal{D}^{good}$
end
- If there are d or more points close to the structure ($|\mathcal{D}^{good}| \geq d$), then $\mathbf{p}$ is kept as a good fit.
- refit the structure using all these points.
- add the result to a collection of good fits $\rightarrow \mathcal{P}^{good}$
end
Choose the best fit from $\mathcal{P}^{good}$, using the fitting error as a criterion

The RANSAC technique then counts the number of inliers that are within ϵ of their predicted location, i.e., whose $\| \mathbf{r}_i \| \leq \epsilon$. The ϵ 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 $\mathbf{p}$ or the full set of computed inliers is then passed on to the next data fitting stage.

Matlab's computer vision toolbox provides a function. In Appendix **L.15** we provide a primitve version to understand it step by step. Python offers the function `ransac` in submodule `skimage.measure`.

16.4 Image Registration

In image registration we aim at aligning entire scenes. Registration can become very complex when objects deform, which is particularly the case in medical image analysis: human organs deform and are often scanned with different scanning modes, the latter aggravating the registration effort.

There exists a large range of techniques, each one with advantages and disadvantages. Those techniques can be classified according to different aspects. One distinction is made based on the number of points selected for registration. In feature-based methods, one detects characteristic features as introduced

in Section 7 and then estimates the motion using RANSAC for instance. That would be a registration based on a subset of points. In intensity-based methods, one uses all pixel values at hand - the entire image - and that can be more accurate but also requires more computation, analogous to the computation of optical flow.

In Matlab, image registration can be carried out with `imregister`. In Python, one rather uses functions from OpenCV.

17 3D Vision

SHB p546, ch11, ch12

3D Vision is the task to perceive depth, the third dimension of space. Depth perception is useful, for example, for industrial robots reaching for tools and for autonomous vehicles trying to avoid obstacles. Depth information is also exploited in gesture recognition sometimes, with the most famous example being the Kinect system.

Depth information can be obtained with different cues. One can try to obtain it from the two-dimensional image only by elaborate reconstruction. Or one can simply use a camera that senses depth directly, a technique called *range imaging*, see again Appendix A. Each cue and technique has advantages and disadvantages. We here mention only the popular ones, of which the first one, stereo correspondence, uses two-dimensional information only:

Stereo Correspondence: takes two or more images taken from slightly different positions and then determines the offset (disparity) between the images - a process also done by the human visual system to estimate depth. To determine disparity one needs to find the corresponding pixels across images, a process that is relatively challenging. The advantage of the technique is that is can be carried out with cheap, light-sensitive (RGB) cameras. We elaborate on that method in Section 17.1.

Lidar (Light Detection And Ranging): is a surveying method that measures distance to a target by illuminating the target with a pulsed laser light, and measuring the reflected pulses with a sensor. It typically works for a specific depth range. This sensor is particularly used for autonomous vehicles and in order to cover the entire depth of the street scene, one employs several Lidar sensors each one tuned for specific depth.

Time-of-Flight (ToF): a relatively new technique that uses only a single pulse to scan the environment - as opposed to Lidar (or Radar). Early implementations required relatively much time to compute the distance - as opposed to Lidar and radar -, but recent implementations can provide up to 160 frames per second.

The output of such a technique is typically a two-dimensional map, in which the intensity values correspond to depth measured in some unit (i.e. centimeters). That map is also called *depth map*, or *range map* if obtained with range imaging. The analysis of such a depth map does not pose any novel computational challenges: to obtain a segragation between foreground and background one can use the segmentation techniques as introduced previously. For that reason there is not more to explain here and in the following we merely elaborate on the process of stereo correspondence as that is an algorithmic issue.

Depth information can also be obtained from single images in principle - called *shape from X* sometimes -, but that has not been as practical yet as the above techniques. We mention those methods nevertheless (Section 17.2).

17.1 Stereo Correspondence

Sze p469, ch11, pdf533

SHB p573, s11.5, s11.6.1

FoPo p227, ch7

Stereopsis is the perception of depth using two (or more) images taken from slightly different viewpoints. The corresponding points between the two images have a slight, horizontal offset (motion), so-called *disparity*, which is inverse proportional to depth. Thus the main challenge is to determine the (stereo) correspondence, which is usually solved by assuming some constraints, see SHB for a nice overview (SHB p584, s11.6.1). Stereo correspondence algorithms can be divided into two groups:

- Low-level, correlation-based, bottom-up, dense correspondence

- High-level, feature-based, top-down, sparse correspondence. Features are for instance the ones introduce in Section 7.

Epipolar Constraint A popular constraint to compute stereo correspondence, see Fig. 33. Pixel $\mathbf{x}_0$ in the left image corresponds to epipolar line segment $\overline{\mathbf{x}_1\mathbf{e}_1}$ in the right image, and vice versa for $\mathbf{x}_1$ and $\overline{\mathbf{x}_0\mathbf{e}_0}$. Both segments form a plane (right graph).

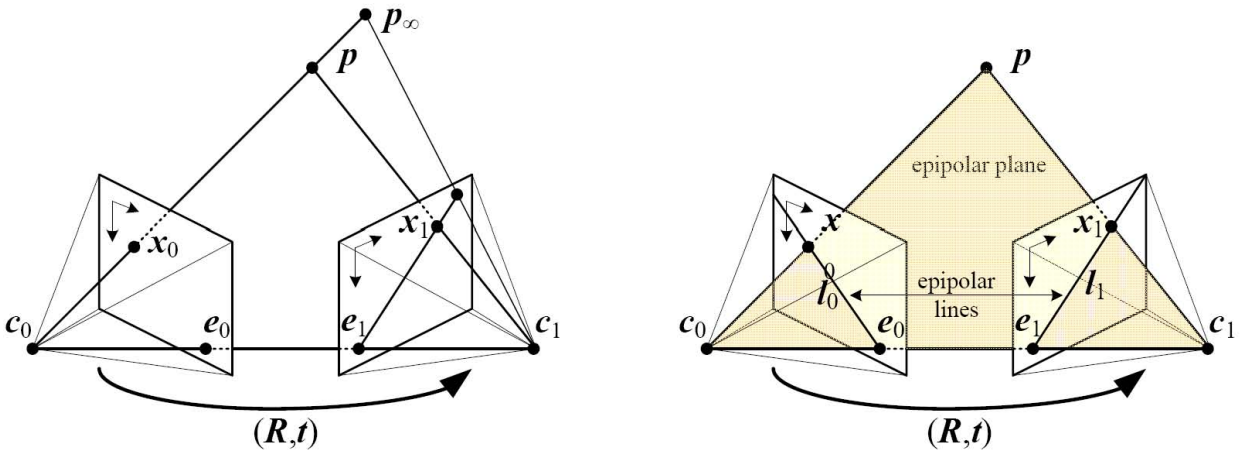

Figure 33: Epipolar geometry. $\mathbf{c}$ camera center; $\mathbf{e}$ epipole. **Left**: epipolar line segment corresponding to one ray; **Right**: corresponding set of epipolar lines and their epipolar plane. [Source: Szeliski 2011; Fig 11.3]

The epipolar constraint in algebraic form:

$$\mathbf{x}_0^T F \mathbf{x}_1 = 0, \tag{37}$$

where T stands for the transpose. Matrix F is called the **fundamental matrix**, a slightly misleading name widely used for historical reasons; more appropriate names like **bifocal matrix** are used by others.

SHB p578 Epipolar geometry has seven degrees of freedom. The epipoles $\mathbf{e}, \mathbf{e}'$ in the image have two coordinates each (giving 4 degrees of freedom), while another three come from the mapping of any three epipolar lines in the first image to the second. Alternatively, we note that the nine components of F are given up to an overall scale and we have another constraint $\det F = 0$, yielding again $9 - 1 - 1 = 7$ free parameters.

The correspondence of seven point in left and right images allows the computation of the fundamental matrix F using a non-linear algorithm [Faugeras et al.,1992], known as the **seven-point** algorithm. If eight points are available, the solution becomes linear and is known as the **eight-point** algorithm.

17.2 Shape from X

Sze p508, s12.1

SHB p606, s12.1

Shape from X is a generic name for techniques that aim to extracting shape from intensity images. Many of these methods estimate local surface orientation (e.g., surface normal) rather than absolute depth. If, in addition to this local knowledge, the depth of some particular point is known, the absolute depth of all other points can be computed by integrating the surface normals along a curve on a surface [Horn, 1986]. There exist:

- from Stereo: as mentioned above.
- from Shading: see below.
- from Photometric Stereo: a way of making 'shape from shading' more reliable by using multiple light sources that can be selectively turned on and off.
- from Motion: e.g. optic flow.
- from Texture: see below.
- from Focus: is based on the fact that lenses have finite depth of field, and only objects at the correct distance are in focus; others are blurred in proportion to their distance. Two main approaches can be distinguished: shape from focus and shape from de-focus.
- from Contour: aims to describe a 3D shape from contours seen from one or more view directions.

17.2.1 Shape from Shading

FoPo p89, s2.4
Dav p398, s15.4

Albedo (latin for "whiteness"): or reflection coefficient, is the diffuse reflectivity or reflecting power of a surface. It is the ratio of reflected radiation from the surface to incident radiation upon it. Its dimensionless nature lets it be expressed as a percentage and is measured on a scale from zero for no reflection of a perfectly black surface to 1 for perfect reflection of a white surface.

Algorithm Most shape from shading algorithms assume that the surface under consideration is of a uniform albedo and reflectance, and that the light source directions are either known or can be calibrated by the use of a reference object. Under the assumptions of distant light sources and observer, the variation in intensity (irradiance equation) become purely a function of the local surface orientation, which is used for instance for scanning plaster casts. In practice, surfaces are rarely of a single uniform albedo. Shape from shading therefore needs to be combined with some other technique or extended in some way to make it useful. One way to do this is to combine it with stereo matching (Fua and Leclerc 1995) or known texture (surface patterns) (White and Forsyth 2006). The stereo and texture components provide information in textured regions, while shape from shading helps fill in the information across uniformly colored regions and also provides finer information about surface shape.

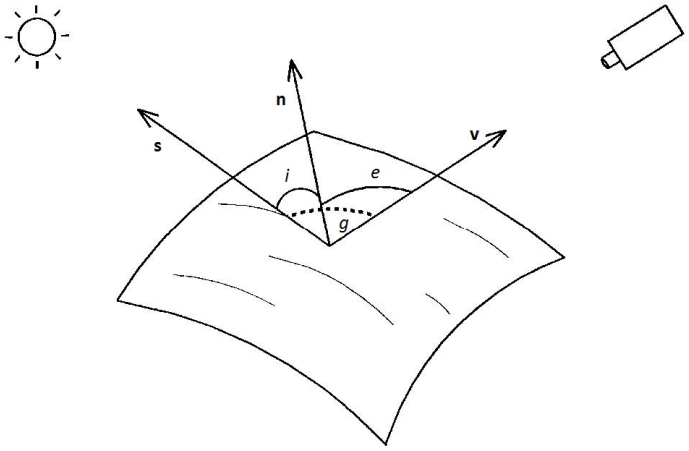

Figure 34: Geometry of reflection. An incident ray from source direction **s** is reflected along the viewer direction **v** by an element of the surface whose local normal direction is **n**; *i*, *e*, and *g* are defined respectively as the incident, emergent, and phase angles. [Source: Davies 2012; Fig 15.8]

17.2.2 Shape from Texture

FoPo p217, s6.5
Sze p510, s12.1.2
SHB p613, s12.1.2

The variation in foreshortening observed in regular textures can also provide useful information about local surface orientation. Figure 36 shows an example of such a pattern, along with the estimated local surface orientations. Shape from texture algorithms require a number of processing steps, including the extraction of repeated patterns or the measurement of local frequencies in order to compute local affine deformations, and a subsequent stage to infer local surface orientation. Details on these various stages can be found in the research literature (Witkin 1981; Ikeuchi 1981; Blostein and Ahuja 1987; Garding 1992; Malik and Rosenholtz 1997; Lobay and Forsyth 2006).

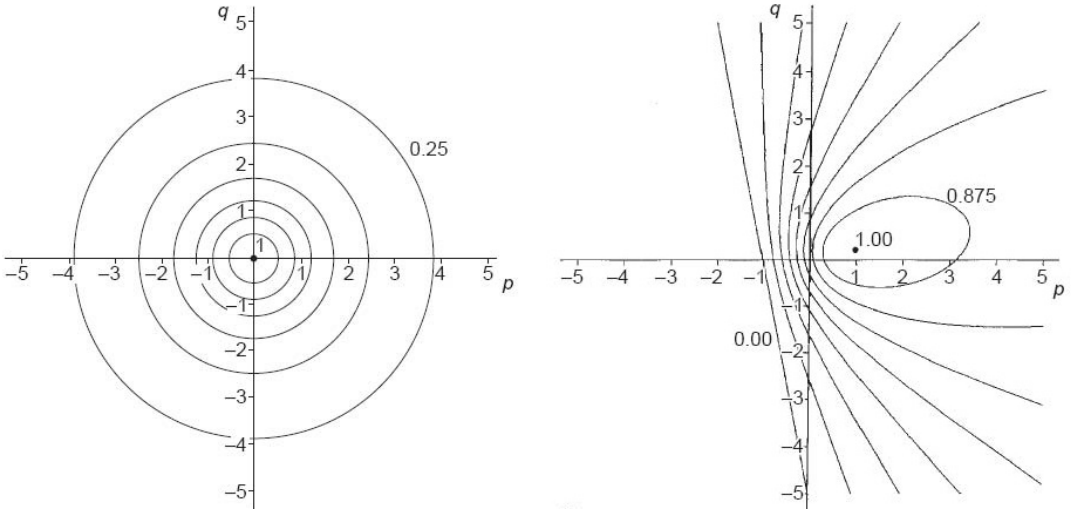

Figure 35: Reflectance maps for Lambertian surfaces: **Left**: contours of constant intensity plotted in gradient (p, q) space for the case where the source direction **s** (marked by a black dot) is along the viewing direction **v** (0, 0) (the contours are taken in steps of 0.125 between the values shown); **Right**: the contours that arise where the source direction (p_s, q_s) is at a point (marked by a black dot) in the positive quadrant of (p, q) space: note that there is a well-defined region, bounded by the straight line $1 + pp_s + qq_s = 0$, for which the intensity is zero (the contours are again taken in steps of 0.125). [Source: Davies 2012; Fig 15.9]

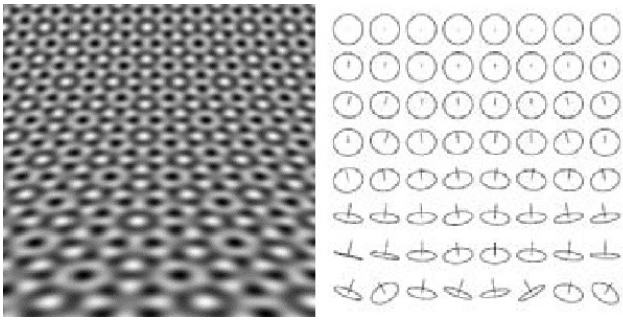

Figure 36: Shape from texture.
Left: the texture.
Right: corresponding estimations of the local surface normal.
[Source: Szeliski 2011; Fig 12.3]

18 Pose Estimation (of Humans and Objects)

Pose estimation is a general term for recognizing the position and/or posture of an object. There exists two specific tasks in particular. In pose estimation of humans, one is interested in the posture of a person, the detailed alignment of the body parts (Section 18.1). In pose estimation of objects, one estimates from which viewpoint in space the object is seen (Section 18.2). Such tasks used to be solved with feature point detection (Section 7): one would learn the configuration of key-points in a set of training images, and then find and match those points in the testing images. Such systems can be fairly complex. Nowadays, deep CNNs provide an easier way to train such poses: those networks are not less complex than traditional approaches, but they are easier to use and achieve better results - sometimes much better results.

18.1 Human Pose

Posture estimation is the task of detecting the configuration of a person's pose (Fig. 37): one localizes keypoints of a person's pose, the limbs or joints or major body parts, from which we then can estimate their precise alignment. There exists a variety of (traditional) approaches that often model explicit structural relations, ranging from few relations expressed deterministically to multiple relations expressed probabilistically. CNNs have taken the latter approach to the extreme and do not formulate any relations: instead those are learned automatically, in particular in the more global (upper) maps of the network hierarchy. The (presently) best performing network is called *OpenPose*, the corresponding article entitled 'Convolutional Pose Machines'.

Figure 37: Posture estimation, here focusing in particular on joints. Estimate generated by OpenPose, a deep CNN that was trained on thousands of poses. See **L.16**.

Figure 37 shows the output of that network. 15 key-points can be detected; with some network variants even more. With the same network architecture one can also train to estimate the pose of faces (Fig. 38), hands (Fig. 39), and in the near future feet:

https://github.com/CMU-Perceptual-Computing-Lab/openpose

The code in Appendix **L.16** shows how to feed a single image to the network and how to read out the key-points from the network output. The ordering of key-points can be looked up on:

https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/doc/output.md

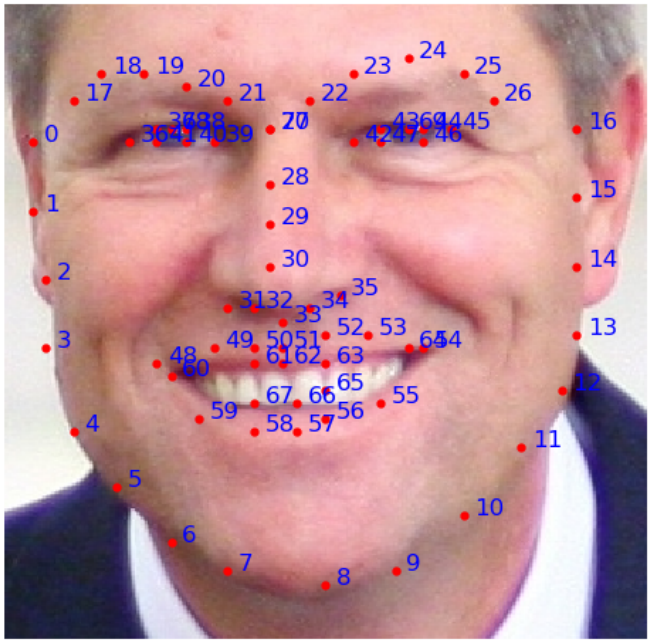

Figure 38: Face pose estimation, also estimated by the OpenPose network, trained on images of faces. See **L.16**.

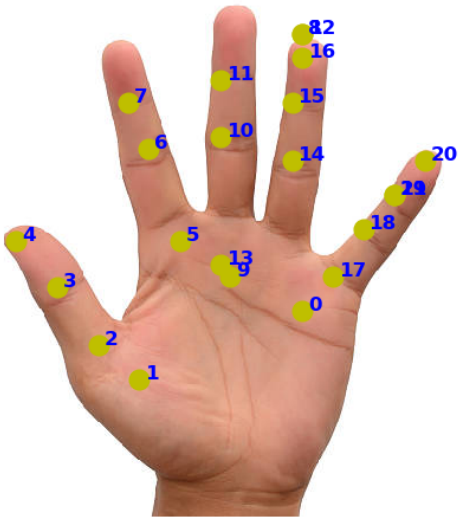

Figure 39: Hand pose estimation. Also estimated by the OpenPose system, trained on images containing hands. See **L.16**.

18.2 Viewpoint Estimation, Pose Estimation

Viewpoint estimation is the task of localizing the precise position of the observer in space by comparing the present viewpoint with previously stored viewpoints. For instance, we see a photo of a known building but the photo is taken from an unusual viewpoint: now we compare in our head the previously seen viewpoint with the one on the photo and infer the viewpoint. We can invert the problem formulation and say we

estimate the *pose* of the object in space for which we require a different reference point than the one of the current observer. This task is particularly used in robot navigation, where one attempts to learn a map of a robot's environment for easier navigation, a problem called *simultaneous localization and mapping* (SLAM).

In Computer Vision, the traditional approach to perform this localization is to detect features between the two photographs and then reconstruct the motion in order to make an estimate as precise as possible, as introduced in Sections 7 and 16, respectively. This is computationally intensive both in calculation as in memory requirements. A recent DNN has however achieved stunning results by solving that task quicker and with less memory. We sketch that network here.

19 Classifying Motions

Classifying motions is the challenge to understand a sequence of motions. First we track the motion, which can include several parts of one object, such as a moving person; then we classify the trajectories of the tracked objects and parts. Systems solving such tasks are typically complex systems - though there exist exceptions such as the Kinect user interface (Section 19.2). Two frequent tasks are gesture recognition and body motion classification.

One severe challenge with recognizing human motions is the segragation of the motion from its background. For the tracking task as introduced in Section 14, this segragation was a lesser problem as the objects in the scene are relatively small and embedded in a homogeneous background - at least phasewise and locally. But when following human motions, one typically zooms into the silhouette for reasons of resolution, and in that situation chances are higher that the gesture's background is heterogeneous, which in turn makes segragation more difficult: think of the background that a labtop camera faces when the user sits in his room at home. For that reason one often seeks additional information by using depth camera (Appendix A), but even with that depth information segragation remains challenging sometimes. It is therefore not surprising that recognizing human motions works best on a homogeneous background with fairly constant illumination. Skeptical voices even say that there still does not exist a convincing implementation of motion classification.

19.1 Gesture Recognition

The most common gestures that are analyzed are hand gestures and facial gestures, the latter often reflecting emotions. Hand motions are recognized for the purpose of human-computer interaction. A daily example is the unlocking of a cell phone. This works well because it is the tracking of the finger tip only on predefined positions, an orthogonal grid. It becomes much more challenging for recognizing hand motions in the air. The automotive industry tries to make gesture recognition work for the control of the dash board.

19.2 Body Motion Classification with Kinect

FoPo p476, s14.5, pdf446

Kinect is a video game technology developed by Microsoft that allows its users to control games using natural body motions. Kinect's sensor delivers two images: a 480x640 pixel depth map and a 1200x1600 color image. Depth is measured by projecting an infrared pattern, which is observed by a black-and-white camera. The two main features of this sensor are its speed - it is much faster than conventional range finders using mechanical scanning - and its low cost (only ca. 200 Euros).

Range images have two advantages in this context:

1) They relatively easily permit the separation of objects from their background, and all the data processed by the pose estimation procedure presented below, is presegmented by a separate and effective background subtraction module.
2) They are much easier to simulate realistically than ordinary photographs (no color, texture, or illumination variations). In turn, this means that it is easy to generate synthetic data for training accurate classifiers without overfitting.

19.2.1 Features

Kinect features simply measure depth differences in the neighborhood of each pixel. Concretely, let us denote by $z(\mathbf{p})$ the depth at pixel $\mathbf{p}$ in the range image. Given image displacements $\boldsymbol{\lambda}$ and $\boldsymbol{\mu}$, a scalar is computed as follows:

$$f_{\boldsymbol{\lambda},\boldsymbol{\mu}}(\mathbf{p}) = z\Big[\mathbf{p} + \frac{1}{z(\mathbf{p})}\boldsymbol{\lambda}\Big] - z\Big[\mathbf{p} + \frac{1}{z(\mathbf{p})}\boldsymbol{\mu}\Big] \tag{38}$$

In turn, given some allowed range of displacements, one can associate with each pixel $\mathbf{p}$ the feature vector $\mathbf{x}(\mathbf{p})$ whose components are the D values of $f_{\boldsymbol{\lambda},\boldsymbol{\mu}}(\mathbf{p})$ for all distinct unordered pairs $(\boldsymbol{\lambda}, \boldsymbol{\mu})$ in that range.

As explained below, these features are used to train an ensemble of decision tree classifiers, in the form of a random forest. After training, the feature $\mathbf{x}$ associated with each pixel of a new depth map is passed to the random forest for classification.

19.2.2 Labeling Pixels

The objective is to construct a classifier that assigns to every pixel in a range image one out of a few body parts, such as a person's face, left arm, etc. There are 10 main body parts in Kinect (head, torso, two arms, two legs, two hands, and two feet), some of which are further divided into sub-parts, such as the upper/lower and left/right sides of a face, for a total of 31 parts.

Random Forest Research in classification has shown that some data sets are better classified with multiple classifiers, also called ensemble classifiers. In this case, the pixel/bodypart classification takes place with multiple tree classifiers, a random forest, see figure 40.

The feature $\mathbf{x}(\mathbf{p})$ (from above) is passed to every tree in the forest, where it is assigned some (tree-dependent) probability of belonging to each body part. The overall class probability of the pixel is finally computed as an average probability of the different trees.

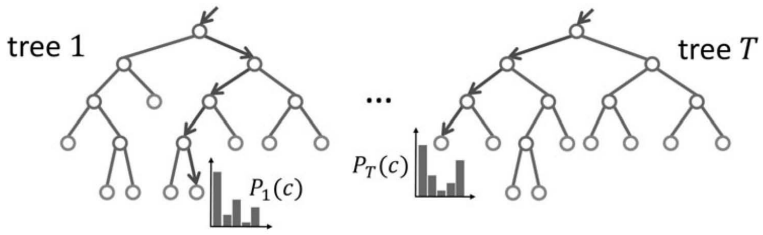

Figure 40: Random Forest: multiple trees are used to classify the same input and pool their decisions.

Creation of Training Set The primary source is a set of several hundred motion capture sequences featuring actors engaged in typical video game activities such as driving, dancing, kicking, etc. After clustering close-by pictures and retaining one sample per cluster, a set of about 100K poses is obtained. The measured articulation parameters are transferred (retargeted) to 15 parametric mesh models of human beings with a variety of body shapes and sizes. Body parts defined manually in texture maps are also transferred to these models, which are then skinned by adding different types of clothing and hairstyle, and rendered from different viewpoints as both depth and label maps using classical computer graphics techniques. 900k labeled images in total were created this way.

Training Classifier The classifier is trained as a random forest, using the features described above, but replacing the bootstrap sample used for each tree by a random subset of the training data (2,000 random pixels from each one of hundreds of thousands of training images).

The experiments described in Shotton et al. (2011) typically use 2,000 pixels per image and per tree to train random forests made of 3 trees of depth 20, with 2,000 splitting coordinates and 50 thresholds per node. This takes about one day on a 1,000-core cluster for up to one million training images. The pixelwise classification rate is ca. 60% (error of 40%!), which may appear as low but chance level is much lower (what is it?).

19.2.3 Computing Joint Positions

The random forest classifier assigns to each pixel some body part, but this process does not directly provide the joint positions because there is no underlying kinematic model. Instead, the position of each body part k could (for example) be estimated as some weighted average of the positions of the 3D points corresponding

to pixels labeled k, or using some voting scheme. To improve robustness, it is also possible to use mean shifts to estimate the mode of the following 3D density distribution:

$$f_k(\mathbf{X}) \propto \sum_{i=1}^{N} P(k|\mathbf{x}_i) A(\mathbf{p}_i) e^{-\frac{1}{\sigma_k^2}\|\mathbf{X}-\mathbf{X}_i\|^2}, \tag{39}$$

where $\mathbf{X}_i$ denotes the position of the 3D point associated with pixel $\mathbf{p}_i$, and $A(\mathbf{p}_i)$ is the area in world units of a pixel at depth $z(\mathbf{p}_i)$, proportional to $z(\mathbf{p}_i)^2$, so as to make the contribution of each pixel invariant to the distance between the sensor and the user. Each mode of this distribution is assigned the weighted sum of the probability scores of all pixels reaching it during the mean shift optimization process, and the joint is considered to be detected when the confidence of the highest mode is above some threshold. Since modes tend to lie on the front surface of the body, the final joint estimate is obtained by pushing back the maximal mode by a learned depth amount.

The average per-joint precision over all joints is 0.914 for the real data, and 0.731 for the synthetic one (tolerance of 0.1m). Thus, the voting procedures are relatively robust to 40% errors among individual voters. The synthetic precision is lower due to a great variability in pose and body shape. In realistic game scenarios, the precision of the recovered joint parameters is good enough to drive a tracking system that smoothly and very robustly recovers the parameters of a 3D kinematic model (skeleton) over time, which can in turn be used to effectively control a video game with natural body motions.

Closing Note

The final component of the system is a tracking algorithm whose details are proprietary but, like any other approach to tracking, it has temporal information at its disposal for smoothing the recovered skeleton parameters and recovering from joint detection errors.

20 More Systems and Tasks

20.1 Video Surveillance

Dav p578, ch22

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. We already mentioned some aspects of surveillance (e.g. in Sections 14 and 9), but we round the picture by adding some more 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 δ) from the horizontal axis (see Fig. 41). 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.

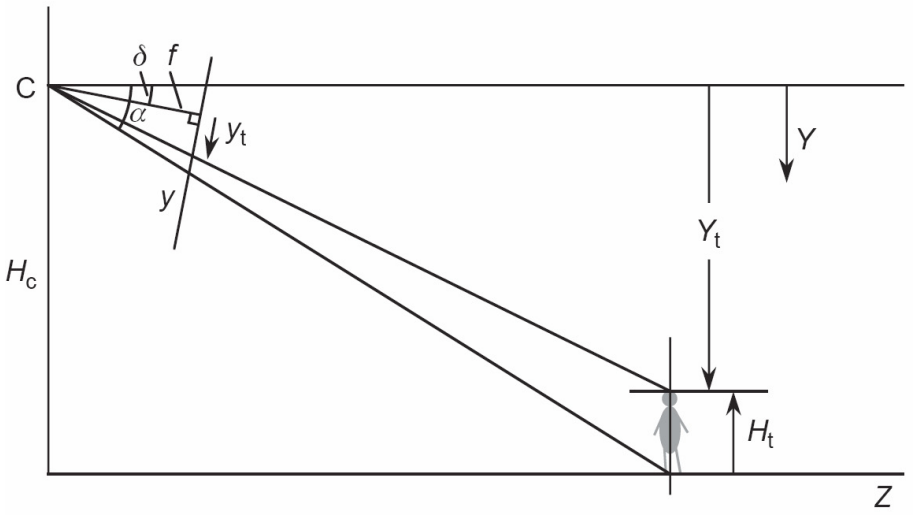

Figure 41: 3-D monitoring: camera tilted downwards. δ is the angle of declination of the camera optical axis. [Source: Davies 2012; Fig 22.2]

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 42.
2) Segmentation of the plate characters. Thresholding as described previously (Section 9.1)
3) Recognition of each character. A typical optical-character recognition system will do, see Pattern Recognition.

Scholar google "Anagnostopoulos" for a review on this topic.

Vehicle Recognition (Research) 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.:

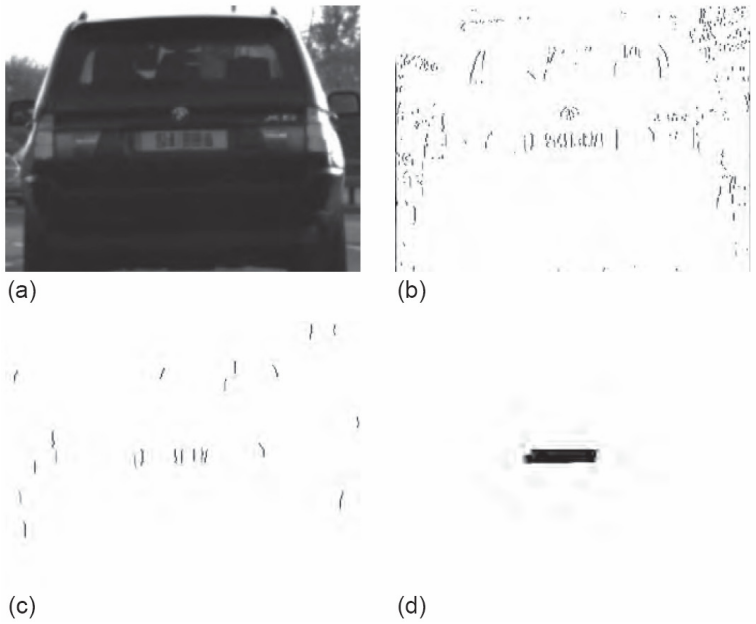

Figure 42: Simple procedure for locating license plates. (a) Original image with license plate pixellated to prevent identification. (b) Vertical edges of original image (e.g. with Sobel detector). (c) Vertical edges selected for length. (d) Region of license plate located by morphological operations, specifically horizontal closing followed by horizontal opening, each over a substantial distance (in this case 16 pixels). [Source: Davies 2012; Fig 22.13]

Figure 43: Make and model identification using the fronts features and their relations to the license plate.

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

20.2 Text Recognition

For this task, one generally distinguishes between two types of applications, optical character recognition (OCR) and recognition of text in the wild. In optical character recognition, the goal is to identify characters in documents or in handwriting; one generally assumes that characters are fairly easy to find - in most documents that is the case. Nowadays, a Deep CNN is used for recognition, in Section 20.2.1 we introduce how to use a package for Python. In recognition in the wild, one attempts to find text in arbitrary scenes, i.e. outdoor scenes, in which case we now are faced with the challenge to firstly locate the text, i.e. a street sign with elaborate instructions (Section 20.2.2).

The most powerful and fastest text recognizers are perhaps the ones by the Big-Three - exploiting their clouds -, for which one needs an internet connection however:

https://cloud.google.com/vision/docs/ocr
https://docs.aws.amazon.com/rekognition/latest/dg/text-detection.html
https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/

20.2.1 Optical Character Recognition

For localizing the letters in documents it is obvious to start with an image analysis first. For example, we threshold the image to obtain candidates for lines of text (Section 9.1); then we proceed to refine the search for letter centers at which locations one would apply the CNN to perform character identification. In case of scanned documents, it is probably necessary to apply an adaptive threshold, as the brightness might change across the page. With help of morphological operations we can generate letter candidate regions.

Identifying handwritten letters is of course much more challenging than identifying printed characters. To compare methods, the following famous dataset set is used in research:

http://yann.lecun.com/exdb/mnist/

In Matlab there exists a function script called `ocr` that does character recognition.

In Python the most popular package is `tesseract`, which - due to its complexity - offers also various wrappers in Python. Appendix **L.17.1** gives a simple example. Below is listed another one.

https://github.com/tesseract-ocr/tesseract/ (the original one)

https://pypi.org/project/tesserocr/ (pip install tesserocr)

20.2.2 Detection in the Wild

Detecting text in an arbitrary scene requires now a systematic whole image search. Matlab gives an example of how to do that. It is based on using so-called MSER features (maximally stable extrema regions). The actual character identification part uses then its `ocr` function. As one can imagine, an efficient search would be useful to have. Here is a list of systems:

https://github.com/chongyangtao/Awesome-Scene-Text-Recognition

The example for OCR given in **L.17.1** - using tesseract - includes detection in the wild, but can perhaps be improved using one of the newer search algorithms.

20.3 Autonomous Vehicles

Dav p636, ch23

Two decades ago, many experts would have considered it impossible that one day fully autonomous vehicles would drive on the road. Meanwhile most car companies develop a system that is capable of nearly or fully autonomous driving. Here we mention some of the principal issues addressed in such a system.

Such a system contains four interacting processes: environment perception, localization, planning and control. About two third of the system are involved in perception. Interestingly enough this proportion is about the same for the human brain.

Vision processing (environment perception) relies to a large degree on range cameras, because a range image can be easier segmented than an intensity image (see Kinect). A set of different range cameras is used covering different depth ranges with resolution down to 0.1 degree and depth up to 300m. Both radar and lidar is used (Appendix A).

The perception processes consist of the detection of pedestrians, bicyclists, cars, traffic signs, traffic lights, telephone poles, curbs, etc. Those recognition tasks are typically solved using multiple techniques complementing each other, as a single technique is often insufficient to reliably solve the problem. One example was given already with car license plate recognition. We give some more examples below where only a 2D gray-level image is processed.

Roadway/Roadmarker Location: is addressed for example with multilevel thresholding (Section 9.1); or with vanishing point detection using RANSAC on edge points (see Figure 44).

Locating of Vehicles: there are two very simple tricks to detect cars in 2D gray-scale images:

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 (Section 9.1).

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.

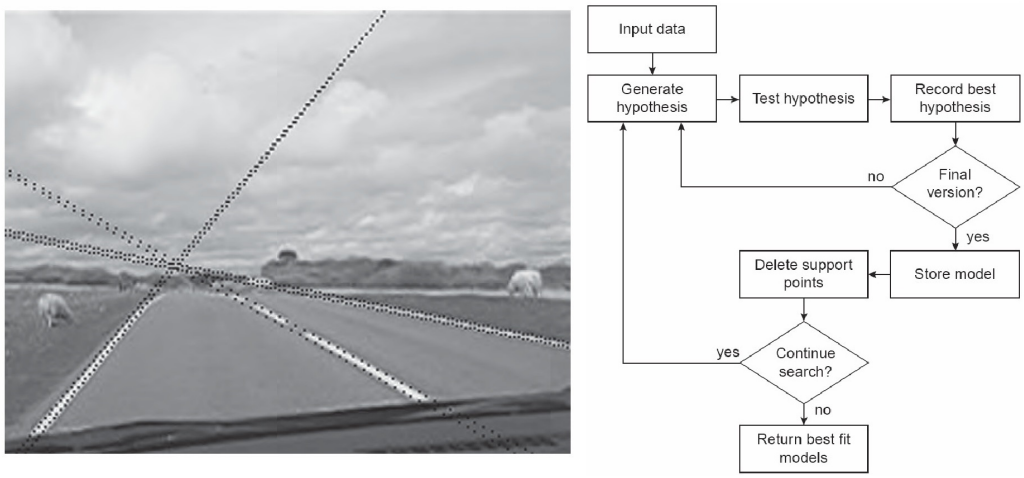

Figure 44:
Left: As the lane markings converge to a point on the horizon, the vanishing point, it is useful to determine that point using RANSAC (Section 16.3), applied to hypotheses of straight lines drawn through detected edge points.
Right: Flow chart of a lane detector algorithm.
[Source: Davies 2012; Fig 23.2], [Source: Davies 2012; Fig 23.4]

Figure 45: Detecting cars by exploiting the symmetry of vertical segments. [Source: Davies 2012; Fig 23.6]

Locating Pedestrians: We have introduced a pedestrian detection algorithm already in Section 6.3, but here we mention some other techniques that ensure a high recognition accuracy:

a) detection of body parts, arms, head, legs. The region between legs often forms an (upside-down) V feature.
b) Harris detector (Section 7.1) for feet localization.
c) skin color (see also Section J.1).

A substantial effort in developing such perception software is spent in temporal optimization of the implemented algorithms. Algorithms are not only developed in automobile companies themselves, but also supplier companies providing car-parts have started to develop recognition software.

20.4 Remote Sensing

wiki Remote_sensing

Dav p738, ch23

Remote sensing is the analysis of images taken by satellites or aircrafts (wiki Satellite_imagery, wiki Aerial_photography). It is used for example in meteorology, oceanography, fishing, agriculture, biodiversity conservation, forestry, landscape, geology, cartography, regional planning, education, intelligence and warfare.

The images one obtains are huge and manipulating them therefore takes a lot of time - it is practically impossible to apply sophisticated image-processing methods on large areas; for instance scanning the ocean for floating debris is unfeasible at this point. From a methodological viewpoint, there is no new technique to explain here. We now merely give some background information on this topic, in particular on satellite imagery.

In satellite imagery one can distinguish between four types of resolution: spatial, spectral, temporal and radiometric. The more modern the satellite, the higher are those resolutions in general.

- Geometric Resolution: specifies the optical precision and is typically given as the Ground Sample Distance (GSD). Modern commercial satellites provide a GSD of 31 centimeters for gray-scale images, thus a car would be ca. 5 x 15 pixels large. A coverage of one square kilometer would fit in a 3300 x 3300 pixel image approximately.

- Spectral Resolution: satellite images can be simple RGB photographs, but also a broad range of electromagnetic waves is typically measured. Early satellites recorded so-called multi-spectral images, where at each pixel several bands of the electromagnetic spectrum were recorded, sometimes up to 15 bands wiki Multispectral_image. Table 1 gives an impression of how some of those bands can be exploited. Meanwhile there exist satellites that record several tens or even hundreds of bands, generating so-called hyperimages, which permit detailed selections; the amount of storage required for such images is very large however.

Table 1: Example of bands and their use. Given ranges are approximate - exact values depend on satellite.

Band Label	Range (nm)	Comments
Blue	450-520	atmosphere and deep water imaging; depths up to 150 feet (50 m) in clear water.
Green	515-600	vegetation and deep water; up to 90 feet (30 m) in clear water.
Red	600-690	man-made objects, in water up to 30 feet (9 m), soil, and vegetation.
Near-infrared (NIR)	750-900	primarily for imaging vegetation.
Mid-infrared (MIR)	1550-1750	imaging vegetation, soil moisture content, and some forest fires.
Far-infrared (FIR)	2080-2350	imaging soil, moisture, geological features, silicates, clays, and fires.
Thermal infrared	10400-12500	emitted instead of reflected radiation to image geological structures, thermal differences in water currents, and fires, and for night studies.
Radar & related tech		mapping terrain and for detecting various objects.

- Temporal Resolution: specifies the time with which a satellite revisits the same location, called revisiting frequency or return period. This is of interest if one intends to track changes over time. The return period can be several days.

- Radiometric Resolution: concerns the 'range' of values and starts typically at 8 bits (256 values) and goes up to 16 bits (65535 values).

There exist software tools that preprocess the raw satellite images in order to transform them into a format that is more suitable for object detection and classification, see wiki Remote_sensing_application. The larger the area under investigation, the more time consuming is this transformation.

To compare methods one can participate in classification competitions:
https://www.kaggle.com/c/dstl-satellite-imagery-feature-detection/data

A Image Acquisition

Digital image acquisition is the process of analog-to-digital conversion of the 'outer' signal to a number, carried out by one or several image sensors (cameras). The conversion can be very complex and often involves the generation of an output that is 'visible' to the human eye.

One principal distinction between acquisition methods is passive versus active. In passive methods, the scene is observed based on what it offers: the simplest case is the regular light-sensitive camera that measures the environments luminance - the reflection of the illuminating sun. In active methods, a signal is sent out to probe the environment, analogous to a radar. Some sensors combines both methods.

The following are the principal sensors used for acquiring images.

Light-Sensitive Camera: measures from the visible part of the electromagnetic spectrum, typically red, green and blue dominance; the RGB camera is an example. Light is the preferred energy source for most imaging tasks because it is safe, cheap, easy to control and process with optical hardware, easy to detect using relatively inexpensive sensors, and readily processed by signal processing hardware.

Multi/Hyper-Spectral Sensors: measure from a broader part of the electromagnetic spectrum (than the light-sensitive cameras) with individual sensors tuned to specific bands. Originally it was developed for remote sensing (satellite imagery, Section 20.4), but is now also employed in document and painting analysis.

Range Sensor (Rangefinder): is a device that measures the distance from the observer to a target, in a process called ranging or rangefinding. Methods include laser, radar, sonar, lidar and ultrasonic rangefinding. Applications are for example surveying, navigation, more specifically for example ballistics, virtual reality (to detect operator movements and locate objects) and forestry.

Tomography Device: generates an image of a body by sections or sectioning, through the use of any kind of penetrating wave. The method is used in radiology, archeology, biology, atmospheric science, geophysics, oceanography, plasma physics, materials science, astrophysics, quantum information, and other areas of science.

The obtained 'raw' image may require some manipulation such as re-sampling in order to assure that the image coordinate system is correct; or noise reduction in order to assure that sensor noise does not introduce false information.

Summarizing, the output of sensors is an array of pixels, whose values typically correspond to light intensity in one or several spectral bands (gray-scale, colour, hyper-spectral, etc.), but can also be related to various physical measures, such as depth, absorption or reflectance of sonic or electromagnetic waves, or nuclear magnetic resonance.

B Convolution [Signal Processing]

Expressed in the terminology of applied mathematics, convolution is the repeated multiplication of one function on the domain of another function producing a third function; it is considered an 'operation' and is similar to cross-correlation. In signal processing terms, the first function is the signal - in our case often an image -, and the second function is a so-called *kernel* and manipulates a local neighborhood of that image at each location. This is easier to understand in one dimension first (Section B.1), then we introduce this for two dimensions (Section B.2).

B.1 In One Dimension

Our signal is for instance the face profile as in Figure 5. We wish to determine the extrema of that signal but that is easier if we smoothen the signal first with a low-pass filter. In plain words, we average the signal over a small neighborhood at each pixel. Let us clarify the convolution operation on a very simple signal `S`: our signal is zero everywhere except for one value in center which holds the value 2. We convolve it with three different kernels, `Ka`, `Kb` and `Kg` using the function `conv` (Python: `scipy.signal.convolve`). Then we plot the three new functions.

```
clear;
S   = [0 0 0 0 2 0 0 0 0];      % the signal
Ka  = [1 1 1]/3;                % averaging kernel
Kb  = [.25 .5 .25];             % triangle function
Kg  = pdf('norm',-2:2,0,1);     % 5-pixel Gaussian

Sa  = conv(S, Ka, 'same');
Sb  = conv(S, Kb, 'same');
Sg  = conv(S, Kg, 'same');

%% ------   Plotting
figure(1);clf;
Xax     = 1:nPix;
plot(Xax, S, 'k');   hold on;
plot(Xax, Sa, 'r*');
plot(Xax, Sb, 'bt');
plot(Xax, Sg, 'g.');
```

So far, not much has happened. The new function looks like its kernel, but scaled in amplitude. It becomes more interesting if we make the signal more complicated: turn on another pixel in signal `S` and observe. To ensure that you understand the detailed convolution process, look at the following explicit example:

```
Sa2     = zeros(1,nPix);
for i = 2:nPix-1
    Nb      = S(i-1:i+1);       % the neiborhood
    Sa2(i)  = sum(Nb .* Ka);    % multiplication with kernel and summation
end
assert(nnz(Sa2-Sa)==0);         % verify that it is same as 'conv'
```

The example is a simplified implementation of the convolution process, namely we do not observe the boundary values and it works only for kernel of length equal three pixels, with `i` running from 2 to `nPix-1`. But it contains the gist of the convolution operation.

When applying a convolution function you need to pay attention to what type of treatment you prefer for the boundary values. Matlab offers three options: full, valid and same; they return outputs of different sizes. We refer to the documentation for details. They do matter, so when you apply a convolution you need to think about the boundary values. If you prefer to set your own boundary values, then you compute only the central part of the convolution - in Matlab with option `valid` - and then use `padarray` to set the boundary values. We did that in the example of the face profiles, see Appendix **L.2**.

Mathematical Formulations In engineering equations the convolution can be written as:

$$(S * K)[k] = \sum_{i}^{n} S[i]K[k-i] \tag{40}$$

where `*` is the convolution symbol, `i` is the signal's variable, `n` the number of pixels of the signal and `k` is the Kernel's variable. That was the formulation for the discrete convolution. The continuous convolution is written as:

$$(S * K)(k) = \int_{-\infty}^{\infty} S(i)K(k-i)\delta i \tag{41}$$

B.2 In Two Dimensions [Image Processing]

In two dimensions the convolution process integrates over a local two-dimensional neighborhood. In case of an image the neighborhood can be a 3x3 pixel neighborhood, or a 5x5 pixel neighborhood, etc. The mathematical formulation remains essentially the same as above; one can understand the kernel as of any dimension, but the signal and kernel need to be defined clearly from the beginning.

In Matlab we now use the function `conv2`, in Python we use `scipy.signal.convolve2d`:

```
clear;
% --- the image
I       = zeros(10,10);
I(5,5)  = 1;
% --- kernels
Ka      = ones(3,3)/9;                  % averaging kernel
Kg      = fspecial('gaussian',5,1);     % 5x5 Gaussian kernel
% --- convolutions
Ia      = conv2(I, Ka);
Ig      = conv2(I, Kg, 'same');

%% ------   Plotting
figure(1);clf;[nr nc] = deal(2,2);
subplot(nr,nc,1); imagesc(I,[0 1]);
subplot(nr,nc,2); imagesc(Ia);  colorbar;
subplot(nr,nc,3); imagesc(Ig);  colorbar;
```

Speeding Up Because image convolution is a relatively time-consuming operation due to the repeated multiplication of two matrices, there exist methods to speed up image convolution. Those speed-ups work only if the kernel shows specific characteristics, in particular it needs to be symmetric. The Gaussian function for instance is suitable for speed up. In that case, an image convolution with a 2D Gaussian function can be separated into convolving the image twice with the 1D Gaussian function in two different orientations:

```
Kg1     = normpdf(-2:2);                          % a one-dim Gaussian
Iblr1   = conv2(I,Kg1'*Kg1,'same');               % with 2D function
Iblr2   = conv2(conv2(I,Kg1,'same'),Kg1','same'); % with 1D functions
Iblr3   = imgaussfilt(I,1);                       % special function
```

The product `Kg1'*Kg1` creates a 2D Gaussian - we could have also generated it using `fspecial('gaussian',[5 5],1)` for instance.

For small images, the durations for each of those three different versions are not much different - the duration differences become evident for larger images. Use `tic` and `toc` to measure time; or use `clock`.

C Filtering [Signal Processing]

An image can be filtered in different ways depending on the objective: one can measure emphasize or even search certain image characteristics. The term filter is defined differently depending on the specific topic (signal processing, computing, etc.). Here the term is understood as a function, which takes a small neighborhood and computes with its pixel values a certain value; that computation is done for each neighborhood of the entire image. That local function is sometimes called kernel.

For educational purposes, we differentiate here between five types of filtering. In the first one, the kernel calculates simple statistics with the neighborhood's pixel values (Section C.1). Then there are three basic techniques to emphasize a certain 'range' (band) of signal values: low-pass, band-bass and high-pass filtering (Sections C.2, C.3 and C.4, respectively). Finally, there can be complex filter kernels.

C.1 Measuring Statistics

Sometimes we intend to measure very simple statistics. Those in turn can be used for a variety of tasks, in particular texture description. One can determine the range of values within the neighborhood, their mean, their standard deviation etc. In Matlab those statistical measurements are provided with `rangefilt`, `stdfilt`, `medfilt2`, `ordfilt2`, etc.

There are various ways to apply your own specialized filters, for which it is easiest to arrange the image first into a matrix in which your neighborhoods are aligned column-wise using Matlab's `im2col/col2im` functions. That allows you to conveniently process the neighborhoods. Here is an example taking the average value for a neighborhood:

```
s   = 5;                                % side length
I   = reshape(linspace(0,1,s*s),[s s])'; % [s s] stimulus
C   = im2col(I,[3 3]);                  % extract [3 3] patches
Mn  = mean(C,1);                        % take mean
Ims = col2im(Mn,[1 1],[s s]-2);         % rearrange to matrix
Im  = padarray(Ims,[1 1]);              % pad to make it same size as I
```

C.2 Low-Pass Filtering

Because signals are often noisy, applying a low-pass filter to the image is often one of the first steps and that serves to squash the 'erratic' values. That is why images are typically filtered with at least a 3x3 Gaussian filter initially, no matter the exact task. The Gaussian filter is perhaps the most frequently used low-pass filter and it is worth looking at its one-dimensional shape, see Figure 46.

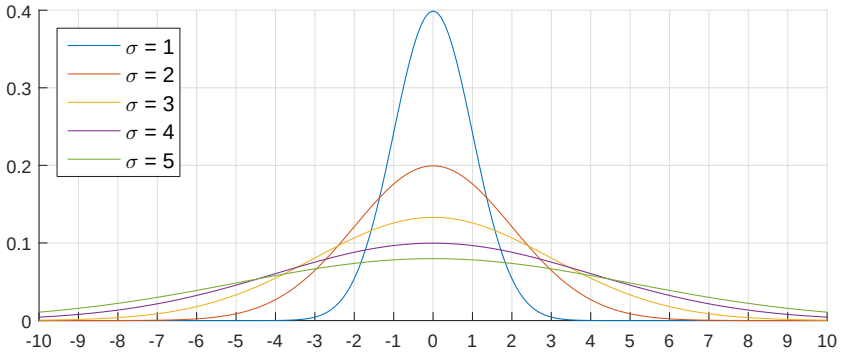

Figure 46: Gaussian function for five different values of sigma (1,..,5) placed at x=0. The parameter sigma determines the width of the bell-shaped curve.

In two dimensions, the Gaussian looks as depicted in Figure 11, namely in the first four patches (filter no. 1-4).

One can take also other functions for low-pass filtering, such as a simple average, see code block above or code in the Appendix on Convolution above, where `Ka = ones(3,3)/9` was used. The reason why the Gaussian function is so popular, is that it has certain mathematical advantages.

C.3 Band-Pass Filtering

One example of band-pass filtering is region detection with the Difference-of-Gaussian (DOG) as we introduced in Section (4.1). Another example are the Laplacian-of-Gaussian (LoG) filters used for texture detection (Figure 11).

C.4 High-Pass Filtering

Edge detection is an example for high-pass filtering. In some algorithms the first derivative of the Gaussian is used as high-pass filter, see also Figure 47.

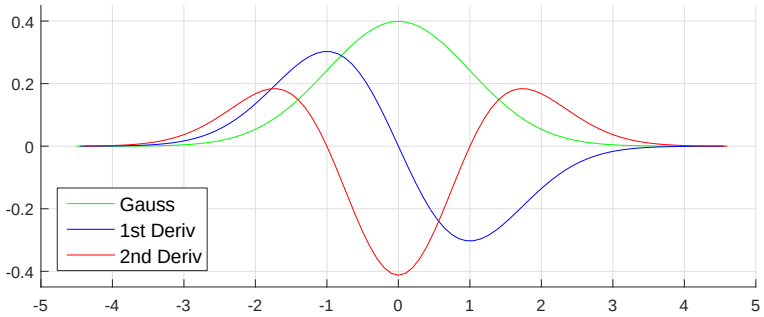

Figure 47: The Gaussian function (green) and its first (blue) and second (red) derivative. The Gaussian function itself is often used as low-pass filter. The first derivative of the Gaussian is often used as high-pass filter, for example in edge detection. The second derivative is occasionally used as band-pass filter, for example as region detection.

C.5 Function Overview

The variety of filtering commands can be confusing sometimes, that is why we provide here a short summary of frequently used filtering operations:

Table 2: Overview of commonly used image filtering methods. `I` = image.

Command	Comments
`rangefilt(I,true(rad*2+1))`	local range. Neighborhood is logical matrix.
`stdfilt(I,true(rad*2+1))`	local std dev. Neighborhood is logical matrix.
`Flt = fspecial('gaussian',[3 3],0.5)`	creates a filter. To be used with `conv2`, `imfilter` ...
`conv2(I,Flt,'same')` `convolve2d(I,Flt,mode='same')`	convolution in `scipy.signal`
`im2col(I,[3 3])`	extracts blocks from image, `[nPix 9]`.
`col2im(B,[3 3])`	inverse of `im2col`.
`colfilt`	column-wise filtering, uses `im2col` and `col2img`.
`imfilter`	filter multi-dimensional images.
`nlfilter`	general non-linear sliding filter.
`filter2(Flt, I, 'same')`	filtering with a FIR filter.

Table 3: Filtering of one-dimensional signals. `S` = one-dimensional signal.

Command	Comments

`F1d = normpdf(-2:2)`
`= pdf('norm',-2:2,0,1)`
`F1d = get_window(('gaussian',1),5)`
	generates a Gaussian (normal) filter for five pixels with μ=0 (mean) and σ=1 (standard deviation)
	in `scipy.signal`

`conv(S,F1d,'same')` `convolve(S,F1d,'same')`	1D-convolution. Watch the orientation of vectors! in `scipy.signal`

D Filtering Compact

A quick reference for filtering in Matlab, Python and OpenCV (accessed through Python).

```
Isum    = conv2(I, ones(3,3), 'same');  % summation
Igss    = imgaussfilt(I, 2);            % Gaussian
Imed    = medfilt2(I, [3 3]);           % median
Irng    = rangefilt(I, ones(3,3));      % range
Istd    = stdfilt(I, ones(3,3));        % standard deviation
Ired    = impyramid(I, 'reduce');       % downsampling by 2
Iexp    = impyramid(I, 'expand');       % upsampling by 2
```

```
from numpy import ones, gradient
from scipy.signal  import convolve2d
from scipy.ndimage import gaussian_filter, median_filter
from skimage.transform import pyramid_reduce, pyramid_expand

Isum    = convolve2d(I, ones((3,3)), 'same')
Igss    = gaussian_filter(I, sigma=2.0)
Imed    = median_filter(I, (3,3))
Ired    = pyramid_reduce(I)
Iexp    = pyramid_expand(I)
Dx, Dy  = gradient(I)
```

For OpenCV, there exists a nice overview:
https://docs.opencv.org/master/d4/d86/group__imgproc__filter.html#ga9fabdce9543bd602445f5db3827e4cc0

Input and output depths at one glance; as well as the border extrapolation types:

```
CV_8U           -1/CV_16S/CV_32F/CV_64F
CV_16U/CV_16S   -1/CV_32F/CV_64F
CV_32F          -1/CV_32F/CV_64F
CV_64F          -1/CV_64F

BORDER_REPLICATE     aaaaaa|abcdefgh|hhhhhhh
BORDER_REFLECT       fedcba|abcdefgh|hgfedcb
BORDER_REFLECT_101   gfedcb|abcdefgh|gfedcba
BORDER_WRAP          cdefgh|abcdefgh|abcdefg
BORDER_CONSTANT      iiiiii|abcdefgh|iiiiiii  with some specified 'i'
```

```
Isum    = cv2.boxFilter(I, -1, (3,3), normalize=False)
Isum2   = cv2.filter2D(I, -1, ones((3,3)))
Igss    = cv2.GaussianBlur(I, (3,3), 1)
Imed    = cv2.medianBlur(I.astype(uint8), 3)
Ired    = cv2.pyrDown(I)
Iexp    = cv2.pyrUp(I)
Dx, Dy  = cv2.spatialGradient(I.astype(uint8))
```

E Image Arithmetics

Image arithmetics is an expression for the pixel-wise arithmetic operations between two images, or between an image and a scalar. In principal this can be done by using the usual operation symbols in software packages, '+' for addition, '-' for subtraction, '*' for multiplication, etc. But software packages often provide specific functions that carry out these operations a bit faster.

When using one of the packages/libraries, one needs to pay attention how the software deals with operations where the result exceeds the limits of the data type (depth). This is handled differently for each software.

In Matlab :

```
imadd(I1,I2);
imsubtract(I1,I2);
imabsdiff(I1,I2);
immultiply(I1,I2);
imdivide(I1,I2);
imcomplement(s1, I1, s2, I2, a);  % s1*I1 + s2*I2 + a
```

In Python there does not seem to exist a specific set of functions. One would use the OpenCV functions instead, see next.

In OpenCV [py] :

```
In    = add(I1, I2)
In    = subtract(I1, I2)
In    = addWeighted(I1, 0.5, I2, 0.5, 0)
```

Some arithmetics are particularly used:

Sum of absolute differences (SAD): is a measure of the similarity between image blocks. It is calculated by taking the absolute difference between each pixel in the original block and the corresponding pixel in the block being used for comparison, `Df=imabsdiff(B1,B2)`. These differences are summed to create a simple metric of block similarity, the L1 norm of the difference image or Manhattan distance between two image blocks. wiki Sum_of_absolute_differences

Sum of squared differences: the square of SAD. deals better with outliers, but is also costlier to compute.

F Neural Networks

The element of a neural network (NN) is a so-called *Perceptron*. A perceptron essentially corresponds to a model of a linear classifier as in traditional Machine Learning. Formulated in the terminology of neural networks, a Perceptron consist of a 'neural' unit that receives input from other units. The inputs are weighted, then summed and then thresholded. For a two-class problem, there is essentially one integrating unit; for a multi-class problem there are several integrating units, whose count corresponds to the number of classes to be discriminated.

If such a Perceptron is stacked, then we talk of a *Multi-Layer* Perceptron (MLP). We show how to tune such a MLP in Keras, an API for Google's tensorflow, coming up next. There are also 'richer' ways to wire neural units; we introduce one such type of network in Section F.2.

In Section F.3 we survey the available, pretrained networks.

F.1 A Multi-Layer Perceptron (MLP) - Warming up to Keras

wiki Multilayer_perceptron

An MLP is a neural network consisting of four or more layers (Figure 48): an input layer, receiving the image in our case; two or more hidden layers that combine information; and an output layer, that indicates the selected class for an input image. A layer consist of *(neural) units*. The input layer has a unit count that typically corresponds to the number of pixels of the input image. The hidden layers have a unit count that is often a multiple of the input layer count. The output layer has a unit count that corresponds to the number of classes to be distinguished.

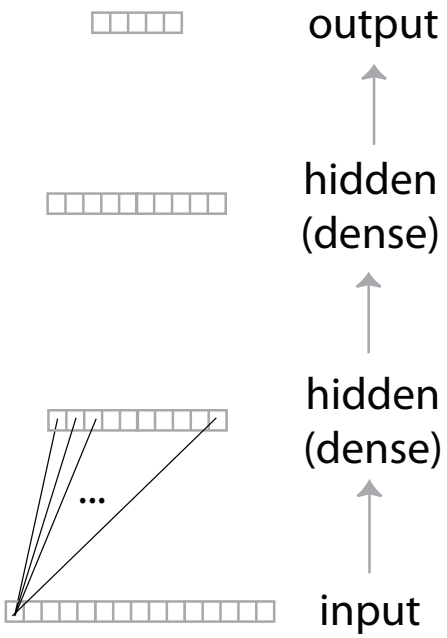

Figure 48: A Multi-Layer Perceptron (a general classifier in fact). The architecture has four layers: input layer, two hidden (dense) layers, and an output; classification flow occurs from bottom to top.
The **input layer** is linear and corresponds to the number of image pixels.
The **(first) hidden layer** receives the values from the input layer with all-to-all connectivity (only the connections from one pixel to all hidden units are indicated). Each connection has a weight whose exact value will be learned during the learning process. The corresponding weight matrix has the size of number of input pixels times number of hidden units.
The **(second) hidden layer** receives the output of the first hidden layer. The corresponding weight matrix has dimensions corresponding to the sizes of the two hidden layers.
The **output layer** has the same number of units as the number of classes to be discriminated. For instance if we intend to discriminate the digits 0 to 9, then there would be 10 output units. The output layer receives the values of the previous hidden layer.

The units between two layers are typically all-to-all connected that is the connections build a *complete bipartite graph* (wiki Bipartite_graph). Each connection holds a weight value that will be learned during the training process.

A unit typically sums the values it receives - weighted by the corresponding connection weights - and then passes the summed value c to a so-called *activation* function that is adjustable by a parameter b:

$$a = f(c, b) \tag{42}$$

The output a of the activation function is then passed to the next layer. There are many types of activation functions (wiki Activation_function), a popular one in Neural Networks is the rectified linear unit (relu), where b is a thresholding parameter: values below b are set to zero. The way such a unit works is essentially like a linear classifier (wiki Linear_classifier).

An Example We demonstrate how such a network can be programmed in Keras, a Python library (module) designed to test network structures wiki Keras. It is recommended to download Python version no. 3.5, which includes the modules for Tensorflow and Keras.

We use a database of handwritten digits (0-9), the so-called MNIST database, containing 60000 training images and 10000 testing images. How those images can be loaded is shown in Appendix **L.5** and we import that function into our script with the line `from LoadMNIST import LoadMNIST`. Note that the function loads the images as two dimensions (28 x 28). The training images are therefore stored as a three-dimensional array 60000 x 28 x 28. A single digit image can be viewed by `imshow(TREN[1,:])` (not shown in code).

```
# https://github.com/fchollet/keras/blob/master/examples/mnist_mlp.py
# Trains a Multi-Layer Perceptron on the MNIST dataset.
# Achieves 98.20% test accuracy after 12 epochs
from __future__ import print_function
from keras.models     import Sequential
from keras.layers     import Dense, Dropout
from keras.optimizers import RMSprop
from LoadMNIST        import LoadMNIST
batchSz     = 128       # batch size (# images per learning step)
nEpoch      = 12        # number of epochs = learning duration

#%% -----   Load Database & Labels  ------
TREN, TEST, Lbl = LoadMNIST()               # load the digits
TREN    = TREN.reshape(60000, 784)          # make image linear
TEST    = TEST.reshape(10000, 784)

#%% =====   Build Network     =======
N       = Sequential()
N.add( Dense(512, activation='relu', input_shape=(784,)) )  # input - hidden1
N.add( Dropout(0.2) )
N.add( Dense(512, activation='relu') )                      # hidden1 - hidden2
N.add( Dropout(0.2) )
N.add( Dense(nClass, activation='softmax') )                # hidden2 - output
N.summary()
N.compile( loss      = 'categorical_crossentropy',
           optimizer = RMSprop(),
           metrics   = ['accuracy'] )

#%% =====   Learning     =======
N.fit(TREN, Lbl.TrenMx,
      batch_size = batchSz,
      epochs     = nEpoch,
      verbose    = 1,
      validation_data = (TEST, Lbl.TestMx) )

#%% =====   Evaluation     =======
score = N.evaluate(TEST, Lbl.TestMx, verbose=0)
print('Test loss:    ', score[0])
print('Test accuracy:', score[1])
```

For the MLP network we do not even bother that the image has two dimensions but we 'linearize' the image by aligning all pixels in a single column: the input to the network is therefore a vector of length 784, `TREN.reshape(60000,784)`. We now discuss the three sections 'Build Network', 'Learning' and 'Evaluation'.

Build Network The network architecture is determined with five `N.add` commands. The first one determines the weights between the input layer and the first hidden layer, meaning there are 512*784 = 401408 weight parameters between those two layers. The second `N.add` determines a certain drop out rate, which helps the learning process by ignoring certain weights occasionally. The third `N.add` adds another hidden layer with 512 units. Thus we have 512*512 = 262144 weight values between the first and second hidden layer. The fourth `N.add` specifies again the drop out rate. The fifth `N.add` specifies the output layer, namely 10 units for 10 classes. With `N.summary()` the unit count is displayed.

There are many types of learning schemes and tricks that can be applied to learn a neural network. Here, those options are specified with the command `N.compile`.

Learning Learning occurs with the `N.fit` command. We learn on the training set `TREN` and the corresponding labels in `Lbl.TrenMx`. The parameter `batchSz` determines how many images are used per training step. There exists a rough optimum, too few or too many images per batch results in less efficient learning. The parameter `nEpoch` corresponds to the learning duration essentially.

Evaluation Evaluation takes place with `N.evaluate`. Here we employ the test set `TEST` and the corresponding labels `Lbl.TestMx` to estimate the prediction accuracy of the network.

The NNs as introduced so far are considered meanwhile traditional Neural Networks or *shallow* NNs, because they use few layers as opposed to the many layers used by a Deep Net.

F.2 Deep Belief Network (DBN)

wiki Deep_belief_network

A Deep Belief Network (DBN) is another type of Deep Neural Network. It is employed more rarely than the CNN, because its learning duration is slow in comparison to a CNN, but has the advantage that its overall architecture is simpler: often, with two layers you can achieve almost the same performance as with a CNN. Such a DBN has essentially a similar architecture to the Multi-Layer Perceptron, but the DBN contains also connections within the same layer. That particular characteristic makes it extremely powerful, but the downside is that learning is terribly slow.

A Belief Network is a network that operates with so-called *conditional dependencies*. A conditional dependence expresses the relation of variables more explicitly than just by combining them with a weighted sum - as is done in most other classifiers. However determining the full set of parameters for such a network is exceptionally difficult. Deep Belief Networks (DBNs) are specialized in approximating such networks of conditional dependencies in an efficient manner, that is at least partially and in reasonable time. Popular implementations of such DBNs consist of layers of so-called *Restricted Boltzmann Machines* (RBMs).

Architecture The principal architecture of a RBM is similar to the dense layer as used in an MLP or CNN. An RBM however contains an additional set of *bias* weights. Those additional weights make learning more difficult but also more capable - they help solving those conditional dependencies. The typical learning rule for a RBM is the so-called *contrast-divergence* algorithm.

The choice of an appropriate topology for the entire DBN is relatively simple. With two hidden layers made of RBMs, one can obtain already fantastic results. A third layer rarely helps in improving classification accuracy.

Learning Learning in a DBN occurs in two phases. In a first phase, the RBM layers are trained individually one at a time in an unsupervised manner (using the contrast-divergence algorithm): the RBMs perform quasi a clustering process. Then, in the second phase, the entire network is fine-tuned with a so-called back-propagation algorithm. As with CNNs, a Deep Belief Network takes much time to train. The downside of a DBN is, that there exists no method (yet) of speeding up the learning process, as is the case for CNNs.

Code Keras does not offer (yet?) methods to run a DBN, but tensorflow does. Tensorflow is however a bit trickier in specifying and running a network.

F.3 Pretrained Nets

There exist several implementations of Deep Nets that are available to the public and that can therefore be exploited for transfer learning. Many of them offer different network instantiations with varying number of layers: the more layers the better the prediction accuracy, but the more weights are required to tune and hence the larger its size and the longer its learning duration. These networks were tested on commonly

used testing sets, such as ImageNet, CIFAR, MNIST, etc. to name a few. The typical input size is a 256 x 256 pixel image, which then is cropped to 224 x 224 pixels. We list some popular ones, of which the first three can be accessed in PyTorch through the module `torchvision.models`.

ResNet (Deep Residual Learning for Image Recognition): by Microsoft. Comes with 18, 34, 50, 101 and 152 layers, ranging from 46 to 236 MB of weights. They show the best prediction accuracies on the testing sets.

DenseNet (Densely Connected Convolutional Networks): by Cornell University and FaceBook. Comes with 121, 169, 201 and 264 layers, with 32 to 114 MB of weights. They show competitive (near best) prediction accuracies on the training sets, but their weight set is smaller - about half of that for ResNet at approximately the same prediction accuracy.

InceptionV3 mainly by Google. Comes as a single instantiation with 107 MB of weights. Shows competitive performance. Appears to use the fewest weights.

PoseNet (A Convolutional Network for Real-Time 6-DOF Camera Relocalization): by Cambridge University. The net solves the task of position estimation. Given a few images of an object as training material, the can predict the viewer's point from a new position. It consists of 23 layers and has 50MB of weights.

G Classification, Clustering [Machine Learning]

Given are the data `DAT` and - if present - some class (group) labels `Lbl`:

`DAT`: matrix of size `[nObs x nDim]` with rows corresponding to images (whose pixels are taken columnwise) or features (some attributes extracted from images); and columns corresponding to the dimensionality (no. pixels per image or no. of attributes, respectively).

`Lbl`: a one-dimensional array `[nObs x 1]` with entries corresponding to the class (group or category) membership. This label vector allows us to train classifiers; if this array does not exist, then we can apply only clustering algorithms.

G.1 Classification

To properly determine the classification performance, the data set `DAT` is divided into a training set and a testing set. With the training set the classifier model is learned, with the testing set the model's performance is determined. To generate those two sets the label vector `Lbl` is used. We generate indices for training and testing set with `crossvalind`, then loop through the folds using `classify` or other classification functions:

```
Ixs = crossvalind('Kfold', Lbl, 3); % indices for 3 folds
Pf  = [];                           % initialize performance structure
for i = 1 : 3
    IxTst = Ixs==i;                 % i'th testing  set
    IxTrn = Ixs~=i;                 % i'th training set
    LbOut = classify(DAT(IxTst,:), DAT(IxTrn,:), Lbl(IxTrn)); % classification
    nTst  = nnz(IxTrn);
    Pf(i) = nnz(LbOut==Lbl(IxTst))/nTst*100; % in percent
end
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.

Classification Errors? Chances are fairly good that you will not succeed with such a straightforward classification attempt. Matlab may complain with some error that the covariance matrix can not be properly estimated. In that case, there are several options:

1. You can always try a kNN classifier (`knnclassify` in the bioinfo toolbox): it is somewhat simple, but you always obtain some results.
2. Use the Principal Component Analysis to reduce the dimensionality, see next Section G.1.1.
3. Use a Support-Vector machine: `svmclassify` (bioinfo toolbox). This is a very powerful classifier but it discriminates between two classes only. To exploit this classifier you then need to learn c classifiers, each one distinguishing between one category and all others (c = number of classes).

G.1.1 Dimensionality Reduction with Principal Component Analysis

The Principal Component Analysis (PCA) finds a more compact data space for the original data: the number of dimensions `nDim` of the data is 'reduced', perhaps only 70% of its original dimensionlity. In the following code `nObs` is the number of training samples:

```
coeff  = pca(DAT);              % [nDim, nDim] coefficients
nPco   = round(min(size(DAT))*0.7);  % suggested # of observations
PCO    = coeff(:,1:nPco);       % select the 1st nPco eigenvectors
DATRed = zeros(nObs,nPco);      % allocate memory
for i = 1 : nObs,
    DATRed(i,:) = DAT(i,:) * PCO;    % transform each sample
end
```

The reduced array `DATRed` is now being used for classification above.

G.2 Clustering

Given are the data `DAT` in the format as above, but there are no labels available: in some sense we try to find labels for the samples.

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):

```
      Ixs = kmeans(DAT, nc);
```

`Ixs` is a one-dimensional array of length `nObs` that contains the numbers ($\in 1..n_c$) which represent the cluster labels. We then need to write a loop that finds the corresponding indices, for instance with the following function. In this function, the variable `Pts` is equal the variable `DAT` in our above notation.

```
% Cluster info.
% IN Cls         vector with labels as produced by a clustering algorithm
%       Pts         points (samples)
%       minSize     minimum cluster size
%      strTyp      info string
% OUT I  .Cen    centers
%          .Ix     indices to points
%           .Sz     cluster size
%
function I = f_ClsInfo(Cls, Pts, minSize, strTyp)
nCls   = max(Cls);
nDim    = size(Pts,2);
H       = hist(Cls, 1:nCls);
IxMinSz = find(H>=minSize);
I.n     = length(IxMinSz);
I.Cen   = zeros(I.n,nDim,'single');
I.Ix    = cell(I.n,1);
I.Sz    = zeross(I.n,1);
for i = 1:I.n
    bCls       = Cls==IxMinSz(i);
    cen         = mean(Pts(bCls,:),1);
    I.Cen(i,:) = cen;
    I.Ix{i}    = single(find(bCls));
    I.Sz(i)     = nnz(bCls);
end
nP          = size(Pts,1);
I.notUsed   = nP-sum(I.Sz);

%% ---- Display
fprintf('%2d Cls %9s Sz %1d-%2d  #PtsNotUsed %d oo %d\n', ...
    I.n, strTyp, min(I.Sz), max(I.Sz), I.notUsed, nP);

end % MAIN
```

H Learning Tricks [Machine Learning]

Here we summarize general learning tricks that are used to improve the prediction accuracy of a classification system. They can be used for any classification method, for traditional or for modern (deep learning) methods.

• Data Augmentation

One challenging issue of training an image classification system is the collection of labeled image material - we need to instruct the system what the categories (classes) are. The collection is a laborous process and sometimes one has few images per class only, think of medical images where the number of images for affected patients is rarer than for healthy patients. In order to maximally exploit the available image material, one often enlarges the material by introducing slight manipulations of the images, a process called *data augmentation*. Data augmentation aims to increase the (class) variability as to mimick the variability present in real-word classes - think of how many types of cars or chairs there exist. Those manipulations consist of stretching the images, cropping them, rotating, flipping, etc. Some of those manipulations are introduced in Section 16.1.

In Matlab one would use functions such as `imtranslate`, `imcrop`, `imresize`, `imtransform`, `flipud`, `fliplr`, etc.

In Python all those functions are found in module `skimage.transform`.

In PyTorch the functions are provided in module `torchvision.transform` starting with the term `Random`, for example `RandomResizedCrop`, as introduced in 5.2, see code in **L.6.3**.

• Drop Out

Drop Out is a method to prevent overtraining. It was originally developed for Neural Networks, but can also be applied to other classification methods. In Neural Networks, during each learning step, a small percentage of neural units is eliminated of a layer. This is specified as drop-out rate. The technique prevents the classifier from overtraining, from becoming overly specific to certain input patterns that are not reflecting the classes well anymore.

• Hard Negative Mining

Hard Negative Mining is a method to collect images that are difficult to discriminate. In a general form, one can collect an entire image class that appears to be difficult for a target class, such as collecting vegetation images to train face detection (Section 6.1). In a more specific form, one focuses on samples that trigger false alarms in a classifier. For instance, if we train a classifier to categorize digits, than we observe and collect (mine) those digits that confuse one category, that is those samples that trigger false alarms. For instance, we collect those digit samples that confuse class '1', which could be '7's or '4's; or for category '3' it could be '2' or '8'. Then we re-train the classifier with those 'hard negatives'. This is systematically exploited in so-called cascade or boosting classifiers, such as the one trained to detect faces (Section 6.1).

I Resources

Databases

The famous one, used for learning the DeepNets:
http://image-net.org/

List of databases:
http://homepages.inf.ed.ac.uk/rbf/CVonline/Imagedbase.htm
https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research

The Kaggle website offers databases and a way to compare results by participating competitions:
https://www.kaggle.com

Libraries

Open-CV is probably the largest (open-source) library. It contains binaries and their corresponding source code, written in C and C++ :
https://www.opencv.org/
https://github.com/opencv
https://www.learnopencv.com
There does not exist separate documentation yet for how to use OpenCV through Python, so you need to browse the tutorials:
https://docs.opencv.org/master/d6/d00/tutorial_py_root.html

First steps to program CUDAs in Python:
https://devblogs.nvidia.com/numba-python-cuda-acceleration/

Code for Matlab can be found in particular on:
http://www.mathworks.com/matlabcentral/fileexchange

Sites with code resources by individual persons:
https://www.pyimagesearch.com/
http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/index.html
http://www.aishack.in
http://www.vlfeat.org/

Varia (Programming Support)

Summary of function names for Matlab, Python, etc:
http://mathesaurus.sourceforge.net/matlab-python-xref.pdf

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

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

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

Goes along with the Image Processing book by Gonzales et al.:
http://www.imageprocessingplace.com/

Embedded Systems

Steps toward embedded systems:
https://github.com/uTensor/uTensor

Community:

https://www.embedded-vision.com

Ideal for exploring:

http://www.jevois.org

J Color Spaces

wiki Color_space

There are certain color spaces in which segmentation of specific objects is sometimes easier, see table 4. To obtain these color spaces, we need to convert the image using commands such as `rgb2hsv` for example:

```
Ihsv     = rgb2hsv(Irgb);
```

In Python we find those conversion functions in module `skimage.color`:

```
Ihsv  = skimage.color.rgb2hsv(Irgb)
```

In OpenCV, those conversions are carried out with the function `cvtColor` and the specifier `COLOR_XXX`:

```
Ihsv    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2HSV)
```

Table 4: Commonly used color spaces. f: some complex function.

Space	Calculation	Comments
RGB wiki RGB_color_model	measured	- high correlation between channels; significant perceptual non-uniformity; mixing of chrominance and luminance data
normalized RGB wiki Rg_chromaticity	with $s = R+G+B$, then $r = R/s$ $g = G/s$ $b = B/s$	+ reduces shadow and shading effects (under certain assumptions) essentially a 2D space because the 3rd component is = 1-the other two Usage: computer vision `bsxfun(@rdivide,I,sum(I,3));`
HSI (HSV HSL) Hue Saturation Intensity (HS Value HS Brightness) wiki HSL_and_HSV	$H = f(R,G,B)$ $S = 1 - 3\frac{\min(R,G,B)}{R+G+B}$ $I = \frac{1}{3}(R+G+B)$	+ gives the user a good impression about the resulting color for a certain color value; describes color with intuitive values, based on the artists idea of tint, saturation and tone. - hue discontinuities; computation of I, V or L conflicts with properties of color vision. • Hue: dominant color such as red, green, purple and yellow. • Saturation: colorfulness in proportion to its brightness. • Intensity, Lightness or Value: related to luminance: Usage: design and editing (e.g. within graphics design tools) `rgb2hsv`
YC$_b$C$_r$ wiki YCbCr	Y = eq. 1 above $C_b = R - Y$ $C_r = B - Y$	Y also called *luma* here C_b, C_r also called *chroma* here Usage: digital video; image compression `rgb2ycbcr`
CIELAB or **CIE Lab** (related: **CIELUV**) wiki Lab_color_space	complex	+ perceptually uniform: small changes to a component do not change perception much - computationally intensive. • L: lightness • a: green-to-magenta • b: blue-to-yellow Usage: design and editing `makecform`, `applycform`

For complex spaces we need to call two commands in Matlab:

```
LabForm = makecform('srgb2lab');       % prepares transformation structure
Ilab    = applycform(Irgb, LabForm);   % applies above structure
```

J.1 Skin Detection

Skin detection is the process of segmenting skin from its background. It is used for example for face detection in videos; or in medical applications, to distinguish between tumor and skin, in which case the skin is background and the tumor is foreground. There are many algorithms for skin detection, often geared towards a specific objective. For instance for face detection in videos, one requires a fast algorithm. Often, the algorithms involve a transformation from the RGB color space into another color space in which the

red tones are (hopefully) isolated or emphasized. However, the advantage of those other color spaces has been occasionally questioned and one may therefore start developing an algorithm using the 'original' RGB space first, in order to see whether it is sufficient. For instance the following fast skin-detection algorithm is based on a set of hard rules in RGB space (I is a RGB color image):

```
% --- Minimum Range Condition
Mn      = min(I,[],3);
Mx      = max(I,[],3);
Brg     = (Mx-Mn)>15;

% --- Minimum Value Condition
R       = I(:,:,1);
G       = I(:,:,2);
B       = I(:,:,3);
Bmn     = R>95 & G>40 & B>20;

% --- Dominant Red Condition
Bc3     = abs(R-G)>15 & R>G & R>B;

% --- All Conditions
S       = Brg & Bmn & Bc3;
```

S is a binary image with on-pixels corresponding to skin.

For tumor/skin discrimination in medical images, the following color information has been used (Vezhnevets et al. 2003):
- The blue channel of the (original) RGB space
- The ratio between the red and the green channel of the RGB space
- The a channel of the Lab color space
- The H channel of the HSV space, see the face tracking example in Appendix **L.12.3**

J.2 Other Tasks

The red component of the YCbCr space can be exploited for the detection of tail lights and road signs in road scenes. For instance, turn a frame Frm into that color space and threshold its chroma luma at some appropriate value:

```
Frm     = cv.cvtColor(Frm, cv.COLOR_RGB2YCR_CB)
BWhih   = Frm[:,:,2] > 140
```

The segmentation of natural objects, such as leafs, fruits and landscapes, is sometimes done with the Hue map of the Hue-Saturation-Value (HSV) space.

J.3 Links

FAQ with the conciseness for programmers:
http://poynton.ca/PDFs/ColorFAQ.pdf

K Python Modules and Functions

Python offers functions for computer vision and image processing in various modules. Some functions exist in multiple modules. Most functions can be found in module `skimage`, which also contains a number of graphics routines. Its documentation can be found at `http://scikit-image.org`. A number of functions can be found in submodule `numpy.ndimage`. For signal processing and clustering we employ `scipy.signal` and `scipy.cluster` respectively.

Table 5: Overview of the `skimage` module, `http://scikit-image.org`

Utility Functions	**Comments**
`img_as_float`, `img_as_xxx`	conversion
`dtype_limits`	returns minimum and maximum intensity
Submodule	**Functions/Comments**
`color`	all types of conversions, e.g. `rgb2gray`
`data`	example images such as `coins`
`feature`	detection of blobs: `blob_dog`, `blob_xxx` edges: `canny` corners: `corner_harris`, `corner_xxx` maxima: `peak_local_max`
`filters`	edge detection (Prewitt, Sobel, ...); threshold determination (`threshold_otsu`); complex filters
`io`	input/output, e.g. `imread`
`measure`	various functions, e.g. `find_contours` (iso-contours), `regionprops`, `label`, `points_in_poly`, `ransac`
`morphology`	black-white: `binary_closing`, `binary_xxx` gray-scale: `dilation`, `erosion`, ... extrema: `h_minima`, `h_maxima`, `local_minima`, `local_maxima` various: `label`, `watershed`
`segmentation`	`watershed`, `quickshift`, `clear_border`
`transform`	`pyramid_reduce`, `pyramid_expand`, `resize`

Table 6: Overview of the `scipy` module, `https://docs.scipy.org/`.

Submodule	**Functions/Comments**
`cluster`	`vq.kmeans`
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
interpolate	Interpolation and smoothing splines
io	Input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
signal	`convolve`, `convolve2d`
`spatial.distance`	`pdist`, `squareform`
special	Special functions
stats	Statistical distributions and functions

Table 7: Matlab-Python equivalents.

Matlab	**Python**
`padarray(A,[2 2])`	`numpy.pad(A,2,'constant',constant_values=0)`

L Code Examples

The variable and parameter names are as follows. For images, frames and maps we usually write:

```
I    images, e.g. Irgb, Ihsv, Iybr for the respective color spaces; Igry, gray-scale image
F    frames, e.g. Frgb, Fhsv, ...
M    maps, e.g. Medg, Mrdg, ...
BW   black-white map
```

For vectors, counters, numbers, etc. we prefer to use:

```
B    logical vectors/arrays
c    counters or count: cF, cI, ...
n    number of something: nF, nI, ...
N    vector of number of something: Nimg, Nsel, ...
```

For indices we write:

```
ix   single (linear) index of something, ixUsd, ixSel, ...
Ix   vector of (linear) indices, IxSel, IxImg, ...
Rw   row indices
Cl   column indices
```

For initializing structures we use:

```
Stc  structure of something: StcElm, structuring element; StcTrk, structure for tracker; etc.
```

The shade colors mean:

```
Matlab code.
```

```
Python code.
```

```
Python code in which functions of OpenCV are called. In many of those OpenCV code snippets we omit
the plotting part, as it is the same as in Python.
```

L.1 Loading an Image/Video

Loading and saving an image in Matlab:

```
Irgb  = imread(fileName) % reads rgb it is rgb, otherwise gray-scale
Igry  = rgb2gray(Irgb) % convert to gray-scale
imsave(Irgb, fileName) % will deduce format from extension [or specify]
```

Loading, playing and saving a video in Matlab:

```
%% ---------    Movie Info      ---------
Vid         = VideoReader(pathLoad);            % movie 'handler'
[wth hgt]   = deal(Vid.Width,Vid.Height);
nFrm        = floor(Vid.duration * Vid.frameRate);  % number of frames

%% =========    Load Movie  =========
Vid.currentTime = 0;                        % set to 0 [in seconds]
fprintf('Reading sequence...' )
MOC(1:nFrm) = struct('cdata', zeros(hgt,wth,3,'uint8'), 'colormap',[]);
f=0;
while hasFrame(Vid) && Vid.currentTime<2    % we take only two seconds
    f       = f + 1;
    MOC(f).cdata =  readFrame(Vid);
end
fprintf('done\n');

%% ---------    Play the Movie (within Matlab)  --------
hf = figure(5);
set(hf,'position',[5 5 wth hgt]);
```

```
movie(hf, MOC, 1, Vid.frameRate);       % plays sequence

%% =========    Save Movie  =========
fprintf('Now writing...');
VidWrt      = VideoWriter(pathSave);
open(VidWrt);
for i = 1:f
    I       = MOC(i).cdata;
    writeVideo(VidWrt, I);
end
close(VidWrt);
fprintf('done\n');
```

Loading an image in Python in two popular ways, with skimage and with PIL (Python Imaging Library).

```
from skimage.io import imread, imsave
from skimage.color import rgb2gray
from PIL import Image
import cv2

path   = 'path to image'

Irgb    = imread(path)          # reads into a numpy array
Igry    = rgb2gray(Irgb)
Irgb    = Image.open(path)      # reads the PIL format

imsave('C:/ztmp/ImgSave.jpg', Irgb) # saves image as jpeg
```

Loading a mp4 video in Python:

```
import imageio
filename = 'pathToVideo.mp4'

#%% -----   Video Pointer & Info      -------
vid     = imageio.get_reader(filename, 'ffmpeg')
inf     = vid.get_meta_data()       # video info
szI     = inf['size']
nF      = inf['nframes']
print('#frames %i   fps %i  [%i %i] dur=%1.2f seconds' % \
      (nF, inf['fps'], szI[0], szI[1], inf['duration']))

#%% -----   Video Frames      -------
from matplotlib.pyplot import *
IxFrm = [10,20,30,40]               # a few selected frames
for f in IxFrm:
    Irgb    = vid.get_data(f)       # read a frame
    figure()
    imshow(Irgb)
    title(f'frame={f}', fontsize=20)
```

Loading an image with OpenCV through Python. Note that with OpenCV, the three chromatic channels are loaded in reversed order than usual: BGR not RGB.

```
import cv2

fp   = 'C:/IMGdown/SILU/Images/airplane/airplane00.tif' # image file path

# ---- cv2.imread loads image into a numpy array:
Igry = cv2.imread(fp,cv2.IMREAD_GRAYSCALE) # gray-scale
Ibgr = cv2.imread(fp,cv2.IMREAD_COLOR)     # loads channels as BGR: blue/green/red
Irgb = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2RGB)    # convert to RGB for displaying

#%% -----   accessing/setting individual pixels     -------
vBGR    = Ibgr[100,100]         # [3,]
```

```
Ibgr[100,100] = [255,255,255]
# faster solution:
v       = Ibgr.item(10,10,2)    # red value at 10,10
Ibgr.itemset((10,10,2),100)     # setting red value to 100

#%% -----   accessing/setting regions     -------
Roi = Ibgr[50:100, 50:100].copy()   # extracting a region
Ibgr[30:60,30:60] = [255,255,255]   # setting a region to one value

#%% ---    Separating into BGR channels: discouraged because slow   --------
B,G,R   = cv2.split(Ibgr)
Ibgr    = cv2.merge((B,G,R))

cv2.imshow('color',Ibgr)        # creates a separate window!
cv2.imshow('gray',Igry)
cv2.imshow('selected',Roi)

k   = cv2.waitKey(0) & 0xFF     # waits for a key press
cv2.destroyAllWindows()
```

Playing and writing a video in Python using the OpenCV functions:

```
import cv2 as cv
import numpy as np
#%% ----------------    Video Info      -------------
vidName = 'C:/Program Files/MATLAB/MATLAB Production Server/R2015a/toolbox/vision/visiondata/atrium.avi'
Cap     = cv.VideoCapture(vidName)
Cap.set(cv.CAP_PROP_POS_MSEC, 5*1000)       # set starting time
nFrm    = Cap.get(cv.CAP_PROP_FRAME_COUNT)  # obtain total # of frames
#%% ===============     LOOP FRAMES       ================
cF=0
while cF < nFrm:
    cF      += 1
    r,Frm   = Cap.read()            # reads one frame
    if r==False: break

    # ----  Plot    -----
    cv.imshow('frame', Frm)
    k   = cv.waitKey(30) & 0xff
    if k==27: break                 # ESC button

Cap.release()
cv.destroyAllWindows()

#%% ===============     LOOP FRAMES to Write      ================
Cap     = cv.VideoCapture(vidName)
fourcc  = cv.VideoWriter_fourcc(*'XVID')    # specify video codec
# specify size as WIDTH/HEIGHT!! (not column/row!)
Wrt     = cv.VideoWriter('c:/ztmp/Output.avi',fourcc, 20, (640,360))

while(Cap.isOpened()):
    r, Frm  = Cap.read()
    if r==False: break

    Wrt.write(Frm.astype(np.uint8))             # write the frame

    cv.imshow('frame',Frm)
    k   = cv.waitKey(1) & 0xFF
    if k==27: break

Cap.release()
Wrt.release()
cv.destroyAllWindows()
```

L.2 Face Profiles

```
clear;
Iorg    = imread('KlausIohannisCrop.jpg');  % image is color [m n 3]
Ig      = rgb2gray(Iorg);                   % turn into graylevel image
Ig      = Ig(35:end-35,35:end-35);          % crop borders a bit more
[h w]   = size(Ig);                         % image height and width

%% ===== Raw Profiles    ======
Pver    = sum(Ig,1);                        % vertical intensity profile
Phor    = sum(Ig,2)';                       % horizontal intensity profile

%% ===== Smoothen Profiles   =======
nPf     = round(w*0.05);                    % # points: fraction of image width
LowFlt  = pdf('norm', -nPf:nPf, 0, round(nPf/2)); % generate a Gaussian
Pverf   = conv(Pver, LowFlt, 'valid');     % filter vertical profile
Phorf   = conv(Phor, LowFlt, 'valid');      % filter horizontal profile
Pverf   = padarray(Pverf,[0 nPf],'replicate'); % extend to original size
Phorf   = padarray(Phorf,[0 nPf],'replicate'); % extend to original size

%% =====    Detect Extrema      =======
[PksVer LocPksVer] = findpeaks(Pverf);     % peaks
[TrgVer LocTrgVer] = findpeaks(-Pverf);    % troughs (sinks)
[PksHor LocPksHor] = findpeaks(Phorf);     % peaks
[TrgHor LocTrgHor] = findpeaks(-Phorf);    % troughs

%% --------     Plotting    -----------
figure(1);clf;
subplot(1,2,1);
    imagesc(Ig); colormap(gray);
subplot(2,2,2); hold on;
    plot(Pver,'k');
    plot(Pverf,'m');
    plot(LocPksVer,PksVer,'g^');
    plot(LocTrgVer,abs(TrgVer),'rv');
    set(gca,'xlim',[1 w]);
    xlabel('From Left to Right');
    title('Vertical Profile');
subplot(3,2,6); hold on;
    plot(Phor,'k');
    plot(Phorf,'m');
    plot(LocPksHor,PksHor,'g^');
    plot(LocTrgHor,abs(TrgHor),'rv');
    set(gca,'xlim',[1 h]);
    xlabel('From Top to Bottom');
    title('Horizontal Profile');
```

This would be the corresponding Python code:

```
from skimage.io    import imread
from skimage.color import rgb2gray
from scipy.signal  import convolve, get_window, argrelmax, argrelmin

#%% ----    Load Image -----
Irgb    = imread('c:/klab/do_lec/compvis/Matlab/KlausIohannisCrop.jpg')  # image is color [m n 3]
Ig      = rgb2gray(Irgb)                # turn into graylevel image
Ig      = Ig[34:-36,34:-36]             # crop borders a bit more
(h,w)   = Ig.shape                      # image height and width

#%% =====   Raw Profiles    ======
Pver    = Ig.sum(axis=0)                # vertical intensity profile
Phor    = Ig.sum(axis=1)                # horizontal#%% ---- Loading and Analysing

#%% =====   Smoothen Profiles   =======
sgm     = 10
nPf     = round(w*0.10)                    # #points: fraction of image width
LowFlt  = get_window(('gaussian',sgm),nPf)  # generate a Gaussian
```

```
LowFlt  = LowFlt / LowFlt.sum()             # normalize the filter
Pverf   = convolve(Pver, LowFlt, 'same')    # filter vertical profile
Phorf   = convolve(Phor, LowFlt, 'same')    # filter horizontal profile

#%% =====    Detect Extrema      =======
LocMaxVer    = argrelmax(Pverf)             # peaks
LocMinVer    = argrelmin(Pverf)             # troughs (sinks)
LocMaxHor    = argrelmax(Phorf)             # peaks
LocMinHor    = argrelmin(Phorf)             # troughs (sinks)

#%% ---- Plotting
from matplotlib.pyplot import figure, subplot, imshow, plot, title, xlim, ylim, cm
figure(figsize=(10,6))
subplot(1,2,1)
imshow(Ig, cmap=cm.gray)
subplot(2,2,2)
plot(Pver,'k')
plot(Pverf,'m')
plot(LocMaxVer[0],Pverf[LocMaxVer],'g^')
plot(LocMinVer[0],Pverf[LocMinVer],'rv')
xlim(1,w)
title('Vertical Profile')
subplot(2,2,4)
plot(Phor,'k')
plot(Phorf,'m')
plot(LocMaxHor[0],Phorf[LocMaxHor],'g^')
plot(LocMinHor[0],Phorf[LocMinHor],'rv')
ylim(1,h)
title('Horizontal Profile')
```

L.3 Image Processing I: Scale Space and Pyramid

```
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;
    % --- Downsampling with stp
    stp     = 2^i;
    Idwn    = downsample(Ilpf,stp);      % first along rows
    PY{i+1} = downsample(Idwn',stp)';       % then  along colums
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
```

For the Python code, we also wrote a function `fspecialGauss`, which mimics Matlab's `fspecial` function, see separate code block below.

```
from numpy import mgrid, exp
from skimage       import data
from skimage.color import rgb2gray
from skimage.transform import resize
from scipy.signal  import convolve2d

def fspecialGauss(size,sigma):
    x, y = mgrid[-size//2 + 1:size//2 + 1, -size//2 + 1:size//2 + 1]
    g   = exp(-((x**2 + y**2)/(2.0*sigma**2)))
    return g/g.sum()

#%% ------   Load & Transform   --------
Irgb    = data.chelsea()
Ig      = rgb2gray(Irgb)    # turn into single type
(m,n)   = Ig.shape

#%% ------   Initialize     ---------
nLev    = 5
SS = {}; PY={}; aFlt={};
SS[0]   = Ig.copy()   # scale space: make original image first level
PY[0]   = Ig.copy()   # pyramid:      "      "      "     "     "

#%% =====    Scale Space and Pyramid    ========
for i in range(1,nLev):
    Flt     = fspecialGauss(2+i*3, i)           # 2D Gaussian
    aFlt[i-1] = Flt                             # store for plotting
```

```
    Ilpf    = convolve2d(Ig, Flt, mode='same')  # low-pass filtered image
    SS[i]   = Ilpf
    # --- Downsampling with stp
    stp     = 2**i
    PY[i]   = resize(Ilpf,(m//stp,n//stp))

#%% -----    Plotting
from matplotlib.pyplot import *
figure()
(nr,nc) = (nLev,3)
for i in range(0,nLev):
    if i<nLev-1:
        subplot(nr,nc,i*nc+1)
        imshow(aFlt[i])
    subplot(nr,nc,i*nc+2)
    imshow(SS[i])
    subplot(nr,nc,i*nc+3)
    imshow(PY[i])
```

L.4 Feature Extraction I

L.4.1 Regions

```
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;
[nr nc] = 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);
```

For the Python example we assume that we've placed the 2D Gaussian filter - as used in the above example L.3 - into a separate module:

```
from skimage       import data
from skimage.color import rgb2gray
from fspecialGauss import *
from scipy.signal  import convolve2d
from matplotlib.pyplot import *

#%% ------  The image   -------
Icol    = data.immunohistochemistry()
I       = rgb2gray(Icol);

#%% ======   Diff-of-Gaussians (DOG) =======
Fs1     = fspecialGauss(11, 3)              # fine   low-pass filter
Fs2     = fspecialGauss(21, 6)              # coarse low-pass filter

Is1     = convolve2d(I, Fs1, 'same')        # convolution with image
Is2     = convolve2d(I, Fs2, 'same')

Idog    = Is2 - Is1                         # diff-of-Gaussians
BWblobs = Idog > 0.02                       # thresholding for blobs

#%% -----    Plotting       -------
figure(figsize=(15,15))
nr, nc = 3,2
subplot(nr,nc,1),    imshow(Icol)
subplot(nr,nc,2),    imshow(I)
subplot(nr,nc,3),    imshow(Idog);colorbar
subplot(nr,nc,4),    imshow(BWblobs)
```

L.4.2 Edge Detection

In the last few lines we also show how to thin the black-white map, an action that is useful for contour tracing (coming up in a later section).

```
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');
BWpwt   = edge(Iblr, 'prewitt');            % similar to Sobel
BWlog   = edge(Iblr, 'log', [], sgm);       % laplacian of Gaussian
BWzex   = edge(Iblr, 'zerocross');
% For Canny we apply the original image I as it performs 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');
subplot(nr,nc,2); imagesc(BWsob); title('Sobel');
subplot(nr,nc,3); imagesc(BWpwt); title('Prewitt');
subplot(nr,nc,4); imagesc(BWlog); title('Log');
subplot(nr,nc,5); imagesc(BWcny); title('Canny');
subplot(nr,nc,6); imagesc(BWlog); title('Zero-Cross');

%% -------  Cleaning Edge Map for Contour Tracing   -------
Medg = BWcny;
Medg = bwmorph(Medg,'clean');   % removes isoloated pixels
Medg = bwmorph(Medg,'thin');    % turns 'thick' contours into 1-pixel-wide contours
```

```
from skimage           import data
from skimage.feature   import canny
from skimage.filters   import roberts, sobel, prewitt
from skimage.morphology import remove_small_objects, thin

#%% -----   Load    ------
I       = data.camera()          # uint8

#%% -----   One-scale edge detection    -------
Mrob    = roberts(I)
Msob    = sobel(I)
Mpwt    = prewitt(I)

#%% -----   Canny for two scales   ------
ME1     = canny(I, sigma=1)
ME2     = canny(I, sigma=3)

#%% -------  Plotting   -------
from matplotlib.pyplot import *
figure(figsize=(12,10)); nr,nc = 2,3;
subplot(nr,nc,1); imshow(I); title('Input Image')
subplot(nr,nc,2); imshow(Mrob); title('Roberts')
subplot(nr,nc,3); imshow(Msob); title('Sobel')
subplot(nr,nc,4); imshow(Mpwt); title('Prewitt')
subplot(nr,nc,5); imshow(ME1); title('Canny $\sigma$=1')
subplot(nr,nc,6); imshow(ME2); title('Canny $\sigma$=3')
```

```
#%% -------  Cleaning Edge Map for Contour Tracing   -------
Medg = ME1.copy()               # we copy for reason of clarity
remove_small_objects(Medg,2,in_place=True)  # removes isoloated pixels
Medg = thin(Medg)                           # turns 'thick' contours into 1-pixel-wide contours
```

```
import cv2
# ---- Stimulus
fp      = 'C:/xxx/xxx.tif'
Igry    = cv2.imread(fp, cv2.IMREAD_GRAYSCALE) # gray-scale

#%% ======    detect edge pixels  =======
lowThr  =  50       # low threshold
hihThr  = 200       # high threshold
BWcan  = cv2.Canny(Igry, lowThr, hihThr)
```

L.4.3 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);
vnc     = sgm^2;
Gy      = Gy.*((PTS(2,:).^2-vnc)/(vnc^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_Norm(D)
D       = D - mean(D(:));
D       = D / sum(abs(D(:)));

end % SUB
```

L.5 Loading the MNIST dataset

Note that this function contains two subfunctions, `ff_LoadImg` and `ff_ReadLab`.

```
% Loads the MNIST data - the 4 files on the following website -
% and converts them from ubyte to single.
%       http://yann.lecun.com/exdb/mnist/
% IN    - (no input arguments)
% OUT   TREN    training images column wise [60000 28*28]
%       TEST    testing (sample) images [10000 28*28]
%       Lbl     struct with training and testing class labels as matrices
%                   .Tren [60000 10] binary matrix with training labels
%                   .Test [60000 10] binary matrix with testing labels
function [TREN TEST Lbl] = LoadMNIST()
fprintf('Loading MNIST...');
filePath    = 'c:\kzimg_down\MNST\';
FnTrainImg  = [filePath 'train-images.idx3-ubyte'];
FnTrainLab  = [filePath 'train-labels.idx1-ubyte'];
FnTestImg   = [filePath 't10k-images.idx3-ubyte'];
FnTestLab   = [filePath 't10k-labels.idx1-ubyte'];

TREN        = ff_LoadImg(FnTrainImg); % [60000 28*28]
TEST        = ff_LoadImg(FnTestImg);  % [10000 28*28]
Lbl.Tren    = ff_ReadLab(FnTrainLab);  % [60000 10]
Lbl.Test    = ff_ReadLab(FnTestLab);   % [10000 10]

TREN    = single(TREN)/255.0;
TEST    = single(TEST)/255.0;
fprintf('done. Normalized\n');
end % MAIN FUNCTION

%% ==========   Load Digits
function IMGS = ff_LoadImg(imgFile)
    fid     = fopen(imgFile, 'rb');
    idf  = fread(fid, 1, '*int32',0,'b');    % identifier
    nImg    = fread(fid, 1, '*int32',0,'b');
    nRow    = fread(fid, 1, '*int32',0,'b');
    nCol    = fread(fid, 1, '*int32',0,'b');
    IMGS    = fread(fid, inf, '*uint8',0,'b');
    fclose( fid );
    assert(idf==2051, '%s is not MNIST image file.', imgFile);
    IMGS = reshape(IMGS, [nRow*nCol, nImg])';
    for i=1:nImg
        Img         = reshape(IMGS(i,:), [nRow nCol])';
        IMGS(i,:) = reshape(Img, [1 nRow*nCol]);
    end
end % SUB FUNCTION

%% ==========   Load Labels
function Lab = ff_ReadLab(labFile)
    fid     = fopen(labFile, 'rb');
    idf     = fread(fid, 1, '*int32',0,'b');
    nLabs   = fread(fid, 1, '*int32',0,'b');
    ind     = fread(fid, inf, '*uint8',0,'b');
    fclose(fid);
    assert(idf==2049, '%s is not MNIST label file.', labFile);
    Lab = false(nLabs,10);
    ind = ind + 1;
    for i=1:nLabs
        Lab(i,ind(i)) = true;
    end
end % SUB FUNCTION
```

```
# Loads the MNIST data - the 4 files on the following website -
# and converts them from ubyte to float.
#       http://yann.lecun.com/exdb/mnist/
# IN    -       (no input arguments)
# OUT   TREN    training images as 3-dimensional array [60000 28 28]
#       TEST    testing (sample) images as 3-dim array [10000 28 28]
#       Lbl     struct with training and testing class labels as vectors and
#               matrices:
#                   .Tren   [60000 1] vector with training labels
#                   .Test   [10000 1] vector with testing labels
#                   .TrenMx [60000 10] binary matrix with training labels
#                   .TestMx [10000 10] binary matrix with testing labels
def LoadMNIST():
    from numpy import fromfile, int8, uint8
    from collections import namedtuple
    import struct
    from keras.utils import to_categorical

    print('Loading MNIST...')
    filePath    = 'c:/kzimg_down/MNST/'
    Lbl         = namedtuple('Lbl',['Tren','Test', 'TrenMx', 'TestMx'])
    #%% --------    TRAINING DATA   ------------
    fipaImg     = filePath + 'train-images.idx3-ubyte'
    fipaLab     = filePath + 'train-labels.idx1-ubyte'

    with open(fipaLab, 'rb') as Flab:
        idf, nLab = struct.unpack(">II", Flab.read(8))
        Lbl.Tren = fromfile(Flab, dtype=int8)

    with open(fipaImg, 'rb') as Fimg:
        idf, nImg, nRow, nCol = struct.unpack(">IIII", Fimg.read(16))
        TREN = fromfile(Fimg, dtype=uint8).reshape(nLab, nRow, nCol)

    #%% --------    TESTING DATA   ------------
    fipaImg     = filePath + 't10k-images.idx3-ubyte'
    fipaLab     = filePath + 't10k-labels.idx1-ubyte';

    with open(fipaLab, 'rb') as Flab:
        idf, nLab = struct.unpack(">II", Flab.read(8))
        Lbl.Test = fromfile(Flab, dtype=int8)

    with open(fipaImg, 'rb') as Fimg:
        idf, nImg, nRow, nCol = struct.unpack(">IIII", Fimg.read(16))
        TEST = fromfile(Fimg, dtype=uint8).reshape(nLab, nRow, nCol)

    #%% --------    NORMALIZATION   ------------
    TREN    = TREN.astype('float32')/255.0
    TEST    = TEST.astype('float32')/255.0

    #%% --------    Convert class vectors to binary class matrices  -------
    Lbl.TrenMx = to_categorical(Lbl.Tren, 10)
    Lbl.TestMx = to_categorical(Lbl.Test, 10)

    print('done. Normalized\n')
    return TREN, TEST, Lbl
```

L.6 CNN Examples [TensorFlow/Keras and PyTorch]

L.6.1 Loading the CIFAR-10 files

```
% Loads images & category labels for CIF10 database.
% IN    -
% OUT   IMGS        images [60000 32*32*3]
%       Lbl         category labels [60000 1] E [1..10]
%       CatNames    strings {10 x 1}
function [IMGS Lbl CatNames] = LoadImgCIF10()
global PATH

%% ---- Train
bt1 = [PATH.IMGS 'data_batch_1'];   BT1 = load(bt1);
bt2 = [PATH.IMGS 'data_batch_2'];   BT2 = load(bt2);
bt3 = [PATH.IMGS 'data_batch_3'];   BT3 = load(bt3);
bt4 = [PATH.IMGS 'data_batch_4'];   BT4 = load(bt4);
bt5 = [PATH.IMGS 'data_batch_5'];   BT5 = load(bt5);

TREN    = [BT1.data; BT2.data; BT3.data; BT4.data; BT5.data];
LblTren = [BT1.labels; BT2.labels; BT3.labels; BT4.labels; BT5.labels];

%% ---- Test
btt     = [PATH.IMGS 'test_batch'];     BTT = load(btt);
TEST    = BTT.data;
LblTest = BTT.labels;

%% ---- Concat Train & Test
IMGS    = [TREN; TEST];
Lbl     = single([LblTren; LblTest])+1; % add one because labeling E [0..9]

%% ----
bm          = [PATH.IMGS 'batches.meta.mat'];     BM = load(bm);
CatNames    = BM.label_names;
end
```

PyTorch provides a method to load this data set, but in case you use TensorFlow only (without access to the PyTorch routine), then here would be an example how to read the data set:

```
# Loads the CIFAR10 data set. Here we load the Matlab files (!)
# For the python files see 'http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz'
# IN    -       (no input arguments)
# OUT   TREN    training images as 3-dimensional array [50000 32 32]
#       TEST    testing (sample) images as 3-dim array [10000 32 32]
#       Lbl     struct with training and testing class labels as vectors and
#               matrices:
#                   .Tren   [50000 1] vector with training labels
#                   .Test   [10000 1] vector with testing labels
#                   .TrenMx [50000 10] binary matrix with training labels
#                   .TestMx [10000 10] binary matrix with testing labels
#from __future__ import absolute_import
from numpy import shape, zeros, reshape, transpose
import os
import scipy.io as sio
#import pickle
from collections import namedtuple
from keras.utils import to_categorical

def LoadImgCIF10():

    path     = 'c:/kzimg_down/CIFAR10/'
    Lbl      = namedtuple('Lbl',['Tren','Test'])

    #%% -----   TRAINING DATA   -------
    TREN        = zeros((50000,32*32*3), dtype='uint8')
    Lbl.Tren    = zeros((50000,1), dtype='uint8')
```

```
    for i in range(1, 6):
        fpath = os.path.join(path, 'data_batch_' + str(i))
        DAT     = sio.loadmat(fpath)
        TREN[    (i-1)*10000 : i*10000, :]  = DAT['data']
        Lbl.Tren[(i-1)*10000 : i*10000]     = DAT['labels']

    #%% -----   TESTING DATA   -------
    fpath       = os.path.join(path, 'test_batch')
    DAT         = sio.loadmat(fpath)

    TEST        = DAT['data']
    Lbl.Test    = DAT['labels']

    #%% -----   Labels   -------
    fpath      = os.path.join(path, 'batches.meta')
    DAT        = sio.loadmat(fpath)
    LbStr      = DAT['label_names']

    #%% -----    NORMALIZATION   ------------
    TREN       = TREN.astype('float32')/255.0
    TEST       = TEST.astype('float32')/255.0

    #%% -----   Reshape   -------
    TREN    = reshape(TREN,(50000,3,32,32))
    TREN    = TREN.transpose(0, 2, 3, 1)
    TEST    = reshape(TEST,(10000,3,32,32))
    TEST    = TEST.transpose(0, 2, 3, 1)

    #%% --------    Convert class vectors to binary class matrices  -------
    Lbl.TrenMx = to_categorical(Lbl.Tren, 10)
    Lbl.TestMx = to_categorical(Lbl.Test, 10)

    return TREN, TEST, Lbl, LbStr
```

L.6.2 A CNN for the CIFAR-10 set [TensorFlow/Keras]

```
# Train a simple CNN on the CIFAR10 (32x32 pixels, 10 classes)
from __future__ import print_function
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
from LoadImgCIF10 import *
batchSz     = 32
nEpoch      = 20

#%% -----   Load Database & Labels  ------
Isz         = (32, 32, 3)
(TREN, TEST, Lbl, LbCat) = LoadImgCIF10()

#%% =====   Build Network     =======
N = Sequential()

N.add(Conv2D(32, (3,3), padding='same', input_shape=Isz))
N.add(Activation('relu'))
N.add(Conv2D(32, (3,3)))
N.add(Activation('relu'))
N.add(MaxPooling2D(pool_size=(2,2)))
N.add(Dropout(0.25))

N.add(Conv2D(64, (3,3), padding='same'))
N.add(Activation('relu'))
N.add(Conv2D(64, (3,3)))
N.add(Activation('relu'))
```

```
N.add(MaxPooling2D(pool_size=(2,2)))
N.add(Dropout(0.25))

N.add(Flatten())
N.add(Dense(512))
N.add(Activation('relu'))
N.add(Dropout(0.5))
N.add(Dense(10))
N.add(Activation('softmax'))

fOptim  = keras.optimizers.rmsprop(lr=0.0001, decay=1e-6)
N.compile(loss      = 'categorical_crossentropy',
          optimizer = fOptim,
          metrics   = ['accuracy'])

#%% =====   Learning     =======
N.fit(TREN, Lbl.TrenMx,
          batch_size = batchSz,
          epochs     = nEpoch,
          validation_data = (TEST, Lbl.TestMx),
          shuffle    = True)

#%% =====   Evaluation     =======
score   = N.evaluate(TEST, Lbl.TestMx, verbose=0)
print('Model Accuracy = %.2f' % (score[1]))
```

L.6.3 Example of Transfer Learning [PyTorch]

Section 5.2

Below are three code blocks. The first code block trains the model. The second code block saves the model - it is a single line. The third code block demonstrates how to apply the model on a single (novel) image.

Block 1 The string variable `dirImgs` is the directory to your database, with two folders: 'train' and 'valid' for training and validation. In each one of those two folders, there is the exact same list of folder names representing the category labels. For those in turn, we provide different images. For small data sets place two thirds of the images into the corresponding training folder (ie. `train/dog/`) and one third into the corresponding validation folder (`valid/dog/`). To obtain optimal results, one would calculate the (normalized) mean and standard devation of the entire dataset - here we take arbitrary values.

```
from numpy import zeros, array

import torch
from torchvision import models, transforms
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
from torch.optim import SGD, lr_scheduler
import torch.nn as nn

import time
from os.path import join
import copy

dirImgs     = 'C:/ImgCollection/'
# computed mean values for all images of entire database
ImgMean     = [0.50, 0.50, 0.50]    # we assume them to be 0.5
ImgStdv     = [0.22, 0.22, 0.22]    # we assume them to be 0.22
szImgTarg   = 224
#%% ================    Parameters      ================
nEpo        = 6         # number of epochs (learning steps)
lernRate    = 0.001     # learning rate
momFact     = 0.9       # momentum factor
szStep      = 7         # step size
gam         = 0.1       # gamma
szBtch      = 4         # number of images per batch
# --- the model
MOD         = models.resnet18(pretrained=True)

#%% ################    DATA PREPARATION        ##################
#%% ----------------    Augmentation & Normalization    ---------------
AUGNtrain    = transforms.Compose([
        transforms.RandomResizedCrop(szImgTarg),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize(ImgMean, ImgStdv)  ])
AUGNvalid    = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(szImgTarg),
        transforms.ToTensor(),
        transforms.Normalize(ImgMean, ImgStdv)   ])
#%% ----------------    Folder and Loader   ---------------
FOLDStrain  = ImageFolder(join(dirImgs,'train'), AUGNtrain)
FOLDSvalid  = ImageFolder(join(dirImgs,'valid'), AUGNvalid)
LOADERtrain = DataLoader(FOLDStrain, batch_size=szBtch, shuffle=True)
LOADERvalid = DataLoader(FOLDSvalid, batch_size=szBtch, shuffle=True)

nImgTrain   = len(FOLDStrain)       # total number of images for training set
nImgValid   = len(FOLDSvalid)       # total number of imges for validation set

aLbClass    = FOLDStrain.classes    # class labels
nClss       = len(aLbClass)

#%% #################   Select Device (GPU | CPU)   ###################
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

#%% #################   PREPARE MODEL   ##################
# freezing layers until last one (to avoid backward computations)
for param in MOD.parameters():
    param.requires_grad = False
# replaces last fully-connected layer with new random weights:
MOD.fc  = nn.Linear(MOD.fc.in_features, nClss)

MOD     = MOD.to(device)

# only parameters of final layer are being optimized
optim   = SGD(MOD.fc.parameters(), lr=lernRate, momentum=momFact)

# Decrease learning rate every szStep epochs by a factor of gamma
sched   = lr_scheduler.StepLR(optim, step_size=szStep, gamma=gam)

crit    = nn.CrossEntropyLoss()

#%% #################        TRAIN AND EVALUATE      ###################
t0 = time.time()
WgtsBest = copy.deepcopy(MOD.state_dict())
accuBest = 0.0
PrfVal, PrfTrn   = zeros((nEpo,2)), zeros((nEpo,2))
# SSSSSSSSSSSSSSSSS    LOOP EPOCHS     SSSSSSSSSSSSSSSSSSSSSS
for i in range(nEpo):
    print(f'---- epoch {i}/{nEpo} -----')
    # ================    TRAIN    ================
    sched.step()
    MOD.train()                             # set model to training mode
    # =======   LOOP Batches  ======
    lossRun = 0.0
    cCrrRun = 0
    for ImBtc, LbBtc in LOADERtrain:
        ImBtc = ImBtc.to(device)            # [nImgBtch 3RGB height width]
        LbBtc = LbBtc.to(device)            # [nImgBtch]
        with torch.set_grad_enabled(True):
            # forward
            Post        = MOD(ImBtc)        # posteriors [nImgBtch nClasses]
            _,LbPred    = Post.max(1)       # predicted labels
            loss        = crit(Post, LbBtc)
            # backward + optimize
            optim.zero_grad()               # zero the parameter gradients
            loss.backward()
            optim.step()
        # statistics
        lossRun += loss.item() * szBtch
        cCrrRun += torch.sum(LbPred==LbBtc.data)
    # --- performance per epoch
    lossEpo = lossRun / nImgTrain
    accuEpo = cCrrRun.double() / nImgTrain
    print(f'train  loss: {lossEpo:.4f}  acc: {accuEpo:.4f}')
    PrfTrn[i,:] = [accuEpo, lossEpo]        # record performance

    # ================    VALIDATE    ================
    MOD.eval()                              # set model to evaluate mode
    # =======   LOOP Batches  ======
    lossRun = 0.0
    cCrrRun = 0
    for ImBtc, LbBtc in LOADERvalid:
        ImBtc = ImBtc.to(device)            # [nImgBtch 3RGB height width]
        LbBtc = LbBtc.to(device)            # [nImgBtch]
        with torch.set_grad_enabled(False):
            # forward
            Post        = MOD(ImBtc)        # posteriors [nImgBtch nClasses]
            _,LbPred    = Post.max(1)       # predicted labels
            loss        = crit(Post, LbBtc)
```

```
        # statistics
        lossRun += loss.item() * szBtch
        cCrrRun += torch.sum(LbPred==LbBtc.data)
    # --- performance per epoch
    lossEpo = lossRun / nImgValid
    accuEpo = cCrrRun.double() / nImgValid
    print(f'valid  loss: {lossEpo:.4f}  acc: {accuEpo:.4f}')
    PrfVal[i,:] = [accuEpo, lossEpo]        # record performance
    # keep weights if better than previous weights
    if accuEpo>accuBest:
        accuBest    = accuEpo
        WgtsBest    = copy.deepcopy(MOD.state_dict())

# ----  Concluding
tElaps = time.time() - t0
print('Training duration {:.0f}min {:.0f}sec'.format(tElaps // 60, tElaps % 60))
print(f'Max accuracy {accuBest:4f}')

MOD.load_state_dict(WgtsBest)   # loads best performing weights for application

#%% -----   Plot Learning Curves    -------
import matplotlib.pyplot as plt
from matplotlib.pyplot import *
figure
plot(PrfVal[:,0],'b')
plot(PrfTrn[:,0],'b:')
plot(PrfVal[:,1],'r')
plot(PrfTrn[:,1],'r:')
gca().legend(('valid acc', 'train acc', 'valid err', 'train err'))
```

Block 2 You can save your model as follows:

```
torch.save(MOD.state_dict(), pathVariable)
```

Block 3 Now we reload the model and test a single image:

```
import torch
from torchvision import models, transforms
import torch.nn as nn
from PIL import Image

pathToYourModel  = 'path and name of your model'
imgPath  = 'path and name of your testing image'

ImgMean     = [0.50, 0.50, 0.50]    # we assume them to be 0.5
ImgStdv     = [0.22, 0.22, 0.22]    # we assume them to be 0.22
szImgTarg   = 224

#%% --- Prepare the Model
MOD         = models.resnet18()
MOD.fc      = nn.Linear(MOD.fc.in_features, nClasses)
# --- Now load the weights
MOD.load_state_dict(torch.load(pathToYourModel))
MOD.eval()

f_PrepImg  = transforms.Compose([
        transforms.Resize((szImgTarg,szImgTarg)),
        transforms.ToTensor(),
        transforms.Normalize(ImgMean, ImgStdv)  ])

#%% --- Load Image and Apply
Irgb        = Image.open(imgPath)   # open as PIL image (not numpy!)
Irgb        = f_PrepImg(Irgb)     # image preparation
```

```
# make it 4-dimensional by adding 1 dimension as axis=0
Irgb        = Irgb.unsqueeze(0)
# now we feed it to the network:
Post        = MOD(Irgb)         # posteriors [1 nClasses]
psT,lbT    = Post.max(1)       # posterior and predicted label (still as tensors!)
lbPred      = lbT.item()     # label as a single scalar value (now as scalar in regular python)
```

L.7 Feature Extraction

L.7.1 Detection

Comparison of four feature detectors.

```
clear;
I       = imread('cameraman.tif');

%% =====    Detect Features     ======
FHar    = detectHarrisFeatures(I);
FSurf   = detectSURFFeatures(I);
FFast   = detectFASTFeatures(I);
FBrsk   = detectBRISKFeatures(I);

%% -----    Plot    ------
figure(1); [nr nc]=deal(2,2);
subplot(nr,nc,1);
    imagesc(I); hold on;
    plot(FHar.Location(:,1),FHar.Location(:,2),'y.');
    title(['Harris: #' num2str(FHar.Count)]);
subplot(nr,nc,2);
    imagesc(I); hold on;
    plot(FSurf.Location(:,1),FSurf.Location(:,2),'y.');
    title(['SURF: #' num2str(FSurf.Count)]);
subplot(nr,nc,3);
    imagesc(I); hold on;
    plot(FFast.Location(:,1),FFast.Location(:,2),'y.');
    title(['FAST: #' num2str(FFast.Count)]);
subplot(nr,nc,4);
    imagesc(I); hold on;
    plot(FBrsk.Location(:,1),FBrsk.Location(:,2),'y.');
    title(['BRISK: #' num2str(FBrsk.Count)]);
```

L.7.2 Finding Feature Correspondence in two Images

This code block is rewritten from Matlab's example on finding feature correspondences. Matlab offers a function that draws a line between a pair of corresponding points.

```
clear;
I1          = rgb2gray(imread('viprectification_deskLeft.png'));
I2          = rgb2gray(imread('viprectification_deskRight.png'));

%% =====    Detect Features     ======
Pt1det      = detectHarrisFeatures(I1);     % struct with .Locations [nPt1 2]
Pt2det      = detectHarrisFeatures(I2);

%% =====    Extract Features     ======
[D1 Pt1vld] = extractFeatures(I1, Pt1det);  % struct with .Features [nDsc 64]
[D2 Pt2vld] = extractFeatures(I2, Pt2det);

%% =====    Match Features     ======
IxPairs     = matchFeatures(D1, D2);        % [nMatches 2]
Pt1Matched  = Pt1vld(IxPairs(:,1), :);
Pt2Matched  = Pt2vld(IxPairs(:,2), :);

%% -----    Plot    ------
figure(1);
showMatchedFeatures(I1, I2, Pt1Matched, Pt2Matched);
```

L.8 Object Detection, Object Recognition/Localization

L.8.1 Face Detection

An OpenCV-through-Python example for face and eye detection using the venerable Viola-Jones algorithm, a quick algorithm that uses so-called Haar features (Section 6.1):

```
import cv2

imPth   = 'pathToSomeImage.JPG'
cascPth = 'C:/SOFTWARE/opencv/build/etc/haarcascades/'
FaceDet = cv2.CascadeClassifier(cascPth+'haarcascade_frontalface_default.xml')
EyeDet  = cv2.CascadeClassifier(cascPth+'haarcascade_eye.xml')

#%% ----    Load Image   ------
Ibgr    = cv2.imread(imPth)
Igry    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2GRAY)

#%% ====    Face Detection  ======
aFaces  = FaceDet.detectMultiScale(Igry, 1.3, 5)
aEyes   = []
for (x,y,w,h) in aFaces:
    Pface   = Igry[y:y+h, x:x+w]                # face patch
    Eyes    = EyeDet.detectMultiScale(Pface)    # eye detection on face patch
    aEyes.append(Eyes)                          # coords relative to face patch

#%% ----    Plot    ------
Irgb    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2RGB) # convert to RGB for displaying
from matplotlib.pyplot import *
import matplotlib.patches as patches
close('all')
figure(1)
imshow(Irgb, cmap=cm.gray)
for i,(x,y,w,h) in enumerate(aFaces):
    rect = patches.Rectangle((x,y),w,h, linewidth=1, edgecolor='r', facecolor='none')
    gca().add_patch(rect)
    # --- adding eyes
    Eyes    = aEyes[i]
    for (xe,ye,we,he) in Eyes:
        rect = patches.Rectangle((x+xe,y+ye),we,he, linewidth=1, edgecolor='g', facecolor='none')
        gca().add_patch(rect)
```

L.8.2 Pedestrian Detection

An example for pedestrian detection using HOG features (Section 6.3) exploiting the functions of OpenCV through Python. The usage is very similar to the face detection example.

```
import cv2

imPth   = 'C:\images\SomePhotoWithPedestrians.jpg'

PedDet     = cv2.HOGDescriptor()
PedDet.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

#%% ----    Load Image   ------
Ibgr    = cv2.imread(imPth)
Igry    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2GRAY)

#%% ====    Pedestrian Detection  ======
(aPers, Like) = PedDet.detectMultiScale(Igry, winStride=(4,4), padding=(8,8), scale=1.3)

#%% ----    Plot    ------
Irgb    = cv2.cvtColor(Ibgr, cv2.COLOR_BGR2RGB) # convert to RGB for displaying
from matplotlib.pyplot import *
```

```
import matplotlib.patches as patches
figure(1)
imshow(Irgb, cmap=cm.gray)
for i,(x,y,w,h) in enumerate(aPers):
    rect = patches.Rectangle((x,y),w,h, linewidth=1, edgecolor='r', facecolor='none')
    gca().add_patch(rect)
```

L.8.3 Object Recognition/Localization

```
import numpy as np
import cv2 as cv
from skimage.io import imread

imgNa           = 'person.jpg'     # the image to be processed

# Initialize the parameters
thrConf         = 0.5           # confidence threshold
thrNonMaxSupp   = 0.4           # non-maximum suppression threshold
wthInput, hgtInput = 416,416   # height/width of network's input image

#%% ------    file paths    ----------
pthDarkNet      = 'c:/IMGdown/DEEPNETS/DarkNet/'
imgPth          = pthDarkNet + imgNa
fileClassLabels = pthDarkNet + 'coco.names'
fileConfig      = pthDarkNet + 'yolov3.cfg'
fileWeights     = pthDarkNet + 'yolov3.weights'

#%% ------    Load image    ---------
Irgb  = imread(imgPth);   hgtImg=Irgb.shape[0];  wthImg=Irgb.shape[1]

#%% ------    Load DarkNet    ---------
NET         = cv.dnn.readNetFromDarknet(fileConfig, fileWeights)
NET.setPreferableBackend(cv.dnn.DNN_BACKEND_OPENCV)
NET.setPreferableTarget(cv.dnn.DNN_TARGET_CPU)
# obtain names of unconnected layers
IxUnConn    = NET.getUnconnectedOutLayers()
aLayersNames= NET.getLayerNames()
aLayUnconn  = [aLayersNames[i[0]-1] for i in IxUnConn]
# load class labels
aClassLabels = None
with open(fileClassLabels, 'rt') as f:
    aClassLabels = f.read().rstrip('\n').split('\n')
nClassLabels    = len(aClassLabels)

#%% ============    Process One Image   =================
Iblob    = cv.dnn.blobFromImage(Irgb, 1/255, (wthInput, hgtInput), \
                        [0,0,0], 1, crop=False) # [1 3 hgtInp wthInp]
NET.setInput(Iblob)      # place input to net
ACONF    = NET.forward(aLayUnconn) # 3 arrays: [507 85],[2028 85],[8112 85]

#%% ============    Analyse Cells  =================
aCanClass, aCanConf, aCanBbox = [],[],[]
# ======    Loop Grids [507, 2028, 8112]     ========
for AConf in ACONF:
    # ======    Loop Cells     ========
    for Cand in AConf:
        # candidate vector is [1 85]: x/y/w/h + 80 class score values
        Scores  = Cand[5:]     # conf scores for the 80 classes
        ixClass = np.argmax(Scores)
        conf    = Scores[ixClass]
        if conf > thrConf:
            xCen    = int(Cand[0] * wthImg)
            yCen    = int(Cand[1] * hgtImg)
            wth     = int(Cand[2] * wthImg)
            hgt     = int(Cand[3] * hgtImg)
```

```
            left    = int(xCen-wth/2)
            top     = int(yCen-hgt/2)
            aCanBbox.append([left, top, wth, hgt])
            aCanClass.append(ixClass)
            aCanConf.append(float(conf))
            print('Candidate %s'%(aClassLabels[ixClass]))

# select winners: non-maximum suppression of bounding boxes
IxSel   = cv.dnn.NMSBoxes(aCanBbox, aCanConf, thrConf, thrNonMaxSupp)
print('%d winners left'%(len(IxSel)))

# classification (inference) duration
dur, _  = NET.getPerfProfile()
durStr  = 'Inference duration: %.2f ms'%(dur*1000.0/cv.getTickFrequency())

#%% -------   Plot    ---------
for i in IxSel:
    ix      = i[0]
    # display bounding box
    box     = aCanBbox[ix];   left=box[0];    top=box[1]
    right   = left+box[2]
    bottom  = top +box[3]
    cv.rectangle(Irgb, (left,top), (right,bottom), (0,0,255))
    # class label and its confidence
    conf    = aCanConf[ix]
    ixClass = aCanClass[ix]
    label = '%s:%.2f' % (aClassLabels[ixClass], conf)
    szLab, baseLine = cv.getTextSize(label, cv.FONT_HERSHEY_SIMPLEX, 0.5, 1)
    top     = max(top, szLab[1])
    cv.putText(Irgb, label, (left, top), cv.FONT_HERSHEY_SIMPLEX, 0.5, (255,255,255))

# duration in upper left of image
cv.putText(Irgb, durStr, (0, 15), cv.FONT_HERSHEY_SIMPLEX, 0.5, (0,0,255))

from matplotlib.pyplot import *
figure(1); imshow(Irgb)
```

L.9 Image Processing II: Segmentation

L.9.1 Gray-Scale Images

```
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);

%% -----    Plotting
figure(1);clf;[nr nc] = deal(3,2);
subplot(nr,nc,1);
    imagesc(I); colormap(gray);
subplot(nr,nc,2);
    bar(histc(I(:),[0:255])); hold on;
    tOts    = tOts*256;
    plot([tOts tOts],[0 1000],'b:');
    title('Histogram');
subplot(nr,nc,3);
    imagesc(BWots); title('Otsu');
subplot(nr,nc,4);
    imagesc(BWmed); title('Median Value');
subplot(nr,nc,5);
    imagesc(BWkmen); title('K-means');
subplot(nr,nc,6);
    imagesc(W); title('Watershed');
```

```
from numpy             import arange, zeros, ones, median
from scipy.signal      import convolve2d
from sklearn.cluster   import KMeans
from scipy.ndimage     import label
from skimage           import data
from skimage.feature   import peak_local_max
from skimage.filters   import threshold_otsu
from skimage.segmentation import watershed
from fspecialGauss     import fspecialGauss
#%% -----   Load    ------
I       = data.coins()          # uint8
(m,n)   = I.shape
nPix    = m*n
#%% -----    Thresholding with Otsu and Median
tOts    = threshold_otsu(I)         # E [0 255]  uint8
BWots   = I > tOts
tMed    = median(I)                 # E [0 255] uint8
BWmed   = I > tMed

#%% -----    k-Means
Ivec    = I.reshape((nPix,1))
RKM     = KMeans(n_clusters=2, max_iter=30, n_init=1).fit(Ivec)
Cen     = RKM.cluster_centers_      # cluster centers [nK 3]
# Distances & nearest neibors
DIS     = zeros((nPix,2),float)
```

```
for i in range(2):
    DIS[:,i] = (Cen[i] - Ivec[:,0]) ** 2    # squared distance
IxNN    = DIS.argmin(axis=1)        # nearest neighbor
BWkmen  = IxNN.reshape((m,n))       # reshape to image for plotting

#%% -----    Watershed
Ilpf    = convolve2d(I, fspecialGauss(5,2), mode='same')
Mmx     = peak_local_max(Ilpf, labels=I, footprint=ones((3,3)), indices=False)
Mmrk    = label(Mmx)[0]
W       = watershed(Ilpf, Mmrk, mask=I);

#%% -----    Plotting
from matplotlib.pyplot import *
figure(figsize=(12,9));(nr,nc)=(3,2)
# image in upper left
subplot(nr,nc,1)
imshow(I, cmap=cm.gray);
# histogram in upper right
subplot(nr,nc,2)
hist(I.flatten(),arange(0,256))
plot([tOts,tOts],[0,1000],'b:')
plot([tMed,tMed],[0,1000],'r:')
title('Histogram')
# output of otsu
subplot(nr,nc,3)
imshow(BWots); title('Otsu')
# output of median
subplot(nr,nc,4)
imshow(BWmed); title('Median Value')
# output of kmeans
subplot(nr,nc,5)
imshow(BWkmen); title('K-means')
# output of watershed
subplot(nr,nc,6)
imshow(W); title('Watershed')
```

L.9.2 K-Means on Color Images

```
clear;
% ------    Load image      -------
Irgb = imread('peppers.png');
szI  = size(Irgb);                  % [rows columns 3]
nPix    = szI(1)*szI(2);
% ------    Reshape to [nPix 3]      -------
ICol = reshape(Irgb,[nPix 3]);      % [nPix 3] format for clustering

%% =========    Kmeans      ==========
L  = kmeans(single(ICol),5, 'MaxIter',50, 'online','off'); % L E [1,..,5]
Lmx = reshape(L, szI(1:2));         % turn into label matrix

%% -----    Plot    -------
figure(1);clf;[nr nc] = deal(1,2);
subplot(nr,nc,1);
    imagesc(Irgb);
subplot(nr,nc,2);
    imagesc(Lmx);
```

```
from sklearn.cluster import KMeans
from skimage.io import imread

#%% -----   Load & Reshape  -------
Irgb    = imread('C:/IMGdown/SILU/Images/airplane/airplane00.tif')
(m,n,nCh) = Irgb.shape
nPix    = m*n
```

```
ICol    = Irgb.reshape((nPix,3)).astype(float)

#%% =======     Kmeans      =========
nK      = 4
RKM     = KMeans(n_clusters=nK, max_iter=20, n_init=1).fit(ICol)
Cen     = RKM.cluster_centers_      # [nK 3]
LbK     = RKM.labels_               # [nPix,1] E [0..nK-1]
```

```
import cv2 as cv
from numpy import float32

# ------    load image  ------
fp      = 'C:/someImage.tif'
Ibgr    = cv.imread(fp, cv.IMREAD_COLOR) # loads channels as BGR: blue/green/red
# ------    reshape to column-wise [nPix 3]     -------
szI     = Ibgr.shape
nPix    = szI[0]*szI[1]
ICOL    = Ibgr.reshape((nPix,3)).astype(float32)

#%% =======     Kmeans      =========
k           = 3         # number of desired clusters
nRep        = 1         # number of repetitions
termCrit    = (cv.TERM_CRITERIA_MAX_ITER, 10, 0.1)     # iter, epsilon
flagInit    = cv.KMEANS_PP_CENTERS                     # kmeans++
mes, Lb, Cen= cv.kmeans(ICOL, k, None, termCrit, nRep, flagInit)
BW          = Lb[:,0].reshape(szI[0],-1)     # labels to map
```

L.10 Shape

In subsection L.10.1 we create a set of shapes. In subsection L.10.2 we measure their similarity with simple properties; in subsection L.10.3 we use the radial signature to provide better similarity performance.

L.10.1 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);
    text(i*sz,140,char(ix),'fontweight','bold','fontsize',16);
end
print('ShapeLetters','-djpeg','-r300');
```

```
from numpy             import ones, arange
from matplotlib.pyplot import *

I       = ones((150,760))       # empty image
I[0,0]  = 0                     # to ensure that the colormap uses its range
sz      = 28
Ix      = arange(65,91)         # indices for A to Z
nShp    = len(Ix)
figure(figsize=(12,5))
imshow(I,cmap=cm.gray); axis('off')
xoff    = 8
for i in arange(0,nShp):
    ix  = Ix[i]
    text(i*sz+xoff,20, chr(ix),fontweight='bold')
    text(i*sz+xoff,50, chr(ix),fontweight='bold',fontsize=12)
    text(i*sz+xoff,110,chr(ix),fontweight='bold',fontsize=12,rotation=45)
    text(i*sz+xoff,80, chr(ix),fontweight='bold',fontsize=14)
    text(i*sz+xoff,140,chr(ix),fontweight='bold',fontsize=16)

savefig('ShapeLettersPy.jpeg', facecolor='w', bbox_inches='tight')
```

L.10.2 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\n', 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);
    Szs     = size(Sbw);
    if i<=nShp2,ID1((1:Szs(1))+(i-1)*mxHgt,    1:Szs(2)) = Sbw*2;
    else        ID2((1:Szs(1))+(i-nShp2)*mxHgt,1:Szs(2)) = Sbw*2;
    end

    % --- similar shapes in rows 2nd to nShp+1
    for k = 1:nSim
        ix      = O(i,k);
      Row     = (1:Hgt(ix))+UL(ix,2);   % rows
        Col     = (1:Wth(ix))+UL(ix,1);   % columns
        Sbw     = BW(Row,Col);
        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');
```

```
from numpy import asarray, asmatrix, concatenate, nan, diagflat, floor, ceil, \
                  arange, zeros, ones, ix_, sort, argsort
from skimage.io        import imread
from skimage.color     import rgb2gray
from skimage.measure   import label, regionprops
from scipy.spatial.distance import pdist, squareform

BW      = rgb2gray(imread('ShapeLettersPy.jpeg')) < 0.3

#%% =====    Shape Properties   =====
LB      = label(BW) #BW.astype(int))
RG      = regionprops(LB)
nShp    = len(RG)
print('# Shapes', nShp)

#%% =====   Create Attribute Vectors    ======
# we use 'asmatrix' instead of 'asarray' to create matrix 'Vec'
Ara     = asmatrix([r.area for r in RG])         # called list comprehension
Ecc     = asmatrix([r.eccentricity for r in RG])
EqD     = asmatrix([r.equivalent_diameter for r in RG])
Vec     = concatenate((Ara,Ecc,EqD)).conj().T   # [nShp 3] three dimensions

#%% =====    Pairwise Distance Measurements
DM      = squareform(pdist(Vec))                # distance matrix [nShp nShp]
DM[diagflat(ones((1,nShp),dtype=bool))] = nan   # inactivate own shape
DO      = sort(DM,axis=1)                       # sort along rows
```

```
O       = argsort(DM,axis=1)                    # obtain indices separately

#%% -----    Plotting First nSim Similar Shapes for Each Found Shape
BBx     = asarray([r.bbox for r in RG])     # Bounding box
UL      = floor(BBx[:,0:2])                 # upper left corner
#LR      = floor(BBx[:,2:4])                 # lower rite corner
Hgt     = BBx[:,2]-BBx[:,0]                 # width
Wth     = BBx[:,3]-BBx[:,1]                 # height
mxHgt   = max(Hgt)+2
mxWth   = max(Wth)+2
nSim    = 20                                # similar ones we plot
nShp2   = int(ceil(nShp/2))
ID1     = zeros((nShp2*mxHgt,nSim*mxWth))
ID2     = ID1.copy()
for i in range(nShp):
    # --- given/selected shape in 1st row
    Row    = arange(0,Hgt[i])+UL[i,0]-1    # rows
    Col    = arange(0,Wth[i])+UL[i,1]-1    # columns
    Sbw    = BW[ix_(Row.astype(int), Col.astype(int))]
    Szs    = Sbw.shape
    RgV    = arange(0,Szs[0])           # vertical range
    RgH    = arange(0,Szs[1])           # horizontal range
    if i < nShp2: ID1[ix_(RgV+i*mxHgt,         RgH)] = Sbw*2
    else:         ID2[ix_(RgV+(i-nShp2)*mxHgt, RgH)] = Sbw*2

    # --- similar shapes in rows 2nd to nShp+1
    for k in range(nSim-1):
        ix      = O[i,k]
        Row     = arange(0,Hgt[ix])+UL[ix,0]-1    # rows
        Col     = arange(0,Wth[ix])+UL[ix,1]-1    # columns
        Sbw     = BW[ix_(Row.astype(int),Col.astype(int))]
        Szs     = Sbw.shape
        RgV     = arange(0,Szs[0])              # vertical range
        RgH     = arange(0,Szs[1])              # horizontal range
        if i < nShp2:   ID1[ix_(RgV+i*mxHgt,        RgH+(k+1)*mxWth)] = Sbw
        else:           ID2[ix_(RgV+(i-nShp2)*mxHgt,RgH+(k+1)*mxWth)] = Sbw

from matplotlib.pyplot import *
figure(figsize=(20,20))
subplot(1,2,1); imshow(ID1); title('First Half of Letters')
subplot(1,2,2); imshow(ID2); title('Second Half of Letters')
```

L.10.3 Shape: Radial Signature

```
clear;
BW      = imread('text.png');       % the stimulus
aBonImg = bwboundaries(BW);
nShp    = length(aBonImg);
fprintf('# Shapes %d\n', 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
    FDn     = Fdsc(2:end)/Fdsc(1);      % normalization by 1st FD
    fprintf('%3d  #pix %2d  FFD %2d\n', i, nPix, length(FDn));
    Vec(i,:) = FDn(1:4);
    aBon{i} = bsxfun(@minus,Bon,cenPt-[10 10]);  % move into [1..20 1..20]
    if 0
        figure(1); imagesc(zeros(20,20)); hold on;
```

```
        plot(aBon{i}(:,2), aBon{i}(:,1));
        pause();
    end
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      = O(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))+k*mxWth) = 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))+k*mxWth) = Sbw;
        end
    end
end
```

L.11 Contour

L.11.1 Edge Following (Boundary Tracing)

Contour tracing is easiest carried out by firstly thinning the black-white map. We've shown how to do that in the last few lines of the code example in L.4.2. Here we use a toy-example to illustrate boundary tracing.

```
% Demonstrates the difference between bwboundaries and bwtraceboundary
% 1 pix:    2 points
% 2 pix:    3 points
clear;
% ------ BW image with some stimuli
M               = zeros(10,11); % an empty map
M(2,2)          = 1;            % a single pixel
M(4:5,2)        = 1;            % two pixels
M([2 5],4:7)=1;   M(2:5,[4 7])=1;       % rectangle
M(7,4:6)=1;   M(7:9,5)=1;               % T-feature
M(8,8:9)=1;   M(7,9)=1;   M(9,10)=1;    % Y-feature at 45 deg

%% =====    BwBoundaries    ======
[aBon Lmx nBon] = bwboundaries(M,'noholes');% finds ALL boundaries
fprintf('Found %d boundaries\n',nBon);

%% =====    Individual Tracing  ======
Bsin1   = bwtraceboundary(M, [2 2], 'E');   % finds a SINGLE boundary
Bsin2   = bwtraceboundary(M, [4 2], 'S');   % going south
Bsin3   = bwtraceboundary(M, [2 4], 'E');   %
Bsin4   = bwtraceboundary(M, [7 4], 'E');   %

%% -----    Plot    -----
figure(1);clf;
imagesc(Lmx); hold on;
% ---   First from bwboundaries...
for i = 1:nBon
    Bon     = aBon{i};
    nPx     = size(Bon,1);
    plot(Bon(:,2),  Bon(:,1), 'r');
    fprintf('%d: #pix %d\n',i,nPx);
end
% ---   ...then from bwtraceboundary:
plot(Bsin1(:,2)+.2, Bsin1(:,1)+.2,'g');
plot(Bsin2(:,2)+.2, Bsin2(:,1)+.2,'g');
plot(Bsin3(:,2)+.2, Bsin3(:,1)+.2,'g');
plot(Bsin4(:,2)+.2, Bsin4(:,1)+.2,'g');

ts  = sprintf('%d', nBon);
titleb(ts,10);
```

```
"""
Closes contour as in Matlab
Finds inside contours (holes) as well
4-connected by default
Coordinates range from [0 sz-1] (not [1 sz] as in Matlab)
1 pix:    5 points
2 pix:    7 points
"""
from numpy import zeros, arange
from skimage.measure import find_contours
from matplotlib.pyplot import *

# ------ BW image with some stimuli
M  = zeros((10,11),bool)
M[1,1]        = 1                       # a single pixel
M[3:5,1]      = 1                       # two pixels
M[[1,4],3:7]=1;  M[1:5,[3,6]]=1         # rectangle
M[6,3:6]    =1;  M[6:9,4]    =1         # T-feature
```

```
M[7,7:9]    =1;  M[6,8]=1;   M[8,9]=1;  # Y-feature at 45 deg

#%% =====    Tracing    =======
aBon    = find_contours(M, 0.99)   # closer to contour pixels
aBonE   = find_contours(M, 0.2)    # closer to exterior pixels

#%% -----    Plotting    -------
figure(1)
imshow(M, interpolation='nearest')
nB = len(aBon)
for i in arange(0,nB):
    Bon     = aBon[i]
    BonE    = aBonE[i]
    print('#pix  %d, %d'%(Bon.shape[0],BonE.shape[0]))
    plot(Bon[:,1], Bon[:,0], 'r')
    plot(BonE[:,1], BonE[:,0], 'y')
```

For the OpenCV example (accessed through Python) we omit the plotting part for brevity.

```
"""
Boundary tracing in OpenCV. Results are more similar to Matlab than Python.
Does not close contour (as in Matlab or Python)
Coordinates range from [0 sz-1] (not [1 sz] as in Matlab)
1 pix:     1 point
2 pix:     2 points
"""
import cv2
from numpy import uint8, zeros

# ------ BW image with some stimuli
M  = zeros((10,11),bool)
M[1,1]        = 1                       # a single pixel
M[3:5,1]      = 1                       # two pixels
M[[1,4],3:7]=1;  M[1:5,[3,6]]=1         # rectangle
M[6,3:6]    =1;  M[6:9,4]    =1         # T-feature
M[7,7:9]    =1;  M[6,8]=1;   M[8,9]=1;  # Y-feature at 45 deg

#%% ---------   Measure     --------
BWout, aCnt, Hier   = cv2.findContours(M.astype(uint8), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

cCnt    = len(aCnt)
print(f'#cnt {cCnt}')
# squeeze middle axis and change from x/y to row/col
for i in range(cCnt): aCnt[i]=aCnt[i][:,0][:,[1,0]]
for i in range(cCnt):
    print(aCnt[i].shape[0])
```

L.11.2 Curvature Signature

```
% Curvature estimated with 2nd derivative. Improvised version.
clear;

%% -----    Boundaries from rice image
I           = imread('rice.png');
BW          = im2bw(I, graythresh(I));
[aBon,L]   = bwboundaries(BW,'noholes');
% -----    Take 1st boundary:
Bon         = aBon{1};              % use row/col (matrix) axis
Ro          = Bon(:,2);             % original row (for plotting)
Co          = Bon(:,1);             % original column
% adds inter-point spacing as 3rd column:
Bon         = [Bon [0; sqrt(sum(diff(Bon,1).^2,2))]];

%% -----    Low-PassFilter (Gaussian)   ------
```

```
sgm         = 2;
w           = 5;
Lpf         = normpdf(-w:w,0,sgm)';     % Gaussian at 0 with sigma=1

%% =====    Low-Pass Filter to Suppress Noise   =======
Rf          = conv(Ro,Lpf,'same');
Cf          = conv(Co,Lpf,'same');
Rf          = [Ro(1:w); Rf(w+1:end-w); Ro(end-w:end)];
Cf          = [Co(1:w); Cf(w+1:end-w); Co(end-w:end)];

%% =====    1st Derivative      =======
Y1          = [0; diff(Rf)];
X1          = [0; diff(Cf)];

%% =====    Curvature Improvised    =====
Dv1     = Y1+X1;                    % adding 1st derivatives
Drv1    = conv(Dv1,Lpf,'same');     % low-passfilter again (to smoothen)
Drv2    = abs([0; diff(Drv1)]);    % 2nd dervivative: that is our curvature

% --- find maximum curvature
[v,ixMx] = max(Drv2);
ixMxs   = ixMx-1;                   % it's shifted by one on curve (plotting)

%% -----    Plot    -------
figure(1); clf;

subplot(2,2,1);
hold on;
plot(Bon(:,1),      Bon(:,2));
plot(Cf,Rf);
plot(Bon(ixMxs,1),   Bon(ixMxs,2),'r^');
legend('original','low-pass filtered','max curvature','location','northwest');
plot(Cf(ixMxs),Rf(ixMxs),'r^');
plot(Bon(1,1),Bon(1,2),'*');        % starting pixel
axis equal;

subplot(2,2,2); hold on;
ArcLength   = [0; cumsum(Bon(:,3))];
plot(ArcLength,Drv1,'g');
plot(ArcLength,Drv2,'r');
plot(ArcLength(ixMx),Drv2(ixMx),'r^');
ylabel('Amplitude');
xlabel('Arc Length');
legend('1st Deriv','2nd Deriv');
```

L.12 Tracking

L.12.1 Background Subtraction

Two examples of background subtraction, one for Matlab, one for OpenCV (through Python).

```
clear;
Vid.reader  = vision.VideoFileReader('atrium.avi');
%% --------------    Init Detection    --------------
ForeGnd     = vision.ForegroundDetector('NumGaussians', 3, ...
                'NumTrainingFrames', 40, 'MinimumBackgroundRatio', 0.7);
BlobAly     = vision.BlobAnalysis('BoundingBoxOutputPort', true, ...
                'AreaOutputPort', true, 'CentroidOutputPort', true, ...
                'MinimumBlobArea', 400);
%% --------------    Init Plotting    --------------
Play.Vido = vision.VideoPlayer('Position', [20, 400, 700, 400]);
Play.Mask = vision.VideoPlayer('Position', [740, 400, 700, 400]);

%% SSSSSSSSSSSSSSSSSSS    LOOP FRAMES     SSSSSSSSSSSSSSSSSSS
while ~isDone(Vid.reader)
    Frm      = Vid.reader.step();        % read one frame

    Msk         = ForeGnd.step(Frm);        % foreground [m n nCh]
    % remove noise and fill in holes
    Msk         = imopen(Msk,  strel('rectangle', [3,3]));
    Msk         = imclose(Msk, strel('rectangle', [15, 15]));
    Msk         = imfill(Msk, 'holes');
    [~, CenDet, BbxDet] = BlobAly.step(Msk);   % region analysis

    %% --------------   Plot    --------------
    % convert the frame and the mask to uint8 RGB.
    Frm     = im2uint8(Frm);
    Msk     = uint8(repmat(Msk, [1, 1, 3])) .* 255;
    % place rectangles around detections
Frm     = insertObjectAnnotation(Frm, 'rectangle', BbxDet, ' ');
  Msk     = insertObjectAnnotation(Msk, 'rectangle', BbxDet, ' ');
    % put on screen
    Play.Mask.step(Msk);
    Play.Vido.step(Frm);
end
```

```
import cv2 as cv

#%% ----------------    Video Info      -------------
vidName = 'C:/Program Files/MATLAB/MATLAB Production Server/R2015a/toolbox/vision/visiondata/atrium.avi'
Vid     = cv.VideoCapture(vidName)
Vid.set(cv.CAP_PROP_POS_MSEC, 2*1000)       # set starting time
nFrm    = Vid.get(cv.CAP_PROP_FRAME_COUNT)  # obtain total # of frames

#%% ---------------     Init Background & Detection     -------------
BckSub  = cv.createBackgroundSubtractorKNN()
StcElm  = cv.getStructuringElement(cv.MORPH_ELLIPSE,(3,3)) # for filtering

#%% ===============     LOOP FRAMES       ================
cF=0
while cF < 500:
    cF      += 1
    r,Frm   = Vid.read()          # reads one frame

    Msk  = BckSub.apply(Frm)      # generates a mask
    Msk  = cv.morphologyEx(Msk, cv.MORPH_OPEN, StcElm)  # filter for large patches

    # ----  Plot    -----
    cv.imshow('frame', Msk)
    k   = cv.waitKey(30) & 0xff

Vid.release()
```

```
cv.destroyAllWindows()
```

L.12.2 Maintaining Tracks

This example is a rewritten version of the example provided by Matlab (“Motion-Based Multiple Object Tracking”).

```
% depends on: predict, distance, correct, configureKalmanFilter and
%                       assignDetectionsToTracks
% Detected 'salient' regions are called objects here.
clear;
Vid.reader   = vision.VideoFileReader('atrium.avi');
%% --------------    Init Detection    --------------
Det.ForeGnd  = vision.ForegroundDetector('NumGaussians', 3, ...
    'NumTrainingFrames', 40, 'MinimumBackgroundRatio', 0.7);
Det.BlobAly  = vision.BlobAnalysis('BoundingBoxOutputPort', true, ...
    'AreaOutputPort', true, 'CentroidOutputPort', true, ...
    'MinimumBlobArea', 400);
% ---   more parameters:
costNonAss      = 20;  % cost of non-assignment
durMaxOff     = 20; % maximum off/invisible duration [frames]
durMinTot    = 8;    % minimum total duration        [frames]
% ---track structure:
StcTrk  = struct('id', [], 'bbox', [], ...          % id-number, bound-box
           'KalFlt', [], ...                        % Kalman filter
           'durTot', 1, 'durOnn', 1, 'durOff', 0); % durations
aTrk    = [];           % track list
cIdT    = 0;            % counter track identity
%% --------------    Init Plotting    --------------
durMinOnnPlot       = 8;
Play.Vido = vision.VideoPlayer('Position', [20, 400, 700, 400]);
Play.Mask = vision.VideoPlayer('Position', [740, 400, 700, 400]);

%% SSSSSSSSSSSSSSSSSSS    LOOP FRAMES     SSSSSSSSSSSSSSSSSSS
cF=0;
while ~isDone(Vid.reader)
    Frm      = Vid.reader.step();        % read one frame
    cF          = cF+1;
    %% SSSSSSSSSSSSSSSS DETECT      SSSSSSSSSSSSSSSSSSS
    Msk         = Det.ForeGnd.step(Frm); % foreground [m n nCh]
    % remove noise and fill in holes
    Msk  = imopen(Msk, strel('rectangle', [3,3]));
    Msk     = imclose(Msk, strel('rectangle', [15, 15]));
    Msk     = imfill(Msk, 'holes');
    [~, CenDet, BbxDet] = Det.BlobAly.step(Msk);   % region analysis

    %% SSSSSSSSSSSSSSSS TRACKS      SSSSSSSSSSSSSSSSSSS
    nTrk    = length(aTrk);
    %% =========    Predict Next Location    ==========
    for t = 1:nTrk
        bbox        = aTrk(t).bbox;
        cenPred     = predict(aTrk(t).KalFlt); % next location
        % shift bounding box to that prediction
        cenPred     = int32(cenPred) - bbox(3:4) / 2;
        aTrk(t).bbox = [cenPred, bbox(3:4)];
    end
    %% =========    Assign Detections to Tracks   ==========
    % cost matrix is quasi a distance matrix
    nDet    = size(CenDet,1);
    Cost    = zeros(nTrk, nDet);
    for t = 1:nTrk
        Cost(t,:)   = distance(aTrk(t).KalFlt, CenDet);
    end
    % solve the assignment problem
    [TrkAss TrkUna ObjUn] = assignDetectionsToTracks(Cost, costNonAss);
```

```
%% =========    Update Assigned Tracks    =========
for t = 1:size(TrkAss,1)
    ixT  = TrkAss(t,1);  % index track
    ixO     = TrkAss(t,2);  % index detection/object
    % replace predicted bbox with detected one
    aTrk(ixT).bbox  = BbxDet(ixO,:);
    % correct the estimate of the object's location
    % using the new detection.
    correct(aTrk(ixT).KalFlt, CenDet(ixO,:));

    aTrk(ixT).durTot = aTrk(ixT).durTot + 1; % duration (age)
    aTrk(ixT).durOnn  = aTrk(ixT).durOnn + 1; % duration visible
    aTrk(ixT).durOff    = 0;                    % reset to 0
end

%% =========    Update Unassigned   ========
for t = 1:length(TrkUna)
    ix    = TrkUna(t);
    aTrk(ix).durTot = aTrk(ix).durTot + 1;
    aTrk(ix).durOff = aTrk(ix).durOff + 1;
end

%% =========    Delete Lost/Unusual     ========
if ~isempty(aTrk)
    DurTot      = [aTrk(:).durTot]; % total duration    [nTrk 1]
    DurOnn      = [aTrk(:).durOnn]; % duration visible  [nTrk 1]
    DurOff      = [aTrk(:).durOff]; % duration invisible [nTrk 1]
    PropOn      = DurOnn ./ DurTot; % proportion visible/on
    % eliminate unusual/lost tracks:
    Blost = (DurTot<durMinTot & PropOn<0.6)  |  DurOff>=durMaxOff;
    aTrk(Blost) = [];               % delete lost tracks
end

%% =========    Create New     ========
CenNew  = CenDet(ObjUn,:);        % unassigned centers
BbxNew  = BbxDet(ObjUn,:);
% ======    Loop newly detected objects   ======
for i = 1:size(CenNew,1)
    KalFlt = configureKalmanFilter('ConstantVelocity', ...
        CenNew(i,:), [200, 50], [100, 25], 100);
    % create a new track & add to list
    cIdT        = cIdT + 1;     % increment the next id.
    newTrk      = StcTrk;
    newTrk.id   = cIdT;
    newTrk.bbox = BbxNew(i,:);
    newTrk.KalFlt = KalFlt;
    if cIdT==1,aTrk        = newTrk;
    else      aTrk(end+1) = newTrk;   end
end

%% --------------   Plot    --------------
% convert the frame and the mask to uint8 RGB.
Frm     = im2uint8(Frm);
Msk     = uint8(repmat(Msk, [1, 1, 3])) .* 255;
if ~isempty(aTrk)

    % Noisy detections tend to result in short-lived tracks.
    % Only display tracks that have been visible for more than
    % a minimum number of frames.
    IxReliable  = [aTrk(:).durOnn] > durMinOnnPlot;
    aTrkRel     = aTrk(IxReliable);

    % Display the objects. If an object has not been detected
    % in this frame, display its predicted bounding box.
    if ~isempty(aTrkRel)

        BbxRel  = cat(1, aTrkRel.bbox);
```

```
            ids     = int32([aTrkRel(:).id]);

            % Create tags for objects indicating the ones for
            % which we display the predicted rather than the actual
            % location.
            Tags    = cellstr(int2str(ids'));
            IxPred  = [aTrkRel(:).durOff] > 0;
            isPredicted = cell(size(Tags));
            isPredicted(IxPred) = {' predicted'};
            Tags    = strcat(Tags, isPredicted);

            Frm = insertObjectAnnotation(Frm, 'rectangle', BbxRel, Tags);
            Msk = insertObjectAnnotation(Msk, 'rectangle', BbxRel, Tags);
        end
    end
    % put on screen
    Play.Mask.step(Msk);
    Play.Vido.step(Frm);
end
```

L.12.3 Face Tracking with CAM shift

This example is a rewritten version of the example provided by Matlab ("Face Detection and Tracking Using CAMShift").

```
% Face tracking using CAMshift with histograms. The exact object to be
% tracked is the nose, because it does not contain any (distracting) back-
% ground pixels - it makes tracking more reliable. The hue channel of the
% HSV space is used, as it represents skin tones better (tendentially).
clear;
%% =========    Detect the face in the 1st frame    =======
DetFace         = vision.CascadeObjectDetector('FrontalFaceCART');
VidReader       = vision.VideoFileReader('visionface.avi');
Frm             = step(VidReader);              % a single frame
BboxFace        = step(DetFace, Frm);           % bounding box [1 4]
DetNose         = vision.CascadeObjectDetector('Nose', 'UseROI', true);
BboxNose        = step(DetNose, Frm, BboxFace(1,:));
% ----  Plot  ----
figure(1); imshow(Frm);
rectangle('Position',BboxFace(1,:), 'EdgeColor',[1 1 0])
rectangle('Position',BboxNose(1,:), 'EdgeColor',[1 0 0])
pause(.5);
%% --------     Init tracker  ---------
Trk             = vision.HistogramBasedTracker; % init tracker object
[Mhue,~,~]      = rgb2hsv(Frm);
initializeObject(Trk, Mhue, BboxNose(1,:));  % hue pixs from nose
% ----  Plot  ----
figure(2); clf; imshow(Mhue); title('Hue Channel Map');
rectangle('Position',BboxFace(1,:), 'EdgeColor',[1 1 0])
pause(.5);
%% ========     Tracking the face   ===========
% ----  Init plotting
VidInfo     = info(VidReader);
VidPlayer   = vision.VideoPlayer('Position',[300 300 VidInfo.VideoSize+30]);
% ----  Tracking
while ~isDone(VidReader)
    Frm         = step(VidReader);
    [Mhue,~,~]  = rgb2hsv(Frm);
    BboxFnow    = step(Trk, Mhue);      % new face bounding box
    % ----  Plot    ----
    Fann        = insertObjectAnnotation(Frm,'rectangle', BboxFnow, 'Face');
    step(VidPlayer, Fann);
end
% ---- release objects
release(VidReader);
```

```
release(VidPlayer);
```

L.12.4 Tracking-by-Matching

```
import cv2 as cv

#%% ------------    Video Prep      -------------
vidPth  = 'C:/Program Files/MATLAB/MATLAB Production Server/R2015a/toolbox/vision/visiondata/'
vidName = 'handshake_left.avi'
Vid     = cv.VideoCapture(vidPth + vidName)

#%% -----------     Find 1st BoundingBox   -------------
ret,Frm     = Vid.read()
PedDet      = cv.HOGDescriptor()
PedDet.setSVMDetector(cv.HOGDescriptor_getDefaultPeopleDetector())
(aPers, Like) = PedDet.detectMultiScale(Frm, winStride=(2,2),
                            padding=(8,8), scale=1.05)
x1,y1,w,h   = aPers[0]
bbox        = (x1,y1,w,h)  # upper left point, width, height

#%% -----------     Init Tracker    -------------
Trkr    = cv.TrackerKCF_create()
bOk     = Trkr.init(Frm, bbox)

#%% ===============     TRACK       ================
cF=1
while cF < 55:
    cF      += 1
    ret,Frm  = Vid.read()
    if ret==False: break

    bOk, bbox = Trkr.update(Frm)

    # ----  Plot    -----
    p1 = (int(bbox[0]), int(bbox[1]))
    p2 = (int(bbox[0] + bbox[2]), int(bbox[1] + bbox[3]))
    Idem = cv.rectangle(Frm, p1, p2, 255,2)
    cv.imshow('Mask',Idem)
    k = cv.waitKey(60) & 0xff
    if k == 27:
        break

cv.destroyAllWindows()
Vid.release()
```

L.13 Optic Flow

As discussed in Section 15.

```
clear;
Vid     = VideoReader('viptraffic.avi');
%% -----    3 Methods:
%OptFlow = opticalFlowHS;
%OptFlow = opticalFlowLK('NoiseThreshold',0.009);
OptFlow = opticalFlowLKDoG('NumFrames',3);

%% SSSSSSSSSSSSS   LOOP FRAMES     SSSSSSSSSSSSSS
while hasFrame(Vid)
    Frgb    = readFrame(Vid);
    Fgry    = rgb2gray(Frgb);
    Flow    = estimateFlow(OptFlow, Fgry);

    imshow(Frgb)
    hold on
    plot(Flow, 'DecimationFactor',[5 5], 'ScaleFactor',25)
    hold off

end
```

```
import numpy as np
import cv2 as cv
#%% ------------    Prepare Video      ------------
vidName = 'someVideo.mp4'
Vid     = cv.VideoCapture(vidName)
Vid.set(cv.CAP_PROP_POS_MSEC, 40*1000)  # set to appropriate starting time

#%% ------------    Parameters      ------------
# corner detection, ShiTomasi technique
PrmCorDet    = dict(maxCorners = 500,
                    qualityLevel = 0.5,
                    minDistance = 7,
                    blockSize = 7 )
# optic flow, LucasKanade method
PrmOptFlo   = dict(winSize  = (15,15),
                   maxLevel = 2,
                   criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 10, 0.03))

#%% ============    1st frame ============
rt, Fold = Vid.read()       # we call the frame old already
Iold     = cv.cvtColor(Fold, cv.COLOR_BGR2GRAY)
Pts0     = cv.goodFeaturesToTrack(Iold, mask = None, **PrmCorDet)

# --- prepare plotting:
Msk     = np.zeros_like(Fold)
Cols    = np.random.randint(0,255,(100,3))  # list of colors
#%% SSSSSSSSSSSSSSSSSSS    LOOP FRAMES     SSSSSSSSSSSSSSSSSSS
while(1):
    rt,Fnew = Vid.read()        # new frame
    Inew    = cv.cvtColor(Fnew, cv.COLOR_BGR2GRAY)

    # === Calculate optical flow ===
    Pts1, St, Err = \
        cv.calcOpticalFlowPyrLK(Iold, Inew, Pts0, None, **PrmOptFlo)
    # === Select good points (status==1) ===
    PtsNew = Pts1[St==1]
    PtsOld = Pts0[St==1]
    # === Update ===
    Iold = Inew.copy()  # new (gray-scale) frame becomes old one
    Pts0 = PtsNew.reshape(-1,1,2)

    # --------   Plot  --------
    # draw the tracks
```

```
    for i,(new,old) in enumerate(zip(PtsNew, PtsOld)):
        a,b  = new.ravel()
        c,d  = old.ravel()
        Msk  = cv.line(Msk, (a,b),(c,d), Cols[i].tolist(), 2)
        Fnew = cv.circle(Fnew,(a,b),5, Cols[i].tolist(), -1)
    Idisp = cv.add(Fnew, Msk)
    cv.imshow('frame',Idisp)
    # keystroke input
    k = cv.waitKey(30) & 0xff
    if k == 27:
        break

cv.destroyAllWindows()
Vid.release()
```

L.14 2D Transformations

We chose as shape a distorted (self-intersecting) rectangle, specified with coordinates in `Co`. In section 'Transforming the Shape' it is then transformed in various ways resulting in shape coordinates `Crot`, `Cshr`, etc. We then try to estimate the motions, once for similarity and once for affinity for which we build the corresponding variables `Asim`, `Aaff` and `bsim` and `baff`, respectively. The estimation procedure is done with `lsqlin` or `lsqnonneg`, whereby $\mathbf{A}$ and $\mathbf{b}$ are passed as arguments. At the very end we use Matlab's function scripts `fitgeotrans` and `procrustes`.

```
% Examples of 2D transforms and their motion estimation. Sze pdf 36, 312.
clear;
%% ------   Stimulus: a distorted rectangle
Co  = [0 0; 0 .8; 0.6 0.75; 1.2 .8; 1.0 0.9; 1.2 0; 0 0]*4+2;
np  = size(Co,1);       % # of points
cpt = mean(Co,1);       % center point

%% SSSSSSSSSSSSSSSSS   Transforming the Shape     SSSSSSSSSSSSSSSSS
% --- choosing some parameters
r   = 0.25;             % rotation/scale
tx  = 0.5;              % x-translation
ty  = 1.5;              % y-translation
a   = 0.25;             % scale (for similarity)
b   = 0.25;             % rotation (for similarity)
[a00 a01 a10 a11] = deal(-0.5, 0.25, 0.125, -0.75); % for affinity
% -----------preparing 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     ------------
Cpt = repmat(cpt,np,1);                 % replicate center point
% 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         "
% --- 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;

%% -----------      Plotting        -----------
figure(1); clf; [rr cc] = deal(1,1); hold on;
plot(Co(:,1), Co(:,2), 'color', 'k', 'linewidth', 2);
plot(Crot(:,1), Crot(:,2), 'color', 'g');
plot(Cshr(:,1), Cshr(:,2), 'color', 'r');
plot(Cscl(:,1), Cscl(:,2), 'color', 'b');
plot(Caff(:,1), Caff(:,2), 'color', 'm');
plot(Ceuc(:,1), Ceuc(:,2), 'color', 'c');
plot(Csim(:,1), Csim(:,2), 'color', ones(3,1)*.5);
axis equal
legend('original', 'rotation', 'shear', 'scaling', 'affine', 'rigid', 'similar',...
'location', 'northwest');
set(gca, 'xlim', [0 8], 'ylim', [0 8]);

%% SSSSSSSSSSSSSSSSS   Estimate Motion   SSSSSSSSSSSSSSSSS
%% =========    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]');
% JEuc = 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
DltSim  = Csim(:,[1 2])-Coc;        % delta (transformed-original 0,0 centered)
DltAff  = Caff(:,[1 2])-Coc;        % delta (transformed-original 0,0 centered)
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;
    Jp      = JAff(pt(1),pt(2));
    Aaff   = Aaff + Jp'*Jp;
    baff    = baff + DltAff(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');
[Prm resn] = lsqlin(Aaff, baff'); % Prm(3:6) contain estimates for a00, a01, a10, a11
Prm'
(Prm(1:2)-cpt')' % tx and ty

%% ---------    FitGeoTransform         ----------
Tffit               = fitgeotrans(Co,Csim(:,1:2),'similarity');

%% ---------    Applying Procrustes     ----------
[d CoProc Tproc]    = procrustes(Co, Csim(:,1:2), 'reflection', false);

figure(2); clf; hold on;
plot(Co(:,1), Co(:,2), 'color', 'k', 'linewidth', 2);
plot(Csim(:,1), Csim(:,2), 'color', 'm');
plot(CoProc(:,1), CoProc(:,2), 'r');
```

L.15 RanSAC

Example of a primitve version of the Random Sampling Consensus algorithm. Below are two scripts: a function script `f_RanSaC` and a testing script that runs the function.

```
% 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\n', np1, np2);
nMinPt      = Opt.nMinPts;              % # of minimum pts required for transformation
nCom        = np1-nMinPt;               % # of complement pts
if ~nCom,  warning('no complementing points');  end
cpt         = mean(Pt1);                % center point of original
Pt1Cen      = Pt1 - repmat(cpt,np1,1);  % original 0,0 centered

%% ====================    Iterating    ====================
cIter       = 0;
nGoodFits = 0;
[Prm Err]   = deal([]);
while cIter < Opt.nMaxIter

    %% ------ 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));
        Atot    = Atot + 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
    fprintf('tx %1.4f  ty %1.4f ', tx, ty);

    %% ==========   Transform Complement
    [a00 a01 a10 a11] = deal(prm(3), prm(4), prm(5), prm(6));
    TAff        = [1+a00 a01 tx;  a10 1+a11 ty;  0 0 1]; % affinity (with row extended)
    Pt1Com      = Pt1Cen(Ixcm,:);                   % the complement pts
    Caff        = [Pt1Com ones(nCom,1)] * TAff';    % we add a col of 1s
    Caff(:,1:2) = Caff(:,1:2)+repmat(cpt,nCom,1);   % THEN add cenpt

    %% ==========   Close enough?
    DfCo    = Caff(:,1:2)-Pt2(Ixcm,:);              % [nCom,2]
    Di      = sqrt(sum(DfCo.^2,2));
    bNear   = Di < Opt.thrNear;
    nNear   = nnz(bNear);
    fprintf('%3d near of %4d  err %12.10f', nNear, nCom, err);
    if nNear >= Opt.nNear
        nGoodFits = nGoodFits + 1;
        % =========   Refit   =========
        Pt2n    = Pt2(Ixcm(bNear),:);
        Pt1Cenn = Pt1Cen(Ixcm(bNear),:);
        Dltn    = Pt2n - Pt1Cenn;
        % --- building A and b
        A2      = zeros(6,6);  b = 0;           % init Atot and b
        for i = 1:nNear
            pt      = Pt1Cenn(i,:);
            Jp      = JAff(pt(1),pt(2));
            A2      = A2 + Jp'*Jp;              % building A
            b       = b  + Dltn(i,:)*Jp;        % building b
        end
        % --- least squares:
        [prm2 err2] = lsqlin(double(A2), double(b'));
        [tx ty]     = deal(prm(1)-cpt(1), prm(2)-cpt(2));   % tx and ty
        Prm         = [Prm; [tx ty] prm2(3:end)'];
        Err         = [Err; err2];
    end

    %% ==========   Plotting
    if b_plot.dist
        Dis = sort(Di, 'ascend');
        figure(10); clf; [rr cc] = deal(1,2);
        subplot(rr,cc,1);
            plot(Dis);
            set(gca, 'ylim', [0 1500]);
            title([Opt.xlb ' ' num2str(cIter)]);
            xlabel(Opt.Match, 'fontweight', 'bold', 'fontsize', 12);
        subplot(rr,cc,2);
            plot(Dis); hold on;
            set(gca, 'xlim', [0 nCom*0.25]);
            set(gca, 'ylim', [0 Opt.thrNear*2]);
            plot([0 nCom], ones(1,2)*Opt.thrNear, 'k:');
        pause();
    end

    cIter   = cIter + 1;
    fprintf('\n');
end
TP.PrmEst       = Prm;
```

```
TP.ErrEst       = Err;
TP.nGoodFits    = nGoodFits;
fprintf('#GoodFits %d out of %d iterations\n', nGoodFits, Opt.nMaxIter);
end % function
```

This is the testing script calling the above function:

```
%% ==================== TESTING SCRIPT t_Ransac
% Testing 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 cenpt: 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;
CoTrns  = Csim(:,[1 2]) + rand(nP,2)*1.2;   % adding some noise

%% ==================   Plotting   ==================
figure(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
```

Python offers a function in `skimage.measure.ransac`

L.16 Posture Estimation

Two files need to be downloaded, see the first two comment lines of the code block below:
- pose_deploy_linevec_faster_4_stages.prototxt: this is a text file that contains the architecture of the CNN. It is written in a format that is suitable for the CAFFE software.
- pose_iter_160000.caffemodel: this is a file containing the corresponding weight values, ca 200MB.

Those two files will be loaded into the program with the function cv.dnn.readNetFromCaffe from OpenCV: it generates the network called here NET.

The output of the estimate, OMPS, consists of ca. 40 confidence maps, of which the first 15 correspond to body parts, and the 16th to the background.

```
# https://github.com/CMU-Perceptual-Computing-Lab/openpose/blob/master/models/pose/mpi/pose_deploy_linevec_faster_4_stages.prot
# http://posefs1.perception.cs.cmu.edu/OpenPose/models/pose/mpi/pose_iter_160000.caffemodel
from numpy import zeros
import cv2 as cv

ifn     = "C:/IMGdown/POSEHUM/tennisPlayer.jpg"     # the image to be evaluated

# Specify the paths for the 2 files
pthFiles    = 'C:/IMGdown/POSEHUM/models/'
protoFile   = pthFiles + "pose/mpi/pose_deploy_linevec_faster_4_stages.prototxt"
weightsFile = pthFiles + "pose/mpi/pose_iter_160000.caffemodel"

#%% ====    construct network   =====
NET       = cv.dnn.readNetFromCaffe(protoFile, weightsFile)

#%% ====    load image and transform to blob image  =====
Ibgr      = cv.imread(ifn)
# generate blob image...
sclFact   = 1.0 / 255           # scale factor
szInp     = (368, 368)          # size of input
menSub    = (0,0,0)             # mean value to subtract
Iblob     = cv.dnn.blobFromImage(Ibgr, sclFact, szInp, menSub, swapRB=False, crop=False)

#%% =====    sent blob image through net     ======
NET.setInput(Iblob)             # enter image    [nImg nChr hi wi]
OMPS    = NET.forward()         # forward image: [nImg nMaps hMap wMap]

#%% =====    analyze maps     =======
nImg, nMaps, hMap, wMap = OMPS.shape
Iback   = OMPS[0, 15, :, :]     # background map [hMap wMap]
# --- Loop the 15 key points
aLoc=zeros((15,2)); Prob=zeros(15)
for i in range(15):
    Mconf       = OMPS[0, i, :, :]              # confidence map
    dmy, prob, dmy, Loc = cv.minMaxLoc(Mconf)   # global peak
    aLoc[i,:]   = Loc                           # coords [1..sizeMap]
    Prob[i]     = prob
print('Key probs range from %1.2f - %1.2f' % (Prob.min(), Prob.max()))

#%% ----    Plot 1   ------
Irgb    = cv.cvtColor(Ibgr, cv.COLOR_BGR2RGB) # convert to RGB for displaying
hFrm    = Ibgr.shape[0]
wFrm    = Ibgr.shape[1]

from matplotlib.pyplot import *
figure(1)
imshow(Irgb, cmap=cm.gray)
for i in range(15):
    Loc     = aLoc[i,:]                     # coords [1..sizeMap]
    x       = (wFrm * Loc[0]) / wMap
    y       = (hFrm * Loc[1]) / hMap
    plot(int(x), int(y), 'yo', markersize=12)
    text(int(x)+5, int(y), str(i), color='red', fontsize=12)
```

```
BodyPart    = [None]*16
BodyPart[0] = 'Head'
BodyPart[1] = 'Neck'
BodyPart[2] = 'Right Shoulder'
BodyPart[3] = 'Right Elbow'
BodyPart[4] = 'Right Wrist'
BodyPart[5] = 'Left Shoulder'
BodyPart[6] = 'Left Elbow'
BodyPart[7] = 'Left Wrist'
BodyPart[8] = 'Right Hip'
BodyPart[9] = 'Right Knee'
BodyPart[10] = 'Right Ankle'
BodyPart[11] = 'Left Hip'
BodyPart[12] = 'Left Knee'
BodyPart[13] = 'Left Ankle'
BodyPart[14] = 'Chest'
BodyPart[15] = 'Background'
```

For the face pose recognition system, one downloads the following two files - to build the network. Apart from that, the code pretty much remains the same as above.

```
# https://raw.githubusercontent.com/CMU-Perceptual-Computing-Lab/openpose/master/models/face/pose_deploy.prototxt
# http://posefs1.perception.cs.cmu.edu/OpenPose/models/face/pose_iter_116000.caffemodel
```

For the hand pose recognition system, one downloads the following two files:

```
# https://raw.githubusercontent.com/CMU-Perceptual-Computing-Lab/openpose/master/models/hand/pose_deploy.prototxt
# http://posefs1.perception.cs.cmu.edu/OpenPose/models/hand/pose_iter_102000.caffemodel
```

L.17 Text Recognition

L.17.1 OCR

```
# pip install pillow
# pip install pytesseract
# set an environment variable called TESSDATA_PREFIX' to:
#           'C:\\Program Files (x86)\\Tesseract-OCR\\tessdata\\
from PIL import Image
import pytesseract
pytesseract.pytesseract.tesseract_cmd='C:/Program Files (x86)/Tesseract-OCR/tesseract.exe'

#%% ----   Load Image   ----
pthImg  = 'someImageWithText.jpg'
Irgb    = Image.open(pthImg)

#%% ====    Detect  =====
Txt     = pytesseract.image_to_string(Irgb)
print(Txt)
```