University of Trento

Computer Vision

By
Alessia Pivotto
Lorenzo Orsingher

Handouts of the course 2023/2024

Hello world! This is a comprehensive recap of the Computer Vision course held by Prof. Conci at the University of Trento. The document covers pretty much all the topics discussed in class and it's a great source to prepare for the exam, the order of the topics is structured in a way that follows the course's progression. Although we put as much love and care as we could there might still be some mistakes or inaccuracies, so feel free to contact us. We hope you find our work useful, and we wish you the best of luck for the exam! ;)

@lorenzoorsingher
@AlessiaPivotto
GitHub repo

Contents

1 Images and Videos **1**

1.1 Definitions . . . 1

1.1.1 Computer Vision . . . 1

1.1.2 Acquisition . . . 1

1.1.3 Digital Images . . . 2

1.1.4 Sampling . . . 2

1.2 Color . . . 3

1.2.1 Additive color model . . . 4

1.2.2 Bayer Pattern . . . 6

1.2.3 Quantization . . . 6

1.2.4 Limits of the 2D . . . 6

1.2.5 Video . . . 6

1.2.6 Histogram . . . 7

1.2.7 Operations . . . 8

1.3 Edge extraction . . . 9

1.4 Filters . . . 11

1.4.1 Low-pass filtering . . . 11

1.4.2 Gaussian filtering . . . 12

1.4.3 Median filtering . . . 12

1.5 Morphology . . . 13

1.5.1 Dilation . . . 13

1.5.2 Erosion . . . 14

1.5.3 Closing and Opening . . . 15

2 Models **16**

2.1 The Pinhole Camera Model . . . 16

2.1.1 Features of the Pinhole Camera Model . . . 17

2.2 Projection . . . 18

2.2.1 The Perspective Projection Model 18
2.2.2 The Orthographic Projection Model 19
2.3 Illumination Models 20
2.3.1 Lambertian Reflectance 21
2.4 Remarks on cameras and lenses 22
2.4.1 Typical issues with lenses 22
2.4.2 Focus 23
2.4.3 Resolution, Blur and Resolving Power 24

3 Motion Detection **27**
3.1 Motion detection 29
3.1.1 Real vs Apparent movement 29
3.1.2 Occlusion 29
3.1.3 Aperture 30
3.1.4 Optical Flow 30
3.2 Motion detection in practice 32
3.2.1 Change Detection 32
3.2.2 Frame differencing 34
3.2.3 Background subtraction 35
3.2.4 Gaussian average 37
3.2.5 Mixture of Gaussians 38

4 Motion Tracking **40**
4.1 Object Tracking 40
4.2 2D Tracking 41
4.2.1 Region-based tracking 41
4.3 Blobs extraction 43
4.3.1 Target association 44
4.3.2 Splitting 44
4.4 Merging 45
4.4.1 Criteria for splitting and merging 45
4.5 Occlusion 46
4.6 Tracking: Feature-based 46
4.6.1 The Lucas-Kanade optical flow 48
4.6.2 Pyramidal implementation 49
4.6.3 Bayesian tracking 50
4.6.4 The Kalman Filter 54

4.6.5 Predict-and-correct stages . . . 56
4.6.6 Simple Kalman Filter example . . . 58
4.6.7 Extended Kalman Filter . . . 59
4.6.8 Particle Filters . . . 61
4.6.9 How PFs work . . . 62

5 Geometry **66**
5.1 Affine Transformations . . . 66
5.1.1 Scaling . . . 67
5.1.2 Rotation . . . 67
5.1.3 Translation . . . 68
5.1.4 Rotation, scaling and translation . . . 69
5.1.5 General Affine Transformations . . . 70
5.2 Going 3D . . . 71
5.2.1 Intrinsic and Extrinsic Parameters . . . 72
5.2.2 3D Affine Transformations . . . 73
5.2.3 3D Translation . . . 74
5.2.4 3D Scaling . . . 74
5.2.5 3D Rotation . . . 74
5.2.6 3D General Configuration . . . 75
5.3 Calibration . . . 77
5.3.1 Calibration Procedure . . . 78
5.3.2 Computing the 3D position of a point . . . 79
5.4 The Binocular Stereo . . . 80
5.4.1 Computing correspondences . . . 81
5.4.2 Stereo Vision and Epipolar Geometry . . . 81
5.4.3 Estimation of the 3D position . . . 84
5.4.4 Matching points . . . 86
5.4.5 Image Normalization . . . 87
5.4.6 General Stereo Configuration . . . 88
5.4.7 The Fundamental Matrix . . . 89
5.5 Homography and friends . . . 92
5.5.1 2D Homography . . . 93
5.5.2 Multiple view geometry . . . 94

6 Local Feature Extraction **96**
6.1 HOG . . . 96

6.1.1 The problem of scale 98
6.1.2 Feature compression 98
6.2 SIFT 99

7 Classification **103**
7.1 The classification process 104
7.1.1 Subject to failure 105
7.2 Regression 105
7.2.1 The ROC curve, Precision and Recall 107
7.3 The face detection problem 108
7.3.1 Recursion 110
7.3.2 AdaBoost 110

Chapter 1

Images and Videos

1.1 Definitions

1.1.1 Computer Vision

Is the science and technology of machines that see, where "see" means that the machine is able to extract information from an image that is necessary to solve some task. The image data can take many forms, such as video sequences, views from multiple cameras, or multidimensional data from a medical scanner.

Some examples of applications of computer vision include systems for controlling processes, like an industrial robot or an autonomous vehicle, detecting events as for visual surveillance or people counting, or again organizing information for example for indexing databases of images and image sequences, even modeling objects or environments like in industrial inspection, medical image analysis or topographical modeling and interaction as the input to a device for computer-human interaction.

1.1.2 Acquisition

It refers to the process of transferring a portion of the real 3D world onto a 2D surface bringing a continuous-parameter real world into a discrete-parameter one. The acquisition process is the first step in the processing chain and it is the transformation of a physical signal into an electrical one, by means of a sensor. Remember that the sensor is a device that responds to a physical stimulus (light, heat, pressure, etc.) and produces an electrical signal. *PS: The representation is in a standard format.*

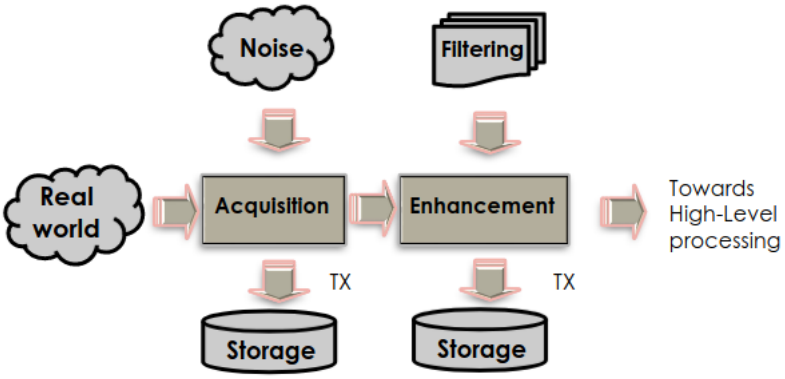


Figure 1.1: The processing chain(The first part is more about signal and video, in this course we'll focus mainly on the second half)

1.1.3 Digital Images

A digital image is a representation of a two-dimensional image as a finite set of digital values, called picture elements or pixels. The digital image contains a fixed number of rows and columns of pixels, it's a collection of coordinates. Pixels are the smallest individual element in an image, a single pixel represents a projection of a portion of the real world, holding quantized values that represent the brightness of a given color at any specific point. Pixels can be **Grayscale** or **Color**. Grayscale images are represented by a single component, typically 8bit, while color images are represented by three components, typically 24bits.

1.1.4 Sampling

Is the process of converting a continuous signal into a discrete signal. This is because the "real world" is a continuous function, while computers are digital. Analog video is a 1-D continuous function where one spatial dimension is mapped onto time by means of a scanning process, while digital video is instead sampled in a three dimensions (2D spatial and 1D temporal).

In a Nutshell:

- Continuous signal

$$s_c(x_1, x_2)$$

Figure 1.2: Sampling

- Spatial rectangular sampling

$$x_1 = n_1 \triangle x_1, x_2 = n_2 \triangle x_2$$

- So

$$s(n_1, n_2) = s_c(n_1 \triangle x_1, n_2 \triangle x_2)$$

- Once we have the digital format, we can manipulate data and apply filters, change colors, store and transmit.

For what concerns handling images we need only to know that pixels are numbered starting from the top left corner (so the top left corner is the origin, arrow down for the rows and right for the columns). The value of a pixel in a certain position is defined as $I(r, c)$, where r is the row index and c the column one. (Start at 0 or 1, depends on you :D) Monochrome images have values normalized in the range $[0, 1]$, where 0 is black, 1 is white and the intensity is called *grey level*. While color pictures have 3 channels (RGB) and the values are normalized in the same way.

1.2 Color

But what is color? It's the attribute the human visual system associates to objects, more scientifically, it's a mathematical relationship that combines different wavelengths. It's important to check whether something we see is what we expect, to recognize objects or to distinguish similar things. For example, a white car, that for us is obviously white, for a computer is a combination of red, green and blue, moreover it has some black points for the wires, some point gray due to the street, etc.
For this reason we need to talk about color perception. The human eye is like a camera with a focal length of about 20mm, where the iris controls the amount of light by adjusting the size of the pupil. The perception of color is possible through cones in the fovea that has around 100M receptors. Cones have peak responses on three main wavelengths, red (700nm), green (546.1nm) and blue (435.8nm).

But going back to the main topic, data can be processed locally but even transmitted remotely or archived on a storage unit. The problem is that images and videos require a lot of bandwidth, so we need to compress them with a codec. A codec is a device or computer program for encoding or decoding a digital data stream or signal in the compressed domain. It stands for coder-decoder, and it's a way to reduce the dimension of the file (e.g JPEG, MPEG, DIVX).
It is important to mention that compression is lossy, so reduces quality (we lose some information) and can even introduce visual artifacts such as blocking, blurring, chromatic aberrations(plus some noise from the sensor). However, the loss is not a problem because the human visual system is not perfect.
NB1: There exists even lossless compression, but it's not used much.
NB2: Processing is typically done in the uncompressed domain.
To make a final comparison, raw images are usually stored in a 1D vector of pixels, while compressed images reduce the dimension of the file with or without loss of information.

1.2.1 Additive color model

The additive color model is a method to create color by mixing the primary colors.

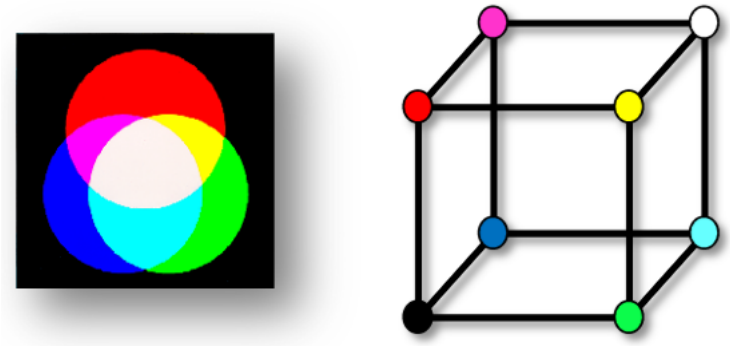


Figure 1.3: Additive Color Model

Colored beams are projected onto a black surface, then overlap so the human eye receives the stimula without generating interference, mixing the components and perceiving the resulting color. Starting from the primary colors RGB we can obtain:

- R+G = Yellow
- B+G = Cyan
- R+B = Magenta
- R+G+B = White

NB: Subtractive color is the inverse process.
Looking at the image above we can identify the gray scale along the diagonal connecting

the black corner with the white corner.
About the way colors combined, we can have:

- White: RGB(1,1,1)
- Black: RGB(0,0,0)
- Gray: RGB(0.5,0.5,0.5)
- Green: RGB(0,1,0)
- Yellow: RGB(1,1,0)

When we take a colored image and separate it into its three RGB components, we notice that these components are correlated. This means that the grayscale versions of the individual RGB channels contain almost the same amount of information. Among these, the green component has a stronger response compared to the red and blue components. Additionally, the human eye is more sensitive to variations in luminance and contrast than to color differences. Therefore, using a different representation might be more effective for certain applications. An example could be the YCbCr color space, where Y, the luminance is separated from Cb and Cr that are the chrominance. Or again the HSV, where H is the hue (color), S is the saturation (brightness) and V is the value (intensity).
PS: YCbCr is a generalization of YUV, just a matter of conversion matrices (Downsampling).

Figure 1.4: HSV Color Space

Summing up just a little bit:

- RGB is used in general for visualization, in displays each pixel in composed by three phosphors (CRT) or LEDs (LCD);
- YUV is suitable for compression since we are less sensitive to chrominance variations and U and V can be downsampled;

- HSV is robust for computer graphics and image analysis.

1.2.2 Bayer Pattern

In the acquisition phase, light is captured by the CCD (Charge Coupled Device) that is an array of cells. Each cell is a photosensitive element that converts light into an electrical signal. The Bayer pattern is a color filter array for arranging RGB color filters on a square grid of photosensors. The best solution would be to have devices with 3 different CCDs and to correctly exploit the human eye response there would be three types of photosensors of color filters 50% green, 25% red and 25% blue.
NB: Green sensors are defined as luminance-sensitive elements, while the red and blue ones are defined as chrominance-sensitive.

1.2.3 Quantization

Like in the mono-dimensional case, signals need to be quantized. That implies the definition of a number of levels to define our signal. Typically, the range 0-1 is quantized using 8bpp but even other representations with 10-12 bpp are available. But why 8bpp? Well, 8bpp represent 256 levels, which is fine for the human eye, that can distinguish 100 levels of grey. Indeed, if we quantize with less that 6bpp (64 levels, minimum to ensure smooth pictures), then false contours will appear $\Rightarrow$ contouring.

1.2.4 Limits of the 2D

Images provide reliable information about static scenes. We lose motion information like temporal evolution of the scene, rapid changes, dynamics of motion (qualitative and quantitative) and even how subjects/objects relate one to each other. Moreover, analyzing a video provides a more consistent representation of the scene. It's closer to what humans do every day.

1.2.5 Video

A video is a sequence of 2D images that represent a projection of moving 3D scene onto the video camera image plane. It's expected that adjacent frames are strongly correlated. For what concerns the resolution, the images are up to 50mp, while in video is typically lower, 4k is $3840 \times 2160 = 8.3$mp, 8k is $7680 \times 4320 = 33.2$mp and full HD is around 2mp. The reasons are that video can last hours, so we need to store a lot of data, and that the human eye is less sensitive to resolution in motion. Just to move on

a little bit, when analyzing an image, key features of interest include color distribution and the presence of edges and contours, which help identify and define objects.

In the context of video analysis, additional benefits emerge. Consistency of features over time allows for tracking object movement, while changes in the scene can reveal dynamic interactions. Videos also enable the detection of objects entering or exiting the frame.

Several factors contribute to the complexity of scene analysis. These include the presence of multiple objects, occlusions, shadows, noise, motion, illumination changes, clutter, and non-rigid objects. Each of these elements introduces challenges that require sophisticated techniques to accurately interpret the scene.

1.2.6 Histogram

The histogram is a simple way to describe the color distribution of a picture, indeed, it can be seen as a probability density function and it represents the occurrence of all colors on a graph. In other words it's a statistical representation of the pixel values. For an $M \times N$ image

$$hist(p) = \frac{I(x,y)}{M \times N}$$

Where $I(x,y)$ is the intensity of the pixel at position (x,y) and $M \times N$ is the total number of pixels in the image.
The equation holds for one component, so in case of more components, a histogram can be obtained from each of them.

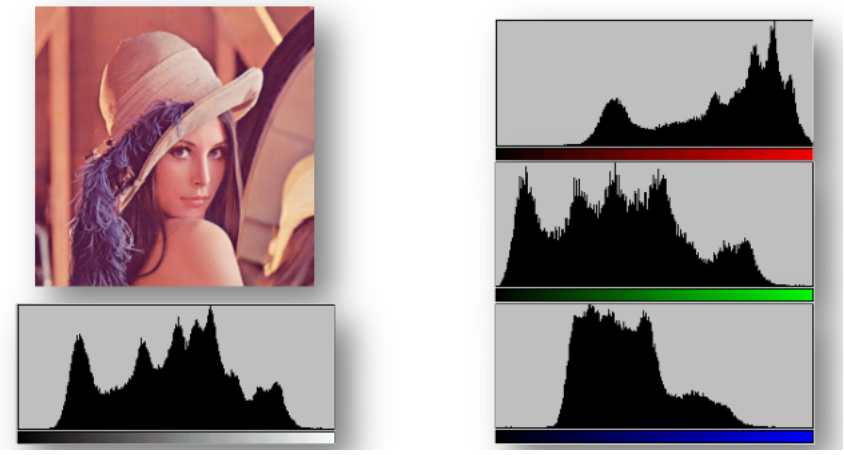


Figure 1.5: Histograms of Lana

A histogram provides valuable insights into various aspects of an image. It can reveal

whether the image is dark or bright by showing the distribution of pixel intensities. Additionally, it indicates whether colors are distributed equally or if there are dominant or missing colors.
Applications of histogram analysis are diverse. In environmental monitoring, histograms help assess whether the illumination of a scene is appropriate. By comparing histograms over time, one can detect changes in the background, such as in the past two hours. Moreover, histograms can assist in object identification by determining whether the observed moving object matches the characteristics of object A or B.
Overall, a histogram serves as a "signature" that can be applied across various domains for image analysis and interpretation.

1.2.7 Operations

Stretching:
Is a simple operation to change the dynamic range of the image by stretching the histogram and increasing the contrast by applying a piecewise linear function.

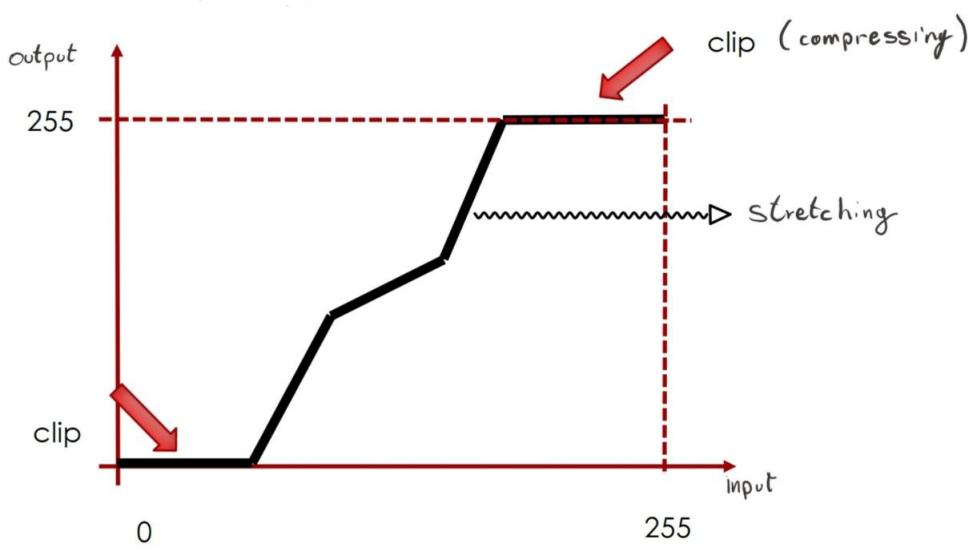


Figure 1.6: Stretching

Equalization:
Working on the statistics of the pixels it is possible to improve the quality of an image. Ideally we'd like to obtain a FLAT histogram, and to do so we have to compute the cumulative histogram (equivalent to CDF). Basically, we compute the sum of all bins, then we subtract the minimum value and normalize the result by dividing for the number of pixels minus 1. Notice that this operation can introduce artifact, indeed, looking at

the image, we can notice that some bins are empty. Concerning the visual final aspect, the image is more contrasted (similarly to stretching), but the noise is enhanced.

$$CHist_I(p) = \sum_{k=0}^{p} hist(k)$$

$$hist_eq(p) = \frac{CHist(p) - CHist_{min}}{M \times N - 1} \times 255$$

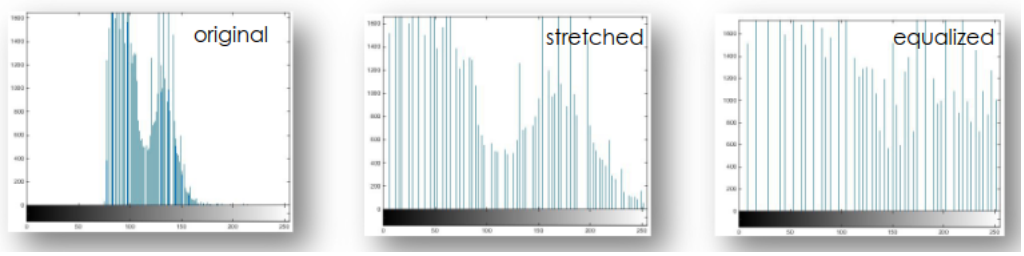


Figure 1.7: Different operations on the histogram

1.3 Edge extraction

We perceive objects primarily through their color (appearance model) and shape. Shape, defined by the boundary between an object and the rest of the image, is crucial for object recognition. A sharp change in image brightness often indicates an edge, highlighting these boundaries.
However, environmental conditions like cast shadows or surface orientation can affect these perceptions, causing changes in appearance. Additionally, edges might not always be meaningful, especially in textured areas or when noise is present. These factors can complicate the task of recognizing objects based on their shape and edges alone.
Steps:

- Determine intensity, and possibly the direction of an edge for each pixel location through gradient or Laplacian;
- Find a threshold and binarize.

NB: The first step is the most difficult one. Different tools are available, but we'll focus here on the gradient-based algorithms.

Edge extraction by gradient analysis

As for mono-dimensional signals, the goal is to find maxima and minima but here, differently from the 1D case, we also have a direction. This means finding a gradient along a line r oriented in the direction θ.

$$\frac{\partial f}{\partial r} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial r} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial r} = f_x \cos(\theta) + f_y \sin(\theta)$$

The edge extraction process consists of computing the 1st order derivative in two orthogonal directions, f_1 and f_2. To each of them we associate an amplitude. Going a little more into the concrete, for edge detection typically we use FIR filters, that are linear and shift-invariant. The most common are the Prewitt, Roberts and Sobel operators.

For the first two we have to choose a convolution mask and apply it to the picture. The convolution is computed for both masks and a threshold is chosen to highlight only the strongest edges.

Roberts operator: $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ **Prewitt operator:** $\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$

NB: The Prewitt operator is preferred because it is isotropic ⇒ you know where is the center.

For the Sobel operator, we have to apply two masks, one for each orthogonal direction. The mask is a 3x3 matrix and the gradient is computed for each point. Then we have to merge the results by doing the square root of the sum of the squares of the two gradients, after that we apply a threshold; this because Sobel tends to introduce a lot of noise, so we need to smooth the result.

Sobel operator: $D_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad D_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

The convolution with the FIR masks is performed similarly to the 1D convolution. Take the mask, rotate, slide from left to right and associate to the central point the value of the convolution.

What you're going to see is the convolution expressed in 2D, in which we have to deal with two components, x and y.

NB: We know that a convolution in space has a counterpart in frequency in forms of multiplication.

In space:

$$g(x, y) = f(x, y) * h(x, y)$$

In frequency:

$$G(u, v) = F(u, v) \cdot H(u, v)$$

$$y(m, n) = x(m, n) * h(m, n) = \sum_{m'=-\infty}^{\infty} \sum_{n'=-\infty}^{\infty} h(m - m', n - n') \cdot x(m', n')$$

From a practical point of view, we rotate the mask by 180° and then we keep shifting it on the image. The result of the multiplication of coefficients and pixel values will be stored in the output image in the application point (the central one).

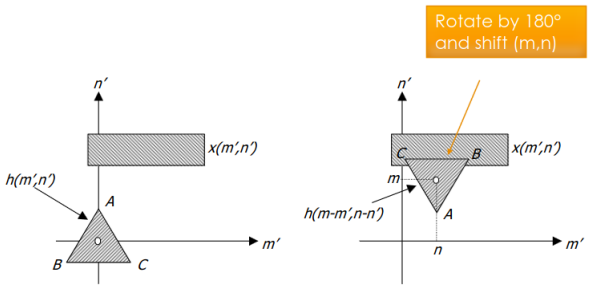


Figure 1.8: How to (convolution edition)

1.4 Filters

Now we are going to see some filters that are using as smoothing or enhancing operators.

1.4.1 Low-pass filtering

Low-pass is the easiest filter and it basically consists of averaging the values in the sliding window. The window is a matrix of size $m \times n$ and the result is obviously the average of the values in the window.

$$I_{LP}(x, y) = \frac{1}{mn} \sum_{x=-a}^{a} \sum_{y=-b}^{b} I(x, y)$$

It preserves low-frequency components and removes the high-frequency ones. It's a sorta of weighted sum of low high frequency functions. So, it cuts the high frequency

components above the threshold and smooths the image.
Average filtering ⇒ average of the pixels in the window ⇒ bigger the filter, more the smoothing (the difference is smaller). It also helps denoising the signal.

1.4.2 Gaussian filtering

Gaussian filtering is a low-pass filter used to remove high-frequency components from an image, effectively blurring it and reducing noise. It operates as a 2D convolution filter that employs a Gaussian function, which is isotropic and does not require flipping the mask. The mask, centered in the middle, is sized to ensure integer values. Using a Gaussian mask is advantageous because, in the Fourier domain, it remains a Gaussian. This symmetry and lack of rotation simplify the convolution process.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

NB: The bigger, the better. The central pixels are more considered, far pixels have a lower impact.

1.4.3 Median filtering

Median filtering is a nonlinear technique used to remove noise from an image. This spatial domain method replaces each pixel value with the median value of its surrounding neighborhood pixels. It is especially effective for removing salt and pepper noise and is ideal for images corrupted by impulse noise.
While smoothing techniques work well for zero-mean noise, they are less effective for handling spikes, as low-pass (LP) filtering tends to blur the noise, spreading it and creating artifacts. Median filtering avoids this issue by not relying on convolution. Instead, it sorts the pixel values in the filter's neighborhood and assigns the median value to the

central pixel. This approach prevents the noise from spreading, although it may leave some noise if the filter size is too small. On the other hand, bigger size can introduce artifacts.

1.5 Morphology

Morphology refers to the study of the shape of regions in an image, focusing on operations that assess and manipulate these shapes. The goal is to determine if one shape fits into another, detect holes of specific sizes, or remove areas smaller than a threshold...
It involves non-linear filtering using a structuring element, which is a predefined arbitrary shape. This element slides over the image, comparing its pattern with the image at each position.
There are four main operations: Erosion, Dilation, Opening and Closing.
Erosion and Dilation are self-explanatory, the first one reduces the area of a shape, the second one enlarges it. Instead, opening and closing are a combination of erosion and dilation. Opening gets rid of small portions of the image close to the boundaries of relevant areas, while closing fills holes and makes region boundaries smoother.
Concerning structuring elements, depending on the type of shape we want to edit, the right element must be chosen.

1.5.1 Dilation

Figure 1.9: Dilation

Dilation is a morphological operation that is used to enlarge the boundaries of regions of foreground pixels in an image. It is used to merge the objects in the image. The structuring element is a binary mask that defines the neighborhood of the pixel. The dilation of the image is obtained by sliding the structuring element over the image and placing the origin of the structuring element at the center of the pixel.
More formally, the dilation of an image B by a structuring element S is given by:

$$B \oplus S = \cup_{b \in B} S_b$$

Sweep the structuring element on the whole image, as the origin of the structuring element touches a "1" of the image all pixels of the structuring element are OR'ed to the output image.
PS: OR/union ⇒ every time that the structuring element touches a 1, the output is 1, so it enlarges the shape.

1.5.2 Erosion

Figure 1.10: Erosion

Erosion is a morphological operation that is used to reduce the boundaries of regions of foreground pixels in an image. It is used to separate the objects in an image. Sweep the structuring element on the whole image, at each position where every 1-pixel of the structuring element covers a 1-pixel of the binary image, the binary image corresponding to the origin is OR'ed with the output image (the pixel is set to 1).
More formally, the erosion of an image B by a structuring element S is given by:

$$B \ominus S = \{b | b + S \in B \quad \forall s \in S\}$$

Erosion of A by B can be understood as the locus of points reached by the center of B when B moves inside A.
NB: If before it was enough for the center to touch a 1, now ALL must touch a 1. All the structuring element must be contained in the image.

1.5.3 Closing and Opening

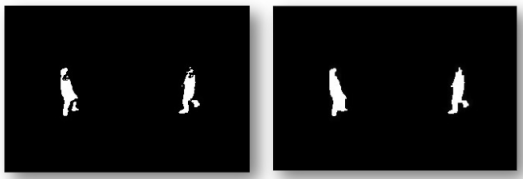


Figure 1.11: Closing

- Closing
  ⇒ first dilate and then erode;
  Non-contiguous regions are first merged, then the boundaries are refined.

- Opening
  ⇒ first erode and then dilate;
  First small areas are removed, then residual information is refined.

Figure 1.12: Opening

These operations are in general non-reversible, so the result depends on the order of the operations.

Chapter 2

Models

Let's begin the chapter with a question. Why do we need models? Well, they represent a good approximation of the real world and allow describing events through a parametric representation (parameters can be extracted and used for processing). In the context of computer vision, models are used to describe the geometry of the world, the appearance of objects, and the imaging process. In this chapter, we will discuss the most common models used in computer vision. We will start with the pinhole camera model, then we will see the perspective projection model and finally, we will discuss the appearance model.

2.1 The Pinhole Camera Model

The pinhole camera model is the simplest model used to describe the imaging process. It is based on the principle that light travels in straight lines. The model is composed of a pinhole, a plane, and an image plane. The pinhole is a small hole in the plane, and the image plane is placed behind the pinhole. The light rays coming from the scene pass through the pinhole and project the scene onto the image plane.

2.1.1 Features of the Pinhole Camera Model

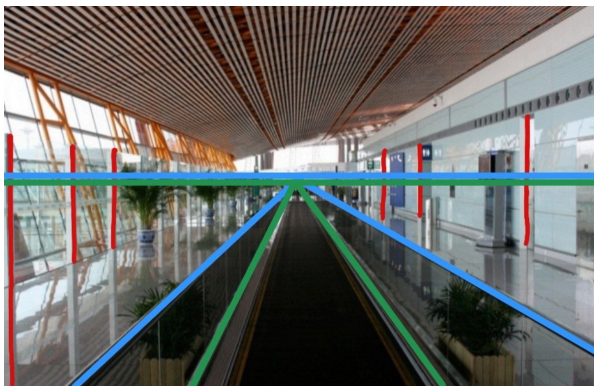


Figure 2.1: The behavior of parallel lines in the world and in the image.

- Parallel lines in the world converge to a single point in the image;
- Parallel lines on the same plane lead to *collinear vanishing* points. In Figure 2.1 we can see how the blue and green lines, that lie parallel on two different planes, eventually converge to the same point in the image plane;
- The line where coplanar parallel lines end up is called the *horizon* for that plane;
- Vertical lines (red) are perpendicular to the horizon.

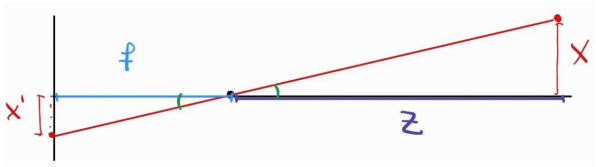


Figure 2.2: Similar triangles in the projection.

The intuition behind the projection of the pinhole camera model is based on the concept of similar triangles. In Figure 2.2 we can see how the triangles formed by the object and the image plane are similar, f is the *focal length* of the camera, Z is the distance of the object to the pinhole and X and X' are respectively the position of the object and its projection on the image plane. This means that the ratio of the sides of the triangles is the same, this concept is used to derive the projection equations.

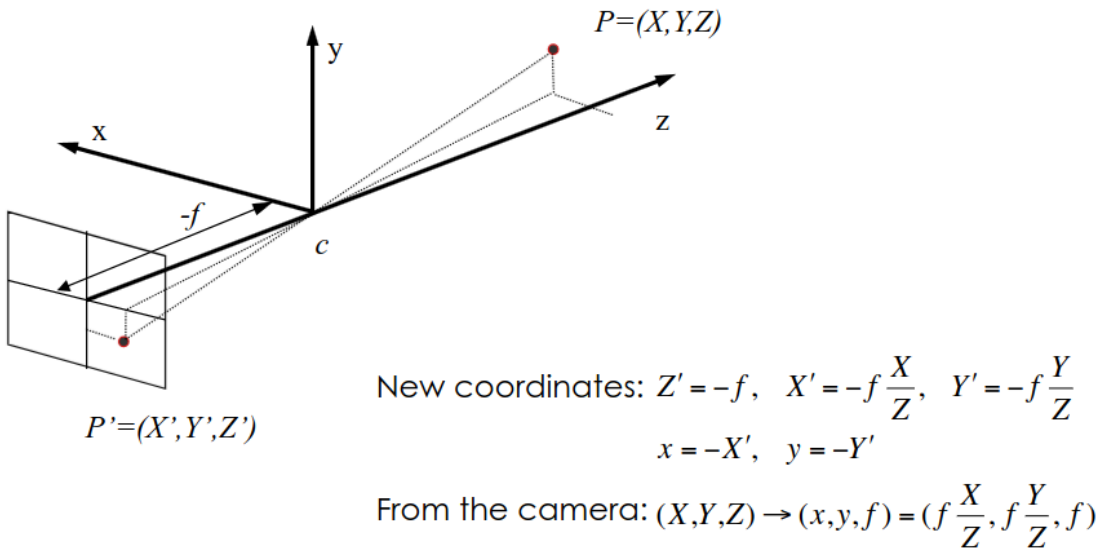


Figure 2.3: If the object is far, it appears small in the image. If the object is close, it appears large in the image.

2.2 Projection

Projection is the fundamental process of mapping 3D points, including time, to 2D points. When projecting a 3D point onto the image plane, a line is defined between the point and the center of projection. The intersection of this line with the image plane yields the 2D projection of the 3D point. This process forms the basis of how 3D scenes are represented in 2D images.

$$f : \mathbb{R}^4 \rightarrow \mathbb{R}^3$$

$$f(X, Y, Z, t) = (x, y, t)$$

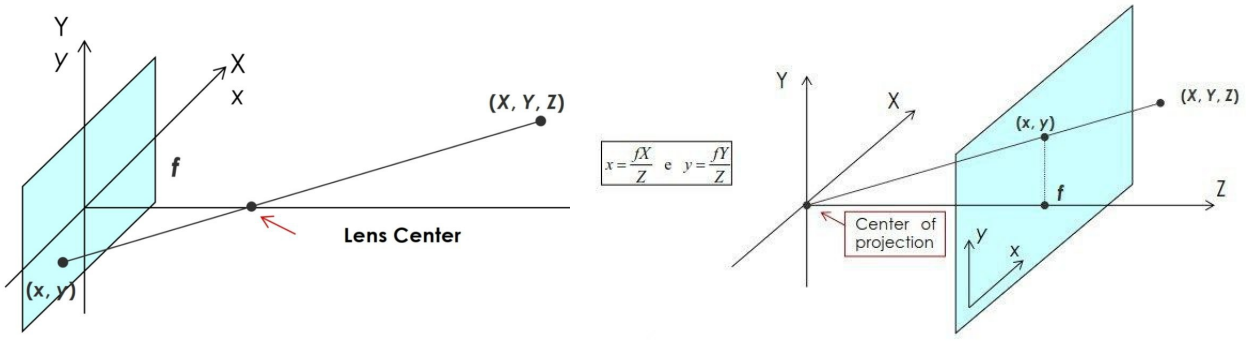
NB: These are continuous variables.
There exist two types of projection:

- Perspective projection;
- Orthographic projection.

2.2.1 The Perspective Projection Model

The perspective projection model is an extension of the pinhole camera model. It is used to describe the imaging process in a more realistic way. The model is based on the principle that light travels in straight lines and that the image is formed by the

intersection of these lines with the image plane. For simplicity, we usually consider the image plane on the same side of the "real world", to avoid the picture flip.



(a) The pinhole camera model.

(b) The perspective projection model.

Figure 2.4: We can see how in 2.4b we moved the camera plane on the right side and the center of projection at the origin, for this reason a $-f$ is added to the denominator.

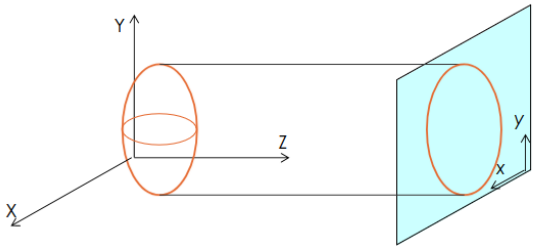
$$\begin{cases} \frac{x}{f} = \frac{X}{Z-f} \\ \frac{y}{f} = \frac{Y}{Z-f} \end{cases} \Rightarrow \begin{cases} x = \frac{fX}{Z-f} \\ y = \frac{fY}{Z-f} \end{cases} \Rightarrow \begin{cases} x = \frac{fX}{Z} \\ y = \frac{fY}{Z} \end{cases}$$

Now the center of projection corresponds to the origin of the 3D space, and the plane (X, Y) is parallel to (x, y). From a computational point of view nothing changes, we simply shift the image plane from back to front in order to have the image straight. Due to the fact that we assume that f is negletable, compared to Z (Z>>f) we can safely approximate the denominator and remove the $-f$, obtaining the same equations as the pinhole camera model.

2.2.2 The Orthographic Projection Model

The orthographic projection model is a simplified version of the perspective projection model. It is assumed that all rays originated from the 3D object, and from the scene in general, are parallel among each other. In the drawing, the image plane is parallel to (X, Y).
Assuming this, the orthographic projection can be simply described in Cartesian coordinates as:

$$\begin{cases} x = X \\ y = Y \end{cases}$$



Or in form of a matrix:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

NB: The distance of the object from the camera does not affect the intensity of the image projected onto the 2D plane. It is a good approximation when the distance of the object is much bigger than the depth of the object itself.

2.3 Illumination Models

Illumination plays a crucial role in how we perceive objects, as it defines their appearance. When a light source interacts with an object, light can be absorbed, reflected, or transmitted. Modeling illumination is complex because we perceive objects based on how they reflect light across specific wavelengths. Remember that reflection can be either specular or diffuse.

Specular reflection concentrates more energy in the direction of the light source, creating highlights. Conversely, diffuse reflection distributes energy uniformly in all directions, making the position of the observer irrelevant.

Surfaces vary in specularity: matte surfaces reflect light uniformly, while glossy ones reflect light directionally. The specularity also depends on factors such as surface distance and the angle of the light source.

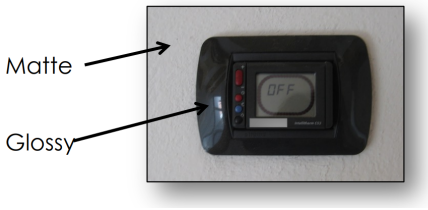
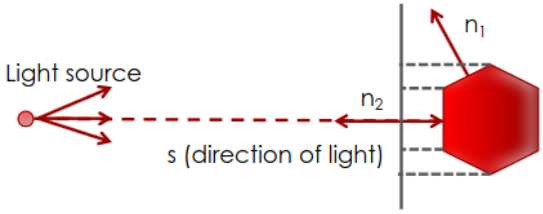


Figure 2.5: Reflections.

Modeling surface irradiation by the light source involves determining how the light interacts with the surface. Assuming a distant light source, we can represent all rays

with a single unit vector. Each surface element receives light based on the cosine of the angle between the surface normal and the light direction, affecting its perceived brightness and color.

$$i = n \cdot s$$



NB: intensity we perceive is equal to dot product of the normal of the surface and the light direction. Dot product ⇒ cosine ⇒ the more close to 90 degrees, the less we see.

2.3.1 Lambertian Reflectance

Lambertian Reflectance is a model used to describe diffuse reflection. It operates under the assumption that the surface is sufficiently rough compared to the wavelength of light. In this scenario, each surface element reflects light uniformly in all directions, regardless of the viewing angle. This means that the luminance of the surface remains constant, irrespective of how it is observed. This model neglects any specular component of reflection, focusing solely on the diffuse characteristics of the surface.
NB: Assumes that the surface we're dealing with is ideal, so looking at it from any angle we'll see the same intensity.

$$I = \rho n \cdot s = \rho |n||s| \cos(\alpha)$$

Where:

- I is the intensity of the light;
- ρ is the albedo, the ratio of the reflected illumination to the total illumination, intrinsic property of the surface (*LOL not true, some surfaces may reflect light differently depending on the view angle*);
- n is the normal to the surface;
- s is the direction of the light.

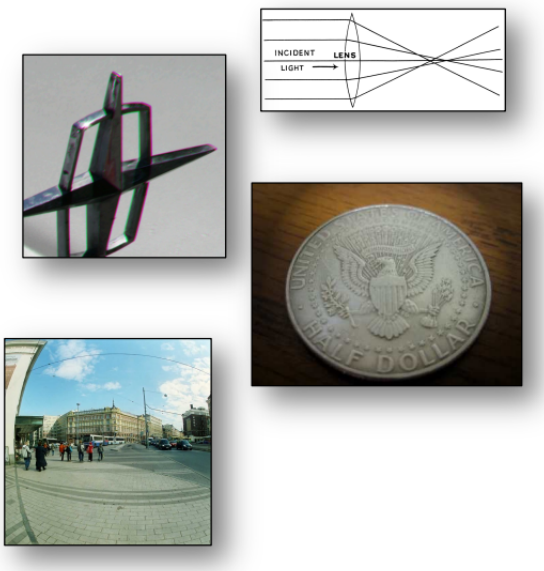
An element is not visible if the angle between the normal and the light direction is greater than 90 degrees. Generally, the pixel in image $I(r, c)$ depends on the light source direction and the normal of the element direction.

2.4 Remarks on cameras and lenses

Cameras use lenses rather than pinholes, that do not exist in real life. This introduces the concept of **focus**. According to the thin lens equation, an object is in focus if the distance from the center of the camera to the image plane satisfies certain conditions. If this is not the case, it leads to **aberrations**, which are deviations from the ideal imaging process.
While we usually assume that the object is in focus, it's important to understand the implications of this assumption. By adhering to the principles of the thin lens equation, we ensure that the camera captures images with minimal aberrations, resulting in clear and accurate representations of the scene.

2.4.1 Typical issues with lenses

Spherical aberration:
The lens does not focus light rays that strike the lens far from the center. This means that the image is not sharp (causes blur). In simple terms the lens does not focus all the rays in the same point, so the light is reflected wrongly because the lens is not properly manufactured.



Chromatic aberration:
The lens does not focus all the colors in the same point, it's not able anymore to convey correctly the ray at different locations (for example cathode ray tube televisions).

Vignetting:
The lens does not focus all the rays in the same point, so the image is darker in the corners. The spreading of light is not uniform due to the impurity of glass.

Barrel distortion:
The focal length is too short or the lens is too wide, so the image is distorted and the

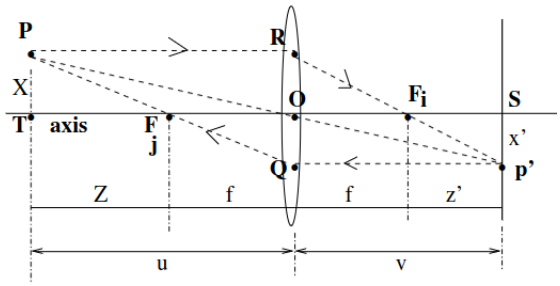
lines are not straight anymore. In simple terms, the lens is able to grasp a wide area of view. To compensate we can use a software to correct the distortion. *NB: Crooked lines can give problems in path analysis... distortion problem ⇒ that's why modern cellphones have more lenses.*

2.4.2 Focus

In contrast to the abstract pinhole model, real-life cameras employ lenses. These lenses have the remarkable ability to converge light rays to a single point, allowing for focused images. The relationship between the distance from the center of the camera to the image plane adheres to the thin lens equation.

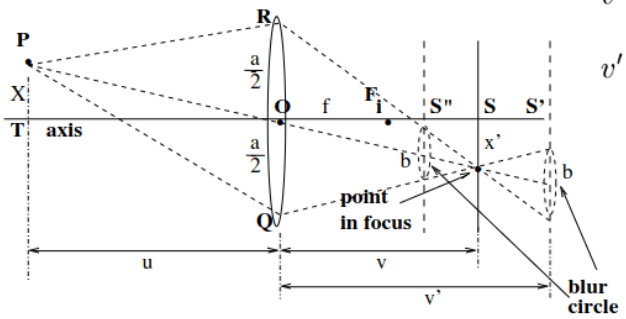
$$\frac{1}{f} = \frac{1}{u} + \frac{1}{v}$$

This equation serves as a fundamental principle in optics, ensuring that the lens accurately focuses incoming light onto the image sensor or film. It's this precise focusing mechanism that enables cameras to capture sharp and detailed images of the world around us.



If we move the image plane, point p' is out of focus ⇒ v changes ⇒ v'. And, if P is moved u changes ⇒ u'. The result is that the image is blurred on the image plane (instead of a point we see a circle, the rays of light don't converge).
NB: If the object is too far you can try to set the focal length, but it won't work.
We want to configure the camera in order to have the quantity b as small as possible. Notice that:

- In general $u > f = u_n < u$;
- If f becomes smaller, u_n is closer to the camera;
- If f becomes smaller, u_r is farther to the camera;
- $u_r > u$;
- If $u- > \infty$ rays are parallel and converge to the camera center;
- The difference between the far and near planes limiting b is called **depth of field**.



$$v' = \frac{a+b}{a}v \quad : in \ \ case \ \ v' > v$$

$$v' = \frac{a-b}{a}v \quad : in \ \ case \ \ v' < v$$

$$u_n = \frac{u(a+b)}{a+\frac{bu}{f}}$$

$$u_r = \frac{u(a-b)}{a-\frac{bu}{f}}$$

Autofocus

Is the capability of focusing a specific portion of the image, it can be active, passive or a combination of both.

Active:

The camera emits a signal and measures the time it takes for the signal to return, using this time to calculate the distance to the object. This method is commonly used in point-and-shoot cameras. However, it faces challenges with obstacles, glossy surfaces, and bright reflections, which can interfere with accurate distance measurement.

Passive:

Professional cameras use image contrast to achieve focus, primarily employing the phase detection method. This technique, commonly found in expensive SLR (single-lens reflex) cameras, involves analyzing a strip of pixels to assess contrast levels. If pixel values are too similar, the object is out of focus; if the contrast is high, the object is in focus.

This method faces challenges with flat surfaces, low contrast, and low light conditions. High-quality cameras address this by computing focus metrics along both vertical and horizontal axes to ensure accurate focusing.

2.4.3 Resolution, Blur and Resolving Power

We might have blurring due to the focus problem described in the previous section, but also due to the quality of the sensor. A CCD sensor of size $N \times M$ can detect up

to $N/2$ horizontal lines, meaning that in order to differentiate two lines there must be some spacing between them, one pixel at minimum, so given N rows of pixels only half of them can be filled with lines.

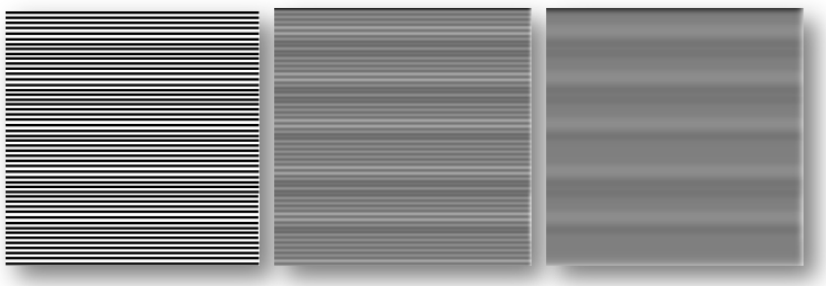
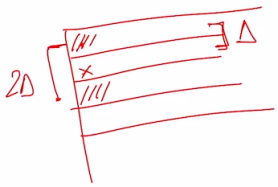


Figure 2.6: The right images are blurred because the sensor is not able to distinguish the lines.

The **resolving power** of a camera is the ability to distinguish between two points, it is defined as:



$$R_p = \frac{1}{2\Delta} \quad [lines/mm]$$

Where Δ is the pixel spacing: the distance between two pixels/lines in the sensor. The reason why we divide by 2 is that we need one separating row of pixels between two lines.

Example

Camera $10Mpx \rightarrow 4000cols \times 2500rows$

Full frame sensor $\rightarrow 36mm \times 24mm$

$$R_p = \frac{1}{2\Delta} \quad \Delta = \frac{24}{2500} \approx 0.01mm$$

$$R_p = \frac{1}{2 * 0.01} = 50 \, \frac{lines}{mm}$$

This means our sensor can distinguish 50 lines per millimeter, now we want to place our sensor in a camera in the real world and understand how well we can perceive details. Our setup is the following:

Focal length $f = 50mm$

Distance from the object $L = 50m = 5000mm$

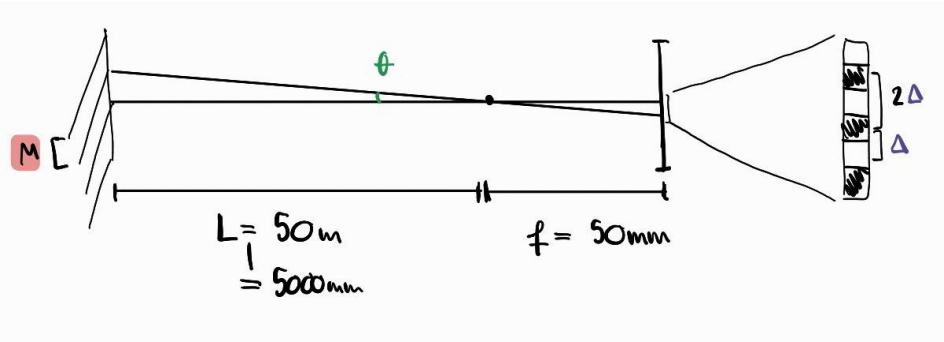


Figure 2.7: On the left we have a wall with some stripes, on the right we have our camera and sensor.

We can see we have an angle θ separating the rays coming from two adjacent lines. In order to distinguish two lines, the spacing between pixel lines must be at least 2Δ, we can write:

$$2\Delta = f \sin(\theta)$$

Since the angle is very small, the sine can be approximated to the angle itself. We can now calculate:

$$2\Delta \approx f\theta = 50\theta$$

We can now determine the angle θ:

$$\theta = \frac{\Delta}{25} \approx 4 * 10^{-4} rad$$

Now that we have the angle and we know the distance from the object to the camera, we can calculate what is M, the distance between two lines on the wall:

$$M \approx L\theta = 5000 * 4 * 10^{-4} = 2mm$$

With this expression we are saying that, given this setup, I can distinguish two lines which are spread vertically by a quantity no smaller than $2mm$.

Chapter 3

Motion Detection

Motion, typically described by a transformation, is a crucial feature in video analysis and processing. Motion detection involves identifying changes in an object's position relative to its surroundings or vice versa. Accurate motion description requires understanding both object and camera movement. By analyzing displacement vectors, we can infer whether the observed changes are due to object movement or camera movement. However, converting the 3D world into a 2D image results in some loss of information, making it challenging to distinguish between object motion and camera motion definitively. We can only observe that something in the image is changing.

$$3D : D(X; t_1; t_2) = X' - X = [Dx, Dy, Dz]$$
$$2D : d(x; t_1; t_2) = x' - x = [dx, dy]$$

NB: x is the projection of X on the image plane.

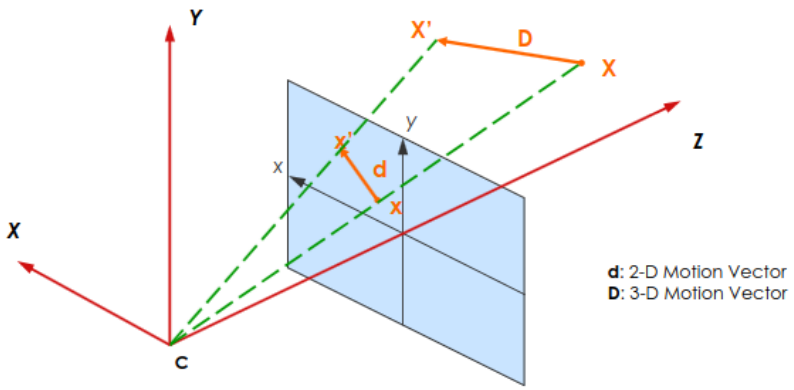


Figure 3.1: 2D Motion of a 3D object

We're able to perceive motion only if exist a projection of the image on the image plane. Single camera involves a problem: you can't understand if the point in front of you is growing or coming closer to you.
Let's consider an example:

The questions that came up are:

- How can I extrapolate the motion field?
- Does it really reflect the object displacement?
- Is the motion due to the camera or to the object?

Let's start by examining the motion field, which consists of dots and arrows. These arrows typically start from a point and indicate the previous position of that same point. However, these motion vectors are not always reliable. For example, in the hat area, we can observe inconsistencies.
To understand this better, imagine looking at a blank sheet of paper under ideal lighting. If you only see a small portion of the sheet, you wouldn't be able to tell if the sheet is moving. You'd need to see the edges and the entire figure in context to detect any movement.
Returning to our initial example, consider a sliding window moving across the flat, uniformly colored central part of the hat. The algorithm struggles to determine the motion of points in this area because they all have very similar colors. The algorithm relies on slight color differences to find the best match and score the motion vector. This is why areas with uniform color present a challenge for accurate motion detection. Finally, we can also see that the points related to the building are also assigned motion but this, barring earthquakes, is due to the motion of the camera itself.

2D motion of a rigid object
When analyzing the 2D motion of a rigid object, we assume the camera is fixed while objects move in 3D space. This motion results from a complex combination of arbitrary movements. The resulting 2D projection is often ambiguous because different

combinations of movements can produce the same image. Consequently, determining the specific types of movements affecting the scene through post-analysis becomes very challenging.

3.1 Motion detection

Motion detection has several applications, including detection, segmentation, recognition, filtering (such as deblurring and noise suppression), and compression for transmission and storage. Generally, the goal is to detect changes in image intensity over time.
However, this is an ill-posed problem with no unique solution, especially since we lose a dimension in the process. For instance, zooming can appear as translation. This issue arises because the detected changes do not always reflect actual motion. Real motion is defined by the velocity $v(x, y, t)$ of pixels between consecutive frames and can result from object movement, camera movement, or changes in illumination.

3.1.1 Real vs Apparent movement

2D motion is perceived by changes in the scene over time, but it's not always identified correctly. With a constant illumination, for example, a perfect rotating sphere is perceived as not moving. With an illumination source that rotates around a static sphere, the sphere is perceived as moving.

3.1.2 Occlusion

Occlusion is a problem that arises when a surface is covered/uncovered by the movement of an object. This makes it challenging to track or detect motion accurately because the missing information can lead to incorrect assumptions about the object's position or movement.
Now take a look at this example of a circle sliding over a surface. There is a portion of the background (the vivid red one) that will be covered by the circle, and a region(the pale red) that will be uncovered, that didn't exist in the previous frame. However, in the end we have notion of movement in these regions but we don't have any information about the white region where nothing seems to be happening.

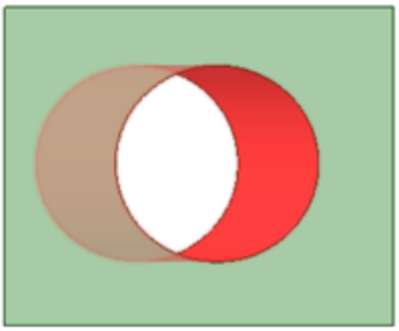


Figure 3.2: Occlusion

3.1.3 Aperture

It's a problem that arises when observing motion through a small, limited view (like looking through a tiny aperture). In such cases, only displacements of the borders are detectable and only in the direction of the intensity gradient, making it difficult to determine the true direction of motion.
To simplify, imagine dragging a paper left and right under a small slit. Even though the paper is moving and the edge is visible, no movement will be detected as the displacement is orthogonal to the edge gradient.

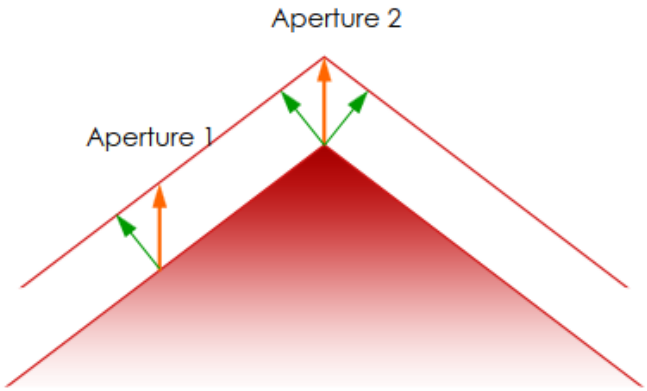


Figure 3.3: Aperture

3.1.4 Optical Flow

Optical flow describes the apparent motion of objects between consecutive frames. It is the combination of the motion of the object and the motion of the camera. At this

point things look complicated... but we can simplify the problem by making some assumptions:

- Object illumination does not change in $[t, t + dt]$;
- Distances do not change significantly;
- Each point $[x, y]$ is shifted in $[x + dx, y + dy]$, we only assume movement to be translational.

NB: This is not true in real video but good enough for a first approximation.

Optical flow equation

Hypothesis: object points keep the same intensity even though they are subject to a displacement along x, y and over time:

$$\psi(x, y, t) = \psi(x + dx, y + dy, t + dt)$$

We can use the Taylor expansion that says

$$f(x + \Delta x) = f(x) + f'(x)\Delta x$$

to approximate the function(for small dx, dy, dz), in this way we obtain:

$$\psi(x + dx, y + dy, t + dt) \approx \psi(x, y, t) + \frac{\partial \psi}{\partial x}dx + \frac{\partial \psi}{\partial y}dy + \frac{\partial \psi}{\partial t}dt$$

Comparing the two we can delete $\psi(x, y, t)$ that appears in both sides and obtain:

$$\frac{\partial \psi}{\partial x}dx + \frac{\partial \psi}{\partial y}dy + \frac{\partial \psi}{\partial t}dt = 0$$

Then, dividing by dt and rearranging the terms we achieve the optical flow equation:

$$\frac{\partial \psi}{\partial x}v_x + \frac{\partial \psi}{\partial y}v_y + \frac{\partial \psi}{\partial t} = 0$$

or, in a better form:

$$\nabla \psi^T \cdot \mathbf{v} + \frac{\partial \psi}{\partial t} = 0$$

NB: if $v = 0 \Rightarrow$ we have no variation.

We can see how the **gradient of the appearance** times the **velocity** compensates the variation over time.

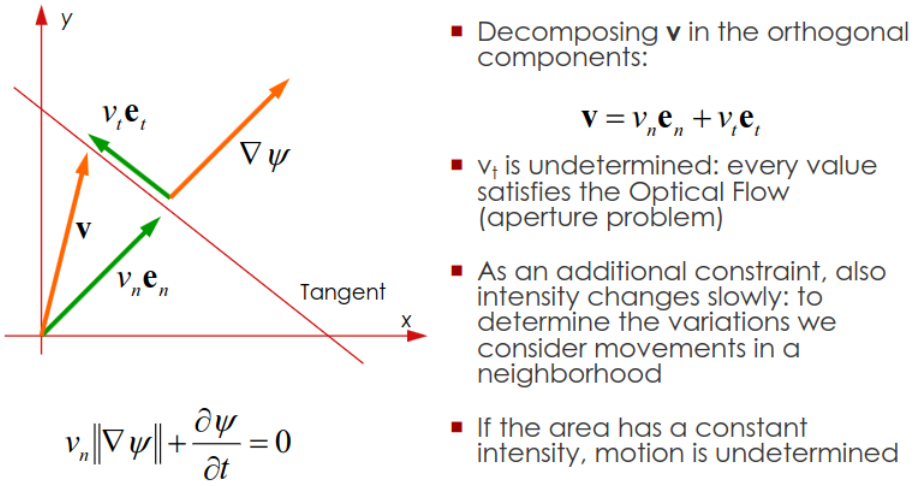


Figure 3.4: Optical Flow

In Figure 3.4 we see how we can decompose the velocity vector in two components: one orthogonal to the gradient and one parallel to it. Intuitively, the component orthogonal to the gradient isn't very informative, it doesn't cause any change in the appearance, while what's relevant to us, and what we can actually see is the component parallel to the gradient.

In order for motion to be **visible**, it needs to occur in the direction orthogonal to the border, parallel to the gradient. If instead the object is moving along the border or the area has a constant intensity, the motion is **undetermined**.

3.2 Motion detection in practice

We have two possible types of algorithms to detect motion: **Intensity based** and **Feature based**. However, we have some problems, like the representation of the motion field and the choice of discriminant parameter.

3.2.1 Change Detection

To begin, just take a look at the following example:
The two images on the left are the reference and current images. We compute the pixel-wise subtraction between them, resulting in a binary image where white pixels indicate changes.
In the first binary image, the moving boat is not detected well, making it difficult to understand the motion. Our goal is to create a motion map that accurately describes the

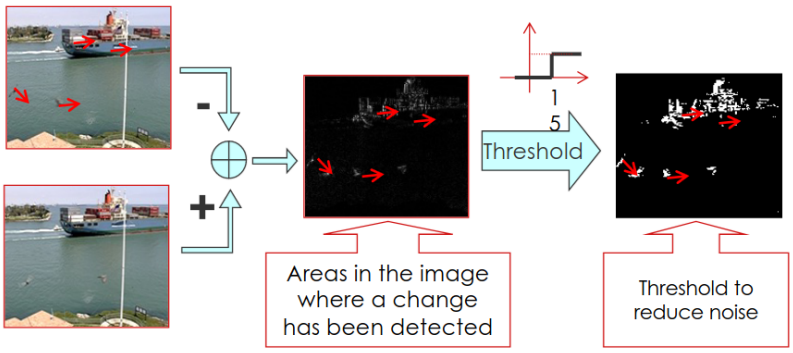


Figure 3.5: Change Detection

motion field and pixel displacement. To improve detection, we can adjust the subtraction threshold, reducing noise by removing pixels with minimal motion components.
Another issue involves birds in the scene. We see four birds instead of two because the chosen Δt is too large. Reducing Δt by increasing the sampling rate resolves this problem.
Additionally, a potential issue arises when using a single camera to track object velocity. For example, a car moving along the diagonal of a rectangle might appear to move slower as it moves away from a camera positioned at the bottom border. This misperception occurs because the velocity cannot be accurately computed from one camera angle alone.
Formally speaking: Change detection involves identifying regions in an image that have changed between two or more images. The objective is to create a motion map that describes the motion field, or the displacement of pixels. This method is effective when the illumination remains relatively constant.
In addition, frame difference corresponds to: $I_k - I_{k-1}$

$$FD_{k,k-1}(x_1, x_2) = s(x_1, x_2, k) - s(x_1, x_2, k-1)$$

If FD is non-null, a change has occurred. Change can be due to noise, so a threshold is needed to control it:

$$z_{k,k-1}(x_1, x_2) = \begin{cases} 1 & \text{if } |FD_{k,k-1}(x_1, x_2)| > \tau \\ 0 & \text{otherwise} \end{cases}$$

A good strategy to avoid noise is to use the so-called cumulative difference.

3.2.2 Frame differencing

The technique involves subtracting the current image from the previous one to produce a binary image. In this binary image, white pixels indicate areas where changes have occurred between the two frames. A crucial assumption for this method to work effectively is that the camera must remain stationary throughout the process. The essence of this method is to continuously update the background in each frame (cumulative difference), this means that this method doesn't have any form of memory, and it's not able to detect the motion of an object that stops moving.
For example:

```
B(0) = I(0);
...
loop time t
I(t) = next frame;
diff = abs[B (t-1) I(t)];
Map(t) =
threshold(diff);
...
B(t) = I(t);
end
```

This method offers several noteworthy considerations:

- **Adaptability to Background Changes:** The technique swiftly adjusts to alterations in the background, allowing it to effectively differentiate between static and dynamic elements within the scene.
- **Object Detection Dynamics:** Once an object comes to a standstill, it ceases to be detected, underlining the method's sensitivity to motion-based changes.
- **Contour Detection Capability:** The approach can discern the contours of objects, enabling a more nuanced understanding of the spatial transformations occurring within the frame.
- **Sparse Pixel Detection:** Typically, only a few pixels undergo detection, minimizing computational load while maintaining precision in identifying areas of change.
- **Potential Artifact Formation:** In cases where motion aligns parallel to the edge, artifact formation may occur, necessitating vigilance in interpreting the results to mitigate potential inaccuracies.

- **Temporal Scale Influence:** Altering the timescale can significantly impact the outcomes, emphasizing the need for meticulous calibration to achieve desired detection accuracy and reliability.

Regarding the last point, we can observe the following:

$$D(N) = ||I(t) - I(t+N)||$$

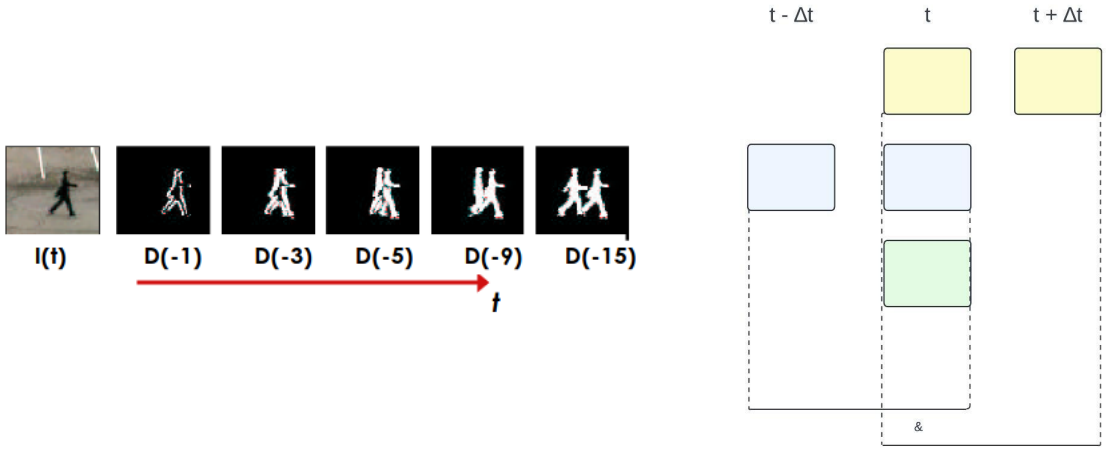


Figure 3.6: The contour of the object becomes clearer, until it doubles on departure and arrival points, doubling is not good because you can't tell which one is the real one. To solve this problem we can AND between 2 different frames and get as a result the real movement.

3.2.3 Background subtraction

In this method, we establish a static image as the background, typically obtained when no objects are in motion. Subsequently, each incoming frame is compared against this background to discern moving entities within the scene. To achieve this, we continuously update the background to accommodate gradual or sudden alterations in the environment.
The process involves calculating the disparity between the current frame and the background, assigning a label of 1 to pixels identified as part of the foreground (indicating movement) and 0 to those categorized as background. This binary labeling facilitates the extraction of dynamic elements within the visual field, enabling effective object detection and tracking.
Example:

```
B = I(0); #initial background
...
loop time t
I(t) = next frame;
diff = abs[B - I(t)];
Map(t) =
threshold(diff);
...
end
```

Comments:

This method presents both strengths and limitations worth noting.

It excels in detecting moving objects when there's a clear color contrast between them and the background. However, it struggles when objects initially enter the scene, often missing them during this phase. Additionally, background movement can lead to "ghosting," where both the real object and its duplicate are detected.

Furthermore, environmental changes like lighting variations, moving objects, and reflections pose challenges, causing false positives or negatives. Lastly, camera motion can confuse the method, blurring the distinction between actual object movement and shifts in perspective.

While valuable for object detection, understanding and addressing these limitations are essential for maximizing its effectiveness across various applications.

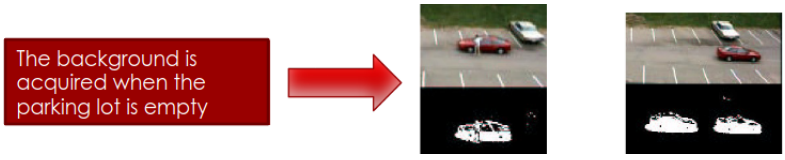


Figure 3.7: Possible problems of Background Subtraction

Adaptive background subtraction

The implementation of adaptive background subtraction is a good compromise between the two previous methods. In this method, a background image, usually acquired when the scene is static, is continuously updated over time to adapt to changes such as lighting variations or gradual scene alterations.

The process begins with establishing an initial background model, typically representing a static scene without any moving objects. As new frames are captured, the current frame is compared to this background model. Pixels that significantly differ from the

background are classified as foreground, indicating potential movement.
To adapt to changes over time, the background model is continually updated based on new observations.
More formally, we use a learning rate α to weight the contributions.

$$B_t = \alpha I_t + (1 - \alpha) B_{t-1}$$

Notice that if $\alpha = 0 \Rightarrow$ we have background subtraction (so no update), while if we have $\alpha = 1 \Rightarrow$ we obtain frame differencing.

3.2.4 Gaussian average

In Gaussian average background modeling, we conceptualize the background as a Gaussian distribution, characterized by its mean and variance. With each new frame, these parameters are updated to reflect any changes in the scene.
The process involves comparing the current frame with the background model to identify areas where movement occurs. This comparison enables the detection of moving objects against the relatively stable background.
To initialize the background model, a training sequence may be used, typically consisting of frames depicting an empty or static scene. This step ensures the accurate initialization of the Gaussian parameters.
In essence, Gaussian average background modeling shares similarities with adaptive background subtraction but offers enhanced stability (because instead using something related to the previous model, that basically consists of an average of frames, we use the mean value of a Gaussian function that are more stable). By leveraging statistical properties to represent the background, this method provides a robust framework for detecting moving objects in various dynamic environments.

$$\mu_t = \alpha I_t + (1 - \alpha) \mu_{t-1}$$

The advantages are its computational efficiency and the low memory requirements. Pixel goes to foreground if: $|I_t - \mu_t| > k\sigma_t$.

Quality depends on α, a too high value can lead to a slow adaptation to changes, a too low value can lead to a noisy background. Slow update = slow response to changes.

Improving the Gaussian average

To improve the Gaussian average we can use a binary operator to update values selectively, so only background values. M=1 if the pixel is foreground and 0 if it's back-

ground. This produces a more stable model and prevents pixels from being considered as background when they are not.

$$\mu_t = M\mu_{t-1} + (1 - M)(\alpha I_t + (1 - \alpha)\mu_{t-1})$$

NB: only if you know the situation, perturbations of BG are legit.

3.2.5 Mixture of Gaussians

With most techniques new objects are sooner or later absorbed in the background. This is because there are changes that appear and disappear faster than the update rate, such as tree leaves, rain, snow, water of a fountain, etc. It can be that the same pixel represents at different time instants tree leaves and sky/building behind. Unlike single Gaussian models, MoG represents the background as a combination of multiple Gaussians (**K** Gaussians), allowing better representation of complex scenes.

$$p(x_t) = \sum_{i=1}^{K} w_{i,t}\mathcal{N}(x_t, \mu_{i,t}, \Sigma_{i,t})$$

Each of them is used to model a foreground or background object. Theoretical model is complex, we have multi-variate Gaussians to model RGB or YUV.
Practically, the theoretical complexity is simplified by assuming independence among components and uniform variance across dimensions (*NB: this means that collapse to* σ^2), leading to a diagonal covariance matrix.
Let's take a deeper look...
$w_{[i,t]}$ = weight for the current Gaussian, $\mu_{i,t}$ = mean for the current Gaussian, $\Sigma_{i,t}$ = variance for the current Gaussian.
The process involves iteratively updating the model parameters based on incoming pixel observations. Pixels are compared against existing Gaussian models, and if no match is found, a new Gaussian is created. When a match is detected, the corresponding model is updated to reflect the pixel's characteristics.
So, select K, rank the Gaussians on the basis of weight, Standard deviation and Peak amplitude.
Notice that the higher the peak and the more compact the distribution (low variance), the more pixel is likely to belong to the background $\Rightarrow$ strong evidence. Among the K Gaussians, some belongs to the background and some to the foreground. The first B Gaussians are associated to the BG if $\sum_{i=1}^{B} w_i > T$. Each incoming pixel is checked against the available models. In case no match is found, a new Gaussian is created. If a

match is found (Es a match is found in the pixel is within $(x_t - \mu_{i,t}) < 2.5\sigma$), the model is updated. If the pixel is not matched with any model, the least probable Gaussian is replaced by the new one centered in x_t with low weight and high variance. Also, update of the weights is necessary:

$$w_{k,t} = \alpha M_(k,t) + (1 - \alpha) w_{k,t-1}$$

where $M_(k,t)$ is 1 for the matching model and 0 otherwise (if it's not the matching model, the weight is decreased).
As you can see, the update process incorporates a learning rate α to balance the influence of new observations with existing model parameters.

Chapter 4

Motion Tracking

Just to introduce the topic, we can say that motion tracking is the process of determining the movement of an object in a video sequence. More in general, it regards the understanding of *what* is moving in a scene and *how* it moves or interacts with the environment.

This is a very important topic in computer vision, and it is used in many applications, such as video surveillance, traffic monitoring, human-computer interaction and biological applications (such as cell tracking). At high level, motion tracking can be used for the discovery of activities, behavior understanding, and detection of threats.

In this chapter, we will discuss the basic concepts of motion tracking, and we will present some of the most common algorithms used in this field.

4.1 Object Tracking

Object tracking is the process of continuously locating a moving object over time. Depending on the application's requirements, tracking can be done in various ways: tracking the 2D coordinates of an object's centroid, tracking in 3D (which requires multiple cameras), or determining the positions of complex objects, such as the articulations of a human body.

The primary goal of object tracking is to estimate the object's state in each frame and predict its future position. This capability has a wide range of applications. In monitoring and surveillance, object tracking can be used for motion classification, identifying anomalous or suspicious behaviors, and following a trajectory.

In human-machine interfaces, it enables interaction with devices without physical barriers like a mouse or keyboard, and aids in natural language understanding. In virtual reality, object tracking contributes to creating immersive experiences and ani-

mating virtual characters.

Additionally, object tracking is valuable in mining and retrieval applications, such as browsing databases for specific motion patterns. Overall, object tracking is a versatile technology with numerous practical uses across various fields.
Some benefits of object tracking are:

- In HCI, control PC (or systems in general) ⇒ no need for additional tools;
- In surveillance, Automated / Semi-automated systems ⇒ reduce the stress of human operators;
- Virtual reality, computer animation ⇒ animate and drive the avatar;
- But also: Training of athletes, Gait disorders detection, Medical applications, etc.

4.2 2D Tracking

Sometimes it is enough to track the object in 2D, for example when the object is moving on a plane. Exists different approaches to 2D tracking:

- **Region-based** ⇒ set of pixels that share similar features (color). This does not work very well with multiple objects moving at the same times, on top of that it's computationally intensive;
- **Contour-based** ⇒ determine position and shape of an object over time. Useful to track deformable objects;
- **Feature-based** ⇒ select meaningful points (contours, corners). Very common and effective method, relevant points are selected and tracked over time;
- **Template-based** ⇒ use specific models (hands, faces, eyes).

4.2.1 Region-based tracking

When it comes to real-time applications, tracking regions with uniform appearance offers several benefits. It enables swift processing, often exceeding 30 frames per second, while maintaining a commendable balance between quality and speed. The idea revolves around identifying areas in an image that exhibit consistent color characteristics. These regions are akin to patches of similar color projected onto the image plane, often achieved through segmentation techniques like background suppression.

One fundamental requirement for successful region tracking is ensuring that these regions possess distinguishable colors. However, this method faces challenges, particularly in scenarios with variable illumination. Changes in lighting conditions can destabilize the uniformity of color, complicating the tracking process. To mitigate this issue, various compensation techniques can be employed, such as utilizing the hue and saturation components of the HSV color space or normalizing the RGB values. While these techniques work reasonably well indoors, outdoor settings may present more difficulties due to unpredictable lighting changes.

In terms of what we aim to track, the possibilities are diverse. It could involve monitoring any moving object, distinguishing between skin and non-skin regions (useful for applications like hand and face tracking), identifying specific colored areas, and more. The approach typically involves techniques such as color thresholding for uniform regions or utilizing color histograms.

However, tracking regions with uniform appearance isn't without its challenges. Color variations over time due to changes in illumination or object posture pose significant hurdles. Additionally, models of tracked objects need continual updates to adapt to evolving conditions.

One conceivable approach to tackle these challenges involves breaking down the object into smaller regions. Each region is then associated with a *color vector* or histogram, representing the average color values within that region. During tracking, the color of each region is computed, and the similarity to a reference model is assessed. If the ratio between the reference and current values is close to 1, it indicates a good match.

Histograms serve as a valuable tool in this process. They provide a quantified representation of the color distribution within a region, enabling comparisons between reference and current models.
Different similarity measures can be employed for evaluation, such as bin-by-bin comparison:

$$\bigcap(O_i^t, O_i^r) = \sum_{n=1}^{U} min\{O_{i,n}^r, O_{i,n}^t\}$$

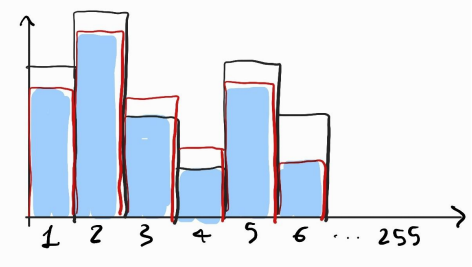


Figure 4.1: For all the bins of each object we compute the intersection (blue), one that yields the highest value is the best match.

or Sum of Squared Differences (SSD), so instead of computing the correlation between the two histograms, we compute the difference between the two histograms, the error:

$$SSD(O_i^t, O_i^r) = \sum_{n=1}^{U} (O_{i,n}^r - O_{i,n}^t)^2$$

Careful consideration must be given to the number of bins used in the histograms, striking a balance between granularity and computational efficiency. A smaller number of bins may also lead to better generalization, as it reduces the impact of noise.

In summary, tracking regions with uniform appearance offers a promising avenue for real-time applications, but it requires robust techniques to address challenges such as variable illumination and color changes over time. Techniques involving region division and histogram analysis can help achieve accurate and efficient tracking in these scenarios.

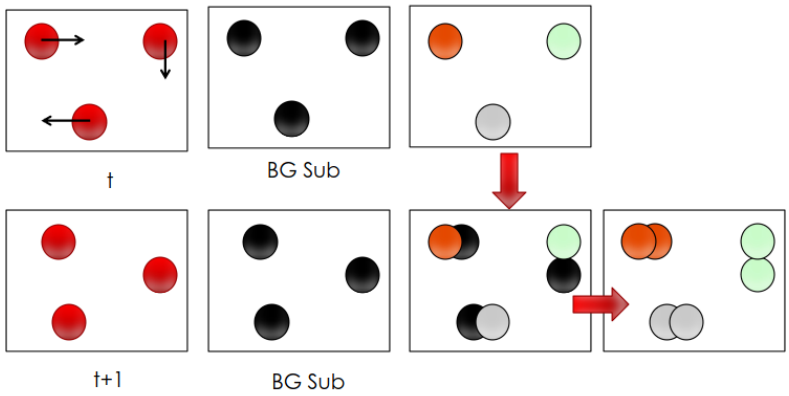
NB: Shadows can be problematic in region-based tracking as they introduce noise, leading to false positives. A shadow does not correspond directly to the motion of a real object; rather, if we are lucky enough, it represents a change in the object's color, specifically a variation in luminance while chrominance remains ideally unaltered. To ensure proper tracking, shadows should be removed beforehand using a suitable algorithm.

4.3 Blobs extraction

An object can consist of multiple blobs, such as one for the torso and one for the head, disjoint one from the other. Blobs are aggregations of connected pixels that share common features. When considering features, position is also important, so pixels with similar color but located far from the object must be discarded. This approach is typically used in combination with background suppression to enhance object detection and tracking.

4.3.1 Target association

This is a procedure common to all trackers, not only region-based. In general, it is worth noting that detection is not carried out on a frame-basis. This could lead to a lot of false positives, and it is computationally expensive. The idea is that once detected, targets are followed on a proximity basis. So, for example, background subtraction informs about the presence of motion, histograms characterize each moving objects, and then, unless occlusion occur, the target in the next frame should be the closest blob.



The steps of target association can be summed up as follows:

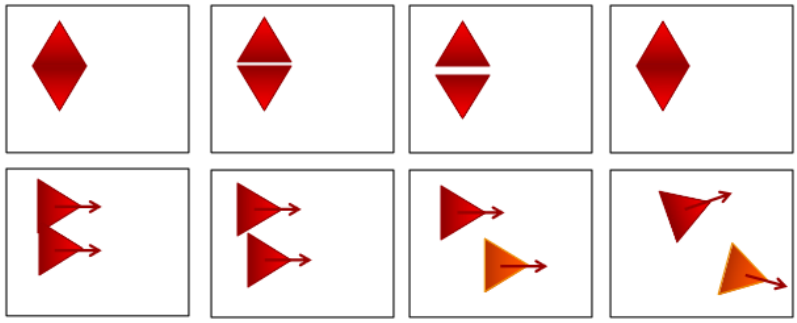
- Overlapping blob (issues with scale ⇒ depends on objects size);
- Centroid with minimum distance (as above);
- Overlapping bounding boxes (may fail due to perspective);
- Bounding box with minimum distance;
- Bounding box to centroid distance.

NB: When association has been completed, for each object or blob update the appearance model to account for small variations. In presence of occlusions, the last saved model can be used to disambiguate.

4.3.2 Splitting

Splitting is a common problem in blob extraction because an object can be made up of several blobs. If each object is identified as a single blob, there are no issues. However, background subtraction can give confusing results. Objects might be broken into

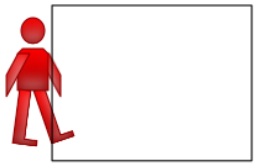
several small blobs, meaning there isn't a clear separation between background and foreground. Similarly, two objects might enter the scene together and then separate.



To figure out what's happening, it's important to gather more information over time. It's impossible to understand right away whether it's a single object that looks fragmented or two objects close together. Therefore, we need to observe for a while to decide if the blobs belong to one object or two separate ones.

4.4 Merging

Merging is the opposite of splitting, and it is due to the fact that two objects are identified as a single blob. This can happen when two objects are close to each other and move together consistently (then perhaps they're the same object). An example can be when a person's arm and foot enter the scene first and are detected as two well-separated FG blobs. Then also the rest of the body enters, and a single blob is created.



4.4.1 Criteria for splitting and merging

Observation is crucial in determining whether regions should be split or merged. To make accurate decisions, we need to monitor regions of interest and evaluate their consistency based on several factors. Consistent direction of motion can indicate that multiple regions belong to the same object, while the distance between centroids or bounding boxes helps to identify whether regions are part of the same object based on their proximity. By observing the temporal range, we can determine if regions consistently behave as a single entity over time. Additionally, regions moving at similar speeds are likely part of the same object, and feature matching (comparing characteristics like color, texture, or shape) further aids in deciding whether regions should be

considered as one object. Evaluating these criteria ensures more accurate tracking and identification of objects.

4.5 Occlusion

Occlusion is an anomalous, but very frequent, situation in tracking, and it occurs when moving objects overlap. Consequently, one or more objects disappear from the scene and bigger blobs appear as a result of the occlusion, with properties that don't belong to any of the models acquired previously, so the acquisition models are not reliable anymore.
NB: Model update should be avoided during occlusions.

In order to solve this problem, we need to re-associate "A to A" and "B to B" (where A and B are the objects that are occluded).
NB: Histograms are a good way out.

4.6 Tracking: Feature-based

Histograms are just one way to track objects, we also mentioned other methodologies, such as feature-based tracking. The objective is to retrieve the motion information from sets of features (such as points, corners, edges...) in local areas of the image.
Considering a set of images

$$A = \{A(0), A(1), A(2), \ldots, A(j-1)\}$$

and the position of the feature in the image plane in each frame:

$$m_i(x_i, y_i) \quad i = [0, j-1]$$

The objective is to determine the displacement vector:

$$d_i = (dx_i, dy_i)$$

That best estimates the position of the feature in the next frame:

$$m_{i+1}(x_{i+1}, y_{i+1}) = m_i + d_i$$

We'd like to get the coordinated x and y of the feature in the next frames at time $i+1$. (In principle a feature point does not convey much information, usually we group them to create a bounding box or a convex hull that can contain the moving object into an independent structure).

The question now is how to choose a good feature point. Generally speaking it should have distinctive characteristics such as brightness, contrast, texture, edges, corners or points with high curvature, basically something that exhibits some elements that let that point be sufficiently diverse compared to the rest of the image.

We can use the **Good Features to Track** algorithm:

$$Z = \begin{bmatrix} \sum_W J_x^2 & \sum_W J_x J_y \\ \sum_W J_y J_x & \sum_W J_y^2 \end{bmatrix}$$

Where J_x and J_y are the gradients evaluated on a certain point in the x and y direction within a certain window W (of size $n \times n$). A good feature point is the with strong gradients component in both directions, so we are interested not only in the x and y components but also in the cross-correlation between them. To maximize this function we compute the eigenvalues of the matrix. In order to evaluate if it's a good feature point we are interested whether the smallest eigenvalue is larger than a certain threshold. This means that in case we have just one of the eigenvalues with high magnitude and the other very small probably it is a feature point with good characteristics only in one direction, what we are looking for is a point with strong components in *both* directions, in practice, it highlights corner points and texture.

In summary, it's important to consider the following key points when analyzing eigenvalues:

- Eigenvalues should surpass the image noise level to ensure reliable information.
- Small eigenvalues indicate strong similarity within the analyzed window.
- Presence of one large and one small eigenvalue suggests unidirectional patterns in the image.
- Points with two large eigenvalues are of high interest, indicating features like salt and pepper texture or corners.

For what concerns tracking, we must ensure that the same points are tracked throughout the video sequence. Ideally we would expect that $A_i(Dm - d) = A_{i+1}(m)$, where A_i, A_{i+1} are the successive frames at time i and $i + 1$, m is the 2D position of the feature point, D the deformation matrix (affine transformation model) and d is the displacement vector (translation models).
However, due to noise the equality usually does not hold. In addition, since motion

across successive frames is assumed to be small, a *translational model* is a good approximation and the term D can be removed. So, after making this simplification, we can use it to find the configuration of the displacement vector d that *minimizes* the residual between the two frames:

$$\varepsilon = \iint_W [A_i(m-d) - A_{i+1}(m)]^2 \, \omega(m) \, dm$$

we are also implementing a weighting function ω such as Gaussian to emphasize the center of the window giving less relevance to the terms that are outside the center.

Inside the scene we are observing a lot of things can happen, for example the points we are looking for get occluded, exit the scene, or change color. In that situation the dissimilarity grows a lot. When the feature dissimilarity grows too large, the feature point should be abandoned and a new one should be selected. To minimize the residual we differentiate and zero w.r.t unknowns (d)

$$e = 2 \iint_W [A_i(m) - A_{i+1}(m)] \, g(m) \, \omega(m) \, dm$$

and

$$g(m) = \begin{bmatrix} \frac{\partial (A_i(m) - A_{i+1}(m))}{\partial x} \\ \\ \frac{\partial (A_i(m) - A_{i+1}(m))}{\partial y} \end{bmatrix}$$

In this case the solution for the displacement vector can be expressed by the 2x2 linear system of equations (see paper for details):

$$Zd = e$$

The typical combination and feature detection and feature tracking takes as input the *Good features to track* algorithm for the detection which is coupled to another algorithm for the tracking, such as the Lucas-Kanade optical flow.

4.6.1 The Lucas-Kanade optical flow

Substantially is a two-frame differential method for optical flow estimation. Consider $u = [u_x, u_y]$ in frame I and $v = [v_x, v_y]$ in frame J the goal is to find d that satisfies the equation $v = u + d$ such as I and J are similar(translation model). We are not expecting rotations or distortions, for this reason the similarity is defined in the 2D domain. d is the vector that minimizes

$$\epsilon(d) = \epsilon(d_x, d_y) = \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} [I(x,y) - J(x+d_x, y+d_y)]^2$$

This above is pretty much the same thing we have seen before substituting the continuous integral with a discrete summation. The ω function has now become a window of size $w_x \times w_y$ centered in u_x, u_y.

4.6.2 Pyramidal implementation

The strong point of the Lukas-Kanade optical flow is the pyramidal implementation. In order to be a good tracker we need to satisfy two requirements: accuracy and robustness.

Accuracy relates to the local sub-pixel accuracy attached to tracking, we want smoothness and to preserve detail information ⇒ **small** integration window preferable

Robustness relates to the sensitivity of tracking with respect to changes of light and big motions. We never know what's going to happen in the scene ⇒ **large** integration window preferable

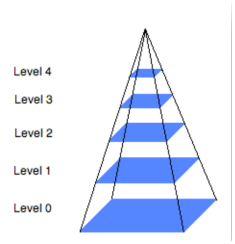


Figure 4.2: Pyramidal representation of the Lukas-Kanade optical flow.

In order to satisfy both requirements, the authors built a tracker which works on multiple scales. It means we are not working on a single image, instead we are working on a number of levels, small movements in the higher levels correspond to big movements in the lower levels, as shown in Figure 4.2.

- Level 0 is the image at original resolution.
- Level 4 is the image at lowest resolution.
- The L-th level is defined as a linear combination of the elements in the previous level.

$$
\begin{aligned}
&I^L(x,y) = \\
&\tfrac{1}{4} I^{L-1}(2x, 2y) + \\
&\tfrac{1}{8}(I^{L-1}(2x-1, 2y) + I^{L-1}(2x+1, 2y) + I^{L-1}(2x, 2y-1) + I^{L-1}(2x, 2y+1)) + \\
&\tfrac{1}{16}(I^{L-1}(2x-1, 2y-1) + I^{L-1}(2x+1, 2y+1) + I^{L-1}(2x-1, 2y+1) + I^{L-1}(2x+1, 2y-1))
\end{aligned}
$$

To construct higher levels, the contributions of the pixels around the pixel in the lower level are taken into account and weighted. Then what we do is that we start our analysis from the highest level, the smallest. How do we do that?

$$\epsilon^L(d^L) = \epsilon^L(d_x^L, d_y^L) = \sum_{x=u_x^L-w_x}^{u_x^L+w_x} \sum_{y=u_y^L-w_y}^{u_y^L+w_y} \left[I^L(x,y) - J^L(x+g_x^L+d_x^L, y+g_y^L+d_y^L)\right]^2$$

Starting from the lowest level we make a guess of the flow **g** and, looking at the observation window, we find the first displacement vector, this means that after minimizing the quantity ϵ we have an initial guess of the displacement.

After that we go to the lower, bigger level. Scaling the motion vector as it doubles its scale, but now we have more pixels and a better chance or refining the guess just by looking in the new neighborhood and fine-tuning the displacement vector. The propagation of information to the lower levels follows this equation:

$$g^{L-1} = 2(g^L + d^L)$$

Overall the displacement becomes:

$$d = \sum_{L=0}^{L_{max}} 2^L d^L$$

4.6.3 Bayesian tracking

The approaches seen so far were based on the assumption we were computing some sort of distance that would give us the chance to find again the point of interest within a certain neighborhood, we will now see an extension of this approach, by giving it a probabilistic twist.

We are considering the tracking problem as a process in which we are making **estimations** about the position of the object given a certain evidence that we are collecting over time. We represent the state of a system in terms of x and y coordinates but not only, we also take into account the *velocity* and the *acceleration* of the object along both dimension.

The point in this image plane will have a 6D state vector that will describe the state of our system at a given time instant. We expect and take into account noise, but we assume that it's smaller than the amount of information about the state.

$$x_k = f_k(x_{k-1}, w_{k-1})$$

The state x at time k can be seen as a function that depends on the previous state of the system plus some noise w that we need to consider making a proper analysis. The process noise is whatever cannot be controlled directly and *just happens*.

$$z_k = h_k x_k + v_k$$

In order to make the prediction it's not enough to just rely on the initial position of the subject and just propagate this information about the speed, because there's noise. In Bayesian Tracking when necessary we take a *measurement* trying to confirm our hypothesis. Our measurement z_k is the result of a funtion h_k which takes into account the state of the system plus some measurement noise v_k (may be due to discrete pixels, ambiguous shape etc...).

We can now construct a model in a way that we can start from a prediction, take a measurement "on the ground" and, similarly to Lukas-Kanade, check if the measurement is in line with the prediction, if not we can correct the measurement, the correction that we take (the measurement of the displacement between prediction and measurement) will guide us in the next step.

We are working *online*, using an observation to perform an estimate. More formally we can say that the initial PDF of the state of the system should be given, we should know *at least* the objects that we are going to track, once we know what we are looking for we can take as initial starting point the initial probability $p(x_0|z_0)$ where at the beginning z_0 contains no measurements. Our goal is, for every time step k, to compute $p(x_k|z_k)$, the probability of the state of the system given the measurements we have taken up to that time instant.

We have said that:

$$x_k = f_k(x_{k-1}, w_{k-1}) \rightarrow p(x_k|x_{k-1})$$

$$z_k = h_k(x_k, v_k) \rightarrow p(z_k|x_k)$$

Considering these two quantities we can create a model for the estimation of the state of the system that consists of these formulations:

$$p(x_k|z_{k-1}) = \int p(x_k|x_{k-1})p(x_{k-1}|z_{k-1})dx_{k-1}$$

Prior at time step k System model Posterior pdf at k-1

We want to create a prior $p(x_k|z_{k-1})$ meaning that I have taken a measurement in the previous step, this measurement relates to that state of the system $k-1$, and thanks to this measurement I can create a *prior*, I can project this measurement ahead in time to give me the new state x_k.

And how is this computed? I can use some posterior information, which is the correction of our $k-1$ prediction given the measurement, this corrected piece of information is propagated forward through our **system model** $p(x_k|x_{k-1})$ which tells me how to go from x_{k-1} to x_k.

Using the Bayes theorem it is possible to obtain the desired pdf:

$$p(x_k|z_k) = \frac{p(z_k|x_k)p(x_k|z_{k-1})}{p(z_k|z_{k-1})}$$

Where $p(z_k|z_{k-1})$ is used for normalization and computed as

$$p(z_k|z_{k-1}) = \int p(z_k|x_k)p(x_k|z_{k-1})dx_k$$

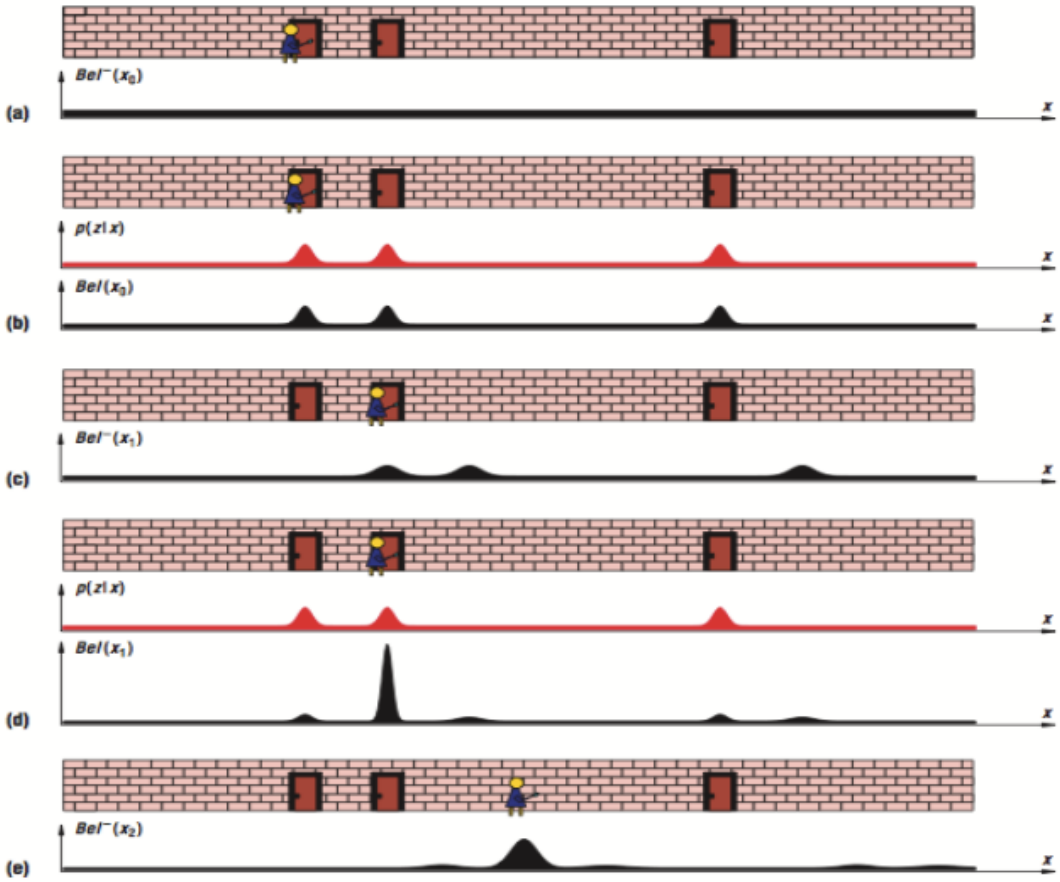


Figure 4.3: Bayesian tracking toy example.

In the toy example 4.3 the person is equipped with a "door sensing device" and we assume constant speed.

(a) We can see how at the very beginning we have no idea where the person is, so our estimation about the position of the person is that it's anywhere in the area.

(b) What happens next is that the guy approaches a door we get a signal, but we only have that information, the person could be at any of the three positions, so our $p(z|x)$, the probability of a measurement given a state of the system, raises 3 peaks in correspondence of the three possible positions of the person, the doors. This becomes out a posteriori information which is something that we are storing and we need to propagate ahead in time.

(c) We can propagate the a posteriori information ahead in time using the system model, in this case, the constant speed model, and we can see that the peaks are moving to the right, this is the prediction of the system. We also know that the

uncertainty is increasing, this is due to the noise, for this reason the peaks are getting wider.

(d) At this point the door sensing device starts ringing again, meaning the guy has reached one of the other 3 doors. We don't know which one but given our model propagated in time, we can construct a new a posteriori taking into account $Bel^{-}(x_1)$ and the new measurement $p(z|x)$ and, while we still have some components due to noise, we are quite sure that the guy is at the second door.

(e) As we progress over time, we propagate the information, the uncertainty increases until at some point we get a new measurement.

4.6.4 The Kalman Filter

The idea of the Kalman filter goes back to Bayesian tracking. We take a prediction, validate that prediction with a measurement and knowing that both prediction and measurement are subject to noise, we can combine them in a way so that we can minimize the error in the estimation of the state of the system.

We start from

$$z_1, \sigma_{z_1}^2$$

$$\hat{x}_1 = z_1$$

$$\sigma_1^2 = \sigma_{z_1}^2$$

Where z_1 is the measurement that already includes some noise that we define with a variance $\sigma_{z_1}^2$. We provide the first prediction by initiating it at the first measurement location, and we just map the variance of the measurement to the variance of the prediction.

Starting from this we can propagate the information over time, make a new prediction, take a measurement, correct the measurement and make the new prediction and so on.

$$z_2, \sigma_{z_2}^2 \quad \hat{x}_2 = ? \quad \sigma_2^2 = ?$$

We need to come up with a second measurement z_2 with its variance, make a new prediction and come up with a new value for the noise.

$$\hat{x}_2 = \hat{x}_1 + K_2(z_2 - \hat{x}_1) \qquad K_2 = \frac{\sigma_1^2}{\sigma_1^2 + \sigma_{z_2}^2}$$

$$\frac{1}{\sigma_2^2} = \frac{1}{\sigma_{z_1}^2} + \frac{1}{\sigma_{z_2}^2} \qquad \sigma_2^2 = \frac{\sigma_{z_1}^2 \sigma_{z_2}^2}{\sigma_{z_1}^2 + \sigma_{z_2}^2}$$

Figure 4.4: Simplified Kalman Filter.

In the first equation we rely on the information we collected before, and we add a displacement to this quantity to get the new prediction $\hat{x}_2$ by taking into account some additional parameters. Capital K is what we call the **Kalman gain**, it's a multiplying factor which weights the distance between the second measurement z_2 and the previous estimate $\hat{x}_1$: we know that from the previous estimate and the new measurement we have an error, and so we use this error information together with the Kalman gain to push ahead our estimation. The Kalman Gain is a combination of the variances we are collecting. It's basically a weighted average between the prediction and the measurement, the weight is given by the ratio of the variances.

The process is rather simple because we basically have to iterate between these two steps, the first one is prediction and the second one is measurement: first we predict the new state and the correspondent uncertainty σ^2, then we correct using the new measurement z for which we know there's a variance associated σ_z^2, this piece of information is the one that will guide us torwards the next prediction and variance.

The Kalman Filter is also a computationally efficient solution to the least squares method. With LSM, we are trying to minimize an error, similarly with Kalman's formulation what we are trying to do is to find the configuration of the Kalman Gain (which is also the only parameter we can tune) to make sure that the result obtained from the prediction is as close as possible to the real position of the object.

Assuming that the noise components w_k and v_k are normal distributions, the overall state and measurement model can be expressed through the following equations:

$$x_k = A_k X_{k-1} + B_k U_k + w_{k-1}$$

$$z_k = H_k x_k + v_k$$

Looking at the measurement equation we can see that the measurement is something that is going to happen on the basis of x_k (we previously mentioned how z_k is a function

of x_k and the noise) and this function is nothing but a multiplication of the measurement matrix H_k and x_k, plus the noise contribution v_k.

Similarly, for the state information x_k we have the system model A_k which projects the state of the system at time $k-1$ forward in time. In this formulation we have an additional term $B_k U_k$ which is the control input: we can model additional information that we can inject into the system to guide the state of the system in a certain direction, for instance if we directly press gas on a car or if we adjust the trajectory of a missile.

The noise in the process is usually modeled as a Gaussian distribution with zero mean and covariance matrix Q and R:

$$p(w) = N(0, Q)$$

$$p(v) = N(0, R)$$

The error is one of the driving forces in our model, a very relevant parameter, in fact we have seen in Equation 4.4 how, if there's no noise, the Kalman Gain would disappear. Not only that, if the measurement is correct and the error zero we wouldn't be able to project ahead in time the state of the system.

4.6.5 Predict-and-correct stages

During the first phase the current state estimate together with the error estimate are propagated forward in time, in the second stage a new measurement is taken to modify the two estimations. This gives us the chance to have an **a priori estimate** of the state of the system, before taking any new measurement, based only on the past knowledge; and an **a posteriori estimate** which is the corrected version of the a priori estimate, after taking the new measurement into account.

NB: while maybe only Equation 4.4 is worth remembering by heart, we will quickly go through all of them to complete the pipeline

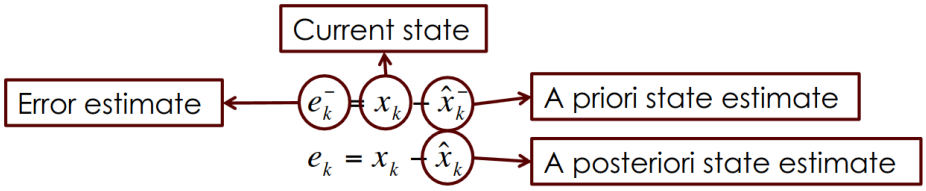


Figure 4.5: The minus means that it is an a priori information, modeled via our current state and our a priori estimate. This error allows us to make an a posteriori estimate $\hat{x}_k$ that will be used to estimate the new error and so on.

Once we have the error we can compute the covariance matrices, both a priori and a posteriori:

$$P_k^- = E[e_k^- e_k^{-T}]$$
$$P_k = E[e_k e_k^T]$$

After that we can start the prediction (the new a priori estimate) and compute the new error matrix (we are discarding the additional control parameter B for simplicity):

$$\hat{x}_k^- = A_k \hat{x}_{k-1}$$
$$P_k^- = A_k P_{k-1} A_k^T + Q_{k-1}$$

This leads to the definition of the new Kalman Gain, which needs to take into account the error estimate:

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

The gain is used to modify the a priori estimate and to compute the a posteriori state estimate:

$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H_k \hat{x}_k^-)$$

And to compute the a posteriori error covariance, which is then used to compute the new prior and so on:

$$P_k = (I - K_k H_k) P_k^-$$

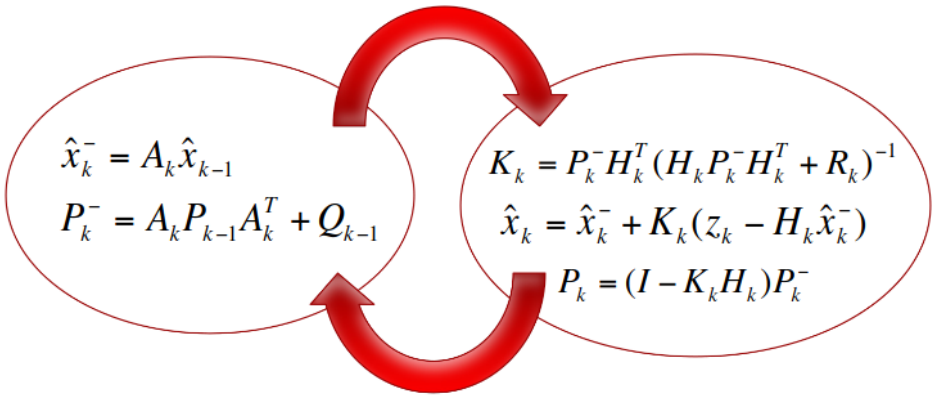


Figure 4.6: While *it's not necessary to remember the equations* it must be clear how the information flows in the algorithm.

4.6.6 Simple Kalman Filter example

The situation is that we are trying to estimate the value of a constant function, the voltage out of a socket, we know that the voltage is 220V but we are not sure about the noise in both the measurement and in the process.

Figure 4.7: The function we are trying to estimate.

The problem is easy: obviously we know something about the model, otherwise it would be impossible to make an estimate, and knowing that this is a constant we are sure there's no evolution in both the model A and the measurement H matrices, so:

$$A = 1$$

$$H = 1$$

$$x_k = x_{k-1} + w_k$$

$$z_k = x_k + v_k$$

Our a priori state estimate is given by the previous state and the error covariance is just changed by the presence of some noise:

$$\hat{x}_k^- = \hat{x}_{k-1}$$
$$P_k^- = P_{k-1} + Q$$

At this point we just have to use the equation of the Kalman filter, compute the Kalman gain, thanks to that estimate the new posterior and the new error matrix:

$$K_k = P_k^-(P_k^- + R)^{-1}$$
$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - \hat{x}_k^-)$$
$$P_k = (1 - K_k)P_k^-$$

To make the algorithm work we just have to select as starting value the initial state "wrong" and the initial non-zero error covariance, then we can iterate the algorithm and see how the estimate converges to the real value of the function.

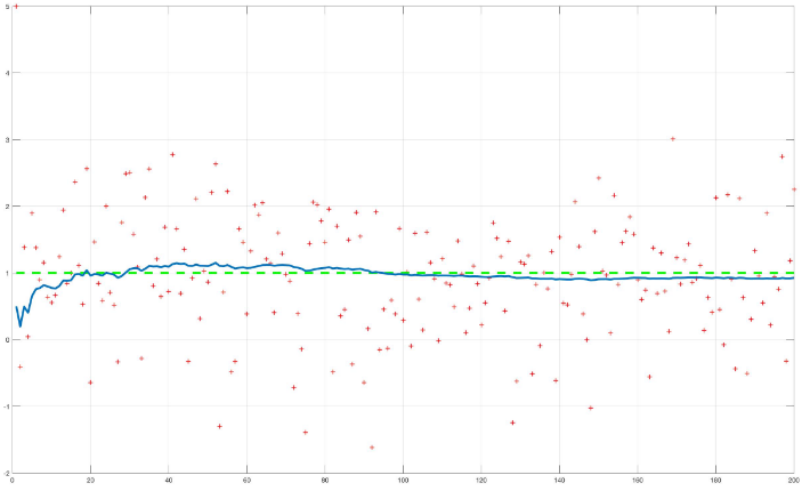


Figure 4.8: The results of our simplified example.

4.6.7 Extended Kalman Filter

An assumption that we made is that the behavior of the system is linear, the question is whether this reflects or not what happens in the real world. Turns out it does not.

The goal of the **Extended Kalman Filter (EKF)** is to adapt the Kalman Filter to linearize the equations that we used in the standard version. To define our state x_k and measurement z_k we are still using f_k and h_k, the only difference is that it's no more linear functions.

$$x_k = f_k(x_{k-1}, w_{k-1})$$

$$z_k = h_k(x_k, v_k)$$

Considering that we are assuming that the process is not linear, in order to linearize them we use partial derivatives of both A and H. Since we don't know the values of noise we can assume here for simplicity that state and measurement are:

$$\tilde{x}_k = f(\hat{x}_{k-1}, 0)$$

$$\tilde{z}_k = h(\tilde{x}_k, 0)$$

Using the same notation as for KF, the **prediction** stage is given by:

$$\tilde{x}_k = f(\hat{x}_{k-1}, 0)$$

$$P_k^- = A_{k-1} P_{k-1} A_{k-1}^T + W_{k-1} Q_{k-1} W_{k-1}^T$$

Where A_k is the Jacobian matrix of partial derivatives of f with respect to x_k.
W_k is the Jacobian matrix of partial derivatives of f with respect to w_k.
Q_k is the process noise covariance matrix
And the update loop becomes:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1}$$

$$\hat{x}_k = \tilde{x}_k + K_k(z_k - h_k(\tilde{x}_k, 0))$$

$$P_k = (I - K_k H_k) P_k^-$$

H_k is the Jacobian matrix of partial derivatives of h with respect to x_k
V_k is the Jacobian matrix of partial derivatives of h with respect to z_k
R_k is the measurement noise covariance matrix

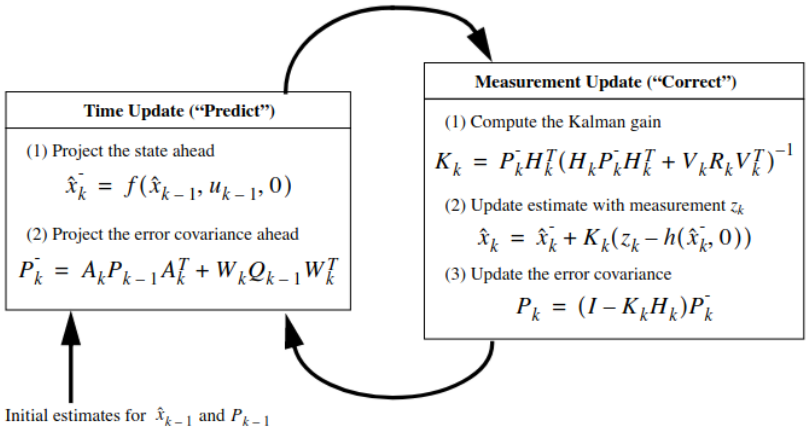


Figure 4.9: This is the final update loop for the EKF.

This helps us to deal with processes that are non-linear. However, this is only partially true in the sense that it's still very common to use the standard Kalman filter also in presence on non-linear processes with some approximations (the assumption that between two successive states we don't have big differences, so the non-linearity can be somewhat neglected), the EKF is used when the non-linearity is very strong.

4.6.8 Particle Filters

Still, there are some situations where the EFK cannot provide satisfactory results. With **Particle Filters** the goal is to make it possible to track objects that are not only *non-linear* but also that are behaving in a *non-Gaussian* way.

The idea is to replace the single estimation that we are doing with KF and EFK with multiple representations: we are providing alternative solutions to our problem.

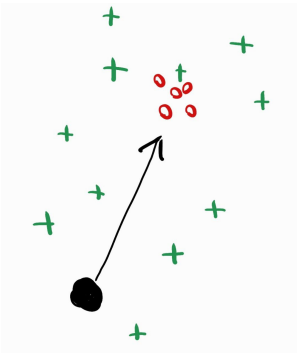


Figure 4.10: Red circles represent possible Gaussian predictions meanwhile the green crosses are possible predictions or hypothesis of a non-Gaussian model.

Instead of looking at a single point I'm checking multiple different solutions. The higher the number of particles the closer we are getting to the optimal Bayesian estimate, in order to implement an effective particle filter, we would need to have a very high and dense number of possible solutions.

All the samples could be, in principle, possible solutions to the problem, however by the time we evolve with the algorithm we should be able to understand that some samples are more likely than others, we typically assign to each sample a weight that is proportional to the likelihood of that sample to be the correct one.

What we do is that we are replacing what, in the KF, was an optimal solution to the linear problem with an approximated solution for linear and non-linear problems.

KF	linear	Gaussian
EKF	non-linear	Gaussian
PF	non-linear	non-Gaussian

4.6.9 How PFs work

At the beginning we take a set of points with corresponding weights, these sets are the possible next states of the system, we have n points that will be evaluated at state k:

$$\{x_k^i\}, \{w_k^i\} \;\; i = 1, 2, \ldots, n$$

Instead of having an integral representation of the a posteriori, we have an approximate solution and we are doing this with a summation that represents the shift of the points with a corresponding weight:

$$p(x_k|z_k) \approx \sum_{i=1}^{n} w_{k-1}^i \delta(x_k - x_{k-1}^i)$$

How do we select $\{x_k^i\}$ and $\{w_k^i\}$ for the approximation? We have to come up with a **proposal distribution** which is kind of an arbitrary choice that tells where to position the points considering the position of the point in the previous state.

$$x_k^i \approx \pi(x_k^i|x_{k-1}^i) \;\; i = 1, 2, \ldots, n$$

We draw these points:

$$\pi(x_k|x_{k-1}) \quad usually \quad p(x_k|x_{k-1})$$

We update the weights of the points and normalize:

$$\hat{w}_k^i = w_{k-1}^i p(z_k|x_k^i)$$

$$w_k^i = \frac{\hat{w}_k^i}{\sum_{j=1}^{n} (w_k^j)^2}$$

At this point we **resample** again the particles.

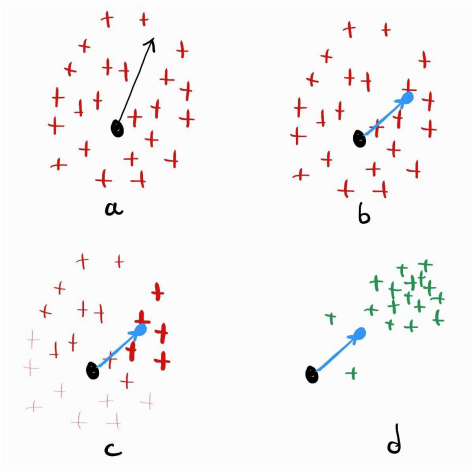


Figure 4.11: Representation of the model update.

a. At the beginning we have no idea what kind of behavior the object will have, it might be the black arrow, but we don't know, so we draw a set of samples in pretty much every direction and these samples will have the same weight at the very beginning.

b. We observe a movement (blue arrow) and we take a measurement to know what are the particles that better approximate the object movement.

c. Some of them are close, some are useless, we assign a weight to each particle based on the goodness of the approximation.

d. Now instead of drawing new particles like in step *a*) we draw more points in a certain direction, this will lead us to be more accurate, taking into account non-linear and non-gaussian behaviors but still keeping in the set of points some "outliers" in case the object decides to move in a different direction.

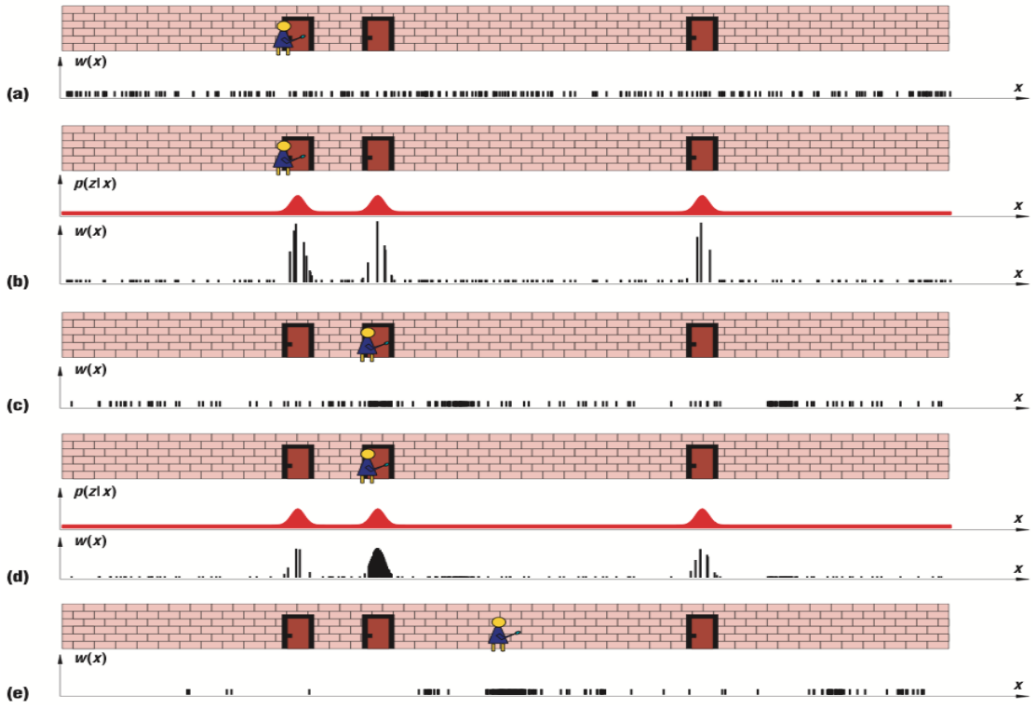


Figure 4.12: The PF solution to the toy problem we have previously seen for the Bayesian Tracking. Just as before we are provided with a door sensing device that will start ringing as we approach a door.

(a) At the beginning we select a certain distribution of the point which to us are potential solutions to the problem, but we have no idea where the guy can be. Any sample could potentially be a solution.

(b) By the time the device rings it means we are close to one among the 3 doors. What happens is that instate of creating a continuous distribution, we assign a higher priority to the samples that are close to the doors, graphically we are increasing the bars of the points in proximity to the doors, and we assign less weight to the samples outside the range of the doors.

(c) Given that we have assigned higher probability to these points, in the next step we draw more samples in the areas of higher probability, but before doing that, we need to propagate the information over time. As a result, we will have high density areas shifted from the peaks computed at step (b) according to our model. The fact that we still have some sparse samples in the areas of low probability makes the algorithm more robust, as we would be able to "catch" the person in case he decided to move in a different direction compared to our prediction, in case this happened the redrawing of the samples would be done accordingly.

(d) At a certain point he approaches the second door. Now we have again evidence that the guy is at one of the three locations, as a consequence we increase the probabilities of the samples in proximity of the doors.

(e) In this step we redraw the particles giving higher priority to the locations where we collected some evidence and we propagate according to the model.

Chapter 5

Geometry

Pixels in the image plane have a direct mapping to the position of the camera in the real world. When we use the approximation of the image we might lose some relevant pieces of information such as the depth of the objects in the scene or the speed of objects, which will appear slower the further they are from the camera.
In this chapter we will introduce the basic concepts of geometry that will allow us to understand the relationship between the camera and the real world.
Typically, we represent a point (pixel) as a set of coordinates:

$$P = [x, y]^t = \begin{bmatrix} x \\ y \end{bmatrix}$$

Often is convenient to use homogeneous coordinates:

$$P = [x, y]^t = [sx, sy, s]$$

Where s is a scaling factor, commonly set to 1.0, this will help us when doing transformations.

$$P = [x, y]^t = [x, y, 1]$$

Sometimes we can also omit the " ^{t} " and write the coordinates as a row vector.

5.1 Affine Transformations

Now formally the pixel is a vector so in general we can apply all the kind of transformations we want, which usually turn out to be combinations of summations and multiplications. The most common transformations are:

- Scaling
- Translation
- Rotation
- Combinations of the above

5.1.1 Scaling

In scaling we take one point or a set of points and we transform them by a multiplying factor.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} c & 0 \\ 0 & c \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

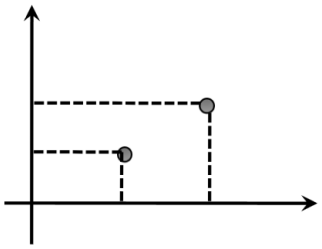


Figure 5.1: Scaling

This is a very simple transformation, it can happen that we have not a single coefficient c but different scaling factors for the different axis.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} c_x & 0 \\ 0 & c_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

5.1.2 Rotation

Rotation is the result of applying a rotation angle to the coordinate points, if we go back to the trigonometry we can see that we can use the sine and cosine information of the angle θ and we can imagine that if we are rotating a unit vector, the coordinates of the new vector will be the sine and cosine of the angle.

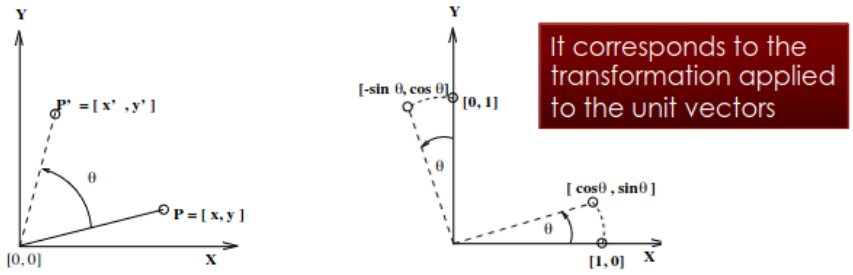


Figure 5.2: Rotation

For instance, we can see on the horizontal unit vector how the x gets reduced while a y component appears corresponding to the sine of the angle.
By the time we have a point that's been shifter by a certain amount θ, saying that the points rotates equals to keeping the point fixed and rotating the coordinate system by the same amount. The resulting operation that we have in our matrix is the multiplication of the point with the transformation of the unit vectors.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

Both scaling and rotation can be constructed using a simple 2 by 2 matrix.

5.1.3 Translation

A translation is simply a shift of the points of our objects into new coordinates. Just like we can think of rotation as a change in the coordinate system, we can think of translation as a change in the origin of the coordinate system.
If we apply a displacement vector it's the same as moving the origin of the coordinate system to the new point and we can model this easily:

$$D([x, y]) = [x + x_0, y + y_0]$$

At this point we can't use anymore a 2 by 2 matrix because we have to introduce the two operators x_0 and y_0 that are not part of the matrix.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

The new coordinates are nothing but a multiplication of a scaling matrix of factor 1.0 by the coordinates x, y and then the addition of the translation vector.
To bring the displacement vector inside the matrix we can use the homogeneous co-

ordinates, we can add a third coordinate to the vector and then we can multiply the matrix by the vector.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_0 \\ y + y_0 \\ 1 \end{bmatrix}$$

5.1.4 Rotation, scaling and translation

Our goal is to be able to map what's happening in the real world to the image plane, we can do this by applying a series of transformations, overall we need to deal with at least 4 parameters:

- One rotation angle (1 parameter)
- One scaling factor (1 parameter)
- A translation vector (2 parameters)

$$^{w}P_j = D_{x0,y0} S_s R_\theta \, ^{i}P_i$$

We can describe this combination of transformations with the formula above, where $D_{x0,y0}$ is the translation matrix, S_s is the scaling matrix, R_θ is the rotation matrix, w is the world coordinates, i is the image coordinates and j a generic point.

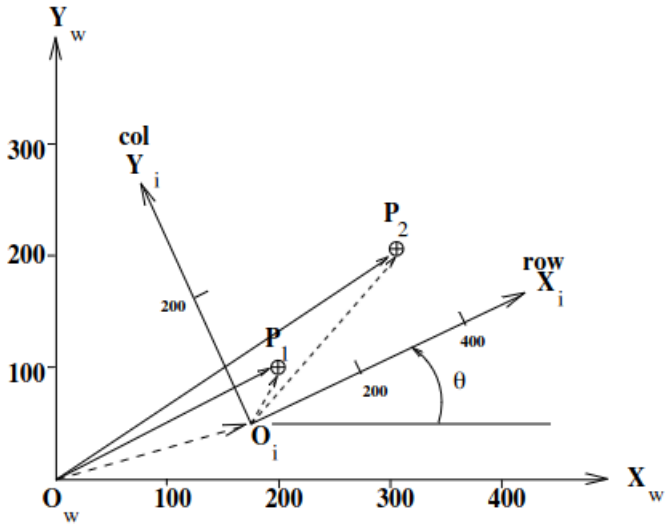


Figure 5.3: Here we are applying a translation, a rotation and a scaling to the point P

We can write the formula above as a matrix multiplication:

$$\begin{bmatrix} x_w \\ y_w \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

So to obtain the transformation matrix we need to solve a system of equations with 4 unknowns, the 4 parameters we mentioned before. To obtain 4 equations we can use 2 points called control points (obtaining x_1, y_1 and x_2, y_2), such points must be clearly visible both in the image and in the real world planes. Once we have found these two points we are able to map these coordinates on the image plane.
We keep referring to the transformations in Figure 5.3 as $2D \rightarrow 2D$ transformations, this implies we have a ground plane on which we are working on and a camera plane, the camera plane is the image plane that might be scaled, translated with respect to the ground plane. At the moment we are not considering a rotation of the camera, we are assuming the image plane is parallel to the ground plane.

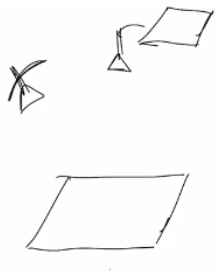


Figure 5.4: The camera plane we are considering now is parallel to the ground plane

5.1.5 General Affine Transformations

Starting from the previous transformations we can see that the 3 matrices are of the same size and if we compute the product between all of them we can think of these transformations as an end-to-end transformation where we can embed all these parameters in a single matrix.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This transformation is the set of coefficients, we don't know exactly the contribution of each of the 3 matrices, but it results in a certain transformation which is represented by what is called in general the **Camera Matrix**. It tells us exactly how to move from

a certain plane where x, y lie to where they will be in the camera plane, and vice versa. We see that we have 6 coefficients instead of the 4 parameters, to determine them it's the same as we did before, with the exception that instead of 2 we need 3 matching control points for which we know the position of in the real world and in the image plane. However, finding these points might now be trivial, the main difficulty is being precise in picking them.
What is usually done is, instead of using the minimum required control points, picking up more points with the hope that on average we are making a very small mistake, averaging out the errors. This way we might end up with 12, 20 equations and 6 unknowns (an *over-determined system*), our goal is to find the 6 equations that minimize the error. We use the *least-squares method* to find the best solution that minimizes the error.

$$\varepsilon(a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}) = \sum_{j=1}^{n} (a_{11}x_j + a_{12}y_j + a_{13} - u_j)^2 + (a_{21}x_j + a_{22}y_j + a_{23} - v_j)^2$$

What we are trying to get is to find the best configuration of these coefficients in a way that the difference between what's expected from the transformation and what's observed is minimized. The resulting equation system is:

$$\begin{bmatrix} \sum x_j^2 & \sum x_j y_j & \sum x_j & 0 & 0 & 0 \\ \sum x_j y_j & \sum y_j^2 & \sum y_j & 0 & 0 & 0 \\ 0 & 0 & 0 & \sum x_j^2 & \sum x_j y_j & \sum x_j \\ 0 & 0 & 0 & \sum x_j y_j & \sum y_j^2 & \sum y_j \\ 0 & 0 & 0 & \sum x_j & \sum y_j & \sum 1 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \end{bmatrix} = \begin{bmatrix} \sum x_j u_j \\ \sum y_j u_j \\ \sum u_j \\ \sum x_j v_j \\ \sum y_j v_j \\ \sum v_j \end{bmatrix}$$

The solution is found by computing the minimum of the error, or in other terms compute the *partial derivative* with respect to each unknown and set them to zero, making sure the error between the result of the transformation and the observed points is minimized.

And this is how the whole thing works, how we determine the **2D camera matrix** used to map the real world plane to the camera plane. Remember that we are talking about two different planes one on top of the other, we don't have full 3D rotations *yet*.

5.2 Going 3D

We are now moving to more general solutions, we can't expect from the system we'll take in consideration to end up in the easy scenario where the two planes are parallel

and facing each other. More in general what happens is that the image plane we are dealing with is something that results from a generic camera prospective where what we see in the image plane is something that look more like shown in Figure 5.5.

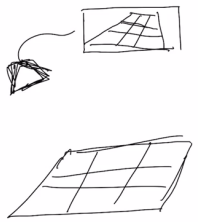


Figure 5.5: A generic 3D scenario.

We need to go through a process called **calibration** where we'll need to determine the so called **intrinsic** and **extrinsic** parameters of the camera. The projections that we have seen so far won't be enough to describe the 3D world, we'll need to add an *additional view* to make possible to determine the unique 3D coordinates XYZ that are necessary to describe the position of a point in the real world.
With that additional information we can do many tasks:

- Point Clouds
- Structure From Motion
- Mesh Reconstruction

5.2.1 Intrinsic and Extrinsic Parameters

- *Intrinsic parameters* refer to the specific characteristics of the camera, such as the focal length, the distortion of the lens, the position of the principal point, etc. These parameters are fixed and don't change with the position of the camera. They are necessary because we want to link the pixel coordinates with the corresponding coordinates in the camera coordinates system.
- *Extrinsic parameters* refer to the position of the camera in the real world, such as the rotation and translation of the camera with respect to the world coordinates. These parameters are not fixed and change with the position of the camera.

When we try to estimate these parameters we go through a process called **calibration**, where we try to find a suitable matrix which helps us to map the points in the real world to what we see through the camera.

5.2.2 3D Affine Transformations

At this point we need to go through the transformations that we have seen in the 2D case, extending them by adding a new dimension. The main difference is that our starting point is a set of 3D coordinates.

$$[P_x, P_y, P_z] \rightarrow [sP_x, sP_y, sP_z, s]$$

Also in this case we move from a 3D vector to a 4D vector with the addition of the scaling factor of the homogeneous coordinates.
Now we have a system for each camera so a triplet of coordinates for Camera 1 and a triplet of coordinates for Camera 2, these two cameras are looking at the world coordinates, in some cases it could be interesting to also know the coordinates of the model in the world (although it's not common).
We end up with:

- Model coordinates
- World coordinates
- Camera 1 coordinates
- Camera 2 coordinates

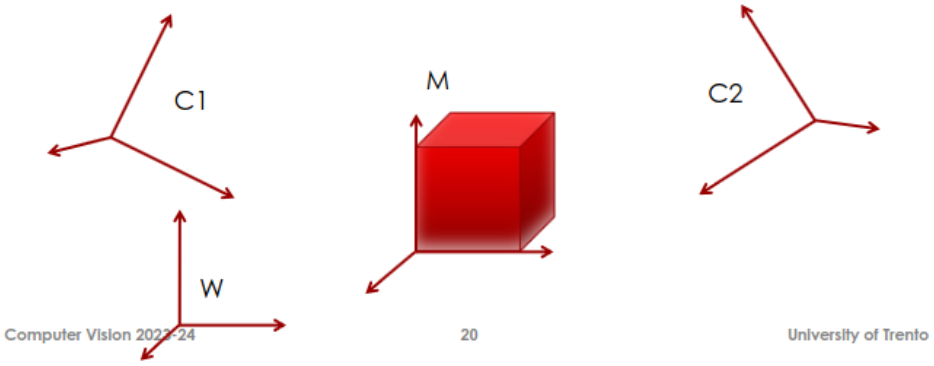


Figure 5.6: The new model with two camera views.

We can see all these 4 systems linked by a transformation, which is nothing more than a rotation-translation to go from any of the 3 systems to the others. At the end it means that we need to deal with a rotation matrix and a translation vector. ${}^wP = TR^mP$ and same for ${}^1P$ and ${}^2P$.
Where T is a translation vector and R a rotation matrix. It must be noted that the

point seen from the first camera might be different from the second camera, both in terms of coordinates and in the sense that Camera 1 and Camera 2 might have opposite views, meaning that there might be no one-to-one mapping of all points. If there's a superset of points that's visible from both cameras, we'll be able to reconstruct the 3D position only for the points visible from both cameras.

NB: The notation for the next section will be ${}^kP_j$ where k is the camera number and j is the point number.

5.2.3 3D Translation

Across the two views in order to move the point from a camera to the other we apply a translation, it consists of applying a translation x_0, y_0, z_0 to the point and no scaling.

$$ {}^2P = T(x_0, y_0, z_0){}^1P $$

$$ \begin{bmatrix} {}^2P_x \\ {}^2P_y \\ {}^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1P_x \\ {}^1P_y \\ {}^1P_z \\ 1 \end{bmatrix} $$

5.2.4 3D Scaling

Here, similarly to the previous case, we can have different scaling factors, although usually we have the same scaling factor for all the axis.

$$ {}^2P = S(s_x, s_y, s_z){}^1P $$

$$ \begin{bmatrix} {}^2P_x \\ {}^2P_y \\ {}^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^1P_x \\ {}^1P_y \\ {}^1P_z \\ 1 \end{bmatrix} $$

5.2.5 3D Rotation

In terms of rotation, things become a little more complicated, we don't have a single rotation (in the 2D space we have to deal with a single angle) but since now we have an additional axis we have a rotation around the three axis, if we want to do the same

analysis as we did in the 2D case with the unit vector we end up with three matrices of coefficients that describe the rotation around each of the three axis.

$$^2P = R(X,\Theta)^1P \qquad\qquad ^2P = R(Y,\Theta)^1P$$

$$\begin{bmatrix} ^2P_x \\ ^2P_y \\ ^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ^1P_x \\ ^1P_y \\ ^1P_z \\ 1 \end{bmatrix} \qquad \begin{bmatrix} ^2P_x \\ ^2P_y \\ ^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ^1P_x \\ ^1P_y \\ ^1P_z \\ 1 \end{bmatrix}$$

$$^2P = R(Z,\Theta)^1P$$

$$\begin{bmatrix} ^2P_x \\ ^2P_y \\ ^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ^1P_x \\ ^1P_y \\ ^1P_z \\ 1 \end{bmatrix}$$

5.2.6 3D General Configuration

Our goal is to then put everything together in a single matrix, this is the generic 3D affine transformation that maps a point in the 3D coordinates of the first system to the 3D coordinates of the second system.

$$\begin{bmatrix} ^2P_x \\ ^2P_y \\ ^2P_z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} ^1P_x \\ ^1P_y \\ ^1P_z \\ 1 \end{bmatrix}$$

It's some sort of camera matrix, but at this point we are dealing with something that returns 3D coordinates and unfortunately doesn't help us much because at the end what we have is an image plane where the coordinates are only rows and columns, x, y. It means that we need to further modify this matrix in order to end up with a representation that's 2D. We need to add to our transformation what are the proprieties of the camera, this gives us the opportunity to move from the 3D coordinates (in our case the world W) to a 2D plane I.

$$^IP = {}^I_WC^WP$$

This matrix here flattens the information and returns a 2D vector corresponding to the position in the rows and columns of the image plane and this happens by adopting a 3 by 4 matrix that gives us the chance to move from the 3D world to the 2D camera coordinates.

$$\begin{bmatrix} s^I P_r \\ s^I P_c \\ s \end{bmatrix} = {}^I_W C^W \begin{bmatrix} {}^W P_x \\ {}^W P_y \\ {}^W P_z \\ 1 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & 1 \end{bmatrix} \begin{bmatrix} {}^W P_x \\ {}^W P_y \\ {}^W P_z \\ 1 \end{bmatrix}$$

Unfortunately we are still not happy because while it's true that we have a 2D representation of the 3D world, we are not yet in the image plane, we are still in the camera coordinates (notice the capital Ps in the formulation) and in the continuous space. We need to further modify this matrix to move from the camera coordinates to the image plane.

We need to rework again on these transformations moving from the camera to some additional coordinates:
(While important the following part till the end of the section was covered briefly)

$$\begin{bmatrix} {}^C P_x \\ {}^C P_y \\ {}^C P_z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^W P_x \\ {}^W P_y \\ {}^W P_z \\ 1 \end{bmatrix}$$

$${}^C P = {}^C_W TR(\alpha, \beta, \gamma, t_x, t_y, t_z)^W P$$

${}^C P$ is about the *camera coordinates*, and not the *image coordinates* ${}^F P$, we need to project these on the image coordinates and successively discretize them in the pixel coordinates.

$${}^F P = {}^F_C \Pi(f)^C P$$

$${}^F P = {}^F_C \Pi(f) TR(\alpha, \beta, \gamma, t_x, t_y, t_z)^W P$$

$$\begin{bmatrix} s^F P_r \\ s^F P_c \\ s \end{bmatrix} = \begin{bmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & 1 \end{bmatrix} \begin{bmatrix} {}^W P_x \\ {}^W P_y \\ {}^W P_z \\ 1 \end{bmatrix}$$

The conversion from mm to pixel consists of a scaling factor related to the real size of the pixels where d_x is the size of the pixel in the x direction and d_y is the size of the pixel in the y direction.

$$^{I}P = {}^{I}_{F}S^{F}P$$

$${}^{I}_{F}S^{F} = \begin{bmatrix} 0 & -1/d_y & 0 \\ 1/d_x & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The minus sign is due to the fact that the origin is usually bottom left but in the image it's top left, so we flip the coordinates.
Overall in order to get the position rows and columns on the image plane we need to go from 3D generic world coordinates to the camera coordinates, then we need the intrinsic information of the camera, and then we need to discretize the pixels.

$$[p_r, p_c]^T = {}^{I}P = {}^{I}_{F}S^{F}_{C}\Pi(f)^{C}_{W}TR(\alpha, \beta, \gamma, t_x, t_y, t_z)^{W}P$$

And we finally get out final camera matrix:

$$\begin{bmatrix} s^{I}p_r \\ s^{I}p_c \\ s \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & 1 \end{bmatrix} \begin{bmatrix} {}^{W}P_x \\ {}^{W}P_y \\ {}^{W}P_z \\ 1 \end{bmatrix}$$

At the end the entire iter of transformations that we use to go from the 3D points in space to the pixel coordinates is included in the 3×4 camera matrix. It means that overall we need to compute 11 parameters (we are not considering the scaling parameter) and in order to create a reasonable mapping of the points we need to compute these coefficients. We'll see that the process that we follow to obtain the coefficients if basically the same as the 2D case.

5.3 Calibration

What we did till now was introducing some geometry problems by saying that what we want to do is to find the relationships between different domains: what happens in the real world and what happens on the camera side. During the process of acquisition the relationships between the two domains change through rotations, translations and

scaling. To describe them we came up with a *camera matrix* that helps us to map what happens in the real world with what happens in the camera.

The 3×4 camera matrix contains 12 coefficients, 11 of which are unknown, we need to find them in order to map the points in the real world to the image plane. The process of finding these coefficients is called **calibration**.

In order to determine the 11 unknowns we need 6 matching pairs (12 equations) which are enough to solve the system. To pick them we do the same as in the simpler 2D case, we choose points in the real world for whom we know the coordinates and we choose the same points in the image plane, we need to be very precise in picking these points, so usually we pick a *higher number* of points to average out the errors.

It's common to use an object of known geometry (a calibration patter) for which we know the relative position of the points and for this we compute the matching.

5.3.1 Calibration Procedure

As we have seen before, we have the points in the 3D world, we have the matching 2D projections and we can start constructing our equations system.
Given a 3D point $[{}^{W}P_x\, {}^{W}P_y\, {}^{W}P_z]$ and its projection $[{}^{I}P_r\, {}^{I}P_c] = [uv]$ for each point in the calibration process we can write the system of equations:

$$\begin{bmatrix} x_j & y_j & z_j & 1 & 0 & 0 & 0 & 0 & -x_j u_j & -y_j u_j & -z_j u_j \\ 0 & 0 & 0 & 0 & x_j & y_j & z_j & 1 & -x_j v_j & -y_j v_j & -z_j v_j \end{bmatrix} \begin{bmatrix} c_{11} \\ c_{12} \\ c_{13} \\ c_{14} \\ c_{21} \\ c_{22} \\ c_{23} \\ c_{24} \\ c_{31} \\ c_{32} \\ c_{33} \end{bmatrix} = \begin{bmatrix} u_j \\ v_j \end{bmatrix}$$

As usual our goal is to rely on the least square to find the configuration of the matrix the minimizes the error between the observed points and the points that we expect from the transformation, the error between the actual image point measurements and the world points comes from.

$$^{I}P = {}^{I}_{W}C\, {}^{W}P$$

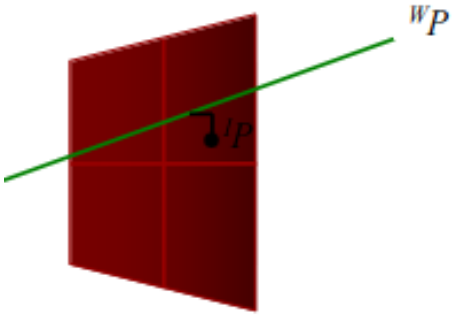


Figure 5.7: Error in the projection.

In an ideal world all the points coming from the real world all the points are being projected in the center of projection, we can trace the ray through the camera plane and that's it. In the real world we make mistakes, and as shown in Figure 5.7 the points are not exactly where we expect them to be, the error is the difference between the expected and the observed points, what we are trying to do is to minimize this error.

5.3.2 Computing the 3D position of a point

Now we are in the situation where we can map points in the real world on the image plane and can get closer to one of our biggest issues: with just a single camera we cannot compute the full 3D coordinates of a point. If we add an additional camera we can have two different systems, at that point we have 4 equations (2 for each camera) and if we know the camera matrices for each camera we can come up with the 3D position of the point.
Given a generic point $[x, y, z]$ and given two projections $[r_1, c_1]$ and $[r_2, c_2]$ we can write:

$$\begin{bmatrix} sr_1 \\ sc_1 \\ s \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad \begin{bmatrix} tr_2 \\ tc_2 \\ t \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

To compute and reconstruct the 3D position of the point we can solve the system of equations and find a solution.

If we assume that we have the projections, we know that they are subject to a certain error. That error is caused by an approximation of the projection and can be seen in the real world as a discrepancy in the intersection of the two rays.

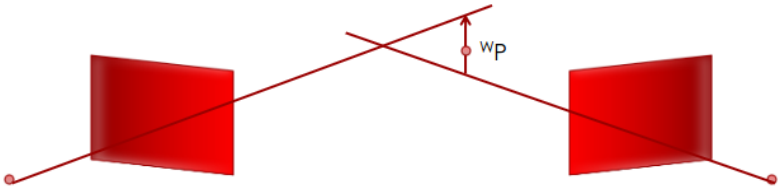


Figure 5.8: Error in the intersection.

Usually what's done is to take the distance between the two lines and use the middle point as the 3D position of the point.

5.4 The Binocular Stereo

As the name suggests it's basically a system that looks like Figure 5.8 where we have two cameras and we try to come up with few equations to model that situation which, by no coincidence, is the same system humans are equipped with. It can be scaled up with multiple cameras, but the idea is the same as it becomes a pairwise binocular system anyways.
The computation of the 3D position of a point usually goes through 2 steps:

- Computation of the correspondences (main source of error)
- Reconstruction of the 3D position (basically deterministic)

The conditions for a binocular system is that overall we have two cameras positioned anywhere in the real world pointing at a scene where there's an area that overlaps. From the area that's visible by both camera we can compute the reconstruction.

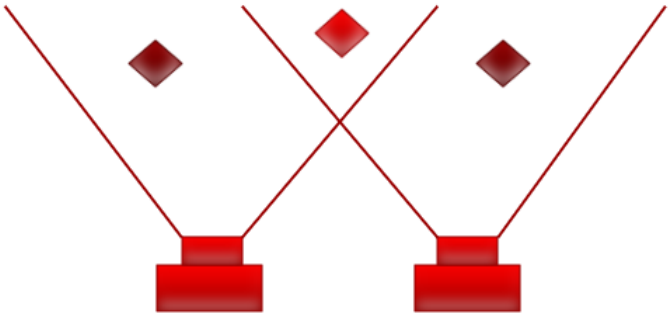


Figure 5.9: This is also a specific case where both cameras are parallel and aligned (coplanar) as in most commercial stereo systems.

Figure 5.10: Coplanar views.

In the case of coplanar cameras the difference between the views will be just a small offset as shown in Figure 5.10. Points will result shifted, and the shift will depend on the depth of the point, the further the point the smaller the shift. On top of that from a single image it's impossible to correctly estimate speed, further points will appear to move slower compared to closer points even if the real speed is the same, this is called **parallax**. Using a stereo rig is quite more informative, we can do the matching in terms of real coordinates, and we can compute the true distance covered.

5.4.1 Computing correspondences

The first step is to compute the correspondences, in order to do so we need to find those points that are representing the same portion of the real world and this is possible mostly because if we have a good acquisition system the distance is not too big, it's normal to look for a certain match in the local area around the point of interest.

We can rely also on the **Epipolar Constraint** which tells us that the correspondences can be met along a horizontal line called the *epipolar line*.

5.4.2 Stereo Vision and Epipolar Geometry

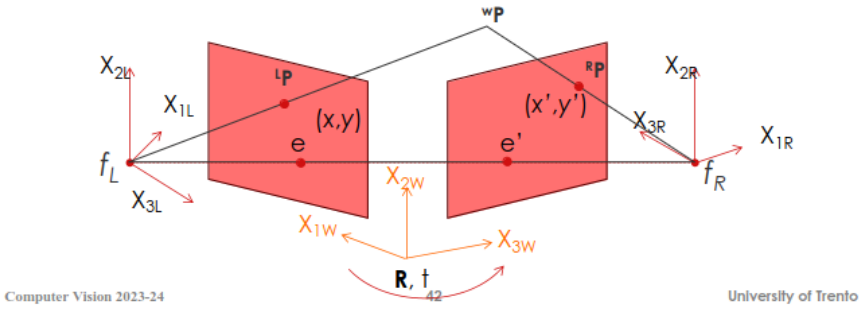


Figure 5.11: A generic stereo system.

This is a slightly more complex system compared to the coplanar one. We have a Left and a Right camera, both will have their camera coordinates system and we also have to coordinates of the real world. Our objective is to start from a point available in the real world, look where the point is being projected in the first and second camera and use the information of this projection to infer the coordinates of ${}^{W}P$.

$$\begin{bmatrix} x \\ y \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} \rightarrow \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

We need to go through the **Epipolar Geometry**.

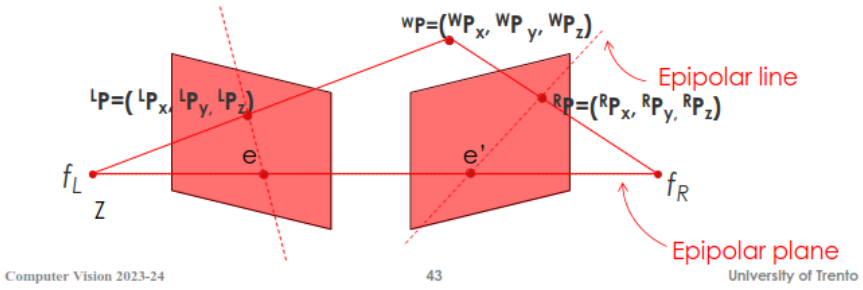


Figure 5.12: Epipolar geometry.

The **epipolar plane** is what connects the ${}^{W}P$ with the two cameras, this plane can be seen as something anchored to axis f_L and f_R, since they don't move, depending on where the point is in the world what chances in the inclination of the plane. The intersection in between the epipolar plane and the image plane is what we call the **epipolar lines**. The points e and e' are the **epipoles** and they are the intersection of the epipolar lines with the image plane, they are useful because, for example, as the ${}^{W}P$ moves along its projection line on f_R (basically the ray that goes from f_R to ${}^{W}P$) its projection on the right image plane will remain the same meanwhile its projection on the left image plane will move exclusively along the epiploar line: we *know* that we only need to look in that area to find the correspondence.

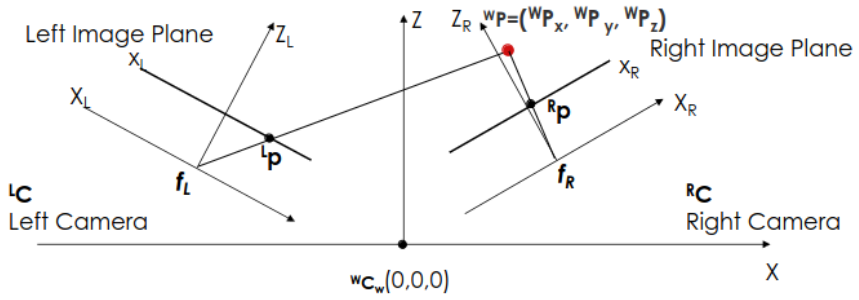


Figure 5.13: The same system from above.

We have

$$^{R}P = (^{R}P_x, ^{R}P_y, ^{R}P_z) \quad ^{L}P = (^{L}P_x, ^{L}P_y, ^{L}P_z) \quad ^{W}P = (^{W}P_x, ^{W}P_y, ^{W}P_z)$$

$$^{R}P = ^{R}R^{W}P + ^{R}T$$
$$^{L}P = ^{L}R^{W}P + ^{L}T$$

We know that the point seen from the camera systems will be the result of a some roto-translations of the point in the real world to the coordinate system of the cameras, R and T are related to the extrinsic parameters of the cameras.

The only common term between the two last equations is ^{W}P, so we can substitute and get:

$$^{L}P = ^{L}R^{R}R^{-1R}P - ^{L}R^{R}R^{-1R}T + ^{L}T$$

$$= M^{R}P + B$$

Where the term M is a matrix that multiplies ^{R}P, plus a term B. This tells us that in between the L and the R we have a roto-translation. Using the simplified perspective projections ($Z >> f$) we obtain the coordinates on the two image planes:

$$^{L}p_x = f\frac{^{L}P_x}{^{L}P_z} \qquad ^{L}p_y = f\frac{^{L}P_y}{^{L}P_z}$$

$$^{R}p_x = f\frac{^{R}P_x}{^{R}P_z} \qquad ^{R}p_y = f\frac{^{R}P_y}{^{R}P_z}$$

from which we obtain

$$\frac{{}^{L}P_z}{f}\begin{bmatrix} {}^{L}p_x \\ {}^{L}p_y \\ f \end{bmatrix} = \frac{{}^{R}P_z}{f} M \begin{bmatrix} {}^{R}p_x \\ {}^{R}p_y \\ f \end{bmatrix} + B$$

5.4.3 Estimation of the 3D position

Some computations are not trivial, so we will see the simplified situation where the cameras are parallel and aligned. The math problem is always the same: we want to get the coordinates of ${}^{W}P$ given the two projections on the two image planes.

$$\boldsymbol{{}^{W}P_z = \frac{fb}{{}^{L}p_x - {}^{R}p_x}}$$

Figure 5.14: Formula to compute the depth of the point. *It's strongly recommended memorizing this formula.*

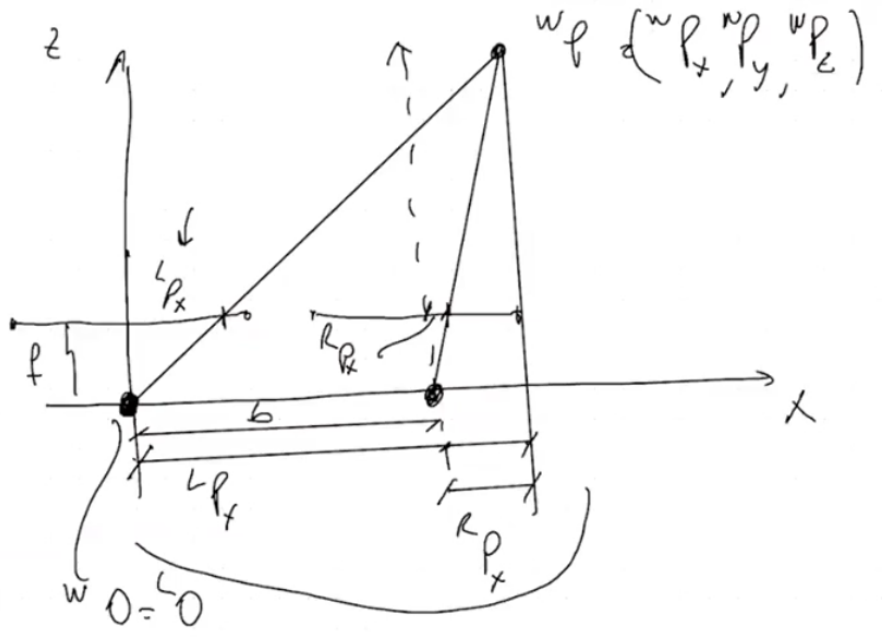


Figure 5.15: Computations to get the depth of a point from the projections.

In Figure 5.15 it's shown our coplanar stereo system with focal length f, ${}^{W}P$ is the point in the real world, the two projections on image planes are ${}^{L}p_x$ and ${}^{R}p_x$ and in order to complete the system we need the ${}^{L}P_x$ and ${}^{R}P_x$ which are the coordinates of the point with respect to the camera systems.

We can say that:

$$^{L}p_x = \frac{f\,^{L}Px}{P_z} \qquad ^{R}p_x = \frac{f\,^{R}Px}{P_z}$$

Figure 5.16: Projection equations.

Note that we are using P_z for both because the cameras are parallel and aligned, so it's the same:

$$^{W}P_z = {}^{L}P_z = {}^{R}P_z$$

We put the origin of the world in the origin of one of the two cameras, this is what happens in real stereo systems. From here we can play around with equations 5.16:

$$^{L}P_x = \frac{^{L}p_x P_z}{f}$$

If the cameras are distant one from the other by a certain amount b we can say (*I guess b is a negative value in this example*):

$$^{L}P_x = b + {}^{R}P_x$$

At this point we use this expression to evaluate the projection on the right camera:

$$^{R}p_x = \frac{f(^{L}P_x - b)}{P_z}$$

We know already what was our $^{L}P_x$, so we can say that:

$$^{R}p_x = \frac{f(^{L}P_x - b)}{P_z} = \frac{f(^{L}p_x Pz)}{fP_z} - \frac{fb}{P_z}$$

$$^{R}p_x = {}^{L}p_x - \frac{fb}{P_z}$$

$$^{L}p_x - {}^{R}p_x = \frac{fb}{P_z}$$

$$\boldsymbol{P_z = \frac{fb}{{}^{L}p_x - {}^{R}p_x}}$$

Finally we have the component P_z that we have been looking for. To get to this point we have many values that play a role:

- b is something that relates to the extrinsic parameters of the camera

- f is one of the intrinsic parameters of the camera
- ${}^{R}p_x$ ${}^{L}p_x$ are the projections of the point in the two image planes. This term is also called the **disparity** and it's the offset between the two projections, this disparity term changes accordingly to the distance of the object from the cameras.

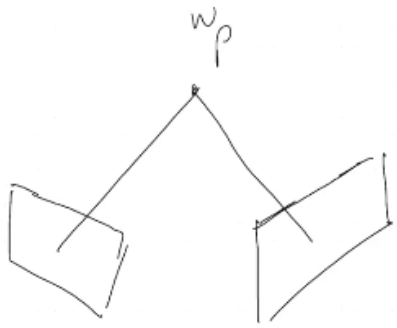


Figure 5.17: Typical exam question: *Please, tell how we compute the 3D coordinates of a point starting from a camera system that is parallel and aligned.* Don't draw this. Just don't.

5.4.4 Matching points

Now that we know how to get the 3 coordinates given the projections we need to understand how to compute the match, and define an evaluation function to understand how good the match is.

The matching is usually done by looking at the intensity of the pixels, a common technique is to use **Window-based approaches**. They are very generic in computer vision, and it goes like this:

- Take a window around the point of interest in the left image
- Along the epipolar line find the windows that best match the right and left image, shifting of a handful of pixels at a time or even pixel by pixel.
- Compute an error function (MSE, SAD, SSD)
- Find the minimum
- Winner-take-all and that's the *disparity*. Usually we define a range of disparities, otherwise we would have to look along the entire epipolar line.

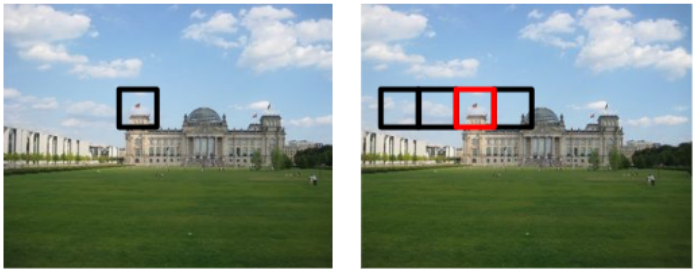


Figure 5.18: Moving the window along the epipolar line.

5.4.5 Image Normalization

Unfortunately the two cameras are not *exactly* the same, and might have small differences in the acquisition in terms of colors, even with an offset of just one or two grayscale values, our metric could become noisy, this why images get **normalized**

$$\bar{I} = \frac{1}{|W(x,y)|} \sum_{(u,v)\in W(x,y)} I(u,v)$$

We can compute the average value of the pixels in the window.

$$\|I\|_{W(x,y)} = \sqrt{\sum_{(u,v)\in W(x,y)} [I(u,v) - \bar{I}]^2}$$

Compute the window magnitude.

$$\hat{I}(x,y) = \frac{I(x,y) - \bar{I}}{\|I - \bar{I}\|_{W(x,y)}}$$

Calculate the normalized values, with this normalization we can eliminate the offset between the two images.

This makes it possible to compute the distance because we now know that what we have on the right side and the left side is the same. Since this normalization makes the windows comparable we can now compute the SAD or the **Sum of Squared Differences**:

$$SSD(x,y,d) = \sum_{(u,v)\in W(x,y)} [\hat{I}_L(u,v) - \hat{I}_R(u-d,v)]^2$$

Another metric is called the **Correlation**, a big error corresponds to small correlation and vice versa. What happens in this case is that instead of comparing the pixel to pixel values, we take the window and represent it as a vector. For instance a 3×3 window will be represented as a 1×9 dimensional vector, we can then compute the correlation between the two vectors.

$$C(d) = \frac{1}{|w - \bar{w}|} \frac{1}{|w' - \bar{w}'|} (w - \bar{w})(w' - \bar{w}')$$

Where d represents the window shift, w and w' are the vectorized windows and $\bar{w}$ and $\bar{w}'$ are the averages of the vectorized windows. Basically it's a comparison between vectors where we want to measure the angle between $w - \bar{w}$ and $w' - \bar{w}'$.

$$C(d) = \sum_{(u,v)\in W(x,y)} [\hat{I}(u,v)\hat{I}(u-d,v)] = w \cdot w' = \cos\theta$$

In the normalized case, the correlation is maximum if the original brightness of the two windows is shifted by an offset and a scale factor. This means that by the time we compute the correlation with have normalized the vectors (for instance between 0 and 1) and that the correlation is maximum when the angle between the two vectors in 0, *however* it can be that the vectors are in fact, shifted and scaled, while maintaining the same angle.

Figure 5.19: Scaled and shifted vectors with maximum correlation.

$$\hat{I} = \lambda I + \mu$$

Computing the correlation at each frame for the whole image can be expensive, in practice we have windows that overlap so one of the tricks in the implementation is to keep track of the parts of the window that are common between the two windows and update the correlation accordingly. In addition, usually the correlation is carried out taking into account the disparity.

5.4.6 General Stereo Configuration

Now we want to move from the simplified stereo rig to a better characterization of a general stereo configuration, we still have our two cameras which are observing an object in the real world, we still have our ${}^{W}P = [{}^{W}P_x, {}^{W}P_y, {}^{W}P_z]$ and then we have two arbitrary cameras positioned somewhere in the real world. Again we want to find the correspondencies between the point ${}^{W}P$ which is seen by the camera C1 in position ${}^{1}P$ and by the camera C2 in position ${}^{2}P$.

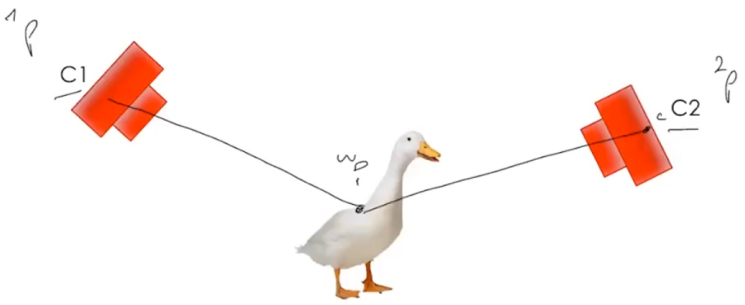


Figure 5.20: Quack.

We know already that the two lines in Figure 5.20 will never match exactly but we want to find the configuration that minimizes the error.

What do we need?

- Position of C1 and some internal parameters such as the focal length, we know that this information is embedded in the camera matrix, which defines the relationship between the points in the real world and the image plane.
- The same stuff for C2.
- The corresponding matching projections of ${}^{W}P$ in the two image planes.
- Finally we compute the 3D position of ${}^{W}P$ starting from the positions of ${}^{1}P$ and ${}^{2}P$.

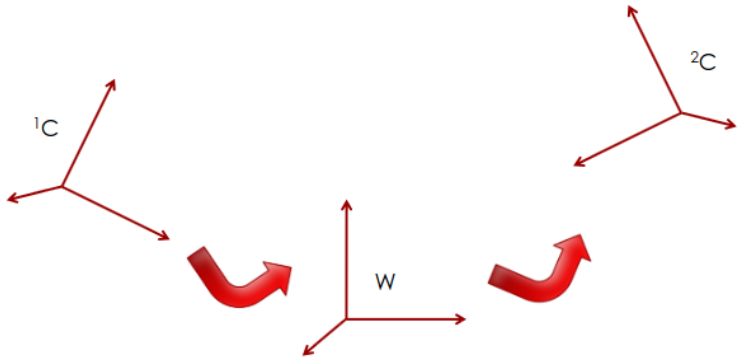


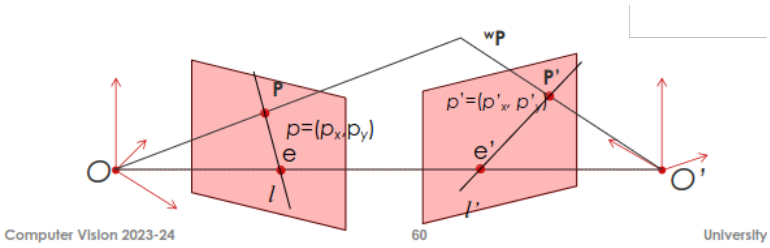
Figure 5.21: The world is the shared information.

5.4.7 The Fundamental Matrix

It's the representation of the epipolar geometry in case of two generic views, it's a 3×3 matrix that can map p into p'

$$p'^T F p = 0$$

We don't need to have the intrinsic parameters, we just need to take a snapshot from the two cameras finding a set of corresponding points, and we can compute the transformation between the points in ${}^1P$ and ${}^2P$ using the fundamental matrix.



But what does it represent?
Let's say for the moment that we know the intrinsic parameters of the camera, this means that everything consists of roto-translations between cameras, we don't need to deal with normalizations, quantizations and distortions due to focal length. In this case if we start from ${}^W P$,in order to match the two views, what we have to find is the transformation that brings the coordinate system of one camera into the other, and this can be seen as a rotation and a translation:

$$Op \cdot [OO' \times O'p'] = 0$$

What we are doing here is taking the vector that connects the two origins of the systems OO' and, by multiplication, bringing the vector $O'p'$ into the other coordinate system. If the dot product between Op and the transformed $O'p'$ is zero, it means that we actually have the match.

$$Op \cdot [OO' \times O'p'] = 0 \quad p \cdot [t \times Rp'] = 0 \quad p = (u, v, 1)^T \quad p' = (u', v', 1)^T \quad p^T [t \times R] p' = 0 \quad p^T E p' = 0$$

Figure 5.22: *"You don't need to understand this"*- cit

After the transformations shown in 5.22 we can map the two points with a single **Essential Matrix** E. This expression says that the matching between the two views is visible through a rotation and a translation of the origins. At this stage the **essential matrix** contains only the extrinsic information because, as we said, the intrinsic parameters are already known.

But we know that we have to deal with also the intrinsic parameters, fortunately it's nothing but a matrix to be added to the system. We can pass the points projected by the essential matrix ($\hat{p}$ and $\hat{p}'$) through the intrinsic parameters (K and K') of the two cameras, and say that the points in the two views are related by the following expression:

$$p = K\hat{p} \quad p' = K'\hat{p}' \quad \rightarrow \qquad F = K^{-T}EK'^{-1}$$

We finally obtain the **fundamental matrix**. The fundamental matrix contains the roto-translation (the extrinsic) and also the instrinsic information. We can say that p and p' correspond as they are different projections of the same point in the two views. So, for each point p in one view, there's a corresponding epipolar line l' in the other image.

$$l' = Fp$$

$$l = F^T p'$$

Now that we have understood that we can go back to this expression, $p'^T Fp = 0$. We use the world as shared information to find the correspondences, but once we have computed the matrix we have found the relationship between the points in the two image planes: we looked at the world to understand where the matches occurred, but now we can "forget" about it because we have a relationship that binds the two cameras.

The fundamental matrix is great because it gives the chance of determining the configuration of a system regardless of what are the coordinates in the real world, it means if we (rigidly) change the coordinates of the cameras in the real world, the same F will hold.

The camera matrices ${}^1M$ and ${}^2M$ are used to determine a *unique* Fundamental Matrix F and while ${}^1M$ and ${}^2M$ are affected by rigid movements in the real world and F is not, on the other hand from a matrix F we can determine the camera matrices only to *up a multiplication factor* of a certain matrix H.

Given a fundamental matrix F for an object it's impossible to determine the absolute position in the world, the orientation and the scale. However, up to a projective transformation, the ambiguity in reconstruction can be solved, the projected points don't change if $MP = (MH^{-1})(HP)$
Where H is a projective transformation that does not affect the projection of ${}^W P$ onto the image plane.

5.5 Homography and friends

The planar homography H is a transformation that helps us make the mapping in between two planes that live in different domain.

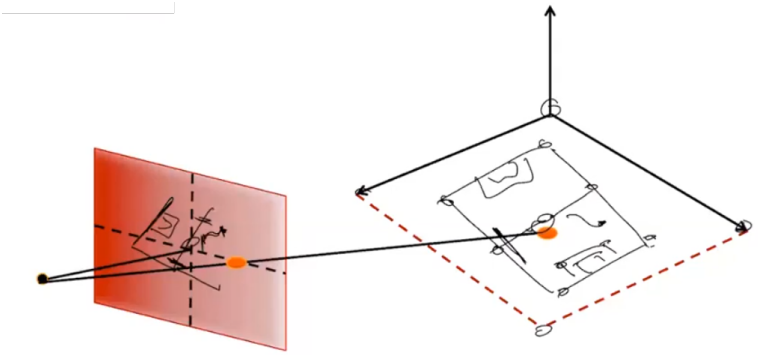


Figure 5.23: The planar homography.

In Figure 5.23 we can see how we are mapping a ground plane on the image plane, they are roto-translated and scaled by certain transformation, but we want to match these two planes in order to understand how things can be transferred from one domain to the other. This is the point where we go back to the real world.
We have the fundamental matrix that deals with the camera configuration and we use the homography information to bind the views with the real world, we are talking about planes so, in the same way as we did before, what we do is to grab some **control points**, easily seen by both planes, such that if we know how this transformation occurs we can determine the correspondence of a displacement in the image plane to the displacement in the real plane. We can know that a displacement of 5px corresponds to a displacement of 3m in the real world thanks to the Homography Matrix H.

This is the basic element in order to determine the trajectory of something that moves on the ground plane, the process is the following:

- Calibrate the cameras in a way that we are rectifying the distortion
- Determine the fundamental matrix, so where two points correspond in one view and in the other
- Take a reference on the real world, once we have that any movement in the image plane can be mapped to the real plane and can be converted in real coordinates.

In principle, we could do this even with a single camera, and we could use the second view to solve problems as occlusion.

5.5.1 2D Homography

It's an invertible transformation between two planes, in fact in our first computation we try to match what is in the real world with what is in the image plane, once we have that we can track the movement in the image plane and we can convert it to the real world. The Homography matrix defined as H is the one that satisfies the following equation:

$$p' = Hp$$

Where p is a point in the world ground plane and p' is the corresponding point in the image plane. Since any vector crossed with itself gives 0 we can rewrite is as:

$$p' \times Hp = 0$$

We rely on this constraint in order to compute the matrix H.

$$p' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \begin{aligned} p' &= Hp \\ p' \times Hp &= 0 \end{aligned}$$

We want to find a linear solution for H, we can rewrite the vector Hp as:

$$Hp = \begin{pmatrix} h^{1T}p \\ h^{2T}p \\ h^{3T}p \end{pmatrix}$$

Now we want to compute the cross product, which is given as follows:

$$\begin{bmatrix} \hat{i} & \hat{j} & \hat{k} \\ x' & y' & w' \\ h^{1T}p & h^{2T}p & h^{3T}p \end{bmatrix} = \begin{bmatrix} y'h^{3T}p - w'h^{2T}p \\ -x'h^{3T}p + w'h^{1T}p \\ x'h^{2T}p - y'h^{1T}p \end{bmatrix}$$

And that's the result of the cross product, we can now rewrite the equation as a function of the components of H:

$$\begin{bmatrix} 0 & -w'p & y'p \\ w'p & 0 & -x'p \\ -y'p & x'p & 0 \end{bmatrix} \begin{bmatrix} h^{1T} \\ h^{2T} \\ h^{3T} \end{bmatrix}$$

At this point we know what's in the first matrix because we know the point in the second plane, we know the coordinates of the point in the first image plane P because we are matching coordinates, now our goal is to find the unknowns for H. Only two

out of the 3 equations are independent, so one can be discarded, what we get in the end is a 2×9 matrix for which we need to determine the coefficients h.

$$\begin{bmatrix} 0 & -w'p & y'p \\ w'p & 0 & -x'p \end{bmatrix} \begin{bmatrix} h^{1T} \\ h^{2T} \\ h^{3T} \end{bmatrix}$$

For each matching point we have two equations, playing around with the number of points that we take we can solve the matrix, 4 points yield to the minimum number of equations to solve the system, but the more points we take the more robust the solution will be. The implementation available in OpenCV relies on the DLT, but otherwise we can use the regular Least-Squares approach.

We managed to set up two cameras, link them to the real world thanks to the Homography matrix in such a way that we are able to determine the position of the object even in the case it's occluded. This is a very effective solution, we are only looking at the ground plane, not at the depth map, these solutions tend to be more robust and less noisy. It depends of course on the specific problem if it's needed or not to get the entire information about the scene depth.

5.5.2 Multiple view geometry

With multiple, we are referring to more than two. So far we have seen stereo systems with two cameras but of course we can generalize to an arbitrary number of views, this might be required in the case we need, for instance, a detailed point cloud. In this case we would end up in the area of so-called **voxels** with respect to pixels, small volume elements instead of screen elements.

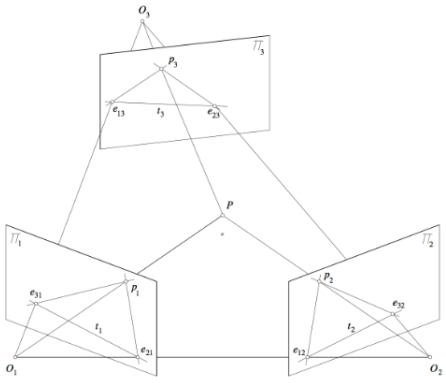


Figure 5.24: Even in this case we can scale down to a two camera problem.

In Figure 5.24 can see how the point ${}^{W}P$ is being observed by the 3 cameras with projections P_1, P_2, P_3. However, we know that these cameras can be linked together with epipolar geometry, provided some parameters of the cameras we should be able to determine the position of ${}^{W}P$ even with just two of them, and the same can be said for any two cameras. This system can be constructed as a combination of pairs of cameras, in this way the geometry constraints are fullfilled and using the essential matrix (or the fundamental matrix if we consider the intrinsic parameters) we can determine the position of the point in the real world and define the epipolar constraint as:

$$p_1^T E_{12} p_2 = 0$$

$$p_2^T E_{23} p_3 = 0$$

$$p_3^T E_{31} p_1 = 0$$

At the end we can use any of the two equations to determine the position of the point in the real world, for instance we could use Camera 1 and Camera 2 in the case Camera 3 is occluded, sharing all these elements make it possible to come up with a very robust tracker. All this scales up to what we call the **problem of transfer**: make the network communicate in a way that the information is actually shared between the different cameras.

Chapter 6

Local Feature Extraction

In this chapter we will provide you a couple of additional feature sets that are commonly used in computer vision applications. More specifically, we are talking about the local features extraction modules. First, we define it *local* because the features points are extracted not at an object level, but from little patches. So we are not considering the whole object, but only a small part of it that can help to characterize the object itself such as intensity and direction of edges.
In particular, we will focus on two different algorithms: HOG and SIFT.

6.1 HOG

The first algorithm we are going to introduce is the Histogram of Oriented Gradients. We already know that gradient is what characterizes the edges of an image, where the edge means that we are dealing with the transition from a gray level to another one. In the Histogram of Oriented Gradients we can easily recognize the object in the image by looking at the vectors. The stronger the gradient, the brighter the arrow will be. While where the edges are weaker the arrow will be less intense.

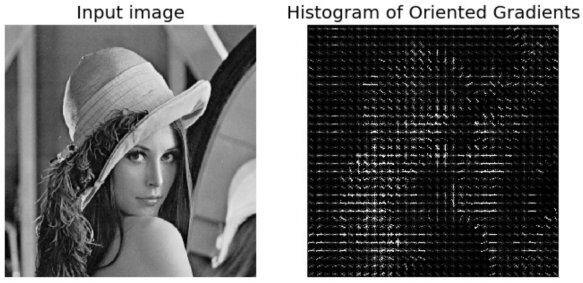
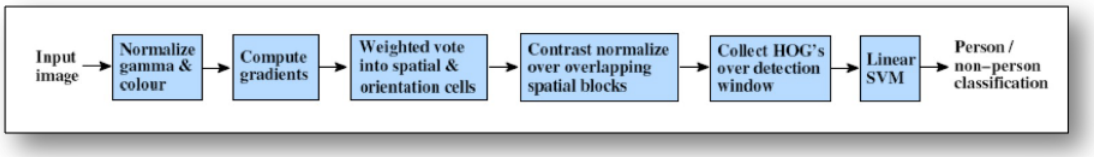


Figure 6.1: Vector representation of the gradients in an image.

So basically, what we are doing inside the picture is to divide it into small patches and look at the gradient of each pixel in the patch. Then we characterize those gradients using a direction information and a magnitude or intensity. At the end of the process we can construct a descriptor which can be used by next algorithms to recognize the presence of object in the image.
But how do we get the descriptor? First of all, we need to divide the image into a number of rectangular or radial cells of arbitrary size.
NB: It's good practice to have only one main edge component inside a cell, so small cells are preferable.
Then, for each pixel in the cell we compute the orientation of edge(*Take a pixel, take his neighbor and see if there is an edge or not*), which gets subsequently quantized and put in one of the histogram's bins. This means that each pixel within the cell can cast a vote, weighted by the gradient magnitude, for an orientation histogram based on the values found in the gradient computation. In other words, each cell is represented through a 1D array (that corresponds to the bins of the histogram) that describes the gradient direction of the cell. In addition, intensity is locally normalized in the RGB or LAB color space using L1 or L2-norm, to account for illumination changes and shadowing especially when using larger areas (blocks), consisting of more cells.
Just to wrap up, let's take a look to the pipeline:



It basically consists of taking the input image, computing the gradients and then the voting for each pixel, applying some additional normalization and finally getting the descriptor.

But what does it mean having a descriptor? It means that, for an object, we have a full description of its shape, obtained by concatenating all the cells, where each cell represents the gradient information within a small patch of the image.

Once we have our descriptor, we can pass it to a classifier, such as an SVM as you can see from the pipeline, to perform tasks like human detection.
Since this is not a machine learning course, we will just take a quick look at the **Dalal and Triggs algorithm**, a popular method for human detection available in OpenCV:

- Arrange the data in a 64×128 window;
- Divide the window into 8×8 pixel cells;

- Blocks are formed by grouping 2×2 cells;
- Each cell has a histogram of 9 directions;
- The final descriptor is formed by concatenating the feature elements from these cells in each block, so a 36D vector.
- Blocks overlap to allow a smoother transition across the different blocks. In a 64×128 window we can fit 7×15 blocks.

This configuration was chosen based on performance tests that measured the miss rate. As depicted in Figure 6.2, this specific setup achieved the best results. Therefore, when implementing similar algorithms, we suggest to use overlapping blocks with a relatively high number of orientation bins and a moderate size of the elements you use.

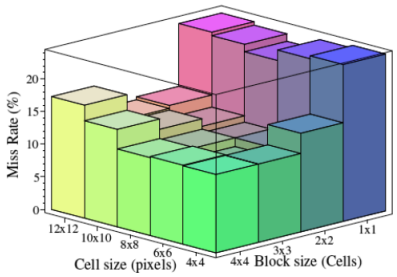


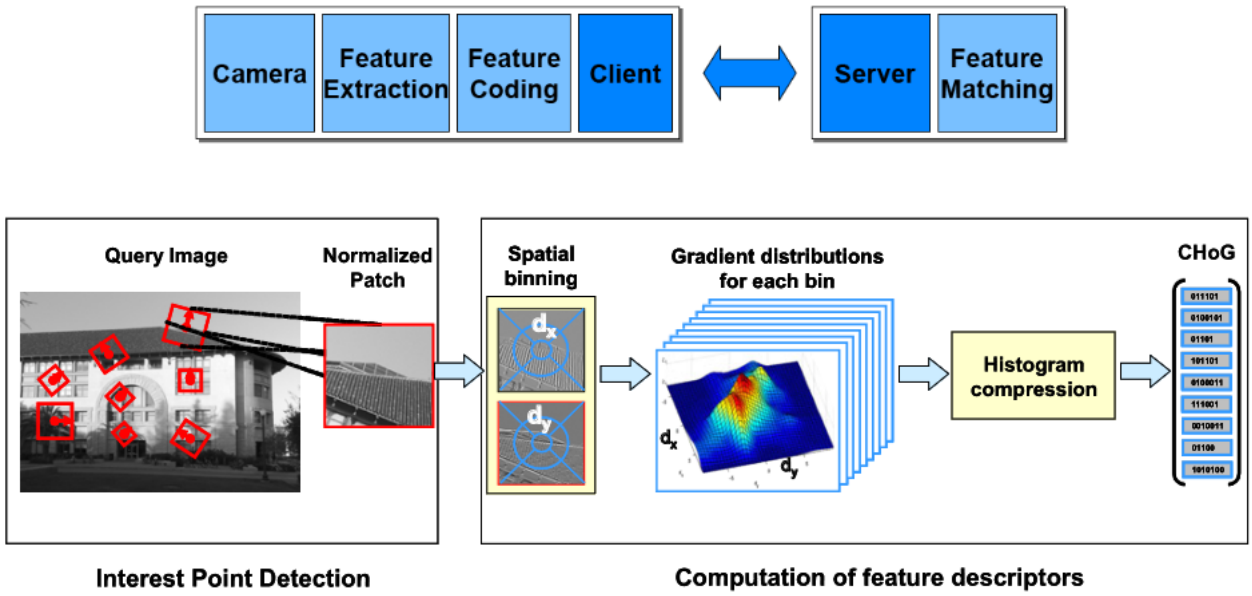
Figure 6.2: Performance of the HOG algorithm with different configurations.

6.1.1 The problem of scale

The problem of scale is a common issue even in current classifiers. The fact is that in video the pedestrian can be detected at different sizes. This can be due to the spacial resolution of the camera, the distance from the object and even the object itself. So, how can we solve this problem? A possible solution is to use a multiscale approach. This means that we need to resize the window at different scales and then apply the algorithm on all different sizes.

6.1.2 Feature compression

The last thing we want to mention is the feature compression. The descriptor can be very large, so we need to compress it in order to reduce the computational cost for delivering.
NB: To compute the compression in a lossless way, we can extract the histograms.
Let's explain a little bit.
We compute the histograms and quantize them, so at this point we have just a few numbers that represent the descriptor. We (client) send them to the server that has a huge database filled up with the information of blocks, objects, songs, etc. The server



takes the features and compares them with the database, then it returns the result. So, in terms of computation, what we are doing is called spatial binning.

6.2 SIFT

The Scale Invariant Feature Transform is an algorithm that extract the salient points of an image, points that exhibit certain proprieties. The Good Features to Track algorithm just extracts corners, and corners by themselves are not enough to *describe* an object, in SIFT case we want to use these features in order to come up with objects' representations made up from a concatenation of features. The main idea is to find the key-points that are **invariant to scale**, rotation, illumination and viewpoint changes. The algorithm is composed by four main steps:

- construct a subspace representation of the image by progressively apply a Gaussian(low pass) smoothing filter;
- at every iteration, each image becomes a blurred version of the previous one. This because, since we are looking for *strong* points, a strong point should survive blurring operations;
- find key-points;
- compute the descriptor.

The filtering is obtained by applying a regular Gaussian filter to the image:

$$L(x, y, \sigma) = I(x, y) * \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where $I(x, y)$ is the image, $L(x, y, \sigma)$ is the scale space of the image after convolution and σ is the standard deviation of the Gaussian filter (strength of the filter).
NB: A large sigma means a strong filter, which will tend to average more points.

The next step is a procedure called the **creation of octaves**. An octave is a set of images of the same size where each image is a blurred version of the previous one. The next octave is obtained by downsampling the original image by a factor of 2, we can crate octaves iteratively as long as the image can be downsampled.

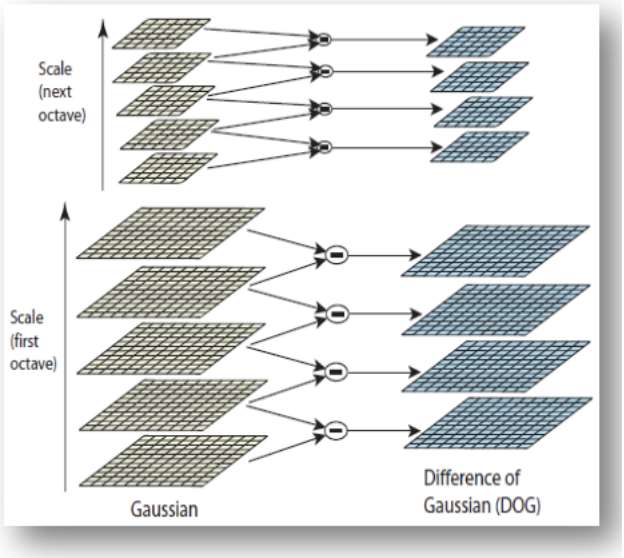


Figure 6.3: Difference of Gaussian

Having the filtered version of the image, we can create the Difference of Gaussian (DoG) by subtracting, for each pair, the blurred image from the previous one as shown in Figure 6.3. More generally, we are trying to *highlight what survives* subtraction after subtraction, so we are looking for the maximums and minimums of the DoG.

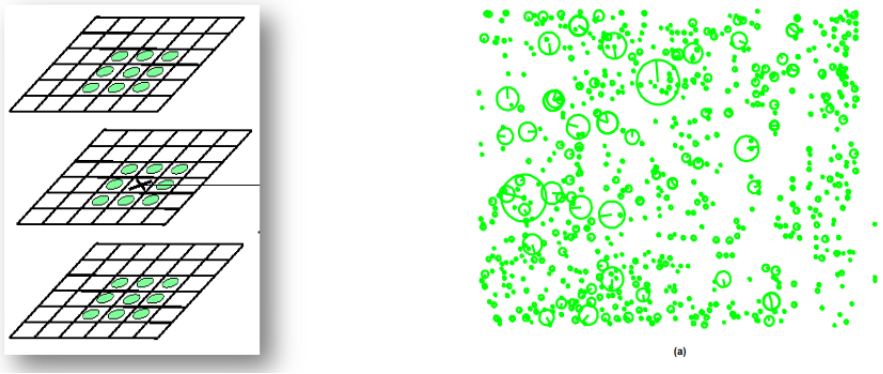


Figure 6.4: Selection of pixels among the DoGs. The points in the right image correspond to the selected candidates. Magnitude and direction are already computed and represented as circles with an arrow although up to this point we are not interested in them.

To grab the most salient points from the DoGs, each pixel from a given DOG is compared to its 26 neighbors (8 in the same level, 9 above and 9 below) as shown in Figure 6.4. A pixel is kept as a key-point only if it is greater (maximum) or smaller (minimum) than all its neighbors.

DoG function exhibits strong response to edges, but edges are not always good key-points, the selection of the points might be noisy, so we can apply a second order Taylor expansion with the derivative set to 0.

$$D(X) = D + \frac{\partial D^T}{\partial X}X + \frac{1}{2}X^T\frac{\partial^2 D}{\partial X^2}X$$

Through this derivative we can obtain the maximum and minimum and construct a Hessian matrix H. The Hessian matrix is the one that tells us if the point that we have found are actually good or not. Having it we can construct the trace and the determinant of the matrix to define a threshold.

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix}$$

What we can check at this point is the ratio between the determinant and the trace of the matrix. If the ratio is greater than a certain threshold, we can consider the point as a key-point. So now consider $r = \frac{\alpha}{\beta}$, where α and β are the largest and smallest eigenvalues in magnitude of the Hessian matrix. If

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r+1)^2}{r}$$

is satisfied, then the point is a key-point.

At his point we can also check the magnitude and the orientation of the gradient, this because the eigenvalues of H are proportional to the principal curvatures of the function D, so if the curvature is high in both dimensions, the gradient and orientation are computed. This step achieves the invariance to rotation:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$
$$\theta(x,y) = \arctan\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}\right)$$

The region within which m and θ are computed is relative to the scale of the key-point, the higher the scale, the larger the computation region.
NB: This is very helpful for us to match the points across different images with different scale and resolution.

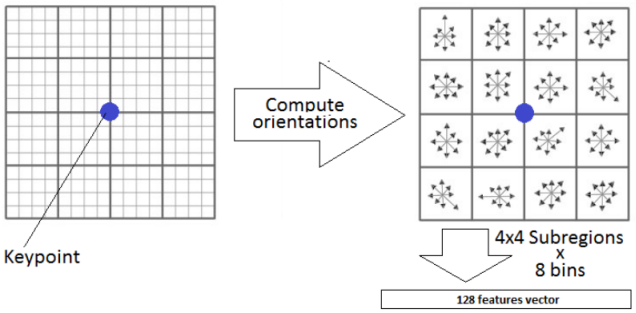


Figure 6.5: Computation of the final descriptor.

An orientation histogram is formed from the gradient orientations of sample points within the region around the key-point. The orientation histogram has 36 bins covering the 360 degree range of orientations. If a key-point exhibits multiple peaks in the histogram that are higher than 80% of the highest peak, a new key-point is instantiated with same location and scale.

The 16x16 area (oriented according to the key-point orientation computed before) around the selected key-point is divided in 4×4 regions, and for each of them a histogram of the gradients (8bins) is computed. These 8 bins histograms are concatenated, and finally it turns out as a **descriptor of 128 elements** $(4 \times 4 \times 8)$.

Once that the descriptor is constructed, we can use it to search and match the key-points in different images. This is done usually by Nearest Neighbor searching, where the Euclidean distance between the descriptors is computed, or by using optimization algorithms like RANSAC.

Chapter 7

Classification

Let's start this chapter saying what means implementing a classifier. This matches a variety of different pattern recognition problems that requires you to take decisions about what you are actually seeing in a picture. So basically, it is when you have to determine what the object present in the picture represents. And, in order to do it, you need to use the features of the object's appearance and determine the category it belongs to. Also, in the previous chapter, we saw how pedestrian detector was working at that level. We just focus mostly on the feature extraction part and not really too much about the classifier itself, but what we did for the history of gradients was exactly to determine what were the silent elements that could characterize the shape of a human body. But there are tons of scenarios in which things like that are useful, some examples are:

- In a supermarket implement a system that recognizes vegetables;
- At the gate of your house, restrict the access to specific subjects or cars;
- At the entrance of a parking lot determine the category of the entering vehicle;
- On a robot, look for objects (doors, humans, obstacles).

The first element that we need to understand is actually what is a class. In a general representation a class is nothing but a set of properties, which can consist in colors, textures, patterns ...
NB: Objects or some parts of objects can be overall represented under the same umbrella over class.
A class is typically made up of a set of descriptions provided by observing numerous examples and classes contain items that share similar properties. Basically, the goal of a classifier is to take an object as input and to output the class label it belongs to.

So, we typically start from an input image and it can be taken as a whole or it can be segmented in advance.

For example, if we want to detect the pedestrians in a scene, it could be an option to run a detector in a way that we can isolate the portion of the screen where things are moving, and we are expecting that pedestrian are moving. So we can isolate some segment, so areas, which are potentially good for being a pedestrian (just a simple example). Or again, we can do a segmentation of the picture filtering out the colors that do not mach the range of color of the skin.

To sum up, segmentation is an option, a step that can be conducted and otherwise we're proceeding with one of the most relevant steps in the classification, which is the extraction of features.
Extraction of features means that we're looking to the history of the gradients which are being located in a certain portion of the segment of the picture itself. Basically, we compare the picture vector with the feature vector that we used for the training phase, where the feature vector is the one that describes what we have extracted from the picture. So we feed this information into the machine that will start reasoning about what it sees as an input and in the end it would return an output. But here we are doing a process called recognition, which is an additional step forward to the detection.

7.1 The classification process

Just to wrap up a little bit, The classification process can be summarized in 4 steps:

- Segment the image if necessary;
- Extract the features;
- Use the feature vector to feed a classifier;
- Output a label.

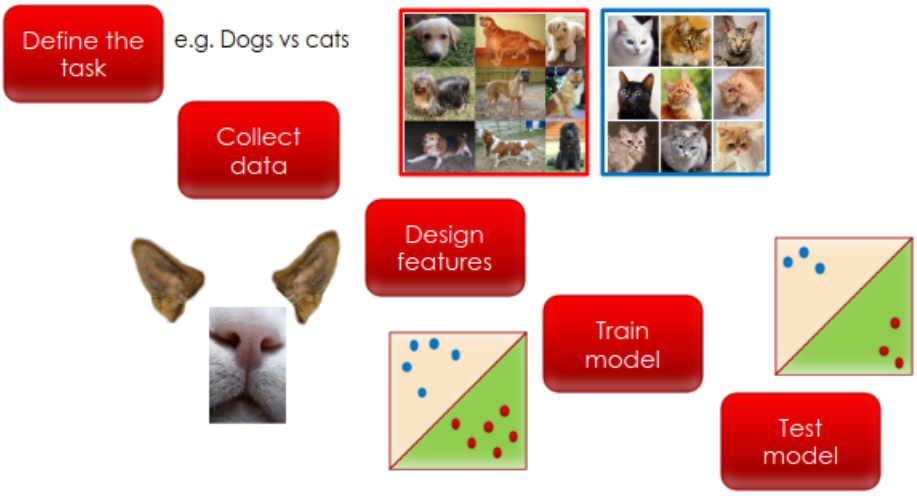
In order to make the concept simple let's take as example children. They learn new things by examples, so, to distinguish a dog from a cat they rely on observing samples of these two animals. We realize they have learned the concept by the time they see a new sample and they're able to recognize it. Essentially, teaching a computer works in the same way.

Without entering the realm of NN, let's take a generic recognition system. It requires a high number of examples in order to be able to distinguish across different classes. Indeed, the creation of a dataset is the tougher job, moreover the dataset need to be annotated (labelled).

7.1.1 Subject to failure

Machines may take days or weeks in order to learn concepts. However, they still tend to fail because of lack of data, wrong annotations, occlusions, perspective, similarity among classes ...

Systems typically are not able to generalize, so for example, you can have classifiers that are mounted on cars which are very robust in detecting cars, but only from a precise perspective.
But what is the pipeline overall over classifier?

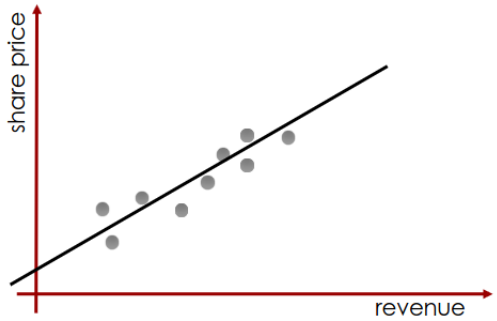


Once the model is trained and has found the boundary that separates the two classes, if we put a cat image it falls on the right part, while if it is a dog one it falls on the left area. But if the points fall around the line it can be that we are making mistakes. In this case we would reduce the performance of our classifier. From this we obtain that one of the simplest problems in classification is regression.

7.2 Regression

To explain it we take an example as a use case. We are trying to predict the share price of a company that is about to go public. So, if we want to go public, what is the expected price? We need to have some examples in order to determine what is the shape, the inclination of this line in terms of slope and intercept. And we're doing it by looking at other companies similar to ours, at their revenue and the corresponding share price(features). In this way we create the training set that will be useful for us to define the parameters of the line.

By looking at the line we have we could try to predict a new pair of coordinates and then compare with the real point. This process is the simplified version of the creation of a loss function. Meaning that we are trying through this loss function to understand how close we are to our ground truth.



- Define loss function (our score to be minimized)
- Find the parameters of the line (slope, intercept)
- Use the line for testing

In our problem:

- We use the training to determine the line
- In testing, given the revenue we can predict the share price

More formally:

- We have a set of training samples $D = \{Z_1, Z_2, \ldots, Z_n\}$;
- We have an unknown process $P(Z)$;
- And a loss function $L(f, Z)$ where f is the decision function;
- In supervised learning each example is a pair $Z(X, Y)$ where:
  - X contains the features for that sample;
  - Y is the known/expected output;
  - f takes X as an argument and outputs something in the range of Y
- Loss function defined as:

$$L(f, (X, Y)) = ||f(X) - Y||^2$$

Overall, a classifier can be schemed down in a very, very, very simplistic way to a function like that. So it tries to minimize the difference between the estimate and the corresponding ground truth.
So far we were talking about binary classification, but usually we have something which is a multi class detector and needs a more complex loss function.

It tends to not have very sharp boundaries and the function, which is typically used, is a lot like a conditional log-likelihood. Very quickly, we are moving into the probabilistic domain, in which we say that $f_i(X)$ estimates $P(Y = i|X)$, where Y is

a class (finite integer) and the loss function is the negative conditional log-likelihood. Of course we have the constraint that the entire sum of all these functions are mapped with one.

$$f_Y(X) >= 0$$

$$\sum_i f_i(X) = 1$$

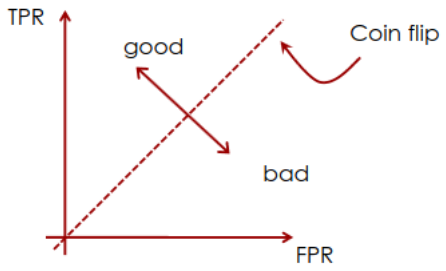
Another way for multi class classification is in the world of unsupervised learning, we are talking about clustering. In this case we do not provide any prior knowledge to the classifier about the input data. We feed samples into the algorithm and it progressively adjusts the positioning of the different clusters and associates the different input in blind. Essentially, through clustering the space is partitioned in regions centered around a prototype (centroid).

As an output you may have correct or incorrect detection, in a binary classification problem you can have True positive (hit), True negative, False positive or False negative (miss).

If we have a classifier for cat and dog and feed a bird into the system we obtain that the likelihood is extremely low, so we can implement a so-called reject option, in which the system prefers to just reject the examples and not give an answer.

7.2.1 The ROC curve, Precision and Recall

The ROC (Receiver Operating Characteristic) curve is a plot of the True Positive Rate vs the False Positive Rate. Each setup of the system is a point in the ROC space.



We simply would like to have something which stays above the coin flip. More precisely we want something that minimize as much as possible the contributions along the x axes and instead maximizes the contributions to the TPR axes.

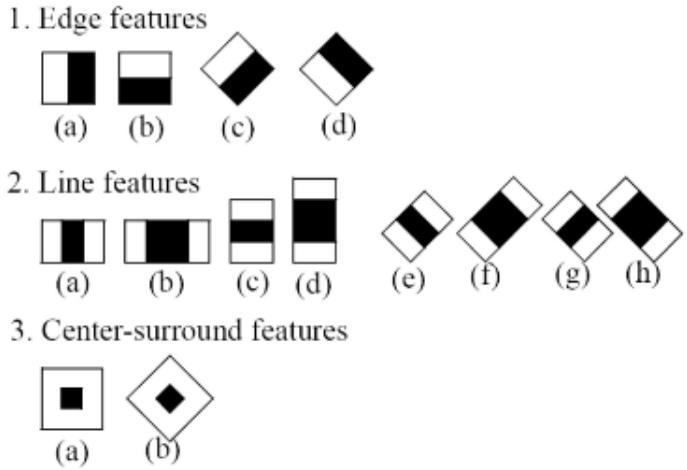
NB: If the performance is on the line or under it'll be faster to use a coin to classify.

Given TRP and FPR we can define other two basic parameters which are the precision

and the recall.
Precision: $TP/(TP+FP) = hit/(allretrieved) \Rightarrow$ probability that a randomly selected elements is relevant.
Recall: $TP/(TP + FN) = hit/(hit + miss) \Rightarrow$ probability that a randomly relevant element is retrieved in the search. If we extend the concept to more complex scenario where we have multiple classes, we can create a confusion matrix.

7.3 The face detection problem

Once we have identified the significant features for the object that we are looking for, we can take some samples and train the classifier. If we're looking for faces the problem can be to find all faces (so binary detection), or find Bart-Lisa-Homer-Marge (recognition). The two scenarios are very different and involve the application of different algorithms. Let's talk about the Viola-Jones algorithm, that is probably the most widespread face detector also in terms of implementation because it is faster, quite robust and available in openCV. The goal was to implement a robust classifier using simple binary features. So the classifier is constructed using 14 Haar-like features: The idea is that the sum

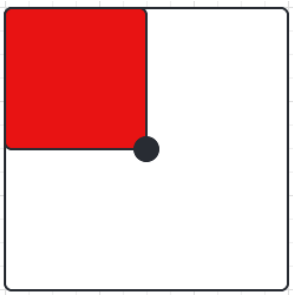


of the pixels within the white rectangles are subtracted from the sum of pixels in the black rectangles.

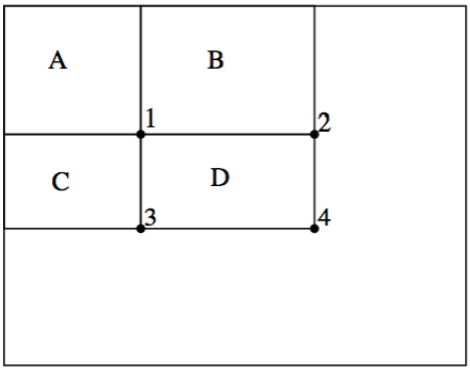
Rectangle features can be computed easily using an intermediate representation for the image called the *integral image*.

$$ii(x, y) = \sum_{x'<=x, y'>=y} i(x', y')$$

It is an intermediate representation of the image in which we are defining a single value $ii(x, y)$ that represents the contribution of the pixels before that location.



So, for example, if we want to compute the integral image at the black point, we just need to dump up the contribution of the pixels in the red area.
This makes the computation faster, if we start scanning the image from the top left corner and we go to the bottom right corner, we can compute the integral image in a single pass. At any shift we can rely on the previous summation, add the last contribution and so on and so forth.
Remember that we are evaluating binary features, so we are looking at the difference between the sum of the pixels in the white area and the sum of the pixels in the black area.



So, looking at the image above, we can compute the sum of the values in D in the following way. Knowing that:

$$1 = A, \quad 2 = A + B, \quad 3 = A + C, \quad 4 = A + B + C + D$$

We obtain that the sum of the values in D is 0:

$$\begin{aligned} D &= 4 - (A + B + C) \\ D &= 4 - (2 + C) \\ D &= 4 - (2 + 3 - 1) \end{aligned} \tag{7.1}$$

7.3.1 Recursion

Using the following recursions, the integral image can be computed over the entire image in one single pass:

$$s(x, y) = s(x, y - 1) + i(x, y)$$

$$ii(x, y) = ii(x - 1, y) + s(x, y)$$

where s is the cumulative row sum.
We can basically rely on what has been computed up to the previous role and then add the last contribution.

Now we go into the detail of the classifier. In literature, we have a lot of different classification algorithms. We have deep networks which are the most popular nowadays, but there is a time when there were other algorithms which are still fine. Usually the most popular algorithms are:

- SVM (Support Vector Machine);
- Artificial Neural Networks such as Multi Layer Perceptron or Radial Basis Function;
- Boosting algorithms such as AdaBoost;

The paper that professor is referring, Viola-Jones, is actually based on the AdaBoost algorithm (adaptive boosting classifier), so now let's see how it works.

7.3.2 AdaBoost

The AdaBoost algorithm is a boosting algorithm that combines multiple weak classifiers to create a strong classifier. They decided to create a final function f, that we recall it to be what we want to get in our classifier, as a linear combination of the weak classifiers h_i. The weak classifiers are trained sequentially, each new classifier is trained on the misclassified samples of the previous classifiers.

So, in terms of output, what we have is a sum of different contributions, where each contribution is weighted by a coefficient α_i. After defining the threshold, it will return either 1 or -1.

$$f(x) = \sum_{i=1}^{T} \alpha_i h_i(x)$$

where h_i is the weak classifier and $H(x) = sign(f(x))$ is the final strong classifier.

$$h_i(x) = \begin{cases} 1 & \text{if } f_i(x) threshold \\ -1 & \text{otherwise} \end{cases}$$

So, if it returns 1 it means that somehow the object is relevant, otherwise it means that we don't find this element as relevant.

$$f_i = Sum(r_{i,white} - Sum(r_{i,black}))$$

More formally, given a set of points $(x_i, y_i), i = 1 \dots m$, with $y = \pm 1$, and a set of weights all set to the same value $D_1(i) = 1/m$, what we do is to evaluate the features and try to find the configuration of the classifiers that minimizes the error. So, we try to minimize the sum of the weights of the misclassified samples.

$$h_t = argmin_{h_j \in H} \epsilon_j = \sum_{j=1}^{m} D_t(i) I(y_i \neq h_j(x_i))$$

where I is the indicator function(binary). If error is less than 0.5 we can use the classifier, otherwise we can discard it. Now we need to choose the weight of the classifier and from here we start to update progressively the system, we adjust the learning procedure for specific features inside a specific image.

$$\alpha_t = \frac{1}{2} log(\frac{1 - \epsilon_t}{\epsilon_t})$$

$$D_{t+1}(i) = \frac{D_t(i) e^{(-\alpha_t y_i h_t(x_i))}}{Z_t}$$

where Z_t is a normalization factor.
NB: Samples that are more difficult to classify will have higher weight in the next iteration.
As we say before, the final output is the summation of the different weak classifiers with the corresponding weight and the sign which basically tell us if the object is relevant or not from a graphical perspective.

$$H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$$

Let's do a quick recap...

- In AdaBoost the combination of the weak classifiers improves the speed of the classification process and in the algorithm proposed by Viola and Jones, classifiers are used in cascade, further reducing the computational complexity;

- The goal is to quickly remove false negatives and focus on the positive samples.

Remember that in face detection, basic features can be used to exclude non-faces. The output of the first classifier is used to trigger the second one and at each stage the negatives are rejected.

Let's make an example, if we have a face detector, and we have a face in the image, the first classifier will check if he can find a symmetry in between the left and right part of the window, if it doesn't find it, it means that at least we have not a frontal face. Then it discards all the windows that he checked so that the second classifier will not waste time on them and has fewer tasks to perform.

- In the cascade, binary features are evaluated on training images of equal size (24x24pix);
- During the training stage, the boundaries of the classifiers are learned;
- During the testing phase, the learned classifiers are used to evaluate unknown samples;
- Images are not scaled at 24x24, so a multi-scale analysis must be conducted (at the feature level $\Rightarrow$ fast by considering the type of detector [binary + integral image]);
- In case of multiple detections at different scales, windows are averaged.

For what concerns the complexity we must say that it depends on the number of classifiers, the number of levels in the multi-scale analysis and the sampling distance between the windows.