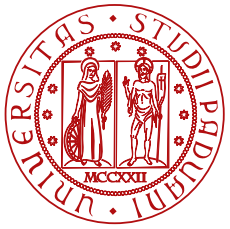


Università
degli Studi
di Padova

Dipartimento
di Ingegneria
dell'Informazione

Master degree in Computer Engineering

Computer Vision

University of Padova

Academic Year
2023/2024

Contents

1

Image representation

An image is represented by many **pixels**, the unit of space in an image. Each pixel has a value of "light acquired": the sun rays are reflected through the object and they reach the sensor of a camera with a quantity of light (energy). The sensor captures this quantity and through a process called **digitalization**, an image is created.

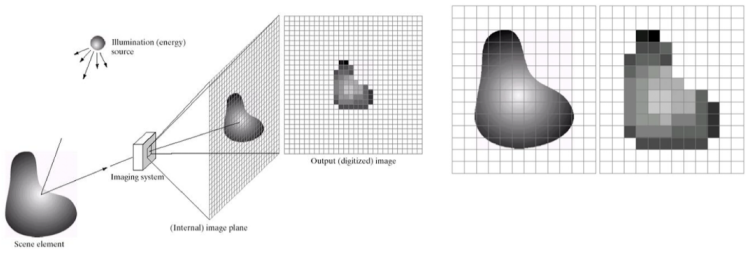


Figure 1.1: Acquisition of an image

In Figure 1.2 how it works the acquisition of an image: the light is given as a continuous signal and it is **sampled in space**, with the step of sampling that is the pixel dimensions. Then it's **quantized** with levels of quantization which depend on the resolution the scale of grey of our image.

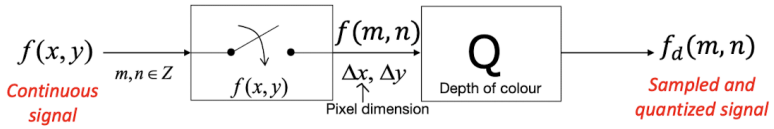


Figure 1.2: Digitalization process

In Figure 1.3 an example of this process.

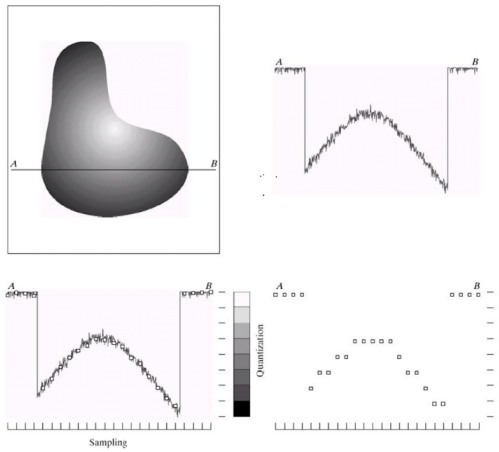


Figure 1.3: Example of digitalization

We can represent an image also with a 3D representation based on the value of each pixel: we represent the shape of the image with a two-dimensional plane and we develop it in another dimension; for each pixel of the 2D plane the value of light represent the third dimension. In Figure 1.4 an example.

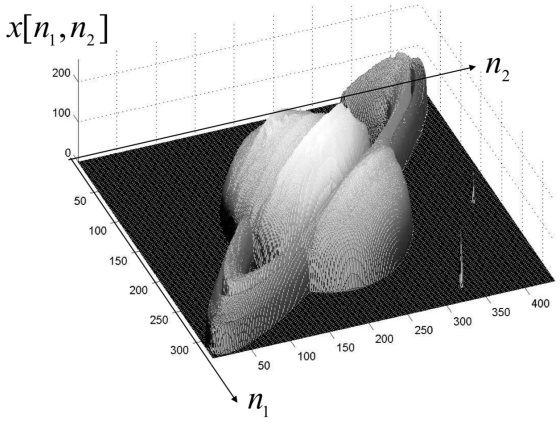


Figure 1.4: 3D representation

There is an important convention that is useful to remember: the coordinate convention is different between the usual one. The y axis points down and if we get a couple (x, y), it represents firstly the column and secondly the rows.

The x-axis runs horizontally from left to right: increasing x-values move from the left edge of the image towards the right edge.

The y-axis runs vertically from top to bottom: increasing y-values move from the top edge of the image towards the bottom edge. **Coordinate Representation**: Each pixel in the image is referenced by a coordinate pair (x, y):x represents the horizontal position (which column), y represents the vertical position (which row). For example, the coordinate (50, 100) refers to the pixel located 50 pixels to the right and 100 pixels down from the top-left corner.
Every pixel is identified by $x(n_1, n_2)$ where x represent the image and n_1, n_2 the position; for each pixel, depending on the type of image, we have a tuple of elements: if it is a colored image, usually the most used representation is **RGB** with three different values, one for each channel, so every pixel is associated with a triple of elements. Instead, if the image is in **grey-scale**, each pixel is associated with only one value. An example of RGB image is in Figure 1.5

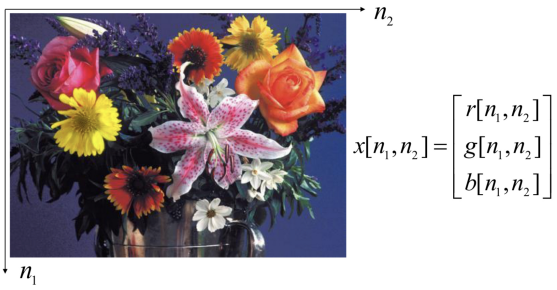


Figure 1.5: RGB image

Moreover, there are some important indicators which provide us information about the goodness of an image: the **spatial resolution** and the **gray-level resolution**.
Talking about spatial-resolution, we refer to the smallest detectable detail in the image or rather the number of pixels per unit distance.
In other words, resolution is how it is well defined our image. An high resolution means a high number of pixels, so we can create a more detailed image.
Instead, the grey-level resolution means the smallest change of color that we can have or rather the number of bits per pixel (with 8 bith we have 256 possible number in binary). In other words, if we get a high number of bit per pixel, we can represent better the different shades of grey of an image.
Retaking the image 1.1, the spatial resolution is defined by Δ_x, Δ_y, then steps of quantization are defined by grey-levels: a higher quantity of grey-scale levels means a better quantization.

2

Transformation of an image

Most of times happen that images are not in the form that we want so we need to transform them.
There are many different ways of transforming an image.

2.1 Single pixel operations

Here we refer about operations which modify each pixel one by one. The value of the output pixel depends only on the initial pixel value.
Considering a common gray scale image with L gray levels (from 0 to $L-1$), **single-pixel** operations are functions that change the gray levels of an image.
Elements which are involved are:

- Function $I(x, y)$ representing the image and, more precisely, a single pixel
- Function $T(\cdot)$ representing the grey level changing function

In Figure 2.1 this concept represented.

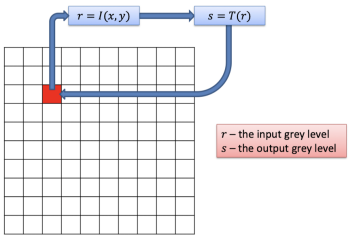


Figure 2.1: Single pixel operation

There are different single-pixel operations but most used are the following.
Negative image: switch dark and light, with the following equation:

$$T(\cdot) \hookrightarrow s = (L-1) - r$$

Log transform: this transformation, as shown in Figure 2.2, increases strongly low levels.

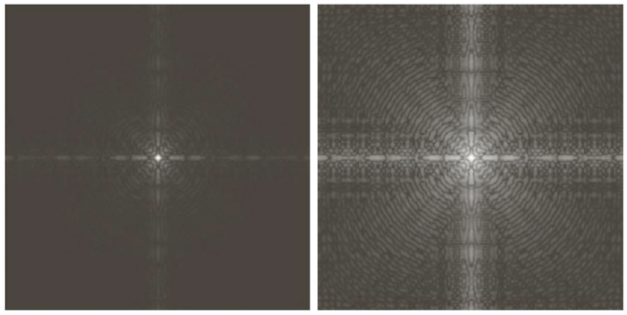


Figure 2.2: Example of log transform

In other words, it allows to highlight differences among pixels by enhancing more darkest pixels. It can be useful when we have a dark image and we want to highlight it.
The function is the following, with the plot shown in Figure 2.3:

$$T(\cdot) \hookrightarrow s = c \ln(1+r), \; c = \frac{L-1}{\ln L}$$

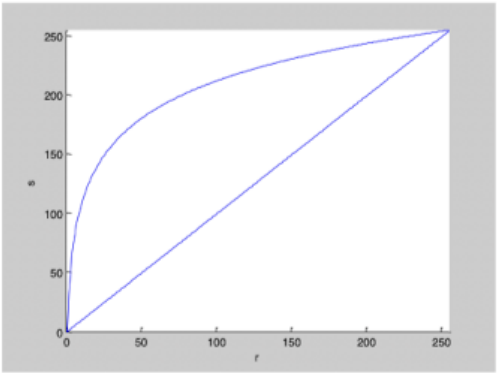


Figure 2.3: Log transform function

As we can see, there is an high improvement of dark pixels, while when we move

to high values of grey, meaning white pixels, the improvement it's not high.
Gamma transform: it's similar to the log transform with the difference that this type of transformation is **tunable**. The equation is the following:

$$s = cr^{\gamma}, \; c = (L-1)^{1-\gamma}$$

while the diagram is below in Figure 2.4.

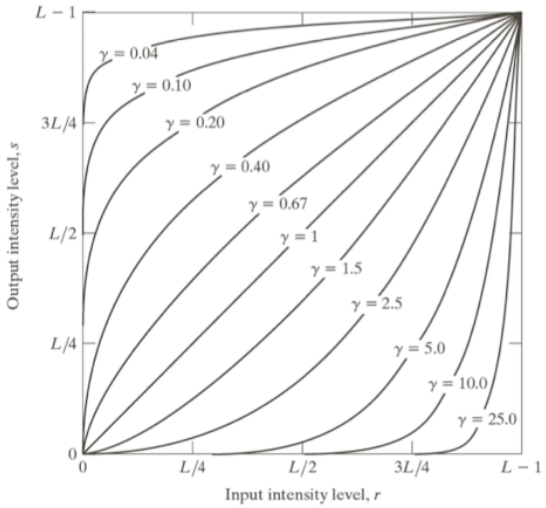


Figure 2.4: Gamma transform function

By variating γ, we can obtain the same result of log transform. It's useful, based on the image, for improving its quality; If $\gamma \approx 0$, dark pixels are enhanced (increased grey-scale value). Then, if $\gamma \approx +\infty$, light pixels are enhanced (so it's reduced the grey-scale value). A clear example can be seen in Figure 2.5.

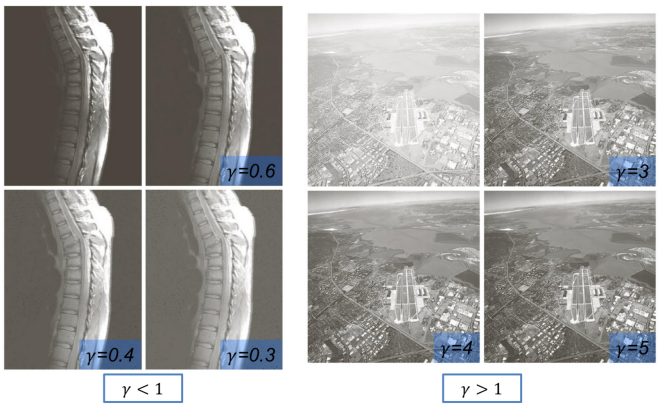


Figure 2.5: Example

Contrast stretching: recalling the definition, contrast is the difference between highest and lowest intensity level in the image. With an high contrast, there is a wider range of grey scale that means more possible details. Instead, a small contrast means that all pixels are very near.
So contrast stretching is a technique used to enhance the contrast of an image, spreading differences between pixels.
It is defined by a set of segments, depending on the type of enhancement that we want to perform.
As we can see in Figure 2.6, it's composed by three segments.

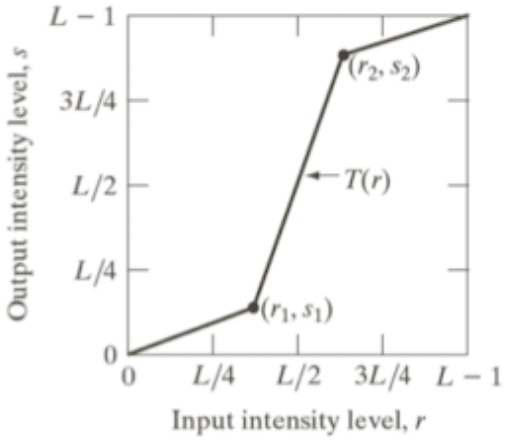


Figure 2.6: Contrast stretching function

We can see that there are two slopes with low angular coefficient: it means that, in that range of greyscale, the contrast is not so strecthed. Rather, from $\frac{L}{2}$ to $\frac{3L}{4}$, it's stretched a small range of grey colors and it means that there will be big change in the contrast and the image will be well defined.
Supposing an image with all grey levels contained from 100 to 150 (small contrast): if we define a stretch with a high slope, which connect $(100, 0)$, $(150, 255)$, the stretching is maximum.
Similar but more extreme is the **thresholding**: is another single-pixel operation based on a threshold T_h and a low level L_L and a high level L_H. The function is shown in Figure 2.7. In other words, if the input grey level is lower than L_H, the output will be L_L and viceversa with L_H. Supposing that the two values are 0 and 255, the image will be black and white.
Slicing: slicing is another single-pixel operation and enables to consider a subset of the given image. There are two types, **intensity slicing** which consider only

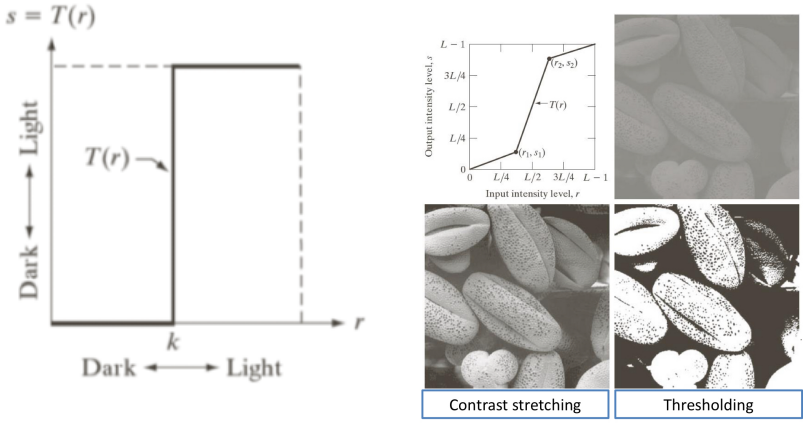


Figure 2.7: Thresholding function and a comparison between thresholding and contrast stretching

a subset of the grey levels and set the others to 0/255, and **bit plane slicing** that consider only a subset of the bits encoding the gray level.

2.1.1 Image histogram and histogram equalization

An histogram is a measurement tool in which on the x axis there are sort of **bins** that represent the possible grey values. For each pixel with a determinate grey value, we "add a ball in the correspondent bin". So the final result will be a precise histogram.

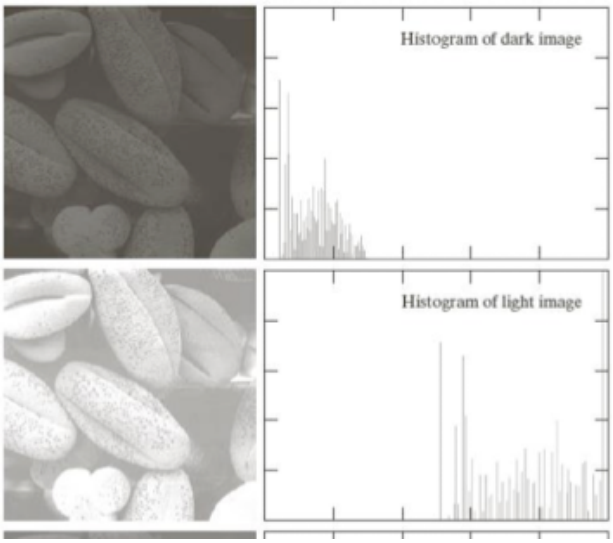


Figure 2.8: Examples of histogram

We can define a probability density function:

$$p(r_k) = \frac{h(r_k)}{MN} = \frac{n_k}{MN}$$

where $p(r_k)$ is the probability of having the grey value r_k, given by the quotient between the number of pixels whose intensity is r_k and the total number of pixels.
Retaking some concept, we can treat it as PDF (Probability Density Function).
For a RV X, PDF is a function $f_X(x)$, which returns the probability that event x occurs.
Instead, the CDF is used to compute the probability that our RV $X \leq x$: $F_X(x) = P(X \leq x)$.
So, by integrating PDF until the value x, we obtain the value of CDF.
Histogram are used for evaluating image statistics, compression, segmentation, image enhancement.
We would like that the histogram is **flat**: it means that grey levels are well

distributed. **Histogram equalization** is the process that flattens the histogram as shown in Figure 2.9.

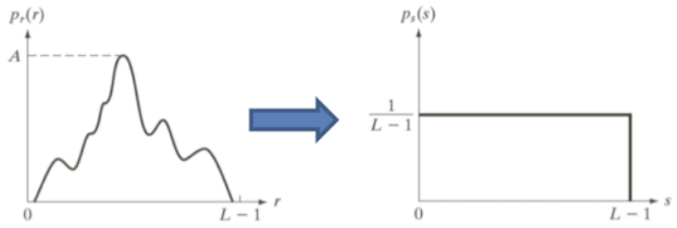


Figure 2.9: Histogram equalization

It's based on a equalization function, the usual $T(r)$, which maps grey level on another grey level.
In poor words, we are essentially redistributing the pixel intensity values to cover all the range.
The function that does this is the CDF:

$$s = T(r) = (L-1)\int_0^r p_r(w)\,dw$$

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{r} p_r(r_j)$$

with the second one for discrete functions.
$T(r)$ is monotonically non-decreasing, so the inverse function is available. Input and output are both bounded between 0 and $L-1$.
Let analyze an example: suppose to have an image with 8 grey levels and the distribution is given by Figure 2.10.

r_k	n_k	$p_r(r_k) = n_k/MN$
$r_0 = 0$	790	0.19
$r_1 = 1$	1023	0.25
$r_2 = 2$	850	0.21
$r_3 = 3$	656	0.16
$r_4 = 4$	329	0.08
$r_5 = 5$	245	0.06
$r_6 = 6$	122	0.03
$r_7 = 7$	81	0.02

Figure 2.10: Example

r_k are the possible grey levels, with beside the number of pixel with that level. In the top-right column the probability that a pixel has r_k grey level.
By considering the example, we can compute the output for a pixel using discrete CDF function. In Figure 2.11 equalized values:

r		s_i	round
0	S_0	1.33	1
1	S_1	3.08	3
2	S_2	4.55	5
3	S_3	5.67	6
4	S_4	6.23	6
5	S_5	6.65	7
6	S_6	6.86	7
7	S_7	7.00	7

Figure 2.11: Example

As we can see, the value of s_i are quantized to the closest integer, due to the fact that we have fixed levels of quantization. In order to have a better flat equalization, it's ideal to have a higher number of values.
For the previous example we can plot histograms and the CDF as shown in Figure 2.12.

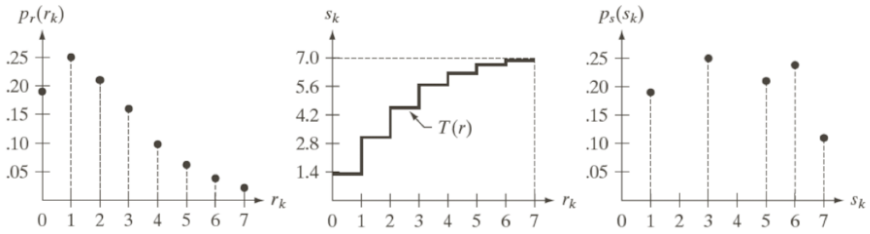


Figure 2.12: Histograms of example

The second plot represent the CDF function, that is our transformation function: given a grey level as input, it returns a value s_k as output.
In order to plot the flatted histogram we have to take each r, see what is the value s_i and take the closest quantized value.
So, for example, all pixels that had grey level = 4 will have a grey level of 6.
In order to plot the flatten histogram, we need to apply the transformation function and then recompute the probabilities: for example, both grey level 3 and 4 are mapped to 6, so the new probability for grey level 6 is obtained by the sum

of all old pixels with values 3 and 4 divided by the total number of pixels.
Different types of image have different equalization functions: if the image is very dark, the equalization function maps dark image to higher value of grey scale, instead, lighter image has an equalization function which does not modify pixel values until a certain value of gray, as we can see from Figure 2.13

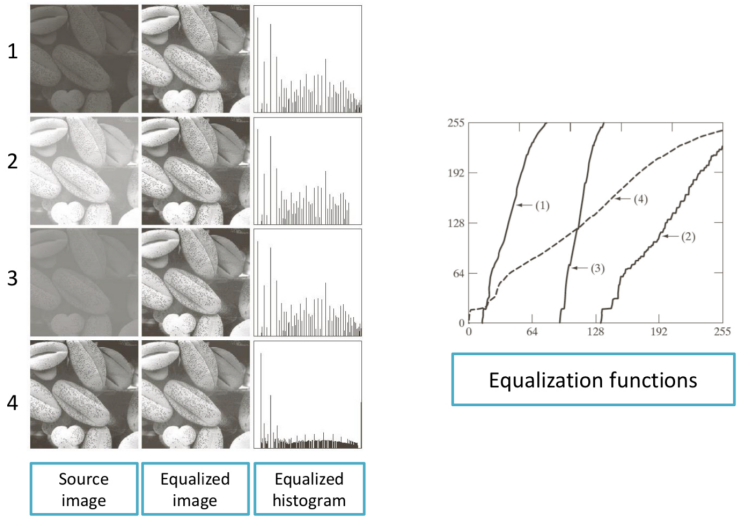
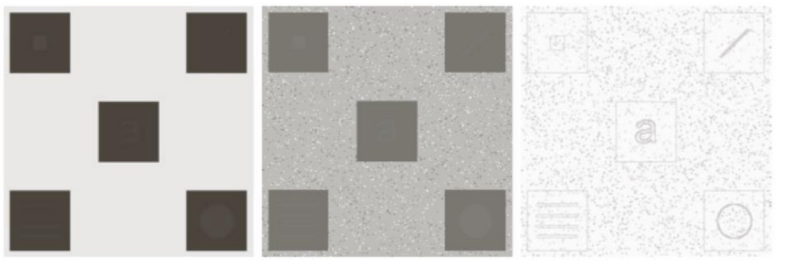


Figure 2.13: Different equalization function of different histograms

The concept of histogram equalization can be extended to **local histogram** equalization or **histogram specification**.
In the first one we consider if we want to equalize a small group of neighbour pixels, useful for image which have very different pixel distribution. As we can see from Figure 2.14, it has very good result respect the global equalization.



a b c
FIGURE 3.26 (a) Original image. (b) Result of global histogram equalization. (c) Result of local histogram equalization applied to (a), using a neighborhood of size 3 × 3.

Figure 2.14: Example of local equalization

Instead, the histogram specification is a process that equalize the histogram

specifying the desired output shape.
Firstly our histogram is equalized based on the input image. Then the desired shape is specified by listing all the points (so it's defined a **desired equalization function**) and this desidered equalization function is inverted. At the end, we map our equalized histogram with values s_k to values $z = G^{-1}(s_k)$ of the desired shape.
In Figure 2.15 this process.

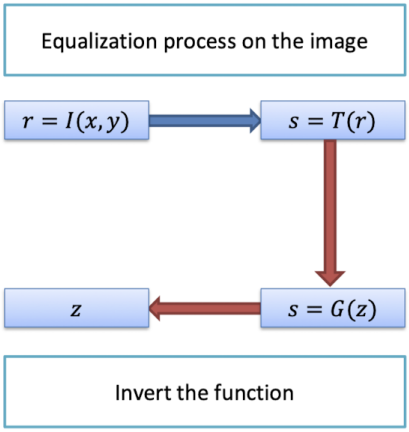


Figure 2.15: Histogram specification process

Firstly we apply the classic equalization function $T(r)$; after that, we connect it with our desired equalization $G(z)$ by giving as input of the inverted function the output of $T(r)$.
We can specify the mathematical formulation:

- Equalize the input image

$$s_k = T(r_k) = (L-1)\sum_{j=0}^{k} p_r(r_j)$$

- Define the desired histogram, giving the target probability $p_z(z_i)$ and evaluating the corresponding CDF

$$s = G(z_q) = (L-1)\sum_{i=0}^{q} p_z(z_i)$$

- Obtain the inverse transformation

$$z_q = G^{-1}(s_k)$$

and apply it: equalize the input image $r \hookrightarrow s$ and then $z = G^{-1}(s)$

Equalization sometimes can be unsuitable, as when the number of pixels with low gray levels is very high. In this case, it's more convenient use a specification function which is manually defined.

2.2 Local operations

Image filtering, in the context of computer vision, refers to a process of modifying or enhancing an image by applying a specific algorithm or a set of mathematical operations to pixel values in a small neighborhood around each pixel.
The goal of image filtering is to manipulate the image in such a way that certain features become more prominent , making it easier to extract relevant information or to improve the image's overall quality.
Now we focus on local operations, which deal not only with a single pixel but with more pixels.
The output value depends on the initial values of the pixel + its neighbors.
Image filters are typically represented by a small **matrix or kernel**, which slides over the entire image.

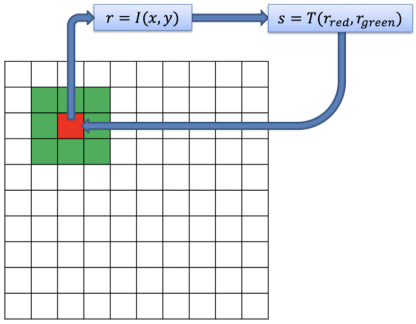


Figure 2.16: Local operations

In Figure 2.16 is illustrated an idea of this concept: the new red pixel (output) depends both on green and red pixels.
Local operations are defined based on **filter** or **kernel**, with the kernel that defines a neighborhood (green pixels) and a weight associated to each pixel involved in the computation (so for each pixel in the neighborhood is associated a weight, so the kernel defines the region of computation and weights of computation).
Local operations are done in the **spatial domain** of the image so that is why it is called **spatial filtering** and why the kernel is also called **spatial filter**.
There are several options in order to apply a spatial filter to an image: the most important subdivision is given by **linear and non linear filtering**. Depending on the process applied to the image filter can be linear or non-linear.

2.2.1 Linear filtering

Linear filtering uses an evaluation of **convolution/correlation**.
At each position (so for each pixel), the kernel is multiplied with the pixel values within its neighborhood, and the resulting values are combined to form a new value for the central pixel.
The size and values of the kernel determine the nature of the filtering operation.
So, it's done a correlation operation, which takes our filter and our pixel input (with its neighbors) and computes in the respective "red" pixel (**destination pixel**) the multiplication of every weight with the associated pixel, and at the end, a sum is performed.
We can see an example of this process in Figure 2.17, where the filter is provided by a sort of oracle; we will see how to give values to filter.

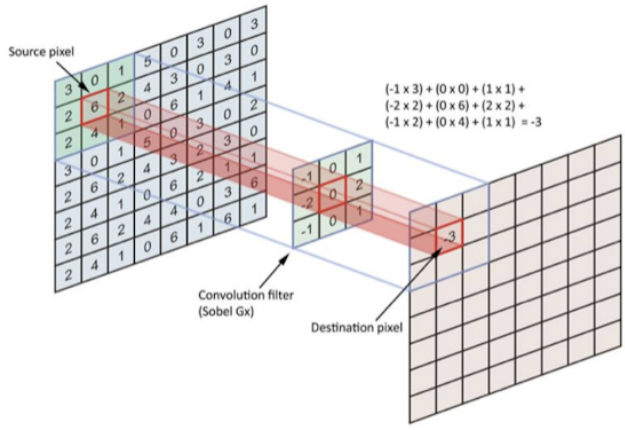


Figure 2.17: Local operations convolution

Supposing that dimensions of a filter are mxn, defining $m = 2a + 1$ and $2b + 1$ such that are odd values.
Correlation is defined by:

$$g(x,y) = \sum_{s=-a}^{s=a} \sum_{t=-b}^{t=b} w(s,t)I(x+s,y+t)$$

So the output image is computed by centering for each pixel the kernel and computing the weighted sum of pixels.

In the CV context, convolution and correlation are often used as synonims, since filters are usually symmetric, convolution and correlation are equal.
The filter weights can change the image **brightness**; it is unchanged if the sum of weight is equal to 1:

$$\sum_i w_i = 1$$

Resuming, in Figure 2.18 are represented possible spatial filters operation.

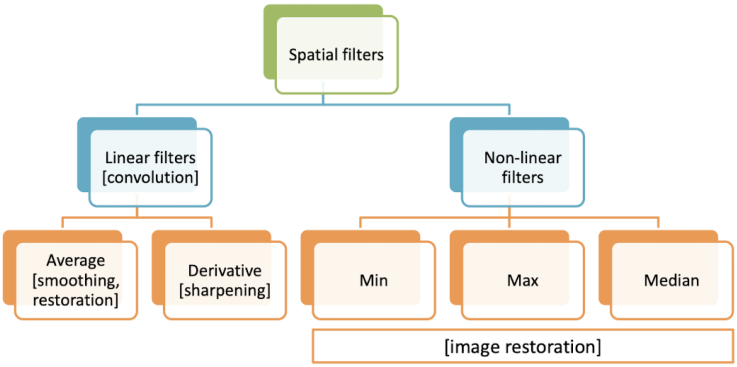


Figure 2.18: Possible local operations

Averaging filters

Convolutional filters are linear filters and now we will focus on different types of linear filter.
The first linear filter that we see is the **averaging filter**, than, as the name suggests, performs a sort of average of the neighborhood of a pixel.
The simplest average filter is the left-most of Figure 2.19.

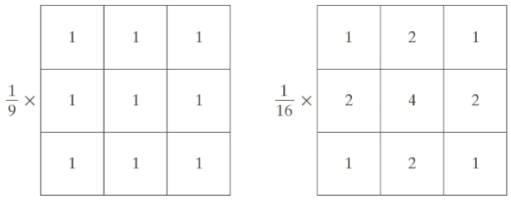


Figure 2.19: Average filters

If you analyze the computation, the new central pixel will be the average of the neighborhood pixels.

There are is another possibility that is the right-most kernel: it has a higher weight value for the input pixel, and close we move to bounds, lower are weights. Averaging filter are used for **smoothing**: larger filters generate stronger smoothing.
In Figure 2.20 an averaging filter is applied to top left most image with different filter dimensions.

Figure 2.20: Different averaging applied

A square filter of dimension $n \; x \; n$ may be separated in a filter of size $n \; x \; 1$ and in a filter of size $1 \; x \; n$ and applying them on rows and columns could lead to a faster operation. In Figure 2.21 the top-left image is the original image and

Figure 2.21: Original-2D-Horizontal-Vertical

different types of averaging filters are applied to it. The central image is the result of the application of a $5x5$ averaging filter, the right-most is the result of the application of a $1x5$ filter, and, as you can see, the cameramen is blurred

horizontally. The same thing for the last image, where is applied a $5x1$ and the blur is vertically.

Derivative filters

Derivative filters cause sharpening that means highlighting strong variations. Use of derivative filter increase the contrast along edges and fine details, making them appear more pronounced and sharper.
These types of filters focus on areas of rapid change (like an edge, in which the image move from dark to light) and enhance the difference in intensity or color between the pixels on either side of the edge or detail.
This enhancement makes the edges and details stand out more prominently, giving the illusion of increased sharpness and clarity.
We want to implement derivative on image. We know that **first order derivative** is zero in flat segments, non-zero on the starting point of a step/ramp and along ramps.
So, related to an image, a flat segment corresponds to a region with constant grey level and the start of a ramp to a region where the grey level is changed.
So, in other words, a derivative filter applied to a region with constant grey scale, has to give as output 0, as result of weighted sum of pixels value. Instead, non-constant regions as edges, that could be represented by ramps (if the variation is slow) or by steps (if the variation is quite instant), would have as result of a derivative filter a non-zero element, as result of weighted sum of the pixels.
Instead, a **second order derivative** is zero in flat segments and along ramps of constant slope but it's non zero on starting and ending point of a step/ramp.
Taking as example Figure 2.22, it represents variations on a single row of pixels so in the x axis we have pixels coordinates of a row.

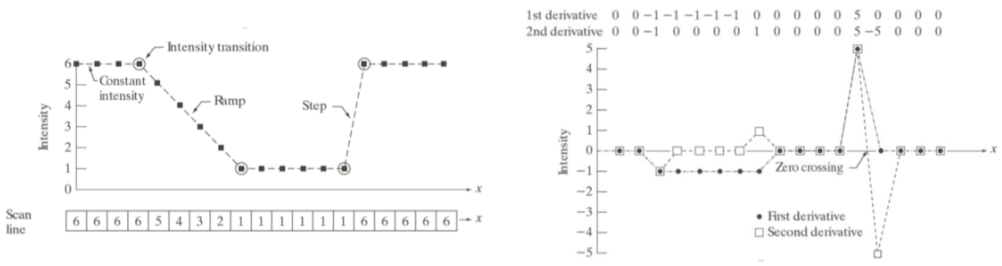


Figure 2.22: Intensity diagram

The first order derivative can be also expressed as:

$$\frac{\partial f}{\partial x} = f(x+1) - f(x)$$

Instead, second order derivative can be expressed as:

$$\frac{\partial^2 f}{\partial x^2} = f(x+1) + f(x-1) - 2f(x)$$

When dealing with images, which are inherently two-dimensional, we use the concept of **partial derivatives** to analyze changes in intensity.
For an image represented as a function $f(x, y)$, where x and y are the coordinates in the image plane, the partial derivatives are:

$$\frac{\partial f}{\partial x}$$

This represents the rate of change of the image intensity in the horizontal direction (x-axis).

$$\frac{\partial f}{\partial y}$$

This represents the rate of change of the image intensity in the vertical direction (y-axis). For **first order derivative**, we are interested in **gradient** which is a vector that combines these two partial derivatives. It is denoted as:

$$\nabla f(x, y) = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$$

It gives information on the **direction of steepest ascent of intensity** and on **rate of change of intensity at a particular point**. The magnitude of the gradient is:

$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

A larger magnitude indicates a more significant change in intensity, typically corresponding to edges in the image. The direction of the gradient is:

$$\theta = \arctan\left(\frac{\frac{\partial f}{\partial y}}{\frac{\partial f}{\partial x}}\right)$$

Gradient is not isotropic (that means that the partial derivative along x and y are different) and it make sense since the derivative on x refers to rows and derivative on y refers to columns.
The magnitude of the gradient (it measures how quickly pixel values are changing from one point to another within the image, high gradient magnitudes typically indicate regions of significant change in pixel intensities, which often correspond to edges or boundaries between objects in the image), is isotropic.
The derivative operations can be implemented using appropriate filters:

- Direct application, 2 filters for first order and three filters for second order
- Operation on larger ares, less exposed to sudden change of pixels intensity

We can compute gradient using a kernel of two vectors 1 x 2 ($[-1, 1], [1, -1]$) but we don't like it because it's not a square and there is not a central point, causing issues.
We can use **Sobel** or **Roberts** filters as shown in Figure 2.23.

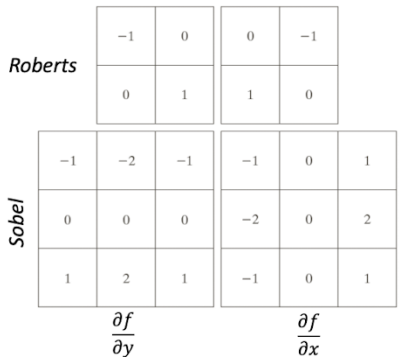


Figure 2.23: Sobel and Roberts

It's important to notice that these representations make sense: focusing on Sobel, suppose to use it on a region of constant intensity. The weighted sum of pixel intensities is 0!
These filters represent a stable representation of first order derivative. The output images are also called **masks**.
The sum of the element of the Sobel filter is 0: the brightness is changed, the image is darker since we're basically removing constant component since it's a derivative operation.
Instead, for second order derivative, we have obviously an adding order respect first order derivative: it has a stronger suppression of homogeneous regions

while strong variations between pixels are enhanced.
The second-order derivative of an image measures the rate of change of the first-order derivative. It is used to identify regions where the intensity changes at different rates, such as corners and fine details.
For an image represented as $f(x, y)$, the second-order partial derivatives are:

$$\frac{\partial^2 f}{\partial x^2}, \frac{\partial^2 f}{\partial y^2}, \frac{\partial^2 f}{\partial x \partial y}$$

We implement the **Laplacian operator**: the Laplacian operator combines the second-order derivatives:

$$\nabla^2 f(x, y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

and implements the following equations

$$\nabla^2 f(x, y) \approx f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$

It's **isotropic, scalar and linear**.
The equation above is converted into a single kernel shown in Figure 2.24.

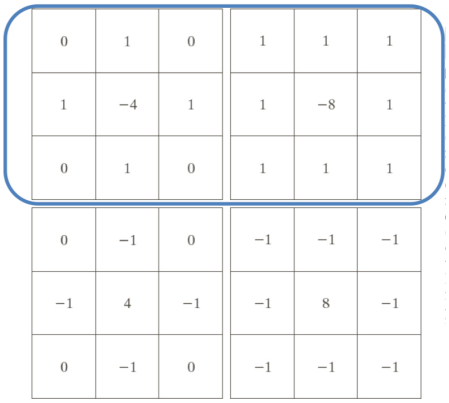


Figure 2.24: Kernels used for implementing Laplacian

All these kernels are correct for representing the Laplacian operator, and the version 8 in center is an enhance version which includes also the diagonal terms.
The laplacian can be used to sharpen the image (highlighting part of image when there are strong variations).
Combination of the image and the laplacian cause a sharp.

If the center pixel is negative, the operation of summing the image with its laplacian is:

$$g(x,y) = f(x,y) - \nabla^2 f(x,y)$$

In Figure 2.25 an example of sharpening using different filters.

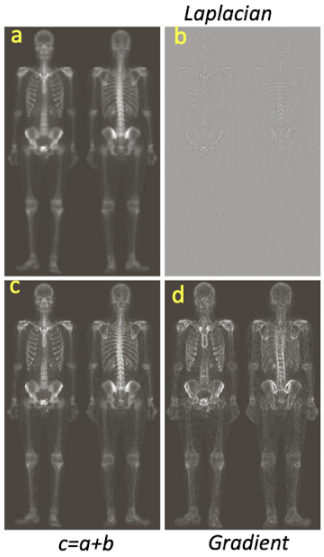


Figure 2.25: Results of different techniques applied to the same image

2.2.2 Non-linear filters

Non-linear filters implement non-linear operations. Examples of non-linear operations that we will see are:

- Min
- Max
- Median

The most advantage of non-linear filters is that these filters can **suppress** components. For example, median filter can remove a spike (a single pixel with high intensity). In our context, an element refers to a single pixel.
This operation (of removing noise) is often called **image restoration**. It's important to say that also smoothing can be used for image restoration but non-linear filters are used only for this task.
There are different sources of noise but the most important is the small area hit by light of sensor: it will be able to catch a lower amount of light respect a larger

sensor.
We can model our image as:

$$f_{acq}(x, y) = f(x, y) + v(x, y)$$

so composed by an **ideal image** + a component of **noise**.
Image restoration estimate noise properties (since there are many types of noise) and after remove it.

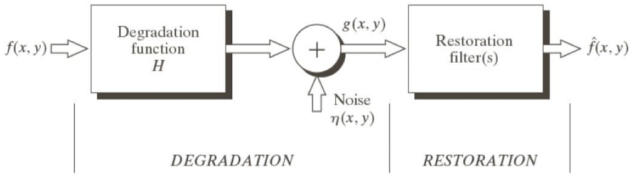


Figure 2.26: Block scheme of restoration process in an image

As we can see in Figure 2.26, noise is addictive and we want a restoration filter that provide us $\hat{f}(x, y)$ as much similar as $f(x, y)$.
As said before, there are several noise models available as shown in Figure 2.27 and they affect in different ways images.

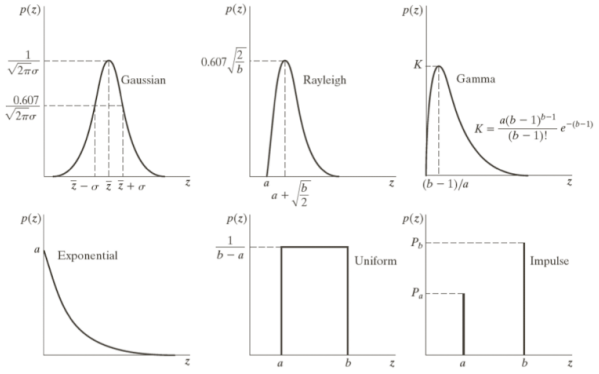


Figure 2.27: Noise models available

We have said that noise is an addition to the image. These diagrams represents the probability, for each grey-level considered, of being "selected".
The x axis represents the possible values of the noise that could be added to the pixel intensities and in the y axis the probability density of each noise value occurring. As we can see, for impulse, the noise has higher probability of be near the mean value of intensities.
A corrupted image may be restored by using **smoothing filter** (averaging or

gaussian filters) or by using non-linear filter (median, min and max filters). Supposing that an image is corrupted with gaussian noise, we can restore it using an average filter.
If we choose averaging filters, we can use **both geometric and arithmetic mean**. In Figure 2.28 an image (top-left) that is corrupted by gaussian noise (top-right) and it's restored using averaging filter (bottom-left arithmetic mean filter and bottom-right geometric mean).

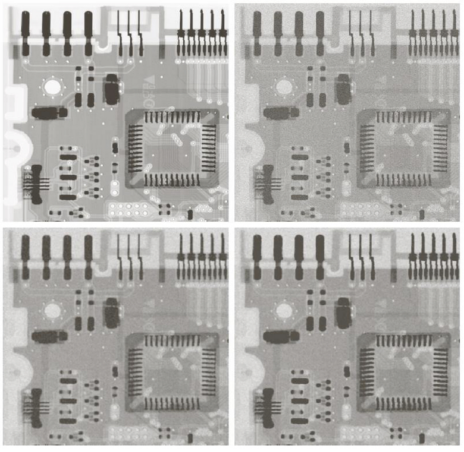


Figure 2.28: Image restored with averaging filters

Arithmetic mean filters are more effective for Gaussian noise but tend to blur edges and fine details, while geometric mean filters preserves edges and details better, making it suitable for images where preserving fine details is important. Several averaging filters available, as shown in Figure 2.29.

Name	*Equation*	*Removes Gaussian Noise*	*Removes Salt Noise*	*Removes Pepper Noise*
Arithmetic mean	$g(x,y)=\frac{1}{mn}\sum_{s,t\in R} f(s,t)$	Yes	No	No
Geometric mean	$g(x,y)=\left[\prod_{s,t\in R} f(s,t)\right]^{\frac{1}{mn}}$	Yes	No	No
Harmonic mean	$g(x,y)=\frac{mn}{\sum_{s,t\in R}\frac{1}{f(s,t)}}$	Yes	Yes	No
Contraharmonic mean	$g(x,y)=\frac{\sum_{s,t\in R} f(s,t)^{Q+1}}{\sum_{s,t\in R} f(s,t)^{Q}}$	Yes (Q=0→mean)	Q<0 (Q=-1→harmonic)	Q>0 (not both)

Figure 2.29: Possible averaging filters

Instead, if there is an image corrupted by salt&pepper noise, or rather impulsive noise, we can use **median filter** or **alpha-trimmed mean filter.**

The median filter replaces the central pixel value in a neighborhood with the median value of all the pixels in that neighborhood.
Instead, the output of a alpha-trimmed mean, is given by the following formula:

$$g(x,y) = \frac{1}{mn-d} \sum_{s,t \in R_r} f(s,t)$$

with d fixed. So it operates by ordering values of neighborhood and eliminating d elements of both tail, and then by taking the mean of remaining values, as shown in Figure 2.30.

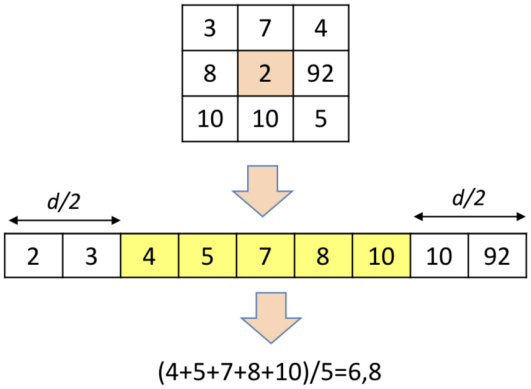


Figure 2.30: Alpha-trimmed mean filter

As we can see from Figure 2.31, results for different filters applied to the same image corrupted with gaussian + salt and pepper noise
As last, there are also **max and min filters**, used for restoring images with salt&pepper noise.
The first one gives as output the max of the neighborhood of the input pixel, removing pepper noise but highlighting salt noise.
Instead, the second one, gives as output the min of the the neighborhood of the input pixel, removing salt noise but highlighting pepper noise.

2.2.3 Adaptive filters

The filters discussed so far operate in the same way on every image and on every part of the image. Smarter behaviors can be designed: **adaptive filters** tune their effect comparing local neighborhood variance σ_L^2 and noise variance σ_η^2.
The variance is how much each pixel is farther respect the mean. A high local

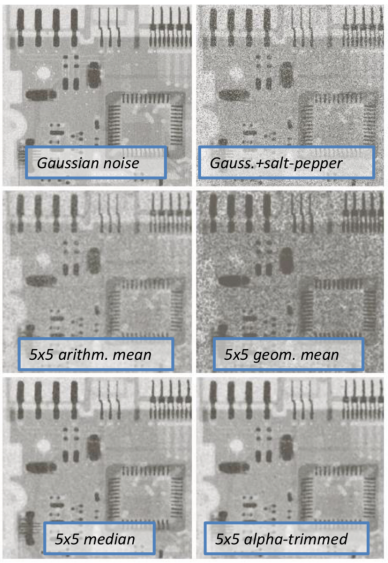


Figure 2.31: Example of different filters applied

variance indicates significant detail or texture, while a low variance suggests a smooth, homogeneous region

When $\sigma_\eta << \sigma_L$, so the filter should be weak to preserve details. However, when $\sigma_\eta \approx \sigma_L$, it suggests that the local variance is similar to the noise variance, indicating that the region is likely affected by noise, so the filter should be strong to reduce noise.

In Figure 2.32 an example of application of adaptive filter:

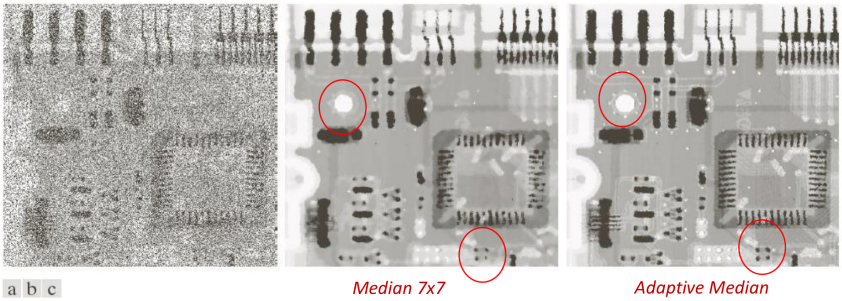


FIGURE 5.14 (a) Image corrupted by salt-and-pepper noise with probabilities $P_a = P_b = 0.25$. (b) Result of filtering with a 7×7 median filter. (c) Result of adaptive median filtering with $S_{\max} = 7$.

Figure 2.32: Adaptive filter application

3

Image in frequency: Analysis and Filtering

Signals and sampling, crucial components of the image acquisition process, can be effectively described using a mathematical tool known as the **Fourier transform**. This tool has two primary operations: the direct Fourier transform and the inverse Fourier transform. These operations define a **transform space**, where, under ideal conditions, no information is lost.
The Fourier transform enables **frequency analysis and filtering** by transitioning from one domain to another, resulting in a change in the independent variable. Commonly there is a change from time to time frequency $t \hookrightarrow f$ but, for computer vision application, we move from space to space frequency $x \hookrightarrow f_x$ and $y \hookrightarrow f_y$.
Considering a 1D example, a typical signal in time, the Fourier transform is:

$$F(f) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i f t} dt$$

while the inverse is:

$$f(t) = \int_{-\infty}^{\infty} F(f) e^{2\pi i f t} \, df$$

For example, considering a rect function $f(t) = A\ if\ \frac{-W}{2} < t < \frac{W}{2}$ and else everywhere, its transform is:

$$F(f) = AW\frac{sin(\pi fW)}{\pi fW}$$

The Fourier transform is generally complex so it is common to deal with **the magnitude of the transform AKA Fourier spectrum** or frequency spectrum, a real quantity:

$$|F(f)| = AT|\frac{sin(\pi fW)}{\pi fW}|$$

In Figure 3.1 it's shown graphs of these three functions.

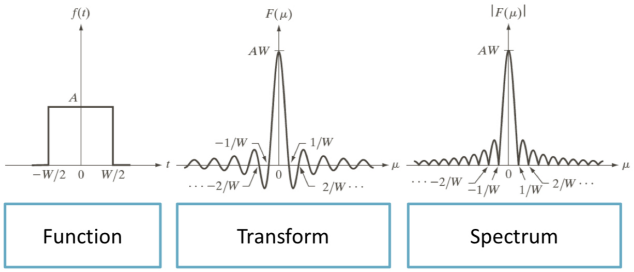


Figure 3.1: Time-frequency transformation

3.1 Signal sampling

Sampling a time continue signal is done by using a set of **regular pulses**, separated by a **sampling interval** ΔT. The sampled signal is the multiplication between the original signal and the train of impulses.
An impulse is represented by **Dirac delta function**, that is:

$$\delta(t) = \begin{cases} \infty & \text{if } t = 0 \\ 0 & \text{if } t \neq 0 \end{cases}$$

Dirac delta function has useful properties:

- Unit area

$$\int_{-\infty}^{+\infty} \delta(t)\, dt = 1$$

- Sifting property
$$\int_{-\infty}^{+\infty} f(t)\delta(t)\,dt = f(0)$$
- In a generic position
$$\int_{-\infty}^{+\infty} f(t)\delta(t-t_0)\,dt = f(t_0)$$

If we deal with **discrete signals**, the definition is:

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases}$$

and an impulse train is expressed by a sum of delta functions:

$$s_{\Delta T}(t) = \sum_{x=-\infty}^{\infty} \delta(t - n\Delta T)$$

Also in a discrete domain, the sampled signal is the multiplication between the original signal and the train of pulses, and we describe the sampled signal as:

$$\hat{f}(t) = \sum_{x=-\infty}^{\infty} f(t)\delta(t - n\Delta T)$$

and so the sampled signal is represented by a set of finite values:

$$f_k = f(k\Delta T)$$

We know that a multiplication in the time domain becomes a convolution in the transformed domain:

$$\hat{F}(f) = F(f) * S(f)$$

where $S(\cdot)$ is the transform of the delta function. So, the sampled signal, becomes:

$$\hat{F}(f) = \int_{-\infty}^{\infty} F(\tau)S(f-\tau)d\tau$$

and after some computations we obtain:

$$\hat{F}(f) = \frac{1}{\Delta T}\sum_{n=-\infty}^{+\infty} F(f - \frac{n}{\Delta T})$$

and this means that in the transformed domain, the sampled signal is original signal's spectrum **replicas**.
Replicas can be at different distances depending on the sampling period.
However, if this sampling period is not chosen well, replicas can overlap, with a phenomenon called **aliasing**.

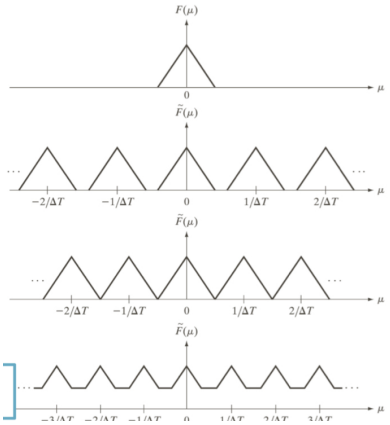


Figure 3.2: Aliasing on a sampled function

That's why we need to select the sampling frequency mindful: the **sampling theorem** impose that:

$$f = \frac{1}{\Delta T} > 2f_{max}$$

where $2f_{max}$ is known as Nyquist rate. To reconstruct a signal we need essentially to isolate one repetition in frequency and it's possible only if there are not overlaps between replicas of signal spectrum: it is possible only if the signal is band-limited and it's satisfied the sampling theorem. The isolation of one spectrum component is done by using rect, as shown in Figure 3.4

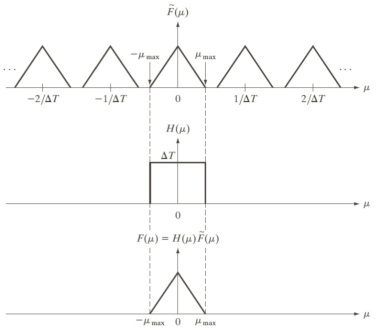


Figure 3.3: Reconstruction of an image

3.2 Images in frequency domain

If we remember, an image is constructed by sampling of the quantity light taken by a sensors. The sampling operation is performed in space, where $\Delta X, \Delta Y$, the dimension of the pixel, is the sampling period. These two values provide us also the sampling frequencies, which have to respect the Nyquist criterion.
What we've said before for 1D signal can be extended to 2D signals as an image.
Continuous 2D Fourier transform of a signal $f(x, y)$

$$F(u, v) = \iint_{-\infty}^{\infty} f(x, y) \cdot e^{-2\pi i(ux+vy)}\, dx\, dy$$

Inverse Continuous 2D Fourier transform of a signal $f(x, y)$

$$f(x, y) = \iint_{-\infty}^{\infty} F(u, v) \cdot e^{2\pi i(ux+vy)}\, du\, dv$$

Discrete 2D Fourier transform of a signal $f(x, y)$

$$F(u, v) = \sum_{x=-\infty}^{\infty} \sum_{y=-\infty}^{\infty} f(x, y) \cdot e^{-2\pi i(ux+vy)}$$

Inverse Discrete 2D Fourier transform of a signal $f(x, y)$

$$f(x, y) = \frac{1}{MN} \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} F(u, v) \cdot e^{2\pi i(ux+vy)}$$

Obviously, the concepts illustrated before remain valid also for different dimensionality: in Figure 3.5 a rect-sinc transform in 2D.

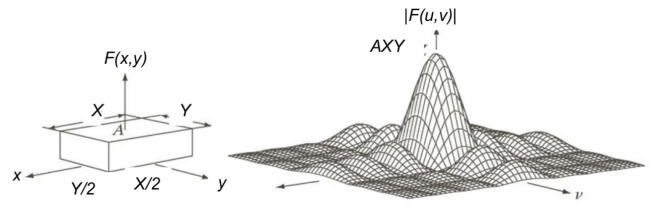


Figure 3.4: Rect in two dimensions

Being a 2D signal, sampling is performed in both directions and also replicas

are generated in both directions, as shown by Figure 3.6.

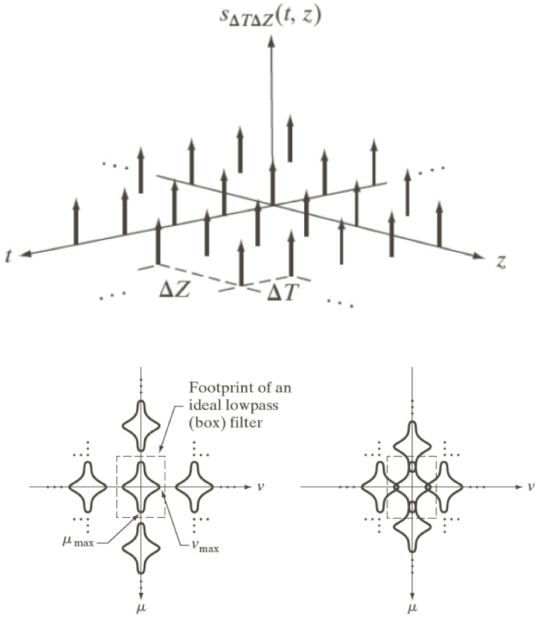


Figure 3.5: Sampling and replicas in 2D

The sampling theorem is valid in 2D and becomes:

$$F_X = \frac{1}{\Delta X} > 2u_{max}$$

$$F_Y = \frac{1}{\Delta Y} > 2v_{max}$$

always supposing that the signal is band limited by u_{max}, v_{max}, which are maximum spatial frequencies of the signal along X and Y.
It's important to say what correspond high frequencies or low frequencies in an image.
High-frequency components of the DFT represent rapid changes or sharp transitions in intensity across the image. These correspond to edges, fine details, or textures in the image. Edges and other sharp transitions produce high-frequency components because they involve sudden changes in intensity from one pixel to the next.
Low-frequency components of the DFT represent smooth variations or slowly varying patterns in the image. For example, regions of constant intensity or slowly varying gradients will have significant low-frequency components.
So, how appears aliasing in an image? In Figure 3.6 there is an example.

Figure 3.6: Aliasing effect on image

How we can see, the first image, focusing on the trousers and on the scarf, has a very sudden pattern, where light and dark pixels rapidly change (so high frequency components). If we sample this image, in an image with lower resolution, so with higher $\Delta X, \Delta Y$ (i.e. bigger pixel), aliasing phenomenom happens. In fact, by increasing $\Delta X, \Delta Y$, sampling frequencies are reduced and image is undersampled.
Aliasing can be compensated by using a Low Pass Filter, which cut some high frequencies components, in order to reduce the second term of Nyquist theorem. We focus on LPF in a section below.
In Figure 3.7 an example of images with different pattern dimension: it's to see that if the pixel dimension is too low, there is aliasing.

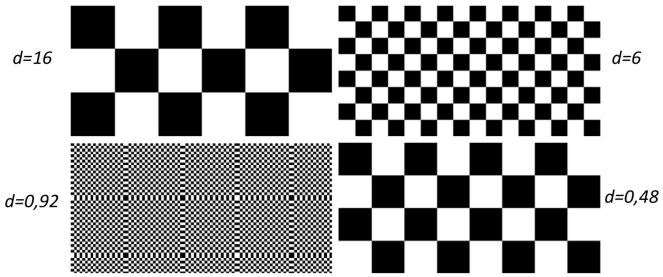


Figure 3.7: Aliasing effect on images

In the image with square size $d = 0,92$ (pixel), where 0.92 is the dimension of the square of the chessboard, the image is sampled but there are some samples where there is a portion of black square and a portion of white square. Images (c) and (d) contain high-frequency components due to the small size of the squares. In images (c) and (d), the square sizes are smaller than the pixel size (less than 1 pixel). This undersampling leads to incorrect representation of the original pattern, causing aliasing artifacts. **Parte su aliasing**

3.2.1 Visual appearance of spectrum and phase

The DFT of an image can be decomposed into **spectrum** and **phase**:

$$F(u,v) = |F(u,v)| \cdot e^{i\phi(u,v)}$$

The spectrum (magnitude) $|F(u,v)|$ can be calculated as:

$$|F(u,v)| = \sqrt{\mathrm{Re}(F(u,v))^2 + \mathrm{Im}(F(u,v))^2}$$

The phase $\phi(u,v)$ can be calculated as:

$$\phi(u,v) = \arctan\left(\frac{\mathrm{Im}(F(u,v))}{\mathrm{Re}(F(u,v))}\right)$$

The Fourier spectrum is usually centered on the 0 value. This means the low frequencies (which represent the overall structure or average color of the image) are at the center, while higher frequencies (which represent fine details and edges) are spread out towards the edges.
By default, the Fourier Transform places the zero-frequency component (representing the average color or overall brightness) at the corner of the spectrum. However, this can make it difficult to analyze and interpret the spectrum, as the low frequencies are spread out towards the edges
A shift operation is typically performed to move the zero-frequency component to the center of the spectrum. This makes the Fourier spectrum more intuitive to analyze. The shift helps in visualizing and interpreting the Fourier spectrum more easily. In the centered Fourier spectrum, the low frequencies are at the center, and the high frequencies are towards the edges. Translation doesn't affect the spectrum while rotation affect it.
However, both spectrum and phase encode information and Figure 3.8 helps us to understand which of these two contain more information.
The magnitude of the DFT represents the amplitude of different frequencies present in the image.
Higher magnitudes correspond to stronger contributions from those frequencies. In the context of images, it indicates how much a particular frequency (or range of frequencies) contributes to the overall appearance of the image. When you visualize the magnitude spectrum, you're essentially seeing the distribution of these frequency contributions.

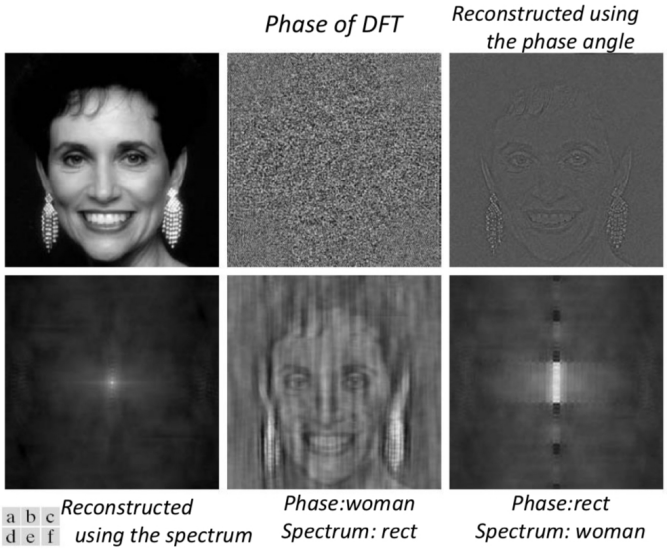


Figure 3.8: Phase and magnitude of an image

The phase spectrum contains information about the spatial relationships and positions of the features within an image.
Specifically, phase information determines the relative positions and orientations of patterns, textures, edges, and other structural elements in the image. The phase spectrum visualizes these spatial relationships across different frequencies.
If you were to reconstruct an image using only the magnitude spectrum and some random phase information, the resulting image would generally be unrecognizable. Conversely, if you use the original phase information and a random magnitude spectrum, the reconstructed image would still retain the essential structure and be more recognizable, though possibly lacking in contrast and details.

3.3 Filtering in frequency

Filters can be defined in frequency. We like filters in frequency since:

- Frequency domain filters allow for selective manipulation of specific frequency components of an image. This enables targeted enhancement or suppression of certain features, such as blurring out high-frequency noise or sharpening edges.
- Some filters in the frequency domain are separable, meaning they can be applied independently to rows and columns of the image in the frequency domain, which further improves computational efficiency.

The process that produce as output a better image is represented in Figure 3.9

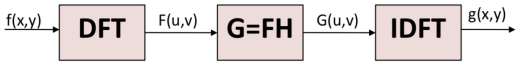


Figure 3.9: Frequency filtering on image

3.3.1 Low Pass Filter

As name suggests, low pass filter cut high frequencies and it's aim is to do **smoothing** on an image by removing noise or fine details.
The **ideal low pass filter** in frequency can be expressed as Figure 3.10, where D_0 is the cut-off frequency.

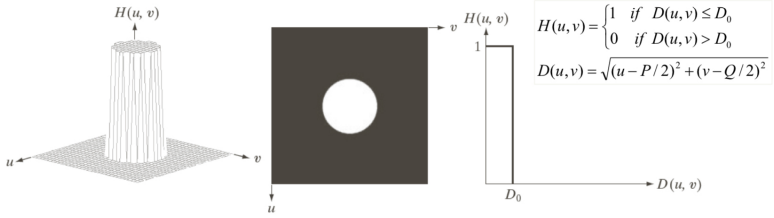


Figure 3.10: Ideal LPF in frequency

The distance $D(u,v)$ measures how far a frequency component (u,v) is from the center of the frequency spectrum. Remember that the center of frequency spectrum represents the average color or overall intensity of the image. Mathematically,

$$D(u,v) = \sqrt{(u - P/2)^2 + (v - Q/2)^2}$$

where P and Q are the dimensions of the image. The ideal lowpass filter $H(u, v)$ is defined based on the distance $D(u, v)$.
This filter in space, has ripples around main central component (due to the sudden passage from stopband to passband) and it's usually applied a padding on both directions (cutting the tails), and, in the frequency domain appear discontinuities which generate the **ringing artifacts**, as shown in Figure 3.11

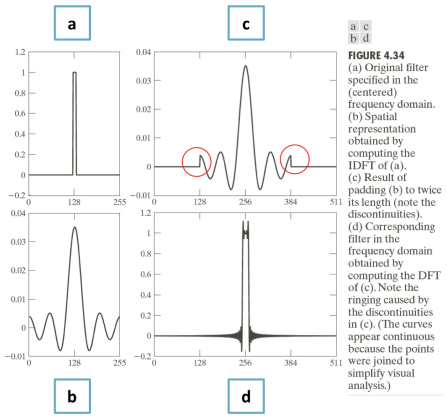


Figure 3.11: LPF in space and frequency with ringing

In Figure 3.13, different Ideal LPF (different cutoff frequencies) applied to the same image. As we can see, the ringing art is noticeable.

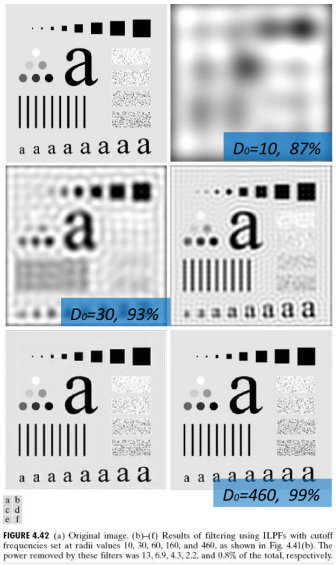


Figure 3.12: Application of ILPF

The ringing effect causes poor image quality and it's typical of ideal filters, where strong transition appears.

Ringing effects can be reduced or eliminated by using filters showing smoother transitions as **butterworth or gaussian filter**.
Butterworth in frequency can be tuned by means of a parameter n and large values of n cause this filter to be similar to ILPF, since increasing n, it's also increase the discontinuity.
In Figure 3.13 the representation of this filter.

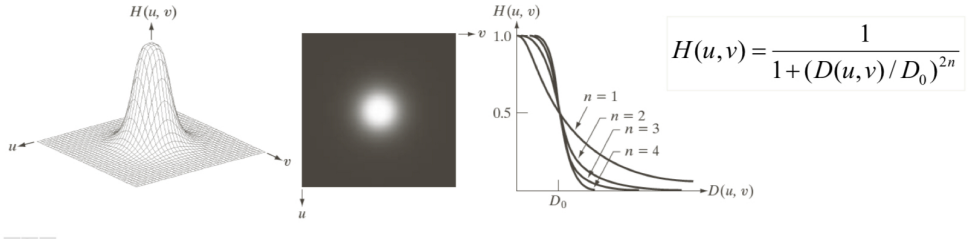


Figure 3.13: Butterworth filter in frequency

In space, the ringing effect grows with the order of the filter, since it moves to be more similar to the ideal low pass filter.
In Figure 3.14, different butterworth LPF in the space domain and the application of different Butterworth LPF to the same image.

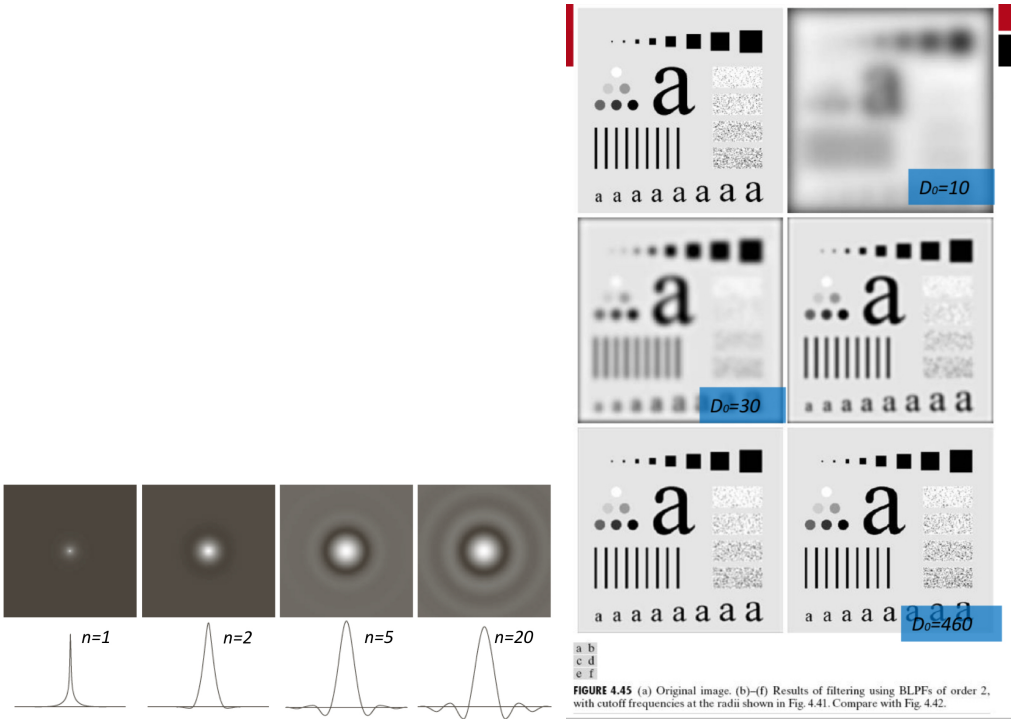


Figure 3.14: Application of BLPF

Gaussian low pass filter in frequency has no ringing, and it's tunable as butter-

worth filter, as shown in Figure 3.15.

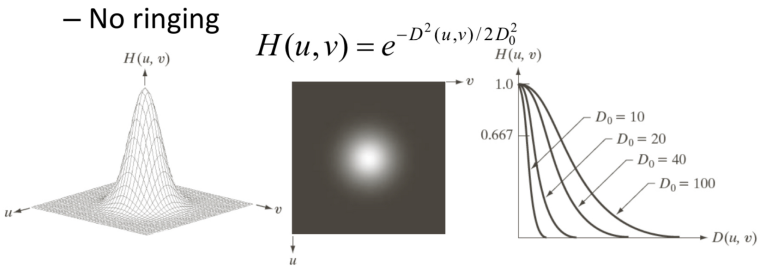


Figure 3.15: Gaussian filter in frequency

Gaussian filter is used for smoothing, for example in beautification process, as shown in Figure 3.16 or, for example, can be used for reducing noise.
Moreover, it can be used for filling gaps in images with low resolution. In Figure

Figure 3.16: Application of gaussian filter

3.17 a recap of lowpass filter

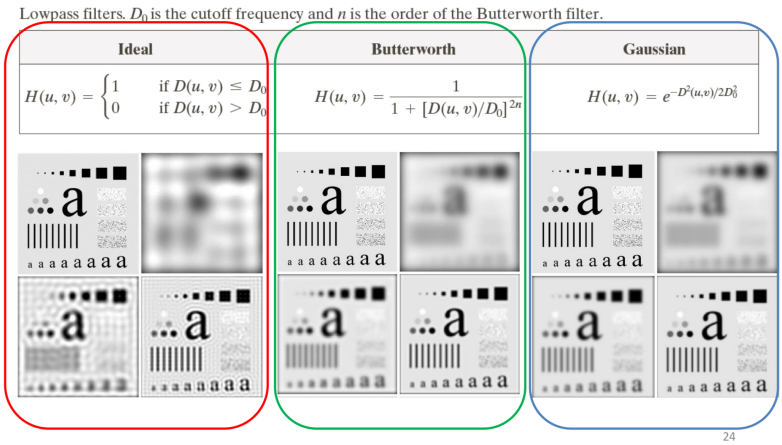


Figure 3.17: Recap of lowpass filter

3.3.2 Highpass filters

Highpass filters are used for sharpening, so for enhancing details in image. A HP filter can be obtained from the corresponding LP:

$$H_{HP}(u,v) = 1 - H_{LP}(u,v)$$

Things that we have said for LPF are valid for HPF, as shown in Figure 3.18. So

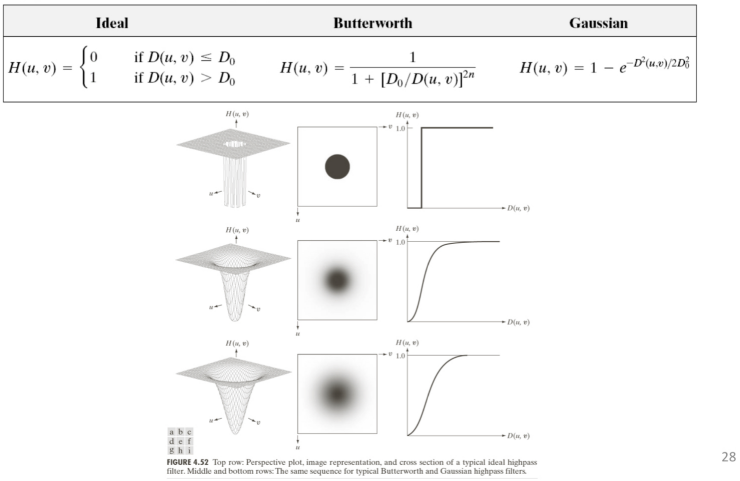


Figure 3.18: HP filters

all these filters depend on the cutoff frequency D_0, that is the distance between the central frequency (0 frequency). Dealing with HPF means that only high frequencies (components at distance greater than D_0 from the center) are maintained. Other components are set to be 0.
Obviously, talking about Ideal HPF, moving to the spatial domain, there is the problem of ringing artifacts. As we can see from Figure 3.19, these problem can

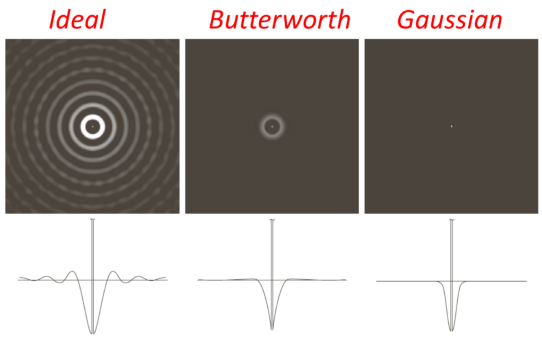
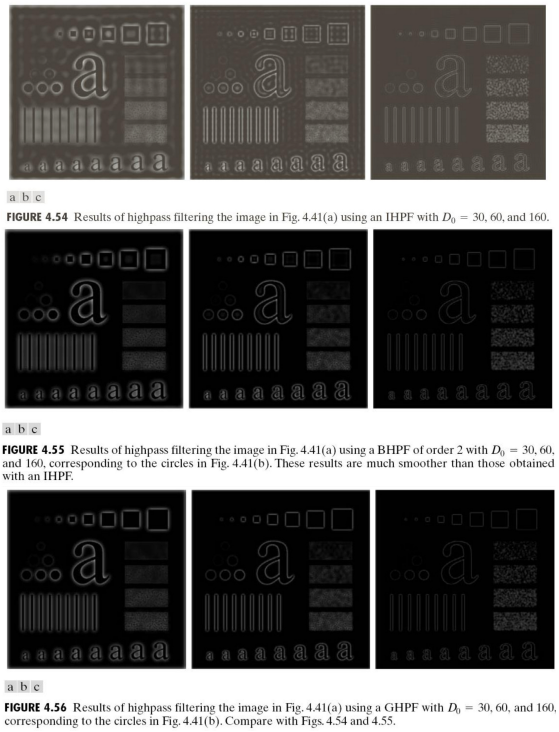


Figure 3.19: Different HPF in spatial domain

be adjusted using Butterworth or Gaussian filters.
In fact, as we can see from the first image of Figure 3.20, the ringing artifact causes very low quality images, also by using different cut-off frequencies.
In the other two images two different filters (BHPF and GHPF) are applied and results are noticeable.



a b c
FIGURE 4.54 Results of highpass filtering the image in Fig. 4.41(a) using an IHPF with $D_0 = 30$, 60, and 160.

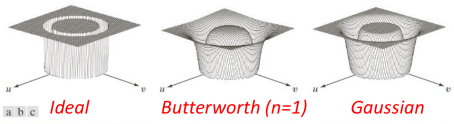
a b c
FIGURE 4.55 Results of highpass filtering the image in Fig. 4.41(a) using a BHPF of order 2 with $D_0 = 30$, 60, and 160, corresponding to the circles in Fig. 4.41(b). These results are much smoother than those obtained with an IHPF.

a b c
FIGURE 4.56 Results of highpass filtering the image in Fig. 4.41(a) using a GHPF with $D_0 = 30$, 60, and 160, corresponding to the circles in Fig. 4.41(b). Compare with Figs. 4.54 and 4.55.

Figure 3.20: Different HPF results

3.3.3 Selective filters

As we have seen, image can be affected by **noise** also in frequency domain, e.g. noise at specific frequencies.
Being in the frequency domain, we can operate on **selected frequency bands**, by using **band-pass filter** or **band-reject filter**.
Also here are valid same concepts illustrated for LPF and HPF and, as we can see from Figure 3.21, the representation in frequency of these filters is clear: are cutted frequencies inside a determinate region, defined by the parameter W, that is the width of the band.
If we want to be more specific, we could talk about **notch filter**, which are a type of band-reject filters with a narrow band.



a b c

FIGURE 5.15 From left to right, perspective plots of ideal, Butterworth (of order 1), and Gaussian bandreject filters.

TABLE 4.6
Bandreject filters. W is the width of the band, D is the distance $D(u, v)$ from the center of the filter, D_0 is the cutoff frequency, and n is the order of the Butterworth filter. We show D instead of $D(u, v)$ to simplify the notation in the table.

Ideal	Butterworth	Gaussian
$H(u,v) = \begin{cases} 0 & \text{if } D_0 - \frac{W}{2} \le D \le D_0 + \frac{W}{2} \\ 1 & \text{otherwise} \end{cases}$	$H(u,v) = \frac{1}{1 + \left[\frac{DW}{D^2 - D_0^2}\right]^{2n}}$	$H(u,v) = 1 - e^{-\left[\frac{D^2 - D_0^2}{DW}\right]^2}$

$$H_{bp}(u, v) = 1 - H_{br}(u, v)$$

Figure 3.21: Different HPF results

General observations: Several filters previously described perform operations that are similar to the filters in the space domain like smoothing or sharpening / edge highlighting. Which are advantages that bring us to filter in the frequency domain?

Filtering in the frequency domain allows for precise targeting of specific frequency components. This is particularly useful for tasks such as removing periodic noise, isolating specific frequency bands, or enhancing certain features. However, image is given in the spatial domain so we need to have spatial filter. What we do is to define in frequency the operation that we want to do and, by using the **inverse transformation**, we apply in the spatial domain the mask which implement operations defined in frequency.

3.4 Bilateral filter

Bilateral filter is a type of spatial filter that is adaptive, it's not the same across the image: it works independently depending on which part of the image we refer.
It's based on the gaussian filter but it preserves edge.
As we can see, in Figure 3.22, we can see that after the application of a generic Gaussian filter, the smoothed version is without edges, which are in the residual.

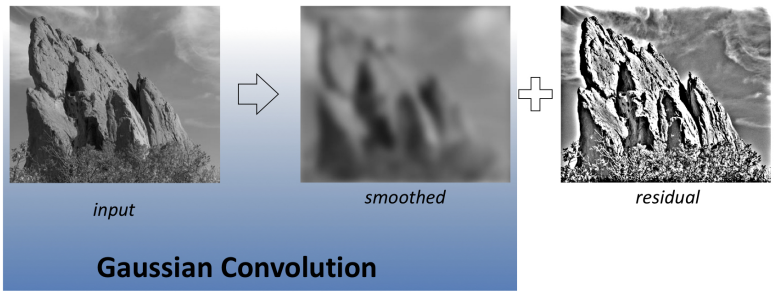


Figure 3.22: Gaussian filter smoothing

Instead, in Figure 3.23, we can see that by using bilateral filter, edges are preserved, with less information about fine details in the residual.

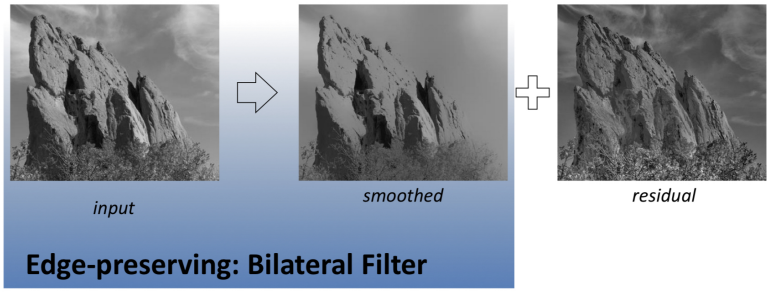


Figure 3.23: Bilateral filter smoothing

As we can see in Figure 3.24, the kernel with bilateral filter depends on the image content.
It's the weighted average of pixels with a new weight, with the equation illustrated in Figure 3.25.
The factor G_{σ_s} depends on geometrical distance and decreases the influence of distant pixels. The factor G_{σ_r} depends on the gray level distance and decreases

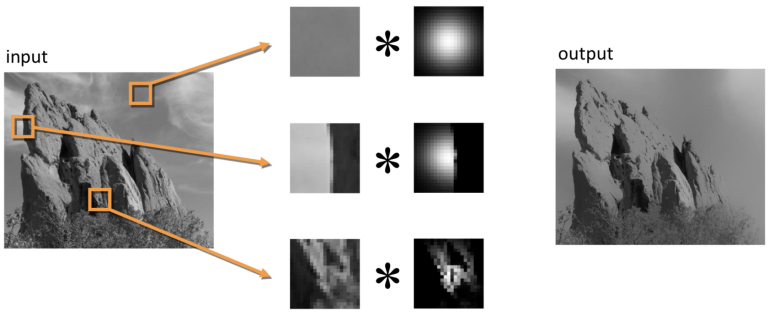


Figure 3.24: Kernel of bilateral filters

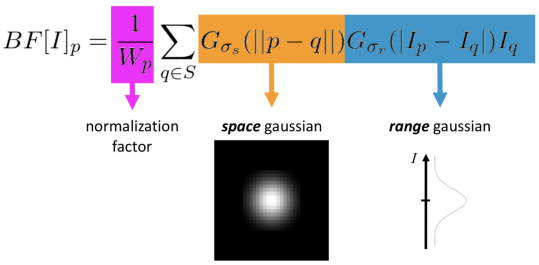


Figure 3.25: Equation of bilateral filters

the influence of pixels with different intensity values.
On notebook there are several images where different values of these two parameters are applied.

4

Edges and lines detections

We have just considered what is called **low level operations** or rather the image enhancement module. In this module we have seen methods to improve quality of an image.
Now we will focus on **mid level operations**, so algorithms based on more complex models, in order to do operations such that edge detection, line detection, texture analysis.
We have seen that derivative filters highlight edges in the image by highlighting edge pixels. However, filtered images lack the concept of edge.

4.1 Edges detection

Edges are caused by a variety of factors, as shown in Figure 4.1.

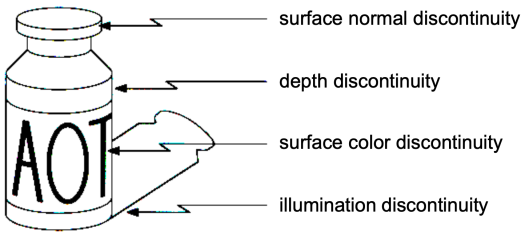


Figure 4.1: Image edges

There are several types of edges, as we can see from Figure 4.2
They correspond to different gradients: if there is a step, the edge is well defined,

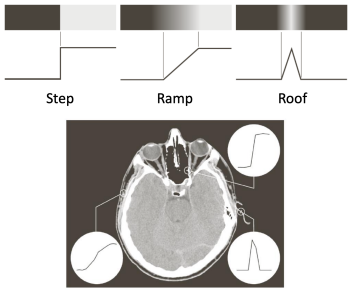


Figure 4.2: Different types of edges

if it is a ramp, there is a lower sudden variation, so it appears more smooth.
We already known several tools that can be useful to solve our problem as:

- Gradient
- Laplacian
- Smoothing filter

By focusing on **gradient**, we recall the formulation of the discrete gradient:

$$g_x = f(x+1, y) - f(x, y)$$

$$g_y = f(x, y+1) - f(x, y)$$

Remember that the gradient is a vector pointing in the direction of the fastest varying of dependent variable.
Since it is a vector we can compute **magnitude and phase**:

- The magnitude corresponds to the **edge strength** and it's computed as follows:
$$||\nabla f(x, y) = \sqrt{g_x^2 + g_y^2}$$
- The phase corresponds to the **fastest varying direction** and it's done by:
$$\alpha = \theta = \arctan(\frac{g_y}{g_x})$$
And it's important to notice that it depends on the magnitude: if the magnitude is high (or rather there is a strong variation) also the phase is high

This concept is well explained by Figure 4.3: we can see the gradient vector, defined by partial derivatives pointing in the fastest varying direction and, at distance α, the vector pointing in the edge direction.
We have already introduced masks for edge detection, computed by the Sobel

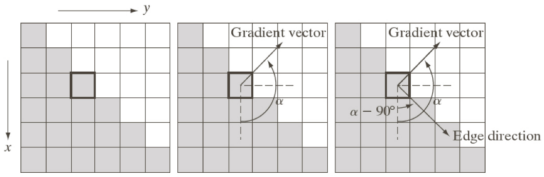


Figure 4.3: Magnitude and phase of a vector

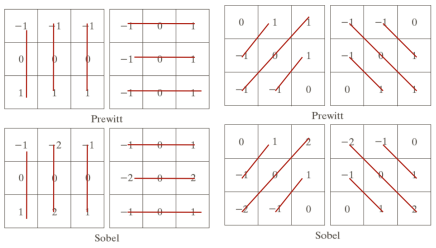


Figure 4.4: Sobel filters

operator that is the most widely used, since it can evaluate in four directions, as shown in Figure 4.4
By applying these different directional filters to an image, you can obtain edge maps that highlight edges oriented in specific directions.
Hence, in Figure 4.5 there is an example: considering the original image, we can apply different Sobel filters (different directions, but all are first derivative filters) with different results.

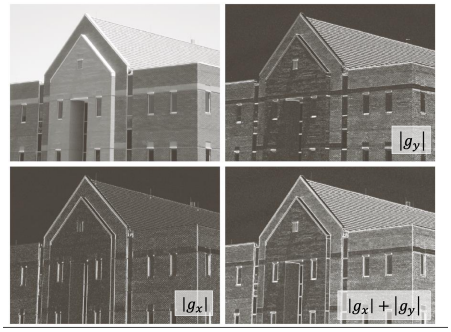


Figure 4.5: Magnitude after application of different Sobel filters

As we can see, g_y enhance horizontal edges (by considering different rows and not different columns), g_x vertical edges and the combination of these two enhance all the image.
Instead, if we compute the gradient angle on image, we obtain Figure 4.6: areas of constant intensity (so not edges) in this image indicate that the direction of the gradient vector is the same in all pixels of this region.

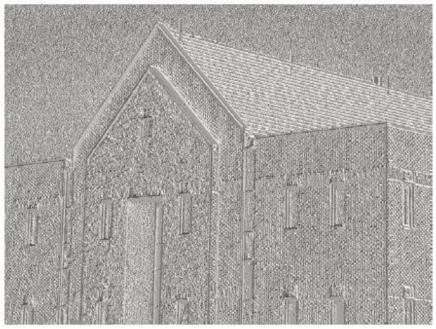


Figure 4.6: Angle of gradient of an image

If we consider **Laplacian filters**, recalling the equation:

$$\nabla^2 f(x,y) = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)$$

we remember that a laplacian filter is isotropic but we can derive from laplacian some **asintropic detectors** (different directions available), as shown in Figure 4.7

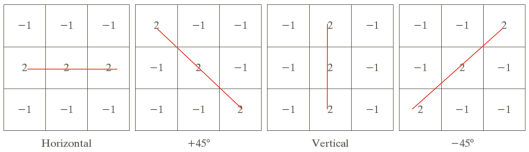


Figure 4.7: Different filters available

Derivative filters are afflicted by noise, as shown in Figure 4.8: the second order derivative is more afflicted by noise than the first order derivative.

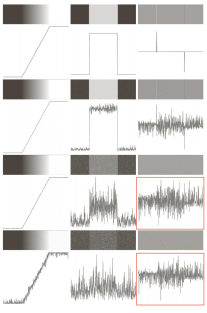


Figure 4.8: Noise in derivative filters

It's important to do a recap of first and second order derivatives: the first order derivative create a single edge but more thick, and it's moderate sensitive to

noise; instead, second order derivative, create double edges but more thin and it's high sensitive to noise.
What could be a possible algorithm for detecting edges?

- Apply a LPF in order to remove noise
- Apply a gradient / laplacian in order to enhance edges
- We apply a threshold in order to define edges

Edges of images are noisy and **smoothing** (LPF) is often used prior to calculation.
In Figure 4.9 a threshold applied two images, one smoothed and one no.

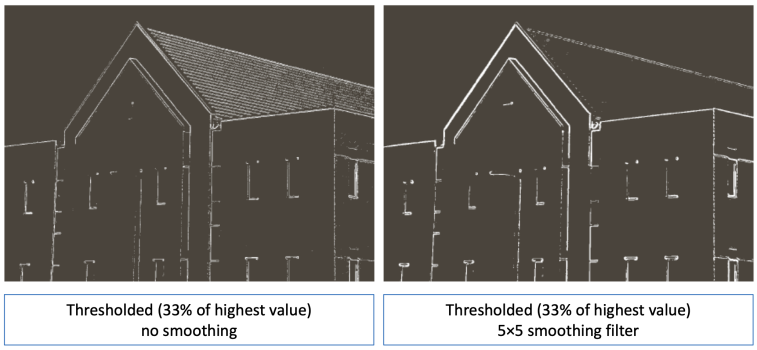


Figure 4.9: Threshold on smoothed image

4.1.1 Canny Edge Detector

The Canny algorithm that implements Canny edge detector, which is a filter addressing the following targets:

- Low error rate
- Edge points well localized
- Single edge point response

The Canny algorithm performs the following operations:

1. Smoothing with Gaussian filters
2. Gradient computation (magnitude and phase)
3. Quantize the gradient angles
4. Non-maxima suppression, that is an operations for thinning edges

5. Hysteresis thresholding

The first step is obviously the **smoothing**, in order to reduce noise.
Then the computation of the gradient is done in four directions: **vertical, horizontal and two diagonal masks**, as shown in Figure 4.10, where each mask provide different details.

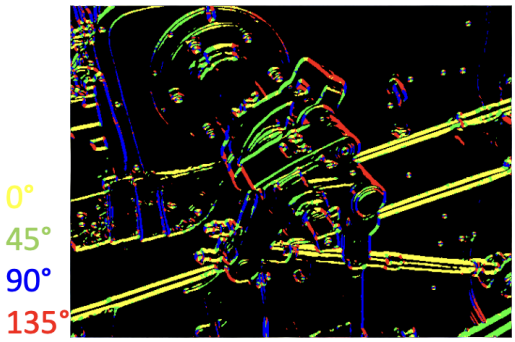


Figure 4.10: Different details provided by different mask

We use these them for computing the **gradient magnitude image** G, remembering that the magnitude is computed as:

$$G(i,j) = \sqrt{G_x(i,j)^2 + G_y(i,j)^2}$$

Moreover, for each pixel of G, we have also the direction of the gradient: The gradient direction $\theta(i,j)$ can be computed as:

$$\theta(i,j) = \arctan\left(\frac{G_y(i,j)}{G_x(i,j)}\right)$$

In the step three we have the **edge quantization**: the value of θ_k, direction of the gradient for each pixel of G, is quantized in precise "bins" of 45 deg: this quantization is often done by rounding the gradient direction to one of several predetermined angles, typically representing directions such as vertical, horizontal, and diagonal. This can help in order to reduce noise, obtain thinner edges ecc.
In Figure 4.11 an idea of this step.
The step four is **non maxima suppression**: it reduces the edge thickness since thin edges are desirable.

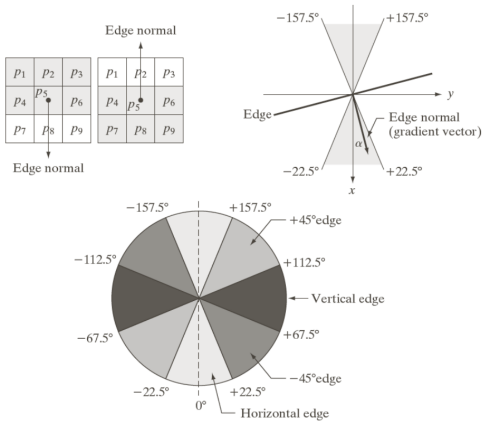
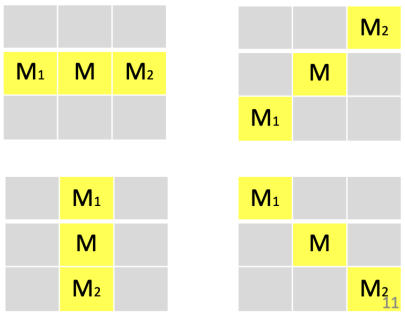


Figure 4.11: Quantization of angle

This process is done by crossing an edge and by selecting the strongest point: for each pixel in the image:

- Identify the two neighboring pixels along the gradient direction.
- Compare the gradient magnitude of the central pixel with the magnitudes of its two neighboring pixels, as shown



- If the magnitude of the central pixel is greater than the magnitudes of both its neighbors, preserve it. Otherwise, suppress the central pixel (set it to zero).
- Repeat this process for all pixels in the image, ensuring that the edges are thinned to a single-pixel width along the detected direction.

The last step is the **hysteresis thresholding** and it's used to keep strong edges and to reject isolated weak edges. The edge strength is measured by means of gradient magnitude.
We use two threshold T_L and T_H and we obtain two images: I_L and I_H.
Edges with intensity gradients higher than the high threshold are marked as

strong edges, while those below the low threshold are suppressed (considered non-edges). Edges with gradient values between the two thresholds are marked as weak edges.
The purpose of hysteresis thresholding is to track along the weak edges, and decide whether to keep or discard them based on their connectivity with strong edges. If a weak edge pixel is connected to a strong edge pixel, it's considered part of an edge, and thus, retained. If a weak edge pixel is not connected to any strong edge pixel, it is discarded. This helps in preserving true edges while eliminating noise and spurious responses.
The process is the one provided below:

- **Input**: Gradient magnitude image G and two thresholds T_{high} and T_{low}.
- **Classify pixels**:

$$\text{Strong edges: } G(i,j) > T_{high}$$
$$\text{Weak edges: } T_{low} \leq G(i,j) \leq T_{high}$$
$$\text{Non-edges: } G(i,j) < T_{low}$$

- Then it's created an edge map: firstly strong pixel as mapped as edge in the edge map; then all weak edges in the neighborhood of a strong pixel are mapped as edge, otherwise neglect it

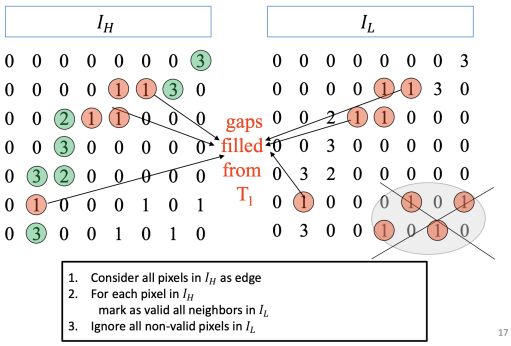


Figure 4.12: Hysteresis thresholding

4.2 Detecting lines

Another important task is the on referred to find **lines**. The aim is to find the equation of a line that cover a good quantity of line pixels.
Before start it's important to make a clarification: edges represent localized changes in pixel intensity and are detected based on gradient information, lines are more global geometric entities defined by sets of connected pixels that approximately follow straight paths. Both edges and lines are essential features in image processing and are utilized for various tasks, each serving different purposes based on their characteristics and definitions. A possible approach is to:

- Compute edges
- Consider all couples of edge points and evaluate the line passing through them
- Count the number of edge points on such line

The complexity is $O(n^3)$ since we have to compare $O(n^2)$ couples.
An alternative approach uses the **Hough transform**: consider the equation of lines passing through (x_i, y_i) edge point:

$$y_i = ax_i + b$$

where a, b are parameter which identify different lines passing through the point.
Now we consider the plane ab, where a, b are variables and x_i, y_i are parameter. Rewriting the equation as:

$$b = -x_i a + y_i$$

so x_i, y_i are parameters which identify a line passing through a, b. The ab-plane is called **parameter space**.
This equation is a line in the parameter space, so a single point in (x, y) space gives a line in (a, b) space. Taking another point x_j, y_j, it provides another line in the ab-space. If the two points in x, y space are on the same line, in the a, b the two corresponding lines, will intersect in a point.
So, all the points in the same line in x, y space will be intersect in one single point in a, b space, creating an **accumulation point**.

In Figure 4.13 the idea for this concept.

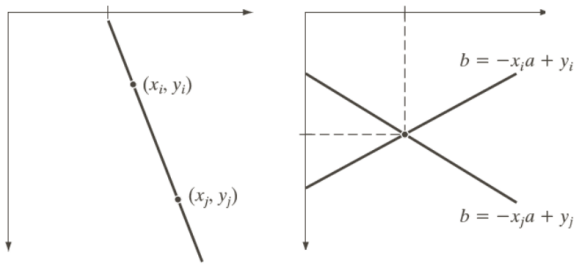


Figure 4.13: (x,y) to (a,b) plane

However, a problem occurs with vertical lines: since a for a vertical line is equal to infinity, we can't represent it into the ab-space: a solution is to use the **normal representation** defined below:

$$x\cos\theta + y\sin\theta = \rho$$

$$y = (-\frac{\cos\theta}{\sin\theta})x + \frac{\rho}{\sin\theta}$$

For each edge point (x_i, y_i), the line can be described as:

$$\rho = x_i\cos(\theta) + y_i\sin(\theta)$$

Here, θ and ρ are variables, and x_i, y_i are constants. This equation describes a **sinusoidal curve**in the (ρ, θ)-plane (parameter space).

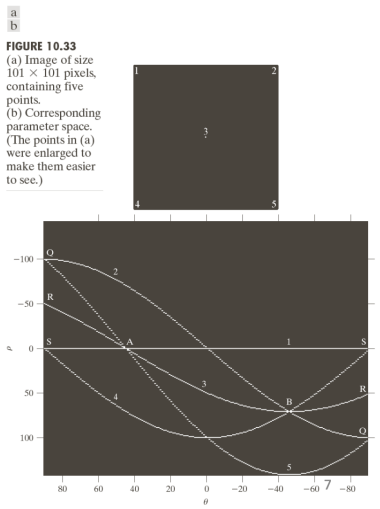


Figure 4.14: (x,y) to (ro,theta) plane

As we can see in Figure 4.15, each curve generated by a single point has a different shape but when two curves are intersected, in the Hough transform there is a **accumulation point** and means, in the x, y space, that these pixels are on the same line.

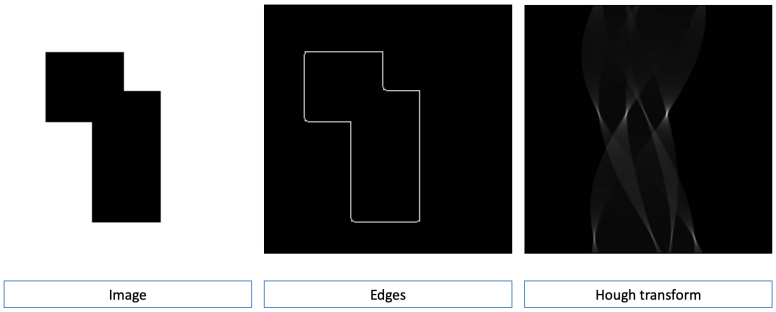


Figure 4.15: Example

The parameter space is quantized along ρ and θ as shown in Figure 4.16

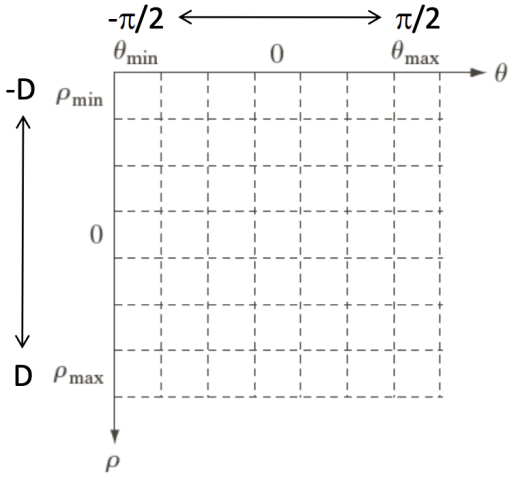


Figure 4.16: (x,y) to (a,b) plane

So, for each cell, that ranges between values of θ and ρ, we are interested about how many curves passes through that cell, so for values of θ and ρ in the range (more intersection, more points are in the same line in the (x, y) space).
If there are few cells, meaning bigger cells, also if points are not perfectly aligned in the x, y space, they pass in the same cell of ab-space. In other words, it has advantage since can recognize also pixels not perfectly aligned but it has a poor lines localization.
Instead, if there are many cells, it requires a precise alignment but it corresponds

to an accurate lines localization.
So, for each edge pixel, we compute the sinusoidal curve and we increment the corresponding cells for values of ρ, θ.
So, at the end, the counter of each cell is the number of pixels in that line, as shown in Figure 4.17

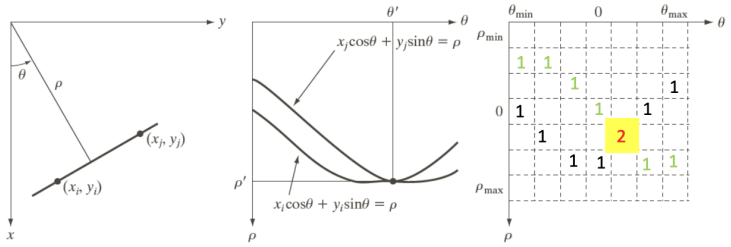


Figure 4.17: Hough transform

As said before, if cells are too big, there could be cases where edge pixels are not in the same line but are mapped in the same cell.
The Hough transform works for more complex shapes by changing the equation, with general equation below:

$$g(\vec{v}, \vec{c}) = 0$$

where $\vec{v}$ is a vector of coordinates and $\vec{c}$ a vector of coefficient.
For example, the equation of the circle is:

$$(x - c_1)^2 + (y - c_2)^2 = c_3^2$$

The parameter space might have high dimensionality.

4.2.1 Morphological Operators

These operators operate on the **shapes** found in an image, despite they are low level. Generally they works on binary images.
They operate basing on **set theory**: an image can be descripted based on set as follows:

- A set is a vector of tuples, each tuple representing the (x,y) coordinates of a point belonging to the set. For example the set of all white pixels in an image is a vector containing pixels (a pixel is a tuple) that has 255 as value of color.

It's possible to process an image working in this set based description, using an operator which can add or remove pixels from a set. So such operators modify the image **working on the shape**.
Erosion: Considering two sets, A and B, with B that is the **structuring element**, the erosion of A by B is a set defined as:

$$A \ominus B = \{z|(B)_z \subseteq A\}$$

so, in other words, we take a pixel (of A), we move in that pixel the structure element, and if there is even only one point not in A, we discard that pixel.
This method is used for thinning or separating weakly connected components.
In Figure 4.18 examples of structuring elements and an application of erosion

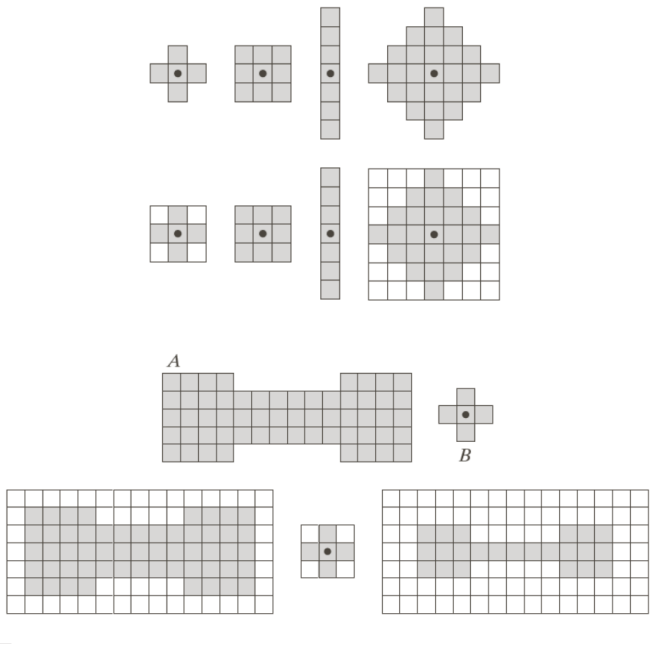


Figure 4.18: Above image: possible structuring elements, below image: results of erosion

Dilation: Considering two sets A and B, with B the structuring element, the dilation of A by B is a set defined as:

$$A \oplus B = \{z|(B)_z \cap A \neq \emptyset\}$$

so, in other words, we take a pixel (not forced of A), we move here the structure of B, and if there is at least one pixel overlapping with A, we take that pixel.

Applications could be for thickening or merging unconnected components.
In Figure 4.19 there is an example: the first one is done by putting the square B side by side with the edges: since there is a thin overlap, we add that pixel.

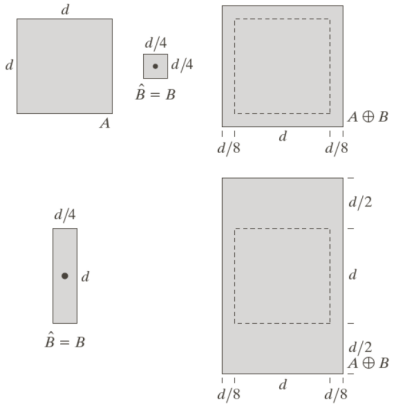
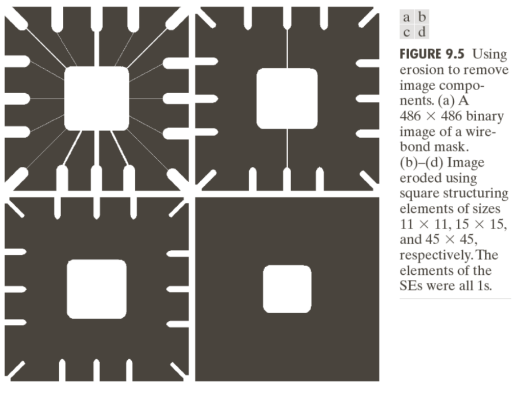


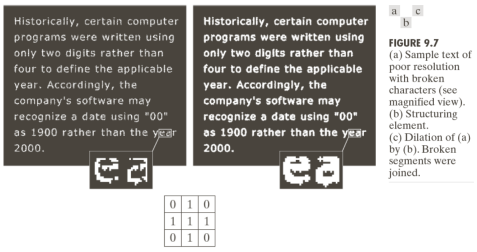
Figure 4.19: Example of dilation

In Figure 4.20, one example of erosion and one example of dilation



a b
c d

FIGURE 9.5 Using erosion to remove image components. (a) A 486 × 486 binary image of a wire-bond mask. (b)–(d) Image eroded using square structuring elements of sizes 11 × 11, 15 × 15, and 45 × 45, respectively. The elements of the SEs were all 1s.



a c
b

FIGURE 9.7 (a) Sample text of poor resolution with broken characters (see magnified view). (b) Structuring element. (c) Dilation of (a) by (b). Broken segments were joined.

Figure 4.20: Examples

The morphological operators can be concatenated:

- **Opening**: erosion + dilation

  $$A \circ B = (A \ominus B) \oplus B$$

  Effects are contour smoothing or eliminating thin protrusions without reducing the element size

- **Closing**: dilation + erosion

  $$A \cdot B = (A \oplus B) \ominus B$$

  Effects are fuse narrow breaks without increasing element size

More complex combinations may efficiently remove noise, as shown in Figure 4.14.

Figure 4.21: Playing with morph operators

5

Segmentation

Segmentation means partition an image into regions corresponding to different categories and it depends on the categories we have chosen.
It refers to the process of partitioning an image into multiple segments or regions based on certain criteria such as color, intensity, texture, or motion. The goal of segmentation is to divide an image into meaningful parts to facilitate analysis, interpretation, and understanding by computer algorithms.
By segmenting an image into distinct regions, computer vision algorithms can focus on specific areas of interest, leading to more accurate and efficient processing.
Subdivide an image into n regions $R_1, R_2, ...R_n$ such that:

- $\bigcup_{i=1}^{n} R_i = R$
- $R_i \cap R_j = \emptyset \; \forall i, j$
- Optionally: each region shall be connected

There are two main criteria used to segment an image:

- **Similarity-based**: groups pixels that share certain characteristics into the same region.
- **Discontinuity**: pixels in different regions should be different under a determinate measure function.

5.1 Segmentation by Thresholding

As said before, segmentation needs one or more criteria. Criteria may be defined on the histogram for similar pixels: for example, applying a threshold and selecting the two resulting segments. This approach can be extended to multiple thresholds/ranges, where we can obtain one segment for range.
These two different approaches are called **global and local thresholding**.
In global thresholding, a single threshold value is applied to the entire image, determined based on the image's histogram or statistical properties. All pixels are compared to this global threshold, assigning them to either the foreground (object) or background class. While effective under consistent lighting conditions, global thresholding may struggle with images featuring non-uniform illumination.
Conversely, local thresholding applies different threshold values to various regions or patches of the image. This adaptation allows for addressing local variations in intensity or texture. Typically, local thresholding involves sliding a window over the image, computing a threshold value for each window based on local statistics (such as mean or median intensity), and then thresholding pixels accordingly. This approach is advantageous for images with uneven illumination or varying contrast across different regions but may be computationally intensive.
As said before, there are two main criteria, similarity and discontinuity: for thresholding method, what is the criteria used?
It's a **discontinuity criteria**, since we look in the segment images based on the difference among grey-levels between pixels, as shown in Figure 5.1.

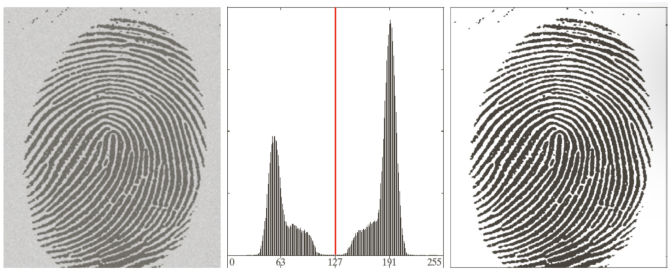


Figure 5.1: Example of segmenting by thresholding

However, selecting the right threshold may be trivial or tricky, since the threshold is **the only parameter** that we have to tune in this kind of segmentation.

Problems arises in:

- noisy image, as shown in Figure 5.2, since the histogram is smoother and it's more different to find a discontinuity in that histogram

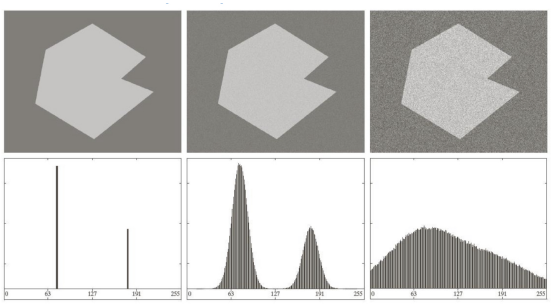


Figure 5.2: Problem which could occurs

- images with illumination changes
- Images with small regions what don't provide a real pattern in the histogram, since too few elements have different pixels-value

So thresholding is effective depending on some things illustrated before as distance between peaks, image noise, ecc.
We have understood that choosing the threshold can be difficult: we have always analyzed the histogram and choose an appropriate value. However, we can make this process automatic.

5.1.1 Otsu's method

The Otsu's method select a global threshold based on the histogram.
It assumes that two classes are created by thresholding and finds the optimal threshold by maximizing **inter-class** (between-class) **variance**, used as measure of the difference between the two classes. Moreover, the threshold also minimizes **intra-class variance**, so minimize the difference between pixels in the same class (region).
We compute the normalized histogram, by dividing the histogram for the total number of pixels, so for each gray value, the value in the new histogram it's probability. Obviously the sum is 1:

$$\sum_{i=0}^{L-1} p_i = 1$$

and we set a threshold $T(k) = k$ which divides the image into two class, setting a below threshold and and an above threshold, identifying respectively class C_1 and C_2.
We compute the probability $P_1(k), P_2(k)$ of the below thresold, that is the probability of a pixel being part of the threshold/class C_1:

$$P_1(k) = \sum_{i=0}^{k} p_i$$

that is the probability of being part of C_1. It's important to say that $P_1(k)$ refers also to the normalized value of the pixels in the below threshold (if we multiply this value by MN we obtain the number of pixels in the below class).
After that we compute the image global mean i.e. the average image intensity m_g:

$$m_g = \sum_{i=0}^{L-1} ip_i$$

and the **cumulative mean up to level** k:

$$m(k) = \sum_{i=0}^{k} ip_i$$

It's important to notice that the cumulative mean up to level k is normalized over the entire image: it means that if we want to refer it to the class C_1, we have to normalize for the number of pixels in C_1 i.e. $P_1(k)$, as shown below:

$$m_1(k) = \frac{1}{P_1(k)} \sum_{i=0}^{k} ip_i$$

But what is the difference between mean intensity value in class C_1, $m_1(k)$, and the cumulative mean of level k?
The mean intensity values is referred to the pixels of class C_1, while the cumulative mean is referred to all the pixels in the image.
Before proceeding, it's important to notice that:

$$P_1 + P_2 = 1$$

$$P_1 m_1 + P_2 m_2 = m_g$$

Now, after we have talked about mean, we can talk about **variance**:

$$\sigma_G^2 = \sum_{i=0}^{L-1}(i - m_G)^2 p_i$$

And we can define also the **inter-class variance**, between class, σ_B^2

$$\sigma_B^2(k) = P_1(m_1 - m_G)^2 + P_2(m_2 - m_G)^2$$

that is high if at least one of two contributions (m_1, m_2) is very different respect the mean of the image m_G. Notice that $\sigma_B^2(k)$ depends on k, since most of elements inside the equation depend on it.
The quality of the thresholding is defined by:

$$\eta = \frac{\sigma_B^2}{\sigma_G^2}$$

and depends only on σ_B^2, so on k.
By rewriting the equation for $\sigma_B^2(k)$ in a different way (need to be known), we can rewrite it as function of only functions depending on k (for $k = 0, 1, .., L-1$):

$$\sigma_B^2(k) = P_1 P_2(m_1 - m_2)^2$$

Moreover, a different formulation can be derived

$$\sigma_B^2(k) = \frac{(m_G P_1 - m)^2}{P_1(1 - P_1)}$$

and it's more efficient since only m and P_1 need to be computed for every value of k.
So, by maximizing η we can found the **optimal threshold**, or rather finding:

$$k^* \; s.t \; \sigma_B^2(k^*) = max_k(\sigma_B^2(k))$$

We can refer also to intra-class variance, that is:

$$\sigma_{in}^2 = P_1\sigma_1^2 + P_2\sigma_1^2$$

In Figure 5.3 an example of an image and diagrams of intra-class and inter-class functions, where the maximum of one corresponds to the minimum of the other.

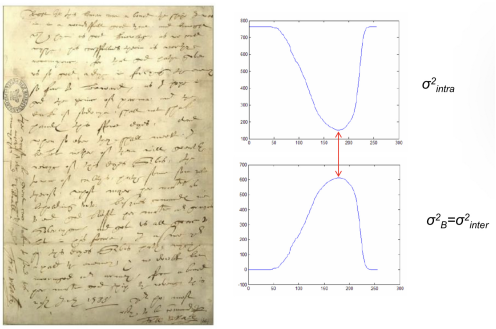


Figure 5.3: Intra and inter class diagram

In Figure 5.4 an example of segmenting by using Otsu method.

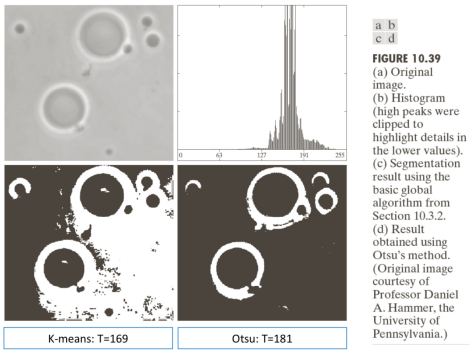


Figure 5.4: Example of segmentation with Otsu

Obviously, also here, by using smoothing with some filters, Otsu's method it's facilitated, as shown in Figure 5.5

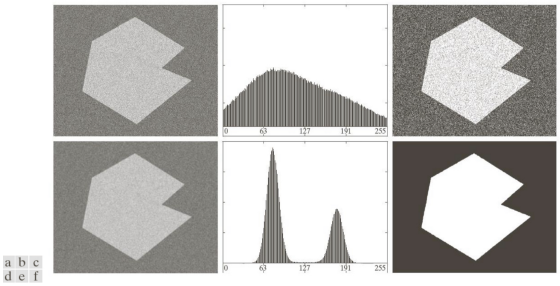


Figure 5.5: Smoothing before Otsu

However, despite the smoothing, thresholding could fail.

Moreover, the Otsu's method can be combined with other techniques as edge detection: for example, by finding the threshold on the edge image and applying it to the original image, as we can see in Figure 5.6

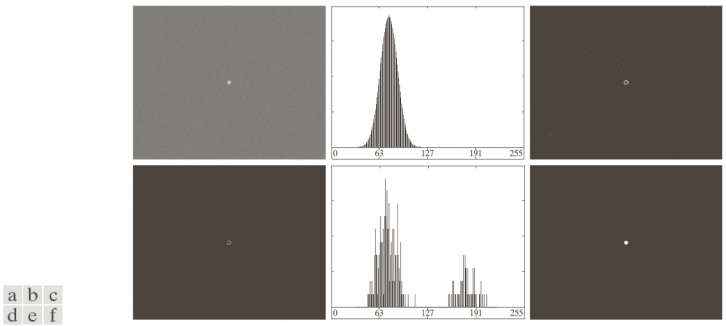


FIGURE 10.42 (a) Noisy image from Fig. 10.41(a) and (b) its histogram. (c) Gradient magnitude image thresholded at the 99.7 percentile. (d) Image formed as the product of (a) and (c). (e) Histogram of the nonzero pixels in the image in (d). (f) Result of segmenting image (a) with the Otsu threshold based on the histogram in (e). The threshold was 134, which is approximately midway between the peaks in this histogram.42

Figure 5.6: Combination of Otsu with other techniques

Otsu's method can be generalized to **non-global thresholding**, meaning process different regions of the image using different threshold, as shown in Figure 5.7: We subdivide the image in multiple regions, we maximize the inter-class

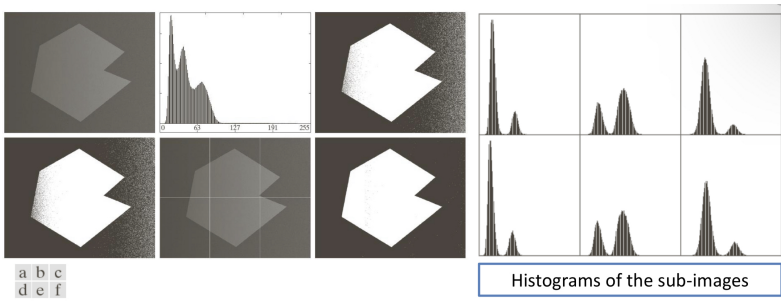


Figure 5.7: Otsu on subregions

variance of each sub-regions, using different threshold, and we merge all regions, as shown above.

5.2 Region growing and Watershed

5.2.1 Region growing

Threshold in an example of segmentation by using the discontinuity criterio: now we focus on a method, the **region growing**, that uses the opposite criterion, the segmentation for similarity, which focuses on grouping pixels that share similar characteristics into the same region.
This criterion ensures that pixels within a region are more alike to each other than to pixels in other regions.
The idea is to group pixels or subregions into larger regions.
The idea behind region growing is to start with from **seed points** to be defined, and iteratively, grow the region by adding neighboring pixels that meet specific **predefined merging criteria**, such as similarity in intensity, color, texture, or other image properties. This process continues until no more pixels can be added to the region or until certain stopping criteria are met.
Consider a basic region growing algorithm working on:

- the image $f(x, y)$
- the **seed array** $S(x, y)$ which contains $1s$ at the locations of seed points and 0 elsewhere. Obviously $f(x, y)$ and $S(x, y)$ they have same dimensions. Observe that the seed array is a mask, so it can have a precise shape
- a predicate Q for controlling growing

We start with the seed array $S(x, y)$, supposing that it's a mask obtained by the application of some derivative filter. The first step is to find **connected components** in $S(x, y)$ and erode each component until one pixel is left: by doing this, we move from a seed array $S(x, y)$ of many pixels in a suitable set of seed points.
Why do we move from a set of seed points to single point, one for each connected component? The reason is easy: we don't know if the set of seed points is obtained from a process different from the following merging criterion, so we use single points.
Then, the region growing algorithm create an image f_Q s.t $f_Q(x_i, y_i)$ is 1 if the predicate for that pixel is satisfied and then, it's created an image f_G: for each seed point in the reduced S, we append the $1s$ of f_Q in a neighborhood of 8-pixel of that seed point.

In f_G, each connected component is labeled with a different label. When a pixel is added to the image f_G, is also added to the set of pixels that need to be evaluated.
We can consider an example: we take an image of an X-ray and, for inizialization, we apply a threshold to highlight well bright areas and this is our starting $S(x, y)$.
After that, we erode each component until 1 point is left, as shown in Figure 5.8

Figure 5.8: Erosion of connected components in region growing

We select a predicate to grow the seeds: in our case 8-connected and "similar" pixels i.e. given a seed x_s, y_s and a threshold T, if the difference in the gray scale between the seed point and another pixel (in a neighborhood of 8) is less than T, we put 1 in that pixel:

$$Q(x_i, y_i) = |f(x_i, y_i) - f(x_s, y_s)| \leq T$$

and we add that pixel to the growing region. The newly added pixel now becomes part of the set of pixels to be evaluated in the next iteration (formally not part of $S(x, y)$). A support image f_Q may be created to evaluate the criterion faster.
The resulting image is that each welding defect is a single region and the output of an algorithm is a region (the same type of regions on which morphological operators work).

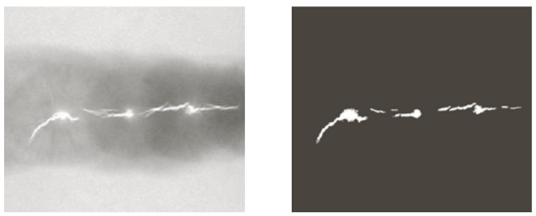


Figure 5.9: Results for the example of region growing

5.2.2 Wathershed

The Watershed algorithm is particularly well-suited for images where there is a clear distinction between objects and the background or between different objects.
It provides connected segmentation boundaries and connected components.
The base idea is that a grayscale image can be seen as a topographic surface, where intensity is an height value. In Figure 5.10 an example of this idea.

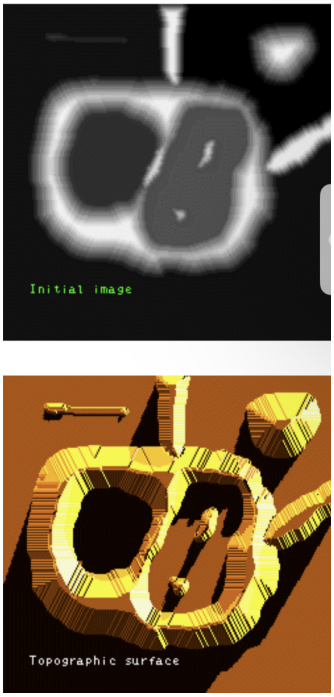


Figure 5.10: Image as topographic surface

There are three types of points:

- Local minima
- Points where if a drop of water would fall, it goes to a given minima
- Points where if a drop of water would fall, it goes into two or more different minima, and these are **watershed lines**

The goal of the algorithm is to find these watershed lines.
This is done by **flooding** (allagare) the surface from the minima (each flooded region is a catchment basin): we "open a tap" such that water could enter in the minima until water covers all surfaces.
When two "zones" merge, build a dam between than, taller than any pixel in the

image. The dam points define the watershed lines, or boundary of the segments. We can consider a merge as the application of subsequent dilations (morphological operators): dams created when dilation merges two components. In Figure 5.11 the result after the application of watershed algorithm.

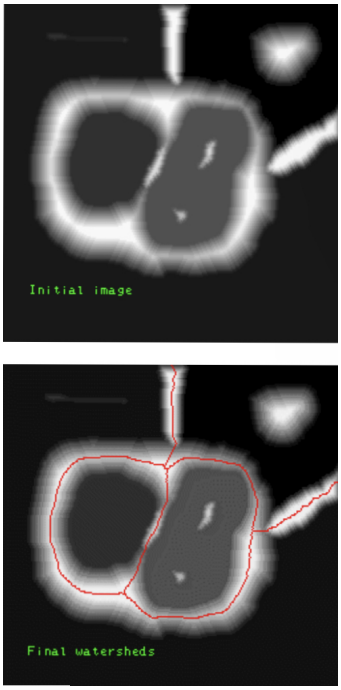


Figure 5.11: Otsu on subregions

Watershed algorithm can be used on the gradient to extract uniform regions, as shown in Figure 5.12

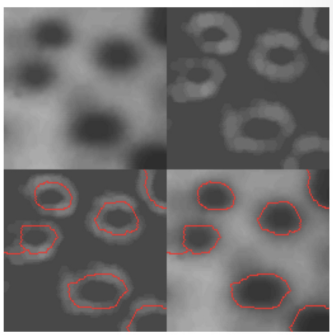


Figure 5.12: Watershed on small gradients image

It's useful when dealing with small gradients aka thick edges: catchment basins correspond to **homogeneous graylevel regions**.
However, direct application often leads to **oversegmentation** due to noise and

irregularities of the image: an enhancement could be to flood the topographic surface from a previously defined set of markers, which can drive the segmentation.

Internal markers are associated with objects of interest (local minima surrounded by lighter points) while external markers as background. The marker selection is performed with preprocessing (smoothing) and criteria definition. In Figure 5.13 an example that encapsulate these concepts.

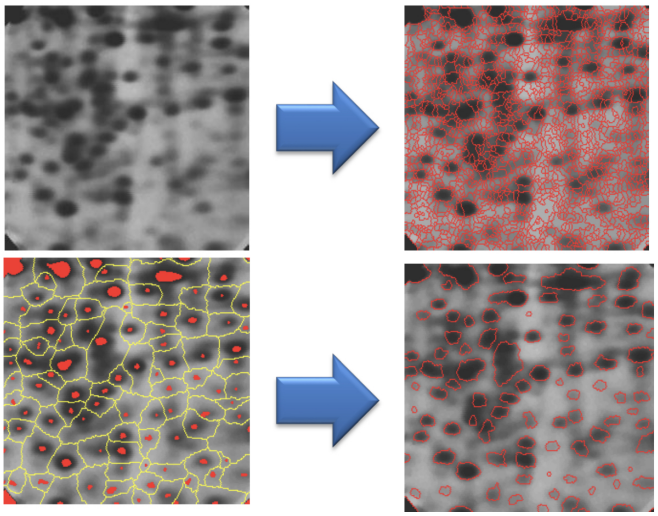


Figure 5.13: Watershed differences between normal and preprocessed image

5.3 Clustering

Clustering: the task of grouping a collection of heterogeneous elements into sets (clusters) of similar elements.

- How are elements described in the context of computer vision?
- What does similar mean?

We represent each pixel with a **feature vector**. This representation depends on the goal of the image analysis process we are implementing. The vector may contains all the measurements that may be relevant to describe a pixel

- Spatial position (coordinates)
- Intensity/brightness (grayscale images)
- Color information (RGB/YUV/CieLAB)

- Including a combination of the above

We now focus on segmentation by clustering, which in poor words provide the vector representation of each pixel and apply a suitable clustering algorithm, where pixels are grouped based on their vectors.
Clustering techniques often evaluate how similar two pixels are by comparing the corresponding feature vectors using a distance function. Some typical distance functions include, assuming D the dimension of the feature vector:

- **Absolute value/Manhattan**:
$$D_a(\mathbf{x}_i, \mathbf{x}_j) = \sum_{k=1}^{n} |x_{ik} - x_{jk}|$$
- **Euclidean**:
$$D_{\text{Euclidean}}(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2}$$
- **Minkowvski**:
$$D_{\text{Minkowski}}(\mathbf{x}_i, \mathbf{x}_j, p) = \left(\sum_{k=1}^{n} |x_{ik} - x_{jk}|^p \right)^{\frac{1}{p}}$$
that is a generalization of others distance function

There are two basic approaches to clustering:

1. **Divisive clustering**: it starts with the entire dataset considered as a cluster and recursively splits each cluster.
2. **Agglomerative clustering**: it starts with every single pixel considered as a cluster and recursively merges each cluster.

There is the need of some form of cluster quality measurement, also called **objective function**.
There are different clustering techniques available, and we analyze all.

5.3.1 K-means

K-means is a simple clustering algorithm based on a fixed number of clusters (k), that shall be provided to the algorithm. After the process, each feature vector, so each pixel, is associated to one feature vector of the k clusters.

- The k clusters are disjoint sets $C_1, C_2, .., C_k$, each with a **centroid** μ_i. There is an objective function which measures the distance between each data point and the centroid of its cluster and we want to minimize this function:

$$min(\sum_{i=1}^{k} \sum_{\vec{x} \in C_i} d(\vec{x}, \mu_i))$$

- The goal is to minimize the error made by approximating the points with the center of the cluster it belongs to, by selecting the best centroids.

However, exhaustive search is computationally unfeasible (too many combinations), hence we need an **heuristic approach**. **Iterative algorithm** are commonly used; these can be applied to vectors containing any set of features.
One of the most famous is **Lloyd's algorithm or k-means algorithm**.
The algorithm consists in:

- Get k initial centroids, one for each cluster, with the inizialization that can be performed by different techinqiues
- Associate each point to the **closest** centroid:

$$C_i = min\{\vec{x} : i = argmin_j d(\vec{x}, \mu_j)\}$$

- Compute the new centroids:

$$\mu_i = \frac{1}{C_i} \sum_{x \in \vec{C}_i} \vec{x}$$

- Repeat steps 2 and 3 until a **termination condition is achieved** (centroids don't sensibily move or is reached the max number of steps)

We have said that is needed an initialization method:

- **Forgy method**: k points randomly chosen among the data points, so centroids are part of the dataset
- **Random partition**: build the k cluster randomly, assigning all the points to clusters and after computing the centroids

Pros include being light and simple, with fast convergence, if heuristic is used. Cons include not guaranteeing optimality, dependency on initialization, and the requirement to know the number of clusters k in advance.
Another cons is **spherical symmetry** which implies that the clusters are assumed to be round or spherical in shape in the feature space. This means that the clustering algorithm considers all directions around the cluster's centroid (the center of the cluster) to be of equal importance and assumes that the data points

are evenly distributed around the centroid within a certain radius, creating a sphere.
The limitation of "forcing spherical symmetry of clusters" arises because real-world data may not always be neatly distributed in spherical shapes.
Data clusters can have elongated shapes, or they might be irregularly shaped, making spherical assumptions inadequate for accurately capturing the structure of the data.
K-means clustering can work considering the histogram (gray levels, faster) or pixel vectors (better results, tunable), with possible distance measures including intensity level difference (grayscale), color channel difference, or combinations of position, color, texture ecc.
In Figure 5.13 we can see how different distance function are applied: remember that, the segmented pixels, are the mean intensity/color of its cluster.

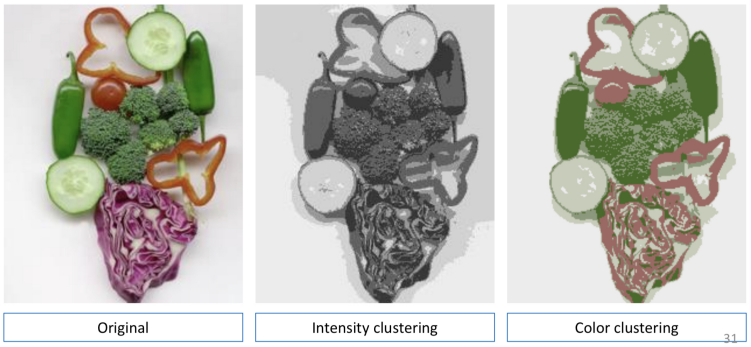


Figure 5.14: Clustering focusing on different distance functions

Instead, in Figure 5.14, we can see that considering color clustering, increasing k, it leads to better results.

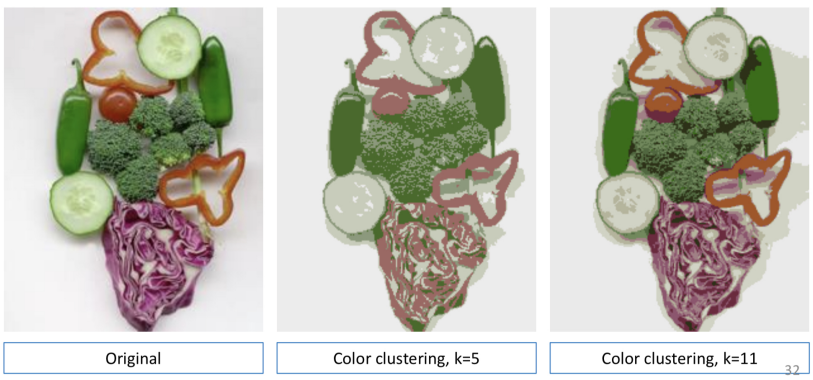


Figure 5.15: Color clustering focusing on different k values

Moreover, as we can see in Figure 5.15 that k-means clustering could have some problems: there are some clusters meaningless, some segmentation shown not necessarily components, and some shapes are difficult to be recognized.

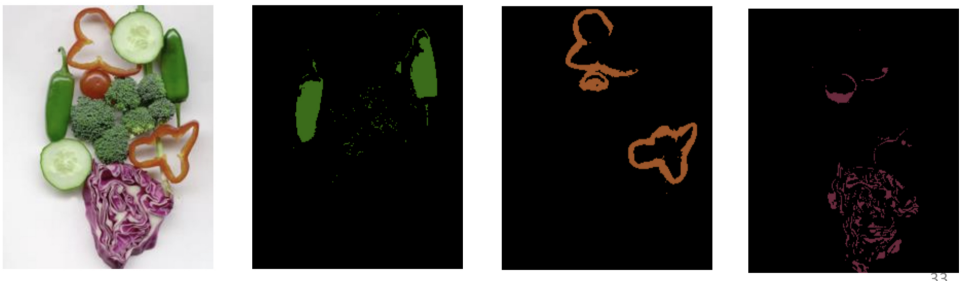


Figure 5.16: Problems arise with clustering

5.3.2 Density-based clustering

We have seen that k-means is good but have some cons that can't be ignored. We could derive a deeper analysis on the dataset: an idea is to create **density function** and look for the **modes** of that density function. Instead of forming clusters based on the shortest distance to the centroid (as in k-means), **density-based clustering** looks at the density of data points in a region. Clusters are defined as areas of high density separated by areas of low density.
At its core, the approach involves creating a density function that estimates how densely packed an area is with data points.
This density can be thought of as the number of points within a given area or volume.
A higher value of the density function in a certain region signifies a higher concentration of data points, which could indicate the presence of a cluster.
The modes of this density function correspond to the centers of clusters. In the context of a density function, a mode is essentially a peak or a high point in the distribution, indicating a region where data points are more densely packed together than in the surrounding areas. Identifying these modes helps in locating the clusters within the dataset.
How we can define a density function on an image? We have as starting point a **set of samples** and we want as output a **density function** (PDF).
A simple approach could be the kernel density estimation or Parzen window technique. Basically it's a convolution of data points with a given **kernel of**

radius r.
We can create a density function by:

- choosing a kernel, for example a gaussian function
- For each data point in the dataset, you center a kernel function on that point.
  This means that, for each sample, you're essentially placing a copy of the kernel function so that its center aligns with the sample's value.
- After placing a kernel on each sample, you sum up the contributions of these kernels across the data range to get a smooth estimate of the density function.
  This summation process blends the distributions centered on each sample, leading to a comprehensive density estimate for the entire dataset. The resulting function gives you a smooth curve that represents the estimated density of data points across different values. The idea can be seen in Figure 5.17

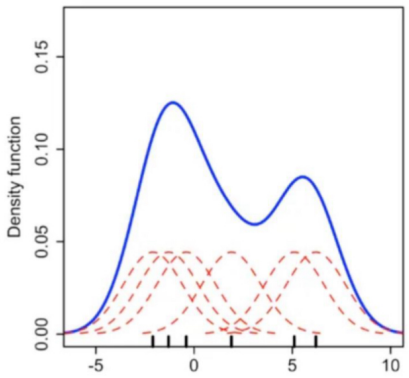


Figure 5.17: Density function estimation

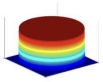
Defined for a point x in the n-dimensional feature space, the kernel function $K(x)$ is given by:

$$K(\vec{x}) = c_k k(||\frac{\vec{x}}{r}||^2)$$

The kernel $K(\cdot)$ integrates to 1, has radius r, and $k(\cdot)$ is a 1-dimensional profile generating the kernel, applied to all dimensions of the feature space.
In the context of KDE, $\vec{x}$ would typically be the distance between a data point and the point where the density estimate is being calculated. Several kernel may be used, as shown by Figure 5.18
The convolution with a kernel of width r is expressed as the sum of a translated

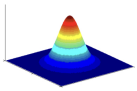
– Uniform kernel

$$k_U(x) = \begin{cases} c_u & \|x\| < 1 \\ 0 & \text{otherwise} \end{cases}$$



– Normal kernel

$$k_N = c_n \exp\left(-\frac{1}{2}\|x\|^2\right)$$



– Epanechnikov kernel

$$k_E(x) = \begin{cases} c_e(1 - \|x\|^2) & \|x\| < 1 \\ 0 & \text{otherwise} \end{cases}$$

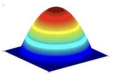


Figure 5.18: Kernel used for this task

kernel for each of N data points inside the radius r:

$$f(\vec{x}) = \frac{1}{Nr^n} \sum_{i=1}^{N} K(\vec{x} - \vec{x}_i) \tag{5.1}$$

Where $\vec{x}_i$ are the input samples and $K(\cdot)$ is the kernel function applied to each point inside the circle of radius r. The factor r^n normalizes by the number of dimensions.
Until now, we described how to derive a density function and we can then find major peaks (modes) and identifying regions of the input space that climb to the same peak (such regions belong to the same region/cluster).
However, creating a density function is computationally complex, possible but inefficient. So we want to investigate in alternative methods for finding peaks.

5.3.3 Mean Shift

Mean shift is a tool for finding stationary points (i.e., peaks) in high-dimensional data distribution without explicitly computing the distribution function, but it implicit models the distirbution using a smooth, continuous, non-parametric model.
Assuming data points are sampled from an underlying density function (PDF), direct PDF estimation can be approached by kernel functions as:

$$f(\vec{x}) = \sum_i c_i e^{-\frac{(\vec{x}-\mu_i)^2}{2\sigma_i^2}}$$

where some parameters like c_i,$\vec{\mu_i}$ need to be estimated.
Data points density is an implicitly description of PDF value, as shown in Figure 5.19: points with higher probabilities correspond to more dense cluster in data samples. The idea of mean shift is not to reconstruct the PDF but to evaluate

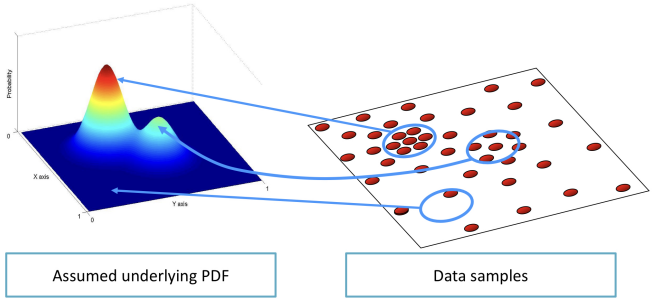


Figure 5.19: PDF vs data samples

agglomerate of point as high density, using **gradient which points to the best increase**.
In order to avoid the estimation of the entire PDF, it's useful to estimate **its gradient**.
Mean shift is a **"steepest ascend method"**, that refers to an optimization strategy used to find the maximum value of a function. It's a part of the broader family of gradient methods used in optimization problems, specifically for locating local maxima in a differentiable function, as shown in Figure 5.20

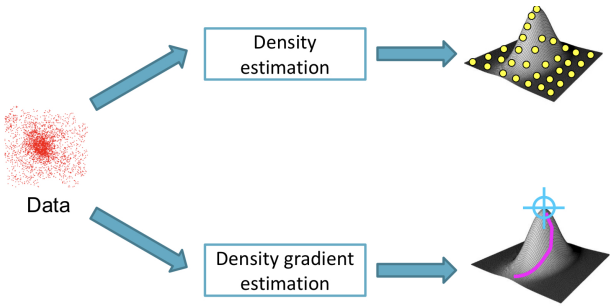


Figure 5.20: Density estimation vs gradient density estimation

We don't consider yet the kernel density estimation but the **kernel density gradient estimation**:

$$\nabla f(\vec{x}) = \frac{1}{Nr^n} \sum_{i=1}^{N} \nabla K(\vec{x} - \vec{x_i})$$

recalling that:

$$K(\vec{x} - \vec{x}_i) = c_k k(\frac{||\vec{x} - \vec{x}_i||^2}{r^2})$$

where $\nabla f(\vec{x})$ is the multiplication of the derivative of k, k' and the derivative of the inner function:

$$\frac{2}{r^2}||\vec{x} - \vec{x}_i||$$

. Defining the derivative of the inner as $y_i = \frac{2}{r^2}||\vec{x} - \vec{x}_i||$ and a new function:

$$g(a) = -k'(a)$$

where the minus is used to express $(\vec{x}_i - \vec{x})$ we can rewrite it as shown below

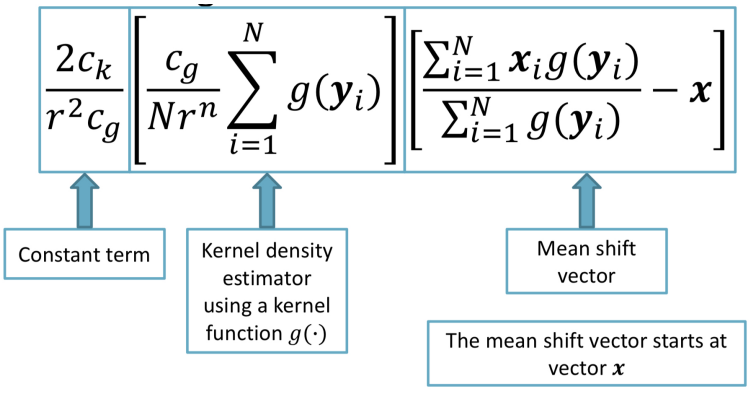
$$\nabla f(\boldsymbol{x}) = \frac{2c_k}{Nr^{n+2}} \sum_{i=1}^{N} [(\boldsymbol{x}_i - \boldsymbol{x}) \cdot g(\boldsymbol{y}_i)]$$

$$= \frac{2c_k}{Nr^{n+2}} \left(\sum_{i=1}^{N} [\boldsymbol{x}_i g(\boldsymbol{y}_i)] - \boldsymbol{x} \cdot \sum_{i=1}^{N} g(\boldsymbol{y}_i) \right)$$

$$= \frac{2c_k}{r^2 c_g} \left[\frac{c_g}{Nr^n} \sum_{i=1}^{N} g(\boldsymbol{y}_i) \right] \left[\frac{\sum_{i=1}^{N} \boldsymbol{x}_i g(\boldsymbol{y}_i)}{\sum_{i=1}^{N} g(\boldsymbol{y}_i)} - \boldsymbol{x} \right]$$

Where constant c_g normalizes the integral in the feature space when $g(\cdot)$ is used as a kernel.

This is one of the possible way of rearranging the equation but we choose this one since it's meaningful for mean shift. Focusing on the last expression we obtain:



We can rewrite it as:

$$\nabla f_k(\vec{x}) = A \cdot f_g(\vec{x}) \cdot m_g(\vec{x})$$

that can be rearranged as:

$$m_g(\vec{x}) = \frac{\nabla f_k(\vec{x})}{A \cdot f_g(\vec{x})}$$

And this shows that the mean shift vector proceeds in the direction of the gradient.
Instead, considering the geometrical formulation, we have the meaning in Figure 5.21

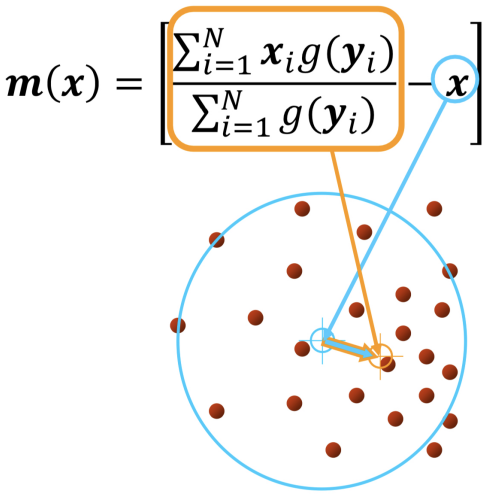


Figure 5.21: Geometric interpretation

The mean shift vector points where the increase in point density is maximum. So the mean shift vector is the direction to follow to find the mode and we can use an iterative procedure:

- Compute mean shift vector $\vec{m}(\vec{x})$
- Move the kernel window by $\vec{m}(\vec{x})$
- Stop when the gradient is 0, so when we encounter a maximum

We could get stuck in **saddle points** which have zero gradient (local minima): a small perturbation cause the process to move away in we are in a saddle point, since it's very unstable. So as last step we could perform a perturbation.
In Figure 5.22 the mean shift procedure
The mean shift vector dimension (how it is fast the computation) depends on

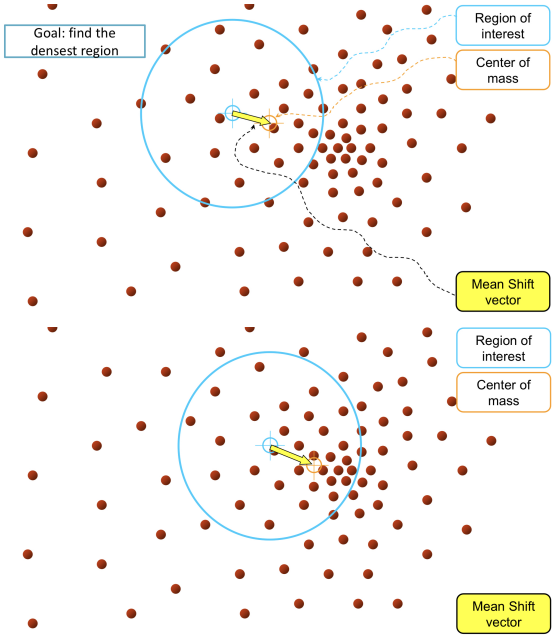


Figure 5.22: Mean shift procedure

the gradient. Obviously, steps are smaller towards a mode.
Moreover, there is an optimized method which consists of subdividing the space in windows and compute the method in **parallel**. The windows trajectories follow the steepest ascent directions, as shown in Figure 5.23

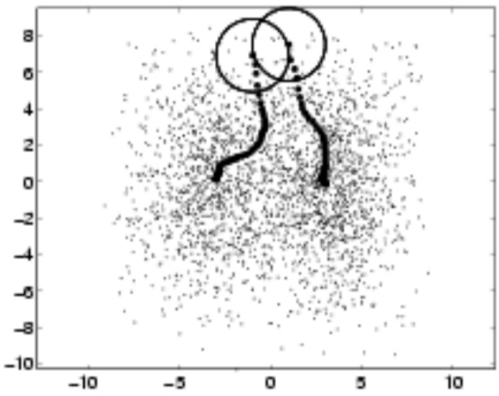


Figure 5.23: Optimized method using more kernels

There are some pros and cons:

- **Pros**:
  1. Doesn't assume clusters to be spherical
  2. One single parameter r

3. Robust to outliers and finds a variable number of modes

- **Cons**:
  1. Depends on window size that could occur problem if it's not selected well
  2. Computationally expensive

Mean shift clustering: In order to perform a clustering task using mean shift, we can use as clustering criterion the following.
We subdivide the space into windows and we run the algorithm in parallel.
We define the attraction basin (bacino di attrazione) as the region for which all window trajectories lead to the same mode. All these data points are merged into a cluster.

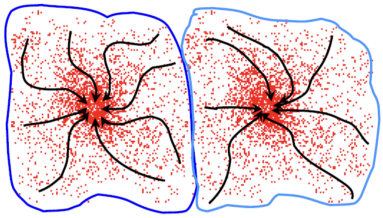


Figure 5.24: Clustering criterion using MS

In Figure 5.25 an example of mapping: we take an input, we plot in a three-dimensional space where we can analyze distributions and mode. At the end we can plot trajectories with corresponding peaks.

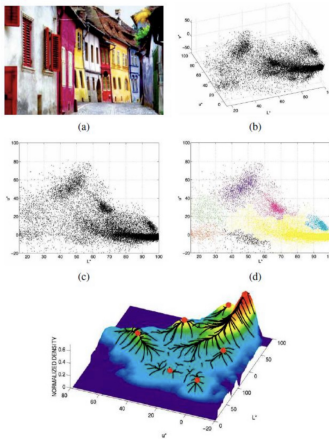


Figure 5.25: Example

Mean shift can effectively separate complex modal structures.

It's possible to preserve discontinuities by using a joint domain, where $\vec{x}$ is not a only feature vector but we use both **spatial and color** defining two vectors:

$$K(x) = c_k \cdot k_s(\frac{\vec{x_s}}{r_s}) \cdot k_r(\frac{\vec{x_r}}{r_r})$$

where $\vec{x_r}$ and $\vec{x_s}$ are respectively color and spatial coordinates.
In Figure 5.26 the effect on window size, with spatial range vertical and color range horizontal.

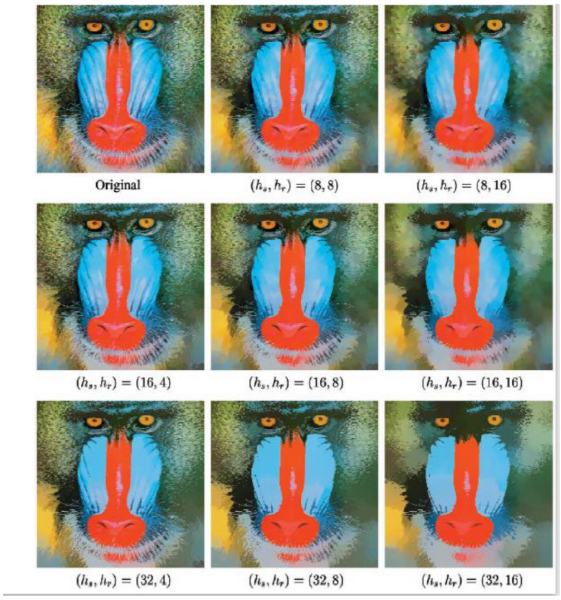


Figure 5.26: Example

6

Features

Considering images, how we could define elements or what is relevant in an image? Could we optimize that task?
Detecting and matching elements of the image is a **key task** in computer vision.
Elements are called **features**, that is a meaningful part of the image.
Features have two main components:

- **Feature detection**: finding a "stable" (easily detectable) point, a point that is worth being selected as a common point
- **Feature description**: a description of the surrounding area

So, given as input an image, as output we want a set of points plus a description of the region, as shown in Figure 6.1

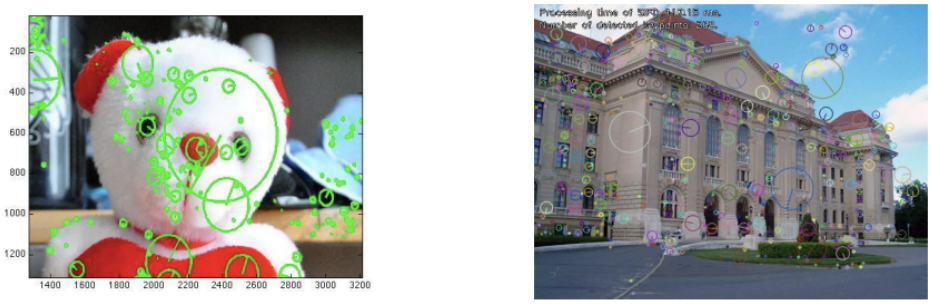


Figure 6.1: Features in an image

The ideal keypoints shall be:

- Stable and repeatable: stability is measured by means of a **repeatability score** on a pair of images. Given two images, it's defined as the ratio between the number of point-to-point that can be established (keypoints which match) and the min number of keypoints in the two images, as shown in Figure 6.2

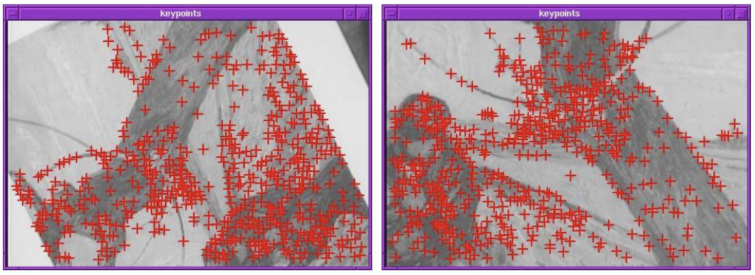


Figure 6.2: Keypoint stability in two images

- Invariant to transformations e.g. rotations
- Insensitive to illumination changes
- Accurate

The **descriptor** is a vector representation of the **local patch** (surrounding area), based on color, texture, orientation, etc. The ideal descriptor shall be robust to occlusion and clutter (occlusion in computer vision refers to the partial or complete obstruction of an object by another, complicating object detection and recognition. Clutter denotes a dense or complex scene that makes identifying specific objects challenging due to the distracting background or overlapping items), to noise, to blur, to compression, to discretization, stable and computationally efficient.

Feature matching is a key task in computer vision: it consists of evaluating features of two images and find similar features, where similarity is applied to descriptor using a distance function. In Figure 6.3 matching points in two images.

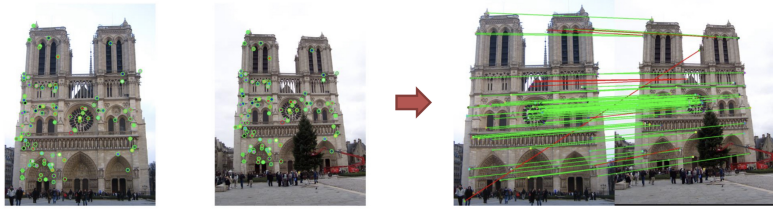


Figure 6.3: LALALA

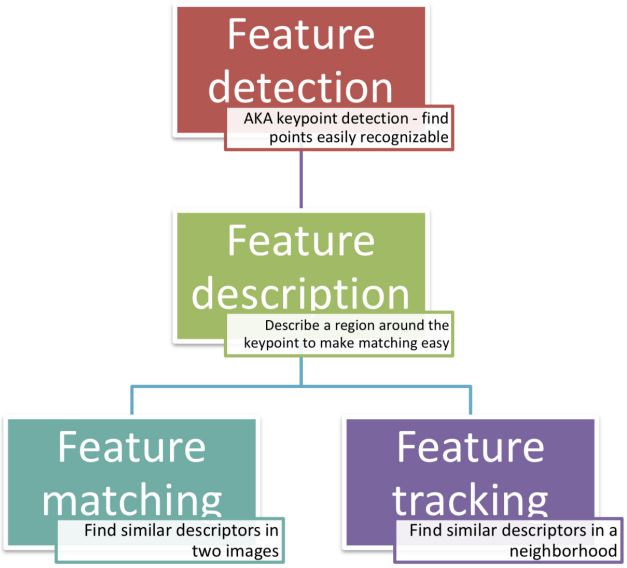


Figure 6.4: Feature pipeline

In Figure 6.4 the feature pipeline
There are several applications in which matching can be used as motion detection, stitching, 3D reconstruction ecc.

6.1 Detecting corners and blobs

Feature detection is also called **keypoint detection**, with the aim of finding points easily recognizable.
We need to start with a question: what are good points?
Uniform regions can be shifted where we want in the image but a corner point is more detectable as an edge point.
Salient points (keypoints) can be detected in many different ways and there are several algorithms existing.
One of the first and most famous is the **Harris corner detector**.
The intuition is: if we consider a patch (a small piece of the image) and its shifted version, if it is an uniform region, the two patches will be similar, if it is a salient point, the two patches will be different. A **corner is a region producing a large difference if the patch is moved**.
Considering a patch in a given position (x, y) and a displacement $(\Delta x, \Delta y)$, the similarity is measured by means of the **autocorrelation**, a function of the

displacement:

$$E(\Delta x, \Delta y) = \sum_i w(x_i, y_i)[I(x_i + \Delta x, y_i + \Delta y) - I(x_i, y_i)]^2$$

where I is the image, w a weight and i goes over all the pixels in the patch. It's the difference between shifted image and normal one.
If we apply Taylor series on E we get:

$$I(x_i + \Delta x, y_i + \Delta y) \approx I(x_i, y_i) + I_x \Delta x + I_y \Delta y$$

where I_x and I_y are partial derivatives of $I(x_i, y_i)$. This approximation holds for **small displacements**.
Substituting it into the auto-correlation function and neglecting the weights, we get:

$$E(\Delta x, \Delta y) = \sum_i [I_x \Delta x + I_y \Delta y]^2$$

that can be rewritten in matrix form as shown below

$$E(\Delta x, \Delta y) = \begin{bmatrix} \Delta x & \Delta y \end{bmatrix} \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

Auto-correlation matrix A

The auto-correlation matrix describes how the region changes for a small displacement. The matrix A is real and symmetric, and the eigenvectors (autovettori) are orthogonal and point to the directions of max data spread (data spread refers to the variance or distribution of pixel intensities (or other image features) across a certain region around a point in the image.)
The corresponding eigenvalues are proportional to the amount of data spread in the direction of the eigenvectors.

In Figure 6.5 the visualization of eigenvalues and eigenvectors

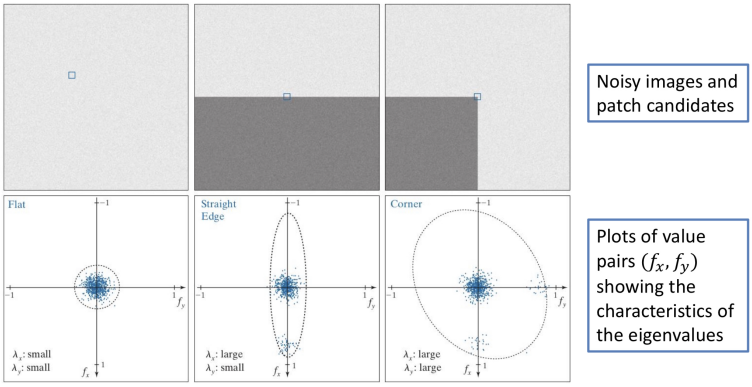


Figure 6.5: Eigenvalues and eigenvectors of A

Studying the eigenvalues we get information about the type of patch:

- if both eigenvalues are small, it's an uniform region
- only one large eigenvalue means an edge
- two large eigenvalues means a corner

The weights of the auto-correlation function were neglected but they can be introduced into our formulation, expressed by means of a **convolution**. There are two main choices:

- Box: 1 inside the patch, 0 outside
- Gaussian, more importance on changes around the center

So, given a matrix A and the eigenvalues/eigenvectors, how a keypoint is selected?
There are several options: assuming λ_0, λ_1 being eigenvalues, we have:

- Minimum eigenvalue *[Shi, Tomasi]*
- $R = det(A) - \alpha \cdot trace(A)^2 = \lambda_0 \cdot \lambda_1 - \alpha(\lambda_0 + \lambda_1)^2$. This define the **Harris corner**.
  Corners: Locations where R is significantly positive suggest that both λ_1 and λ_2 are large and $\lambda_1 \neq \lambda_2$, indicating a corner.
  Edges: If R is significantly negative, it indicates that one eigenvalue is large and the other is small, typical for edges.
  Flat regions: If R is small in absolute value, it suggests that both eigenvalues are small, indicating flat regions.
- $\lambda_0 - \alpha\lambda_1$ *[Triggs]*
- $\frac{\lambda_0\lambda_1}{\lambda_0+\lambda_1}$

Focusing on Harris corner detector is:

- Invariant to brightness offset $I(x, y) \hookrightarrow I(x, y) + c$
- Invariant to shift and rotations (corners maintain their shape)
- Not invariant to scaling

In Figure 6.6 we could see an example

Figure 6.6: Example of Harris corners algorithm applied

There are several other detectors, e.g. **USAN/SUSAN corner detector** which analyzes a circular window around the point. No derivatives involved and it's an edge+corner detector, very robus to noise.
USAN make a comparison between the nucleus (central point) and the pixels in the mask. So USAN is the portion of window with intensity difference from the nucleus (within a given threshold). As we can see from Figure 6.7, it individuates corners and edges.

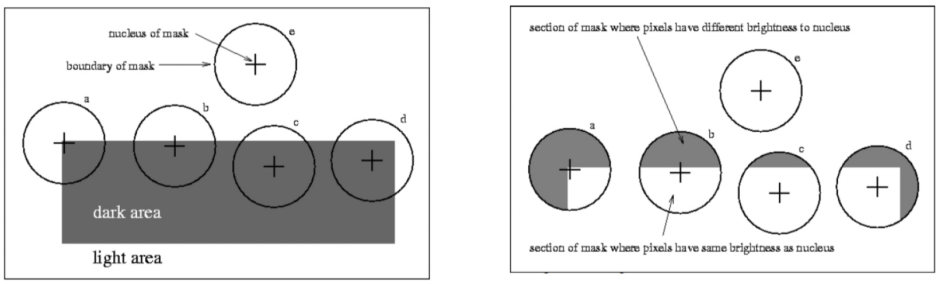


Figure 6.7: Usan

As we can see in Figure 6.8, depending on the USAN shape, we get different results.

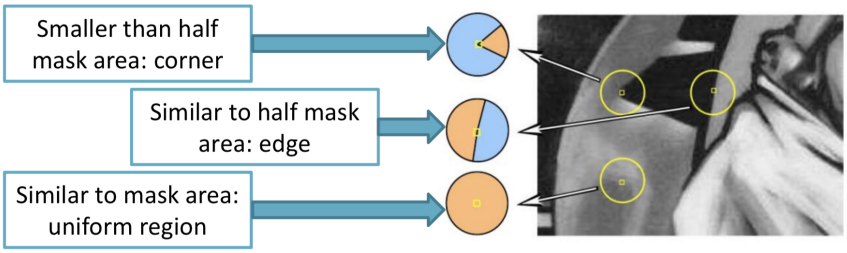


Figure 6.8: Different result based on USAN

SUSAN is simply the Smallest USAN. In Figure 6.9 how it works.

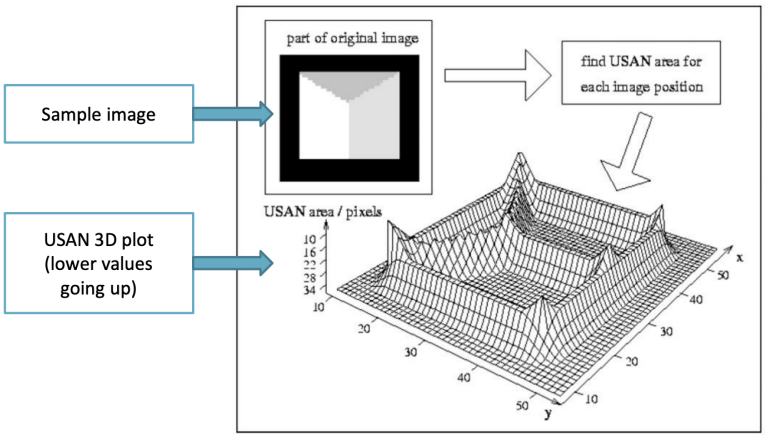


Figure 6.9: 3D plot of Usan

Harris corners focus on specific points, instead, other features focus on **blobs**: a blob is a region where properties taken into account are different from surrounding regions and are approximately constant inside the region.
MSER are connected areas characterized by almost uniform intensity, surrounded by contrasting background (blob). MSER feature detector can be used as a blob detector.
In Figure 6.10 a blob detector applied.
MSER is the acronym of Maximally Stable Extremal (refers to the property that all pixels inside the MSER have either higher (bright extremal regions) or lower (dark extremal regions) intensity than all the pixels on its outer boundary) Regions and the algorithm works as follow: apply a series of thresholds, compute

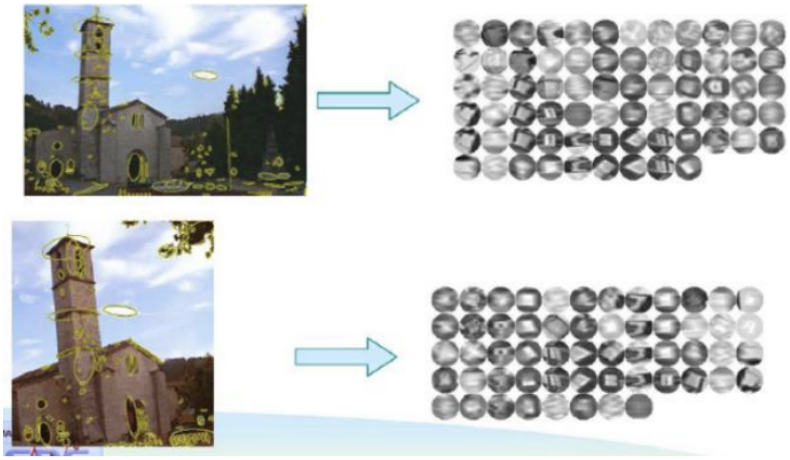


Figure 6.10: MSER as blob detector

the connected binary regions and some statistics for each region and at the end, analyze how persistent each blob is.

6.2 SIFT

SIFT is the acronym of **Scale Invariant Feature Transform** and it's a very reliable keypoint detector and descriptor. Retaking Figure 6.4, SIFT algorithm covers the first two boxes.
Scale refers to the size or resolution at which objects or features appear in an image. If we decrease the resolution, distinguish object become more difficult; moreover, also taking a picture at long distance could create some problems. These concepts are related to the concept of scale.
Scale captures the notion that the same object or feature can appear larger or smaller in an image depending on its distance from the camera or the resolution of the image.
Many computer vision algorithms aim to be "scale-invariant," meaning they can correctly interpret or classify features or objects in an image regardless of their size. Achieving scale invariance is a significant challenge because the visual appearance of objects can change with scale due to perspective, occlusion, and other factors.
SIFT features are:

- Local robust to occlusion
- Distinctive, can distinguish objects in large databases
- Dense, or rather many features can be founded even on small objects
- Efficiency, fast computation

6.2.1 Scale space and keypoint localization

The **SIFT algorithm** makes use of a **scale space**. Scale space is a concept and framework used in computer vision and image processing for representing an image at multiple scales, primarily to identify and analyze structures across different levels of detail, in order to obtain scale invariance.
It provides a systematic way to handle the scale variability of image features, which is fundamental for many applications, including feature detection, object recognition, and texture analysis.
Scale space represents an image in a continuous or discrete set of scales, essentially creating a family of derived images from the original by applying

smoothing filters with progressively larger scales (e.g., Gaussian blur) and lowering size dimensions. This process simulates viewing the image from different distances or resolutions.
The scale space is organized in octaves, as shown in Figure 6.11

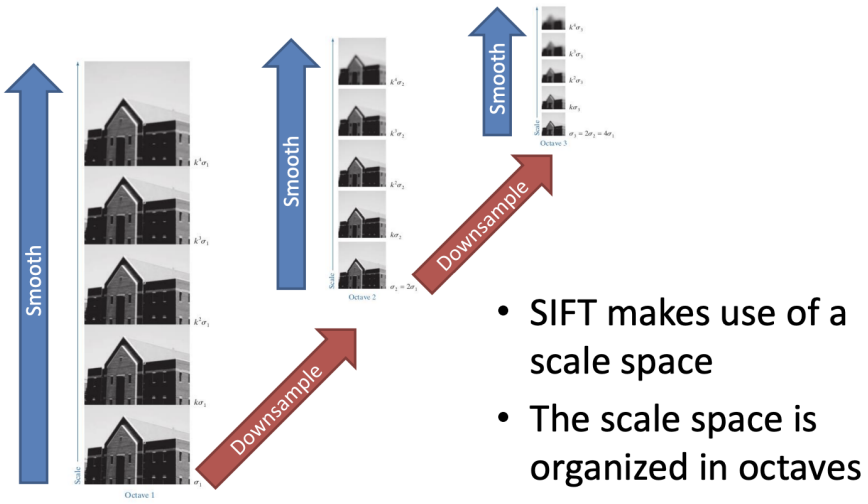


Figure 6.11: Scale space

Within an octave, from the first image to the last, it's increased the value of σ of the smoothing by multiplying it for a factor of k.
It's important to remember that by augmenting the value of the smoothing, details become less findable.
Then, from one octave to the next one, the σ of the gaussian smoothing is doubled respect to the starting image, not increasing k but simply using the same filter on a downsampled image.
Downsampling refers to the process of reducing the resolution of an image by decreasing the number of pixels it contains.
So we don't increase σ by multiplying by a factor k again but by passing from one octave to another, we downsample the starting image of the last octave, maintaining invariate σ.
The last image of one octave has the same smoothing as the first of the next octave.
This concept is carry out well by Figure 6.12, which represent this idea:
In a single octave we have s intervals (scales), so $s + 1$ images, with σ increased by a factor k at each interval. If we merge two octaves:

$$k^s\sigma = 2\sigma \hookrightarrow k = 2^{1/s}$$

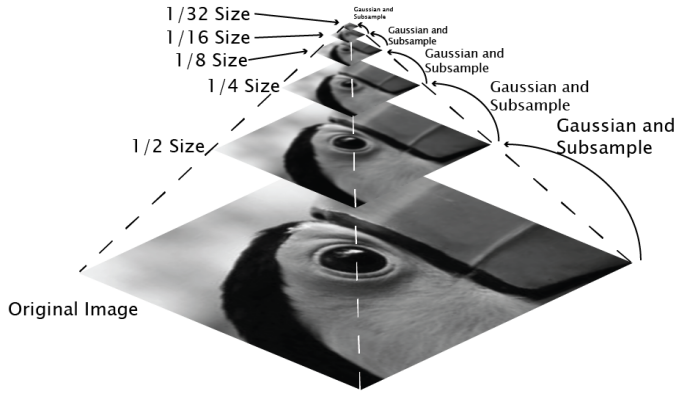


Figure 6.12: Idea behind scale space

so k depends on the number of intervals s.
A critical step in the SIFT algorithm is the creation of the **Difference of Gaussian (DoG)** images, which are used to detect and localize keypoints effectively. **Layers** are intervals within an octave and are combined by subtraction in order to form the DoG, as we can see from Figure 6.13

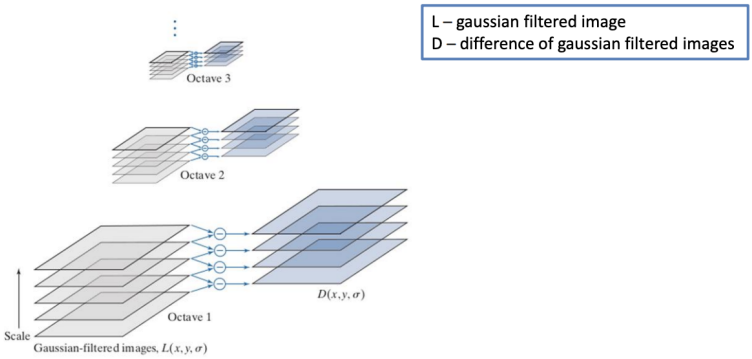


Figure 6.13: L and D images in a scale space

This requires two additional images in the scale space to process the first and last image.
Subtracting consecutive layers means: evaluate the difference between two smoothed images with same smoothing, different smoothing intensity.
The subtraction effectively filters out regions of constant intensity while highlighting regions of high spatial frequency (significant changes). This filtering is called precisely DoG and it's expressed in math form as $D(x, y, \sigma)$.
In Figure 6.14 the output images or rather the subtractions
What is the meaning of this filter? Going back on derivative filters: we have

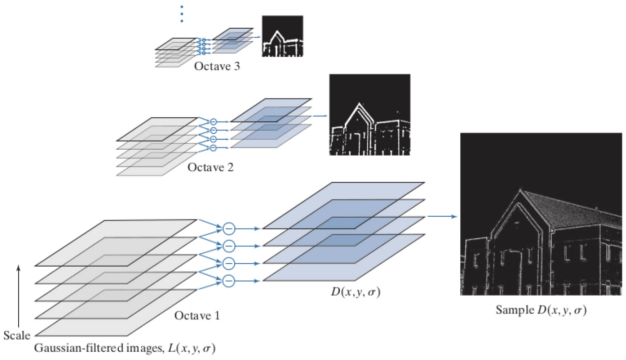


Figure 6.14: Result of DoG

observed many time the pattern smoothing + derivative filter. They can be combined in one filter, considering the derivative of the smoothing filter. This concept is exploited in the **Laplacian of Gaussian (LoG)** filter.
LoG is obtained using gaussian filter:

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

and considering its laplacian $\nabla^2 G(x, y)$ to filter the image. Filtering with LoG filter corresponds to filter the input using the Gaussian filter and compute the laplacian of the resulting image. In Figure 6.15 the LoG filter and its characteristics.

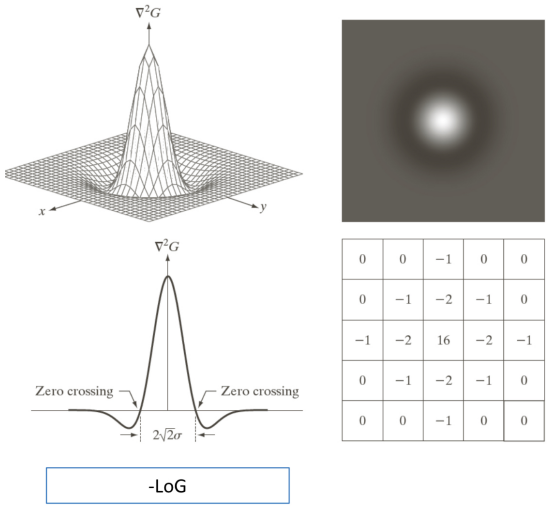
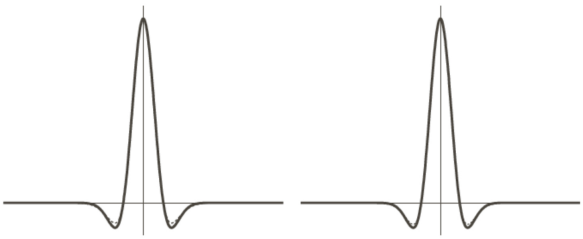


Figure 6.15: Characteristics of LoG filter

It is isotropic, zero crossing and noise remover at scales smaller than σ.
If we consider DoG, it has a very similar behavior of the LoG: in fact, it is an approximation of the first, as shown in Figure 6.16

$$D(x,y) = \frac{1}{2\pi\sigma_1^2} e^{-\frac{x^2+y^2}{2\sigma_1^2}} - \frac{1}{2\pi\sigma_2^2} e^{-\frac{x^2+y^2}{2\sigma_2^2}}$$



a b

FIGURE 10.23
(a) Negatives of the LoG (solid) and DoG (dotted) profiles using a standard deviation ratio of 1.75:1. (b) Profiles obtained using a ratio of 1.6:1.

Figure 6.16: DoG vs LoG

Back to the SIFT, scale space + substracting implement a DoG filter (which is a good approximation of LoG) where output images are used to find keypoint location.
In fact, maxima and minima are searched in the DoG images in order to find keypoint. As it is shown in Figure 6.17, these are three contiguous DoG, and we compare the black pixel with all pixels in blue zones.

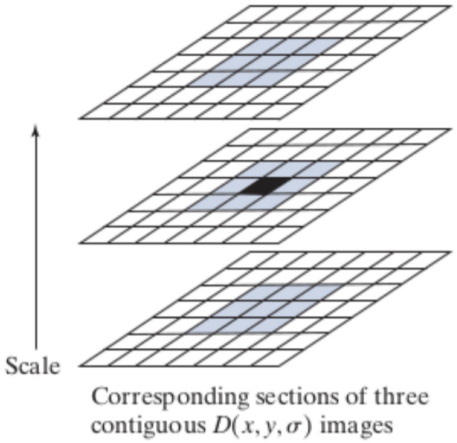


Figure 6.17: Seeking for local extrema

In each DoG image, each pixel is compared to its eight neighbors in the same image and its nine neighbors (since we don't consider it) in the next and previous

scales (layers). This is done for all scales, obtaining **scale independence**.
A pixel is selected as a keypoint candidate if it is either greater than all of its compared neighbors (a local maximum) or less than all of its compared neighbors (a local minimum).
The rationale behind using extrema from DoG images lies in their ability to signify points that are invariant to changes in scale and illumination. These points often represent significant changes in image gradients, which are indicative of potentially important features like edges, corners, and blobs.
Once potential keypoints are identified by finding local extrema, they are further refined to increase accuracy.
For example, we could use **subpixel location**: the location of each keypoint is refined to subpixel accuracy using a Taylor series expansion of the DoG scale-space function. This helps in achieving greater positional accuracy, which is crucial for high-precision tasks.
Moreover, with the task of refining the location of the keypoint, we use the Hessian matrix.
The Hessian matrix, in the context of image processing, is a square matrix of second-order partial derivatives of the image intensity function.
In SIFT, the derivatives are not calculated directly from the image but from the DoG images, which are smoother and more robust to noise. So, in Figure 6.18, the Hessian matrix of a DoG image D:

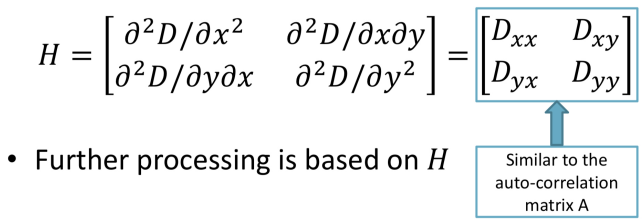


Figure 6.18: Hessian matrix of a DoG image

The Hessian matrix is similar to autocorrelation matrix A already seen.
The Hessian matrix H is used to discard keypoints with a low gradient:

$$|D(\hat{x})| < 0.03$$

and is used for discarding edge points, requiring that both eigenvalues of H are large, meaning that both curvature are high, so that there is a corner.
In Figure 6.19 a table similar to the one discuss in the previous section.

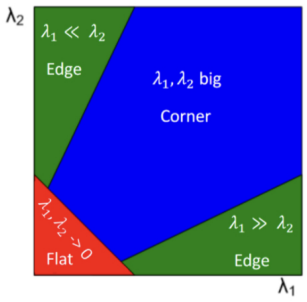


Figure 6.19: Results from eigenvalues

In order to obtain a better result, to get more point and to be more unstable, we test different values for s parameter.
Repeatability refers to the percentage of features that are detected in both images of a pair when these images are subjected to different transformations (such as those mentioned above).
A high repeatability rate indicates that an algorithm can reliably find the same physical points in the scene across different images. The output is a set of keypoints with associated scales, as shown in Figure 6.20

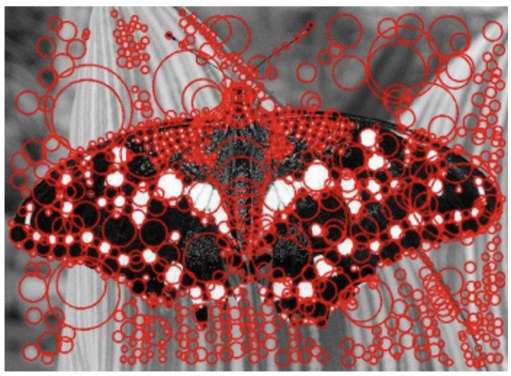


Figure 6.20: Output of keypoint localization

6.2.2 Keypoint orientation

Until now, we have dealt only with the concept of keypoint detection but it's important to talk with the concept of **keypoint orientation**.
Each keypoint comes with its scale (where the keypoint is found, determined by the sigma value and the octave): then it's selected the gaussian smoothed image

closest to that scale and we call it L. This process is **independent from the scale**. It's important to remember that we don't select the DOG image but the gaussian image.
Now, we take a neighborhood of the keypoint location, supposing a 15x15 square. For each pixel (x, y) in the neighborhood we compute the gradient magnitude and orientation (an angle, the direction of the strongest change in intensity), using image L.

$$M(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}\right)$$

Here, $L(x, y)$ refers to the pixel intensity in the Gaussian-smoothed image at the scale corresponding to the keypoint.
In order to achieve **rotation invariance** in the keypoint, the dominant local direction shall be found. It's builded the histogram of gradient orientations.
We create an histogram with a bin every 10 degrees. Then, for each pixel's gradient in the neighborhood, a contribution is made to the corresponding bin of the orientation histogram.
Each contribution is weighted, typically by the gradient magnitude.
We take the peak and we fit a parabola to the 3 bins close to the peak, in order to refine the peak location, we take that value, as shown in Figure 6.21.

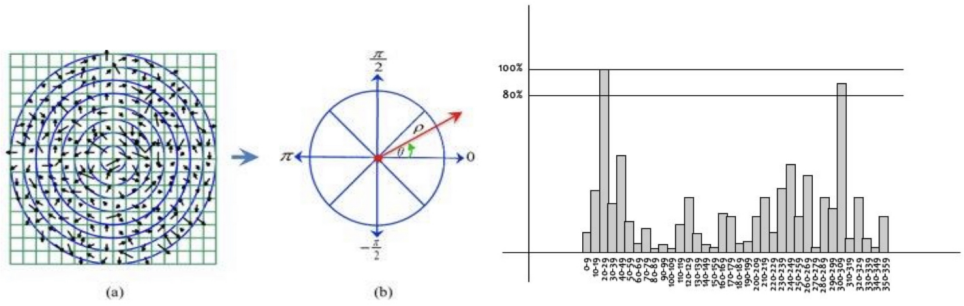


Figure 6.21: Keypoint orientation

Secondary peaks are also considered: we consider only peaks > 80% highest peak.
For each additional peak that is within 80% of the strength of the highest peak, a new keypoint is created. This new keypoint is identical in location and scale to the original keypoint but has an orientation corresponding to the secondary

peak. In Figure 6.23 we can see a source image with different keypoints detection, where arrows represent keypoint orientation.
Rememeber that the gradient is a vector that points in the direction of the greatest change of intensity.

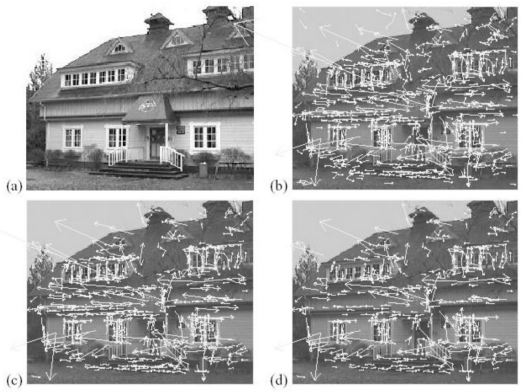


Figure 6.22: Example

6.2.3 Keypoint descriptor

At the end, we can evaluate the **descriptor of a keypoint**. A descriptor for a keypoint is essentially a compact representation of the image information surrounding that keypoint. The goal of the descriptor is to provide a robust means to compare keypoints between different images.
SIFT evaluates a descriptor for each keypoint in the image L, considering a neighborhood of 16x16 pixels that are rotated based on the measured keypoint orientation.
This means to rotate the coordinate system around the keypoint to align with this orientation.
By aligning the coordinate system with the keypoint's orientation, features that are captured in the descriptor are invariant to rotation of the image. This means if the same object is photographed from different angles, the keypoint descriptors will still match.
So we compute the gradient magnitude and direction for that neighborhood, assigning decreasing weight moving far from the keypoint.
Then, we consider sub-regions of 4x4 pixels, individuating 16 sub-regions and, for each sub-region, we compute an histogram of directions, with 8 bins of 45 degrees each and we save the values, as shown in Figure 6.23

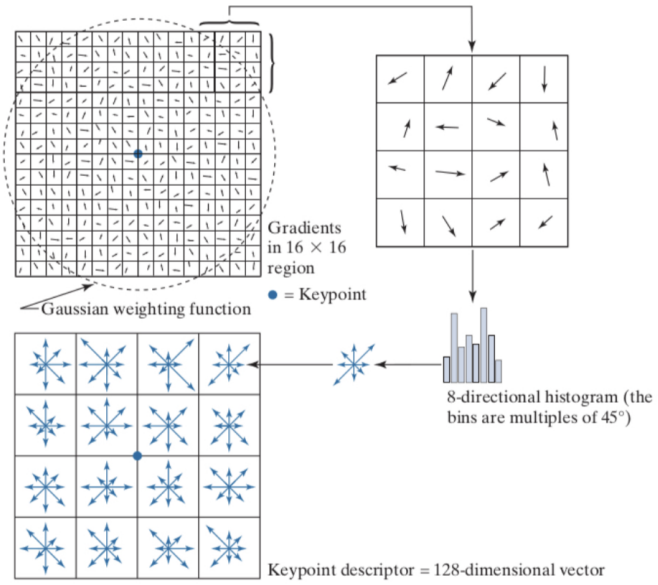


Figure 6.23: SIFT descriptor

So the descriptor size is composed by 16 regions with 8 histogram values for region, for a total of 128 elements, save in a single byte.
Finally, to ensure the descriptor robustness to changes in lighting or contrast, the descriptor vector is normalized. This step enhances the descriptor's resistance to variations in image brightness or exposure.
The descriptor components (each element of the histogram vector) are typically capped or thresholded to a certain value, often set at 0.2. This means any component of the descriptor that exceeds this value is set to 0.2. The purpose of this thresholding is to reduce the influence of very strong gradients which may dominate the descriptor's representation, potentially skewing the matching process. Strong gradients might not be representative of the keypoint's general environment, often being due to lighting effects or edges that do not necessarily correspond to distinctive features of the object.
The robustness of the SIFT features to noise factor was accurately measured since some images are taken and some random rotations are applied (incrementing the noise level) and we evaluate how much features are find in the rotated image. In Figure 6.24 results for 2% noise and 30 degrees rotation.

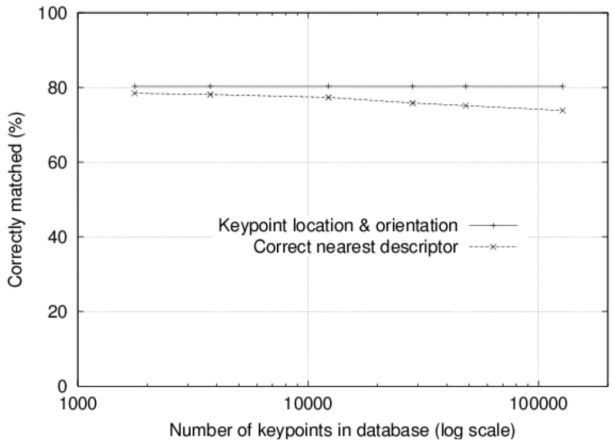


Figure 6.24: Result

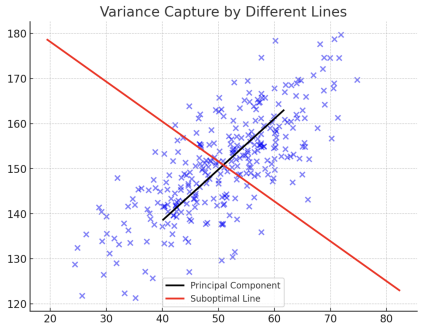
6.3 Other features

Until now that we have analyze SIFT feature in details, we now focus on other possibilities.
Going beyond SIFT, we can found PCA-SIFT, SURF and GLOH, which share roughly the same idea.

6.3.1 PCA

It's a dimensionality reduction technique where reducing dimensions can be useful to work on more compact representations, reduce computational workload and to highlight the most important information hiding the details.
So, to sum up, the idea of PCA is simple: reduce the number of variables of a data set, while preserving as much information as possible.
PCA, the acronym of **Principal Component Analysis** is an orthogonal projection of data onto a lower dimensional linear space.
Principal components are new variables that are constructed as linear combinations or mixtures of the initial variables. The idea is to squeeze information of variables into low dimension variables, iterating this process.
Organizing information in principal components will allow you to reduce dimensionality without losing much information, and this by discarding the components with low information and considering the remaining components as your new variables.
Geometrically speaking, **principal components** represent the directions of the

data that explain a maximal amount of variance, that is to say, the lines that capture most information of the data. The relationship between variance and information here, is that, the larger the variance carried by a line, the larger the dispersion of the data points along it, and the larger the dispersion along a line, the more information it has.
As we can see from Figure, the optimal lines go through the higher spread of data



So we want to find a linear space that

- Maximizes variance of projected data (purple line)
- Minimizes mean squared distance between original data points and their projection (blue line)

Supposing a dataset in $\mathbb{R}^2$, we have to do this operation for each variable, e.g. x_1, x_2.
The **first principal component** points in the direction of largest variance and each other principal component is orthogonal to the previous ones and points in the direction of largest variance of the residual subspace, as we can see in Figure 6.25

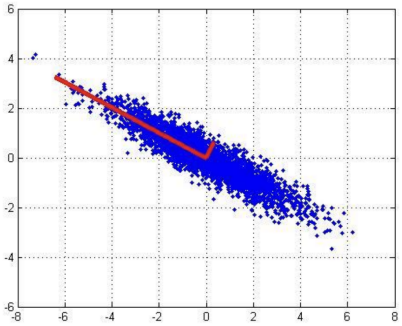


Figure 6.25: PCA

In **PCA-SIFT**, from step 1 to step 3 all is unvaried: we modify the step 4, that is the descriptor calculation.
We refer to a 41x41 patch at the given scale, centered around the keypoint, normalized to a canonical direction: we consider, instead of the weighted histograms, the concatenation of the horizontal and vertifcal gradients (39x39, since the bound are not computed) into a long vector (2 directions x 39x39) that is firstly normalized and then reduced using PCA.

6.3.2 SURF

SURF is the acronym of **Speeded Up Robust Features**, which speed-up computations by fast approximation of hessian matrix and descriptors using integral images.
SURF uses the **integral image** $I_\Sigma(x, y)$ that is an image of the same size of the original image but each pixel doesn't represent the gray-level but the sum of all pixels in the rectangle between the origin and that pixel:

$$I_\Sigma(x_i, y_i) = \sum_{i=0}^{x_i-1} \sum_{j=0}^{y_j-1} I(i, j)$$

as shown in Figure 6.28.

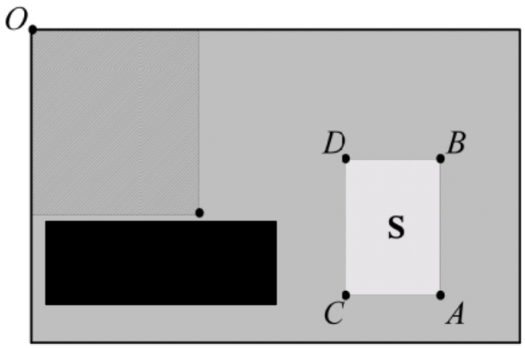


Figure 6.26: Integral image

It's useful since we can compute rapidly the sum of intensity values in a rectangle of the image. This means that for instance, if I want to sum elements in S, I can sum A and D and then subtract B and C.
Given the integral image, we can evaluate in a fast way the sum of the pixel in any given area of the original image with small computation.
Before going on with SURF it's important to deal with **box filters**: box filters are rectangular filters, composed by black and white rectangles. White boxes have positive weights and black boxes have negative weights. With box filter, as shown in Figure 6.27, we can approximate a Laplacian filter and, thanks to integral image, this approximation is fast to compute.
We have seen that in SIFT, for keypoint detection, we have used DoG and the computation of Hessian matrix.
SURF uses the **determinant of the Hessian matrix** for selecting the location and

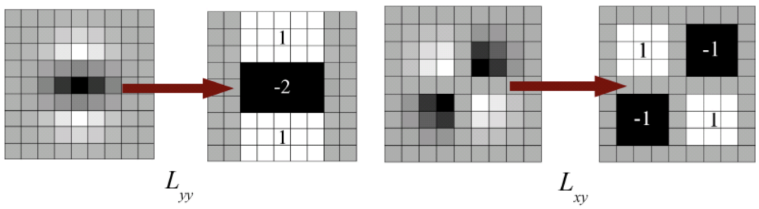


Figure 6.27: Box filters

the scale of keypoint. The $9x9$ box filters in the above images are approximations for Gaussian second order derivatives with $\sigma = 1.2$.
So, for keypoint detection, we compute the Hessian matrix, approximating the second order derivative of L using box filters, (the tradeoff is a fast computation) and we look for the maximum of the determinant (point where there is a strong intensity change).
If in SIFT images are smoothed with a gaussian and subsampled to achieve higher level of pyramid, so different scales, SURF does not have to iteratively apply the same filter to the output of a previously filtered layer.
Instead, it can simply apply box filters of increasing size in order to obtain a different scale. Therefore, the scale space is analyzed by up-scaling the filter size rather than iteratively reducing the image size.
Using integral image, computation doesn't depend on filter size.
In Figure 6.28 an image that represent this idea.

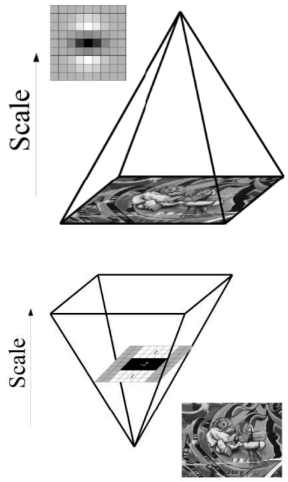


Figure 6.28: Pyramid of scale space

As for SIFT, before talking about descriptor, we need to find the **orientation** of our keypoint based on information from a circular region around the keypoint.

In order to find the orientation, we consider a circular neighborhood of radius $6s$, whit s the scale where the point was detected. Then we evaluate gradients in x and y, focusing precisely on the orientation, not computing precisely the gradient (too slow), by using the **Haar wavelents**, that are the two filters in Figure 6.29.

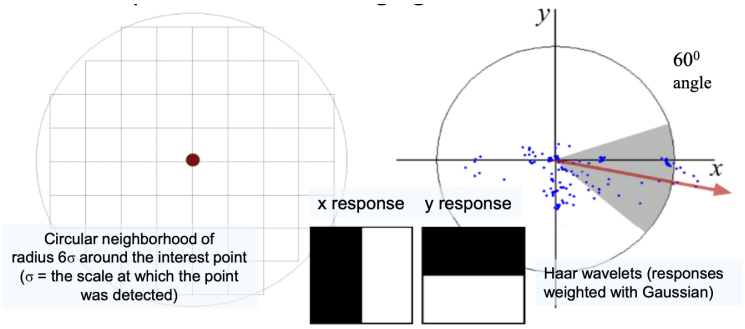


Figure 6.29: SURF orientation

We place these two filters in each point of the neighborhood and we take two values, obtained by the subtraction of white region values from black region values, which approximate the gradient.
We plot these values in a (x, y) plane and, each 60 degrees, we sum all values of x and y for each bin and, at the end, we take the bin (so the orientation) with higher value as shown in Figure 6.29.
At the end we need to talk about **descriptor**: the first step consists of constructing a square region centered around the keypoint and oriented along the orientation we already got above.
Then the region is split up regularly into smaller 4×4 (16) square sub-regions and for each sub-region we sum the response for d_x and d_y and sum the modules for d_x and d_y of each point of the sub-region, with values obtained before by Haar wavelets.
Hence, each sub-region has a four-dimensional descriptor vector v for its underlying intensity structure $V = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$.
This results in a descriptor vector for all 4×4 sub-regions of length 64 (16x4). In Figure 6.30 this idea
Conlcuding, SURF is faster than SIFT but it's less robust to illumination and viewpoint changes w.r.t SIFT.

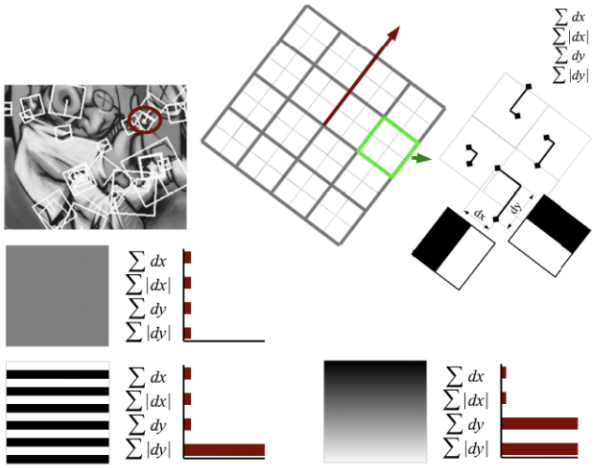


Figure 6.30: SURF descriptor

6.3.3 GLOH

GLOH is the acronym of **Gradient Location-Orientation Histogram**, obtained, as for PCA-SIFT, by modifying the 4th step of the SIFT algorithm, or rather the descriptor.
Once we have the image orientation of a neighborhood of our keypoint detected, we compute SIFT descriptor using a log-polar location gird, as shown in Figure 6.31 There are 3 bins in radial direction, at radii $6, 11, 15$. Then 8 bins in angular

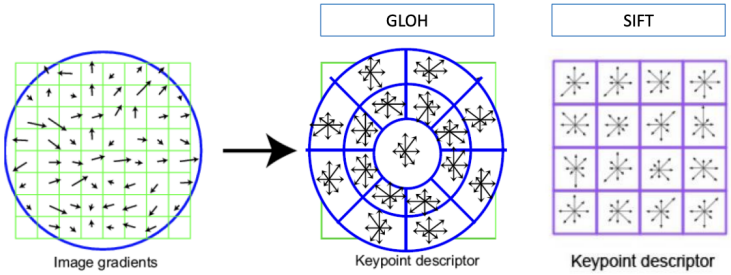


Figure 6.31: GLOH descriptor

direction with the central bin not subdivided. For each sub-sub-region there are 16 orientations.
Overall there are 272 orientation bins, with PCA used for reducing dimensionality to 128.

6.3.4 Other approaches

Shape Context descriptor: Firstly we start with a 3D edge point locations and orientations. Locations and orientations are quantized using log-polar coordinate system, with 5 bins for distance, 12 bins for orientations, as shown in Figure 6.32.

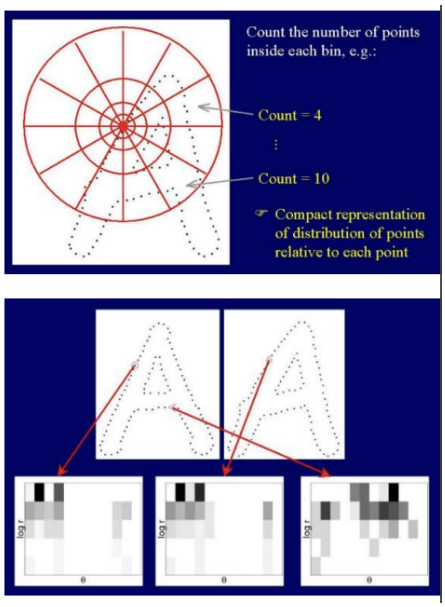


Figure 6.32: Shape Context descriptor

So we accumulate distribution of edge points in each bins.
Are achieved translation invariance and scale invariance and it is very robust to deformations, noise and outliers.
LBP descriptor: It's the first proposed descriptor for text recognition. It compares the central pixel with surrounding samples and it builds a binary sequence of signs of differences, as shown in Figure 6.33.

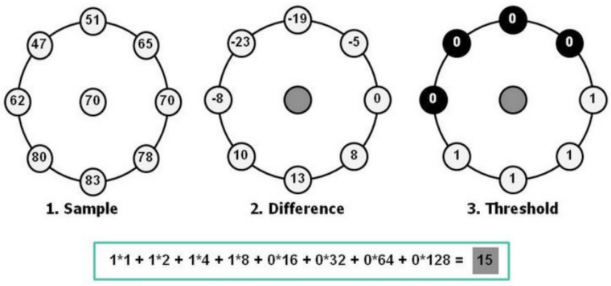


Figure 6.33: LBP idea

Basically, a pixel is selected and compared with its 8 neighbors: if the pixel is greater than the center, it assign a 1, instead a 0.
Then it's computed an histogram of such values and we can subdivide the image in subregions and analyzed (histograms are concatenated).
In Figure 6.35 an example of how can be used LBP.

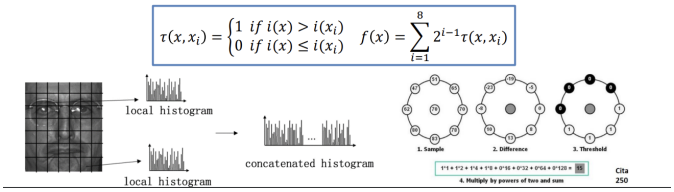


Figure 6.34: LBP approach

BRIEF descriptor: Is the acronym of binary Robust Independent Elementary Features. It's based on concepts illustrated in Figure 6.35

- Gaussian smoothing
- Pairs of pixels compared inside a window

$$\tau(\boldsymbol{p}; x, y) = \begin{cases} 1 & if\ p(x) < p(y) \\ 0 & otherwise \end{cases}$$

- Build a vector with comparison output

$$f(\boldsymbol{p}) = \sum_i 2^{i-1}\tau(\boldsymbol{p}; x, y)$$

Figure 6.35: BRIEF descriptor

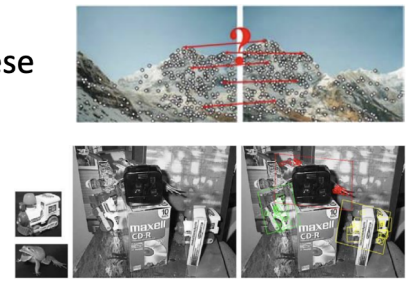
We concatenate a vector containing 1 and 0 and then vectors are compared using the Hamming distance.
How we sample the pixels couple? There are several ways, uniform (randomly selected) and the best solution is the isotropic gaussian distribution.
ORB: Oriented FAST and Rotated BRIEF. It uses the FAST corner detector and add rotation invariance to BRIEF. Orientation assignment based on the intensity centroid w.r.t the central pixel.

6.4 Feature Matching

Feature matching is another bullet point of our pipeline shown in Figure 6.4.
Feature matching means find similar descriptors in two images.
We need to identify the best **matching strategy**, so what features to compare,

the best and **match evaluation**, so assign a number to the match.
The matching strategy depends on the application at hand because for instance, in some cases similar features can be in similar positions and in other cases no and also, two features in the same positions are not forced to be related.

- Example
  - How many matches in these two cases?
    - Image stitching
    - Object detection in clutter



Several strategies are possible:

1. **Maximum distance**: match against features within a geometric distance but it's difficult to set the threshold
2. **Nearest Neighbor (NN)**: Consider only the nearest neighbor in **feature space** (means that I have matched given one feature, the closest, in the feature space). A threshold is also used (the nearest neighbor may be distant)
3. **Nearest Neighbor Distance Ratio (NNDR)**: it is the ratio between d_1 nearest distance and d_2 second nearest distance

After we choose some association method, we want to measure performance using the following indicators:

- **True Positive (TP)**: number of correct matches
- **True Negatives (TN)**: number of correct non-matches
- **False Positive (FP)**: number of non-matches that were wrongly macthed
- **False Negatives (FN)**: number of matches that were wrongly missed

Are also used **precision** and **recall** evaluation score:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

6.5 Face detection: the Viola-Jones approach

Face detection is a key task in computer vision: the algorithm has to be able to recognize if there are faces on an image and where are; we will see the Viola-Jones approach, that can be used also on other tasks.
The approach is the **sliding window**: the window move over the image and stop when find a match. So it's a classification task: for each sliding window, we want to know if there is a face or not.
It's a challenging task due to the huge number of pixels, to the multiple locations and scales, where all combinations need to be evaluated, to the fact that faces are an unlikely event.
Moreover, we would like key elements for an optimal face detector such that there is fast processing for non-face candidates and a very low false positive image.
Key elements of the Viola-Jones face detector are surely **Haar features**. Haar features looks like rectangular filters, composed by a black zone and a white zone. We sum up intensities for both zones and we subtract the value of the white area from the value of the black area:

$$f(x) = \sum_i p_{black}(i) - \sum_i p_{white}(i)$$

It's possible a fast feature evaluation based on the **integral image** using Haar features. The feature value will be around zero for "flat regions", i.e., where all the pixels have the same value.
A large feature value will be obtained in the regions where the pixels in the black and white rectangles are very different.
There are a lot of possible Haar features: they can be composed by 2,3 or 4 rectangle inside and positioned in a very different way.
We will see that the most used and characterizing Haar features are shown in Figure 6.36, and by "posteriori thinking" that's reasonable since these features remind to face pattern like nose, eyebrown, ecc.
Haar features are coarse features but very sensitive to edges, bars and other simple structures. Moreover are computationally efficient, since a large number of features can be computed and this compensates for the coarseness.
However, considering a sliding window of 24x24, and considering that there are a huge number of possible Haar features, near 160k.

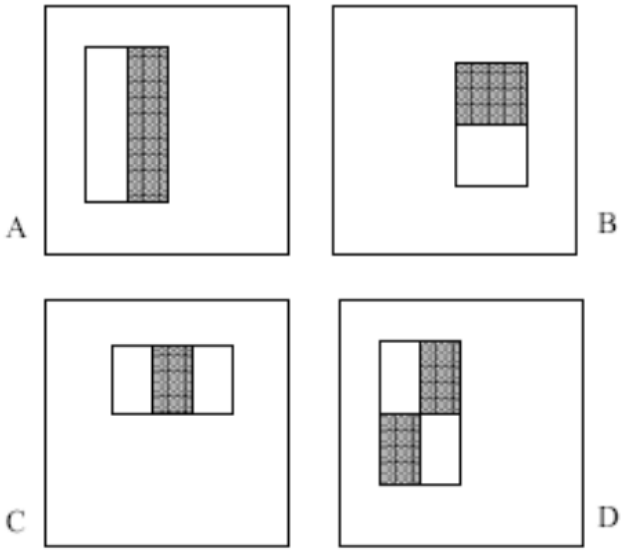


Figure 6.36: Most used features

So, if we want to implement a classifier that given a sliding window, provide as output if it is a face or not, testing for the response 160k different features it's computationally intractable.
The idea that we follow is to train a model that learns which are the best features, with an algorithm called **AdaBoost**, based on the ML technique of **boosting**.
For this purpose, we could use **weak learners for Haar features**: after have computed the value f_i for a single Haar feature, we use these which work on the value f_i.
The behaviour of a weak learner is like a classifier: it works on the number evaluated by the Haar feature f_i; it's set a threshold (that has to be found) and if the value of the Haar feature is greater than the threshold, the image could contain a face, as shown in Figure 6.29:

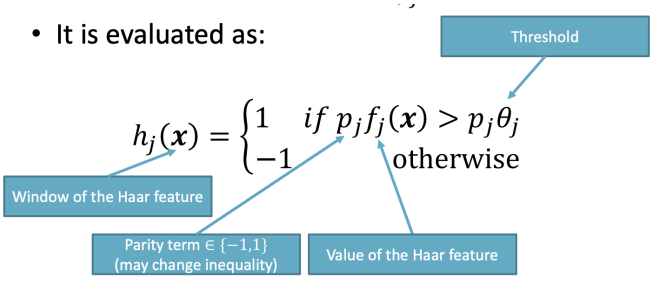


Figure 6.37: Weak classifier

With AdaBoost we build a strong classifier by combining several weak classifiers,

that, in math terms, can be expressed as a **weighted sum of simple weak learner**:

$$h(x) = sign[\sum_{i=1}^{m-1} a_j h_j(x)]$$

To each example in the training set is initially assigned a weight that determines its importance in the training of the first weak classifier. Typically, all examples start with equal weights.
At each iteration, we test each possible Haar feature, in each position, on each example (so each weak leranr). We select the best threshold for each feature and we select the best combo Haar feature + threshold (the one that has lowest error).
Then we reweight examples: examples that were misclassified by the chosen weak classifier are given more weight, whereas correctly classified examples have their weights reduced.
This reweighting emphasizes the harder-to-classify examples in the next round of training, pushing the algorithm to focus on correcting its previous mistakes, as shown in Figure 6.38

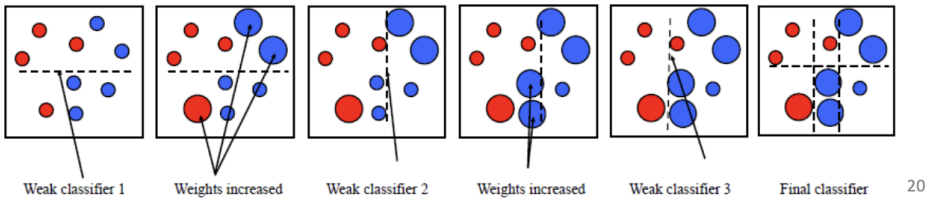


Figure 6.38: Re-weighting

The computational complexity is $O(MNK)$. So, supposing that M, the number of rounds, is 200, the algorithm finds the best 200 weak classifiers (Haar feature plus its threshold).
The last step of Viola Jones algorithm is, once weak classifiers are found, to subdivide weak classifiers into **stages that are applied in cascade**. Each stage makes use of several weak learners, in increasing order, reaching at the end, in the last stage, by summing weak learners of other stages, the number M.
Each stage works as a classifier and if it is recognized a face, the image is moved in the next stage.
We have to notice that a false negative cause a failure while a false positive is acceptable at a high rate.

So the idea is to train several classifiers with an increasing number of features until the target decision rate is reached.
It's important to re-weight training examples after each stage giving an higher weight to samples wrongly classified in previous stages: classifiers are progressively more complex and have lower false positive rates.
In Figure 6.39 the structure used by Viola and Jones, composed by 38 stages which uses, totally. 6061 features.

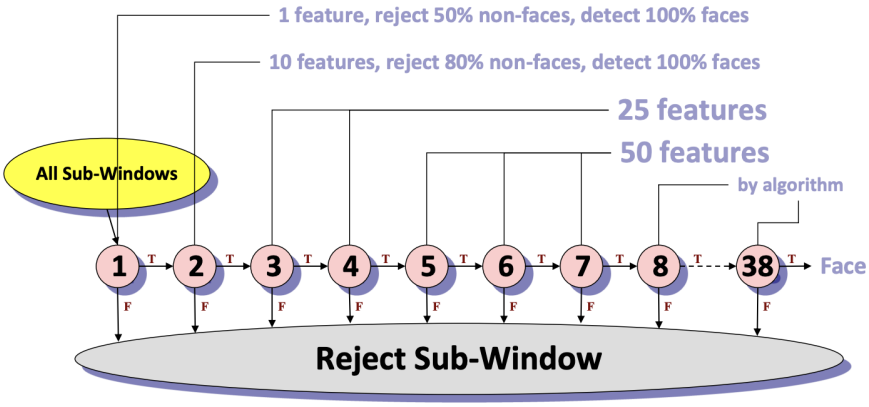


Figure 6.39: Cascade for Viola and Jones

This method has the same accuracy of using all identified features for each image but has a 10x difference in processing time. In Figure 6.40 an idea of this method and the first two features identified by AdaBoost

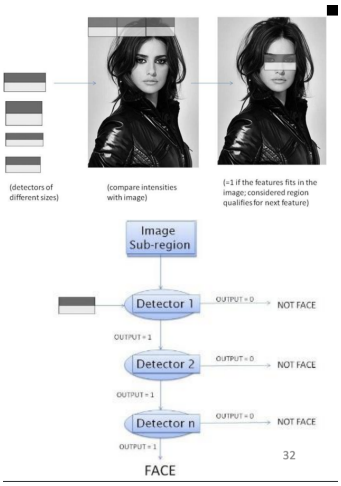


Figure 6.40: Features found by Viola and Jones

7

Camera model

How are images created?
In order to have an image, we need an **object**, a **light source** and a **sensor**: as shown in Figure 7.1, solar rays are reflected on the object and then are captured by the sensor of the camera.

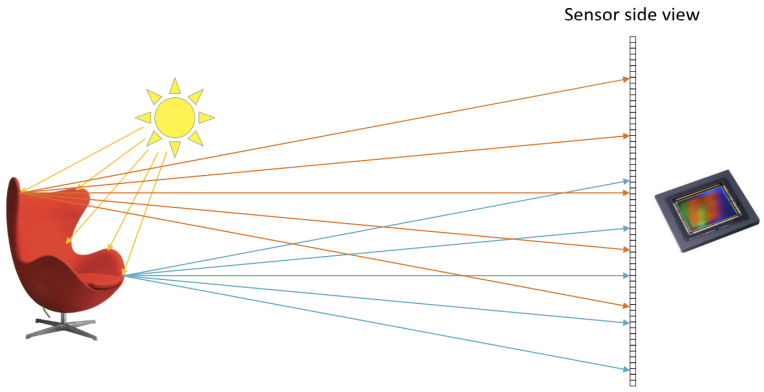


Figure 7.1: Image acquisition

However, in the Figure above we can see that rays for a single point are many and this could cause a bad image, with distortion and rotation.
So, for this reason, each camera has a **hole** which captures only few rays, as shown by Figure 7.2.
The selection of rays depends on the size of the hole: we get a perfect definition of the image if only one ray per point reaches the sensor. This happens if the hole is just a point, called **pinhole camera model**.

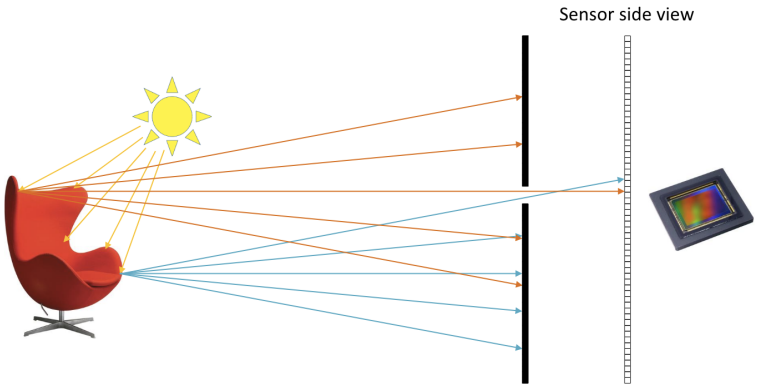


Figure 7.2: Insertion of a hole for better result

Let's try to model a real camera using a pinhole model (assuming that the hole is puntiform).
Observations: Some effects, as diffraction, are neglected, since it's a simplified model to start studying the geometric projection.
In Figure 7.3 the camera that we are studying and from we want to know how images are captured, represented with its axes.

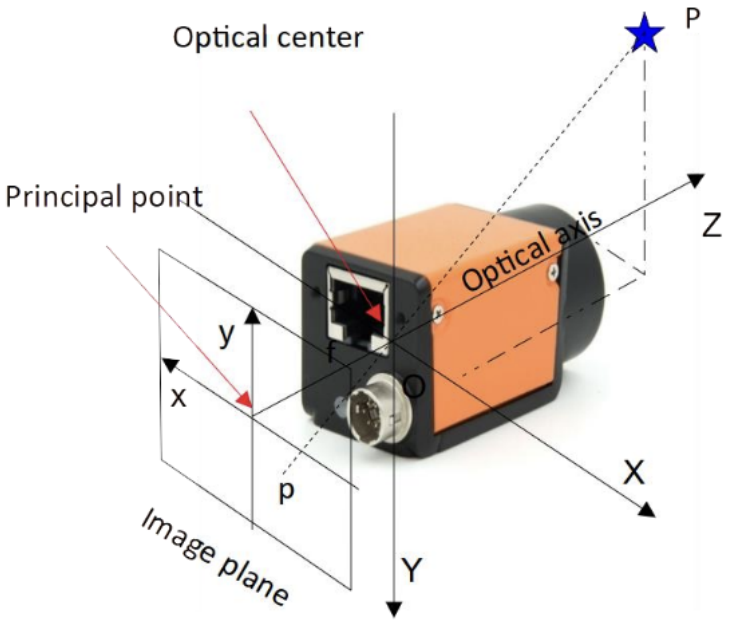


Figure 7.3: Camera with its model

In Figure 7.4 the model of this camera:

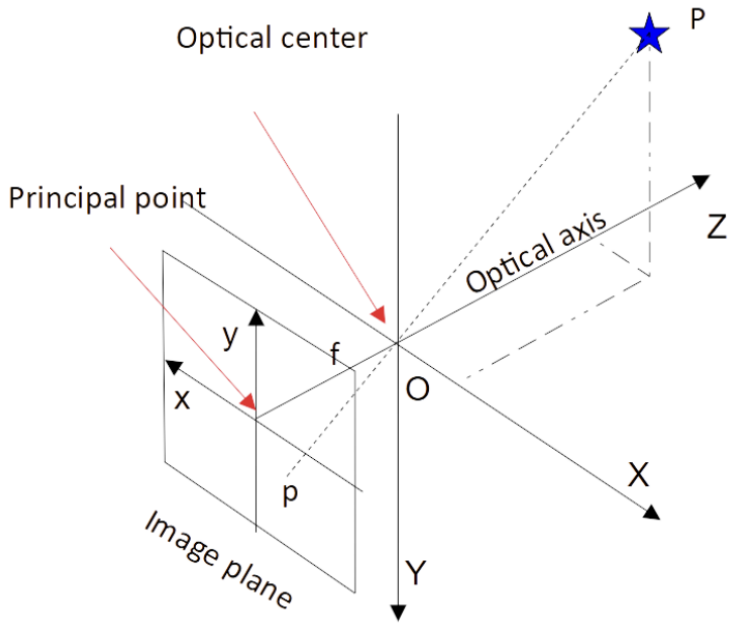


Figure 7.4: Model of a camera

The main elements are:

- **Optical center** O, where is the location of the pinhole
- **Image plane** π. We can see that in the image plane there is the projection of the point P that camera takes
- **Optical axis** Z, perpendicular to image plane and passing through the optical center
- **Principal point** that is the intersection between image plane and optical axis
- **Focal length** f, that is distance between optical center and image plane
- **Reference systems**, more than one. One describes the camera and one describes the image plane. The one that describes the camera is "fixed": the optical axis is z, and x is on the right. Using right-hand rule, y axes points down.

7.1 Projective geometry

We need to describe the geometry of projection quantitatively, so how objects are projected in the image.

The first element to take into account is the **relation between the two reference systems** or rather 3D point in the world, **seen from the camera**, and the 2D point **visualized on the image plane**.
Consider a point P and its projection p, as shown in Figure 7.5, and we want to find the relation between this two.

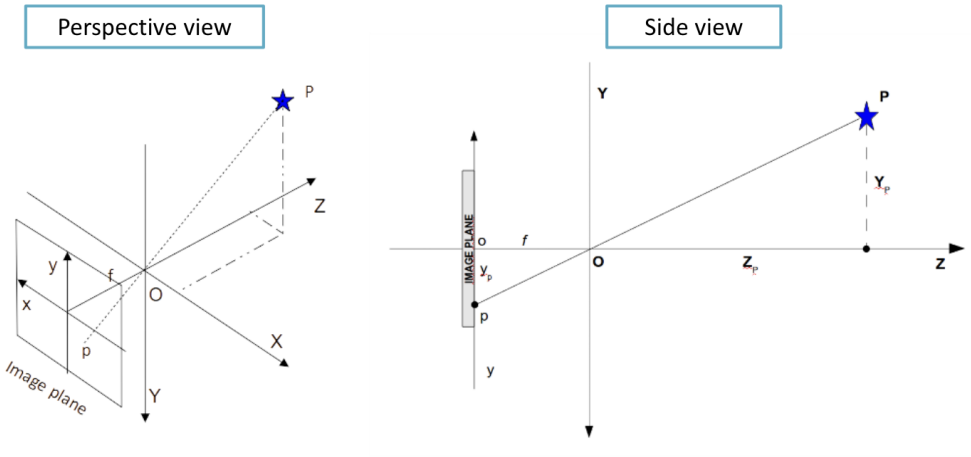


Figure 7.5: Point projection

Consider now a plane that is parallel to the image plane, in front of the optical center (so in the 3D space, in opposite side of the image plane) and at the same distance f from the optical center, as shown in Figure 7.6

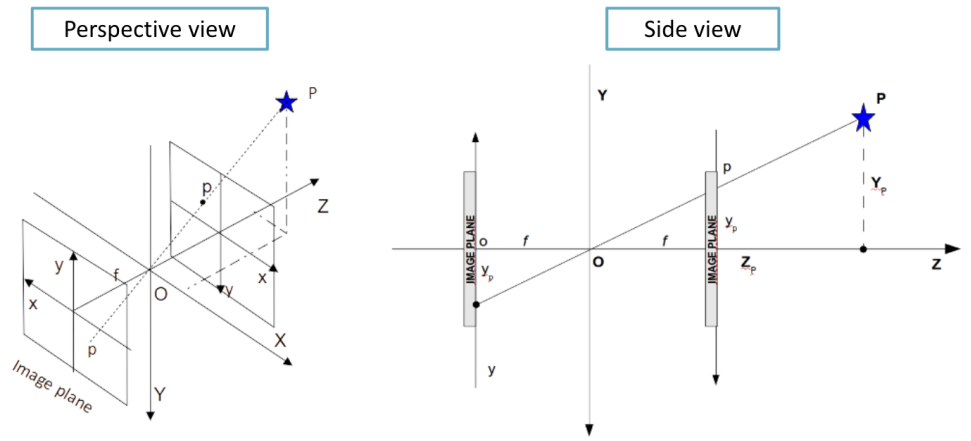


Figure 7.6: Point projection

It's easier to work on this plane since **it has the same geometrical relation and avoid the upside-down effect**, so we move to this plane for deriving the geometrical description, as shown in Figure 7.7.

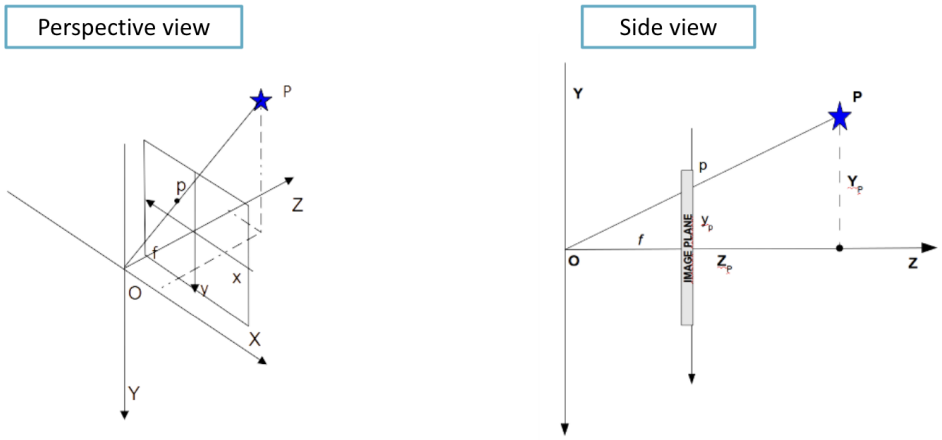


Figure 7.7: Point projection

The **Field of View** (FoV) of a camera is the angle of view perceived by the camera. We define α that is the angle created under the projection of P on O. By moving P, α changes but it has a maximum value that can have that is $\frac{1}{2}$ of the FoV, represented by the red line in the Figure 7.8.

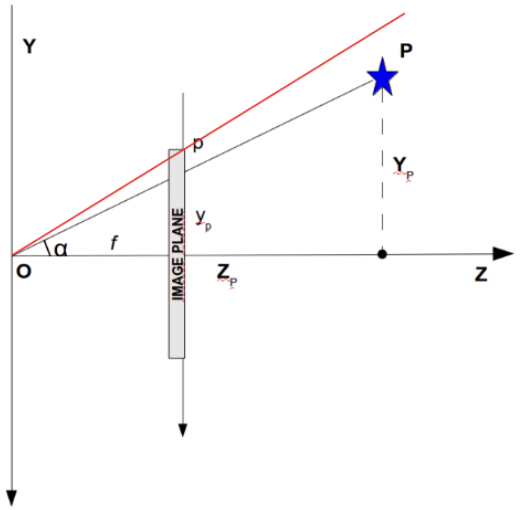


Figure 7.8: Maximum angle

The FoV depends on the sensor size d, on the focal lentgh f and so, the maximum angle α is equal to φ:

$$\varphi = \arctan(\frac{d}{2f})$$

As we can see from Figure 7.9, using trigonometry, we can obtain φ. It's important to notice that FoV is equal to double of φ.

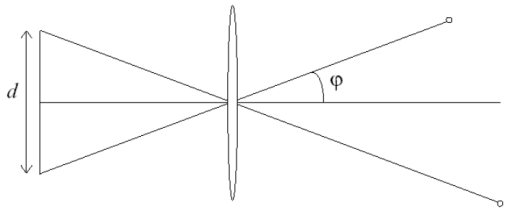


Figure 7.9: Trigonometry

So now we're focusing on the yellow box of Figure 7.10

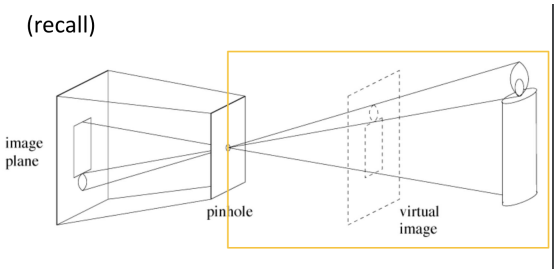


Figure 7.10: Region under consideration

Our task is to relate P and p: we can show that there is a relation between these two.
We know that a point in the 3D space is defined as (X_P, Y_P, Z_P) and we want to extract (x_p, y_p) for the 2D plane. From the similar triangle rule we obtain:

$$\frac{Y_P}{y_p} = \frac{Z_P}{f}$$

and analogous for x, obtaining:

$$x_p = f \cdot \frac{X_P}{Z_P} \ , \ y_p = f \cdot \frac{Y_P}{Z_P}$$

where $\frac{Y_P}{Z_P} = \tan(\alpha)$
However, projecting points on a 2D surface causes the loss of the distance information, the depth. That's because there are different values of Y_p, Z_p which brings to the same y_p, and are for those P that are in the same projection line connecting O and the point.
So, now that we've seen that starting from a point in the 3D plane we can obtain

the coordinate in the image plane, it's useful to rearrange equations used for conversion in matrix form.
This is done by exploiting **homogeneous coordinates**. Points in 2D can be expressed in homogeneous coordinates:
To homogeneous coordinates:

$$\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{w} \end{bmatrix}$$

From homogeneous coordinates:

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{w} \end{bmatrix} \rightarrow \begin{bmatrix} \frac{\tilde{x}}{\tilde{w}} \\ \frac{\tilde{y}}{\tilde{w}} \end{bmatrix}$$

This concept can be extended to N dimensions: N-dimensional point transformed into $(N+1)$ homogeneous coordinates.
So, using homogeneous coordinates both for 2D and 3D coordinates, we obtain this rearrangement:

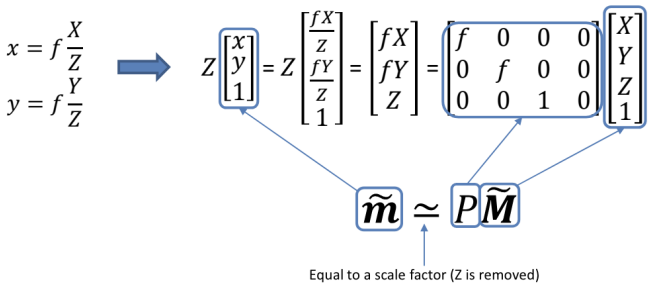


Figure 7.11: Rearrengement of equation

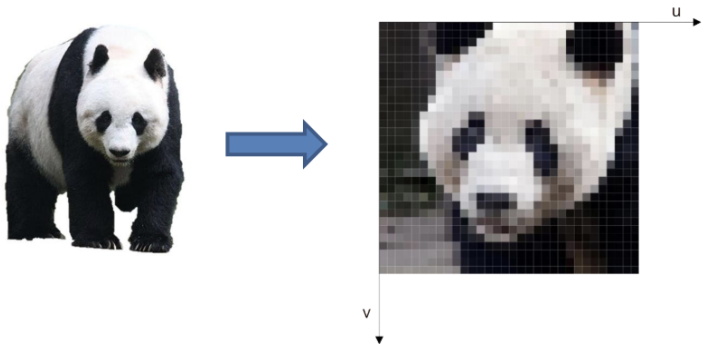
Where P is the **projection matrix**.
It's important to do a couple of observations:

- It's used as value of $\tilde{w} = 1$ in order to have $x = \tilde{x}$
- We have separated all parameters of the camera from the parameters of the point in two different structures
- When $f = 1$ we obtain the **essential perspective projection** as shown in Figure 7.11, which represents the core of the process

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = [I|\mathbf{0}]$$

Figure 7.12: Essential perspective projection

Without taking into account non-ideality, this matrix could seem redundant. However, we will see that this matrix embedds all information about camera.
What we have seen so far is that the projection matrix describes how the 3D world is mapped onto the image plane.
However, a point in the image plane has not the same measure unit of a point of the image, due to the fact that the digital image uses pixel.
We need to map points projected onto the image plane in the coordinates used for pixels, from (x, y) to (u, v).



The transformation can be defined considering the coordinates of the principal point to be (u_0, v_0), remembering that image coordinates have the origin in top-left corner.
Metric distances are converted to pixels using the pixel width w and h height where the conversion factor are defined as:

$$k_u = \frac{1}{w},\ k_v = \frac{1}{h}$$

Mapping a point (x_p, y_p) to (u, v) is obtained by translation an scaling

$$u = u_0 + \frac{x_p}{w} = u_0 + k_u x_p$$

$$v = v_0 + \frac{y_p}{h} = v_0 + k_v y_p$$

where u_0, v_0 are used for translating the point in order to be in the same position for the new coordinate system, as shown in Figure 7.13.

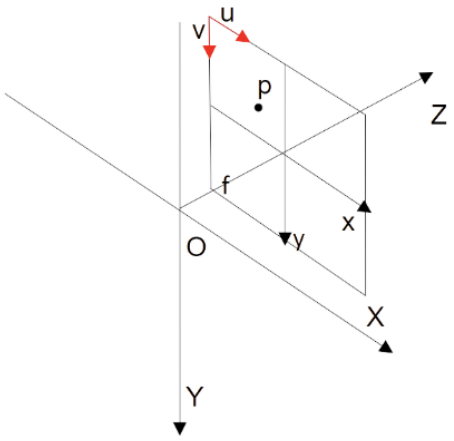


Figure 7.13: Reference system

We can combine the mappings from 3D to 2D image plane and from image plane to pixels by substituting the last equations inside the projection equation:

$$u = u_0 + k_u x_p = u_0 + k_u f \frac{x_p}{z_p} = u_0 + f_u \frac{x_p}{z_p}$$

where $k_u f = f_u$ is the **focal length in pixels**.
Summarizing all, doing the same for v, we get:

$$u = u_0 + f_u \frac{x_p}{z_p}$$

$$v = v_0 + f_v \frac{y_p}{z_p}$$

So we can now express the projection matrix as:

$$P = \begin{bmatrix} f_u & 0 & u_0 & 0 \\ 0 & f_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \triangleq K[I|\mathbf{0}]$$

where K is the **camera matrix** (the previous equation $\tilde{m} \approx P\tilde{M}$ holds, there is only a different formulation for P; $\tilde{m}$ has coordinates of the pixel information).

Considering the camera matrix, it depends on k_u, k_v, u_0, v_0, f that are called **intrinsic parameters** which define the projection characteristics of the camera, embedded in the matrix.
It's important to highlight that f_u, f_v embed three parameters.
If we move the camera, intrinsic parameters doesn't change since are characteristic of sensors etc.
Until now we have mapped world to image plane and image plane to pixels. However, a different reference frame can be defined on the world, for example by moving the camera. It's always defined as a **rototranslation**.
We need to define a rototraslation matrix in 3D in homogeneous coordinates, expressed as:

$$T = \begin{bmatrix} R & \boldsymbol{t} \\ \mathbf{0} & 1 \end{bmatrix}$$

So, now the equation for computing the pixel in the image: $\tilde{m} \approx PT\tilde{M}$
It's in this position due to the fact that we first apply a rototranslation in order to align with the camera reference system. Then with the projection matrix P we convert it in pixel coordinates.
The rototranslation matrix T has 3 parameters for translations and 3 for rotations, called **extrinsic parameters**, which define the relation between camera and world.
So, resuming, the whole process involves four reference systems and three transformation, as shown in Figure 7.14

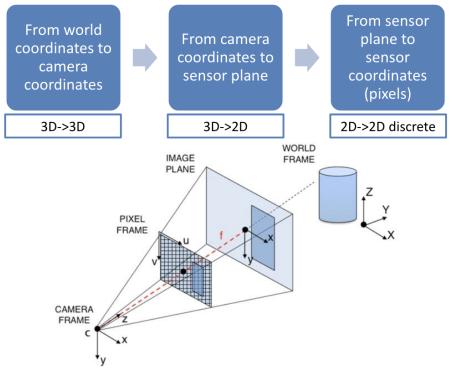


Figure 7.14: Pipeline

If we want to invert the process, so passing from pixel to 3D coordinates, we need to invert some elements; however, projection matrix P and pixel quantiza-

tion are not invertible. (Remember that when we pass from 3D to 2D we loose a dimension so it's difficult to think to reconstruct it.)
We can invert the projection if:

- We accept as a result the direction of the object, not the 3D position or
- we have additional constraints providing the location on the line

We neglect the quantization effect: the pixel location and the projected point are considered the same: it's acceptable for high-resolution sensors.

7.2 Geometric Transformations

Geometric Transformations go back to the idea of moving pixels. We have seen that through image processing we transform pixels; we have analyzed several methods for modifying the pixels of an image.
We add another type of transformation, the **geometric transform**: a geometric transform is a modification of the spatial relationship among pixels. It involves two steps:

- **Coordinate transform** $(x', y') = T\{(x, y)\}$. This step involves mapping the coordinates of the original image to new coordinates based on the transformation that is being applied. For example, if an image is being rotated, each pixel's new position needs to be calculated relative to the rotation center and angle.
- **Image resampling**: After calculating where each new pixel should be located (which might not align with the original grid of pixels), the next step is to determine the pixel values at these new locations. This step is crucial because pixel coordinates in a digital image are discrete, and the transformation typically results in non-integer coordinates.

Coordinate transformation works on **geometrical points** and not single pixel.
In Figure 7.15 we can see an overview of basic planar transformations, so 2D transformation.
Every transformation in the table preserves the properties listed at rows below of its row.
As we already said, coordinates in 2D can be expressed as homogeneous coordinates and, using as example a translation, the translation in homogeneous coordinates become as shown in Figure 7.16
The **affine transform** is a more generic transformation(rotation, scaling, etc.), since it's a linear transform (by a multiplication of a matrix A) followed by a

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \boldsymbol{I} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} \boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	3	lengths	
similarity	$\begin{bmatrix} s\boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	4	angles	
affine	$\begin{bmatrix} \boldsymbol{A} \end{bmatrix}_{2\times3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{\boldsymbol{H}} \end{bmatrix}_{3\times3}$	8	straight lines	

Figure 7.15: Planar transformations hierarchy

- Translation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

- Translation in hom coords

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & b_1 \\ 0 & 1 & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Yielding

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & b_1 \\ 0 & 1 & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + b_1 \\ y + b_2 \\ 1 \end{bmatrix}$$

Figure 7.16: Translation

translation.
It's very useful since it preserves point collinearity and distance ratios along a line.
In homogeneous coordinates it becomes:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = A \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \qquad \begin{bmatrix} \tilde{x} \\ y \\ 1 \end{bmatrix} = T \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

which represents multiple operations combined into a single matrix multiplication. As we can see from Figure 7.17, the affine transform can be used in different ways in order to perform different operations.
Then we need to going back to pixels, using **resampling**.

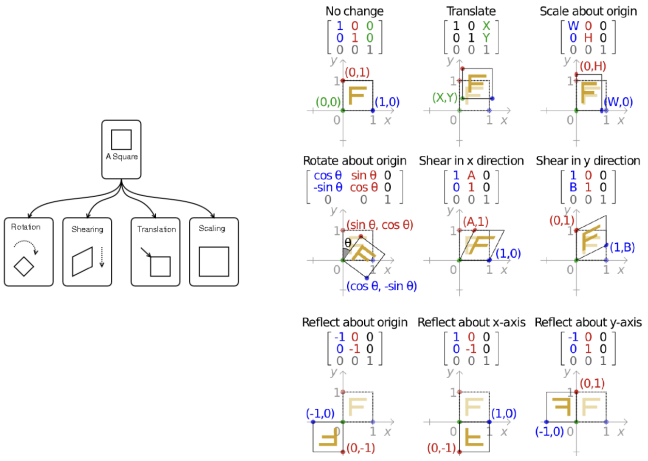


Figure 7.17: Affine transform operations

7.3 Camera and Lenses

Now we know everything about projection of a pinhole camera model. We want to get further in the description of the camera and we will study the effect of an ideal lens.

Recalling the pinhole, it has a major limit that is the few light entering: since only one ray enters, we lose a lot of light intensity. However, as we already said, the fact that there is only one ray, creates a sharper image. If we add a lens we could handle this problem, obtaining a sharper image without needing a pinhole.

Introducing a lens centered on the pinhole allows us to leverage on **thin lens model** (thin because we can neglect the lens width): the lens has the main axis lying on the optical axis and the center lying in place of the pinhole.

The thin lens is modeled as a 2D plane where deviation occurs: lens deviates light and focuses rays as shown in Figure 7.18

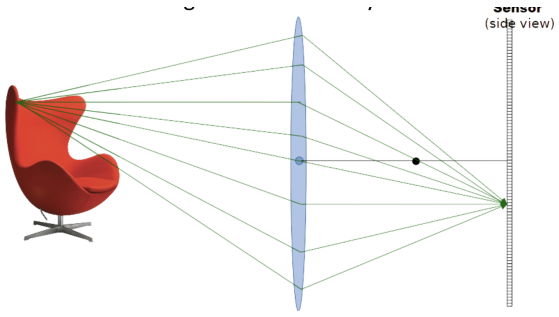
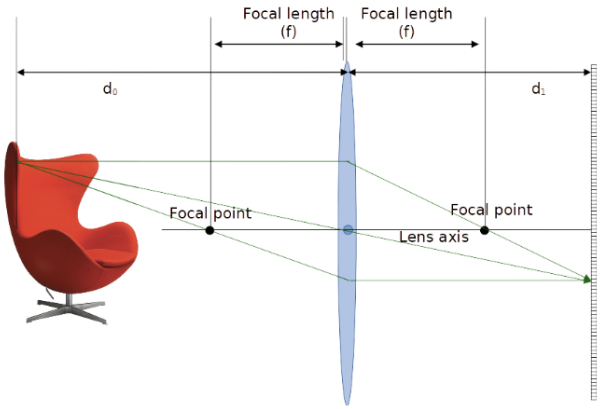


Figure 7.18: Lens rays deviation

Rays are deviated following simple rules:

- Rays passing by the center are not deviated
- Rays parallel to the optical axis to the lens axis are deviated through the focal point
- Rays passing through the focal point in the object plane exits the lens parallel to the optical axis



The **thin lens equation** relates distances between position of the object and position of the image: assuming d_0 the distance between object and lens and d_1 the distanc between lens and image, the relation is given by the focal length f:

$$\frac{1}{d_0} + \frac{1}{d_1} = \frac{1}{f}$$

However, the distance of the object from the lens can change the localization of the point in the image: points at a given distance are in **focus**; all other distances are not in focus and generate a **circle of confusion** and rays don't convert in a single point of the image, reducing quality, as we can see from Figure 7.19

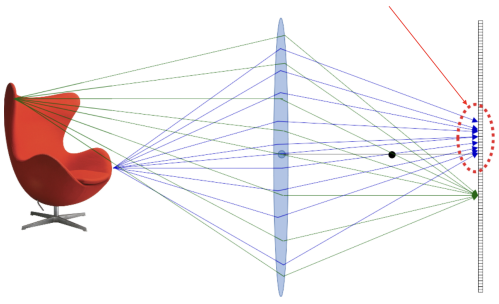


Figure 7.19: Circle of confusion

Changing the distance between lens and sensor (so changing the focal length) changes the characteristics of the focus. If the image sensor would be translated further (so we increase focal length), the circle of confusion would be for green rays and not for blue rays.
In order to reduce the circle of confusion, we can add an **aperture** as we can see from Figure 7.20

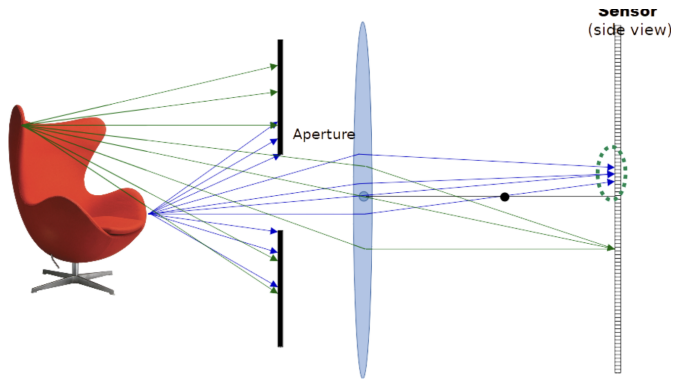


Figure 7.20: Affine transform operations

We have seen different expression of **focal length**:

- Thin lens: distance at which parallel rays intersect
- Pinhole camera model: distance between pinhole and sensor

However, lens used so far is thin and ideal: real lenses are affected by additional effects like distortion, chromatic aberrations and other minor effects.

- **Distortion**: is a deviation from the ideal behavior described so fare and a deviation could be **radial** or **tangential**:

    1. Radial distortion: this type of distortion occurs because the lens's magnification varies with the distance from the optical axis (the center of the image).
    he amount of distortion generally increases the further a point is from the center of the image. This means that objects located at the edges of an image will appear more distorted than objects that are closer to the center.
    As we can see from Figure 7.21, there are two types of distortion, **pincushion and barrel**.
    The radial distortion can be analyzed by means of distortion patterns, so it can be experimentally measured and analytically described if the lens structure is known in detail.
    A **distortion chart** can be used.
    Camera models commonly handle this problem by considering a polynomial approximation for the radial distortion, that is simply a mathematical model and not a phisycal one

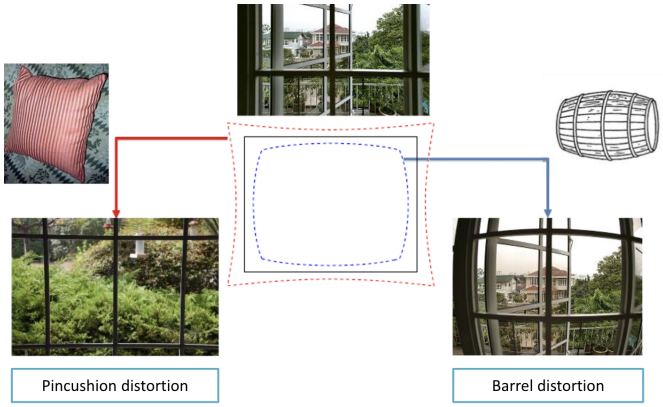


Figure 7.21: Radial distortion

2. **Tangential distortion**:is caused by the non-ideal alignment between lens and sensor. Tangential distortion causes the image to tilt or shift in a way that is not symmetric around the image center. This typically occurs when the lens and the image plane are not perfectly parallel. It's usually negligible and can be corrected using some models. Also tangential distortion is modeled in terms of a mathematical approximation

- **Chromatic aberration**: near the image edges there are color fringes due to the fact that there are different waves with different wavelengths and, since the refractive index depends on λ, it causes this phenomenon

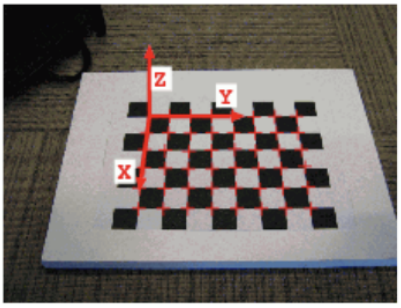
7.4 Calibration of a camera

Camera calibration is the process of estimating the camera parameters. Very often are used intrinsic calibration only, including distortion parameters.
Calibration is needed if we want to measure the projection characteristics of a camera.
The calibration procedure usually is made in a such way that the real 3D position should not be defined but anyway links 3D objects and projection. A dedicated process can be used to evaluate the camera position and orientation in world coordinates (camera orientation is expressed by means of three angles: yaw, pitch and roll).
There are several methods for calibrating a camera and the general process is:

1. Take an object of known shape and appearance

2. Take pictures of the object
3. Analyze the projection process

In principle, the calibration pattern can be any object of **known shape and dimensions** in the image. An example could be a checkerboard where the points used for calibration are the square corners.



So we have to collect N images of the pattern and for each image list the M 3D corner positions in the pattern reference system and find the corner positions in the image reference system.
Then we need to initialize the intrinsic calibration parameters to **default values**:

- For K matrix: $f_u, f_v, u_0, v_0 \hookrightarrow \vec{\theta}_k$
- For distortion: $k_1, k_2, k_3, p_1, p_2 \hookrightarrow \vec{\theta}_d$

Then we need to initialize also the extrinsic parameters to **default values** and then we need to solve the non-linear least squares problem:

$$\min_{\boldsymbol{\theta}_K, \boldsymbol{\theta}_d} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \left\| K[I|\mathbf{0}] T_i \widetilde{\boldsymbol{P}}_{i,j} - \widetilde{\boldsymbol{p}}_{i,j} \right\|^2$$

Projected onto the image plane

Point on the image

3D point

However, the minimization process is done over a large set of parameters: it can hardly provide a good result.
Good initial guesses would be very desirable and, to do that, we can exploit **homography**. Homography refers to a transformation that maps the points from

object plane to the points in image plane. It's possible only if the object is **planar** and if we neglect distortion.
Using a homography simplifies the mathematical description since is found a good initial guess.

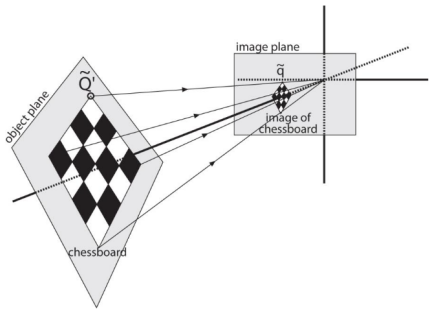


Figure 7.22: Homography representation

How many views of our object are needed to calibrate a camera? Each view of a planar object can be represented by a homography: each image of the planar object can be described using a homography. This means we can use the homography to relate the object points (in 3D) to the image points (in 2D), as we can see in Figure 7.22.
A homography has 8 degrees of freedom. This means that to fully determine a homography, you need 8 independent pieces of information (or constraints).
Each corner point of the checkerboard provides two constraints because each point has an x-coordinate and a y-coordinate. For example, if you know the position of a point on the checkerboard in the 3D space and its corresponding position in the 2D image, you have two constraints (one for x and one for y).
To determine the homography, you need to solve for 8 variables (due to the 8 degrees of freedom).
While a checkerboard provides many corner points, only four points are necessary to determine the homography because these four points provide the 8 necessary constraints. The rest of the points are not "free" in the sense that they do not provide independent new information once the homography is determined by the initial four points.
Imagine you have a checkerboard with 25 corner points. When you use these points to compute the homography, you essentially use 4 points to solve for the 8 variables in the homography. The additional points can be used to improve accuracy and verify the consistency of the homography, but they do not add

new degrees of freedom. They help in refining the model, especially to handle noise and improve robustness.
Neglecting distortion, we have 4 or 5 intrinsic parameters and 6 extrinsic parameters, so a minimum of 2 views are needed. With a minimum of 2 views, we get enough constraints to solve for the intrinsic and extrinsic parameters. Each view provides multiple constraints from the observed points in the image.
The calibration process is handled by a minimization and it's unstable, so a larger number of views is needed in practice.
In Figure 7.23 a typical calibration image set

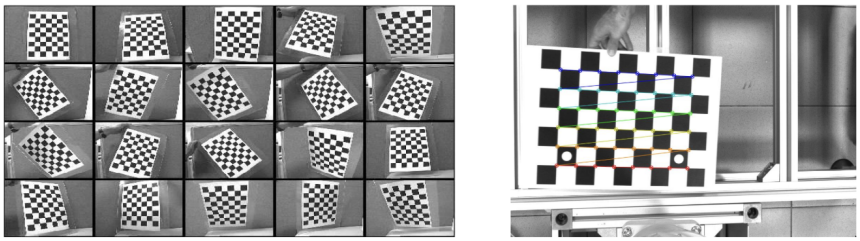


Figure 7.23: Affine transform operations

7.5 Real Cameras

We have analyzed all details in transformation of a 3D point of the real world to 2D point of the image.
The transformation seems like Figure 7.24

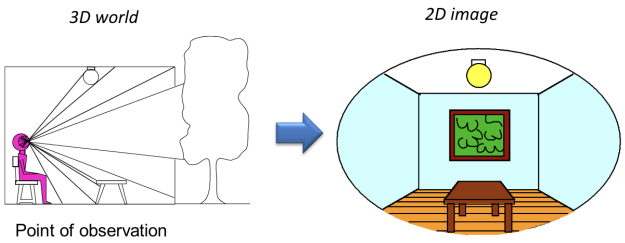


Figure 7.24: Transformation

As we can remember, we have lost a dimension and the effect is the loss of angles, distances, parallel lines (instead straight lines remain straight).
In Figure 7.25 we can see the effect of parallel lines that are not parallel in the image and that lengths in an image can't be trusted.

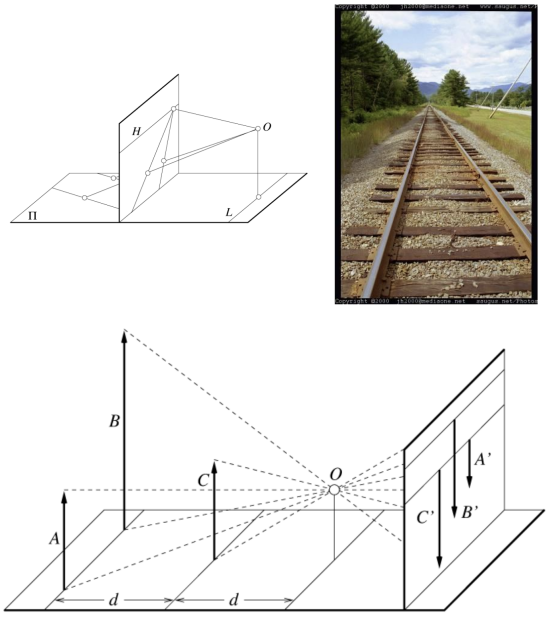


Figure 7.25: Problems of transformation

Our brain interprets length in images with some illusion, as the **Muller-Lyer illusion**.
We have seen that, thanks to the focal length of a lens, we can change the subject

size in an image.
Focal length is not the only element to change subject size: we can achieve the same effect in images taken with two different lenses by adjusting the viewpoint. Specifically, if you use a camera with a short focal length lens and move it closer to the subject, you can capture an image similar to one taken with a longer focal length lens from a further distance. This way, despite the different focal lengths, the subject's appearance remains consistent, while the perspective and background may vary.

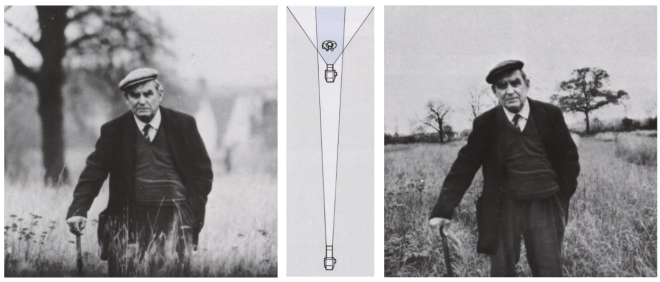


Figure 7.26: Equal images taken by different cameras

As we can see in Figure 7.26, the left image is taken using a camera with a short focal length, so it's taken close to the subject. In contrast, the right image is taken using a camera with a longer focal length, so it's taken from a greater distance from the subject. We can observe that the object remains the same, but the background changes.
As shown in Figure 7.27, a short focal length results in a large field of view (FOV), capturing more of the surroundings and distorting the object. In contrast, a longer focal length results in a smaller FOV, capturing less of the surroundings and focusing more on the object.

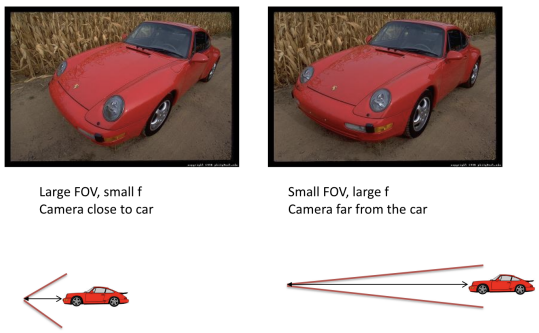


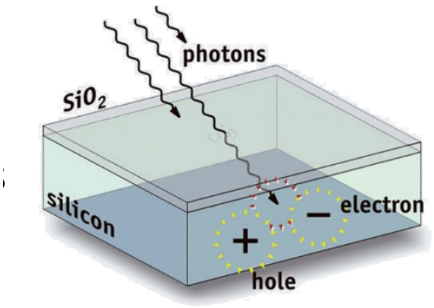
Figure 7.27: Two images with different FOV

Telephoto makes it easier to select the background (a small change in viewpoint

is a big change in the background).

7.6 Sensor

In digital camera, photons (light particles) hit the sensor and are converted into electrons (electrical signals). This conversion process is the basis of digital imaging.



There are two main types of image sensors used in digital cameras: **CCD and CMOS**.
These sensors measure the total energy of incoming photons but do not inherently differentiate between different wavelengths or colors of light. They are sensitive to the intensity of light but not its color. Both CCD and CMOS sensors are fundamentally greyscale. They detect light intensity (brightness) rather than color. Each pixel on the sensor measures the amount of light hitting it, but without further processing, it cannot distinguish between colors.
To capture color, additional methods are used:

- **3-chip color**: the incoming light is split into three separate beams using a prism or a similar device. Each beam passes through a different color filter (red, green, and blue) and hits a dedicated sensor for each color.
  In Figure 7.28 the prism used for subdivide light rays.

- **Single-chip color**: single sensor with a **color filter array** (CFA) placed on top.
  The most common CFA is the Bayer filter, which has a repeating pattern of red, green, and blue filters. The standard pattern is composed of 50%green, 25% red, and 25% blue filters. This distribution reflects the human eye's higher sensitivity to green light.
  When light hits the sensor, each pixel is filtered to record only one color component (red, green, or blue). For example, a pixel under a red filter will only capture the intensity of red light, ignoring green and blue light.

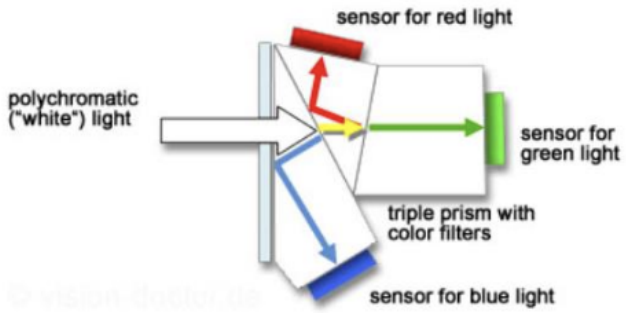


Figure 7.28: Prism for capturing colors

This results in a sensor array where each pixel only has partial color information (either red, green, or blue).
To reconstruct a full-color image, the camera uses an algorithm called demosaicing. This algorithm interpolates the color information from neighboring pixels to estimate the missing color values for each pixel.
For example, a pixel with a red filter will use the values from adjacent green and blue pixels to estimate the green and blue components of that pixel. In Figure 7.29 how it works and other filter used.

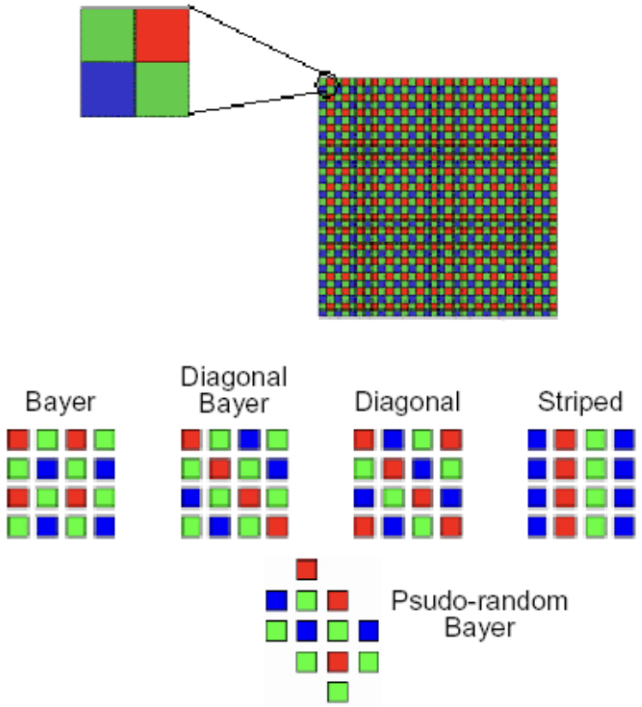


Figure 7.29: Color Filter Array

- **Chip Penetration Depending on Wavelength**

Someone thinks that the Bayer pattern is obsolete and there are some efficient sensors that can sense in a different way colors, without needing to interpolate. These are **direct image sensors** like Foveon: it consists of three layers of photodiodes stacked vertically. Each layer is sensitive to a different wavelength of light: blue, green, and red

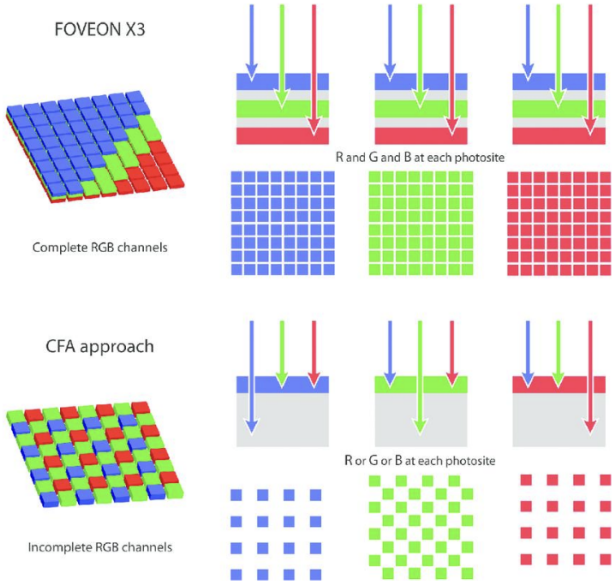


Figure 7.30: Problems of transformation

7.7 Incoming light

When you deal with professional cameras there are other two parameters to take into account:

1. **Aperture**: Aperture refers to the opening in a camera lens through which light passes to enter the camera. It's the nephew of the pinhole! We can choose how large the hole is. The size of the aperture (diameter of lens opening) is measured as a fraction of focal length with f-stops (f-number). The f-stop is the ratio of the focal length of the lens to the diameter of the aperture: for example $f/2$ on a $50mm$ means an aperture of $25mm$.
   The size of the aperture affects the exposure of the image. A larger aperture (low f-stop number) allows more light to reach the sensor, making the image brighter. Conversely, a smaller aperture (high f-stop number) allows less light to reach the sensor, making the image darker.
   A metric aperture would refer to the actual physical size of the aperture in millimeters. This approach does not account for the focal length and thus does not provide a consistent measure of the light-gathering ability (when

the focal length doubled, the amount of light gathered is subdivided by 4) and this is why $f - number$ is preferred.
Small $f - number$ (that is the denominator number) means a large aperture

2. **Shutter speed**: The shutter is a sort of shield of the sensor from the light. The picture is acquired when the shutter opens and closes, exposing the sensor.
The **exposure time** is the time the sensor is exposed to light.
Video cameras use electronic shutters where the effect of shielding and unshielding is obtained by electronic controls. There are two key parameters that are **exposure time** and **framerate**: the second is the time distance between two consecutive acquisition.
There are two technologies: **global shutter** where all the pixels are acquired at the same time and **rolling shutter** where the image is acquired row by row.

Given some amount of light, the same exposure to the light (light captured) is obtained by multiplying by a factor of 2 the time of exposure and by subdividing by a factor of 2 the aperture area.
So, the f-number progression is by multiplying it for $\sqrt{2}$: since the aperture area is proportional to the square of the diameter, if we subdivide the aperture area by two, the diameter is reduced by $\sqrt{2}$ and so we obtain this result by multiplying the $f - number$ by $\sqrt{2}$.
In Figure 7.31 several pairs of shutter speed and aperture: as we can see, with a larger aperture, there is a lower time exposure and when the aperture is smaller, there is the need of higher time exposure in order to capture the same light.

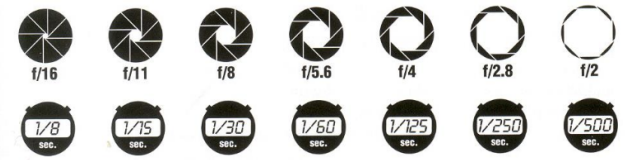


Figure 7.31: Problems of transformation

The choice on shutter speed controls freeze motion vs motion blur.
Longer exposure time means more light but more **motion blur** and a shorter exposure time means less light but a better **freeze motion** as we can see from Figure 7.32
In Figure different shutter speed compared: as we can see, using a very low time of exposure, also a rapid train can be captured freezed.
However, we don't automatically choose the smallest aperture since if the light is not enough, this means **diffraction effects**
The aperture **controls the depth of field**: a smaller aperture increases the range

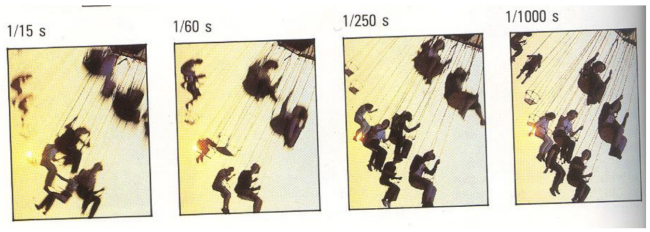


Figure 7.32: Problems of transformation

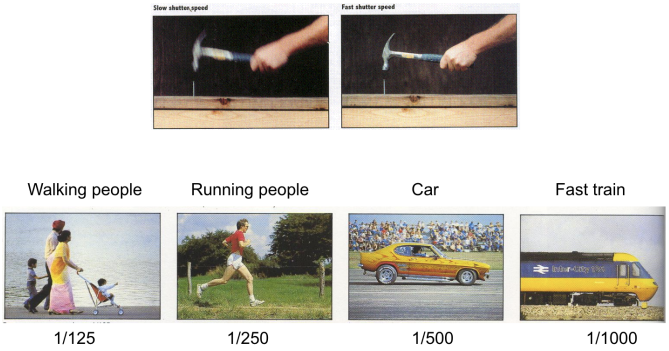


Figure 7.33: Problems of transformation

in which the object is approximately in focus.
The depth of field is the depth in which we can neglect the presence of the circle of confusion.
In Figure 7.33 we can see some examples

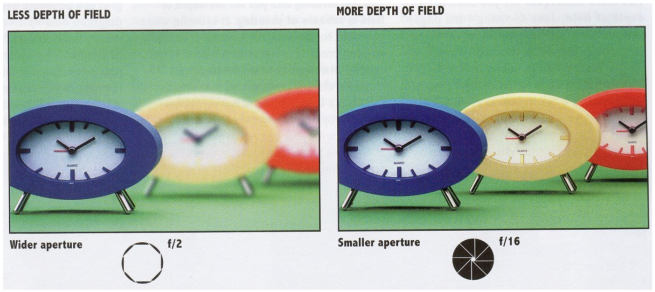


Figure 7.34: Problems of transformation

7.8 Object recognition

Consider the high-level task of getting some information from an image about objects.
In low level we focused on pixel, in mid level we focused on figures (lines, circles, edge), now the output is something understable to us.
Object recognition is a general term to describe a collection of related computer vision tasks that involve identifying objects in images or videos.
In Figure 7.34 different tasks with objects in computer vision.

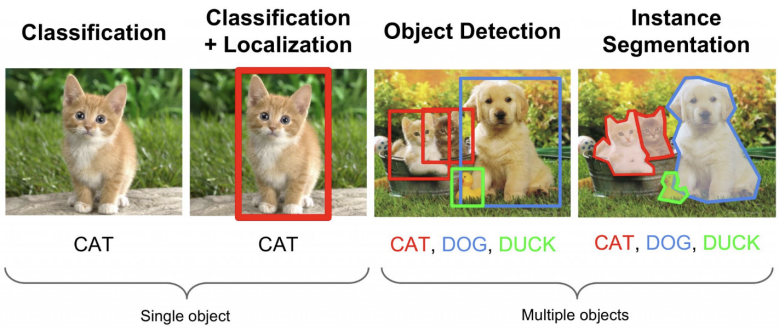


Figure 7.35: High level tasks

The tasks illustrated above have to give good result despite:

- Different camera positions
- Perspective deformations
- Illumination changes
- Intra-class variations

We already covered a method for object detection, the Viola and Jones detector for face detection, which is implemented with AdaBoosting. It can be applied to other target.
However, there are other approaches available.

7.8.1 Template matching

A **template** is something fashioned, shaped or designed to serve as a model, a representative instance, an example, as the coin in Figure 7.35.
The task is to find **instances of the templates in an image**, with a reasonable similarity measure chosen.

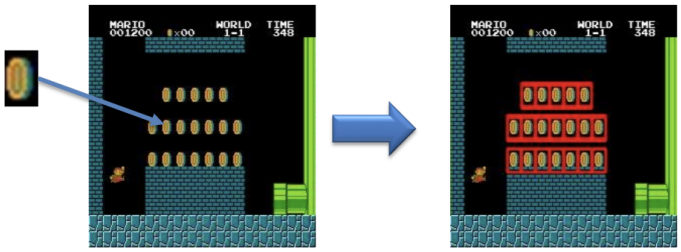


Figure 7.36: Example of template

It's a challenging task due to template variability, viewpoint changes, affine transforms as scale, rotations, translations, noise, illumination changes, as we can see from Figure 7.36

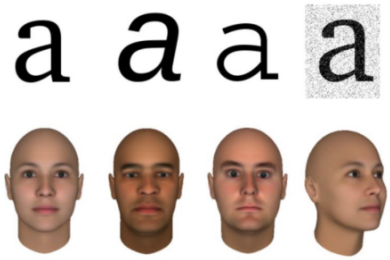
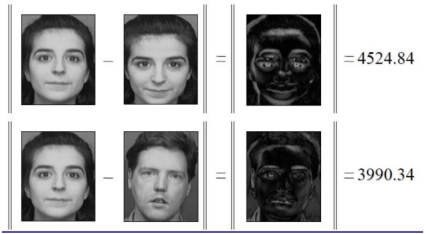


Figure 7.37: Challenges

Given an image and a template, how can we evaluate the match?
A common option is a **correlation based approach**. Suppose a template T, a rigid object, often a small image.
Template-based approaches are often based on a sliding window across the target image at various positions. The template is placed in every possible position across the image and, at each position, the region within the window is compared with the template to measure similarity.
Pixel values, features and edges or gradient orientation are compared. However, not always results are reliable, as we can see in Figure.



The absolute values of differences is higher while comparing the same face from different perspectives respect to two different faces.
For computing similarity, several **similarity metrics are used**, as we can see in Figure 7.37 (not asked).

- Sum of Squared Differences (SSD)

$$\phi(x,y) = \sum_{u,v \in T} (I(x+u, y+v) - T(u,v))^2$$

- Sum of Absolute Differences (SAD)

$$\phi(x,y) = \sum_{u,v \in T} |I(x+u, y+v) - T(u,v)|$$

- Zero-mean Normalized Cross-Correlation (ZNCC)

$$\phi(x,y) = \frac{\sum_{u,v \in T}(I(x+u, y+v) - \bar{I}(x,y))(T(u,v) - \bar{T})}{\sigma_I(x,y)\sigma_T}$$

- $\bar{I}(x,y)$: average on window, $\bar{T}$: template average, $\sigma_I(x,y), \sigma_T$: standard deviation on image window and on template

Figure 7.38: Similarity metrics used

However, template matching has some weak points and we have methods for resolving them:

- Dealing with illumination changes, we can use edge maps instead of images or also ZNCC which subtracts the uniform illumination component.
- Dealing with scale changes, we can do matching with several scaled versions of the template and we can work with multiple rescaled copies of images.
- Dealing with rotation, we can do matching with several rotated versions of the template

As we already said, TM is not an unique method but TM generate a family of approaches, as we can see from Figure 7.38, which differs in the definition of the template or in ways of dealing with the template.

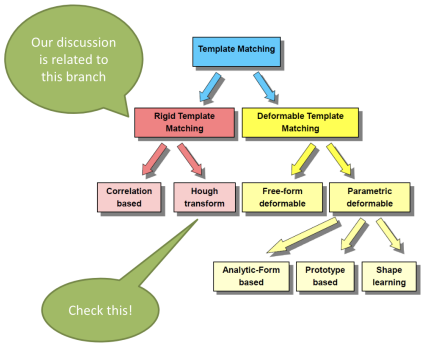


Figure 7.39: Different types of template matching

The generalized Hough transform (already seen) can be seen as a form of template matching, which works for more complex shapes.
As we can remember, the general equation is $g(\vec{v}, \vec{c}) = 0$ where $\vec{v}$ is a vector of coordinates and higher is its dimension, more difficult is the shape.

7.8.2 Histogram of Oriented Gradients (HOG)

HOG-based detectors work by sliding a window, characterizing it by evaluating the edge magnitude and phase (producing a descriptor) and classifying the descriptor.
The **HOG-descriptor** evaluation works as follow:

1. Firstly it's done an intensity normalization/histogram equalization + smoothing to the image
2. An edge map (magnitude + phase) is computed. This involves calculating the derivative of the image intensity values in both the horizontal (x) and vertical (y) directions. Common methods to compute gradients are using filters like the Sobel operator.

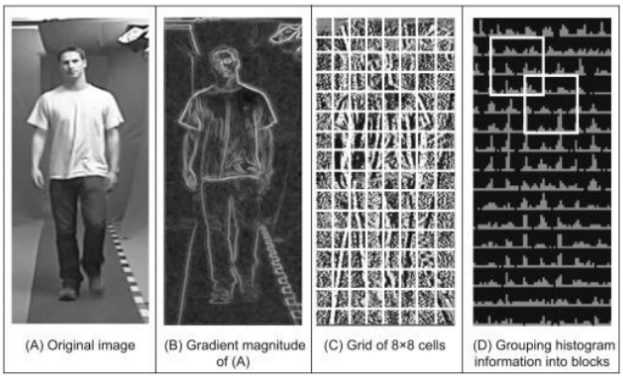


Figure 7.40: HOG descriptor

3. The image is divided into small connected regions called cells (typically of size $8x8$ pixels).
   For each cell, a histogram of gradient directions (or orientations) is compiled. The magnitude of each gradient contributes to the histogram bin corresponding to the gradient's direction. For example, if we use 9 bins, each bin might represent an angular range of 20 degrees (if we consider 0-180 degrees).
   For each pixel in the cell, we assign the gradient magnitude to the appropriate histogram bin based on the gradient direction.
   With this step we create **voting vectors**, of number of bins dimension.

Then we create multiple blocks, in the image a 3x3, we normalize each block (subdividing it by the sum of cells), and we create a $9x9$ block vector (serialize the block vector)

It's important to notice that the number (cell and block size) described above are one possible implementation for HOG. This has an influence on the descriptor size and meaning. Consider a 64x128 window: we have 8x16 cells, 7x15 blocks (since we take one block for each cell, and the last column hasn't a block). So, in overall, we have 105 block with 36 elements, so a total of 3780 elements.
The HOG descriptor characterizes the content of a bounding box and the HOG approach need bounding boxed normalized to a standard size.
The HOG descriptor is commonly used to train a classifier.

7.8.3 Bag of words

This approach is "stolen" from document analysis. The idea is shown in Figure 7.41
We want to do image and object classification and our system is invariant to several factors like different viewpoints and deformations.

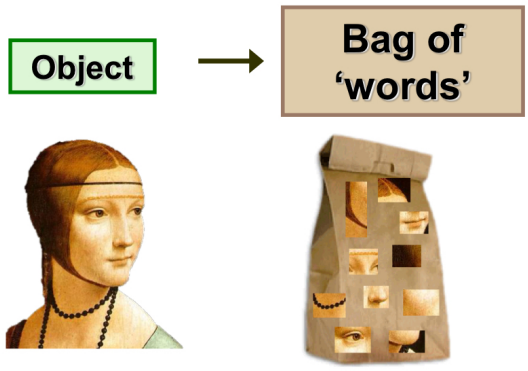


Figure 7.41: Bag of words idea

The idea is to decompose **complex patterns** into semi independent features, or **visual words**, from which we can extract also a **histogram representation**, as shown in Figure 7.42
Visual words can be represented using **features** (SIFT, ORB, GLOH) which exploit discriminative and invariance properties and uses an efficient description.
So, using BoW for image classification works as follows:

1. We have a dataset and we just extract features from each image, so keypoints and descriptor, as seen in Figure, memorizing them.

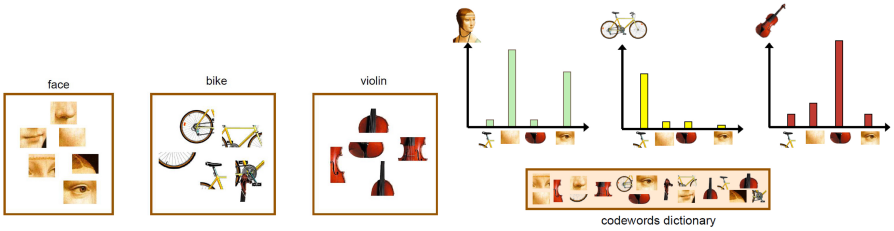


Figure 7.42: Visual words + histogram

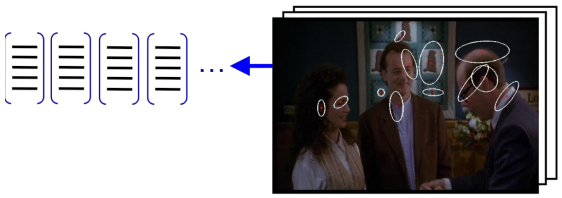


Figure 7.43: Keypoints as features

2. Then, in the feature space, is done **clustering** on descriptors, grouping features with same descriptor. Then, **codebook** is generated: each cluster generates a **representative sample**, used as a **visual dictionary's vocabolary**.
   Finally, for each image, we make frequency histogram from the vocabularies and the frequency of the vocabularies in the image. Those histograms are our bag of visual words (BOVW).

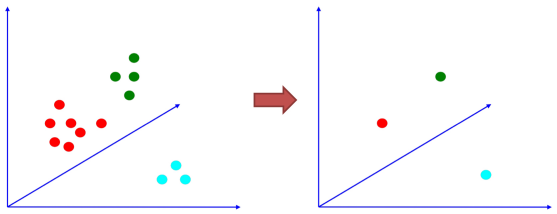


Figure 7.44: Bag of visual words

3. Once we have generated the codebook, we have our visual words and our histograms.
   So, for classifying an image, we evaluate features of that image and create an histogram, evaluating the occurrence of each word (feature descriptor) in the codeword. By using bag of visual words representation from our dataset, we can compute this image's nearest neighbors. We can do it by using nearest neighbors algorithm or another algorithm.

8

Deep Learning

Before talk about deep learning, we need to talk about **machine learning**: it's a field of computer science which investigates how it is possible to learn from data, where the goal is to make predictions from data.
As we can see from Figure 8.1, from labeled data, it's constructed an algorithm capable of learning a model which it's able to make prediction on unlabeled data.

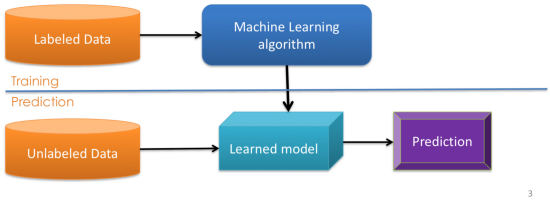


Figure 8.1: Machine learning basics

- Input: dataset of input and target pairs.
- Goal: find f^* that best approximates the unknown function f.
- f^* is parametric, optimal parameters need to be found.
- Supervised learning: desired outcome is available.
- Unsupervised learning: desired outcome is not provided.

A simple case, with a linear model

$$f(\vec{x}_i) = W\vec{x}_i$$

Where $W : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a linear operator mapping the observation space into the target space. Simple and easy to understand, closed form solution may be available.
However, non-linear phenomena cannot be accurately described. Non-linear phenomena can be linearized using a kernel function such that a linear model can be applied.
However, kernel is often application-dependent, hard to generalize.
The input of machine learning is data, which represent a given phenomenon. Talking about CV, data are images.
In traditional machine learning, the representation of images as input is critical for effective learning.
Images are **inherently high-dimensional data**, often represented as vectors of pixel values. For example, a grayscale image of size 100x100 pixels would have 10,000 dimensions (one for each pixel). When dealing with color images, the number of dimensions increases further as each pixel typically has three values (red, green, blue), leading to 30,000 dimensions for a 100x100 image.
Here happens the **curse of dimensionality**: learning is not effective when the input space has too many dimensions.
To address these challenges, traditional machine learning relies on **manual feature extraction**. This process involves transforming the raw high-dimensional data into a lower-dimensional, more meaningful representation. Techniques used in manual feature extraction include PCA, SIFT, HOG etc.
Here it's worth to talk about **deep learning**: also deep learning learns data representations, and as we can see from Figure 8.2, the main difference with Machine Learning, is that is not needed the feature extraction; handcrafted features might be incomplete and require a lot of human work to be designed and tested.

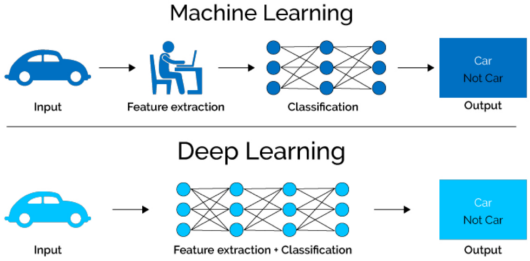


Figure 8.2: Deep learning vs Machine learning

Instead, **learned features** from a NN are adapted, constructed based to the input data. In deep learning data extraction and classification are done into a single neural network and that's why it's called **end-to-end learning**

8.1 Deep Learning basics

In Figure 8.3 a **deep neural network**, where basic elements are h the activaction function, and f, functions depending on the specific layer.

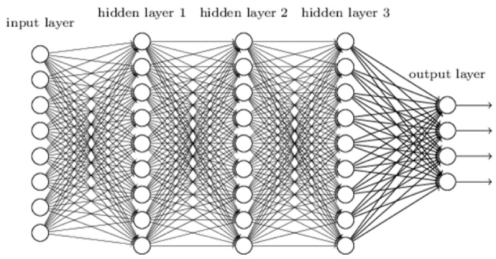


Figure 8.3: Deep NN

Each layer is made of a set of neurons and NNs include non-linear elements, in order to learn more complex pattern respect to linear model.
The input to the neurons of one layer are the output of the neurons in the preceding layer while the output of a neuron is activation function applied to a weighted average of the inputs plus a bias.
There are several types of activaction function:

- **Sigmoid Function**: Defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  Output range: (0, 1). Commonly used in binary classification problems. The derivative is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

  When the input is very large or very small, the gradient approaches zero, leading to the vanishing gradient problem, which can slow down the training process significantly. The **vanishing Gradient Problem** occurs during the training of deep neural networks when the gradients of the loss function with respect to the weights become very small.
  This leads to extremely slow updates to the weights, effectively preventing the network from learning. It is especially problematic in networks with many layers, where gradients are propagated through several layers during backpropagation.

- **Tanh Function**: Defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  Output range: (-1, 1). Zero-centered, which can be advantageous over the sigmoid function. The derivative is:

$$\tanh'(x) = 1 - \tanh^2(x)$$

  Similar to the sigmoid function, when the input is very large or very small, the gradient becomes very small, leading to the vanishing gradient problem.

- **ReLU (Rectified Linear Unit)**: Defined as:

$$\text{ReLU}(x) = \max(0, x)$$

  Output range: $[0, \infty)$. Introduces sparsity by zeroing out negative inputs, leading to efficient computation. ReLU helps mitigate the vanishing gradient problem as it has a constant gradient of 1 for positive inputs.
  However, it can suffer from the **dying ReLU problem** where neurons can get stuck during training, outputting zero for all inputs if they fall into the negative region. Variants like Leaky ReLU and Parametric ReLU (PReLU) address this by allowing a small, non-zero gradient when the input is negative.

In Figure 8.4 a NN layer: as we can see, despite very few layers and neurons, the **learnable parameters** are enough.

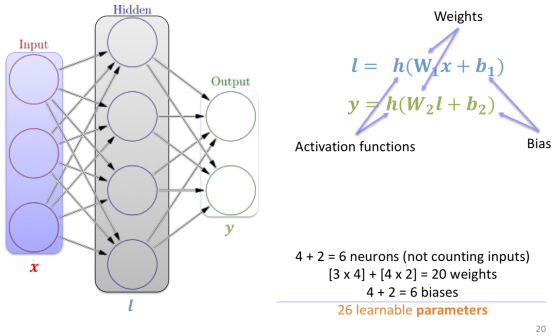


Figure 8.4: NN layer: a lot of parameters

Usually for **classification** are used **fully connected multilayer feedforward network**, composed by several layers, without loop and with an high number of connections. Usually the number of output neurons is equal to the number of classes to classify.
Training is done by defining a cost function, that is propagated through the network and, depending on the error between desired and actual values, weights are updated using **backpropagation**.

8.1.1 Convolutional Neural Networks

In traditional machine learning, achieving invariance to transformations such as rotation, scaling, and translation in images typically requires a vast and diverse dataset.
This diversity helps the model learn to generalize across different variations of the same object or scene.
For example, to recognize a cat in various poses and lighting conditions, the model needs to be trained on many images of cats in those various conditions.
This manual process is not only time-consuming but also requires extensive domain knowledge to manually engineer features that capture these invariances.
Convolutional Neural Networks (CNNs) address these limitations by automatically learning features that capture spatial hierarchies and invariances directly from the data. CNNs consist of several key components that enable them to effectively handle image data.
First of all, in order to achieve these properties, we have a first layer that is called the **convolutional layer**.
The convolutional layer works as follow:

- The input image is fed into the CNN. The **local connectivity** mechanism ensures that only a small patch **(receptive field)** of the image is considered at a time.
  In a CNN, each output neuron in the convolutional layer is connected to a local region of the input image, known as the receptive field. This means that instead of being connected to every pixel in the image, a neuron is only connected to a small, localized patch of pixels. This local connectivity helps the network focus on small, meaningful features like edges or textures.
  The **local connectivity concept** can be seen also here: supposing that each neuron is a pixel, we can see from Figure 8.5 that only few neighboring nodes are connected (**receptive field**)

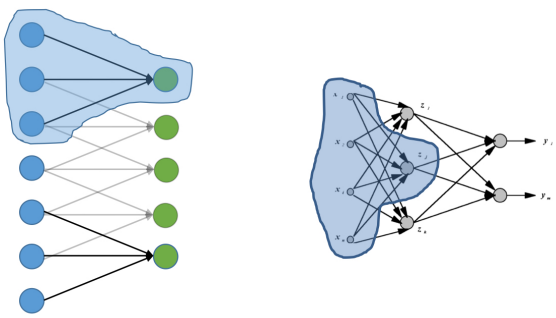


Figure 8.5: Some pixels mapped to a single neuron

Moreover, it's important to see that a pixel (a node) appears in more than once receptive field.

- A convolutional filter (with **shared weights**) is applied to the receptive field, followed by the addition of a bias term and an activation function (such as ReLU). This process is repeated across the entire image to produce a **feature map**.
  The same set of weights (i.e., the same filter) is used to scan across the entire input image. This concept of shared weights means that the learned feature detector (e.g., an edge detector) is applied consistently across different parts of the image. This leads to spatial invariance, meaning the feature can be detected anywhere in the image.
  As well explained from Figure 8.6, all green units share the same parameters but operate on a different input window.

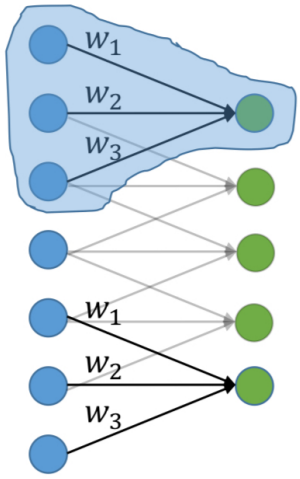


Figure 8.6: Shared weights

Remember that there is **convolution**, so the same neuron (the same pixel) appear in more than one computation.

- Results of this operation are mapped into a **feature map**, that is simply the application of a filter to the image (plus other operation, like activation function).
  So the output layer of the filter application on a receptive field is a single neuron (a pixel) of the feature map.
  However, **multiple filters** are used in the convolutional layer, each producing a different feature map. Each filter detects a different type of feature, such as vertical edges, horizontal edges, or textures and, moreover, different activaction functions are used.
  The output of the convolutional layer is a stack of these feature maps.
  This idea can be seen in Figure 8.7: brown and green neurons belong to different feature maps, which operate on the same window using different functions (e.g different filters).

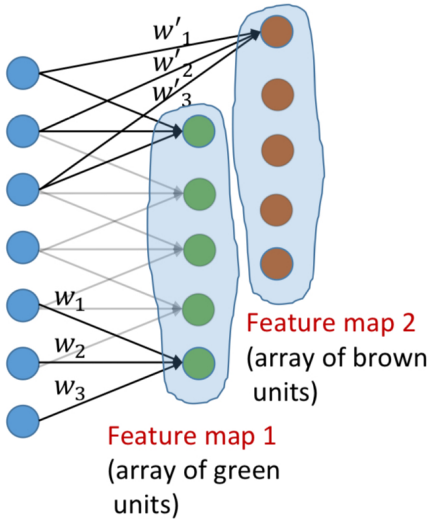


Figure 8.7: Multiple feature map

The next step is **subsampling or pooling**: the main goal of subsampling (or pooling) is to progressively reduce the spatial dimensions of the feature maps, which helps to decrease the computational load for subsequent layers, reduce the number of parameters, mitigating overfitting and introduce some form of translation invariance by summarizing the features.
Pooling is simply an application of operation (e.g max or average) to a region, in order to obtain a single point from different points.
It's important to talk about **pooling area and pooling stride**: the pooling area is the region considered for the operation, the pooling stride is the step size, how much we move from one operation to another. An higher stride educes the amount of overlap between the window positions and results in a smaller output feature map.
There are two main types of pooling:

- **Max pooling**: Selects the maximum value in each window, reducing dimensionality while keeping important features (pixels).
- **Average pooling**: Takes the average value in each window, offering an alternative way to reduce dimensionality.

Pooling is applied to each feature map so, as output, we will have a stack of small feature maps.
This process is repeated more times until we have sufficient features for training our neural network, a **fully connected neural network**.

The process is shown in Figure 8.8 So, **summarizing**: Each pixel of an image is

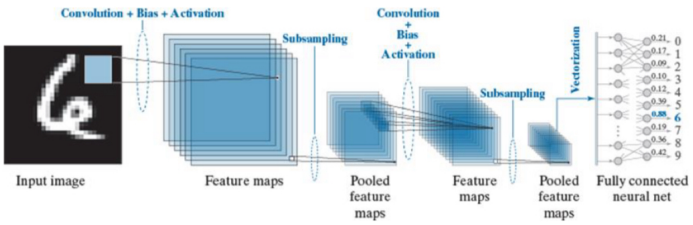


Figure 8.8: Full process pipeline

represented as a value inside a neuron. For example, a 28x28 grayscale image has 784 input neurons, each corresponding to a pixel intensity. In the convolutional layer, a small patch of the image, known as the receptive field, is taken, which includes a few neurons (pixels).
A filter (kernel), which is a small matrix of weights, is applied to this patch. The filter performs a convolution operation, mapping these pixels to a single value in the feature map.
This process is repeated across the entire image by sliding the filter over the input image with a defined stride, ensuring that the entire image is covered.
The result is a feature map where each value represents the presence of a specific feature detected by the filter in different parts of the image.

The same filter (weights) is used for each receptive field across the image. This means that the parameters (weights) are shared across different parts of the image, ensuring spatial invariance.
Pooling layers follow the convolutional layers to reduce the spatial dimensions of the feature maps. The pooling area defines the size of the pooling window (e.g., 2x2), and the stride determines the step size for moving the pooling window across the feature map.

The benefits of pooling include dimensionality reduction, which reduces the size of the feature maps and decreases the computational load and the number of parameters.
Pooling also provides translation invariance by summarizing the features, making the network more robust to small translations or distortions in the input image. Additionally, pooling helps in reducing noise by focusing on the most significant features.

By leveraging local connectivity, shared weights, and pooling, Convolutional

Neural Networks (CNNs) efficiently detect and learn hierarchical features from images, enabling powerful and scalable image processing capabilities.
Considering the MNIST dataset, that is a database of handwritten digits, it's used in many CNN examples.
Consider a CNN to solve the MNIST problem: we need to choose a structure (so the number of trainable parameters).
Considerin an example with $5x5$ convolutional filter and $6/12$ different feature maps: there are many parameters to learn!
In Figure 8.9 is represented what happens to a single image after the first convolutional layer.

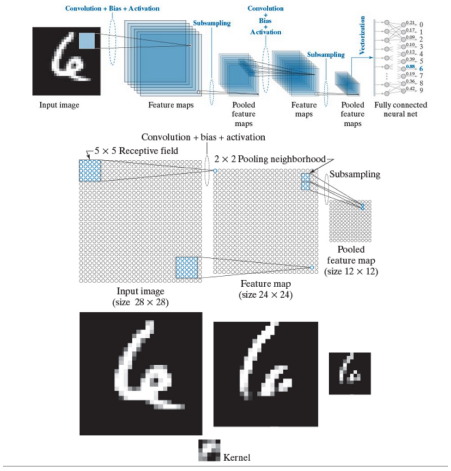


Figure 8.9: Running of first convolutional layer on an image

Instead, in Figure 8.10 we can see that from an image, after the first convolutional layer, 6 different convolutional filters are created, with weight determinated by learning, and, in the second layer,12 other kernels are created, one for each image of the feature maps after the first layer
In Figure 8.11 an example of the computation of feature maps from a single image and, in Figure 8.12, the pipeline of the process with some results.
A typical problem which could occur is **overfitting**: the model is too complex w.r.t. training data and it really knows only the training set. So, it will predict in an uncorrect way new elements.
Dropout is a mechanism which randomly suppresses some random neurons, which will be not active in that layer. Each unit has a probability p of being retained.
Early stopping is the using of **validation error** for deciding when training should be stopped; it stops when monitored quantity is not improved after n subsequent

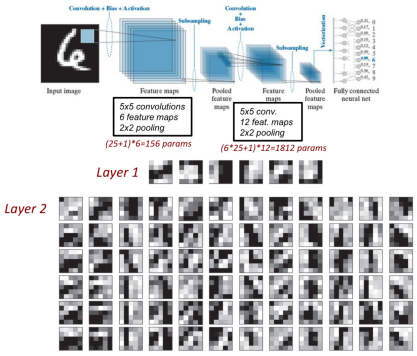


Figure 8.10: Filters after first two convolutional layers

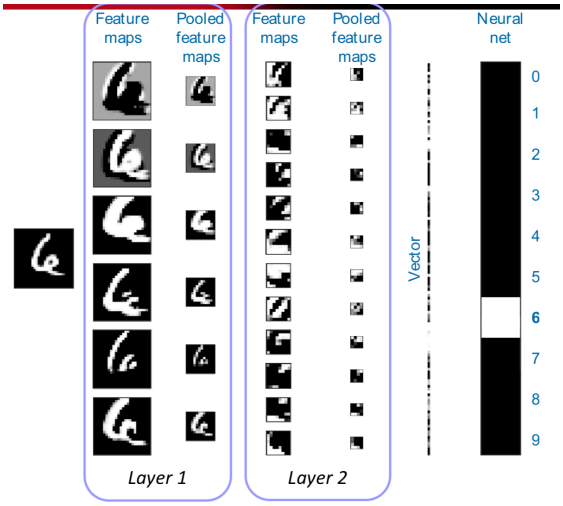


Figure 8.11: Results obtained after a single image as input

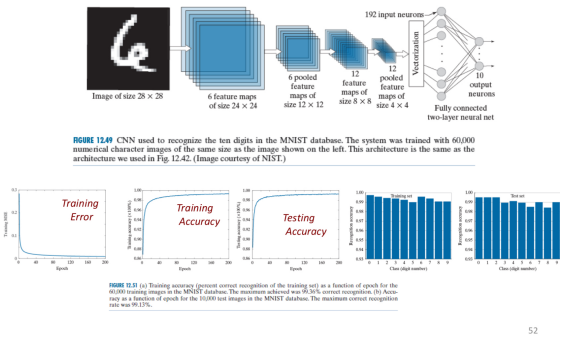


Figure 8.12: Accuracy of the model

epochs.
It's also very important to **pre-process** our data: we add some variance such as noise, rotation, color, saturation. This process is also called **data augmentation**.

8.2 Transfer learning

In order to reduce the computational costs and data costs, leveraging on **transfer learning** could be useful.
It's a technique very useful when we don't have a large dataset: training a CNN could be very data consuming. This technique exploits already trained network, reducing time and data required.
In a generic NN, when we train a classifier, we can see from Figure 8.13, as long as our network goes near to classification, the algorithm focuses on different things.

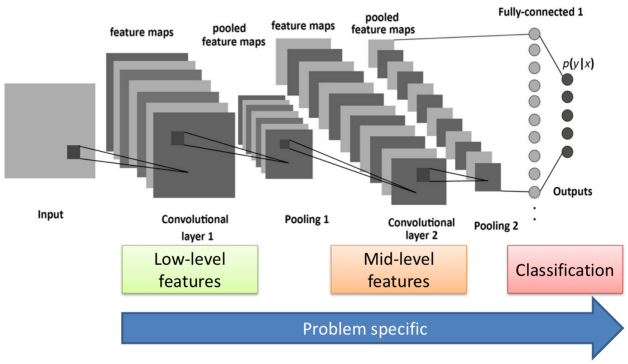


Figure 8.13: Process specification

Firstly on low level features and only at the end on single pixels. With this increase, **increase also the specificity of the problem**: so, the idea of transfer learning, is to insert in the middle, when the specification is not too high, and leverage on already trained parameters that could be shared across different applications.
A simple pattern could be to take a **pre-trained network**, reset the last layers, freeze the deeper layers and then train the only last layers on our small dataset.
As we said, until a certain point, parameters are similar across different problem: only when we move close to the end born the specification of the problem.
There are other different pattern for doing transfer learning:

- Exploit the output of pretrained networks as feature vector. Useful when the output stage provides multiple values.
- Exploit the features provided by inner layers. High feature dimensionality, reduction might be needed

These methods are shown in Figure 8.14.

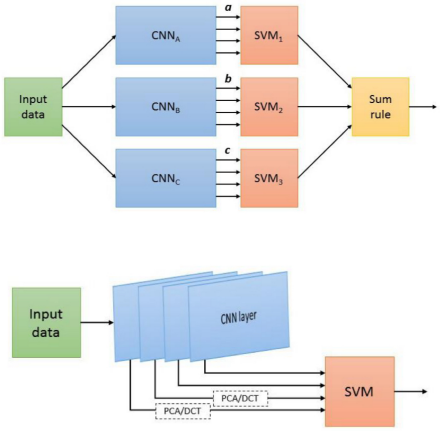


Figure 8.14: Patterns for transfer learning

Fine-tuning is slightly different: optionally reset or remove part of the network, typically at classification stage, and then resume training on new dataset.

8.3 Deep Learning for Object Detection and Segmentation

8.3.1 Object Detection: YOLO architecture

YOLO is a popular detector used for finding object.
From the architecture published in the original paper, a family of detectors was born.
YOLO is the acronym of **You Only Look Once** and that's why it's **one stage** or **proposal free**.
To understand what "proposal-free" means, it's useful to first understand proposal-based detectors. In proposal-based detectors, like the R-CNN family (including Faster R-CNN), the detection process involves two main stages:

- The algorithm first scans the image to **identify potential regions (or "proposals")** where objects might be located. This step generates a set of candidate regions that are likely to contain objects, but does not yet classify them.
- Each proposed region is then extracted and fed into a classifier to determine what object, if any, is present in that region.
  The bounding boxes are refined, and class probabilities are assigned to each region.

Instead, YOLO relies on a **unified detection**: both bounding boxes and category for each class are found in one pass.
As first, yOLO divides the input image into an SxS grid of equal size cells.
Each cell in this grid is responsible for detecting objects whose center points fall within the cell, where **center point** refers to the centroid or geometric center of the object's bounding box.
For each grid cell, YOLO predicts a fixed number B of bounding boxes. For each possible **bounding box within a cell** is computed a **confidence score** that refers to how confident the model is that the box contains an object.
Each bounding box is defined as $\{x, y, w, h.con\}$ where x, y are coordinates (relative to the bounds of the grid cell), w, h respectively weight and height of the bounding boxes and con that is the confidence score, ndicating the presence of an object and the accuracy of the bounding box. These quantities are determined by the regression problem.
This part was referred to the **bounding box detection**. Now we move to the

object classification.
Each grid cell also predicts class probabilities for the object within each bounding box: if there are C classes, each grid cell provide C class probabilities.
The probability are given assuming that an object is present in the cell. At test time, these probabilities are multiplied for the confidence score, so how much we are confident that there is an object in the cell.
The idea can be seen in Figure 8.15 and, in Figure 8.16, we can see the architecture used for implementing this method.

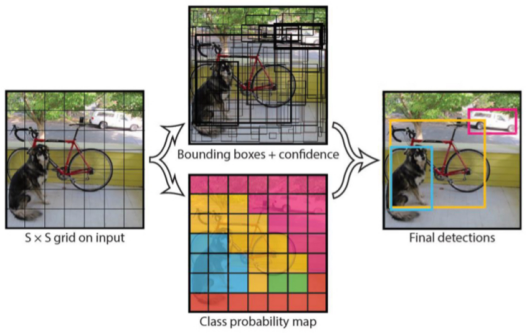


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

Figure 8.15: Idea of the method

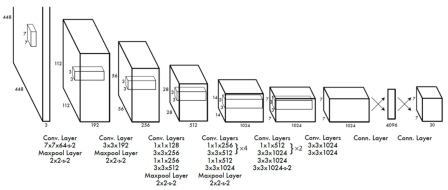


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

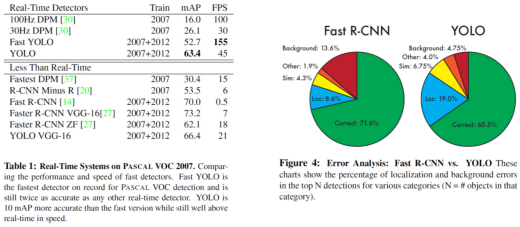
Figure 8.16: Architecture used for implementing the method

The shape of the output layer depends on the number of bounding boxes B, the number of categories C, which comes from the dataset, and the number of cell SxS.
YOLO is trained on ImageNet or Pascal VOC as follows:

- Pre-training of the first 20 convolutional layers on ImageNet, at reduced resolution

- Training of the whole network on Pascal VOC

YOLO is evaluated based on mean Average Precision metric and on speed, and, as we can see from Figure 8.16, it has very good results.



Real-Time Detectors	Train	mAP	FPS
100Hz DPM [30]	2007	16.0	100
30Hz DPM [30]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [37]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[27]	2007+2012	73.2	7
Faster R-CNN ZF [27]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

Table 1: Real-Time Systems on PASCAL VOC 2007. Comparing the performance and speed of fast detectors. Fast YOLO is the fastest detector on record for PASCAL VOC detection and is still twice as accurate as any other real-time detector. YOLO is 10 mAP more accurate than the fast version while still well above real-time in speed.

Figure 4: Error Analysis: Fast R-CNN vs. YOLO These charts show the percentage of localization and background errors in the top N detections for various categories (N = # objects in that category).

Figure 8.17: Metrics and result

8.3.2 SEGMENTATION: U-NET

U-Net is a popular convolutional neural network architecture primarily used for image segmentation tasks.
U-net derives from the fact that the architecture has the shape of a U, as we can see from Figure 8.17

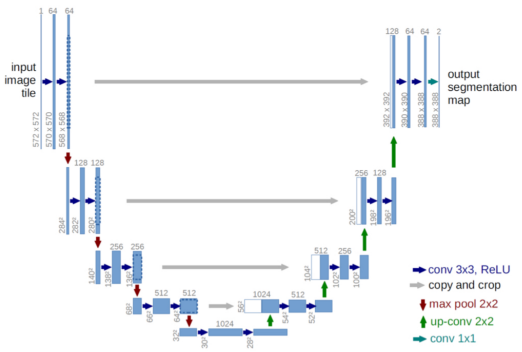


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

Figure 8.18: U-net architecture

It's a fully **convolutional network**, originally designed for biomedical tasks.
The working principle is the following:

- **Contracting path (Encoder)**: The encoder is similar to a traditional convolutional network, consisting of repeated application of two 3x3 convolutional layers (with ReLU activation) followed by a 2x2 max-pooling operation with stride 2 for downsampling.
  The number of **feature channels** (feature maps, defined by the paper at 64) doubles at each downsampling step.

- **Expansive Path (Decoder)**: The decoder consists of an **upsampling** of the feature map followed by a 2x2 convolution ("up-convolution") that halves the number of feature channels.
  The upsampled feature map is concatenated with the corresponding feature map from the contracting path (skip connections).

As last layer, a $1x1$ convolution for mapping into the number of categories.
Data augmentation is strongly needed, in order to be invariant to shift, rotations, smooth deformations.