COMPUTER VISION

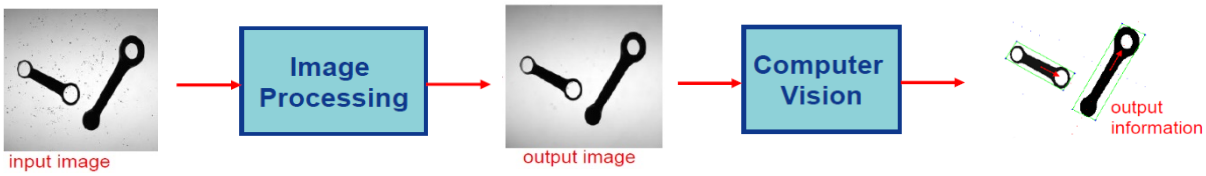
Prof. Di Stefano

Laura Gruppioni

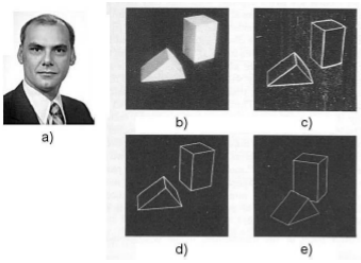
SOMMARIO

0 – INTRODUCTION

Computer vision deals with extraction of information from images, so we have an input image and some information as output. In Image Processing, on the contrary, we have an input image and an output image, usually improved. Quite often, Image Processing helps Computer Vision.



In the first image we have noise and with Image Processing we can filter that noise in order to obtain a clearer image. After that, the Computer Vision System can isolate images in which it is interested in and it can elaborate them to solve the problem of object recognition, for example. It can retrieve the objects' position or their orientation.

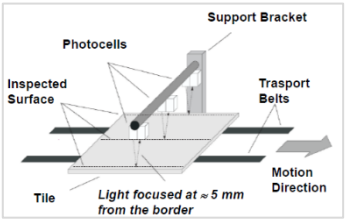


ROBERTS (MIT, 1965) invented the "Block World" in which he tried to make computers able to recognize simple shapes. He's considered the father of CV. After more or less 20 years of research, robots started to do this act of recognition in factories.
Through successes and failures, computer vision has eventually reached maturity. The technology is now as effective, reliable and affordable as to be deployed in countless industrial applications.

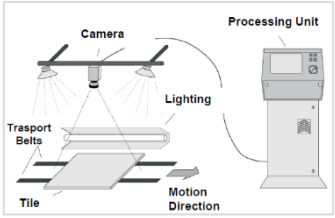
Today CV is a key process technology in most industries regarding food, tobacco, packaging, pharmaceuticals, printing and automotive. Particularly, it deals with:

- **Inspection**: we have a camera that controls the quality of the product. This approach is used in many factories because it is more accurate, fast and efficient than the human eye. Furthermore, for humans this job is boring and tiring
- **Gauging**: similar to inspection, but we take also some measurements
- **Guidance**: related to automotive in which we need to guide vehicles or robots

A CASE STUDY: TILE INSPECTION



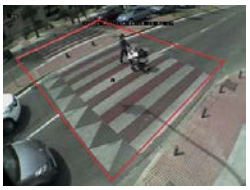
A factory needs to control if the tiles if produces are broken or have some manufacturing defects and this process needs to be done before the tile is painted.
In the first image, the factory has installed a system in which light doesn't cover the whole tile, it has 3 photocells. If tiles have small rotations and are not perfectly positioned, we can have false positives.
Furthermore, if we change the dimensions of the tile, we need to recalibrate the photocells, so we have poor flexibility.
In the second image, we have the solution, i.e. a camera that covers the whole border of the tile using a computer vision approach that is significantly more effective.



e.g. Pancake picking: https://www.youtube.com/watch?v=wg8YYuLLoM0

Some Enterprise Products



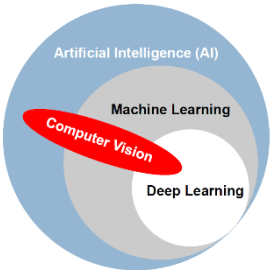

Some Mass-Market Consumer Products



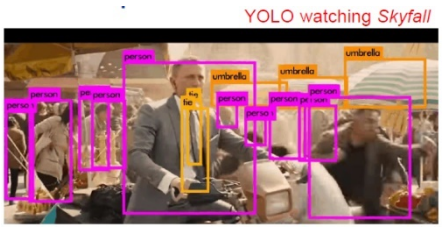

KEY ADVANCES → **FEATURES**

- **LOCAL INVARIANT FEATURES**: the detector/descriptor paradigm allows finding and matching salient regions across images invariantly to viewpoint changes, e.g. SIFT
- **AFFORDABLE DEPTH CAMERAS**: key to controlling machines by gestures and/or body movements (Natural User Interfaces), e.g. Kinect for Xbox

AI: any technique that enables computers to mimic human intelligence.
Machine Learning: computers learn from examples rather than being explicitly programmed.
Deep Learning: Machine Learning realized through Deep Neural Networks. Here we use a truly "data-centric" paradigm in which we use Big and Good Data and the value shifts from domain knowledge to data



Example of Computer Vision applied to Deep Learning. The problem is that sometimes it is incorrect.



Linked/Augmented Life: Devices based on geo-localization, inertial sensors and computer vision may link seamlessly digital content to physical objects.

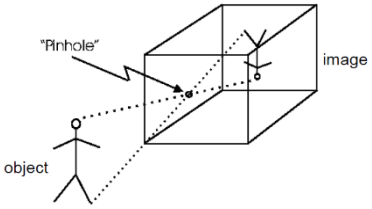
1 - IMAGE FORMATION AND ACQUISITION

An **imaging device gathers the light reflected by 3D objects to create a 2D representation of the scene**, i.e. the image. In computer vision we basically try to invert such a process, so as to **infer knowledge on the objects from one or more digital images**. It is therefore worth to understand the image formation and acquisition process. This requires studying:

- The **geometric relationship** between scene points and image points
- The **radiometric relationship** between the brightness of image points and the light emitted by scene points
- The **image digitization process**

At the beginning, we'll see how digital images are created and acquired. We need to infer information on those images. The dual problem of Computer Vision is Computer Graphics in which we give information on the objects and images appears. In Computer Vision we have images, but we need to extract information on the objects.

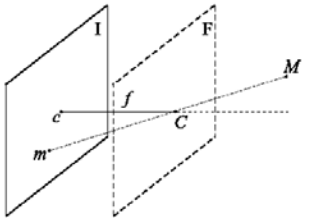
1.1 – PERSPECTIVE PROJECTION



1.1.1 - PINHOLE CAMERA

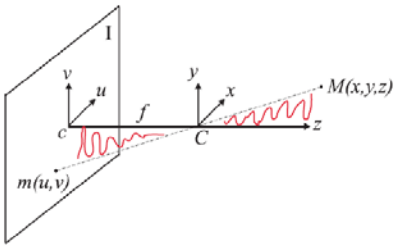
The "pinhole camera" is the **simplest imaging device**: lights goes through the very small pinhole and hits the image plane. **Geometrically, the image is achieved by drawing straight ray from scene points through the hole up to the image plane.**
Although useful images can hardly be captured by means of a pinhole camera, its remarkably simple geometrical model turns out a **good approximation of the geometry of image formation in most modern imaging devices.**

The geometric model of image formation in a pinhole camera is known as **PERSPECTIVE PROJECTION.**



- **M**: scene point
- **m**: corresponding image point
- **I**: image plane
- **C**: optical centre
- **f**: focal length
- **F**: focal plane
- **Optical Axis**: line through C and orthogonal to I
- **c**: intersection between optical axis and image plane. Called "image centre" or "piercing point"

1.1.2 - COORDINATES



We need to understand how to create an image and how to process it considering the geometric pov.
The following equations have a specific meaning: they give us information on the **coordinates of the objects**.
So, considering the reference frames shown in the figure, the equations to map scene points into their corresponding image points are as follows:

$$\frac{u}{x} = \frac{v}{y} = -\frac{f}{z} \qquad \rightarrow \qquad u = -x\frac{f}{z} \qquad v = -y\frac{f}{z}$$

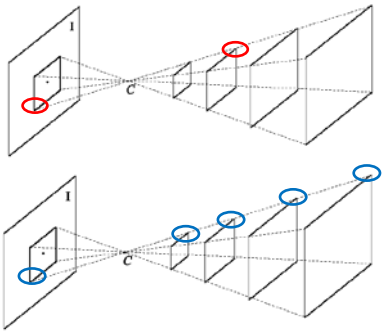
To get rid of the up-down and left right inversions, **the image plane can be thought of as lying in front rather than behind the optical centre, so:**

$$u = x\frac{f}{z} \qquad v = y\frac{f}{z}$$

Points that are close to the camera are bigger, points that are far from the camera are littler. Higher is the distance from the camera, lower is the coordinate. These equations are **NON linear**.

1.1.3 – IMAGE FORMATION PROCESS

The image formation process deals with **mapping a 3D space onto a 2D space**, thus leading inevitably to loss of information. Indeed, the **mapping is not a bijection**: a given scene point is mapped into a unique image point, but a given image point is mapped onto a 3D line, i.e. the line through the point m and the optical centre C.



Thus, **recovering the 3D structure of a scene from a single image is an ill-posed problem** (the solution is not unique), as once we take an image point we can only state that its corresponding scene point lay on a line, but cannot disambiguate a specific 3D point along such a line, i.e. we know nothing about the distance to the camera.

If we take the corner of the second rectangle on the right (red), its mapping is unique. But if we start from the first rectangle on the left (blue), this corresponds to every corner on top of the rectangles on the right: there's not a unique solution. In practice, from a 2D image we can't have exact information on the 3D structure.

We know that there are several sensors or in general several approaches that allow us to recover information of 3D images and the most used is **Stereo Imaging**.

1.1.4 – PROPERTIES OF THE PERSPECTIVE PROJECTION

The farther objects are from the camera, the smaller they appear in the image. As a matter of example, the image of a 3D line segment of length L lying in a plane parallel to the image plane at distance z from the optical centre will exhibit a length given by:

$$l = L\frac{f}{z}$$

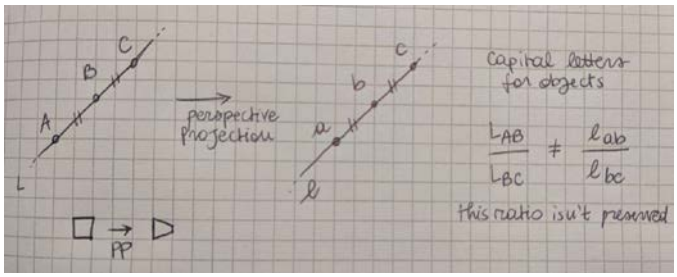
An object can have a big scale and appear big, or a small scale and appear small, but it depends both on the dimension of the object and its distance from the camera. A segment with length L will be scaled in relation to the depth and the dimension of the image will be inversely proportional to that depth.

The relationship turns out more complicated for an arbitrarily oriented 3D segment, as its position and orientation need to be accounted for as well. Nonetheless, **for a given position and orientation, length always shrinks alongside distance.**

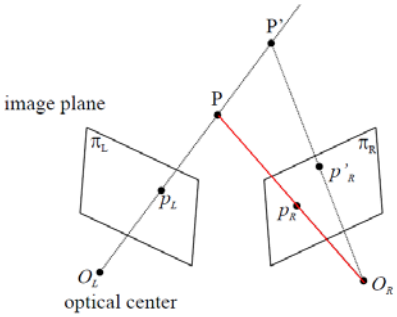
If the line is oblique, the segments seems smaller, so the ratio of the lengths are not preserved. If we know that our object has certain specific proportions among its parts, those proportions are not equal in the image and this is caused by the perspective projection. A rectangle, could become a trapezoid. Another property that we lose with the Perspective Projection is that parallel lines are no more parallel in the image plane.

To resume:

- Perspective projection maps 3D lines into image lines
- **Ratios of lengths are not preserved**, unless the scene is planar and parallel to the image plane
- **Parallelism between 3D lines is not preserved**, except for lines parallel to the image plane



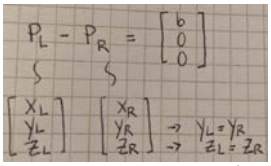
1.2 – STANDARD STEREO GEOMETRY

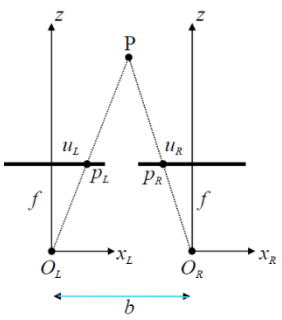


If we consider only the first image, we don't have information on the depth, on the 3D coordinates of the image. Nonetheless, if we take the second image and we consider a point PR, **assuming that PR** (Point Right) **and PL** (Point Left) **are corresponding points, we can see that their projection corresponds, is unique.**

There are different Stereo Devices with 2 cameras perfectly calibrated to do this triangular projection and they are very used, e.g. to help navigation of drones to infer depth.

Typically, a stereo camera is similar to human eyes: cameras are put in a specific way because they are adjusted as in the human face. Here we don't talk about reference system with x, y, z but we have coordinates of the images that are required to measure the perspective: only in this reference system the coordinates exist, we can't see them in the real-world reference system. We need to represent those coordinates in the camera reference system and to do that we know that **the origin is the pinhole**. The **z axis is the optical axis**, and the **horizontal axis are parallel to the image's axis u**: in that way we can use the equation we've seen previously.





There's a constraint between the two camera reference systems: **those two systems allow only one kind of transformation, i.e. the horizontal translation**, in which the distance of translation is called "base line".

We have 6 degrees of freedom that need to be specified to define the change of position. In the standard stereo geometry, the two 3D systems given by the camera's frames are specified by a translation called "**BASE LINE**", i.e. b.

$$xL - xR = b \qquad yL = yL = y \qquad zL = zR = z$$

Since y and z are the same, the 2 v (that are the vertical coordinates) must be the same. X are different, so horizontal coordinates are different. We subtract the 2 horizontal coordinates of the image and we obtain the baseline multiplied for the scale factor f/z: this relation is called **disparity** (d). We want to find z, the distance of the camera's points.

$$vL = vR = y\frac{f}{z} \qquad uL = xL\frac{f}{z} \qquad uR = xR\frac{f}{z} \qquad \rightarrow \qquad uL - uR = b\frac{f}{z} = d$$

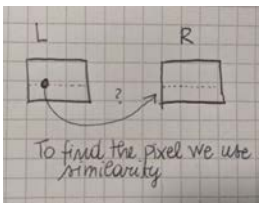
So, the distance of a point of the camera that doesn't change if comes from the right of the left because we calculated the baseline and it is given by this relation.
We can measure the distance of a point of our lens and it inversely proportional to the disparity. If we know the disparity, we know the depth. The scale factor is a parameter we need a priori, and the baseline too. We can calibrate them in the real world.
Now we can infer the position 3D of a point. Higher is d, lower is the depth, so the points are nearer to the camera. Lower is d, higher is the depth, so the points are more far from the camera.

$$d = b\frac{f}{z} \qquad \rightarrow \qquad z = b\frac{f}{d}$$

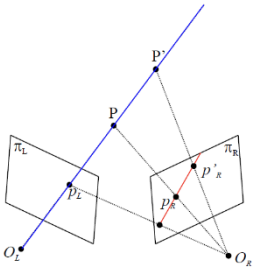
Experiment of the thumb near the nose: if we open and close one eye at a time, we can see that the thumb jumps so there's a very high disparity. That's because it is very close to our cameras.
If we keep our thumb far from our eyes, for example with our arm extended, the disparity is lower.

The equations we've seen previously tell us that correspondent points will be at the same height. The disparity is the difference between horizontal coordinates, but the equation in red related to the vertical coordinates is still useful.
We are not really sure that the pixel of the first image corresponds exactly to the pixel of the second image. Who tells us that every pixel of the image on the left has a correspondent pixel in the other image? There's a piece of hardware and software that does this matching, this correspondence.
Furthermore, there are pixels that I see in one image, but not in the other one: that fact depends on the area that every camera can cover. There are methods to do this matching finding point in common.

Sometimes people color a point with a color that is very different from its context to find correspondent pixels, but we can't imagine a drone that during its fly colors the world to find the depth of what it is interested in measuring.



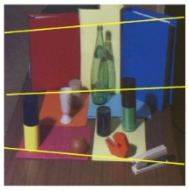
We need to control the **SIMILARITY**: if we take a pixel on the left, where can we find it in the image on the right? We need to look more or less at the same height.
So, the vertical position should be the same, we can avoid to scan the whole image, we need just to consider the same horizontal line. In that way, the similarity check is very much faster because points are in the same line.
The most important benefit of this approach is that we can use this property also for cameras that are positioned in an arbitrary way.



1.3 - EPIPOLAR GEOMETRY

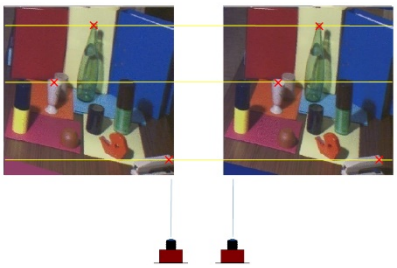
The search space of the stereo correspondence problem is always 1D!
All the epipolar lines in an image meet at a point called **epipole** (i.e. the projection of the optical center of the other image). However, searching through oblique epipolar lines is awkward.

We need to understand better how to find points in which we are interested in with cameras that are positioned arbitrarily. Let's consider where to find the points in the image: we don't know which points on the blue line correspond to the points on the right image. They could be 3, in this example. The red line is called epipolar. Every point in an image has an epipolar line in the other image, even if this line is not horizontal as in the stereo standard geometry, we can reduce the search because it is sufficient to find on this epipolar line and we don't need to control the whole image, we can scan the image with those oblique lines.



However, this is not simple as before with the horizontal lines, because the correspondent problem is solvable in a stereo geometry problem in an easy way, but in reality, we're not able to build a system like that in which it is sufficient to control the same line of the matrix that describes two images.

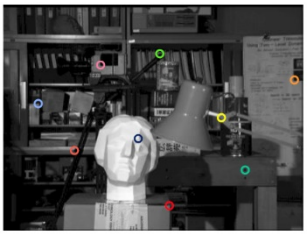
1.3.1 - RECTIFICATION



What we can do is taking two images in epipolar geometry and, with the help of a software, we can rectify them in order to make a comparison with the stereo geometry. Since the stereo geometry is impossible and the epipolar one is difficult to manipulate, we do this rectification as a trade-off.

More formally, we can always warp the images as if they were acquired through a standard geometry (horizontal and collinear conjugate epipolar lines) by computing and applying to both a transformation called **homography**, known as rectification.

THE STEREO CORRESPONDENCE PROBLEM



Given a point in one image (e.g. L) find that in the other image (R) which is the projection of the same 3D point. Such image points are called **corresponding points**. Basic cue: corresponding points look similar in the two images.

When one of the pixels is occluded in a camera, we can have non correspondent points and, in that case, we need to apply the similarity.

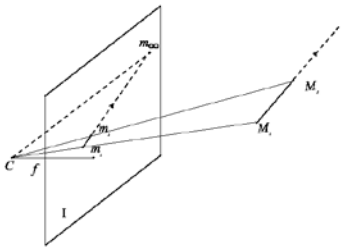
The Standard Stereo approach looks at the window of the image on top and tries to find correspondences in the other image along the same horizontal line.

1.4 - VANISHING POINTS

The images of 3D lines meet at a point, which is referred to as **vanishing point**. Vanishing points are image points that, for example, painters used to realize perspective paintings that seemed more real.

The vanishing point of a 3D line is the image of the point of the line which is infinitely distant from the optical centre. As such, it can be **determined by the intersection between the image plane and the line parallel to the given one and passing through the optical centre**. Accordingly, **all parallel 3D lines will share the same vanishing point**, i.e. the "meet" at their vanishing

point in the image, but in the "special case" of such a point being at infinity, i.e. the 3D lines are parallel to the image plane.
We project the M points along the line and we obtain some rays that ends in the vanishing point. We can simply draw a line parallel to the other. **When the line is parallel to the image plane, the vanishing point goes to infinite.**



Let's now find the vanishing point of the line:

$$M = M0 + \lambda D = \begin{bmatrix} x0 \\ y0 \\ z0 \end{bmatrix} + \lambda \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

where M0 is whatever point on the line and D is the direction cosines vector. It is a unit vector that represents the orientation of the line with the lambda parameter. If we want a point that is very far, lambda will be very high. Here we are representing a 3D object, a 3D line in the real world. Those coordinates will be represented in the camera reference system.
So, We first project a generic point of the line:

$$u = x\frac{l}{z} \qquad v = y\frac{l}{z} \qquad M\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x0 + \lambda a \\ y0 + \lambda b \\ z0 + \lambda c \end{bmatrix}$$

$$m = \begin{bmatrix} u \\ v \end{bmatrix} \qquad \boxed{u = f\frac{x0 + \lambda a}{z0 + \lambda c}} \qquad \boxed{v = f\frac{y0 + \lambda b}{z0 + \lambda c}}$$

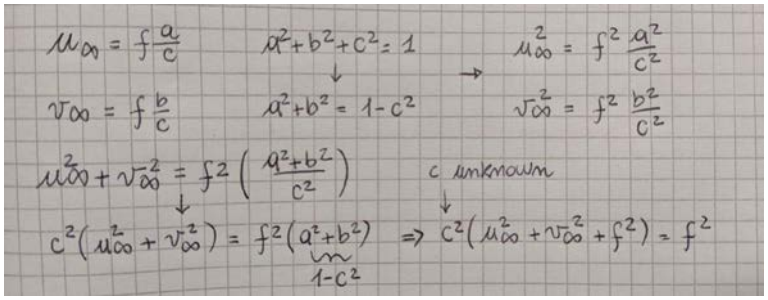
The vanishing point of this 3D line is the image in the special point on the line, i.e. the one that goes to infinite. Its coordinates will be bigger, so I project an arbitrary point with the coordinates rounded by the black rectangles. The point that I'm searching for is the point that goes to infinity, so I need to consider the equation of u in which lambda is very big: I'll take the limit.
Then, to get the vanishing point we consider the infinitely distant point along the line:

$$m\infty = \begin{bmatrix} u\infty \\ v\infty \end{bmatrix} \qquad u\infty = \lim_{\lambda \to \infty} u = f\frac{a}{c} \qquad v\infty = \lim_{\lambda \to \infty} v = f\frac{b}{c}$$

As expected, **the vanishing point depends on the orientation of the line only, not on its position**, and whenever the line is parallel to the image plane (c=0) it goes to infinity. It can also be shown easily that in such a case the image of the line as the same orientation as the 3D line.
Lines that are parallel to the image plane don't have a vanishing point.

1.4.1 – ORIENTATION OF PARALLEL LINES FROM VANISHING POINTS

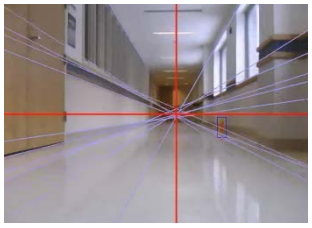
Knowledge of the vanishing point of a sheaf of parallel lines (and of the focal length) allows for determining the unknown orientation of the lines.



From the previous formulas we obtain that:

$$\begin{cases} a = \dfrac{u_\infty}{\sqrt{u_\infty^2 + v_\infty^2 + f^2}} \\ b = \dfrac{v_\infty}{\sqrt{u_\infty^2 + v_\infty^2 + f^2}} \end{cases} \quad \rightarrow \quad \boxed{\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \frac{1}{\sqrt{u_\infty^2 + v_\infty^2 + f^2}} \begin{bmatrix} u_\infty \\ v_\infty \\ f \end{bmatrix}}$$

A mobile robot can be driven through indoor hallways by tracking the dominant vanishing point and steering the robot so as to keep it at the centre of the image. In order to do that, the last relation is used.



1.5 - USING LENSES

A scene point is on focus when all its light rays gathered by the camera hit the image plane at the same point. In a pinhole device this happens to all scene points because of the very small size of the hole, so that **the camera features an infinite DEPTH OF FIELD (DOF).**
The drawback is that such a small aperture **allows gathering a very limited amount of light**. Thus, getting sufficiently bright images mandates very **long exposure times**. As a result, **only static scenes can be acquired by a pinhole device to avoid motion blur.**
Therefore, cameras rely on **lenses to gather more light** from a scene point and focus it on a single image point. This enables much **smaller exposure times**, as required e.g. to avoid motion blur in dynamic scenes. However, the DOF is no longer infinite, for only points across a limited range of distances can be simultaneously on focus in a given image.

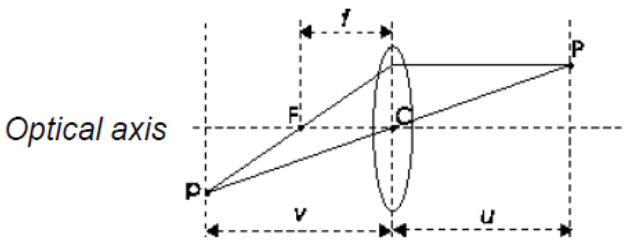
1.5.1 – THIN LENS EQUATION

In the real world, cameras that we know have lenses that are not simply lenses, but are very complex optic systems. It can be difficult to model the reality, so. Knowing that we can use a pinhole camera, we need to model the system geometrically. Luckily, with certain conditions, we can do that.
Often, we can model a system that is very complex using a **THIN LENS**. This lens takes the light that comes from the point on the top right (P) and moves that point on the bottom left (p = image obj).
f in that case is always the focal length but now there's a slightly different concept beside. We can't change independently u and v because they have a relationship: if u increases, v decreases and vice versa.
Every ray that is parallel to the optic axis goes to the focus, while the others pass through C, they goes inside the lens. If the scene points are on the focus, we'll see a good image, but the quality of the image depends on what we want to do, it is not always necessary to have this condition.
The plane of the image needs a sensor and the only place in which we can put it is the center of the lens C: it is exactly the same model of the pinhole camera!
But what we put in the equation if we consider the scale factor? The role of f, as it has been defined in the pinhole camera, is given by the distance of the image, but in this specific case the distance of the image is equal to the effective focal length.

Formally, cameras often feature complex optical systems, comprising multiple lenses. Yet, we will consider here the approximate model known as thin lens equation.

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

P: scene point
p: corresponding focused image point
u: distance from P to the lens
v: distance from p to the lens
f: focal length (parameter of the lens)
C: centre of the lens
F: focal point (or focus) of the lens



To graphically determine the position of a focused image point we can leverage on the following two **PROPERTIES OF THIN LENSES**:

- Rays parallel to the optical axis are deflected to pass through F
- Rays that pass through C are undeflected

It is worth pointing out that if the image is on focus the image formation process obeys to the perspective projection model, with the centre of the lens being the optical centre and the distance v acting as the effective focal length of the projection. This is a different concept than the focal length of the lens.

1.5.2 – CIRCLES OF CONFUSION

Due to the thin lens equation, choosing the distance of the image plane determines the distance at which scene points appear on focus in the image:

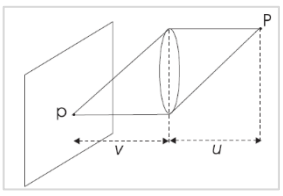
$$\frac{1}{u}+\frac{1}{v}=\frac{1}{f} \quad \rightarrow \quad u=\frac{vf}{v-f}$$

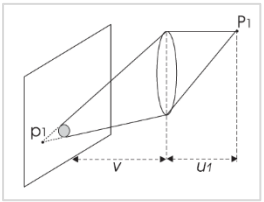
If we choose the image plane distance and we choose v, we'll have a fixed distance. So, u is then calculated on the basis of what we've already fixed. Only a plane in this case is on focus, all the others are put on focus automatically and give us the quality of the image.
Likewise, to acquire scene points at a certain distance we must set the position of the image plane accordingly:

$$\frac{1}{u}+\frac{1}{v}=\frac{1}{f} \quad \rightarrow \quad v=\frac{uf}{u-f}$$

Given the chosen position of the image plane, **scene points both in front and behind the focusing plane will result out-of-focus**, thereby appearing in the image as circles, known as **Circles of Confusion** or Blur Circles, rather than points.

P belongs to the focusing scene plane
P1 lies closer to the lens than P (u1<u)
P2 is farther away to the lens than P (u2>u)

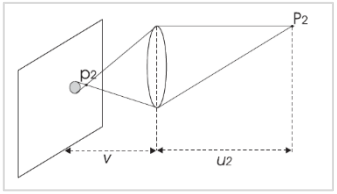




In the **first image** we have the exact match between the real point and the point of the image.

In the **second image** we have that the real point is closer, so v increases. All the light goes behind the plane of the image, so there's a circle: there's no the top of the cone.

In the **third image** u is bigger and the cone is a way reversed.



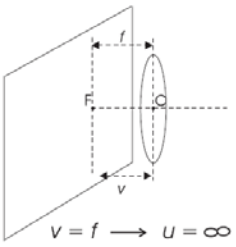
1.5.3 – DIAPHRAGM, DOF AND F-NUMBER

In theory, when imaging a scene through a thin lens only the points at a certain distance can be on focus, all the others appearing blurred into circles.
However, as long as such circles are smaller than the size of the photo-sensing elements, the image will still look on-focus. **The range of distances across which the image appears on focus** - due to blur circles being small enough - **determines the DOF (Depth of Field) of the lens.**

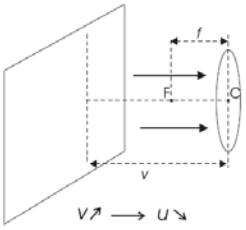
Cameras often deploy an **adjustable diaphragm (iris) to control the amount of light gathered** through the effective aperture of the lens. The closer the diaphragm aperture is, the larger turns out the DOF as a result of the smaller size of blur circles.
If we close the iris, we'll have less light so we'll have more things on focus, but we need to have a scene with much light. If the scene is dark, we need a long exposition time.

The **F-number is the ratio of the focal length to the effective aperture of the lens (f/d)**. F-number discrete units (also known as **stops**) are usually reported on the diaphragm (e.g. 1.4, 2, 2.8, 4, 5.6, 8, 11, 16…) to allow the user to adjust the effective aperture. The higher the chosen stop, the closer is the diaphragm and thus the larger is the actual DOF.

1.5.4 – FOCUSING MECHANISM



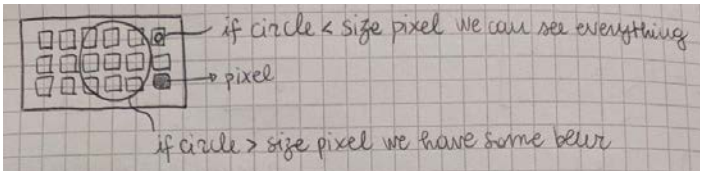
To focus on objects at diverse distances, another mechanism allows the lens (or lens subsystem) to **translate along the optical axis with respect to the - fixed - position of the image plane** (in nowadays cameras, a solid-state sensor mounted on a PCB).
At one end position (v=f) the camera is focused at infinity, then the mechanism allows the lens to be translated farther away from the image plane up to a certain maximum value (the second end position), which determines the minimum focusing distance.

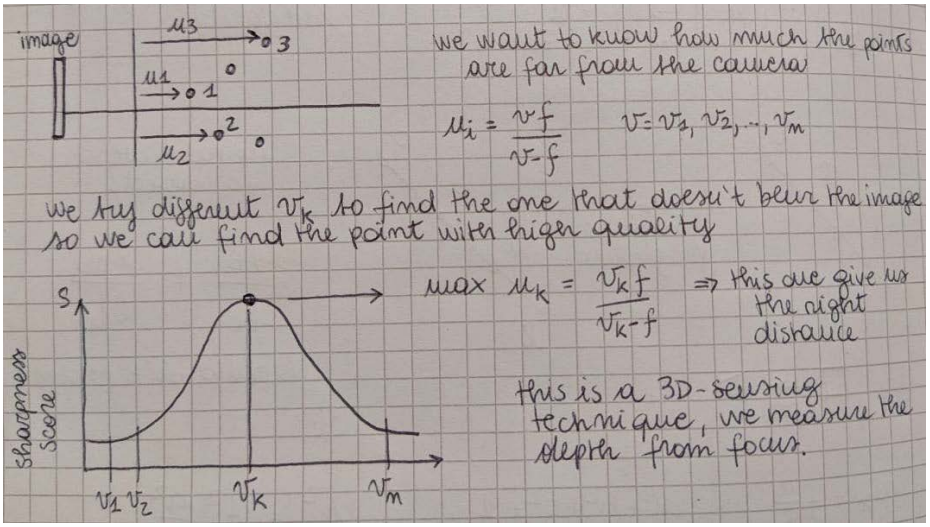


When we focus on something or the lens moves and the PCB is fixed or the lens is fixed and the PCB moves. Typically, is the lens that moves, while the PCB is fixed.
In the first image v=f, so u tends to infinite. So, in the first position, if f is small, we're doing the focus on infinite. When we increase the distance v, u decreases.
Typically, a camera has a focus range from infinite to a certain number of cm, e.g. 20.
There are manual cameras that we can fix, other that have the autofocus (they have a motorized focus). There are two ways to realize an autofocus:

- There's an active autofocus, i.e. a sensor such as the time of light, that calculates the distance at which the lens is in relation to the objects
- There's an algorithm that looks at the image and calculates how much it is on focus. Then it decides, if necessary, to change the position of the lens

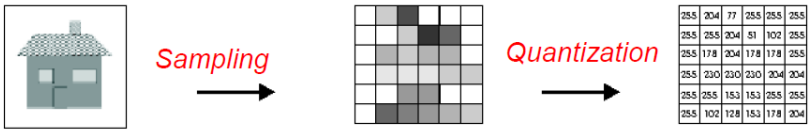
Images with a good focus have high frequencies, so we can also look at the frequencies of the images to know if the lens is sufficiently close or far.





1.6 – IMAGE DIGITIZATION

Generally speaking, **the image plane of a camera consists of a planar sensor which converts the Irradiance at any point into an electric quantity** (e.g. a voltage).
Afterwards, such a continuous "electric" image is sampled and quantized to end up with a digital image suitable to visualization and processing by a computer.



Continuous Image *Sampled Image* *Sampled and Quantized Image*

SAMPLING – **The planar continuous image is sampled evenly along both the horizontal and vertical directions** to pick up a 2D array (matrix) of N×M samples known as **pixels**:

$$I(x,y) \Longrightarrow \begin{bmatrix} I(0,0) & I(0,1) & \dots & I(0,M-1) \\ \vdots & & & \\ \vdots & & & \\ I(N-1,0) & I(N-1,1) & \dots & I(N-1,M-1) \end{bmatrix}$$

QUANTIZATION – **The continuous range of values associated with pixels is quantized into $l=2^m$ discrete levels** known as **gray-levels**. Thus, m is the number of bits used to represent a pixels, with the memory occupancy (in bits) of a gray-scale image given by: $B = N \times M \times m$

Usually, m=8 in gray-scale digital images, where m=bits per entry, so that, e.g. a VGA format (480×640) image requires 300 Kbytes for storage while a 1mpx image requires 1 Mbytes.
Colour digital images are instead typically represented within computers using 3 bytes per pixels (one byte for each of the RGB channels). For each coloured pixel we have a triplet and typically every pixel is 24 bytes.

1.6.1 – IMAGE QUALITY

512×512, l=256
(original)

64×64, l=256
(coarser sampling)

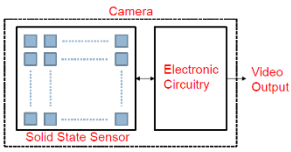
The more bits we spend for its representation, the higher the quality of the digital image (we get a closer approximation to the ideal continuous image). This applies to both sampling as well as quantization parameters.

In the second image of Lenna, we see that there's a loss of details and jagged contours due to aliasing. If we use instead a coarser quantization like in the third image, we'll have a loss of smooth gray-scale transitions: false contouring or posterization.

512×512, l=16
(coarser quantization)

1.6.2 – CCD and CMOS

The sensor is a 2D array of photodetectors (photogates or photodiodes). During exposure time, each detector converts the incident light into a proportional electric charge (i.e. photons to electrons). Then, the companion circuitry reads-out the charge to generate the output signal, which can be either digital or analog. In the former case, the camera includes also the necessary ADC circuitry, while nowadays the latter type of cameras is mostly manufactured for the sake of legacy systems.

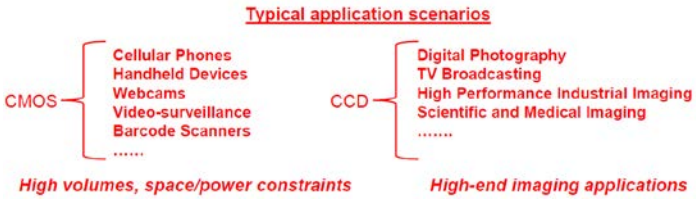


Hence, **there is never a continuous image in practice**, for the image is sensed directly as a sampled signal. In analog cameras though, the native sampling taking place at the sensor is lost in the generation of the analog output, which is then sampled and quantized by a dedicated circuitry within the computer known as analog frame grabber. As a result, **the pixels in a digital image coming from an analog camera do not correspond to those sensed by the photodetectors.**

Today, **the two main sensor technologies are:**

- **CCD** (Charge Coupled Devices)
- **CMOS** (Complementary Metal Oxide Semiconductor).

Unlike CCD, **CMOS technology allows the electronic circuitry to be integrated within the same chip as the sensor** ("one chip camera"). This provides more compactness, less power consumption and often lower system cost.
Furthermore, unlike CCD, **CMOS sensors allow an arbitrary window to be read-out without having to receive the full image**. This can be useful to inspect or track at a higher speed a small **Region Of Interest (ROI)** within the image.
CCD technology typically provides higher SNR, higher DR and better uniformity.



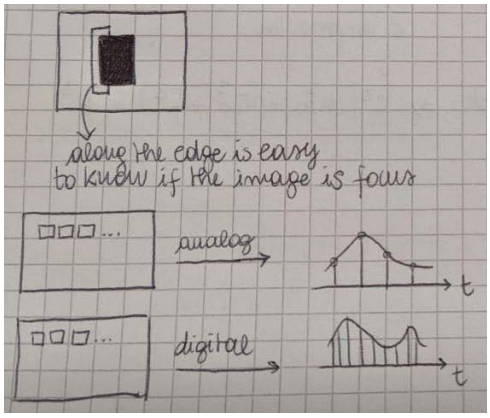
CMOS: we have one chip camera so the image sensor can be realized in a single silicon piece. It is used also for general purpose chips. This is not the case of CCD: we cannot have a single silicon die, but we have a couple of elements.

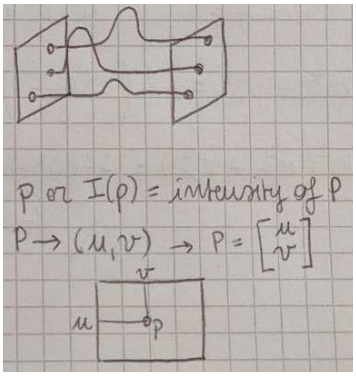
In CMOS we can read a window, in CCD we have to download a full frame every time, we have to read everything. Moreover, in CMOS we can scan pixel in a window and it is interesting because it's faster, e.g. when we want to predict where the object will be in the next frame.

CCD has higher quality: more uniformity, so the space is that of high-quality images like digital photography or scientific imaging, while CMOS is more compact, so it's used for cellular phones, webcams, barcode scanners, etc.

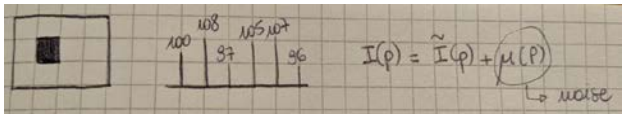
1.6.3 – CAMERA PARAMETERS

PRO single camera → no correspondance problem
CON single camera → images has to be static





Experiment: we have a point and we find its image on the plane. Then, we take that value, i.e. 100, and we put it on the time line calling it t0. We take another image that is exactly the same, with the same light and the same distance. The image won't be 100, but 105. If we take another, it will be 97. We have an oscillation of values: with perfect static conditions those values change continuously due to the noise. When we read a pixel, we don't have a deterministic value, but for each value we add a random value that is due to the noise.



Main camera parameters:

1. SIGNAL-TO-NOISE RATIO (SNR)

The **intensity** measured at a pixel under perfectly static conditions varies due to the presence of random noise (i.e. a pixel value is not deterministic but rather a random variable). **Higher is the SNR, higher is the quality**. The main noise sources are:

- **Photon Shot Noise** – The time between photon arrivals at a pixel is governed by a Poisson statistic and thus the number of photons collected during exposure time is not constant.
- **Electronic Circuitry Noise** – It is generated by the electronics which reads-out the charge and amplifies the resulting voltage signal.
- **Quantization Noise** – related to the final ADC conversion (in digital cameras).
- **Dark Current Noise** – a random amount of charge due to thermal excitement is observed at each pixel even though the sensor is not exposed to light.

The SNR can be thought of as quantifying the strength of the "true" signal with respect to the unwanted fluctuations induced by noise (i.e. the higher the better). It should be measured according to standard procedures and it is usually **expressed either in decibels or bits**.

$$\mathrm{SNR}_{\mathrm{dB}} = 20\log(\mathrm{SNR}); \qquad \mathrm{SNR}_{\mathrm{bit}} = \ln(\mathrm{SNR})$$

2. DYNAMIC RANGE (DR)

If the sensed amount of light is too small, the "true" signal cannot be distinguished from noise: let's call $\mathbf{E_{min}}$ **the minimum detectable irradiation**.
On the other hand, the charge stored at each pixel cannot exceed a certain quantity: let's call $\mathbf{E_{max}}$ **the saturation irradiation**, i.e. the amount of light that would fill up the capacity of a photodetector. The DR of a sensor is specified in dB or bits and is defined as:

$$DR = \frac{Emax}{Emin}$$

To observe a quantity of significant changes we need to use the sensitivity of the pixel. The minimum and maximum quantity of light characteristic of a pixel is like a capacity, it has a min and a max.
e.g. we have a container that is full of rain. There's a min quantity of water that tells us that is raining and there's a maximum because if the water is too much, it goes out of the container. The bucket is the sum of our pixels: the ratio between the max and the min is the DR.

As it is the case of the SNR, **<u>also for the DR the higher is the better</u>**. Indeed, the higher the DR the better is the ability of the sensor to simultaneously capture in one image both the dark and bright structures of the scene. An active research field in image processing deals with creating **High Dynamic Range (HDR) images** by combining together a sequence of images of the same subject matter taken with different exposure times.
This HDR is necessary because, e.g. in the image, there's a point that is saturated of light and we cannot see the details: that's because the majority of the photo is dark. If we take a long exposition, the light piece would become even more light! So, DR is not sufficient in that case.

3. SENSITIVITY (RESPONSIVITY)

It deals with the amount of signal that the sensor can deliver per unit of input optical energy.

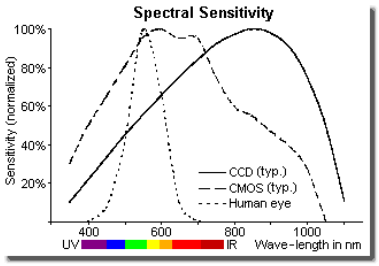
4. UNIFORMITY (SPATIAL OR PATTERN NOISE)

Due to manufacturing tolerances both the response to light and the amount of dark noise vary across pixels.

1.6.4 – COLOUR SENSORS

CCD/CMOS sensors are sensitive to light ranging from near-ultraviolet (200 nm) through the visible spectrum (380-780 nm) up to the near infrared (1100 nm). The sensed intensity at a pixel results from integration over the range of wavelengths of the spectral distribution of the incoming light multiplied by the spectral response function of the sensor. As such **<u>CCD/CMOS sensor cannot sense colour</u>**.

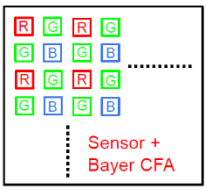
To create a colour sensor, an array of optical filters (**Colour Filter A rray**) is placed in front of the photodetectors, so as to render each pixel sensitive to a specific range of wavelengths.
In the most common **Bayer CFA green filters** are twice as much as red and blue ones to mimic the higher sensitivity of the human eye in the green range.
To obtain an RGB triplet at each pixel missing samples are interpolated from neighbouring pixels (**demosaicking**). However, the true resolution of the sensor is smaller due to the green channel being subsampled by a factor of 2, the blue and red ones by 4.



A "full resolution" – though more expensive - **colour sensor** can be achieved by deploying an optical prism to split the incoming light beam into 3 RGB beams sent to 3 distinct sensors equipped with corresponding filters.

In the image we have the electromagnetic spectrum and we have the spectral sensitivity. When the light comes into a pixel, the device converts the electronic radiation so they are mono-chromatic.
A CCD do not sense colors, a CMOS neither. So how is it possible to sense color? The sensors only gather all the light. We have a triplet of numbers: the red, blue and green channel.

We don't have sensors which recognise these colors, so we use a color filter array (CFA): we apply a small color filter which is sensitive only in a specific range of wavelengths.

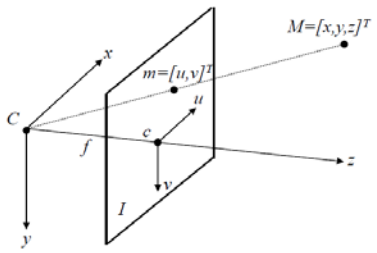


With this trick we have created green pixel, red pixels and blue pixels and we can have a color image.
There's a problem: these pixels have a primary color channel plus the two additional channels.
We don't have full resolution for every pixel, but in some applications, where we need very high-quality images, we should have 3 colour channels, so 3 sensors: we'll have a prism.

2 - CAMERA CALIBRATION

Let us consider a point in the 3D space $M=[x,y,z]^T$ with coordinates given in the Camera Reference Frame (CRF). Its projection onto the image plane I is denoted as $m=[u,v]^T$.
The non-linear equations providing image coordinates as a function of the 3D coordinates in the CRF are as follows:

$$u = \frac{f}{z}\,x \qquad v = \frac{f}{z}\,y$$



2.1 – PROJECTIVE SPACE

The physical space is a **3D Euclidean Space (R^3)** whose points can be represented as 3D vectors in a given reference frame. In this space **parallel lines do not intersect**, or intersect "at infinity" and **points at infinity cannot be represented** in this vector space.
Let's now append one more coordinate to our Euclidean triples, so that e.g. (x y z) becomes (x y z 1) and assume that both vectors are correct representations of the same 3D point.
Moreover, we do not constrain the 4th coordinate to be 1 but instead assume:

$$(x\ y\ z\ 1) = (2x\ 2y\ 2z\ 2) = (kX\ kY\ kZ\ k)\ \forall k \neq 0$$

In this representation a **point** in space is represented by an **equivalence class of quadruples**, wherein **equivalent quadruples differ just by a multiplicative factor**.
This is the so-called **HOMOGENEOUS COORDINATES** (a.k.a. projective coordinates) representation of the 3D point having Euclidean coordinates (x, y, z). The space associated with the homogeneous coordinate representation is called **Projective Space**, denoted as P^3.
Extension to Euclidean spaces of any other dimension is straightforward ($R^n \rightarrow P^n$).

Remember that we can use a pinhole model also if the camera is not a pinhole camera. We have the center of projection, the image plane then, the perspective projection model establishes a mapping between world coordinates and image coordinates. u and v are the image coordinates, particularly u is the horizontal one and v is the vertical one.
The Camera Reference Frame is not the reference frame in which we measure pixel coordinates, it is not a 2D reference frame. It is the 3D reference frame in which we apply the perspective projection equations to measure the image points.
We can compute the horizontal and vertical image coordinates just by scaling the corresponding world coordinates x and y based on the depth. The larger is the depth, the smaller are u and v.
When we watch at camera calibration, the equations we saw have properties that we dislike: they are non-linear. With camera calibration we can determine camera parameters.
To move from non-linear to linear equations we need to consider a different formalization based on defining coordinates (world and image) into projective rather than Euclidean spaces.

So, we move from 3D image into a new representation with a 1 as another coordinate. If we multiply this per 2, we get the same point. So kx, ky, kz, k is the same point.
This new representation is called "Projective Representation" of our point. We are now in P3 but we have 4 coordinates that are homogeneous.

So, we can multiply with non-zero k to lift a representation and this lifting can be done with every dimensional space.

M=M0+λ[a, b, c] M=[x, y, z] M0=[x0, y0, z0]

That's the parametric equation of the line. Doing some simple calculation:

M = [xo+λa, yo+λb, zo+λc]

Now we can add 1 and multiply for a non-zero k.

M = [xo+λa, yo+λb, zo+λc, 1]

Now we multiply for 1/λ and we obtain:

M = [(xo/λ) + a, (yo/λ) + b, (zo/λ) + c, 1/ λ]

Now we are representing whatever point on the line. We can take the limit of λ to infinity:

M = [a, b, c, 0]

When last coordinate is 0, it's a point at infinity.

If I want to find a point to infinity in a line (in the projective space), we take the direction cosine vector or another parallel vector and append 0.

2.1.1 – POINTS AT INFINITY

More formally, the points of the 3D Projective Space having the fourth coordinate equal to 0, e.g. (x,y,z,0), cannot be represented in the 3D Euclidean Space. Indeed, to map such points into the Projective Space we would divide by the null fourth coordinate, so as to get (x/0, y/0, z/0), i.e. infinite coordinates, which is not a valid representation in the Euclidean Space.

By the homogenous coordinates it is therefore possible to represent such points at infinity, which do not exist in the Euclidean Space, as those projective points having the last coordinate equal to 0.

Point (0, 0, 0, 0) is undefined. Indeed, this point is NOT the origin of the Euclidean Space (0, 0, 0) because such point is represented in homogeneous coordinates as (0, 0, 0, k), with k≠0.

It can be shown that all points at infinity of P3 lie on a plane, which is called the **PLANE AT INFINITY**.

Example

Let's consider the linear equations to get the intersection between two lines:

$$\begin{cases} ax + by + c = 0 \\ a'x + b'y + c' = 0 \end{cases}$$

These are two lines and we want to solve the intersection. They are in 2D space and they cannot be parallel because if they're parallel there's no solution since it's the point at infinity that we can't represent.

We can determinate the solution below and it exists only if the lines are not parallel.

The solution is given by (Cramer's rule):

$$x = \frac{\begin{vmatrix} -c & b \\ -c' & b' \end{vmatrix}}{\begin{vmatrix} a & b \\ a' & b' \end{vmatrix}} = \frac{\alpha}{\beta} \qquad y = \frac{\begin{vmatrix} a & -c \\ a' & -c' \end{vmatrix}}{\begin{vmatrix} a & b \\ a' & b' \end{vmatrix}} = \frac{\gamma}{\beta}$$

Should the lines be parallel, the denominator would vanish and hence the solution does not exist (i.e. it is a point at infinity, which does not exist in the 2D Euclidean Space R^2).

As shown above, we can represent the intersection point in homogeneous coordinates, so as to express the point in the projective space P2.

$$\underset{R^2}{[x, y]} \equiv \underset{P^2}{[\alpha, \gamma, \beta]}$$

In the projective space the above representation is always a valid one, regardless of the third coordinate being either different from 0 or not. Hence there always exists the solution of the system, i.e. the intersection point.
It can be shown easily that the point at infinity of a set of parallel lines can be represented in the Projective Space by appending a null coordinate to any Euclidean vector directed as the lines:

$$\begin{bmatrix} bc'-b'c \\ a'c-ac' \\ 0 \end{bmatrix} \approx \begin{bmatrix} -b \\ a \\ 0 \end{bmatrix} \approx \begin{bmatrix} b \\ -a \\ 0 \end{bmatrix} \approx \begin{bmatrix} -b' \\ a' \\ 0 \end{bmatrix} \approx \begin{bmatrix} b' \\ -a' \\ 0 \end{bmatrix}$$

To summarize, any Euclidean Space R^n can be extended to a corresponding Projective Space P^n by representing points in homogeneous coordinates.
The projective representation features one additional coordinate, referred to here as k, wrt the Euclidean representation:

- k ≠ 0 denotes point existing in Rn, their coordinates given by xi/k, i = 1…n
- k = 0 denotes points at infinity (a.k.a. ideal points) in Rn, which do not admit a representation via Euclidean coordinates

The Projective Space allows then to represent and process homogeneously (i.e. without introduction of exceptions or special cases) both the ordinary and the ideal points of the Euclidean Space.
We are interested in Projective Spaces because Perspective Projection is more conveniently dealt with using projective coordinates.

2.1.2 – PERSPECTIVE PROJECTION IN PROJECTIVE COORDINATES

We already know that there exists a non-linear transformation between 3D coordinates and image coordinates:

$$u = \frac{f}{z}\, x \qquad v = \frac{f}{z}\, y$$

Let's now go back to our 3D point M (with coordinates expressed in the CRF) and its projection onto the image plane, m:

$$M = [x, y, z]^T \qquad m = [u, v]^T$$

and represent both points in homogenous coordinates (from now on always denoted by symbol ~) as projective points:

$$\tilde{m} = [u, v, 1] \qquad \tilde{M} = [x, y, z, 1]$$

In homogeneous coordinates (hence considering the mapping between projective spaces) the perspective projection becomes a **linear transformation**:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f\frac{x}{z} \\ f\frac{y}{z} \\ 1 \end{bmatrix} = \begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

In matrix notation: $k\tilde{m}=\tilde{P}\tilde{M}$ often expressed also as $\tilde{m} \approx \tilde{P}\tilde{M}$, where ≈ means "equal up to an arbitrary scale factor".

2.1.3 – PESPECTIVE PROJECTION MATRIX (PPM)

Matrix $\tilde{P}$ represents the geometric camera model, and is known as Perspective Projection Matrix (PPM). If we assume distances to be measured in focal length units (f = 1), the PPM becomes:

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

This form is useful to understand the core operation carried out by the perspective projection, which is indeed scaling lateral coordinates (i.e. x, y) according to the distance from the camera (z). The actual focal length just introduces an additional, fixed (i.e. independent of z) scaling factor of projected coordinates. The above form is usually referred to as **canonical or standard PPM**.

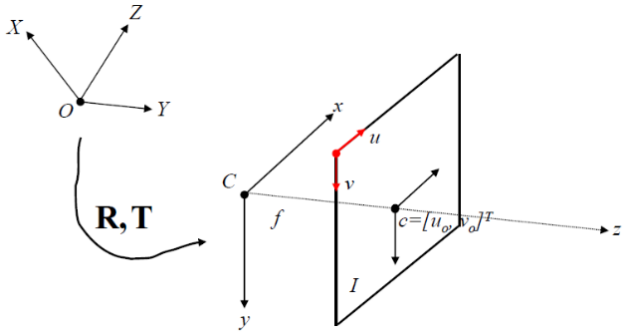
The vanishing point of a line it's the image of the point at infinity. To use this equation to find the vanishing point we have to project the point at the infinity and we can do it because it's an ordinary point.

$\tilde{m}$ = [f 0 0 0
0 f 0 0 → $\tilde{m}$ = [fa, fb, c] → $\tilde{M} = [a, b, c, 0]$
0 0 1 0] * $\tilde{M}$

In R2 we have: [fa/c, fb/c] = m infinity
Our perspective projection matrix is $\tilde{P}$ = [I|0], so we can do $\tilde{P}$ * [0, 0, 1, 0] which is $\tilde{M}$ and we have: $\tilde{m}$ = $\tilde{P}$ * $\tilde{M}$ = [0, 0, 1] → in R2 we have [0,0] and this is the vanishing point.

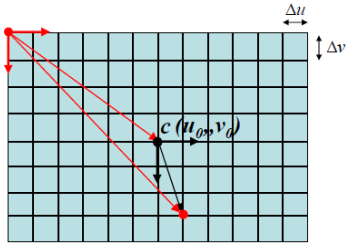
To come up with a really useful camera model we need to take into account two additional issues:

- Image Digitization
- The 6 DOF (3D rotation and translation) rigid motion between the Camera Reference Frame (CRF) and the World Reference Frame (WRF)



2.2 - IMAGE DIGITIZATION

Digitization can be accounted for by including into the projection equations the scaling factors along the two axis due to the quantization associated with the horizontal and vertical pixel size.
Moreover, we need to model the translation of the piercing point (intersection between the optical axis and the image plane) wrt the origin of the image coordinate system (top-left corner of the image).



Δu= horizontal pixel size
Δv= vertical pixel size

$$u = \frac{f}{z}x \qquad \rightarrow u = \frac{1}{\Delta u}\frac{f}{z}x = k_u\frac{f}{z}x + u_0$$

$$v = \frac{f}{z}y \qquad \rightarrow v = \frac{1}{\Delta v}\frac{f}{z}y = k_v\frac{f}{z}y + v_0$$

2.2.1 – INTRINSIC PARAMETER MATRIX

Based on the equations in the previous slide, the PPM can be written as

$$\tilde{\mathbf{P}} = \begin{bmatrix} fk_u & 0 & u_0 & 0 \\ 0 & fk_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} fk_u & 0 & u_0 \\ 0 & fk_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \mathbf{A}[\mathbf{I}\,|\,\mathbf{0}]$$

(αu → fk_u; αb → fk_v)

Matrix A, which models the characteristics of the image sensing device, is called **Intrinsic Parameter Matrix**. Intrinsic parameters can be reduced in number by setting $\alpha u = fk_u$ and $\alpha v = fk_v$ such quantities representing:

- αu → focal length expressed in horizontal pixel sizes
- αv → focal length expressed in vertical pixel sizes

The **smallest number of intrinsic parameters** is thus 4.
A more general model would include a 5th parameter, known as **SKEW**, to account for possible non orthogonality between the axis of the image sensor. The skew would be A[1,2], but it is usually 0 (=ctg(π/2)) in practice.

If we want to calibrate a camera and we want to know the parameters, what we shall do?
Fundamental relation with stereo: z = bf/d which are all in mm.
I have double cameras, so uL-uR=dp and the disparity is in pixels, not in mm.
We have to multiply the horizontal pixel size → d=dp*δu

$$z = \frac{b * \alpha u * \delta u}{dp * \delta u} = \frac{b * \alpha u}{dp} \qquad with \quad z[mm] \quad dp[pixels]$$

2.3 – RIGID MOTION BETWEEN CRF AND WRF

So far, we have assumed 3D coordinates to be measured into the CRF, though this is hardly feasible in practice. More generally, **3D coordinates are measured into a World Reference Frame (WRF) external to the camera**. The WRF will be related to the CRF by:

- A **rotation around the optical centre** (e.g. expressed by a 3x3 rotation matrix R)
- A **translation** (expressed by a 3x1 translation vector T)

Therefore, the relation between the coordinates of a point in the two RFs is:

$$W = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad M = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \rightarrow \quad M = RW + T$$

Which can be rewritten in homogeneous coordinates as follows:

$$\begin{cases} \widetilde{W} = [X, Y, Z, 1] \\ \widetilde{M} = [x, y, z, 1] \end{cases} \quad \rightarrow \quad \widetilde{M} = \begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix} \widetilde{W} = G\widetilde{W}$$

We have also seen how to map a 3D point expressed in the CRF:

$$k\widetilde{m} = A[I|0]\widetilde{M}$$

We need now to consider also the rigid motion between the WRF and the CRF:

$$\widetilde{M} = G\widetilde{W} \quad \rightarrow \quad k\widetilde{m} = A[I|0]G\widetilde{W} \quad \rightarrow \quad k\widetilde{m} = A[I|0]\begin{bmatrix} R & T \\ 0 & 1 \end{bmatrix}\widetilde{W}$$

Accordingly, the general form of the PPM can be expressed as follows:

$$\tilde{P} = A[I|0]G \qquad \text{or also} \qquad \tilde{P} = A[R|T]$$

2.3.1 – EXTRINSIC PARAMETERS

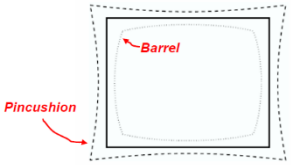
Matrix G, which encodes the position and orientation of the camera with respect to the WRF, is called **Extrinsic Parameter Matrix**.
As a rotation matrix (3x3=9 entries) has indeed only 3 independent parameters (DOF), which correspond to the **rotation angles around the axis of the RF**, the **total number of extrinsic parameters is 6** (3 translation parameters, 3 rotation parameters).
Hence, the general form of the PPM can be thought of as encoding the position of the camera wrt the world into G, the perspective projection carried out by a pinhole camera into the canonical PPM [I|0] and, finally, the actual characteristics of the sensing device into A.

2.4 - LENS DISTORTION

However, to explain observed images we often need to model also the effects due to the optical distortion induced by lenses, which indeed **renders the pure pinhole not accurate enough a model in many applications**. Lens distortion is modelled through additional parameters that do not alter the form of the PPM.

As we can see, we have straight lines in the world that are not straight in the camera because they're curved. What can we say about the problem projection? No matters what kind of model we have, we have a problem related to lens distortion and not to the projection.
We can create so an image undistorted starting from the distorted image.

The PPM is based on the pinhole camera model. However, real lenses introduce distortions wrt to the pure pinhole model, this being true especially for cheap and/or short focal length lenses.



The most significant deviation from the ideal pinhole model is known as radial distortion (lens "curvature"). Second order effects are induced by **tangential distortion** ("misalignement" of optical components and/or defects).
Lens distortion is modelled through a non-linear transformation which maps ideal (i.e. undistorted) image coordinates (blue) into the observed (i.e. distorted) image coordinates (red).

$$(x', y') = L(r)\,(\tilde{x}, \tilde{y}) + (d\tilde{x}, d\tilde{y})$$

Depending on the distance r from the distortion centre $(\tilde{x}_c, \tilde{y}_c)$.

$$r = \sqrt{(\tilde{x} - \tilde{x}_c)^2 + (\tilde{y} - \tilde{y}_c)^2}$$

2.4.1 - LENS DISTORTION PARAMETERS

The **radial distortion function L(r)** is defined for positive r only and such as L(0) = 1. This non-linear function is typically approximated by its Taylor series (up to a certain approximation order):

$$L(r) = 1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \dots$$

The tangential distortion vector is instead approximated as follows:

$$\begin{pmatrix} d\tilde{x} \\ d\tilde{y} \end{pmatrix} = \begin{pmatrix} 2p_1\tilde{x}\tilde{y} + p_2(r^2 + 2\tilde{x}^2) \\ p_1(r^2 + 2\tilde{y}^2) + 2p_2\tilde{x}\tilde{y} \end{pmatrix}$$

The radial distortion coefficients k1, k2, …, kn, together with the distortion centre $(\tilde{x}_c, \tilde{y}_c)$ and the two tangential distortion coefficients p1 and p2 form the set of the lens distortion parameters, which extends the set of parameters required to build a useful and realistic camera model.
Typically, for the sake of simplicity, **the distortion centre is taken to coincide with the image centre** (i.e. the piercing point).

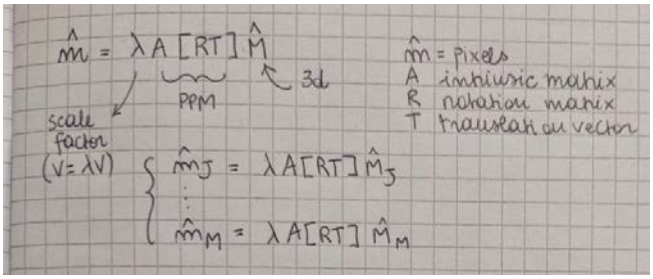
Lens distortion is modelled as a non-linear mapping taking place after canonical perspective projection onto the image plane. Afterwards, the intrinsic parameter matrix allows applying an affine transformation which maps image coordinates into pixel coordinates.
Accordingly, the image formation flow can be summarized as follows:

- Transformation of 3D points from the WRF to the CRF, according to extrinsic parameters:
$$M=RW+T$$
- Canonical perspective projection (i.e. scaling by the third coordinate):
$$\tilde{x} = \frac{x}{z} \qquad \tilde{y} = \frac{y}{z}$$
- Non-linear mapping due to lens distortion:
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = L(r) \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} + \begin{pmatrix} d\tilde{x} \\ d\tilde{y} \end{pmatrix}$$
- Mapping from image coordinates to pixels coordinates according to the intrinsic parameters:
$$m = A(x', y')^T$$

2.5 – CALIBRATION

We have now at disposal a camera model thoroughly explaining the image formation and digitization process. Such a model is the **PPM**, which in turn can be **decomposed into 3 independent tokens**:

- **Intrinsic parameter matrix (A)**
- **Rotation matrix (R)**
- **Translation vector (T)**

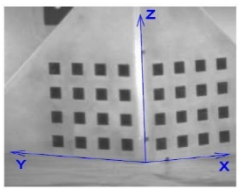


Camera calibration is the process whereby all parameters defining the camera model are - as accurately as possible - estimated for a specific camera device. Depending on the application, either the PMM only or also its independent components (A, R, T) need to be estimated.
Many camera calibration algorithms do exist. **The basic process, though, relies always on setting up a linear system of equations given a set of known 3D-2D correspondences**, so as to then solve for the unknown camera parameters.
To obtain the required correspondences, specific physical objects (referred to as calibration targets) having easily detectable features (such as e.g. chessboard or dot patterns) are typically deployed.

We can start from points in the 3D reference system then, accordingly to the extrinsic parameters, we can move the points in the camera reference system and then, accordingly to the intrinsic parameters, we transform them.
We have seen the lens distortion model: when we are calibrating the camera, we don't know the parameters. If we know other elements, we can find the PPM, but in general **we need to know world coordinates and pixel coordinates.**

We use calibration targets thanks to which it is very easy to measure world coordinates and image coordinates. We're talking about features for specific object in order to find the correspondences.

Camera calibration approaches can be split into two main categories:

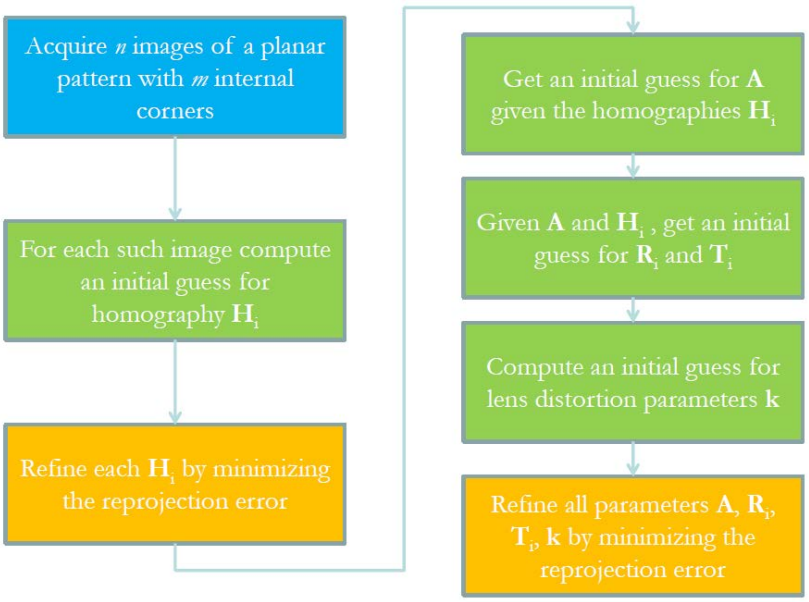


- Those relying on a **single image featuring several (at least 2) planes** containing a known pattern (e.g. dotted paper)
- Those relying on **several (at least 3) different images of one given planar pattern** (e.g. chessboard)

In practice, it is difficult to build accurate targets containing multiple planes, while an accurate planar target can be attained rather easily.
Implementing a camera calibration software requires a significant effort. However, the main Computer Vision toolboxes include specific functions, for example OpenCV, Matlab CC Toolbox, Halcon, etc.

2.5.1 – ZHANG'S METHOD

Zhang was a Microsoft researcher. In his method we have many steps and we have to understand the whole process. Generally, we acquire n images of a planar target, typically a "chessboard" pattern in which n is more or less 15/20.



Given a planar chessboard pattern, **known** are:

- The **number of internal corners of the pattern**, different along the two orthogonal directions for the sake of disambiguation (i.e. rows, columns). **Internal corners can be detected easily by standard algorithms** (e.g. the Harris corner detector, possibly with sub-pixel refinement for improved accuracy)
- The **size of the squares** which form the pattern

In each image, the **3D WRF is taken at the top-left corner of the pattern**, with plane z=0 given by the pattern itself and the x, y axis aligned to the two orthogonal directions, in particular so as to keep always the same association between axis and directions (e.g. x=rows, y = columns).
Having z=0 simplifies the equation a lot. In the next step we setup a linear equation and then we optimize these parameters by a non-linear optimalization, so we have an alternative algorithm but we have to start from a good initial guess.

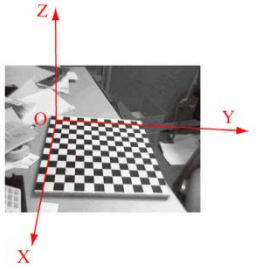
Thus, as for 3D points:

- The **third coordinate is always 0**
- **x and y are determined by the known size of the squares** forming the chessboard

To get camera parameters, **two main steps** are carried out:

- **Initial guess by a linear optimization** → minimization of an algebraic error
- **Refinement by a non-linear minimization** → minimization of a geometric error

ESTRINSIC PARAMETERS



It is worth pointing out that **each image requires its own estimate of the extrinsic parameters**, as they are different from one to the other.
A global WRF can be taken to coincide with that associated with one of the images, e.g. the first.

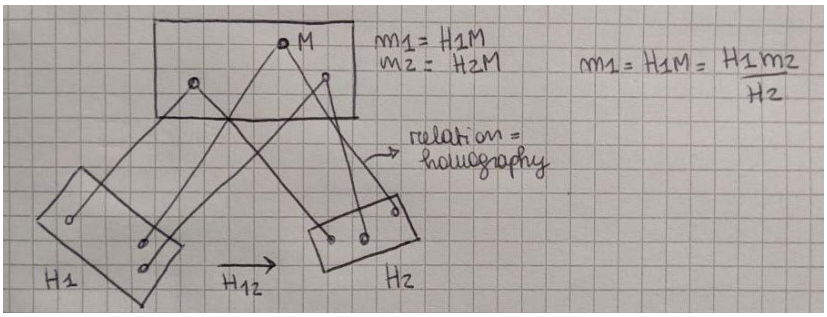
The number of squares is different in the two direction so the pattern contains an intrinsic asymmetry → we can easily find the two axis. Remember that they all have the third coordinate z = 0.
These are World coordinates, so we are in the WRS. There will be a different WRS for each image, so the extrinsic parameters will be as many as the number of calibration images. We can always chain them together because we have also the camera RS.

P AS HOMOGRAPHY

Due to the choice of the WRF associated with calibration images, in each of them we consider only 3D points with z=0. Accordingly, **the PPM boils down to a simpler transformation defined by a 3x3 matrix**:

$$k\tilde{\mathbf{m}} = \tilde{\mathbf{P}}\tilde{\mathbf{w}} = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} & p_{1,4} \\ p_{2,1} & p_{2,2} & p_{2,3} & p_{2,4} \\ p_{3,1} & p_{3,2} & p_{3,3} & p_{3,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,4} \\ p_{2,1} & p_{2,2} & p_{2,4} \\ p_{3,1} & p_{3,2} & p_{3,4} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbf{H}\tilde{\mathbf{w}}'$$

Such a transformation, denote here as **H**, is known as homography and **represents a general linear transformation between planes**. Above, w' represents vector (x, y, 1). H can be thought of as a simplification of P in case the imaged object is planar.
Given a pattern with m corner, we can write m systems of 3 linear equations as above, wherein both 3D as well as 2D coordinates are known due to corners having been detected in the ith image and the unknowns are thus the 9 elements in Hi. However, as Hi, and Pi alike, is known up to an arbitrary scale factor, **the independent elements in Hi are indeed just 8**.

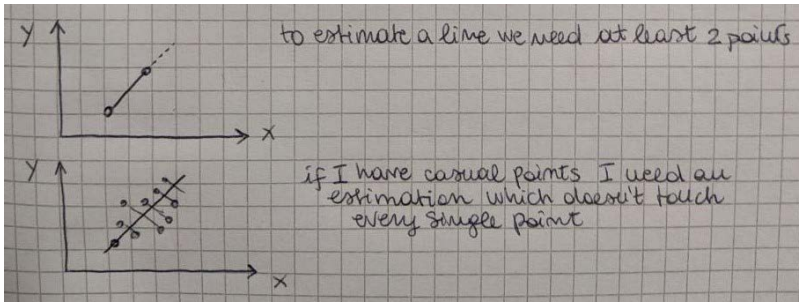


P=PPM=A[RT]
If the coordinates of the 3D control points are planar coordinates (they lie in a plane), it means that z=0. So, the third column of the PPM is =0 and we could rewrite this equation in a simpler form obtaining a representation with only 2D coordinates.
Estimating homography is quite common in CV because, when we are looking at a plane, the mapping between the world and the image becomes an homography.

So, whenever we look at a plane, we have this homography and also if we look at two images of the same world plane, the relation between these two images is still an homography.

4 correspondences are the minimum required to get an homography, like the corners of the chessboard. The homography has 8 degrees of freedom.
Can we solve for the homography using more equations? It is worth do it or is better to use the minimum number? Should we solve with more equation another constraint system so? It is possible but there's no solution which satisfies truly every equation.

In the second one we don't have a line who touches every point, but we can have noise, we can have noisy measurement.



So, it is useful to have all the measurements because we can have a better estimation.
The second line doesn't fit exactly every point but it's kind a good estimation.
When we have real world problems, we have ALWAYS noise, so at least our features will be very much noisy.
Now we use all the control points, all the corners, to set up the system of equations to find the unknown homography. We need to estimate as many homographs as the calibration images.

ESTIMATING Hi

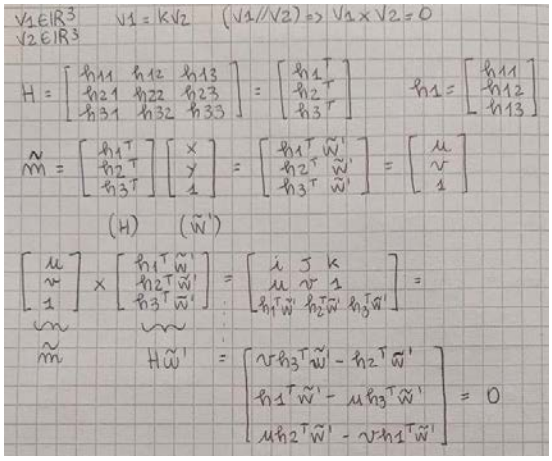
Starting from the previous homography equation we can write:

$$k\tilde{\mathbf{m}} = \mathbf{H}\tilde{\mathbf{w}}' \quad \Rightarrow \quad \tilde{\mathbf{m}} \times \mathbf{H}\tilde{\mathbf{w}}' = \mathbf{0} \quad \Rightarrow \quad \begin{bmatrix} v\mathbf{h}_3^{\mathrm{T}}\tilde{\mathbf{w}}' - \mathbf{h}_2^{\mathrm{T}}\tilde{\mathbf{w}}' \\ \mathbf{h}_1^{\mathrm{T}}\tilde{\mathbf{w}}' - u\mathbf{h}_3^{\mathrm{T}}\tilde{\mathbf{w}}' \\ u\mathbf{h}_2^{\mathrm{T}}\tilde{\mathbf{w}}' - v\mathbf{h}_1^{\mathrm{T}}\tilde{\mathbf{w}}' \end{bmatrix} = \mathbf{0}, \mathbf{H} = \begin{bmatrix} \mathbf{h}_1^{\mathrm{T}} \\ \mathbf{h}_2^{\mathrm{T}} \\ \mathbf{h}_3^{\mathrm{T}} \end{bmatrix}$$

i.e.

$$\begin{bmatrix} \mathbf{0}^{\mathrm{T}} & -\tilde{\mathbf{w}}'^{\mathrm{T}} & v\tilde{\mathbf{w}}'^{\mathrm{T}} \\ \tilde{\mathbf{w}}'^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} & -u\tilde{\mathbf{w}}'^{\mathrm{T}} \\ -v\tilde{\mathbf{w}}'^{\mathrm{T}} & u\tilde{\mathbf{w}}'^{\mathrm{T}} & \mathbf{0}^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} \mathbf{h}_1 \\ \mathbf{h}_2 \\ \mathbf{h}_3 \end{bmatrix} = \mathbf{A}\mathbf{h} = \mathbf{0}$$

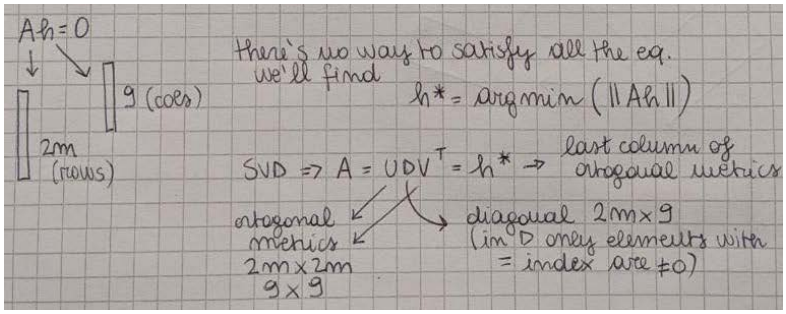
Between the previous 3 equations in 9 unknowns, only 2 are linearly independent, so that typically only the first 2 of them are kept. By deploying all correspondences, **for each image we can build a linear system of 2m equations in 9 unknowns.** This allows for achieving an initial estimation of Hi by minimization of the "algebraic error" represented by the norm of vector **Ah** and enforcing the additional constraint **||h||=1** (DLT Algorithm).
It is well known that the solution of the above estimation problem can be obtained by the Singular Value decomposition **(SVD) of matrix A**.



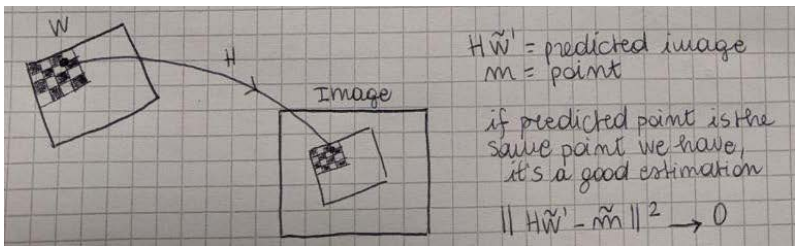
DLT algorithm (first implication): we have a vector containing the coordinates of a control point. We have two 3x1 vectors (km e Hw) so this is an equation because they're the same.

If I forget this relation and I consider m and Hw, we have a Euclidean vector, so they will be vectors in R^3. m and Hw are parallel vectors so their product is zero.

After the second implication → we have 3 equations and they're =0. This is just a way to rewrite the previous equation in another way. It seems that for each correspondent image we have 3 equations but if we look at these 3 equations, they're not linear independent. We have in reality 2 equations for each control point in 9 elements.
Considering all these systems, e.g. with 20 control points, we obtain 40 equations, we have over-constraints homogeneous linear systems.



Smallest Right Singular Vector
So, we minimize ||Ah|| and this is the norm of vector Ah.
Can we give it a geometrical meaning? This quantity is called **ALGEBRAIC ERROR**, it's a mathematical entity useful to formalize the system. To really find the good homography we should minimize what?



Given the previous initial estimation, Hi is later refined by the least-squares solution of the non-linear minimization problem:

$$\min_{H_i} \sum_j \left|\left|\tilde{m}_j - H_i \tilde{w}'_j\right|\right|^2 \qquad j = 1, \ldots, m$$

which can be obtained in practice using by the **LEVENBERG-MARQUARDT ALGORITHM**.

If we don't have an initial solution, we can solve it iteratively:

1) base on the DLT algorithm
2) plug that in non-linear optimization problem
3) minimize a physical error

To reach the perfect solution we need an initial good guess. Geometric or reprojection error is referred to the non-linear solution. There's many way to solve this non-linear: we can use Newton iteration, the first order gradient… in CV we use Levenberg-Marquardt.

This additional optimization steps corresponds to the **minimization of the reprojection error** measured for each of the 3D corners (with z=0) of the pattern by comparing the pixel coordinates predicted by the estimated homography to the pixel coordinates of the corresponding corner extracted in the image. The rationale is that **the "best" homography would predict with the best accuracy the positions of the corner features actually found in the image**. Such a reprojection error is typically referred to as "**GEOMETRIC ERROR**".

DLT ALGORITHM

By considering 4-point pairs, we obtain a system of 8 equations in 9 unknowns:

$$Ah = 0, \qquad A = [A_1, A_2, \dots, A_9], \qquad h = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = \begin{bmatrix} h_1 \\ \vdots \\ h_9 \end{bmatrix}$$

where A is a 8x9 matrix, A1…A9 are 8x1 vectors, h is a 9x1 vector, h1,h2,h3 are 3x1 vectors representing the rows of the 3x3 matrix H defining the homography, h1...h9 are the 9 elements of such matrix.

As **H is defined up to a scale factor**, we can set h9=1, so as to obtain a non-homogeneous linear system with 8 equations and 8 unknowns:

$$\tilde{A}\tilde{h} = b, \qquad \tilde{A} = [A_1, A_2, \dots, A_8], \qquad \tilde{h} = \begin{bmatrix} h_1 \\ \vdots \\ h_8 \end{bmatrix}, \qquad b = -A_9$$

which can be solved easily by standard methods (Cramer's rule, Inversion of the coefficient matrix, Gaussian Elimination).

Alternatively, it is possible to constrain the nom of h, e.g. so as to render it equal to 1. Purposely, we can assume again h9 as fixed, but no longer equal to one:

$$\tilde{\mathbf{A}}\tilde{\mathbf{h}} = \mathbf{b}, \quad \tilde{\mathbf{A}} = [\mathbf{A}_1\ \mathbf{A}_2 \dots \mathbf{A}_8], \quad \tilde{\mathbf{h}} = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_8 \end{bmatrix}, \quad \mathbf{b} = -h_9\mathbf{A}_9$$

$$\tilde{\mathbf{h}} = -h_9\tilde{\mathbf{A}}^{-1}\mathbf{A}_9 \rightarrow \mathbf{h} = \begin{bmatrix} -h_9\tilde{\mathbf{A}}^{-1}\mathbf{A}_9 \\ h_9 \end{bmatrix} = h_9\begin{bmatrix} -\tilde{\mathbf{A}}^{-1}\mathbf{A}_9 \\ 1 \end{bmatrix}$$

$$\|\mathbf{h}\| = 1 \rightarrow h_9\sqrt{\left\|\tilde{\mathbf{A}}^{-1}\mathbf{A}_9\right\|^2 + 1} = 1 \rightarrow h_9 = \frac{1}{\sqrt{\left\|\tilde{\mathbf{A}}^{-1}\mathbf{A}_9\right\|^2 + 1}}$$

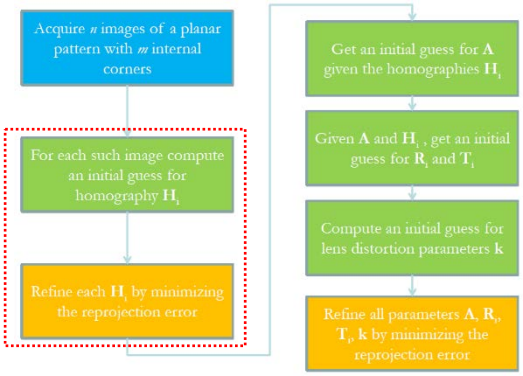
According to this method, the coefficient matrix of the linear system needs is inverted and then, using the formulas shown above it is possible to compute h9 first and then h.

ESTIMATION OF THE INSTRINSIC PARAMETERS

As Hi is known up to a scale factor, we can establish the following relation between Hi and the PPM:

$$H = [h_1 \quad h_2 \quad h_3] = [p_1 \quad p_2 \quad p_4] = \lambda A[r_1 \quad r_2 \quad T]$$

h1, h2, and h3 are the column of the 3x3 Homography. (in the previous notation h1, h2 and h3 were the rows)
p1 = 1st column, p2 = 2nd column, p4 = 4th column of the P metric.



$$p = [p_1 \quad p_2 \quad p_3 \quad p_4] = \lambda A[RT] = \lambda A[r_1 \quad r_2 \quad r_3 \quad T]$$

$$p_1 = \lambda A r_1 \rightarrow p_1 = h_1 \rightarrow r_1 = A^{-1}h_1/\lambda$$
$$p_2 = \lambda A r_2 \rightarrow p_2 = h_2 \rightarrow r_2 = A^{-1}h_2/\lambda$$
$$p_4 = \lambda A T \rightarrow p_4 = h_3$$

r1 and r2 are orthogonal, so $r_1^T * r_2 = 0$. We can write:

$$\frac{1}{\lambda^2}\{(A^{-1}h_1)^T * A^{-1}h_2\} = 0 \rightarrow (A^{-1}h_1)^T \Rightarrow \boxed{h_1^T A^{-T} A^{-1} h_2 = 0}$$ with h_1, h_2 unknown

Since there is the property that $||r1|| = ||r2|| = 1$ → $||r1||^2 = ||r2||^2 = 1$,

$$\rightarrow \frac{1}{\lambda}[r_1^T r_1] = \frac{1}{\lambda}[r_2^T r_2] \rightarrow \boxed{(h_1^T A^{-T})(A^{-1}h_1) = (h_2^T A^{-T})(A^{-1}h_2)}$$

Formally, **R is an orthogonal matrix**, its vectors being orthonormal. This constrains the intrinsic parameters to obey to the following relations:

$$r_1^T r_2 = 0 \Rightarrow \boxed{h_1^T A^{-T} A^{-1} h_2 = 0}$$
$$r_1^T r_1 = r_2^T r_2 \Rightarrow \boxed{h_1^T A^{-T} A^{-1} h_1 = h_2^T A^{-T} A^{-1} h_2}$$

where the unknowns are the entries of $B=A^{-T}A^{-1}$. As A is upper triangular, B turns out to be symmetric, so that the unknowns are just 6.
By stacking together the above two equations provided by each calibration image, we obtain a 2n x 6 linear system, which can be solved in case at least 3 images are available.

We have two equations with the unknown parameters:

- A is Upper triangular so it is non zero in the elements above the diagonal
- B is symmetric so we have 6 unknowns

We have 2 equations in 6 unknows per each Homography. We need at least 3 H to find a solution, but with more is better so we can use the least square.

ESTIMATION OF THE EXTRINSIC PARAMETERS

By posing:

$$\mathbf{B} = \mathbf{A}^{-\mathrm{T}}\mathbf{A}^{-1} = \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{pmatrix}, \quad b = [B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33}]^T$$

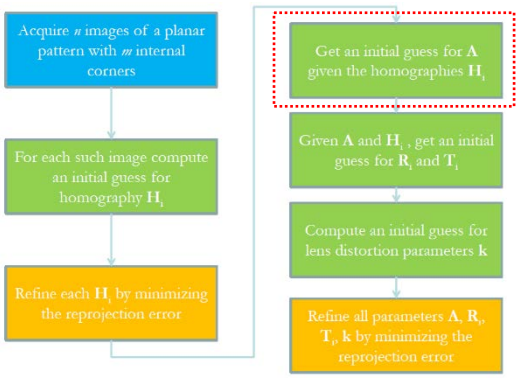
$$\mathbf{h}_i^T = [h_{i1}, h_{i2}, h_{i3}], \qquad \mathbf{v}_{ij}^T = [h_{i1}h_{j1},\ h_{i1}h_{j2} + h_{i2}h_{j1},\ h_{i2}h_{j2},\ h_{i3}h_{j1} + h_{i1}h_{j3},\ h_{i3}h_{j2} + h_{i2}h_{j3},\ h_{i3}h_{j3}]$$

It can be noticed that:

$$\mathbf{h}_i^{\mathrm{T}}\mathbf{B}\,\mathbf{h}_j = \mathbf{v}_{ij}^{\mathrm{T}}\mathbf{b} \longrightarrow \begin{array}{l} \mathbf{h}_1^{\mathrm{T}}\mathbf{B}\mathbf{h}_2 = 0 \Rightarrow \mathbf{v}_{12}^{\mathrm{T}}\mathbf{b} = 0 \\ \mathbf{h}_1^{\mathrm{T}}\mathbf{B}\mathbf{h}_1 = \mathbf{h}_2^{\mathrm{T}}\mathbf{B}\mathbf{h}_2 \Rightarrow \mathbf{v}_{11}^{\mathrm{T}}\mathbf{b} = \mathbf{v}_{22}^{\mathrm{T}}\mathbf{b} \Rightarrow (\mathbf{v}_{11} - \mathbf{v}_{22})^{\mathrm{T}}\,\mathbf{b} = 0 \end{array}$$

Therefore, each image provides 2 equations in the 6 independent unknowns in B, so that with con n calibration images we get a homogeneous linear system of equations in the form Vb = 0, which can be solved in a least squares sense.
Once b has been calculated, the intrinsic parameters (i.e. A) can be obtained in closed form.
This system is over constraint linear homogeneous and we solve it by the SVD: the solution is the last columns of the 6x6 metric, is the same solution as previous.

We estimate as many homographies as the images, we put all together. If we have now all the homographies and we have the intrinsic, we can estimate the extrinsic parameters.



Given A, and having obtained Hi for each image, it is possible to compute Ri first, and then Ti, for each image i:

$$\mathbf{H}_i = \begin{bmatrix} \mathbf{h}_{1,i} & \mathbf{h}_{2,i} & \mathbf{h}_{3,i} \end{bmatrix} = \lambda \mathbf{A} \begin{bmatrix} \mathbf{r}_{1,i} & \mathbf{r}_{2,i} & \mathbf{T}_i \end{bmatrix}$$

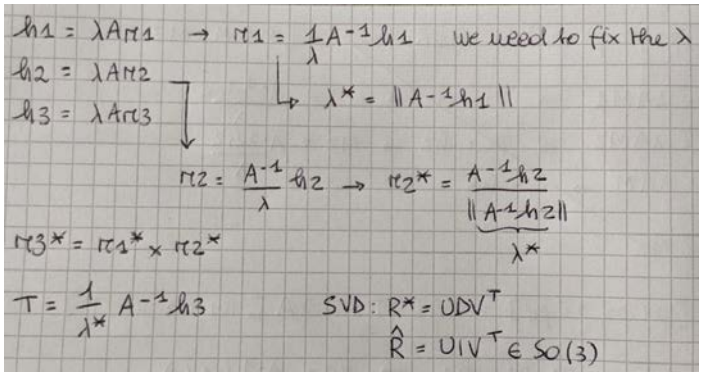
$$\mathbf{h}_{k,i} = \lambda \mathbf{A} \mathbf{r}_{k,i} \quad \Rightarrow \quad \lambda \mathbf{r}_{k,i} = \mathbf{A}^{-1} \mathbf{h}_{k,i}, \; k = 1,2$$

As fk,j is a unit vector:

$$\mathbf{r}_{k,i} = \frac{1}{\lambda} \mathbf{A}^{-1} \mathbf{h}_{k,i}, \; \lambda = \left\| \mathbf{A}^{-1} \mathbf{h}_{k,i} \right\|, \quad k = 1,2$$

R3 can be derived from r1 and r2 by exploiting orthonormality: $r3,i = r1,i \times r2,i$

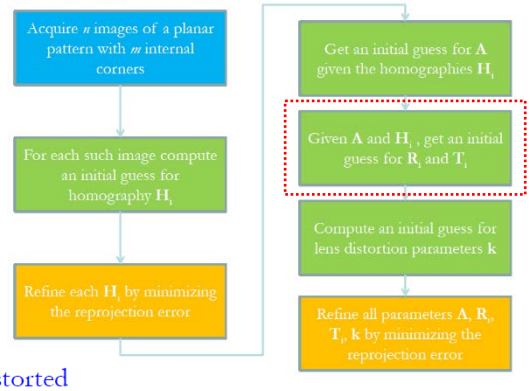
Finally, we can get Ti: $Ti = \frac{1}{\lambda} A^{-1} h_{3,i}$



LENS DISTORTION COEFFICIENTS

Given the homographies, we have both the real (distorted) coordinates of the corner features found in the images as well as the ideal (undistorted) coordinates predicted by the homographies. Zhang's method deploys such information to estimate coefficients k1, k2 of the radial distortion function.

Given the already known intrinsic parameter matrix, A, and the lens distortion model reported below:



$$\text{distorted} \begin{pmatrix} x' \\ y' \end{pmatrix} = L(r) \overbrace{\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix}}^{\text{undistorted}} = \left(1 + k_1 r^2 + k_2 r^4\right) \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix}$$

We need first to establish the relationship between distorted $(\boldsymbol{u}', \boldsymbol{v}')$ and **ideal** $(\tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}})$ **pixel coordinates**:

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = A \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{pmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \rightarrow \begin{cases} x' = \dfrac{u' - u_0}{\alpha_u} \\ y' = \dfrac{v' - v_0}{\alpha_v} \end{cases} \qquad \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ 1 \end{bmatrix} = A \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ 1 \end{bmatrix} \rightarrow \begin{cases} \tilde{x} = \dfrac{\tilde{u} - u_0}{\alpha_u} \\ \tilde{y} = \dfrac{\tilde{v} - v_0}{\alpha_v} \end{cases}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \left(1 + k_1 r^2 + k_2 r^4\right) \begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix} \rightarrow \begin{cases} \dfrac{u' - u_0}{\alpha_u} = \left(1 + k_1 r^2 + k_2 r^4\right) \left(\dfrac{\tilde{u} - u_0}{\alpha_u}\right) \\ \dfrac{v' - v_0}{\alpha_v} = \left(1 + k_1 r^2 + k_2 r^4\right) \left(\dfrac{\tilde{v} - v_0}{\alpha_v}\right) \end{cases} \longrightarrow \begin{cases} u' = \tilde{u} + \left(k_1 r^2 + k_2 r^4\right)\left(\tilde{u} - u_0\right) \\ v' = \tilde{v} + \left(k_1 r^2 + k_2 r^4\right)\left(\tilde{v} - v_0\right) \end{cases}$$

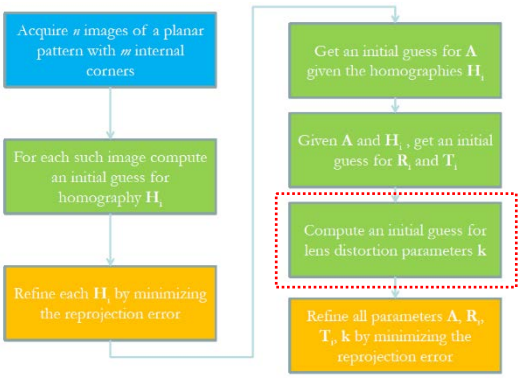
It is therefore possible to set up a linear system where the unknowns are the distortion coefficients.

$$\begin{bmatrix} (\tilde{u}-u_0)r^2 & (\tilde{u}-u_0)r^4 \\ (\tilde{v}-v_0)r^2 & (\tilde{v}-v_0)r^4 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} = \begin{bmatrix} u'-\tilde{u} \\ v'-\tilde{v} \end{bmatrix}$$

$$r^2 = \left(\frac{\tilde{u}-u_0}{\alpha_u}\right)^2 + \left(\frac{\tilde{v}-v_0}{\alpha_v}\right)^2$$

With m corner features in n images, we get a linear system with 2mn equations in 2 unknowns, which admits a least-squares solution:

$$\mathbf{Dk} = \mathbf{d} \rightarrow \mathbf{k} = \mathbf{D}^{+}\mathbf{d} = \left(\mathbf{D}^{\mathrm{T}}\mathbf{D}\right)^{-1}\mathbf{D}^{\mathrm{T}}\mathbf{d}$$



REFINEMENT BY NON-LINEAR OPTIMIZATION

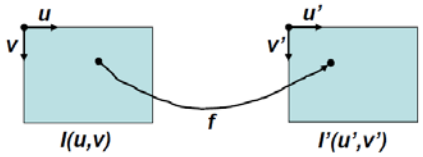
Again, the procedure highlighted so far seeks to minimize an algebraic error, without any real physical meaning.
A more accurate solution can instead be found by a so-called **Maximum Likelihood Estimate (MLE)** aimed at minimization of a geometric (i.e. reprojection) error.
Under the hypothesis of **i.i.d. (independent identically distributed)** noise, the MLE for our models is obtained by minimization of the error:

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\left\|\mathbf{m}_{ij} - \hat{\mathbf{m}}\left(\mathbf{A},\mathbf{k},\mathbf{R}_i,\mathbf{T}_i,\mathbf{w}_j\right)\right\|^2$$

with respect to all the unknown camera parameters. $\hat{m}$ are the predicted corners and w are the world point coordinates. The solution of the above non-linear optimization problem is again provided by the **Levenberg-Marquardt algorithm**.

2.5.2 - IMAGE WARPING

This is a very simple operation: image warping is about **transforming pixel coordinates from a source image to a target image.** It's a mapping between pixel coordinates.



$$\begin{cases} u' = f_u(u,v) \\ v' = f_v(u,v) \end{cases}$$

$$I'\left(f_u(u,v), f_v(u,v)\right) = I(u,v)$$

Examples of image warping:

Rotation → $\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$

Removal of perspective deformation → $s\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{32} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$

Other examples:
- *Correct lens distortion*
- *Stereo rectification*
- .....

This image warping is defined by two functions: u' and v'.
Given the coordinates from u and v, it provides us the u and v coordinates in the target image, where U = horizontal, V = vertical.

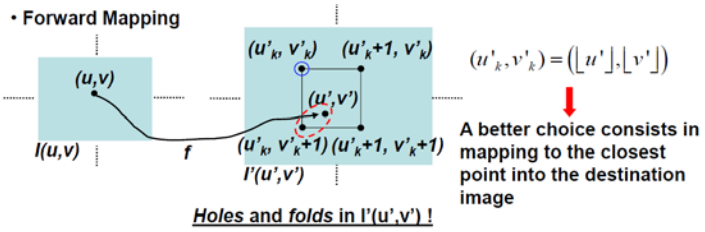
So, we have **intensity transfer**: we pick the intensity of a pixel and we remap this into another image. We can do many types of image transformation, e.g. if we want to do a **rotation**. Another type is **removal of perspective distortion**: many times, we have seen planes which were distorted.
When we see that parallel line are no longer parallel, maybe we have to consider this distortion, for example when we have a picture showing the license plate of a car and we want to read the plate.

OCR: Optical Character Recognition. From image to a string of text. OCR system are used to read strings and dates on product, e.g. on food product. They're wide spread in industrial application.

If we take the plate in this way, an OCR doesn't work well. We have to normalize it, so we need to find the corners of the plate and to compute the homography of the plane into a frontal parallel frame, we cancel the distortion.
To estimate the homography we need correspondences, but how many? The **minimum number is 4**, but if we have more than 4 it's better. Assume that we have 4 points, we can estimate the homography.
This is a **warping** because the transformation is the homography and we estimate the pixel of the left image on the right image. There's other type of warping like correct lens distortion: we transfer the pixel from the position they are into the targe position such that the new pixel wouldn't show the distortion.

FORWARD/BACKWORD MAPPING

In the real world we'll have float values. So, we don't have the right exact points, but something that is very close.



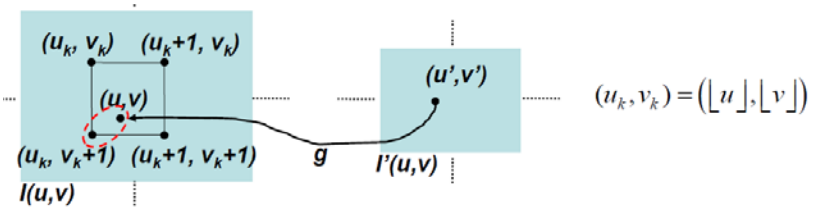
We can truncate pixel coordinates and keep only the integer part: it could be an option. Unfortunately, this is not very good because this image we reach is very bad due to the correct image that has holes and folds.

• **Backward Mapping**

$$\begin{cases} u = g_u(u',v') \\ v = g_v(u',v') \end{cases} \Rightarrow \forall (u',v'):\ I'(u',v') = I\left(g_u(u',v'), g_v(u',v')\right)$$

Not all the target pixels have an assigned value, so we have some pixels who are total black because they don't have a pixel assigned in the source image. In order to avoid these holes and folds, we have to scan the target image and then we have to map in **backward** the source image and every pixel will have an assigned pixel in the other image.
We define the mapping as a function from target to source and we always use the backword mapping. However, we need a strategy to choose the correct pixels.

MAPPING STRATEGIES



There's something better we could do: if the backward mapping is exactly here, ok it makes sense, but if there's something in the middle? We can **interpolate between 4 closest points**.
There are many kinds of interpolation, in the example we have bilinear interpolation which is a compromise: is not very fast but hasn't an high computation.

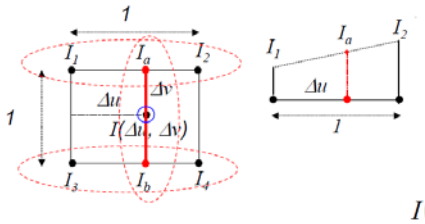
$$\Delta u = u - u_k \qquad I_1 = I(u_k, v_k) \qquad I_2 = I(u_k+1, v_k)$$
$$\Delta v = v - v_k \qquad I_3 = I(u_k, v_k+1) \qquad I_4 = I(u_k+1, v_k+1)$$

Δu = horizontal fractional part, Δv = vertical fractional part.

The most well-known strategies are:

- **Mapping from the closest point (Nearest Neighbour Mapping)**
- **Interpolation between the 4 closest points (bilinear, bicubic, …)**

BILINEAR INTERPOLATION



$$\frac{I_a - I_1}{\Delta u} = I_2 - I_1$$

$$I_a = (I_2 - I_1)\Delta u + I_1$$

$$I_b = (I_4 - I_3)\Delta u + I_3$$

$$I(\Delta u, \Delta v) = (I_b - I_a)\Delta v + I_a$$

$$I(\Delta u, \Delta v) = ((I_4 - I_3)\Delta u + I_3 - ((I_2 - I_1)\Delta u + I_1))\Delta v + (I_2 - I_1)\Delta u + I_1$$

$$I(\Delta u, \Delta v) = (I_2 - I_1)\Delta u + (I_3 - I_1)\Delta v + (I_4 - I_3 - I_2 + I_1)\Delta u \Delta v + I_1$$

$$\boxed{I(\Delta u, \Delta v) = a\Delta u + b\Delta v + c\Delta u \Delta v + d}$$

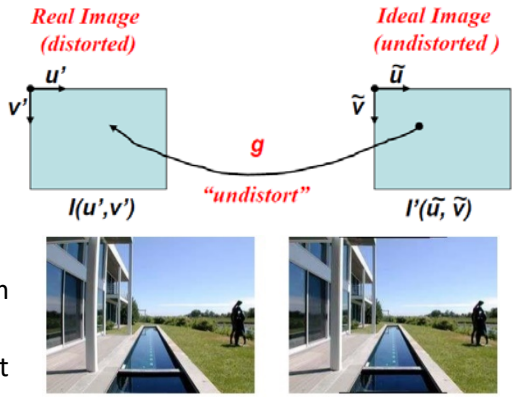
Bilinear = twice linear so we have a linear horizontal interpolation and a linear vertical interpolation. We take Ia between I1 and I2, we take Ib between I3 and I4. We'll interpolate vertically between Ia and Ib.
Considering first I1 and I2, the slope will be I2-I1 over 1, while considering Ia and I1, we have Δu.
We could re-arrange the terms and get the same formula in which we have the contribution of the new intensity:

$$\boxed{I'(u', v') = (1 - \Delta u)(1 - \Delta v)I_1 + \Delta u(1 - \Delta v)I_2 + (1 - \Delta u)\Delta v I_3 + \Delta u \Delta v I_4}$$

When Δu and Δv are =0, we have exactly I1. To have I4, we need Δu and Δv = 1.
When Δv and Δu are 0.5, they are at the center.

e.g. Warping to compensate lens distortion

We said that we can use warping to remove lens distortion. We use those parameters to undistort the image and it will be better.
It's important to do this if we want correct measurements and a correct projection.
Performing the warping Forward means that we go from the distorted image to the undistorted. Backward is from target to source, so from undistorted to distorted.
We input to the model undistorted coordinated and we get distorted, so we apply the Backward Warping.



Once the lens distortion parameters have been computed by camera calibration, the image can be corrected by a backward warp from the undistorted to the distorted image based on the adopted lens distortion model:

$$\forall(\tilde{u}, \tilde{v}): \quad I'(\tilde{u}, \tilde{v}) = I\left(g_u(\tilde{u}, \tilde{v}), g_v(\tilde{u}, \tilde{v})\right)$$

For example, using Zhang's calibration method:

$$\begin{cases} u' = \tilde{u} + \left(k_1 r^2 + k_2 r^4\right)(\tilde{u} - u_0) \\ v' = \tilde{v} + \left(k_1 r^2 + k_2 r^4\right)(\tilde{v} - v_0) \end{cases}$$

Remember that we find k1 and k2 from the camera calibration.

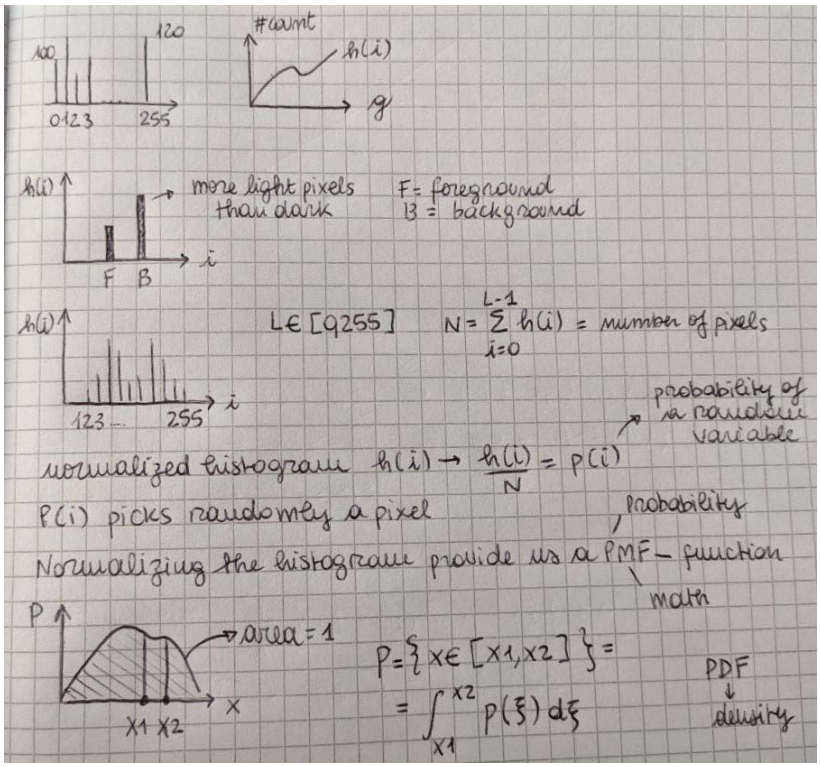
3 - INTENSITY TRANSFORMATIONS

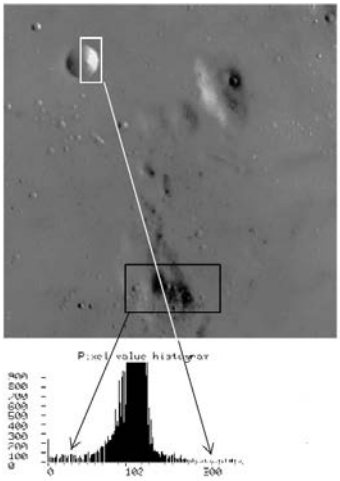
3.1 – GRAY-LEVEL HISTOGRAM

Intensity Transformations (aka Point Operators) are **image processing operators aimed at enhancing the quality** (e.g. the contrast) of the input image.
As most such operators rely on the computation of the **GRAY-LEVEL HISTOGRAM** (intensity histogram) of the input image, we start by defining this useful and widespread function. The gray-level histogram of an image is simply a **function associating to each gray-level the number of pixels in the image** taking that level.

```
Int Histogram[256]:
……
for (i=0; i<N; i++)
        (j=0; i<M; j++)
              Histogram[Image[i][j]]++;
```

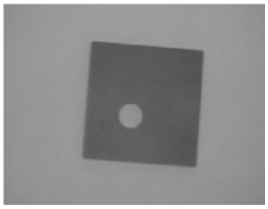
Computing the histogram is straightforward: we just **define a vector as large as the number of gray-levels** and then scan the image to increment the entry of the vector corresponding to the gray-level of the current pixel:

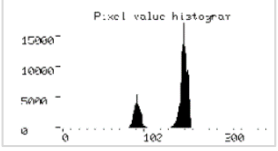




The histogram provides often useful information on image content, although it must be highlighted that **it does not encode any information related to the spatial distribution of intensities**, so that, e.g., if we shuffle randomly the pixels of an image, we end up with a new image having exactly the same histogram as before.

Normalization of histogram entries by the total number of pixels yield relative frequencies of occurrence of gray-levels, which can be interpreted as their probabilities. Accordingly, the normalized histogram can be thought of as the **PMF (probability mass function)** of the discrete random variable given by the gray-level of a randomly picked pixel in the image.





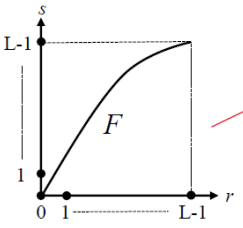
Given the picture, what's the shape of the histogram? This image comes from an industrial machine. By applying this lighting technique, it is very simple for a robot to pick the object up because it can see the borders.

Segmentation: is about classifying the pixel into those belonging to background and those belonging to an object (foreground). Assuming the light is uniform and the material is homogeneous, we'll find in this picture two channels.
There's noise in the image, there's always noise, so the histogram will not be composed by simply 2 values, but there's variation around both the bars.
We can use **segmentation with a threshold**: if the pixel is before the threshold, it'll be white, else it'll be black.

3.2 – INTENSITY TRANSFORMATION or POINT OPERATOR

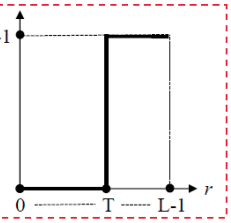
An image processing operator whereby the intensity of a pixel of the output image is computed based on the intensity of the corresponding pixel of the input image only. As such, this operator is just a function which maps a gray-level into a new gray-level:

$s = F(r)$;
r : input gray - level,
s : output gray - level



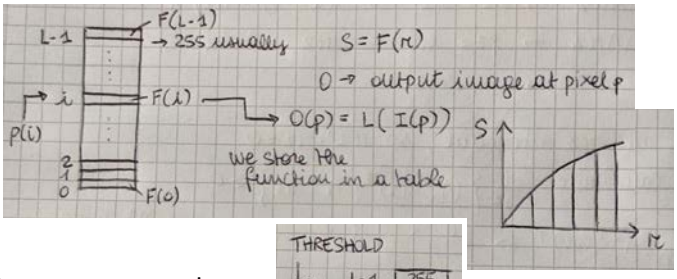
e.g.: Thresholding operator

$$s = \begin{cases} 0 & \text{if } r \le T \\ L-1 & \text{if } r > T \end{cases}$$

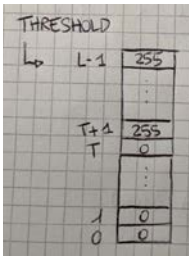


Any such operator can be implemented as a **Look-Up Table (LUT)**, which is often convenient especially in case of hardware implementation.

```
Int LUT[256]:
……
for (i=0; i<N; i++)
        (j=0; i<M; j++)
                OutImage[i][j]=LUT[InImage[i][j]];
```

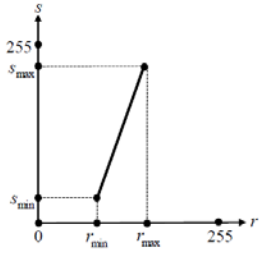


In practice, we precompute the function and then apply it.
Let's consider a table, an array of entries of levels. If we have a stream a pixel acquired from a camera, it's very used this operation.
There are many frame rubbers (board to install in a computer to have certain images, it's used in industrial application) which use it.
We can also use the threshold.



3.2.1 - LINEAR CONSTRAST STRETCHING

An image featuring a small gray-level range has likely poor contrast. The contrast can be enhanced by linearly stretching the intensities to span a larger interval (typically, the full available range).

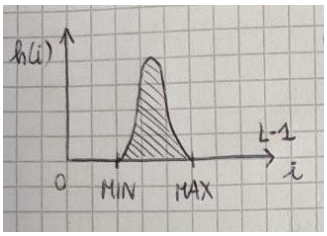


$$s = \frac{s_{max} - s_{min}}{r_{max} - r_{min}}\left(r - r_{min}\right) + s_{min}$$

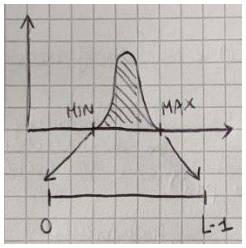
$$s_{min} = 0,\ s_{max} = 255 \rightarrow s = \frac{255}{r_{max} - r_{min}}\left(r - r_{min}\right)$$

Should most pixel lie in a small interval while there exist a few dark and bright outliers, the linear function would approximate the identity and thus be ineffective.
In such a case, rmin and rmax can be taken equal to some percentiles of the distribution (e.g. 5% and 95%), with smaller and higher input intensities mapped to smin and smax respectively.

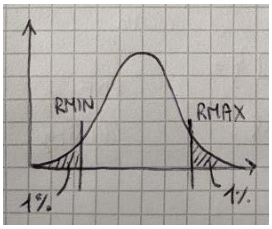
In the example we have a picture with this image. We cannot say if it's the photo of a cat, but we can say something about the quality: since the range is thin, there's no contrast, so the image has a low quality.
If we have an image like the previous one, there's no difference between the object and the background and we cannot perceive the borders of the object.



If an image doesn't use a large range, maybe it's a pure contrast image: we can have this situation and we can expand the gray-level range such that we'll be able to use more gray levels to improve the contrast.



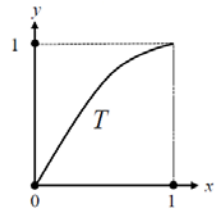
Let's assume that rmin and rmax are the input gray level in our image, so all the pixels lie within this range. To improve the contrast, we expand in an output range between smin and smax and this stretching is done linearly. So that's why linear contrast stretching.
This is a kind of general formulation, but often we use the full range and not a portion of the full range.



There's a small variant of this in which we cannot use this formula, but most of the information are in a smaller interval, so we know that the image is quite bad. We fix some percentile of the distribution and fix two new rmin and rmax, we throw away 1% of the bright pixels and 1% of the dark pixels.

3.2.2 - HISTOGRAM EQUALIZATION

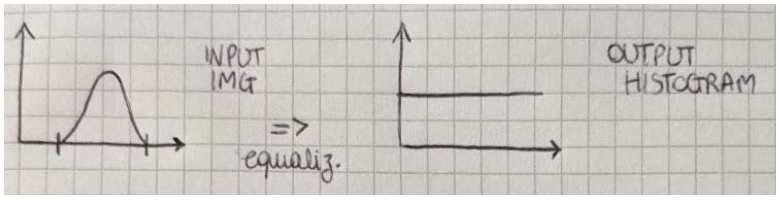
Histogram Equalization (HE) spreads uniformly pixel intensities across the whole available range, which usually improves the contrast. Unlike linear stretching, which may require manual intervention to set rmin and rmax, **HE provides fully automatic contrast enhancement.**



More formally, HE maps the gray-level of the input image so that the histogram of the output image turns out (ideally) flat. To find the mapping, let us consider a **continuous random variable** x, and a strictly monotonically increasing (i.e. invertible) function, T, such that:

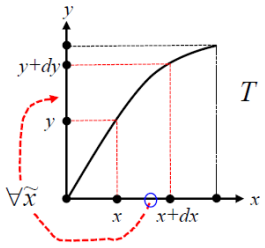
$$x \in [0,1] \rightarrow y = T(x) \in [0,1]$$

This is another stretching contrast method. It's about spreading uniformly all pixel in a range, so we start with ANY histogram, we apply the transformation and we reach a histogram which is flat.
It's useful because it's a fully automatic method and we don't need any parameter.
I pick my image and find the transformation that turns my flat histogram into an ideal histogram.



If we didn't use the entire scale, now we have stretched the range and so the contrast is improved.
We don't get a flat histogram, but trying to reach that goal, we stretch the range and we obtain usage of the full range. We'll try to get a flat histogram, but we'll not obtain it. Pay attention with this thing because he asks it very often during the exam.

Now we're going to do some probability calculations because this method is lied to distribution and probability theory result. We consider a CONTINUOUS random variable (it CAN'T be proved with a discrete variable, so that's the point).

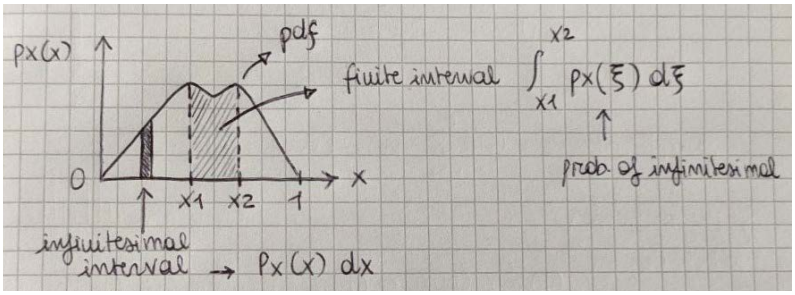


Let us now denote as px(x) and py(y) the **probability density function (pdf)** of x and y respectively. As T is monotonically increasing:

$$\forall \tilde{x} \in [x, x+dx] \rightarrow \tilde{y} = T(\tilde{x}) \in [y, y+dy]$$
$$with \quad y = T(x),\ y+dy = T(x+dx)$$

We can say something about the relationship between these two pdfs: we take an infinitesimal interval of the input random variable. I'm looking to find the best one, but now I consider a generic and I draw it.
If I take any generic X tilde what can I say about the corresponding Y? Because the function is monotonically increasing, a variable in an interval should have Y in the same interval.

What about the probability of X to belong to this interval? The probability is the same the probability Y belong to the other interval. So, they have the same probability to be in the interval.
The probability that x is in that interval is pxdx and for y is pydy, so I can just say that pxdx = pydy.



Therefore, the probability of x and y to belong to their –infinitesimal- intervals is exactly the same, which allows deriving the pdf of y as a function of T and the pdf of x:

$$p_y(y)dy = p_x(x)dx \rightarrow p_y(y) = p_x(x)\frac{dx}{dy}$$

derivative of the inverse function $x = T^{-1}(y)$

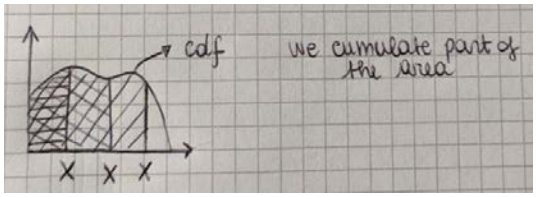
dx/dy is the derivative of the inverse function and it makes sense because it's the reciprocal of the derivative of the function that I'm looking for.
What should be found to do equalization? We want to have py(y) flat, so we want the cumulative distribution function.

Let us now consider a specific mapping function T, i.e. the **cumulative distribution function (cdf)** of x, which is guaranteed to map into [0,1] and be monotonically increasing: It's the integral of the pdf up to x.
Assuming also strict monotonicity we are led to notice that y turns out uniformly distributed in [0,1].

$$y = T(x) = \int_0^x p_x(\xi)\, d\xi \qquad p_y(y) = p_x(x)\frac{dx}{dy} = p_x(x)\frac{1}{dy/dx} = \frac{p_x(x)}{p_x(x)} = 1$$

The derivative of T gives exactly the pdf of x. Any random variable can be turned into an uniform random variable (CONTINUOUS).

We have thus found that by **mapping any continuous random variable through its cdf** (assumed strictly increasing) we attain a uniformly distributed random variable. Next, we shall see how to make use of this theoretical results to equalize an image.



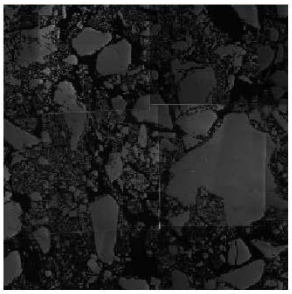
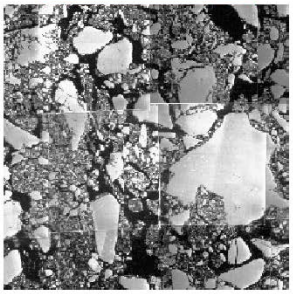
We proceed by discretizing the previous result, i.e. by considering the cumulative probability mass function of the discrete random variable associated with the gray-level of a pixel, whose pmf, as already pointed out, is given by the normalized histogram:

$$\begin{cases} N = \sum_{i=0}^{L-1} h(i) \\ \mathrm{p(i)} = \dfrac{h(i)}{N} \end{cases} \rightarrow j = T(i) = \sum_{k=0}^{j} p(\mathrm{k}) = \frac{1}{N}\sum_{k=0}^{j} h(\mathrm{k}) \rightarrow j = \frac{L-1}{N}\sum_{k=0}^{j} h(\mathrm{k})$$

to map *j* in [0..L-1]

The red square is the input of a normalized image, while T has to be the cumulative of my probability must function (pmf). We discretize a random variable, so we don't have an integral any more.
The above function does not perfectly equalize the histogram due to the several approximations involved (proof in the continuous case, strict monotonicity assumption, rounding errors). Nonetheless, it is usually **effective in spreading the intensities over a wider range** so as to improve image contrast.

Original Images (low-contrast) *Equalized Images*

also some posterization effects due to reduction of gray levels caused by rounding

3.2.3 - HISTOGRAM SPECIFICATION/MATCHING

Starting from an arbitrary input image, we wish to transform it into a new image having a given, desired histogram. Let us denote as: $x, p_x(x)$ $z, p_z(z)$

the random variables associated respectively with the gray-level of a pixel in the input and output image, and let's consider also the random variables attainable by equalization of both:

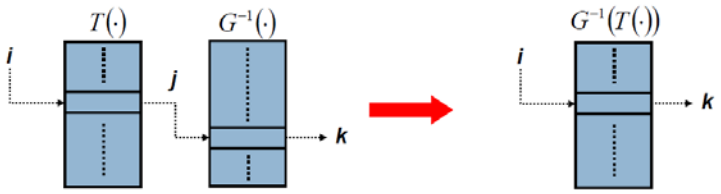
$$y = T(x) = \int_0^x p_x(\xi)\, d\xi$$

$$w = G(z) = \int_0^z p_z(\xi)\, d\xi$$

As y and w are uniformly distributed within the same interval:

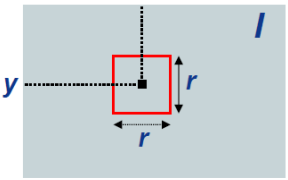
$$z = g^{-1}(w) \quad w = y \Rightarrow z = G^{-1}(T(x))$$

We can notice that T(.) can be obtained by equalizing the histogram of the input image, which is known, and $G^{-1}(.)$ by equalization of the known desired histogram and then inversion of the obtained transformation.



3.2.4 – LOCAL HISTOGRAMS

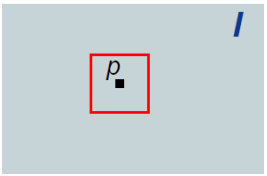
The methods described thus far can fail in enhancing details in small image areas, due to pixels belonging to such areas being too small. A fraction of the overall pixel quantity is able to influence significantly the estimated global transformation.
To enhance small image details, the studied methods (contrast stretching, histogram equalization/specification) can instead be applied to local histograms calculated over a sliding window.



Accordingly, the intensity transformation to map a pixel is estimated based upon the histogram of a window centred at the pixel. Once the current pixel has been mapped, the window is slid over the next one to compute a new transformation.
For the sake of computational efficiency, it is worthwhile computing local histograms through incremental calculation schemes.

4 - SPATIAL FILTERING

Spatial Filters (aka Local Operators) compute the new intensity of a pixel, p, based on the intensities of those belonging to a neighbourhood of p. They accomplish a variety of useful image processing functions, such as e.g. denoising and sharpening (edge enhancement).
An important sub-class is given by the so-called **Linear Shift-Invariant (LSI) operators**.
Straightforward extension of 1D signal theory dictates their application to consist in a 2D convolution between the input image and the impulse response function (point spread function or kernel) of the LSI operator.



When the output value at a certain pixel p is based on the input value and also on his neighbours, this operator is called Spatial Filter and we would like to apply it because we can improve the quality of the image, in particular we can improve the noise.

4.1 - LINEAR SHIFT INVARIANT (LSI) OPERATORS

We have linear and non-linear spatial filters and linear spatial filter are known as LSI operators (linear shift invariant). An LSI operator computes a convolution between the input image and the so-called **kernel of the filter**.
An LSI operator applied to a 1D signal consists in computing a convolution, so it's just a 2D extension rather than a 1D convolution.
We use a little bit of notation to reach the definition. Let's consider continuous 2D signals (then we will study how to apply it to digital images).
We define T and we apply it to get the output o (also 2D signal). T is linear if the relation in the blue square is satisfied. If we have the input as a combination of the weighted sum of ai and bi, the output is the same result but of the responses (known as **superposition of effects**).

Given an input 2D signal i(x,y), a 2D operator $T\{\cdot\} = o(x,y) = T\{i(x,y)\}$ is said to be **LINEAR** iff:

$$T\{ai_1(x,y) + bi_2(x,y)\} = ao_1(x,y) + bo_2(x,y)$$

with $o_1(\cdot) = T\{i_1(\cdot)\}, o_2(\cdot) = T\{i_2(\cdot)\}$ and a, b two constants.

The second property we define is the shift invariant: we have a signal and we apply the operator obtaining a certain output. If we shift the input (it's a 2D shift), if we translate somehow the input and apply T, we get a translated version of the output we had before. The same translation applied into the input will be seen into the output. We define the shift as x0 and y0.

The operator is said to be **SHIFT-INVARIANT** iff: $T\{i(x-x_0, y-y_0)\} = o(x-x_0, y-y_0)$

Now we can combine the 2 properties together and something interesting will happen. Assume that the signal is not generic but it's expressed as a weighted sum of elementary functions translated by a certain amount. We can express our input as a linear combination of the elements.
So, the input is a weighted sum: if this is true, we can very easy come up with the expression of the output: we can compute analytically the output.
Let's see why: what could we say if we apply linearity? If we compute T on top, we get that the weights are going to be the same, we need just to substitute to the input signal, the output signal.

Then we also know that our T is shift invariant, so if the input is shifted and the filter is shift invariant, the output will be just a shift of the input: T applied to the shift ek is the same as hk.
What we have found? If our T is linear and shift invariant and the input can be expressed as a weighted sum, the output is just expressed as the weighted sum translated of the elementary functions.
We can build the output just by combining the weights and the responses to the elementary functions and translating them.

Let's now assume that $i(x,y) = \sum_k w_k e_k(x - x_k, y - y_k)$ and pose $h_k(\cdot) = T\{e_k(\cdot)\}$, it follows that:

$$\begin{aligned} o(x,y) &= T\left\{\sum_k w_k e_k(x - x_k, y - y_k)\right\} \\ &= \sum_k w_k T\{e_k(x - x_k, y - y_k)\} \qquad (L) \\ &= \sum_k w_k h_k(x - x_k, y - y_k) \qquad (SI) \end{aligned}$$

i.e., in the **input is a weighted sum of displaced elementary functions**, the **output** is given by the same weighted sum of the **displaced responses** to the elementary functions.

4.1.1 - IMPULSE RESPONSE AND CONVOLUTION

It can be shown that **any 2D signal can be expressed as a (infinite) weighted sum of displaced unit impulses (DIRAC DELTA FUNCTION),** known as the sifting property of the unit impulse:

$$i(x,y) = \iint_{-\infty}^{+\infty} i(\alpha,\beta)\delta(x - \alpha, y - \beta)\, d\alpha d\beta$$

Accordingly, due to linearity and shift-invariance, the output signal can be expressed as the same (infinite) weighted sum of the displaced responses to the unit impulses:

$$o(x,y) = T\{i(x,y)\} = \iint_{-\infty}^{+\infty} i(\alpha,\beta)\, h(x - \alpha, y - \beta)\, d\alpha d\beta$$

$h(x,y) = T\{\delta(x,y)\}$ is the impulse response (also point spread function or kernel) of the operator, i.e. the output signal when the input signal is a unit pulse. The above operation between the two functions i(x,y), i.e the input signal, and h(x,y), i.e. the impulse response of the operator, is called CONTINUOUS 2D CONVOLUTION.

There are certain classes of signals which can be expressed in this way and, thanks to these properties, this allows to easily compute the output.
Other signals cannot be expressed in this way, but can we decompose them in this form with a special manipulation? Yes, we can always decompose a signal (in every dimension) in elementary functions and the simplest functions we can think of are the infinite number of impulses.

My input signal i(x,y) is expressed with a double integral. The input is given by the input in α and β multiplied by δ (unit impulse, known as delta function).
What we get here is just the impulse response and so we come up with the final result. This operation is called convolution, so applying a LSI operator to the input is about computing the convolution between the input signal and the input response of the operator.

$$\delta(x,y) = \begin{cases} \neq 0: & (x,y) = (0,0) \\ = 0: & elsewhere \end{cases}$$

$$i(x,y) = \iint_{-\infty}^{+\infty} i(\alpha,\beta)\,\delta(x-\alpha, y-\beta)\,d\alpha d\beta = \quad \leftarrow \text{we apply linearity}$$

$$= \iint_{-\infty}^{+\infty} i(\alpha,\beta)\,T\{\delta(x-\alpha, y-\beta)\}\,d\alpha d\beta = \quad \leftarrow \text{we apply shift invariant}$$

$$= \iint_{-\infty}^{+\infty} i(\alpha,\beta)\,h(x-\alpha, y-\beta)\,d\alpha d\beta = o(x,y)$$

PROPERTIES OF CONVOLUTION

We will often denote the convolution operation by the symbol *, e.g.

$$o(x,y) = i(x,y) * h(x,y)$$

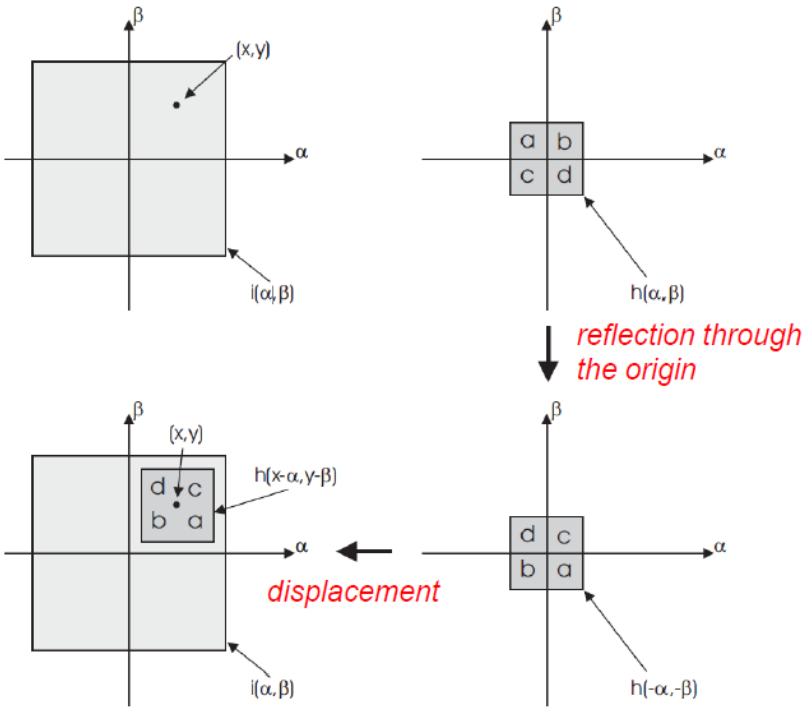
Some useful properties of convolution are as follows:

ASSOCIATIVE PROPERTY: $f * (g * h) = (f * g) * h$

COMMUTATIVE PROPERTY: $f * g = g * f$

DISTRIBUTIVE PROPERTY WRT THE SUM: $f * (g + h) = f * g + f * h$

CONVOLUTION COMMUTES WITH DIFFERENTIATION: $(f * g)' = f' * g = f * g'$



If "i" is the image and "h" is the response, what we can do with these 2 signals? A graphical view of convolution is the following: we multiply together some values and we add them because of the integral, so we multiply and add the 2 signals.

These 2 signals are not treated in the same way: here we can represent "i" as the plane in which both i and h are defined, so this is the domain according to this formula.

We take a little domain non zero in a small neighbourhood of the origin.

In the third image we can see that h is swapped about the origin, it is reflected. We use h-α and h-β and in the visualization we shift h at (x,y), so in a convolution before taking the multiplication, we reflect the image and we shift it.

4.1.2 - CORRELATION

The correlation of signal i(x,y) with signal h(x,y) is defined as:

$$i(x,y)\ o\ h(x,y) = \iint_{-\infty}^{+\infty} i(\alpha,\beta)\ h(x+\alpha, y+\beta)\ d\alpha d\beta$$

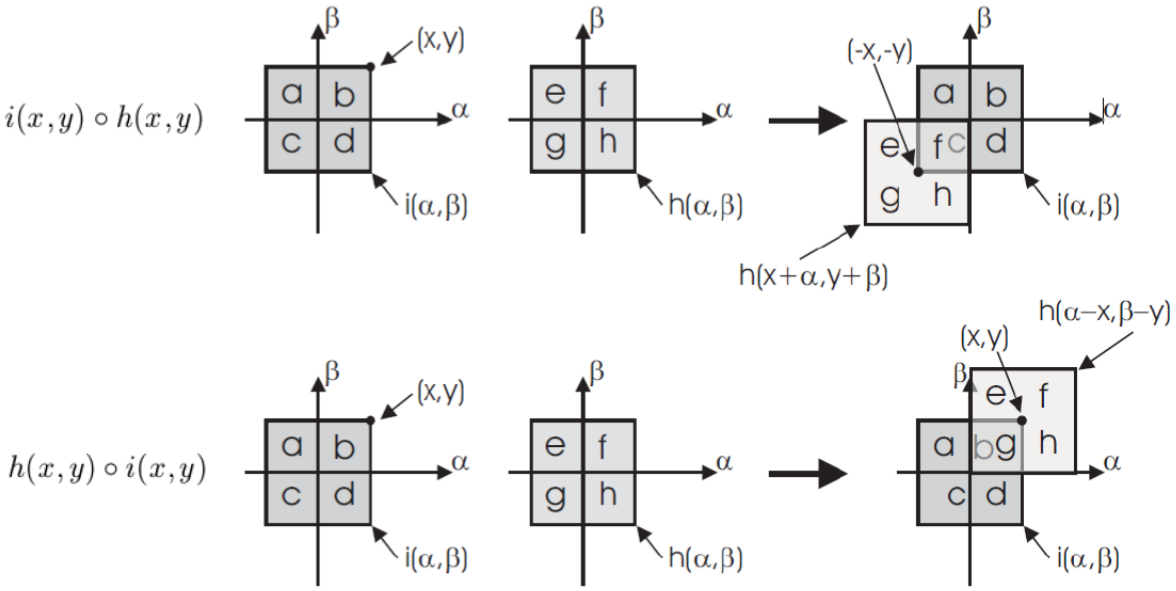
Accordingly, the correlation of h(x,y) with i(x,y) is given by:

$$h(x,y)\ o\ i(x,y) = \iint_{-\infty}^{+\infty} h(\alpha,\beta)\ i(x+\alpha, y+\beta)\ d\alpha d\beta$$

Unlike convolution, correlation is **not commutative**:

$$\begin{aligned} h(x,y) \circ i(x,y) &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} h(\alpha,\beta)\, i(x+\alpha, y+\beta)\ d\alpha\, d\beta \\ &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} i(\xi,\eta)\, h(\xi-x, \eta-y)\ d\xi\, d\eta \\ &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} i(\alpha,\beta)\, h(\alpha-x, \beta-y)\ d\alpha\, d\beta \\ &\neq i(x,y) \circ h(x,y) \end{aligned}$$

It is important to not confuse correlation and convolution: in the convolution we have minus, in the correlation we have plus! Moreover, this is a correlation of i towards h because correlation isn't commutative!



4.1.3 – CONVOLUTION VS CORRELATION

The correlation of h with i is similar to convolution: the product of the two signals is integrated after displacing h without reflection. Hence, if h is symmetric about the origin (h(x,y)=h(-x,-y)), the convolution between i and h (i*h=h*i) is the same as the correlation of h with I:

$$\begin{aligned} i(x,y) * h(x,y) = h(x,y) * i(x,y) &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} i(\alpha,\beta)\, h(x-\alpha, y-\beta)\ d\alpha\, d\beta \\ &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} i(\alpha,\beta)\, h(\alpha-x, \beta-y)\ d\alpha\, d\beta \\ &= h(x,y) \circ i(x,y) \end{aligned}$$

It is worth observing that correlation is never commutative, even if h is symmetric about the origin. To recap, in correlation we don't reflect the image, while in convolution we do that, furthermore:

1.	$i * h = h * i$	*(convolution is commutative)*
2.	$i \circ h \neq h \circ i$	*(correlation is not commutative)*
3.	$i * h = h * i = h \circ i$	*(if h is symmetric about the origin)*

4.1.4 – DISCRETE CONVOLUTION

Let us now consider a discrete 2D LSI operator, T{·}, whose response to the 2D discrete unit impulse (**KRONECKER DELTA FUNCTION**) is denoted as H(I,j):

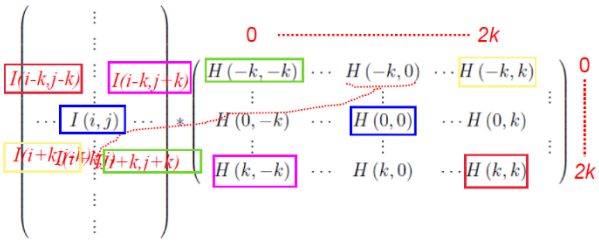
$$H(i,j) = T\{\delta(i,j)\} \quad \text{with} \quad \begin{cases} \delta(i,j) = 1 & at(0,0) \\ \delta(i,j) = 0 & elsewhere \end{cases}$$

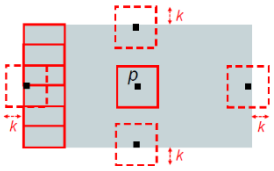
Given a discrete 2D input signal, I(i,j), the output signal, O(i,j), is given by the discrete 2D convolution between I(i,j) and H(i,j):

$$O(i,j) = T\{I(i,j)\} = \sum_{m=-\infty}^{+\infty} \sum_{n=-\infty}^{+\infty} I(m,n)H(i-m, j-n)$$

Analogously to continuous signals, **discrete convolution consists in summing the product of the two signals where one has been reflected about the origin and displaced**. The previously highlighted four major convolution properties hold for discrete convolution alike.

On one hand we have the image and on the other hand we have the impulse response.
Here m and n are the corresponding discrete integrations to the alfa and beta (we do not have continuous integrals any more).
I would like to have the whole output image, so I need to do that operation across the whole input image. If I start with the position (i, j), then I have to flip the kernel. What if I have to compute the whole output image? I slide the kernel along the input image to compute at each position the output. This is also called **SLIDING WINDOW COMPUTATION** and it is a quite common paradigm.
We have to do it as many times as the number of inputs.





If we try to implement this algorithm, we have to take a second output image because when we compute the next value, if we override the input image, we do not do correctly the computation.

In image processing both the input signal (image) and the impulse response(kernel) are stored into matrixes of given sizes, such the one in the figure.
Conceptually, we need to slide the kernel across the whole image to compute the new intensity at each pixel without overwriting the input matrix.

```
\*  I: M*N pixels, H: (2k+1)*(2k+1) coefficients *\

for (i=k; i<M-k; i++)
   for (j=k; j<N-k; j++){
      temp=0;
      for (m=-k; m<=k; m++)
         for (n=-k; n<=k; m++)
            temp=temp+I[i-m,j-n]*h[m+k,n+k];
      O[i,j]=temp;
   }
```

If we cannot compute the convolution on the borders, we can have a smaller image, so the central part of the image become the output.
Another option is to pad the input image adding k columns and k rows and we put into these additional rows and columns zeros. In this way we can apply the convolution in the whole image.

4.2 - MEAN FILTER

Mean filtering is the **simplest (and fastest) way to carry out an image smoothing** (i.e. low-pass filtering) **operation**. Smoothing is often aimed at image denoising, though sometimes the purpose is to **cancel out small-size unwanted details** that might hinder the image analysis task.
In modern feature-based computer vision algorithms smoothing is key to create the so-called **scale-space**, which endows these approaches with scale invariance.
The mean filter consists just in **replacing each pixel intensity by the average intensity over a given neighbourhood.**

$$\begin{bmatrix} 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \\ 1/25 & 1/25 & 1/25 & 1/25 & 1/25 \end{bmatrix} = \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

It is an LSI operator, as it can be defined through a kernel: below, the 3x3 and 5x5 mean filters.
Mean filtering is inherently fast because multiplications are not needed. Moreover, it can be implemented very efficiently by incremental calculation schemes (BOX-FILTERING).

In case of an image, frequencies are the same in all n-dimensional signals, so we can say that frequencies in 1D create the output with combinations of cosine functions. So, the low pass filtering throws away high frequencies.
So, we can use the mean filter to throw away noise in the image.

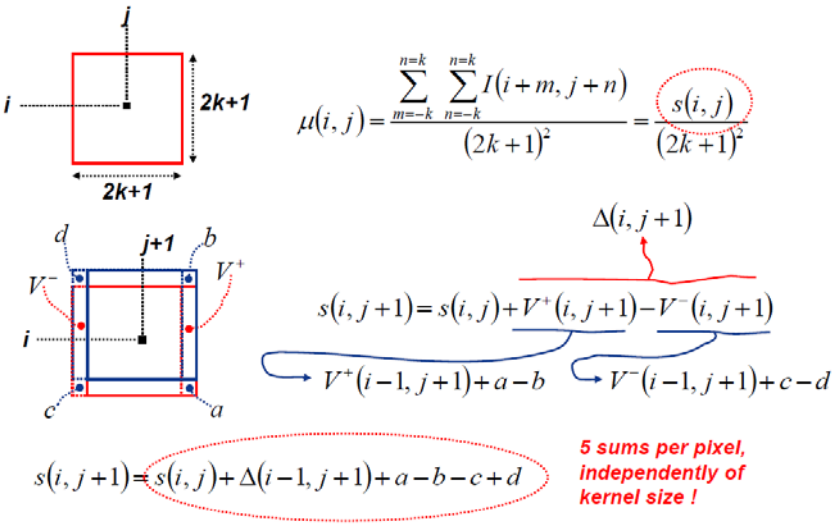
The problem is that if I have a window that is too large, I start mixing white pixels with dark pixels and the image will become more blur and it's not what we want. I should average only similar pixels.
So, taking the **size of the window is a trade-off**: **the more the window is big, the more is powerful, but the more is high the risk to have blurring.**

4.2.1 – BOX FILTERING

This is an LSI operator. The main application is denoising (or smoothing), but also in CV we can use Mean Filter because smoothing is a way to cancel out small things in the image.
The larger the size of the smoothing filter the larger the number of little things cancelled out in the image.

Difference between size and scale? size = object size in the world, scale = object size in the image: it depends on distance of the focal lens.

Smoothing filters are also used to perform multiscale images.
If we are really interested in applying a mean filter to reduce noise, computationally we compute a convolution or can we do better? When we apply a particular type of smoothing filter like the mean filter, we wouldn't do that: we don't implement a mean filter to a convolution because there are useless multiplications.
Mean Filtering is very fast because it doesn't compute multiplications and we can compute this filter according to an incremental calculation that makes computation independent of filter size.



In the formula $\mu(i,j)$, we have the sum of all pixels of the image over the number of pixels of the window. We call the numerator s(i, j): the core of the computation is to compute the sum.
What's the difference between the sum of the dotted windows and the other? We use the sum along these columns and we subtract the pixels we have already summarized (V+ and V- are the column sums).
To have o(1) complexity we consider the window at position (i-1, j+1) so the pixel in the same column position but the previous row.
The previous window was the red one, the new V+ is related to this? To get the new V+, we could take a previous V+ and add a pixel and subtract another pixel.
The new V- is related to the previous V-: same column index, previous row index.
We'll have that s at the new position is the same + Δ + 2 pixels - 2 pixels: that's an optimization which considers vertical and horizontal overlapping.
So, we have only to count 4 pixels and not the whole window: that's why the filter is superfast!

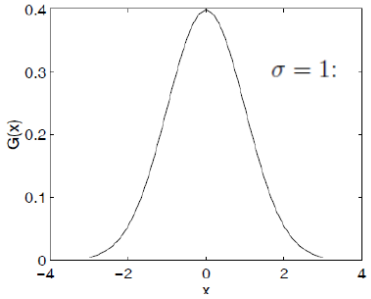
Every image has noise, but in this experiment, we have corrupted the image by **GAUSSIAN NOISE**.
Gaussian noise is a synthetic noise added to intrinsic noise of every image. (slide 15 pack 4)
In practice, adding a gaussian noise means that **we sample random values and use a standard deviation of a gaussian.**
In the 3x3 mean there's less variation and the zones are more homogeneous. So, we have a smooth image but we have also blur in the contours, the borders. Larger filter would mean better denoising so there's less noisy variation, but more blur: that's because the window size is larger.

Another type of noise is called **IMPULSE NOISE**: it is a noise that is found only in a certain number of pixels. So, some pixels are totally wrong, they're crazy: we pick them randomly and we color them as black or white. Their name is **outliers**. (slide 16 pack 4)

4.3 – GAUSSIAN FILTER

1D Gaussian

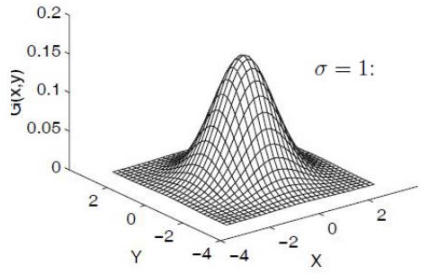
$$G(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$



The Gaussian Filter is an **LSI operator** whose **impulse response is a 2D Gaussian function** (with zero mean and constant diagonal covariance matrix).
This is probably the most useful and used filter. How do we define a filter? The filter is defined by the impulse response that is a 2D gaussian function. The mean of the gaussian in zero and the bell is more spread if sigma is big.
2D gaussian function is a product of due gaussian function along X and along Y. We multiply them and we have the formula. It could be seen as a probability distribution, but it's not so interesting.

2D Gaussian

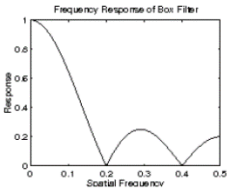
$$G(x,y) = G(x)G(y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



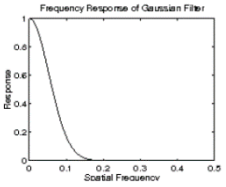
Circularly Symmetric

It is **circularly symmetric**: any point at a certain distance from di origin has the same value. **If sigma is smaller, the bell is taller, else is lower and spread.**

The higher σ, the stronger the smoothing caused by the filter. This can be understood, e.g., by observing that as σ increases, the weights of closer points get smaller while those of farther points larger. Likewise, the Fourier transform of a Gaussian is a Gaussian with $\sigma_\omega = 1/\sigma$, so that the higher σ the narrower the bandwidth of the filter.

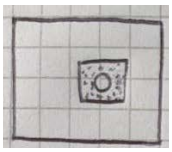


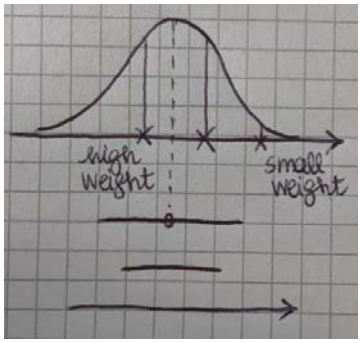
The Gaussian filter is a more effective low-pass operator than the Mean Filter, as the frequency response of the former is monotonically decreasing (the higher the frequency the higher the attenuation) while the latter exhibits significant ripple.



What if we perform a convolution? Why do we get a smoothing effect? Convolution is a weighted sum and we get a smoothing because we are taking averages of neighbours where we have small variation.
The key parameter in gaussian filter is the sigma, so the larger is sigma the stronger is the smoothing: **a larger sigma in space means a smaller sigma in frequency**.
We prefer Gaussian Filtering because more is the pixel closer to the pixel chosen, more is the variation smaller and the probability of the pixel to be part of a surface is high.

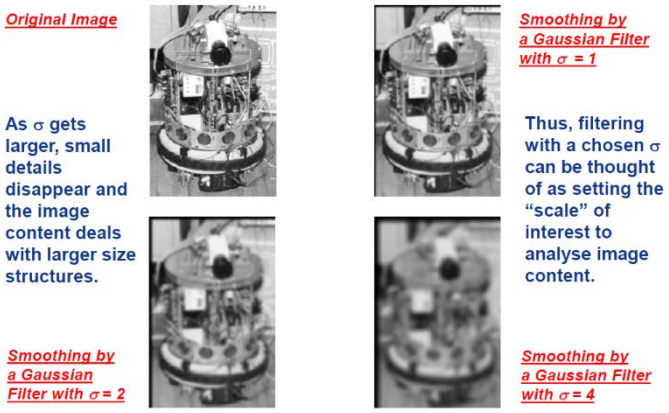
We'll not discuss the frequency domain, but we can look at why the gaussian filter is better than a mean filter is the frequency of response. In the second image we are sure that the higher the frequency the higher is the smoothing, the attenuation, while in the first image it is different because if we go in different points, we have ambiguous information.





Furthermore, a low pass filter like the Gaussian is preferable to others because it **goes to 0 quite fast**.
If we use a Mean Filter, all the neighbours have the same weight 1/A, while with Gaussian Filter we smooth together pixels which are similar and belong to the same surface.

On one hand, it is showing that the higher the sigma more smoothing we have.
On the other hand, we have that more smoothing causes more blurring.



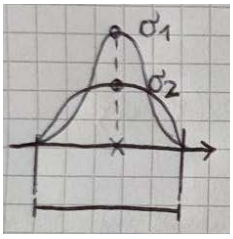
Scale space: is related to a certain class of algorithms. The level of details will decrease in these images.
Scale is the size of the image: in the first image we have small scale of the image because we see small details.
In the last image we have a higher scale. We can use the smoothing to carry out the scales in analysis.

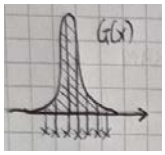
4.3.1 – DISCRETE GAUSSIAN

The **discrete Gaussian kernel** can be obtained by sampling the corresponding continuous function, which is however of infinite extent. A finite size must therefore be properly chosen.
To this purpose, we can observe that:

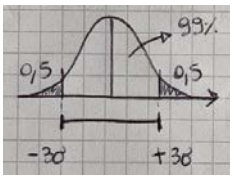
- The larger is the size, the more accurate turns out the discrete approximation of the ideal continuous filter
- The computational cost grows with filter size
- The Gaussian gets smaller and smaller as we move away from the origin

Therefore, we should use larger sizes for filters with high σ, smaller size whenever σ is smaller. A rule-of-thumb to choose the size of the filter, given σ, would then be quite useful.
As the interval [-3σ, +3σ] captures 99% of the area ("energy") of the Gaussian function, a typical rule dictates taking a (2k+1)x(2k+1) kernel with: k=$\lceil 3\sigma \rceil$





How many samples we need? We have to approximate the operation. More samples we take, better's the approximation. Sometimes we pick lots of zero, we increase the overhead for no reason, so we need a trade-off.
$\sigma_2 > \sigma_1$ → in σ_2 we can take more samples because we've less zero values.



It may be either convenient (to speed-up the filtering operation) or mandatory (e.g. on embedded platforms without floating-point unit) to convolve the image by an integer rather than floating point kernel.
An **integer Gaussian kernel** can be attained by dividing all coefficients by the smallest one, rounding to the nearest integer and finally normalizing by the sum of the integer coefficients. The final normalization yields unity gain.

Obviously, integer multiplication is faster than float multiplication. In general purpose computer, we'd like to use integer multiplications, while in other systems, where cameras are embedded, we can use floating point units.

$$k \rightarrow \quad k_1 = \frac{1}{k_{min}} k \rightarrow k_2 = round(k_1) \rightarrow k_3 = \frac{1}{sum(k_2)} k_2$$

Let's consider a flat area that is all white: all pixels are 255 and if we multiply them for the kernel, the final result will be the sum of the elements of the kernel * 255. So, we change the range of the input

signal in a larger range. We should **guarantee a unity gain**! → the sum must be 1 and if we divide all of them by their sum, we reach the correct values.

To further speed-up the filtering operation, one can **deploy the separability** property: due to the 2D Gaussian being the product of two 1D Gaussians, the original 2D convolution can be split into the chain of two 1D convolutions, i.e. either along x first and then along y, or vice versa.

$$I(x,y) * G(x,y) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} I(\alpha,\beta)G(x-\alpha,y-\beta)d\alpha\, d\beta$$

$$G(x,y) = G(x)G(y)$$

Our function can be seen as a couple of function G(x)*G(y). It's the case of a gaussian because it's separable.

$$I(x,y) * G(x,y) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} I(\alpha,\beta)G(x-\alpha)G(y-\beta)d\alpha\, d\beta$$

$$I(x,y) * G(x,y) = \int_{-\infty}^{+\infty} G(y-\beta)\left(\int_{-\infty}^{+\infty} I(\alpha,\beta)G(x-\alpha)d\alpha\right)d\beta$$

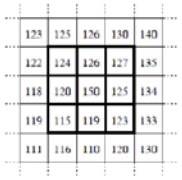
We can have two 1D convolution along the two directions. If we have the convolutions, we can put themself together.

$$I(x,y) * G(x,y) = (I(x,y) * G(x)) * G(y) = (I(x,y) * G(y)) * G(x)$$

Accordingly, the **speed-up brought in by the separability property can be expressed as:**

$$S = \frac{(2k+1)^2}{2\cdot(2k+1)} = k + \frac{1}{2} = 3\sigma + \frac{1}{2}$$

4.4 – MEDIAN FILTER



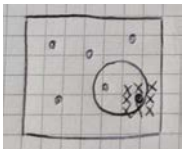
Neighbourhood values:

115, 119, 120, 123, 124, 125, 126, 127, 150

Median value: 124

Non-linear filter whereby each pixel intensity is replaced by the median over a given neighbourhood, the median being the value falling half-way in the sorted set of intensities.

$$median\,[A(x)+B(x)] \neq median\,[A(x)] + median\,[B(x)]$$



Median filtering counteracts impulse noise effectively, as outliers (i.e. noisy pixels) tend to fall at either the top or bottom end of the sorted intensities. Median filtering tends to **keep sharper edges** than linear filters such as the Mean or Gaussian:

Example of Median Filter

If we have some pixels whose value is: 0 0 0 255 0 0 0

We pick the center 0 255 0 and then we cancel/ignore the 255.

So the line becomes: 0 0 0 0 0 0 0.

This is not providing any blur, so the outliers are filtered but there's no blur.

Example of Mean and Median Filter

… 10 10 40 40 … → … 10 20 30 40 … (Mean Filter

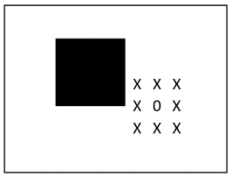
→ … 10 10 40 40 … (Median Filter)

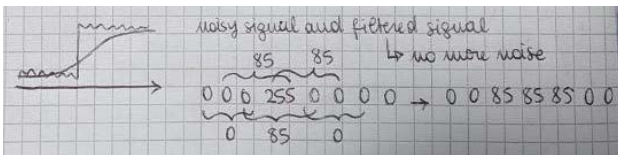
With 10 10 40 40 we see a lot of contrast at this edge, but if we use a mean filter we have 10 20 30 40 because we have: 10 10 10 → 10 10 10 40 → 20

10 40 40 → 30 40 40 40 → 40

What about a median? 10 10 10 → 10 10 10 40 → 10
10 40 40 → 40 40 40 40 → 40

It will pick already existing intensities and doesn't introduce any blur. We'll never see an intensity which wasn't in the line.



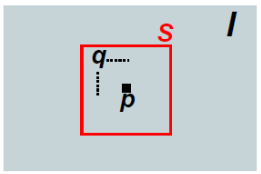


When we compute this filter, it is better to take more neighbours, as much as possible. But if we take an initial pixel near dark ones, the mean would be gray: not working!

If the problem is impulse noise, is recommended to use the median filter because it works very well.
Sometimes we found outliers in images which may produce wrong results like the case of disparity estimation.

4.5 – BILATERAL FILTER

Bilateral Filter is an **advanced non-linear filter** to accomplish denoising of Gaussian-like noise without blurring the image (aka edge preserving smoothing).



$$O(p)=\sum_{q\in S} H(p,q)\cdot I_q$$

$$H(p,q)=\frac{1}{W(p,q)}\cdot G_{\sigma_s}\left(d_s(p,q)\right)\cdot G_{\sigma_r}\left(d_r\left(I_p,I_q\right)\right)$$

$$d_s(p,q)=\|p-q\|_2=\sqrt{(u_p-u_q)^2+(v_p-v_q)^2}$$ (euclidean distance) → ***Spatial Distance***

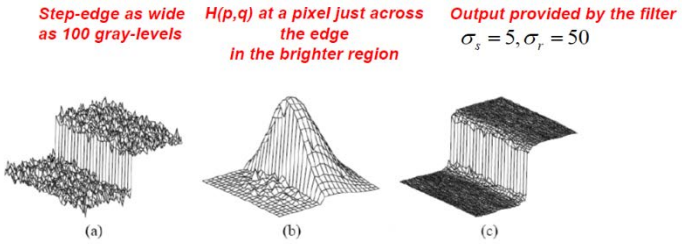
$$d_r\left(I_p,I_q\right)=\left|I_p-I_q\right|$$ → ***Range (Intensity) Distance***

$$W(p,q)=\sum_{q\in S} G_{\sigma_s}\left(d_s(p,q)\right)\cdot G_{\sigma_r}\left(d_r\left(I_p,I_q\right)\right)$$ → ***Normalization Factor (Unity Gain)***

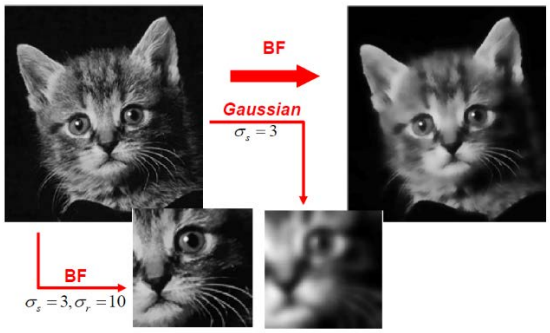
We'll try to denoising the image without blurring the image. We give more weight to pixels whose difference is higher. The similarity of the pixels is the main concept of the filter.
If we consider P, the pixel, we have a supporting window maybe 7x7 or 11x11 (it is a choice). q denote the running pixels. We compute the output as a weighted sum.
What rule should H implement according to our previous reasoning? It should be large when q is close to p and Iq is similar to Ip, small otherwise.

We have 2 gaussian functions: Gσs (s = space) and Gσr (r = range).
The argument of the first gaussian Gσs is the gaussian function of the distance between P and Q. It's small when the argument is large, so is a decreasing function of the distance.
Second gaussian Gσr: the argument is the difference between Ip and Iq. We have large weight only if the pixels are close.

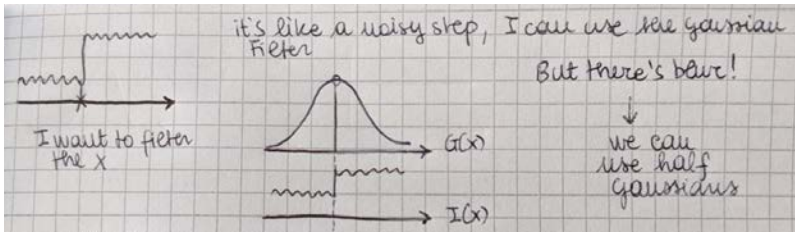
We want **guarantee to have a unity gain**, se we have to consider also the W(p,q). W is a function which aims to **normalize our coefficients**. It is the sum of all H values of the 2 gaussians.

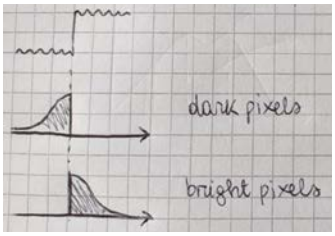


Given the supporting neighbourhood, **neighbouring pixels take a larger weight as they are both closer and more similar to the central pixel**. At a pixel nearby an edge, the neighbours falling on the other side of the edge look quite different and thus cannot contribute significantly to the output value due to their weights being small. Here we have an **asymmetric type of impulse response**.



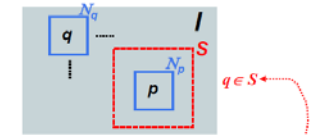
This type of filter is used in many programs for photos but it's **not fast** because it has to compute all the pixels of the image before applying the algorithm.
If we have to do something in a short time, we have to reach an approximation and cannot use the bilateral filter.





4.6 – NON-LOCAL MEANS FILTER

More recent **edge preserving smoothing filter**. The key idea is that the similarity among patches spread over the image can be deployed to achieve denoising.

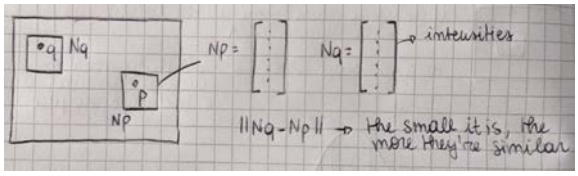


$$O(p) = \sum_{q \in I} w(p,q) I(q)$$

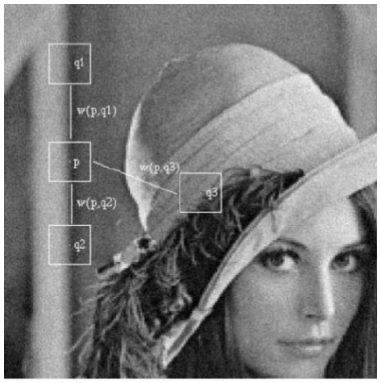
$$w(p,q) = \frac{1}{Z(p)} e^{-\frac{\|N_p - N_q\|_2^2}{h^2}}$$

$$Z(p) = \sum_{q \in I} e^{\frac{\|N_p - N_q\|_2^2}{h^2}}$$

We want to look at the whole image to find samples to smooth. Why should I focus just to a small window? That's why the non-local means filter.
We take local means in a bilateral filter because the underline assumptions is more or less that: if we go farther away probably we'll have different objects, but maybe it exists an object which is large in the image, so we can consider not only a window, but a **pool of windows**.
When we want to smooth a pixel P, we can use the whole image to find similar pixels where similar means likely belonging to the same object.
I'll ask for a certain pixel to be a friend and if it's similar, I will use it for the filter.



In this image we can consider all the pixels on the vertical line, like q1 and q2, but q3 is not a friend because it doesn't belong to the same object: we need a criterion to recognise friends and non-friends.



We compare the patch around the given pixels. So, we're going to smooth P considering all the other pixels in the image and we'll weight them using the similarity of the patches. When Np is similar to Nq, w is high.

Image Corrupted by Gaussion Noise (σ=20)

Gaussian Filter

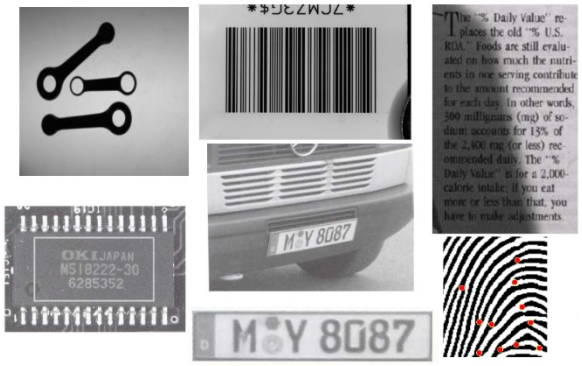
Non-local Means Filter (N=7×7, S=21×21, h =10·σ)

With the Gaussian Filter the face is more homogeneous but we have a lot of blur, while with the Non-local Means Filter we do not observe any significant blur.

5 - IMAGE SEGMENTATION

Denoted as ***P(x,y)*** a vector-valued function encoding a set of image properties (such as intensity, colour, texture, contrast), the **goal of segmentation** is to **partition the image into disjoint homogeneous regions according to P**.
Typically, a good segmentation should **preserve spatial proximity**, i.e. two nearby pixels must belong to the same region unless they exhibit significantly different P values, and **provide relatively large regions** featuring a few holes and well-localized, smooth boundaries.
In many computer vision's tasks, segmentation brings in key **semantic knowledge** on the scene, as it splits the image into semantically meaningful parts (e.g. foreground/background, individual objects, moving/static pixels…) on which further analysis can then be focused.
In most practical applications related to industrial vision, segmentation relies on just a single image property, e.g. intensity ***P(x,y) = I(x,y)*** or colour ***P(x,y) = [I_r(x,y) I_g(x,y) I_b(x,y)]^T***.

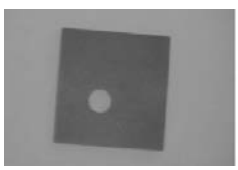
5.1 – IMAGE BINARIZATION

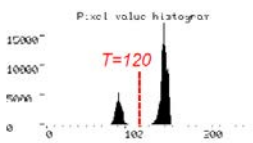


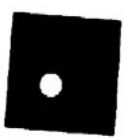
In a variety of applications, the objects of interest (foreground) are neatly darker/brighter than the irrelevant areas of the scene (background).
In many industrial applications this is achieved by **backlighting**, whereby the **object is placed between the light source and the camera** so as to cast onto the image a very dark shadow representing object's shape.
In such circumstances, the first image analysis step consists typically in **image binarization**, i.e. **segmentation of image pixels into two disjoint regions corresponding to foreground** (dark/bright intensities) **and background** (bright/dark intensities).
If the application calls for analysis of a single object per image, binarization would deliver all the required segmentation information.
Conversely, the **foreground region must be further split into sub-regions corresponding to individual objects**, which is usually achieved by a successive image analysis step referred to as **image labeling** (also connected components labeling) due to pixels belonging to different objects being given different labels.

5.1.1 – BINARIZATION BY INTENSITY THRESHOLDING

Inherently-binary images exhibit a clearly bimodal gray-level histogram, with **two well-separated peaks** corresponding to foreground and background pixels.
Therefore, binarization can be achieved straightforwardly by a means of a thresholding operator deploying a suitably chosen threshold, **T**.

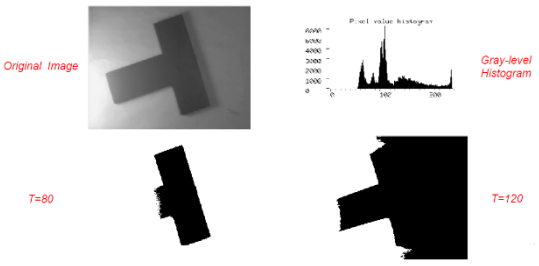






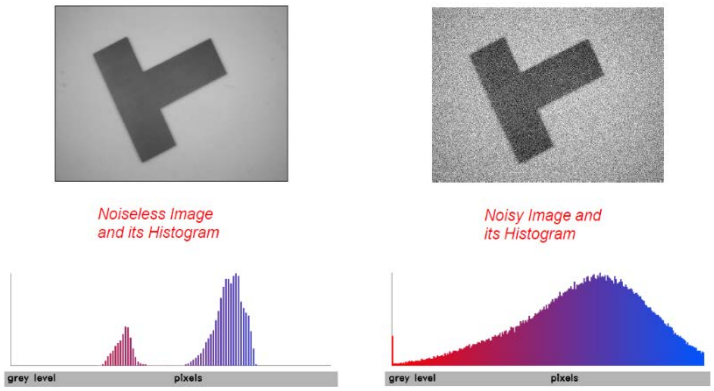
```
/* Binarization by a global threshold, objects darker than background
# define  FOREGROUND  0
# define  BACKGROUND 255
# define  THRESHOLD  120

for(i=0;i<N; i++)
for(j=0;i<M; j++)
if (I[i][j]=<THRESHOLD)
   O[i][j]=FOREGROUND;
      else
   O[i][j]=BACKGROUND;
```

Whenever the histogram is not clearly bimodal, e.g. due to illumination varying significantly across the scene (example below), binarization by **intensity thresholding fails** to provide the correct segmentation.



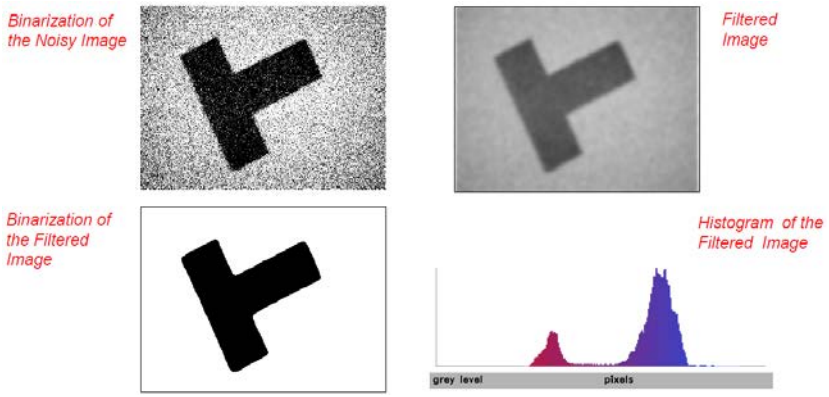
5.1.2 – SMOOTHING AND THRESHOLDING

When the overlap between the two modes is due to noise, **image smoothing** may help improving binarization.



For example, in the first figure we have a dark object against a bright background. Its histogram is quite simple to separate with a threshold to perform a binarization. There's a gap between the two values, so that's ok, it will work.
Now, let's consider adding synthetic noise to the image, like in the second image. Each grey level becomes different and we have a histogram like that: no chance to perform segmentation with threshold.

Obviously, **binarization of the noisy image fails**. Yet **smoothing by a Gaussian filter** provides clearly improves the histogram and allows for a more correct binarization.



So, by applying a Gaussian filter, we obtain that histogram. The levels are not completely separate, but is better so we can use a threshold. Sometimes **smoothing provide a better segmentation, but only in rare cases.**

5.1.3 – AUTOMATIC THRESHOLD SELECTION

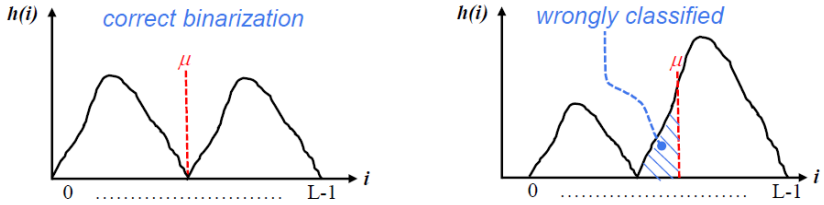
In many practical applications, **stability over time of the lighting conditions cannot be guaranteed.** Such applications mandate a robust, though computationally more demanding, approach whereby an algorithm computes automatically a suitable binarization threshold in each image under analysis.

We understood that the approach is very simple and we can divide a simple binary image in two segments, but how we can accomplish this in practice? We acquire the images, we build and we show the histogram and we pick a suitable threshold.
However, how can we take a good model for the histogram? Picking a threshold manually isn't really advisable in practice because in industrial applications we cannot use a specific threshold for every kind of image. Moreover, we don't know if the power of the lighting or the brightness of the image will never change in the next years, so we have to pick threshold in a dynamic way.

Simple heuristic approaches are as follows:

EMPIRIC METHOD: $T = \mu$

It works as long as **pixels** are **equally distributed between the two classes**. Likewise, a certain properly estimated percentile may be chosen (e.g. either the 20th or 80th percentile if we know dark/bright objects to cover approximately 20% of the image)



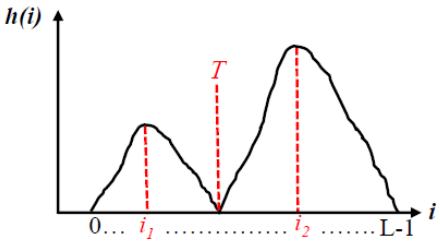
Sometimes there's a hidden assumption: we pick the threshold near the middle, but not always the % of pixel can be divided into two parts more or less equals, as in the first figure. Indeed, in the second figure, we can have a type of histogram where bright pixels are more than 80% of the picture. Here, if we pick the mean, we have a wrong classification!
In industrial application, generally we know the size of the object in the image and we have to find the pixels of the object in the image. We have to choose the correct percentage of the pixels.

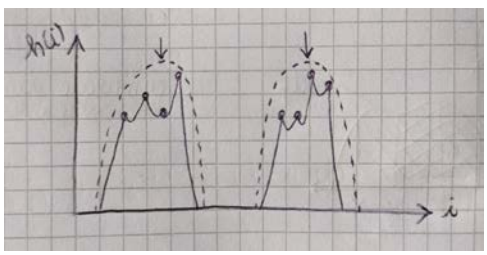
PEAKS METHOD: $T = arg\ min\{h(i) \mid i \in [i_1, i_2]\}$

This method requires **finding the two main peaks**, which often implies smoothing the histogram before looking for peaks to avoid the search be trapped into spurious local maxima.
To find the two main peaks, i_1,i_2:

```
/* 3-levels neighbourhood
((h[i]>h[i-1])&&(h[i]>h[i+1]))

/* 5-levels neighbourhood
((h[i]>h[i-1])&&(h[i]>h[i+1])&&(h[i-1]>h[i-2])&&(h[i+1]>h[i+2]))
```

This is a little more accurate method than the empirics previous method.
Nonetheless, if we use these methods, we generally find spurries elements.





OTSU'S ALGORITHM

A principled and effective automatic threshold selection algorithm is due to Otsu. The key intuition is to **segment the image into two maximally homogeneous regions.**

Accordingly, the optimal threshold is chosen so as to minimize across the gray-level range the so-called ***Within-group Variance*** of the resulting regions, such an indicator measuring **how spread turn out region intensities upon binarization** by a given gray-level.

This is the best known and widely used approach. We can pick a threshold and this threshold is splitting pixels into 2 groups: dark and bright pixels. Which one is the optimal split? Can we estimate the optimality of the splits? We could just try a split and find the optimal one according to some criteria.

Let us define:

$i = 1 \dots L$: gray-levels of the image

N : number of pixels of the image

$h(i)$: i^{th} entry of the image histogram

$p(i) = h(i)/N$: probability of gray-level i $\left(\sum_{i=1}^{L} p(i) = 1\right)$

Accordingly, the mean, μ, and variance, σ^2, of the pmf associated with image gray-levels can be expressed as:

$$\mu = \sum_{i=1}^{L} i\, p(i) \qquad \sigma^2 = \sum_{i=1}^{L} (i-\mu)^2 p(i)$$

Any threshold value, t, would split pixels into two disjoint regions whose associated pmfs have mean and variance given by:

$$\mu_1(t) = \sum_{i=1}^{t} i\, p(i)/q_1(t) \qquad \sigma_1^2(t) = \sum_{i=1}^{t} (i-\mu_1(t))^2\, p(i)/q_1(t)$$

$$\mu_2(t) = \sum_{i=t+1}^{L} i\, p(i)/q_2(t) \qquad \sigma_2^2(t) = \sum_{i=t+1}^{L} (i-\mu_2(t))^2\, p(i)/q_2(t)$$

with

$$q_1(t) = \sum_{i=1}^{t} p(i) \qquad q_2(t) = \sum_{i=t+1}^{L} p(i)$$

The Within-group Variance of the two regions is defined as the weighted sum of their variances:

$$\sigma_W^2(t) = q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t)$$

Minimization of the above quantity would require computing μ_1, μ_2, σ_1^2, σ_2^2 and q_1 ($q_2 = 1 - q_1$) for each gray-level values. Yet, a more efficient approach can be pursued.

$$\begin{aligned}
\sigma^2 &= \sum_{i=1}^{L} (i-\mu)^2 p(i) \\
&= \sum_{i=1}^{t} (i-\mu_1(t)+\mu_1(t)-\mu)^2\, p(i) + \sum_{i=t+1}^{L} (i-\mu_2(t)+\mu_2(t)-\mu)^2\, p(i) \\
&= \sum_{i=1}^{t} \left[(i-\mu_1(t))^2 + 2\,(i-\mu_1(t))\,(\mu_1(t)-\mu) + (\mu_1(t)-\mu)^2\right] p(i) \\
&+ \sum_{i=t+1}^{L} \left[(i-\mu_2(t))^2 + 2\,(i-\mu_2(t))\,(\mu_2(t)-\mu) + (\mu_2(t)-\mu)^2\right] p(i)
\end{aligned}$$

$$\sum_{i=1}^{t} (i - \mu_1(t))\,(\mu_1(t) - \mu)\,p(i) = 0$$

$$\sum_{i=t+1}^{L} (i - \mu_2(t))\,(\mu_2(t) - \mu)\,p(i) = 0$$

Therefore, σ^2 can be expressed as:

$$\sigma^2 = \sum_{i=1}^{t} (i - \mu_1(t))^2\,p(i) + \sum_{i=t+1}^{L} (i - \mu_2(t))^2\,p(i) + (\mu_1(t) - \mu)^2\,q_1(t) + (\mu_2(t) - \mu)^2\,q_2(t)$$

$$\begin{aligned} \sigma^2 &= \left[q_1(t)\sigma_1^2(t) + q_2(t)\sigma_2^2(t)\right] + \left[(\mu_1(t) - \mu)^2\,q_1(t) + (\mu_2(t) - \mu)^2\,q_2(t)\right] \\ \sigma^2 &= \sigma_W^2(t) + \sigma_B^2(t) \end{aligned}$$

$\sigma_B^2(t)$ being the *Between-group Variance*, i.e. an indicator of how well classes are separated one to the other.

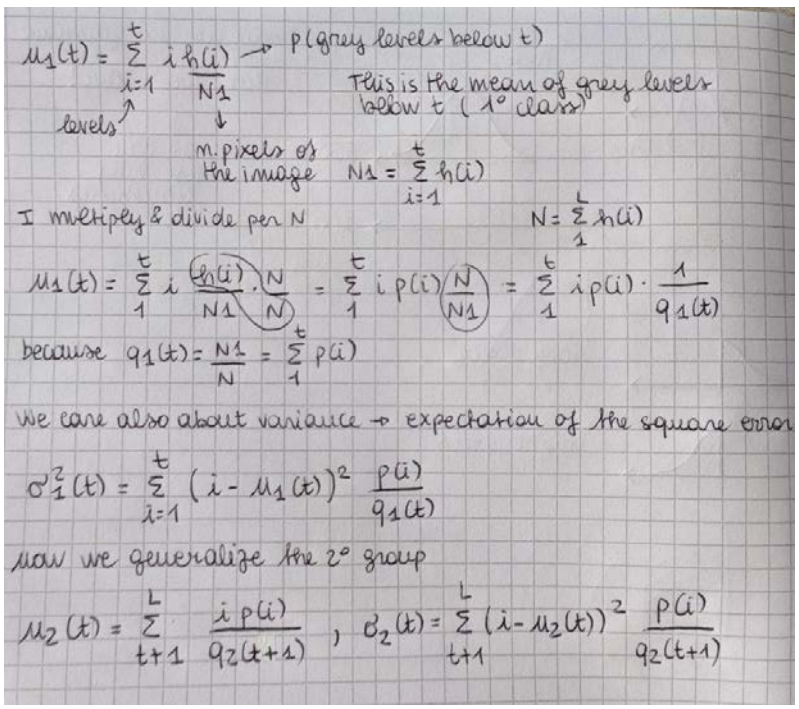
As σ^2 is independent of t, the above relation suggests we might wish to maximize σ_B^2 rather than minimizing σ_W^2, the former being a more efficient procedure as variances need not be calculated.

Further computational savings can be achieved as follows:

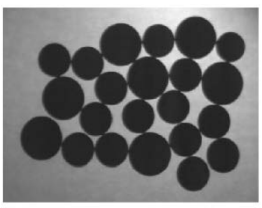
$$\mu = q_1(t)\mu_1(t) + q_2(t)\mu_2(t) \qquad q_2(t) = 1 - q_1(t)$$

$$\sigma_B^2(t) = q_1(t)(1 - q_1(t))(\mu_1(t) - \mu_2(t))^2$$

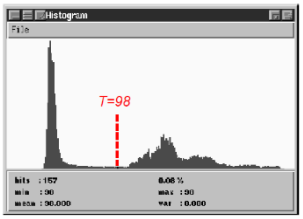
$$\begin{aligned} q_1(t+1) &= q_1(t) + p(t+1) \\ \mu_1(t+1) &= \frac{q_1(t)\mu_1(t) + (t+1)p(t+1)}{q_1(t+1)} \\ \mu_2(t+1) &= \frac{\mu - q_1(t+1)\mu_1(t+1)}{1 - q_1(t+1)} \end{aligned}$$



Input Image

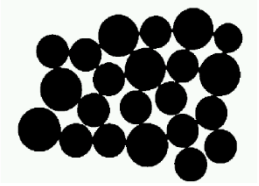


Gray-level Histogram



with threshold computed by Otsu's

Binarized Image

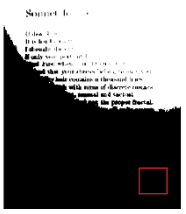


5.1.4 – ADAPTIVE THRESHOLDING

Image of an unevenly lit scene

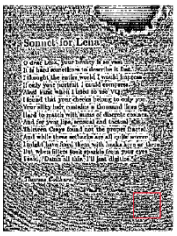
Any global thresholding method relies on the **assumption of uniform lighting** across the scene.
If this assumption is violated, e.g. due to shadows, a **spatially varying (i.e. adaptive) threshold** ought to be employed in case one still wishes to binarize the image.
Usually, adaptive methods compute a specific threshold **at each image pixel** ($T \rightarrow T(x,y)$) based on the intensities within a small neighbourhood (e.g. 5×5, 7×7, 9×9).
However, too small neighbourhood might lack either background of foreground pixels, which would imply segmentation errors unless the issue is dealt with explicit.
Indeed, the smaller is the window, the higher is the probability to have similar pixels, so an homogeneous window.
For the sake of efficiency, rather simple operators, such as **the Mean or the Median, are typically adopted** to compute the threshold value at each pixel.

Binarization by global thresholding



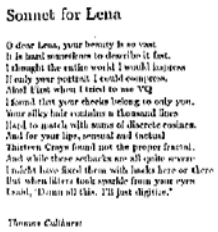
Looking at the pictures, we can say that Otsu's method is wonderful, but it is not magic and, in this case, in the second picture, we don't have a histogram enough good to find the characters.
If we use instead a local window and apply local binarization, we find something interesting, like in the third image. This is very bad because the background pixels are binarized quite randomly, however character pixels seem to be highlighted well, so it's not so bad. So, we can use this application as a pre-operation of a sequence of elaborations.

Binarization by adaptive thresholding



$T(x,y) = \mu(x,y)$, 7×7 neighbourhood

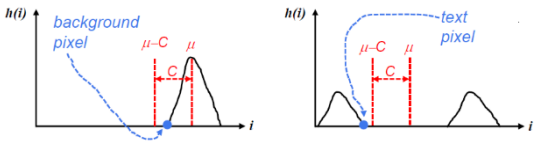
Binarization by adaptive thresholding



$T(x,y) = \mu(x,y)$, C=10 7×7 neighbourhood

In text images, the issue of neighbourhoods lacking both types of pixels can be dealt with by deploying the peculiarity of most such neighbourhoods being associated with background pixels. Hence, a simple trick to improve segmentation dramatically consist in **subtracting a suitable constant** so as to push above threshold most wrongly classified background pixels.

$$T(x,y) = \mu(x,y) - C$$



We can have a pixel below the mean or above the mean. In this case, nearly all the pixels belong to a class, so we can subtract a constant value.

5.2 – COLOUR-BASED SEGMENTATION

In several applications, the sought-for objects exhibit a known colour quite different from that of background structures. Hence, it is reasonable to try segmenting-out foreground from background based on colour information. Let us denote a pixel's colour as:

$$I(p) = \begin{bmatrix} I_r(p) \\ I_g(p) \\ I_b(p) \end{bmatrix}$$

Then, segmentation can be achieved by **computing and thresholding the distance** (e.g., Euclidean) **between each pixel's colour and the reference foreground colour μ**.

For example, in the food industry, people use typically blue because food is unlikely blue.

We assume that we know the color of the object and don't know the color of the background (or the assumption can be the opposite). A pixel color is a vector of 3 color channels: red channel, green channel, blue channel. So, a pixel is a triple, i.e., a 3D vector.

Let's assume we have the reference color that we call μ, pixels of background are different from μ, while pixels of foreground are similar to μ. It's all about computing a distance between the current color and the reference color. Since it is a distance in the 3D space, we can use the Euclidean norm. If the distance is less than the threshold, it is a foreground pixel. Otherwise, it is a background pixel.

$$\forall p \in I: \begin{cases} d(I(p),\mu) \leq T & \rightarrow \quad O(p) = F \\ d(I(p),\mu) > T & \rightarrow \quad O(p) = B \end{cases}$$

$$d(I(p),\mu) = \left((I_r(p)-\mu_r)^2 + \left(I_g(p)-\mu_g\right)^2 + (I_b(p)-\mu_b)^2\right)^{\frac{1}{2}}$$

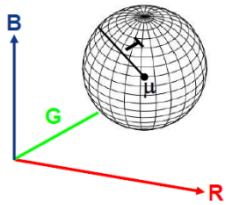
$$\underset{\text{scalar 1x1}}{d(I(p),\mu)} = \underset{\text{1x3}}{(I(p)-\mu)^T} \; \underset{\text{3x1}}{(I(p)-\mu)}^{\frac{1}{2}}$$

5.2.1 – ESTIMATING THE REFERENCE COLOUR

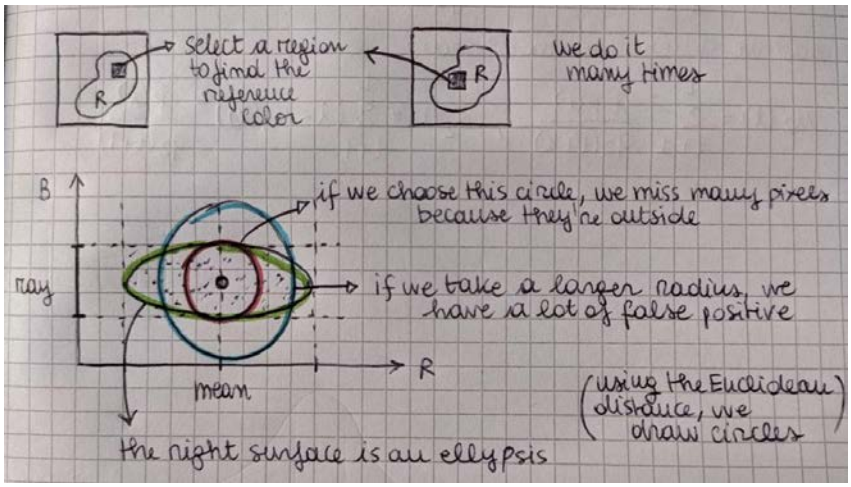
EUCLIDEAN DISTANCE

It is thus necessary to **know the reference colour, μ,** which can be conveniently **estimated ("learned") from one or more training images.**

For example, modelling the colour of a foreground pixel as a multivariate random variable (i.e., a 3D random vector), the reference colour can be taken to be the mean (expected value) over the available training samples ($I(p_k)$, k=1..N):



Accordingly, **segmentation consists in classifying as foreground all pixels lying within a 3D sphere of the RGB colour space centred at μ and having radius as large as T.**



$$\boldsymbol{\mu} = \begin{bmatrix} \mu_r \\ \mu_g \\ \mu_b \end{bmatrix} = \frac{1}{N}\sum_{k=1}^{N} \mathbf{I}(p_k)$$

Unfortunately, there is a problem.
Let's focus on it in 2D.

So, a better approach could be the one of using ellipsis. We can't do that by estimating only the mean, we have to find the variation along the axis and for this we have plenty of variances: it depends on how many dimensions we have.

MAHALANOBIS DISTANCE

A far richer probabilistic characterization of colour distribution among foreground pixels can be obtained estimating **from training samples not only the mean but also the covariance matrix**:

$$\Sigma = \begin{pmatrix} \sigma^2_{rr} & \sigma^2_{rg} & \sigma^2_{rb} \\ \sigma^2_{gr} & \sigma^2_{gg} & \sigma^2_{gb} \\ \sigma^2_{br} & \sigma^2_{bg} & \sigma^2_{bb} \end{pmatrix} \rightarrow \begin{cases} \sigma^2_{ij} = \frac{1}{N}\sum_{k=1}^{N}\left(I_i(p_k)-\mu_i\right)\left(I_j(p_k)-\mu_j\right) \\ i,j \in \{r,g,b\} \end{cases}$$

(note that along the diagonal, we have the individual channel variances)

Recalling that the Euclidean can also be written as:

$$d(I(p),\mu) = \left((I(p)-\mu)^T\,(I(p)-\mu)\right)^{\frac{1}{2}}$$

the Mahalanobis Distance is given by:

$$d_M(I(p),\mu) = \left((I(p)-\mu)^T \sum\nolimits^{-1} (I(p)-\mu)\right)^{\frac{1}{2}}$$

Thanks to this covariance, we can compute the distance with the Mahalanobis distance. The Euclidean distance is a vector product between 3x1 and 1x3 vectors, whereas in the M distance we consider the inverse of the covariance matrix: it is a **covariance weighted distance**.
This allows us exactly to perform the correct segmentation with an ellipsoidal surface.
It is important to note that the covariance is on the diagonal, so we assume that the **diagonal should not be zero**.

To understand the difference between the two distances, it is worth considering the case of diagonal covariance matrix, i.e., independent components in I(p).

Firstly, we computer the inverse, so we take the reciprocal of the values.

$$\Sigma = \begin{pmatrix} \sigma^2_{rr} & 0 & 0 \\ 0 & \sigma^2_{gg} & 0 \\ 0 & 0 & \sigma^2_{bb} \end{pmatrix} \rightarrow \Sigma^{-1} = \begin{pmatrix} 1/\sigma^2_{rr} & 0 & 0 \\ 0 & 1/\sigma^2_{gg} & 0 \\ 0 & 0 & 1/\sigma^2_{bb} \end{pmatrix}$$

Then, we put the values in the formula.

$$d_M\left(\mathbf{I}(p),\boldsymbol{\mu}\right) = \left(\left(\mathbf{I}(p)-\boldsymbol{\mu}\right)^T \Sigma^{-1}\left(\mathbf{I}(p)-\boldsymbol{\mu}\right)\right)^{\frac{1}{2}} \rightarrow \begin{bmatrix} \left(I_r(p)-\mu_r\right)/\sigma^2_{rr} \\ \left(I_g(p)-\mu_g\right)/\sigma^2_{gg} \\ \left(I_b(p)-\mu_b\right)/\sigma^2_{bb} \end{bmatrix}$$

$$d_M\left(\mathbf{I}(p),\boldsymbol{\mu}\right) = \left(\frac{\left(I_r(p)-\mu_r\right)^2}{\sigma^2_{rr}} + \frac{\left(I_g(p)-\mu_g\right)^2}{\sigma^2_{gg}} + \frac{\left(I_b(p)-\mu_b\right)^2}{\sigma^2_{bb}}\right)^{\frac{1}{2}}$$

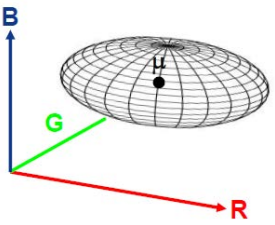
Notice that in the standard Euclidean distance $\sigma = 1$, while in this formula we **normalized the difference of the colors**.
Contrary to the Euclidean distance, the Mahalanobis distance weighs unequally the differences along the components of the random vector, in particular according to inverse proportionality to the learned variances. As such, **the more spread has been learned to be a component, the less the difference along it will contribute to the overall distance**.

This approach has no faults because we can always transform the matrix to have the covariance non zero on the diagonal. **Under any rotation, distance don't change** so we have to pick the reference axis and then the covariance. Even if it is not diagonal, by rotating it becomes diagonal.

Which region of the RGB space determines now the segmentation?

$$d_M(I(p),\mu)^2 = \left(\frac{(I_r(p)-\mu_r)^2}{\sigma_{rr}^2} + \frac{(I_g(p)-\mu_g)^2}{\sigma_{gg}^2} + \frac{(I_b(p)-\mu_b)^2}{\sigma_{bb}^2}\right) \leq T^2$$

We pick a threshold and we find an ellipsoid with axis length given by the 3 variances * the threshold. The higher are the variances, the longer is the corresponding axis of the ellipsoid.



Ellipsoid centred at : $\boldsymbol{\mu} = \begin{bmatrix} \mu_r \\ \mu_g \\ \mu_b \end{bmatrix}$ With axes aligned to coordinate axes and semi-axes lengths given by: $\mathbf{L} = \begin{bmatrix} L_r \\ L_b \\ L_g \end{bmatrix} = T \begin{bmatrix} \sigma_{rr} \\ \sigma_{bb} \\ \sigma_{gg} \end{bmatrix}$

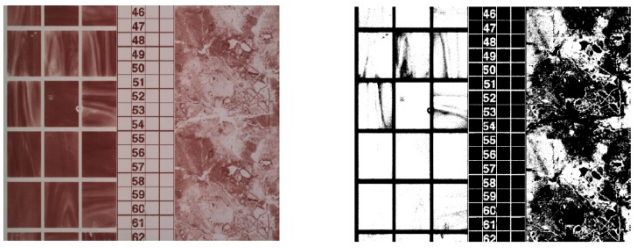
The interpretation of the Mahalanobis distance which has been given in case of diagonal covariance matrix is of general validity. Indeed, **THE COVARIANCE MATRIX CAN ALWAYS BE DIAGONALIZED BY A ROTATION** of the coordinate axes. Thus, in the new coordinate system the Mahalanobis distance is a weighted sum of the contributions along the new axes, with weights being inversely proportional to the variances along the new axes.
This is due to the ***EigenValue Decomposition (EVD)*** of any real and symmetric matrix (Σ):

$$\boldsymbol{\Sigma} = \mathbf{R}\mathbf{D}\mathbf{R}^T: \quad \mathbf{R} = (\mathbf{e}_1\ \mathbf{e}_2\ \mathbf{e}_3), \quad \mathbf{D} = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix}$$

With

- R = a rotation applied to the Data D
- e_i = orthonormal eigenvectors of Σ
- λ_i = corresponding eigenvalues
- R^T = rotation matrix to transform the data into a new coordinate system having axes aligned to the eigenvectors

Upon rotation by R^T the new covariance matrix becomes D, so that the eigenvalues of Σ represent the variances along the direction of the eigenvectors.

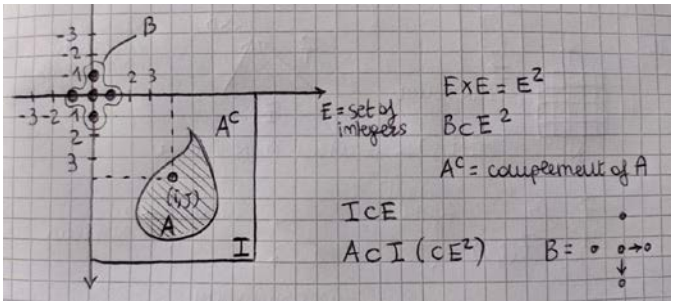


$$\mathbf{I}(p) \rightarrow O(p) = \begin{cases} 255, & d_M(\mathbf{I}(p),\boldsymbol{\mu}) \leq T \\ 0, & d_M(\mathbf{I}(p),\boldsymbol{\mu}) > T \end{cases}$$

6 - BINARY MORPHOLOGY

Let's assume we have background pixels and foreground pixels. The segmentation is never perfect, there are always mistakes: we say that it's fragile.
What kind of mistakes could we find in foreground/background segmentation? Some pixels belonging to the foreground are in reality pixels of the background or, vice versa, we missed some object pixels. So, we have to use some operators to clean up false positive or to recover false negative.

Binary operators are really very powerful, they're not only filters, as the filters for noise we studied previously; they can accomplish relative advanced analysis tests. **Binary operators process SETS**, e.g., the cartesian product between integer numbers.



Within the set I, we denote A as the set of foreground pixels. So, A are the elements in I classified as foreground by whatever kind of segmentation we have previously employed.
Then, there's another set: B. This set is a subset of E^2 and it is called **structuring element (or kernel)**. It is similar to a kernel in the convolution and it is a small type of set.

What are the **Binary Morphology Operators**? They manipulate A or Ac based on B, so they manipulate foreground or background according to the structuring element.
The dots are the elements in the set, so if the central element is the origin,

B= { (0,0), (0,-1), (0,1), (1,0), (-1,0) }

Now, we could process A or the complement of A through this structuring element. We have other examples of structuring elements.

More formally, **Binary Morphology operators are simple though effective tools to improve or analyse binary images**, in particular those achieved by any kind of foreground/background segmentation (e.g., based on intensity, colour, motion estimation, joint deployment of multiple cues). **Binary Morphology operators manipulate SETS** defined over the binary image, which is itself seen as a subset of the discrete plane $I \subset E^2 = E \times E$, with E representing the set of integer numbers and O the origin.

Given I, the set of foreground pixels will be referred to as A, that comprising background pixels as A^C. Binary Morphology operators manipulate either A or A^C through a second set, $B \subset E^2$, known as **STRUCTURING ELEMENT (SE)**.

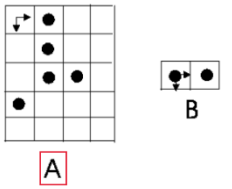
6.1 – DILATION (MINKOWSKY SUM)

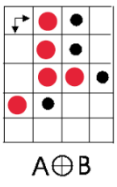
The first operation is called Dilation [pron: Dailescion, = dilatazione, espansione] or Minkowsky Sum. We get the elements just by **adding together the elements of the two sets**.
Now we apply the definition, but only to understand the behaviour, then we'll do it in a faster way.

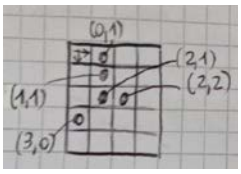
$$A \oplus B = \{c \in E^2 : \ c = a + b, \quad a \in A \ \text{ e } \ b \in B\}$$

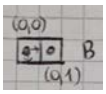
Example

We can notice that if there's the origin, we have the same set A with the addition of a dot on the right of the others dots.



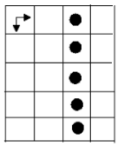


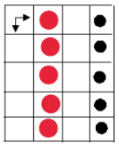




$A = \{(0,1), (1,1), (2,1), (3,0), (2,2)\}$
$B = \{(0,0), (0,1)\}$
$C = A \oplus B \quad = A \cup \{(0,1)+(0,1), (0,1)+(1,1), (0,1)+(2,1), (0,1)+(3,0), (0,1)+(2,2)\}$
$= A \cup \{ \quad (0,2), \quad (1,2), \quad (2,2), \quad (3,1), \quad (2,3)\}$

Example





In this example, we have to translate all the elements in A on the right and then we can put the other dots two columns before.

6.1.1 - PROPERTIES OF DILATION

Traslation A_t of set A by t is defined as: $A_t = \{c \in E^2 : \ c = a + t, \ \ a \in A\}$

It this follows that Dilation can be expressed as the union of the translations of either of the two sets by the elements of the other one:

$$A \oplus B = \bigcup_{b \in B} A_b = \bigcup_{a \in A} B_a$$

Some relevant properties are as follows

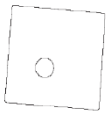
- Dilation is commutative:
  $A \oplus B = B \oplus A$
- Dilation is associative:
  $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
- If the structuring element includes the origin ($\mathcal{O} \in B$) then dilation is *extensive*: the initial set is contained in the dilated set ($A \subseteq A \oplus B$)
- Dilation is an increasing transformation:
  $A \subseteq C \Rightarrow A \oplus B \subseteq C \oplus B$

Similarly to convolution, **associativity** allows dilation by a large structuring element to be decomposed into a chain of operations by smaller elements in order to speed-up execution time, e.g., dilation by a (2n+1)×(2n+1) square can be conveniently accomplished by n successive dilations by a 3×3 square.
Typical structuring elements contain the origin and are symmetric about it, so that dilation expands isotropically foreground regions.
Such operators can be deployed to correct segmentation errors dealing with foreground pixels falsely classified as background, e.g., to connect object's parts or fill holes.

The shape of the structuring element determines that of the dilated foreground objects. In the example on the right, dilation by a circular structuring element results in the outer contour featuring rounded rather than sharp corners. To figure out the dilated shape one may imagine sliding the structuring element so as to traverse all contour points of the original object.

Input binary image *Image dilated by a circular structuring element (radius=11 pixels)*

We can use dilation also to obtain the contours of the binary object. Remember that we are using this **<u>binary edge detector</u>** because we have binary images. If we have grey scale images, we'll use other algorithms.



Contours extracted by dilation followed by subtraction

<u>DON'T USE algorithms for grey scale images on the binary images because is like using a bomb to kill a fly.</u>

Dilation by a 3x3 square followed by subtraction of the original image from the dilated one yields the outer contours of foreground regions, **<u>Outer Contour</u>** meaning here background pixels adjacent to foreground ones.

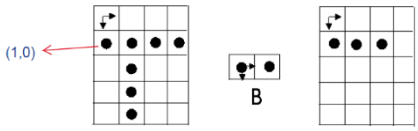
6.2 – EROSION (MINKOWSKY SUBTRACTION)

$$A \ominus B = \{c \in E^2 : \; c + b \in A, \; \forall b \in B\}$$

Example

A = {(1,0), (1,1), (1,2), (1,3), (2,1), (3,1), (4,1)}
B = {(0,0), (0,1)}
A⊕B = A U {(0,1)+(1,0), (0,1)+(1,1), (0,1)+(1,2), (0,1)+(1,3), … } =
= A U {(1,1), (1,2), (1,3), (1,4), … }
Firstly, we remove A. Then, we check every sum:
Is (1,1) in A? Yes, so it's also in A⊖B
Is (1,2) in A? Yes, so it's also in A⊖B

A⊖B = {(1,1),(1,2),(1,3)}



6.2.1 – PROPERTIES OF EROSION

Erosion can be expressed in terms of translations of the structuring element:

$$A \ominus B = \{c \in E^2 : \; B_c \subseteq A\}$$

Erosion involves subtraction of the elements of one set from those of the other:
$A \ominus B = \{c \in E^2 : \forall b \in B \; \exists a \in A : c = a - b\}$

Some relevant properties are as follows

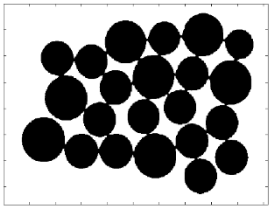
- Erosion is <u>not commutative</u>:
  $A \ominus B \neq B \ominus A$
- If the structuring element can be decomposed in terms of dilations then erosion is associative:
  $A \ominus (B \oplus C) = (A \ominus B) \ominus C$
- If the structuring element includes the origin ($\mathcal{O} \in B$) then erosion is anti-extensive: the eroded set is contained into the original one ($A \ominus B \subseteq A$)
- Erosion is an increasing transformation:
  $A \subseteq C \Rightarrow A \ominus B \subseteq C \ominus B$

Associativity allows erosion by a large structuring element to be decomposed into a chain of operations by smaller elements in order to speed-up execution time. e.g., erosion by a (2n+1)×(2n+1) square can be conveniently accomplished by n successive erosions by a 3×3 square.

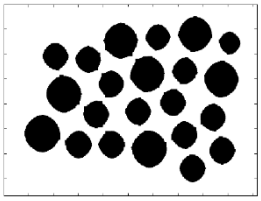
Typical structuring elements contain the origin and are **symmetric** about it, so that erosion shrinks isotropically foreground regions.

Such operators can be deployed to correct segmentation errors dealing with background pixels falsely classified as foreground, e.g., to split wrongly connected objects.

Wrongly connected objects yielded by segmentation

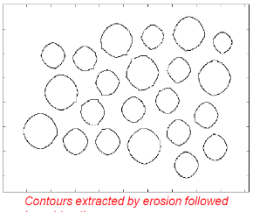


Objects can be split (e.g. to allow counting them correctly) by 5 successive erosions with 3×3 square



If we want to count the number of objects in the image, is difficult because of the fragility of the segmentation.

We have a lot of false positive here. So, we have to separate these objects and it can be accomplished by an erosion.



Contours extracted by erosion followed by subtraction

Erosion by a 3x3 square followed by subtraction of the eroded image from the original one yields the inner contours of foreground regions, **inner contour** meaning here foreground pixels adjacent to background ones.

6.3 – DUALITY BETWEEN DILATION AND EROSION

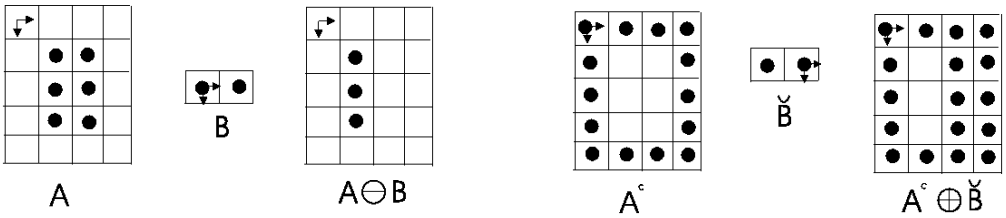
$$(A \oplus B)^c = A^c \ominus \check{B}$$
$$(A \ominus B)^c = A^c \oplus \check{B}$$

Given $\check{B}$ (B breve): $\check{B} = \{\check{b}: \quad \check{b} = -b, \quad b \in B\}$, it can be shown that:

If B is symmetric ($B = \check{B}$):

$$(A \oplus B)^c = A^c \ominus B$$
$$(A \ominus B)^c = A^c \oplus B$$

i.e., **dilation of foreground is equivalent to erosion of background, and vice versa**.

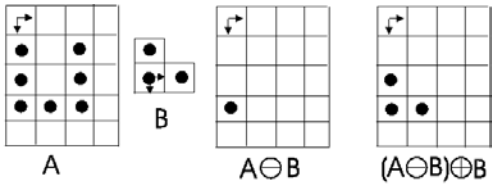


6.4 – OPENING AND CLOSING

Erosion and dilation by the same structuring element can be chained to remove selectively from either foreground or background the parts that do not match exactly the structuring element **without causing any distortion to the other parts**.

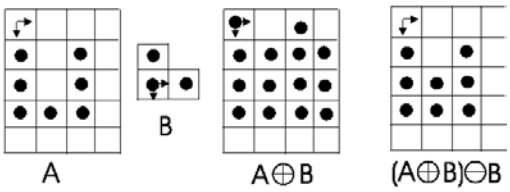
Erosion followed by Dilation = **Opening**

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



Dilation followed by Erosion = **Closing**

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



We want to connect two objects and we do that by a dilation. If we apply this, we create false positives because we change the shape of the object.
Opening is a kind of smart erosion. By opening we can remove according to the shape and it removes from the background. **Closing is a kind of smart dilation.**

In the first example the erosion is only the element on the bottom left. So, we can translate only that element and we get the result on the right.
If I apply a second opening to the first example, anything will change, and it is the same also if we apply other 10 or 1000 openings.
This property is valid also for the closing: applying other closings doesn't change anything.

6.4.1 – PROPERTIES OF OPENING AND CLOSING

Contrary to erosion and dilation, opening and closing are idempotent:

$$(A \circ B) \circ B = A \circ B \qquad (A \bullet B) \bullet B = A \bullet B$$

Opening e closing are not commutative

$$A \circ B \neq B \circ A, \quad A \bullet B \neq B \bullet A$$

because they contain Erosion that is not commutative.

Opening is anti-extensive, closing is extensive:

$$A \circ B \subseteq A, \quad A \bullet B \supseteq A$$

Opening e closing are increasing transformations:

$$A \subseteq C \Rightarrow A \circ B \subseteq C \circ B, \quad A \bullet B \subseteq C \bullet B$$

We said about dilation that it is the union of all the translations of the Structuring Element (SE) at the object points. For erosion we said that is given by the set of points where we can translate the SE and find that it is contained in A. Applying these definitions to the definition of opening, we can say that A opening B is equal to A⊖B translated.
If we call **y** any element in A⊖B, we have the union of By. If **y** belongs to A⊖B, then By must be in A (erosion interpretation), so the **opening is given by the union of the translation such that the translation of the points is in A.** In practice, it splits the shape into all the pieces which are the translation of the SE.

The result of an opening operation can be expressed as the union of those elementary foreground parts that exactly match the structuring element:

$$A \circ B = (A \ominus B) \oplus B = \bigcup_{y \in A \ominus B} B_y = \bigcup_{B_y \subseteq A} B_y$$

Opening can be thus thought of as comparing the structuring element to foreground parts, so as to remove those which turn out different and keep unaltered equal ones.

Duality between erosion and dilation implies duality between opening and closing:

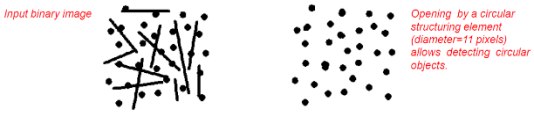
$$(A \circ B)^c = [(A \ominus B) \oplus B]^c = (A \ominus B)^c \ominus \check{B} = \left(A^c \oplus \check{B}\right) \ominus \check{B} = A^c \bullet \check{B}$$
$$(A \bullet B)^c = [(A \oplus B) \ominus B]^c = (A \oplus B)^c \oplus \check{B} = \left(A^c \ominus \check{B}\right) \oplus \check{B} = A^c \circ \check{B}$$

If B is symmetric $\left(B = \check{B}\right)$: $(A \circ B)^c = A^c \bullet B$, $(A \bullet B)^c = A^c \circ B$

Because of duality, closing can be thought of as comparing the (flipped) structuring element to background parts, so as to remove (i.e. change to foreground) those which turn out different and keep unaltered equal ones.

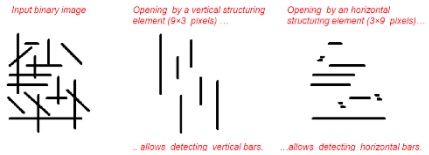
We can use duality to have an intuitively interpretation of closing. We apply the definition and we apply duality between dilation and erosion and we get the elements in the square.
A similar reasoning can be done to see that A closing B complemented is just the complement of A opening $\check{B}$.
Up to a complement, dilation and erosion just do the same but one is going to work on the object and the other on the background. If B is symmetric, then we get that type of duality: A opening B complemented is the complement of A opening B.
So, in practice, **<u>if we just implement dilation, we have every other operation</u>**.
Due to duality, closing is like opening the background, so it is going to include the background parts not according to the SE. We remove from the background parts and we include them into the object.

Example



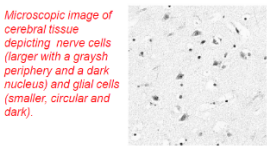
Starting from the first figure, we want to **detect all circles**: can we use a simple morphological processing?
Opening is perfect in this case because we throw away points that don't match the SE.
The diameter of the circles is larger than the width of these bars, so we can open the circle of the objects and we throw away all elements not covered by the circle. Pay attention because the diameter of the point MUST be larger than the width of the bars or it wouldn't work.

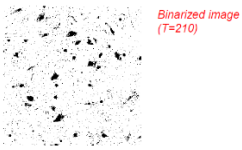
Example



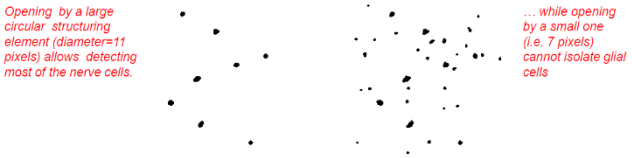
In the first figure, here we have vertical, horizontal and diagonal bars. In order to detect only vertical lines, we can use opening by vertical SE.
The same can be done to detect horizontal bars: we use opening by horizontal SE.

Example





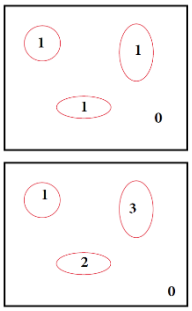
HIV cells on cerebral tissue: doctors have to count the amount of neurons. The large cells with dark nucleus are neurons, while other smaller cells are glial and not neurons.
In order to detect only neurons we can firstly binarize the image with a threshold = 210. Then, we can apply segmentation, but it is ALWAYS fragile because it has errors.
So, we have to apply also opening: if we apply a very small SE, probably we will kill all the noisy spots. If we enlarge the SE more and more, we can throw away everything but neurons.
Note that we cannot isolate ONLY glial cells because they're smaller than neurons, so they will be always together with neurons.



7 - BLOB ANALYSIS

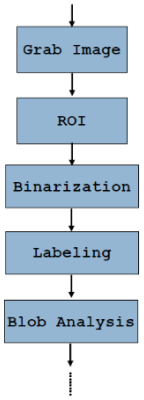
Once foreground/background segmentation has been accomplished, in most applications, the next task deals with analysis of the individual foreground objects to achieve some kind of high-level knowledge on the observed scene.
Such knowledge may pertain detecting what types of objects are contained in the scene, measuring their position and orientation, assessing whether objects show or not manufacturing defects or inaccuracies and so on.
Accordingly, individual objects, usually referred to as **BLOBS (Binary Large Objects)** or **regions**, first need to be isolated within the overall foreground region. As already pointed out, this step is known as **connected components**.

Then, individual objects can be processed to extract specific features related to the required kind of high-level knowledge. For example, several features may be computed to determine the **shape** of objects, so as to detect whether one or more specific types of objects to be picked-up by a robot are present in the observed scene. For the robot to pick detected objects, other features related to their **position and orientation** need typically to be computed.



Let's assume we have an application with a robot guidance and a binary image. The robot has to pick up objects. The first image is not good enough because we want to see pixels belonging to the different objects. We just divided foreground and background.
So, if the labels assigned to a pixel are all different one in relation to the other, we can specify which object the robot has to pick up, like in the second image.

The process whereby features are extracted from blobs is often referred to as **BLOB ANALYSIS**. Features may be computer either from all the pixels belonging to the blob (**region features**) or from boundary pixels only (**boundary or contour features**). A plethora of diverse features has been proposed in literature: we will study the most widely used.
Shape-related features need to exhibit **invariance to the transformations** the image of the object may undergo in the addressed settings. Mostly, shape-features are required to fulfil invariance to a **similarity transformation (translation, rotation and scale change)**, i.e., need not to change if the object appear at a different position or rotated or show a different size in the image.



Object detection based on blob features can be accomplished either according to heuristic rules or deploying pattern recognition techniques, such as is particular **machine learning techniques**. With the latter approach, a classification function to map a feature vector into a set of object types is learned from training sample.
In the figure we have the process of a Text Analysis example. **ROI** means Region on Interest.

7.1 – DISTANCES AND CONNECTIVITY

To discuss algorithms aimed at labeling the connected components of a binary image, first we need to introduce the **notion of connectivity**, which in turn is related to that of **distance on the discrete plane, E^2.**

E^2 is the set of pairs, it is the domain of the binary image (background and foreground).
The third definition is the so-called **Triangle Equality**.

Given $p_1(i_1, j_1), p_2(i_2, j_2), p_3(i_3, j_3) \in E^2$, D is a distance iff:

$$\begin{aligned} D(p_1, p_2) &\geq 0,\ D(p_1, p_2) = 0 \Leftrightarrow p_1 = p_2 \\ D(p_1, p_2) &= D(p_2, p_1) \\ D(p_1, p_3) &\leq D(p_1, p_2) + D(p_2, p_3) \end{aligned}$$

The two main discrete distances defined in E^2 are defined in the following.

7.1.1 – CITY-BLOCK or MANHATTAN DISTANCE

The city-block distance, D_4, between p_1 e p_2 is given by:

$$D_4(p_1, p_2) = |i_1 - i_2| + |j_1 - j_2|$$

Manhattan Distance (D4)

According to the above definition, a pixel can be reached from another through horizontal or vertical shifts only.

7.1.2 – NEIGHBOURHOOD DISTANCE

We can also measure the distance with the set of neighbours. The set with D=1 from p is always a rhombus but each p has only 4 neighbours and this set of neighbours is called **neighbourhood**.

The set of points having distance $\leq r$ from a given one is a rhombus with diagonals of length $2r + 1$, e.g with $r = 2$:

$$\begin{array}{ccccc} & & 2 & & \\ & 2 & 1 & 2 & \\ 2 & 1 & 0 & 1 & 2 \\ & 2 & 1 & 2 & \\ & & 2 & & \end{array}$$

Defined as *neighbours of p* the set of points having $D = 1$ from p, it follows that, using $D = D_4$, such a set is given by:

$$\begin{array}{ccc} & n & \\ n & p & n \\ & n & \end{array}$$

and is usually called 4-neighbourhood of p (hereinafter also denoted as $n_4(p)$).

7.1.3 – CHESSBOARD or HEIGHT DISTANCE

The third distance we consider is the chessboard distance or height distance. We can move to points in three ways: horizontal, vertical and **DIAGONAL**. We can appreciate that the set of points is now a square and not a rhombus.
We can define also a neighbourhood if the distance D8=1: here we have 8 neighbours.

The chessboard distance, D_8, between p_1 and p_2 is given by:

$$D_8(p_1, p_2) = max\left(|i_1 - i_2|, |j_1 - j_2|\right)$$

According to this distance, horizontal or vertical and diagonal shifts have the same weight.

The set of points having distance $\leq r$ from a given one is a square with side of length $2r+1$, e.g with $r=2$:

$$\begin{matrix} 2 & 2 & 2 & 2 & 2 \\ 2 & 1 & 1 & 1 & 2 \\ 2 & 1 & 0 & 1 & 2 \\ 2 & 1 & 1 & 1 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{matrix}$$

The set of neighbours of p such as $D_8 = 1$ is called 8-neighbourhood of p (hereinafter also denoted as $n_8(p)$):

$$\begin{matrix} n & n & n \\ n & p & n \\ n & n & n \end{matrix}$$

7.2 – CONNECTED COMPONENTS OF A BINARY IMAGE

A path of length n from pixel p to pixel q is a sequence of pixels $p = p_1, p_2, \ldots p_n = q$ such as p_i and p_{i+1} are neighbours according to the chosen distance. Both $D_4(p,q)$ and $D_8(p,q)$ have the same length as the smallest path between p and q.

We can have a 4 path or an 8 path. **R is a region** and it said to be **connected** if, for any 2 points belonging to the region, the path is between p and q. We can go from a point to another without go out the region.

A set of pixels, R, is said to be a *connected* region if for any two pixels p, q in R there exists a path contained in R between p and q. Again, depending on the chosen distance being either D_4 or D_8, R is said to be a *4-connected* or *8-connected* region respectively.

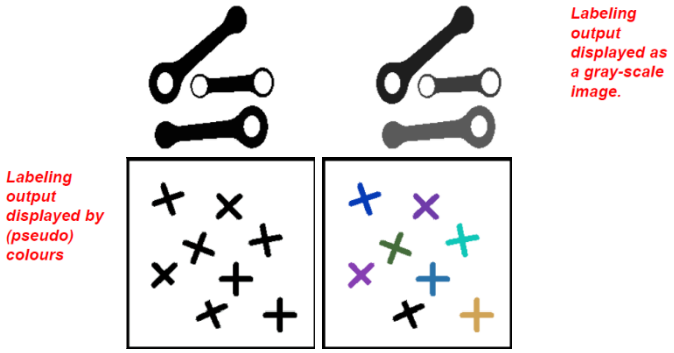
A **set of pixels** is said to be a **connected foreground/background** if it is a connected region and it contains all foreground/background pixels only.

A set of pixels is said to be a *connected* foreground (background) region if it is a connected region and includes foreground (background) pixels only.

A connected component of a binary image is a maximal connected foreground region. Since it has to be maximal, if we have another pixel, we use connectivity.

7.2.1 – CONNECTED COMPONENTS LABELING

Computation whereby pixels belonging to different connected components are given different **labels**, while background pixels are left unaffected (e.g., they may keep the previous label or, equivalently, a new one).



7.3 – ALGORITHMS FOR CONNECTED COMPONENTS LABELING

7.3.1 - THE CLASSICAL 2-SCANS ALGORITHM

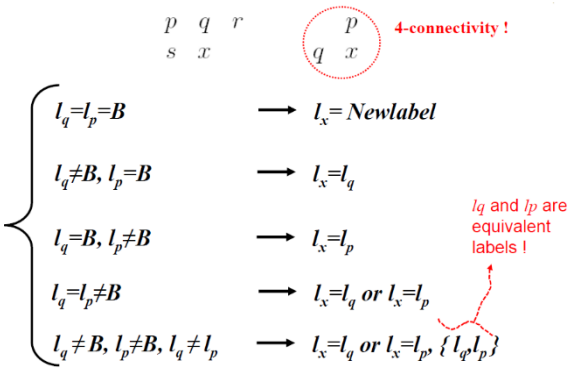
By the **first scan**, foreground pixels take temporary labels based on those given to already visited neighbours, which depend on both the chosen distance, e.g., D4, and the scan order, e.g., left-right, top-down.
Upon the first scan, different blobs have certainly been given different labels, though, depending on shape, this may be the case for connected parts of a single blob too.
In the first scan we can have several mistakes: for example, it may happen that different blobs have been given to different labels. So, the second scan is about assigning the parts belonging to the same blob that have been given different temporary labels.

Hence, the **second scan** allows a unique final label to be assigned to those parts belonging to the same blob that have been given different temporary labels by the first scan.
Purposely, **equivalent temporary labels need to be found between the two scans, so as to assign a unique final label to each of the equivalence classes among temporary labels**.

Example

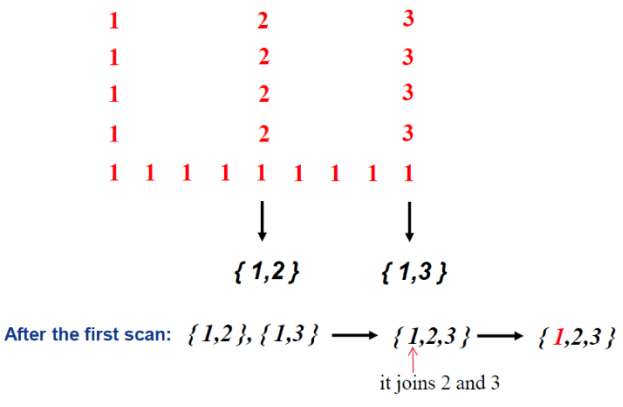
We scan the image from left to right and from top to bottom. Let's assume that the label given to q is the same given to p and it is = background. Which label will be assigned to x?

The label given to x is a new label: is x connected to q? Yes, so it has to have the same label as q. Is it connected to p? Yes, so it has to have the same label as p. So, these labels are all the same.



Now we have a problem: x is a neighbour of p, but it also neighbour of q, so x is a pixel which joins together 2 parts of the same object, so it doesn't matter.
The important thing is to store somewhere the information that these 2 labels are equivalent because they're temporary labels assigned to the same component.

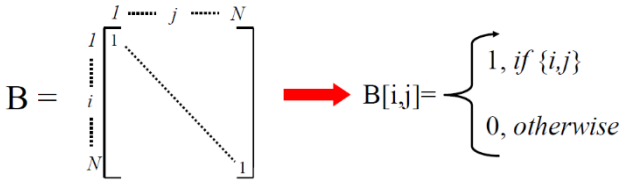
```
// lp,lq,lx: labels assigned to p,q,x
// B:background, F:foreground.
// FIRST SCAN:
for(i=1; i<NROWS-1; i++)
for(j=1; j<NCOLS-1; j++) {
if (I[i,j]==F) {
lp = I[i-1,j];
lq = I[i,j-1];
if(lp == B && lq == B) {
NewLabel++;
lx = NewLabel;}
else if((lp != lq)&&(lp != B)&&(lq != B)){
// REGISTER EQUIVALENCE (lp,lq)
lx = lq;}
else if(lq != B) lx = lq;
else if(lp != B) lx = lp;
I[i,j] = lx;} }
// FIND EQUIVALENCE CLASSES
// SECOND SCAN
```



HANDLING EQUIVALENCES

STEP 1

During the first scan the equivalences found between temporary label pairs are recorded into an N×N matrix, where N = number of temporary labels:



"if {i,j}" means "if i has been found with the equivalence to j"

Because label equivalence is an equivalence relation:

i = j → B[i, j] = 1 **B[i, j] = B[j, i]** **∀i,j = 1..N** → B is symmetric

STEP 2

After the first scan, B is processed so as to elicit the equivalences among temporary labels implied by virtue of the transitive property:

$$\{i,j\} \rightarrow \{i,k\} \ \forall \{k,j\}$$

$$B[i,k] = B[i,k] \ OR \ B[j,k] \ \forall k = 1...N$$

```
for (j=1,....N)
 for (i=1....N)
  if (B[i,j]=1)AND(i≠j)
   for(k=1....N)
     B[i,k]=B[i,k] OR B[j,k];
```

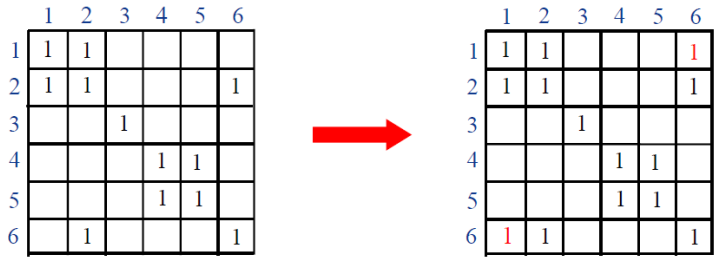
Thus, through a scan by columns the equivalences implied by the transitive property can be recorded into B:

Example

If we consider rows, we can look at the 1 in the columns:

- 1st column: 1=1 and **1=2**
- 2nd column: **2=1**, 2=2 and **2=6**
- 3rd column: 3=3
- 4th column: 4=4 and **4=5**
- 5th column: **5=4** and 5=5
- 6th column: **6=2** and 6=6

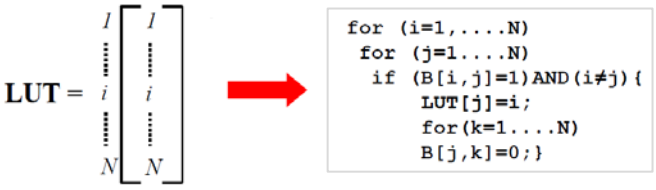
So, we found the following equivalences during the first scan: {1,2}, {4,5}, {2,6}.
Since the **transitive property** is valid, {1,2} and {2,6} implies that {1,6}, so we add the red 1.



After the computation, B contains the information related to equivalence classes, a row recording a "1" in each column associated to a label belonging to the same equivalence class as the row.

STEP 3

Having found equivalence classes, it is now necessary to assign a unique final label to each of them. To this purpose, a simple table initialized by the identity function may be conveniently deployed:



We can just put 0 in the corresponding rows because we don't need to process them again and we assign to the image I the LUT (Look Up Table).

STEP 4

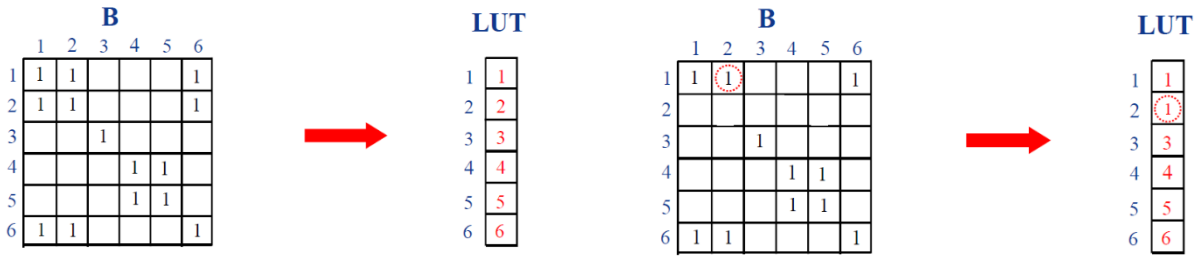
The second scan boils down to an image look-up operation by the table:

```
for (i=1....NROWS-1)
 for (j=1....NCOLS-1)
     I[i,j]=LUT[I[i,j]];
```

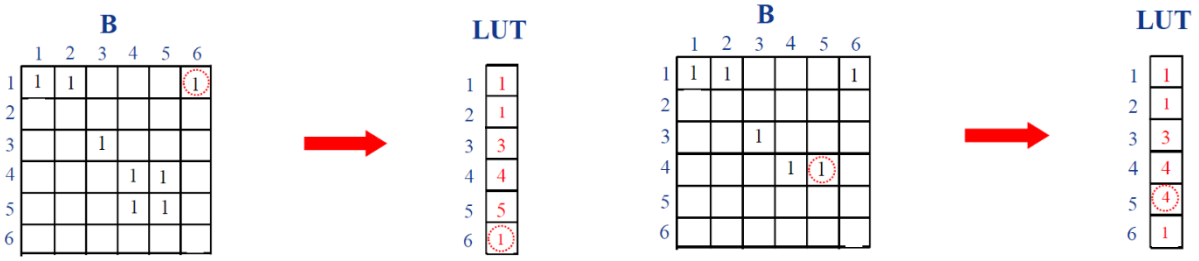
Example

The LUT is initialized with the identity.

Then, we scan B: we pick 1 to represent 2 and we write 1 in the entry 2.



Then, we do the same with 1 that represents 6 and with 4 that represents 5.



Further straightforward manipulation of the LUT before the second scan would allow final labels to be consecutive numbers (i.e., 1,2,3…)

7.3.2 – HANDLING EQUIVALENCES DURING THE FIRST SCAN

A simple and **efficient variation of the classical two-scan algorithm**[1] allows handling equivalences directly during the first scan.

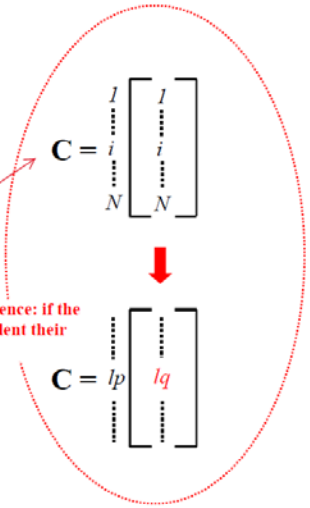
Efficiency comes from equivalence classes being always up-to-date during the first scan, **so that newly detected equivalences** already implied by virtue of the transitive property **need not be handled**. Indeed, checking for equivalences occurs in the equivalence rather than label domain: one thus needs

[1] **Di Stefano**, Bulgarelli "A Simple and Efficient Connected Components Labeling Algorithm", ICIAP 1999

to do nothing whenever two conflicting labels are already known to belong to the same equivalence class.

Handling of equivalences relies on a simple **1D array** (referred to as Class *Array*), which maps each label onto its equivalence class, and is then used as a LUT to carry out the second scan.

```
// lp,lq,lx: labels assigned to p,q,x
// B:background, F:foreground.
// FIRST SCAN:
for(i=1; i<NROWS-1; i++)
for(j=1; j<NCOLS-1; j++) {
if (I[i,j]==F) {
lp = I[i-1,j];
lq = I[i,j-1];
if(lp == B && lq == B) {
NewLabel++;
lx = NewLabel;}
else if((lp != lq)&&(lp != B)&&(lq != B)){
// REGISTER EQUIVALENCE (lp,lq)
lx = lq;}
else if(lq != B) lx = lq;
else if(lp != B) lx = lp;
I[i,j] = lx;} }
// FIND EQUIVALENCE CLASSES
// SECOND SCAN
```



Handling the equivalence: if the two *labels* are equivalent their classes are merged

IMPLEMENTATION

Class merging: if *lp* is equivalent to some other labels, all such labels should be marked as equivalent to *lq* (transitive property). The real advantage is that in this way we handle much less conflicts, we have to check less equivalences.

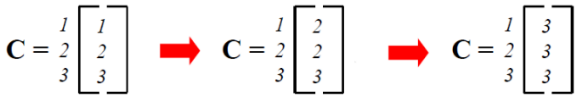
```
for (k=1;k<=NewLabel;k++)
if (C[k]==lp) C[k]=lq;
```

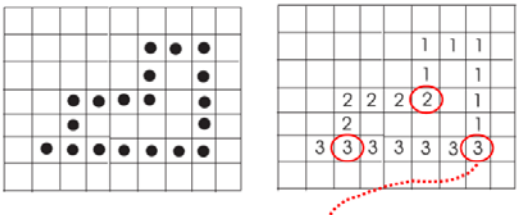
Checking whether two labels are equivalent takes place **within the class rather than label domain**, so as to avoid wasting computation to handle equivalences already implied by previous ones due to the transitive property.

```
if((C[lp]!=C[lq])&&(lp!=B)&&(lq!=B))
{C_lp = C[lp];
for(k=1; k<=Newlabel; k++)
if (C[k] == C_lp) C[k]=C[lq];
}
```

Example

Initially, the array is equal to the identity: 1, 2, 3.
We find the equivalences: {1,2}, {2,3} and {1,3}.

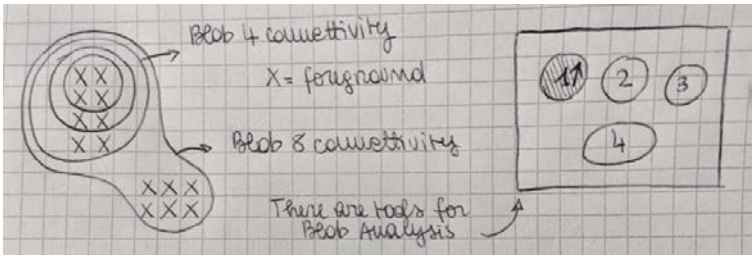


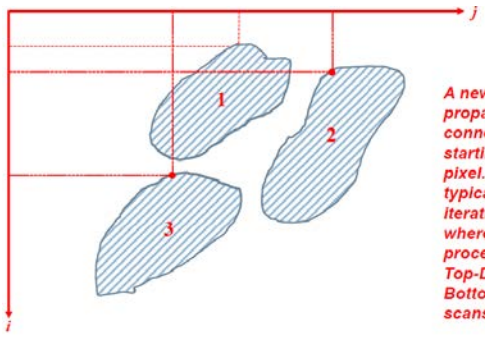


Not handled because it is already implied by previous equivalences !

7.3.3 – FLOOD-FILL APPROACH

This approach is very simple to implement but **notably less efficient** due to the **image being scanned several times** rather than just two.





A new label is propagated throughout a connected component starting from a "seed" pixel. Propagation is typically carried out by a iterative procedure, whereby the image is processed alternating Top-Down Left-Right and Bottom-Up Right-Left scans.

Outer loop to detect the seed pixels of the connected components

Iterative label propagation within each connected component

7.4 – FEATURES TO COMPUTE

As already pointed out, once regions have been identified, usually the next step consists in computing features associated to each region. In the following we will introduce several feature types.

7.4.1 – AREA AND BARYCENTRE

The simplest features to extract are the area (A) and the barycentre (B).
The **area A** just amounts to the **number of pixels belonging to the region**, while the **barycentre p** is the **centre of mass of the region**, which is seen as a **set of particles of unitary mass**.

$$A = \sum_{p \in R} 1 \qquad p = \begin{bmatrix} i \\ j \end{bmatrix} \rightarrow B = \begin{bmatrix} i_b \\ j_b \end{bmatrix} : \; i_b = \frac{1}{A}\sum_{p \in R} i; \; j_b = \frac{1}{A}\sum_{p \in R} j$$

7.4.2 – PERIMETER

The perimeter is defined as **the length of the contour of the region**. To measure such a length, we need to define first which pixels of the region belong to its contour.
A pixel, p, belonging to a region is said to **belong to the contour of the region if there exists at least one background pixel, q, between its neighbours**. Accordingly, we end up with defining a different contour (C4 or C8) based on the adopted neighbourhood (4- or 8-neighbourhood).

This definition says something interesting: **the notion of neighbourhood is related to distance** and **the distance depends on the connectivity** we're using.

$$\exists q \in n_4(p) \rightarrow p \in C_4$$
$$\exists q \in n_8(p) \rightarrow p \in C_8$$

It can be easily observed that C_4 is a 8-connected curve while C_8 is 4-connected.

$$P_8 = \sum_{p \in C_4} 1$$

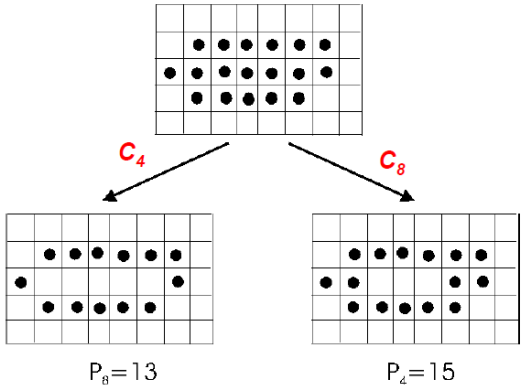
Length of the 8 connected contour C_4

$$P_4 = \sum_{p \in C_8} 1$$

Length of the 4 connected contour C_8

P8 tends to underestimate the "true" length, P4 to overestimate it: this is a topological paradox. Given the contour, we can count how many pixels are found. Looking at the first picture, we know that it is a blob.
Let's find C4: We have to find all those dots which have a background in the 4 neighbourhood. The length of the curve is equal to the perimeter, so it is 13 pixels.

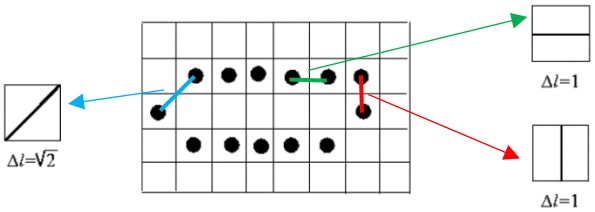
Now let's find C8: we need to find all those dots which have got a background in the 8 neighbourhood.
Here we have that the perimeter = 15 pixels.



So that one might wish to **<u>average the two measurements when aiming at a more accurate approximation:</u>**

$$\widetilde{P} = \frac{P_4 + P_4}{2}$$

Considering C4 (i.e., the 8-connected contour), a more accurate approximation of the perimeter can be obtained by taking into account whether the "ideal" curve would better join two nearby pixels through a horizontal/vertical segment or a diagonal one:



When computing the final perimeter, pay attention to the diagonal segments because they're not equal to 1. Once a scan order around the contour has been defined:

$$C_4 = \{p_1, p_2 \dots p_m\}$$

$$\widetilde{P}_8 = \sum_{p_k:\, p_{k+1} \in n_4(p_k)} 1 + \sum_{p_k:\, p_{k+1} \in (n_8(p_k) - n_4(p_k))} \sqrt{2}$$

7.4.3 – COMPACTNESS (aka FORM FACTOR)

Some other descriptors that we may want to compute are **shape features**. Shape features are scale, rotation and translation-invariant measurements.
(Remember that the scale is the size of an object in the image and not the real size of the object. Indeed, it depends on the size of the object and the distance of the object from the camera.)
<u>Compactness or Form Factor</u> is the **ratio between the perimeter and the area of a shape**.

For example, it could be the squared perimeter divided by area. In case of continuous 2D shapes it takes the minimum value (i.e., 4π) for a circle. It is a **dimensionless quantity**, so it is scale-invariant.

$$C = \frac{P^2}{A}$$

Other examples are the square with side of length l (a) or the isosceles right triangle (b):

$$(a) \quad C = \frac{P^2}{A} = \frac{(4l)^2}{l^2} = 16$$

$$(b) \quad C = \frac{P^2}{A} = \frac{\left(l(2+\sqrt{2})\right)^2}{l^2/2} \cong 23.3$$

7.4.4 – HARALICK'S CIRCULARITY

If the compactness is minimum for circles, this is not true in case of the digital/discrete domain. There are other digital shapes that could be used, such as the octagon/diamond.
Let's consider the contour: we have ***m*** pixels belonging to the contour. We can compute the barycentre of the contour and we can use the Euclidean distance between each pixel and the barycentre. If this is a circle, all these distances are going to be the same.
For a digital shape this is false due to errors, but they still should be quite similar relative to each other.

So, if we want a shape feature that is going to capture how spread is the distribution of distance, what kind of parameter can we use? We could consider the **inverse of the standard deviation**.

In the discrete domain, compactness takes its smallest value not for a circle but for an **octagon** o diamond (depending on whether the 8-connectivity or 4-connectivity is employed to calculate the perimeter). Therefore, Haralick has proposed the following circularity feature:

$$C = \{p_1, p_2, \dots, p_m\} \qquad p_k = \begin{bmatrix} i_k \\ j_k \end{bmatrix} \qquad B = \begin{bmatrix} i_b \\ j_b \end{bmatrix} \qquad d_k = \sqrt{(i_k - i_b)^2 + (j_k - j_b)^2}$$

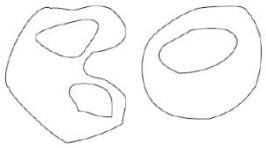
where dk = distance from a contour point to the barycentre (in a circle, all such distances are = to the radius)

Again, a **scale invariant feature** due to the **ratio between the mean distance and the standard deviation of distances** being dimensionless. The smaller the standard deviation (i.e., distances are more similar one to another) the higher the circularity.
So, the circularity tells us **how much the points are close to the barycentre**.

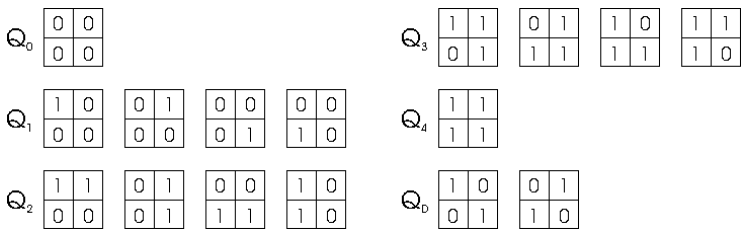
$$\mu_R = \frac{1}{m}\sum_{k=1}^{m} d_k \qquad \sigma_R^2 = \frac{1}{m}\sum_{k=1}^{m} (d_k - \mu_R)^2 \qquad \rightarrow \qquad \tilde{C} = \frac{\mu_R}{\sigma_R}$$

7.4.5 – EULER NUMBER

The Euler number of a binary image is defined as ***E = C – H,*** where C is the number of connected components and H the number of holes. It represents a **topological feature** and it is invariant to the so-called **rubber sheet transformations**.
e.g., in the picture we have C=2 and H=3, so E = 2 - 3 = -1.
In practice, E is computed for each connected component, so as to determine the number of its holes. **If E=0, every object has a single hole.**



The Euler number can be computed by matching across the image a set of 2x2 binary pattern referred to as **Bit Quads**:



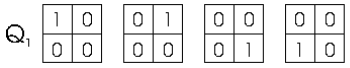
Practically, we consider all the configurations and we count how many Bit Quads of different type are in the binary image.
The Euler Number can be computed in 2 different ways depending on the neighbourhood we use.

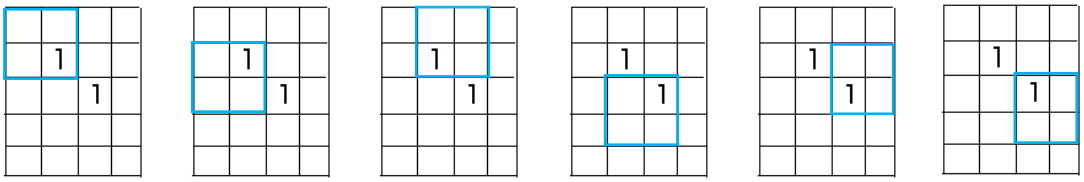
It can be shown that:

$$E_4 = \frac{1}{4}[nQ_1 - nQ_3 + 2nQ_D] \qquad E_8 = \frac{1}{4}[nQ_1 - nQ_3 - 2nQ_D]$$

Example

If we check the Q1:





We obtain that: Q1=6 Q3=0 QD=1

So,

$$E_4 = \frac{1}{4}[6 + 2 * 1] = \frac{1}{4} * 8 = 2 \qquad E_8 = \frac{1}{4}[6 - 0 - 2 * 1] = \frac{1}{4} * 4 = 1$$

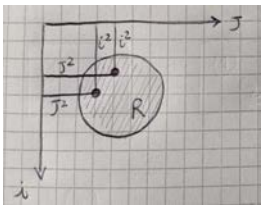
7.4.6 – MOMENTS

The moment of order (m,n) of a region is defined as:

$$M_{m,n} = \sum_{p \in R} i^m j^n \qquad m \geq 0, n \geq 0$$

The Area is the moment of order (0,0):

$$M_{0,0} = \sum_{p \in R} i^0 j^0 = \sum_{p \in R} 1 = A$$



Other relevant ones are the second moments (m+n=2):

$M_{0,2} = \sum_{p \in R} j^2$ → sum of the squared j coordinate: it is the moment of inertia wrt the i axis

$M_{2,0} = \sum_{p \in R} i^2$ → sum of the squared i coordinate: it is the moment of inertia wrt the j axis

$M_{1,1} = \sum_{p \in R} ij$ → deviation moment of inertia

INVARIANCE TO TRANSLATION AND SCALING

The previously defined moments change according to the position of the region in the image. **Invariance to translation** can be achieved by simply calculating the moments relative to the barycentre:

$$M'_{m,n} = \sum_{p \in R} (i - i_b)^m (j - j_b)^n$$

with such moments being usually referred to as **central moments**.

To achieve also **invariance to scaling**, central moments need to be **normalized**:

$$V_{m,n} = \frac{M'_{m,n}}{A^\alpha} \qquad with \qquad \alpha = \frac{m+n}{2} + 1$$

HU'S MOMENTS

Hu has shown that **shape features invariant to rotation, translation and scaling can be defined based on normalized central moments**. Such invariant features, usually referred to as Hu's moments, are shown on below.

Hu proved invariance for continuous shapes. Nonetheless, when computed on blobs the quantities below turn out reasonably stable across similarities.

$$h_1 = V_{20} + V_{02}$$

$$h_2 = (V_{20} - V_{02})^2 + 4V_{11}^2$$

$$h_3 = (V_{30} - 3V_{12})^2 + (V_{03} - 3V_{21})^2$$

$$h_4 = (V_{30} + V_{12})^2 + (V_{03} + V_{21})^2$$

$$h_5 = (V_{30} - 3V_{12})(V_{30} + V_{12})\left((V_{30} + V_{12})^2 - 3(V_{03} + V_{21})^2\right) + (3V_{21} - V_{03})(V_{03} + V_{21})\left(3(V_{30} + V_{12})^2 - (V_{03} + V_{21})^2\right)$$

$$h_6 = (V_{20} - V_{02})\left((V_{30} + V_{12})^2 - (V_{03} + V_{21})^2\right) + 4V_{11}(V_{30} + V_{12})(V_{03} + V_{21})$$

$$h_7 = (3V_{21} - V_{03})(V_{30} + V_{12})\left((V_{30} + V_{12})^2 - 3(V_{03} + V_{21})^2\right) + (3V_{12} - V_{30})(V_{03} + V_{21})\left(3(V_{30} + V_{12})^2 - (V_{03} + V_{21})^2\right)$$

7.4.7 – ORIENTATION

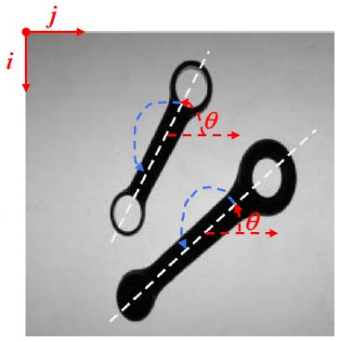
Many applications, such as robot guidance, require to determine the orientation of the objects appearing in the image.

If an object is somewhat elongated, its orientation can be defined according to the direction of the axis of **least inertia**, i.e., the **line through the barycentre of least moment of inertia**, which is often referred to in the CV literature as **major axis**:

$$major\ axis = arg\min_{l}\left(\sum_{p\in R} d_l^2(p)\right)$$

Accordingly, the orientation is typically determined as the angle, θ, between the major axis and the horizontal axis (i.e., the j axis with the adopted n otation).

As the major axis is a line, the thus defined object orientation can be determined modulo π, i.e., both θ and θ+π represent the same orientation.



We studied that from Blobs we can extract perimeter, area, shape features, normalized moments, etc.

Another type of information we can extract from blobs is the orientation of the objects: this is important because if a robot has to pick objects, it has to know how the objects are oriented.

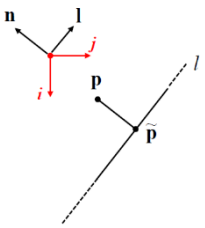
If the blob is kind of a circle, there's no orientation: orientation starts being relevant when object has a certain direction.

These objects here show a certain elongatedness and show their orientation. How could we proceed following this intuition? **We capture the inner orientation of a Blob using the major axis which is a line through the barycentre which has the least inertia. There's no mass, we only determine inertia using the distance of the points.**

DISTANCE FROM A POINT TO A LINE

Given the line $l: aj + bi + c = 0$, the **squared distance** from a point p to the line can be expressed as:

$$d_l^2(p) = \frac{(aj + bi + c)^2}{a^2 + b^2} \qquad with \qquad p = \begin{bmatrix} i \\ j \end{bmatrix}$$



If we take:

- $\tilde{p}$ = perpendicular line as the intersection between the line and the point
- l = vector parallel to the line
- n = vector perpendicular to the line, given by the dot product
- ω = angle between n and $\tilde{p}$

In the figure, ω = 0 because p and $\tilde{p}$ are parallel, so the cosine = ±1 (-1 if the vector has the other direction).

$$\tilde{p} = \begin{bmatrix} \tilde{\imath} \\ \tilde{\jmath} \end{bmatrix} \qquad l = \begin{bmatrix} a \\ -b \end{bmatrix} \qquad n = \begin{bmatrix} b \\ a \end{bmatrix}$$

$$n(p - \tilde{p}) = |n||p - \tilde{p}|cos\omega \qquad with \qquad cos\omega = \pm 1$$

We can expand the computation using the norm of the vector.

$$aj + bi + a\tilde{\jmath} - b\tilde{\imath} = aj + bi + c = 0 \qquad \rightarrow \qquad -a\tilde{\jmath} - b\tilde{\imath} = c$$

$$d_l(p) = \frac{n\,(p - \tilde{p})}{|n|cos\omega} = \frac{b(i - \tilde{\imath}) + a(j - \tilde{\jmath})}{\sqrt{a^2 + b^2}\,cos\omega} = \frac{\boldsymbol{aj + bi + c}}{\sqrt{\boldsymbol{a^2 + b^2}}\,cos\omega} \qquad \rightarrow \qquad d_l^2 = \frac{(aj + bi + c)^2}{a^2 + b^2}$$

We obtain a number whose absolute value is the distance between the point and the line, but this number could be also negative because it underlines the orientation (remember the value of the cosine).

The formula in red is the **signed "distance"**, in which the sign depends on p lying on either of the two sides wrt the line.

LINE THROUGH THE BARYCENTRE OF THE OBJECT

Let's consider now a new reference system which has its origin in the barycentre B, a generic line and a θ which defines the orientation of the line.

$$\mathbf{p} = \begin{bmatrix} i \\ j \end{bmatrix}, \mathbf{B} = \begin{bmatrix} i_b \\ j_b \end{bmatrix}, \overline{\mathbf{p}} = \begin{bmatrix} \bar{i} \\ \bar{j} \end{bmatrix} = \begin{bmatrix} i - i_b \\ j - j_b \end{bmatrix}, \bar{\mathbf{l}} = \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Any point along the line is given by a parameter λ times l. We used this parametric equation also for vanishing points.

$$\bar{l}: \overline{\mathbf{p}} = \lambda\bar{\mathbf{l}} \rightarrow \begin{cases} \bar{\imath} = \lambda\alpha \\ \bar{\jmath} = \lambda\beta \end{cases} \rightarrow \frac{\bar{\jmath}}{\bar{\imath}} = \frac{\beta}{\alpha} \rightarrow \alpha\bar{\jmath} - \beta\bar{\imath} = 0$$

Since we aim to find the orientation, but to compute the orientation we need to find the distance of each pixel from the line, we are considering all pixels of the object.

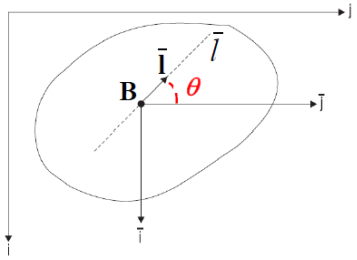
Moreover, we are considering the parametrized line because we want to find the expression of the **moment of inertia** wrt a generic line, then we want to minimize this quantity.

Finally, we have to write the distance of a point to the line and for writing it, we can use a simplification.

$$d_l^2(p) = \frac{(a\bar{j} + b\bar{i} + c)^2}{a^2 + b^2} : \begin{cases} a = \alpha \\ b = -\beta \\ c = 0 \\ a^2 + b^2 = \alpha^2 + \beta^2 = \sin^2\theta + \cos^2\theta = 1 \end{cases} \rightarrow d_l^2(p) = (\alpha\bar{j} - \beta\bar{i})^2$$

The moment of inertia with respect to the above line through the barycentre can thus be expressed as:

$$M(\bar{\mathbf{l}}) = \sum_{p\in R} d_l^2(p) = \sum_{p\in R} (\alpha\bar{j} - \beta\bar{i})^2$$



This formula represents the sum of the distances with respect to the i axis, so, this is the moment of inertia.

FINDING THE MAJOR AXIS

$$M(\bar{\mathbf{l}}) = \sum_{p\in R} (\alpha\bar{j} - \beta\bar{i})^2$$

$$M(\bar{\mathbf{l}}) = \alpha^2 \sum_{p\in R} \bar{j}^2 - 2\alpha\beta \sum_{p\in R} \bar{i}\bar{j} + \beta^2 \sum_{p\in R} \bar{i}^2$$

$$\sum_{p\in R} (j - j_b)^2 = M'_{0,2} \qquad \sum_{p\in R} (i - i_b)(j - j_b) = M'_{1,1} \qquad \sum_{p\in R} (i - i_b)^2 = M'_{2,0}$$

$$M(\bar{\mathbf{l}}) = \alpha^2 M'_{0,2} - 2\alpha\beta M'_{1,1} + \beta^2 M'_{2,0}$$

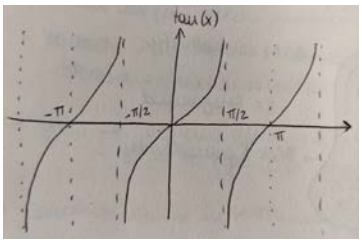
$$\begin{cases} \alpha = -\sin\theta \\ \beta = \cos\theta \end{cases} \rightarrow \quad M(\theta) = \sin^2\theta\, M'_{0,2} + 2\sin\theta\cos\theta\, M'_{1,1} + \cos^2\theta\, M'_{2,0}$$

Therefore, finding the major axis boils down to determining the **minimum of the above function of θ.** A necessary condition to determine such a minimum consists in the first derivative being zero:

$$M(\theta) = \sin^2\theta\, M'_{0,2} + 2\sin\theta\cos\theta\, M'_{1,1} + \cos^2\theta\, M'_{2,0}$$

$$\frac{dM(\theta)}{d\theta} = \underbrace{2\sin\theta\cos\theta}_{\sin 2\theta} M'_{0,2} + 2\underbrace{(\cos^2\theta - \sin^2\theta)}_{\cos 2\theta} M'_{1,1} - \underbrace{2\sin\theta\cos\theta}_{\sin 2\theta} M'_{2,0}$$

$$\frac{dM(\theta)}{d\theta} = \sin 2\theta (M'_{0,2} - M'_{2,0}) + 2\cos 2\theta\, M'_{1,1}$$



The peculiarity of the tangent is that it is periodic as it can be seen in the picture.

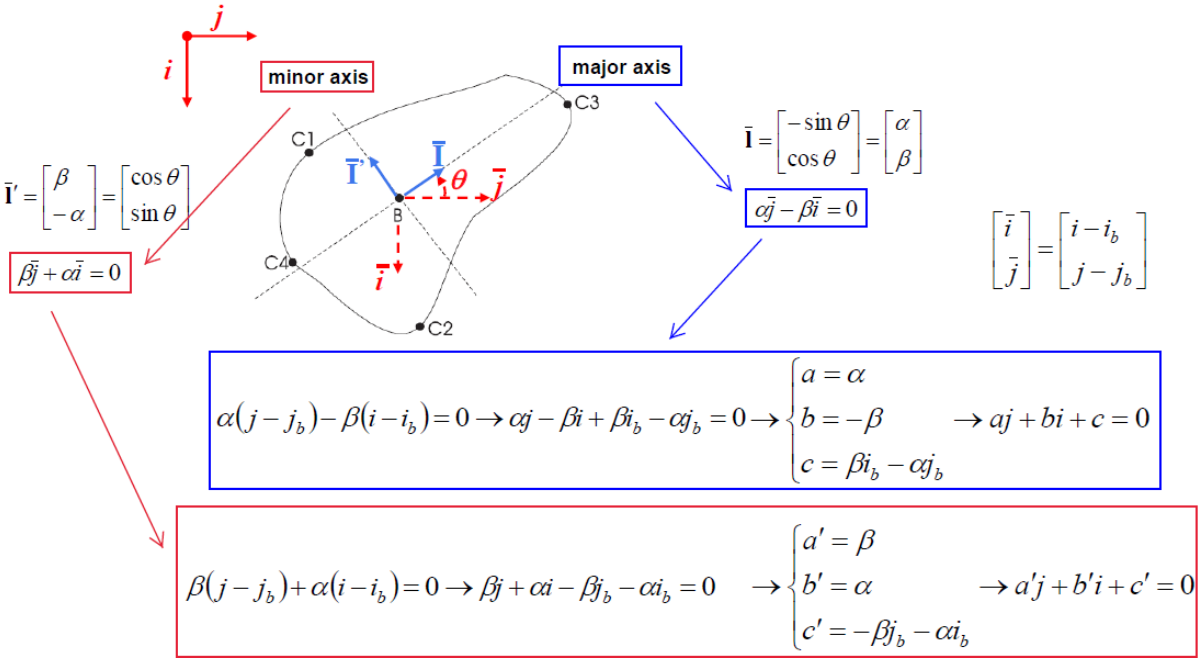
$$\frac{dM(\theta)}{d\theta} = 0 \rightarrow \tan 2\theta = -\frac{2M'_{1,1}}{(M'_{0,2} - M'_{2,0})} \rightarrow 2\theta = \arctan\left(-\frac{2M'_{1,1}}{(M'_{0,2} - M'_{2,0})}\right) = -\arctan\left(\frac{2M'_{1,1}}{(M'_{0,2} - M'_{2,0})}\right)$$

The analysis of the second derivative (**positive**) will show the solution to provide a **minimum major axis,** while the analysis of the second derivative (**negative**) will show the solution to provide a **maximum minor axis**.

$$\theta = -\frac{1}{2}\arctan\left(\frac{2M'_{1,1}}{(M'_{0,2} - M'_{2,0})}\right), \quad 2\theta = -\arctan\left(\frac{2M'_{1,1}}{(M'_{0,2} - M'_{2,0})}\right) + \pi$$

$$\theta = -\frac{1}{2}\arctan\left(\frac{2M'_{1,1}}{(M'_{0,2} - M'_{2,0})}\right) + \frac{\pi}{2}$$

The major and minor axes can be expressed conveniently in the image reference frame:



ORIENTED ENCLOSING RECTANGLE

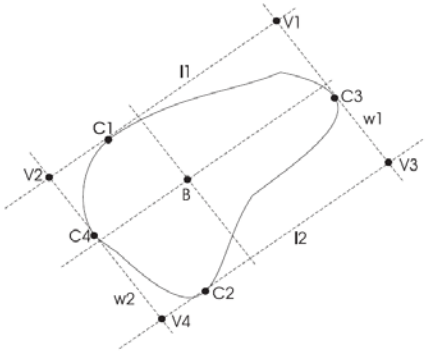
Given the two axes, we might wish to draw a **bounding box aligned to the object**, which is usually referred to as [oriented MER (Minimum Enclosing Rectangle).] Purposely, the four points laying at maximum distance on opposite sides of the two axes need to be determined.

The algorithm relies on the signed "distance" to a line turning out positive or negative depending on the point lying on either of the two sides wrt the line.

If we use an image oriented rather than a MER, this is going to be smaller or bigger depending on the object. That's why we prefer very much to compute a MER.

We have barycentre, major axis, minor axis and we build the rectangle, so we need to find C1, C2, C3, C4: C1 and C2 are the 2 max points on the major axis, whereas C3 and C4 are the 2 max points on the minor axis.

```
% major axis: aj+bi+c=0, minor axis: a'j*b'i+c'=0,
% edges is a binary image representing the contour of the object,
% image size is (nr,nc).

dMAmin=100000; dMAmax=-100000; dMImin=100000; dMImax=-100000;
normMA=sqrt(a*a + b*b); normMI=sqrt(a'*a' + b'*b');
for i=1:nr
for j=1:nc
if (edges(i,j)==1)
dMA=(a*j+b*i+c)/normMA;
dMI=(a'*j + b'*i+c')/normMI;
if (dMA<dMAmin)
dMAmin=dMA; i1=i; j1=j; end
if (dMA>dMAmax)
dMAmax=dMA; i2=i; j2=j; end
if (dMI<dMImin)
dMImin=dMI; i3=i; j3=j; end
if (dMI>dMImax)
dMImax=dMI; i4=i; j4=j; end
end
end
end
% C1: (i1, j1), C2: (i2, j2), C3: (i3, j3) C4: (i4, j4)
```



There is a problem here: which algorithm can we use to find C1 and C2? These points are on the contour of the region, of the blob. So, we can use for example erosion and subtraction, or we just scan the picture and we find which pixels are black and surrounded with white neighbourhood.

In order to find C2 we have to use the signed distance, instead, because C2 is negative and it tells us about the position of the

pixels, while unsigned distance tells us what pixel is the biggest, but it is only positive, so with that we cannot find C2.

Then, we can then find the lines through C1, C2 parallel to the major axis and those through C3, C4 parallel to the minor axis:

$$\begin{array}{lll} l_1: & aj+bi+c_{l_1} & \Rightarrow \quad c_{l_1}=-(aj_1+bi_1) \\ l_2: & aj+bi+c_{l_2} & \Rightarrow \quad c_{l_2}=-(aj_2+bi_2) \\ w_1: & a'j+b'i+c_{w_1} & \Rightarrow \quad c_{w_1}=-(a'j_3+b'i_3) \\ w_2: & a'j+b'i+c_{w_2} & \Rightarrow \quad c_{w_2}=-(a'j_4+b'i_4) \end{array}$$

Eventually, the vertexes of the oriented MER are given by:

$$\begin{array}{ll} j_{V_1}=\left(bc_{w_1}-b'c_{l_1}\right)/\left(ab'-ba'\right), & i_{V_1}=\left(a'c_{l_1}-ac_{w_1}\right)/\left(ab'-ba'\right) \\ j_{V_2}=\left(bc_{w_2}-b'c_{l_1}\right)/\left(ab'-ba'\right), & i_{V_2}=\left(a'c_{l_1}-ac_{w_2}\right)/\left(ab'-ba'\right) \\ j_{V_3}=\left(bc_{w_1}-b'c_{l_2}\right)/\left(ab'-ba'\right), & i_{V_3}=\left(a'c_{l_2}-ac_{w_1}\right)/\left(ab'-ba'\right) \\ j_{V_4}=\left(bc_{w_2}-b'c_{l_2}\right)/\left(ab'-ba'\right), & i_{V_4}=\left(a'c_{l_2}-ac_{w_2}\right)/\left(ab'-ba'\right) \end{array}$$

FEATURES RELATED TO THE MER

Length (L) and width (W):

L: extent of the object along the major axis

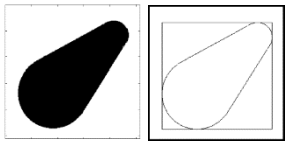
$$L=d_{V_1V_2}=d_{V_3V_4}=\sqrt{\left(i_{V_1}-i_{V_2}\right)^2+\left(j_{V_1}-j_{V_2}\right)^2}=\sqrt{\left(i_{V_3}-i_{V_4}\right)^2+\left(j_{V_3}-j_{V_4}\right)^2}$$

W: extent of the object along the minor axis

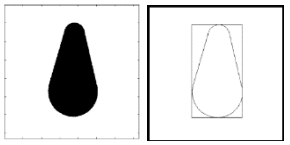
$$W=d_{V_1V_3}=d_{V_2V_4}=\sqrt{\left(i_{V_1}-i_{V_3}\right)^2+\left(j_{V_1}-j_{V_3}\right)^2}=\sqrt{\left(i_{V_2}-i_{V_4}\right)^2+\left(j_{V_2}-j_{V_4}\right)^2}$$

- **Elongatedness:** $E=\dfrac{L}{W}$
- **Rectangularity:** $R=\dfrac{A}{LW}$ — A: object's area, LW: area of the oriented MER
- **Ellipticity:** $E=\dfrac{A}{A_{LW}}, \qquad A_{LW}=\dfrac{\pi}{4}LW$ — A: object's area, A_{LW}: area of the ellipse oriented exactly as the object and having major and minor axis lengths equal to L and W respectively.

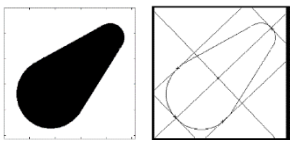
If we use Image Oriented Bounding Boxes, rotating the image, the features change dramatically, in fact computation of L, W and R based on an image-aligned MER does not provide invariance to rotation.
Conversely, invariance to rotation is achieved as long as the features (L, W, R) are computed according to the oriented MER.



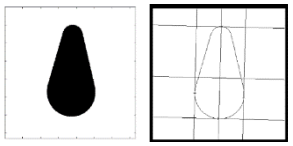
$W=207,$
$L=212,$
$R=0.5535$



$W=257,$
$L=137,$
$R=0.6899$



$W=135.3212,$
$L=254.8374,$
$R=0.7043$



$W=135.3099,$
$L=255.2466,$
$R=0.7033$

FEATURES FROM THE COVARIANCE MATRIX

Orientation, size and elongatedness of a region can also be estimated based on a different interpretation of the data.

In particular, we assume the image coordinates of the pixels belonging to the region to be samples of a 2D random variable (i.e., a bivariate random variable). Accordingly, we compute the mean (i.e., the barycentre of the object) and the covariance matrix.

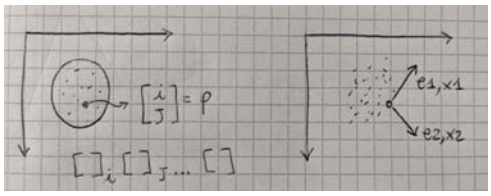
$$\mathbf{p} = \begin{bmatrix} i \\ j \end{bmatrix}, \ \mathbf{p} \in R, \quad \mathrm{A} = \sum_{\mathbf{p} \in R} 1$$

$$\mathbf{B} = \begin{bmatrix} i_b \\ j_b \end{bmatrix} = \frac{1}{A} \sum_{\mathbf{p} \in R} \mathbf{p} = \frac{1}{A} \begin{bmatrix} \sum_{\mathbf{p} \in R} i \\ \sum_{\mathbf{p} \in R} j \end{bmatrix}, \qquad \boldsymbol{\Sigma} = \begin{pmatrix} \sigma^2_{ii} & \sigma^2_{ij} \\ \sigma^2_{ij} & \sigma^2_{jj} \end{pmatrix}$$

$$\sigma^2_{ii} = \frac{1}{A} \sum_{\mathbf{p} \in R} (i - i_b)^2, \quad \sigma^2_{jj} = \frac{1}{A} \sum_{\mathbf{p} \in R} (j - j_b)^2, \quad \sigma^2_{ij} = \frac{1}{A} \sum_{\mathbf{p} \in R} (i - i_b)(j - j_b)$$

$$\frac{M'_{2,0}}{A} \qquad \frac{M'_{0,2}}{A} \qquad \frac{M'_{1,1}}{A}$$

In practice, there's another way to find the previous result, but it underlines other pov. Let us use the covariance matrix: if we are talking about covariance, we assume that we interpret the object as a 2D random variable.



If we compute the min of the samples of the random variable, and we divide by the area, we obtain the barycentre. If it is the 1D variable, so a vector, we can calculate also the covariance.

When we discussed the Mahalanobis distance, we said that the covariance can be diagonalized.

DIAGONALIZATION OF THE COVARIANCE MATRIX

As already pointed out, as Σ is real and symmetric:

$$\boldsymbol{\Sigma} = \mathbf{R}\mathbf{D}\mathbf{R}^T : \quad \mathbf{R} = (\mathbf{e}_1 \ \mathbf{e}_2), \quad \mathbf{D} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

$$\begin{cases} \lambda_1 : \text{largest eigenvalue}, \rightarrow \mathbf{e}_1 \\ \lambda_2 : \text{smallest eigenvalue}, \rightarrow \mathbf{e}_2 \end{cases}$$

direction of maximal dispersion of the random variable (major axis)

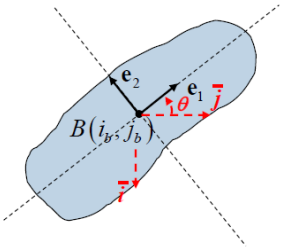
direction of minimal dispersion of the random variable (minor axis)

In order to determine the orientation:

$$\Sigma \mathbf{e_1} = \lambda_1 \mathbf{e_1} \rightarrow (\Sigma - \lambda_1 \mathbf{I}) \mathbf{e_1} = 0$$

$$\begin{pmatrix} \sigma^2_{ii} - \lambda_1 & \sigma^2_{ij} \\ \sigma^2_{ij} & \sigma^2_{jj} - \lambda_1 \end{pmatrix} \begin{pmatrix} e_{1i} \\ e_{1j} \end{pmatrix} = 0$$

$$(\sigma^2_{ii} - \lambda_1) e_{1i} + \sigma^2_{ij} e_{1j} = 0$$
$$\sigma^2_{ij} e_{1i} + (\sigma^2_{jj} - \lambda_1) e_{1j} = 0$$



Let's then consider the unit eigenvector: $\mathbf{e}_1 = \begin{bmatrix} e_{1i} \\ e_{1j} \end{bmatrix} = \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix}$

$$-(\sigma^2_{ii} - \lambda_1)\sin\theta + \sigma^2_{ij}\cos\theta = 0$$
$$-\sigma^2_{ij}\sin\theta + (\sigma^2_{jj} - \lambda_1)\cos\theta = 0$$

$$-(\sigma^2_{ii} - \lambda_1)\sin\theta + \sigma^2_{ij}\cos\theta = 0 \rightarrow -\sigma^2_{ii}\sin\theta + (\sigma^2_{jj} - \sigma^2_{ij}\tan\theta)\sin\theta + \sigma^2_{ij}\cos\theta = 0$$

$$-\sigma^2_{ij}\sin\theta + (\sigma^2_{jj} - \lambda_1)\cos\theta = 0 \rightarrow \lambda_1 = \sigma^2_{jj} - \sigma^2_{ij}\tan\theta$$

$$-\sigma^2_{ii}\sin\theta + (\sigma^2_{jj} - \sigma^2_{ij}\tan\theta)\sin\theta + \sigma^2_{ij}\cos\theta = 0$$

$$-\sigma_{ii}^2\tan\theta + (\sigma_{jj}^2 - \sigma_{ij}^2\tan\theta)\tan\theta + \sigma_{ij}^2 = 0$$

$$-\sigma_{ii}^2\tan\theta + \sigma_{jj}^2\tan\theta - \sigma_{ij}^2(\tan\theta)^2 + \sigma_{ij}^2 = 0$$

$$\sigma_{ij}^2(1 - (\tan\theta)^2) = (\sigma_{ii}^2 - \sigma_{jj}^2)\tan\theta$$

$$\frac{\tan\theta}{1 - (\tan\theta)^2} = \frac{\sigma_{ij}^2}{\sigma_{ii}^2 - \sigma_{jj}^2} \rightarrow \frac{1}{2}\tan 2\theta = \frac{\sigma_{ij}^2}{\sigma_{ii}^2 - \sigma_{jj}^2} \rightarrow \tan 2\theta = \frac{2\sigma_{ij}^2}{\sigma_{ii}^2 - \sigma_{jj}^2}$$

$$\theta = \frac{1}{2}\tan^{-1}\frac{2\sigma_{ij}^2}{\sigma_{ii}^2 - \sigma_{jj}^2} \rightarrow \theta = -\frac{1}{2}\tan^{-1}\frac{2\sigma_{ij}^2}{\sigma_{jj}^2 - \sigma_{ii}^2} = -\frac{1}{2}\tan^{-1}\frac{2M_{1,1}'}{M_{0,2}' - M_{2,0}'}$$

In order to compute the eigenvalues:

$$\boldsymbol{\Sigma}\mathbf{e} = \lambda\mathbf{e} \rightarrow (\boldsymbol{\Sigma} - \lambda\mathbf{I})\mathbf{e} = 0 \rightarrow \det(\boldsymbol{\Sigma} - \lambda\mathbf{I}) = 0$$

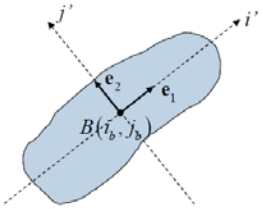
$$\det\begin{pmatrix} \sigma_{ii}^2 - \lambda & \sigma_{ij}^2 \\ \sigma_{ij}^2 & \sigma_{jj}^2 - \lambda \end{pmatrix} = 0 \rightarrow (\sigma_{ii}^2 - \lambda)(\sigma_{jj}^2 - \lambda) - (\sigma_{ij}^2)^2 = 0$$

Characteristic Polynomial

$$\lambda^2 - \lambda(\sigma_{ii}^2 + \sigma_{jj}^2) + \sigma_{ii}^2\sigma_{jj}^2 - (\sigma_{ij}^2)^2 = 0 \rightarrow \lambda_1, \lambda_2$$

This. solving the quadratic equation yields the eigenvalues:

$$\lambda_1 = \frac{\sigma_{ii}^2 + \sigma_{jj}^2 + \sqrt{(\sigma_{ii}^2 - \sigma_{jj}^2)^2 + 4(\sigma_{ij}^2)^2}}{2}, \quad \lambda_2 = \frac{\sigma_{ii}^2 + \sigma_{jj}^2 - \sqrt{(\sigma_{ii}^2 - \sigma_{jj}^2)^2 + 4(\sigma_{ij}^2)^2}}{2}$$



$$\mathbf{p}' = \begin{bmatrix} i' \\ j' \end{bmatrix} : \mathbf{p}' = \mathbf{R}^T(\mathbf{p} - \mathbf{B}) \; \forall \mathbf{p} \in R$$

Hotelling Transform: brings pixel coordinates into a central reference frame with axes aligned to the eigenvectors

$$L = i'_{max} - i'_{min}$$
$$i'_{max} = \max(i'), \mathbf{p} \in R$$
$$i'_{min} = \min(i'), \mathbf{p} \in R$$
$$W = j'_{max} - j'_{min}$$
$$j'_{max} = \max(j'), \mathbf{p} \in R$$
$$j'_{min} = \min(j'), \mathbf{p} \in R$$

Faster approximations based on computing the eigenvalues only (i.e. without applying the Hotelling Transform).

$$L \approx 2\sqrt{\lambda_1}$$
$$W \approx 2\sqrt{\lambda_2}$$
$$E = \sqrt{\frac{\lambda_1}{\lambda_2}}$$

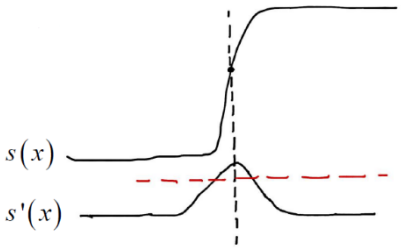
The square roots of the eigenvalues are the semi-length of the major and minor axes of the ellipse, which is centered at the barycentre and has the same orientation as the object (fitting ellipse).

$$\mathbf{p}^T\Sigma^{-1}\mathbf{p} = 1$$

8 - EDGE DETECTION

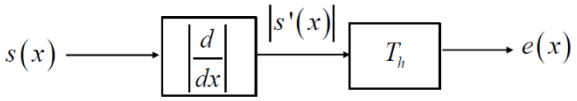
Edge or contour points are local features of the image that capture effectively important information related to its semantic content.
Therefore, edges are exploited in countless image analysis tasks, such as foreground-background segmentation, object description, template matching, stereo matching, visual tracking…
Edges are pixels that can be thought of as lying exactly in between image regions of different intensity, or, in other words, to separate different semantic regions.

8.1 – 1D STEP-EDGE



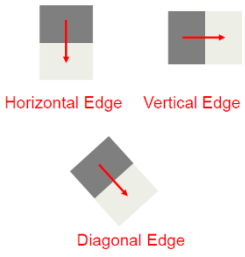
We can model an edge in 1D like a signal like this. We consider a 1D signal and we call a Step Edge as quick transition between two constant signals.
So, we can localize the edge in the point where the function changes.
The first derivative of the signal is high only in the region where we localize the edge. So, we could apply a thresholding which highlights changes, i.e., the region where the edge is.

Looking at the red line, we notice that **in the transition region** between the two constant levels, **the absolute value of the first derivative gets high** (and reaches an extremum at the inflection point).
Accordingly, the simplest edge-detection operator relies on thresholding the absolute value of the derivative of the signal:



8.2 – 2D STEP-EDGE

A 2D Step-Edge is characterized not only by its strength but also its **direction**.
Hence, we cannot use a directional derivative but instead need an operator allowing to sense the edge whatever its direction is.
Such operator is the **gradient**:

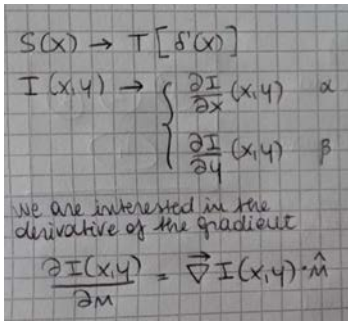


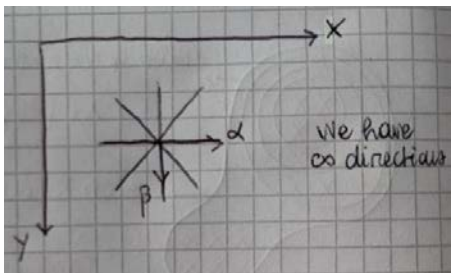
$$\nabla I(x,y) = \frac{\partial I(x,y)}{\partial x}\mathbf{i} + \frac{\partial I(x,y)}{\partial y}\mathbf{j}$$

Gradient's direction gives the direction along which the function exhibits its maximum variation, gradient's magnitude being the absolute value of the directional derivative along such direction (i.e., the absolute value of the maximum directional derivative).
Indeed, **a generic directional derivative can be computed by the dot product between the gradient and the unit vector along the direction.**

If I take the image, I compute the derivative and I apply a thresholding, what's wrong with that? What derivative should I compute? The derivative is undefined in the image.
Let's assume that an image is a continuous 2D function, can we calculate the derivative? This would be a function of what? We have to calculate partial derivative of x and y (the vertical and horizontal variables).

Let's apply edge detection, but we have to generalize the concept: are we sure we have just 2 derivatives? Is that actually true?

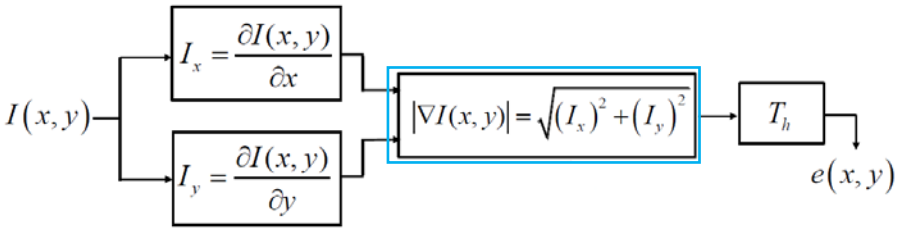




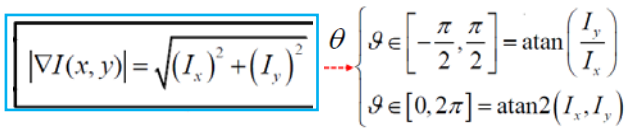
We calculate the derivative, so the variation in each direction, but we have an infinity of directions. Each of these directions tells us the variation along this direction: if the derivative is low, the signal is not changing that much.
In case of 1D, we need to apply a threshold, a number. But here what can we do? We cannot threshold all derivatives, but we can do a generalization applying the gradient.
The point is that we have infinite directions and if at least in one direction there's an interesting value, that's the direction we have to focus on. Indeed, in a uniform area, the derivative would be low.
Nonetheless, we cannot try all directions, we cannot afford that.
The gradient will help us in multi dimension functions giving us the maximum variation, so **the derivative to which we can apply the threshold is the derivative along the gradient direction because it is pointing the direction of the maximum variation.**

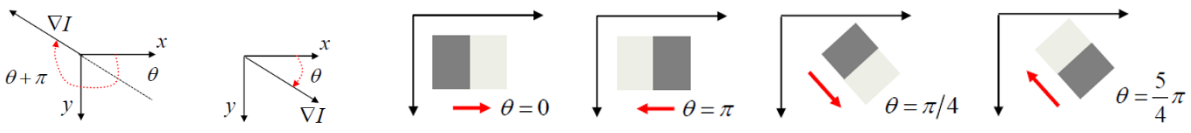
8.3 – EDGE DETECTION BY GRADIENT THRESHOLDING

Firstly, we have to compute the partial derivatives, then we compute the gradient vector and we obtain the maximum derivative which is the highest.

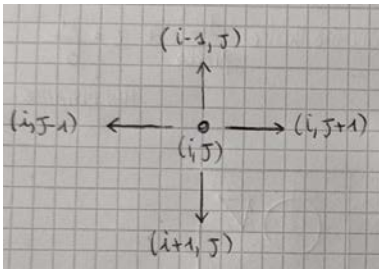


Then, we compute the inverse tangent and we can compute also the atan2 which uses the angles: this is useful to find the contrast polarity. The **<u>contrast polarity</u>** appears when in the **same direction we have different signs**.



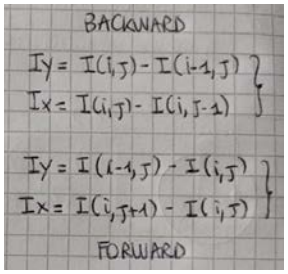


8.3.1 – DISCRETE APPROXIMATION OF THE GRADIENT



We can approximate derivatives which are continuous entities in discrete entities. For doing that, we compute only the differences.
To approximate we use 2 type of differences: backward and forward. We compute the derivatives as a difference between:

- The pixel and the previous one → **Backward**
- The next pixel and our pixel → **Forward**



We can use either backward or forward differences, because they're equivalent:

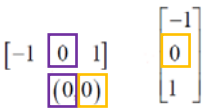
$$\frac{\partial I(x,y)}{\partial x} \cong I_x(i,j) = I(i,j) - I(i,j-1) \qquad \frac{\partial I(x,y)}{\partial y} \cong I_y(i,j) = I(i,j) - I(i-1,j)$$

$$\frac{\partial I(x,y)}{\partial x} \cong I_x(i,j) = I(i,j+1) - I(i,j) \qquad \frac{\partial I(x,y)}{\partial y} \cong I_y(i,j) = I(i+1,j) - I(i,j)$$

We which may be thought of as correlation by:

Or also central differences with the corresponding correlation kernels being:

$$I_x(i,j) = I(i,j+1) - I(i,j-1), \qquad I_y(i,j) = I(i+1,j) - I(i-1,j)$$



We use different approaches to estimate the approximations, like the magnitude. This sounds as the best way because it is very precise, but is quite computationally expensive.
We have an edge, maybe horizontal, but if we rotate the object within the background, the angle is the same and the edge is the same.

To compute the gradient of the first figure Ev, we compute the partial derivative and we compute the magnitude based on the components. For the vertical derivative we can use forward and it is "0", while for the horizontal derivative we have "h".
If we compute the magnitude using the norm, we still obtain "h".
In the first and in the second figure (Ev and Eh), there is **isotropy**, while in the third figure (Ed) if we consider the estimation of the magnitude, what kind of magnitude we're going to compute?
We have $h + h = 2h$, but it is wrong because it is only rotated. To obtain an isotropic response, we have to consider the maximum between the 2 derivatives.

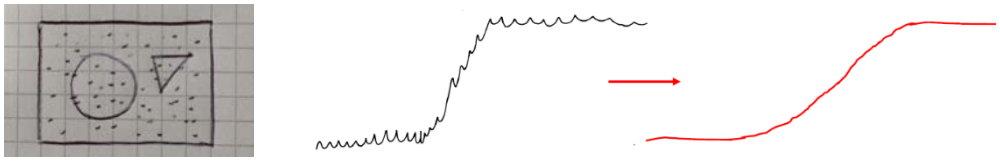
We can estimate the magnitude by different approximations:

$$|\nabla I| = \sqrt{(I_x)^2 + (I_y)^2} \qquad |\nabla I|_+ = |I_x| + |I_y| \qquad |\nabla I|_{max} = max\left(|I_x|, |I_y|\right)$$

	$\lvert\nabla I\rvert$	$\lvert\nabla I\rvert_+$	$\lvert\nabla I\rvert_{max}$
E_h	h	h	h
E_v	h	h	h
E_d	$h\sqrt{2}$	$2h$	h

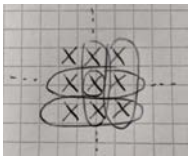
8.4 – DEALING WITH NOISE

Due to noise, in real images an edge will likely look as depicted in the picture below:



This makes it hard to detect the main step edge out of the many spurious signal changes due to noise.

To work with **real images**, an edge detector should therefore be **robust to noise**, so as to highlight the meaningful edges only and filter out effectively the spurious transitions caused by noise.
This is usually **achieved by smoothing the signal before computing the derivatives** required to highlight edges. Unfortunately, **smoothing yields also the side-effect of blurring true edges**, making it more difficult to detect and localize them accurately.



8.4.1 – SMOOTHING DERIVATIVES

This is the simplest way to design an edge detector.
Let's focus on a 3x3 neighbourhood. We have to calculate differences: we can use forward or backward, but there's a lot of noise everywhere so what we get is only noise.
The higher the order of derivative we take, the worse is the signal to noise ratio. As long as we go to apply derivative, the signal becomes more and more worse.
If we apply instead smooth derivatives, we will have differences of averages.

Smoothing and differentiation can be carried out jointly within a single step. This is achieved by computing differences of averages (rather than averaging the image and then computing differences).
To try avoiding smoothing across edges, the two operations are carried out along orthogonal directions.

Example
If we wish to smooth out noise by averaging over 3 pixels:

$$
\begin{aligned}
I_{3y}(i,j) &= \frac{1}{3}[I(i-1,j)+I(i,j)+I(i+1,j)] \\
I_{3x}(i,j) &= \frac{1}{3}[I(i,j-1)+I(i,j)+I(i,j+1)]
\end{aligned}
$$

$$
\begin{aligned}
\tilde{I}_x(i,j) &= I_{3y}(i,j+1) - I_{3y}(i,j) = \frac{1}{3}[I(i-1,j+1)+I(i,j+1)+I(i+1,j+1) \\
&- I(i-1,j)-I(i,j)-I(i+1,j)] \\
&\Rightarrow \frac{1}{3}\begin{bmatrix} -1 & 1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
\tilde{I}_y(i,j) &= I_{3x}(i+1,j) - I_{3x}(i,j) = \frac{1}{3}[I(i+1,j-1)+I(i+1,j)+I(i+1,j+1) \\
&- I(i,j-1)-I(i,j)-I(i,j+1)] \\
&\Rightarrow \frac{1}{3}\begin{bmatrix} -1 & -1 & -1 \\ 1 & 1 & 1 \end{bmatrix}
\end{aligned}
$$

PREWITT, SOBEL and FREI-CHEN OPERATORS

Given the same smoothing, one might wish to approximate partial derivatives by central differences (**Prewitt operator**). In this way, we obtain more isotropic response, i.e., less attenuation of diagonal edges.

$$\tilde{I}_x(i,j) = I_{3y}(i,j+1) - I_{3y}(i,j-1) \quad \Rightarrow \quad \frac{1}{3}\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\tilde{I}_y(i,j) = I_{3x}(i+1,j) - I_{3y}(i-1,j) \quad \Rightarrow \quad \frac{1}{3}\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Likewise, the central pixel can be weighted more to further improve isotropy, using the **Sobel operator**. The Sobel Operator is the most common edge detector operator and it weights the central pixel.

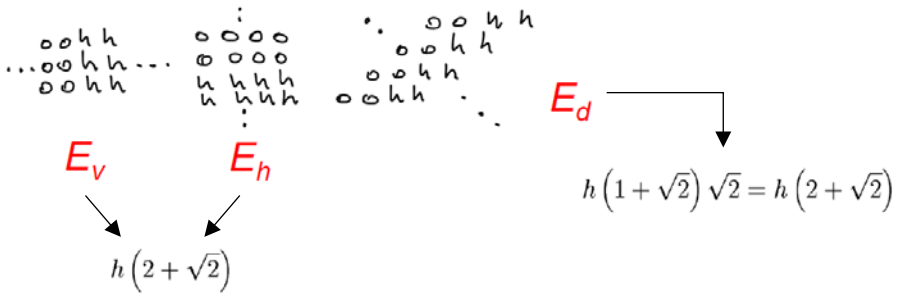
$$I_{4y}(i,j) = \frac{1}{4}[I(i-1,j) + 2I(i,j) + I(i+1,j)]$$
$$I_{4x}(i,j) = \frac{1}{4}[I(i,j-1) + 2I(i,j) + I(i,j+1)]$$

$$\tilde{I}_x(i,j) = I_{4y}(i,j+1) - I_{4y}(i,j-1) \quad \Rightarrow \quad \frac{1}{4}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\tilde{I}_y(i,j) = I_{4x}(i+1,j) - I_{4y}(i-1,j) \quad \Rightarrow \quad \frac{1}{4}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Though at the expense of a significantly higher computational complexity, full isotropy is provided by the **Frei-Chen operator**:

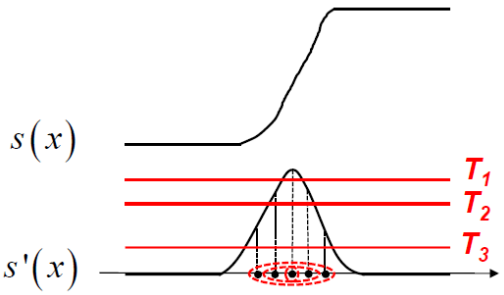
$$\tilde{I}_x = \begin{bmatrix} -1 & 0 & 1 \\ -\sqrt{2} & 0 & \sqrt{2} \\ -1 & 0 & 1 \end{bmatrix} \qquad \tilde{I}_y = \begin{bmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & \sqrt{2} & 1 \end{bmatrix} \qquad |\nabla I| = \sqrt{(I_x)^2 + (I_y)^2}$$



8.5 – EDGE DETECTION BY FINDING MAXIMA

Detecting edges by gradient thresholding is inherently inaccurate as regards localization: it turns out difficult in practice to choose the right threshold whenever the image contains meaningful edges characterized by different contrast (i.e., "stronger" as well as "weaker").
Trying to detect weak edges implies poor localization of stronger ones.

Analysing the picture, we understand that a **better approach to detect edges may consist in finding the local maxima of the absolute value of the derivative of the signal**.

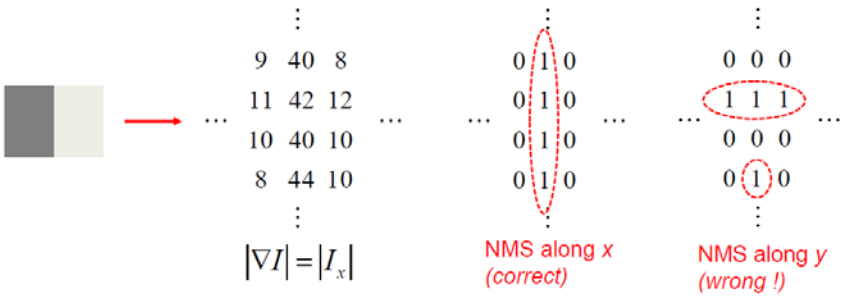


8.6 – EDGE DETECTION BY SMOOTH DERIVATIVES AND NMS

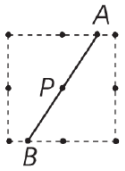
8.6.1 – NON-MAXIMA SUPPRESSION (NMS)

When dealing with images (2D signals) one should look for maxima of the absolute value of the derivative (i.e. the gradient magnitude) along the gradient direction (i.e. orthogonally to the edge direction). This process is called **NMS: Non-Maxima Suppression**.

In practice, here we suppress everything except the maxima. How can we find the maxima of a 2D signal? We should perform the NMS directionally: along the gradient direction. Is the value higher than the label along the direction?

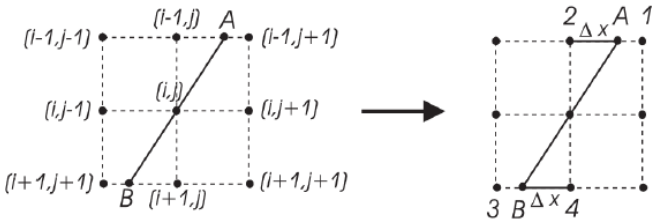


The last image is not correct because we don't have only 1 in the horizontal line, but also in another line. We should perform this according to the correct direction.
In fact, we don't know in advance the correct direction to carry out NMS. Indeed, such a direction has to be estimated locally, of course based on gradient's direction.



NMS BY LINEAR INTERPOLATION

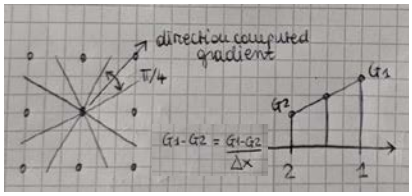
To perform NMS at pixel P precisely, the magnitude of the gradient has to be estimated at points which do not belong to the pixel grid (e.g. A, B in the picture).
Such values can be estimated by linear interpolation of those computed at the closest points belonging to the grid (possibly after projection along the gradient at P).



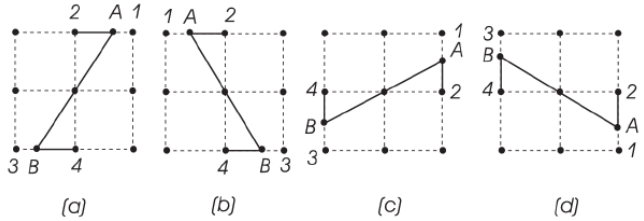
$$G_1 = |\nabla I(1)| \quad G_2 = |\nabla I(2)| \quad G_3 = |\nabla I(3)| \qquad G_A = G_2 + (G_1 - G_2)\,\Delta x$$
$$G_4 = |\nabla I(4)| \quad G_A = |\nabla I(A)| \quad G_B = |\nabla I(B)| \qquad G_B = G_4 + (G_3 - G_4)\,\Delta x$$

So, we consider the pixel and we look at th e direction. We have this smooth gradient along the image. Firstly, we have to look at the neighbours (3x3 neighbourhood).
There are two main ways to find the edges: the simplest think we can do is that we can take sectors (we have 8) and they are all the same size. E very slice is 45° so $\pi/4$ e the gradient direction for this pixel is the arrow. So, when we do the NMS, we consider a pair of elements.

The other approach is better and it's a more advanced approach: we need to find the gradient in various parts and then we can calculate what we need. Since we don't have whole gradients, we have gradients applied to singular pixels. So, we linear interpolate the gradients.
Ga and Gb have to be compared to know if our line is an edge.



Points previously denoted as 1,2,3,4 are indeed different ones depending on gradient's direction:



$$(a),(b) \;:\; |\vartheta| \geq \frac{\pi}{4} \quad \left(|\tilde{I}_y| \geq |\tilde{I}_x|\right)$$

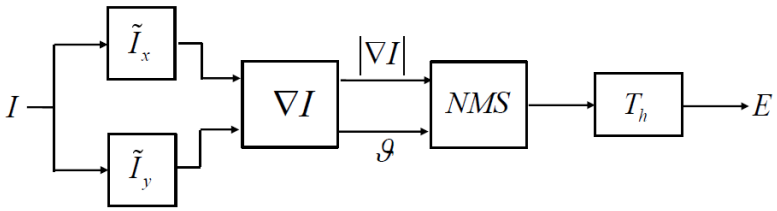
$$(c),(d) \;:\; |\vartheta| < \frac{\pi}{4} \quad \left(|\tilde{I}_y| < |\tilde{I}_x|\right)$$

$$(a),(b) \Rightarrow \begin{cases} (a) \;:\; \vartheta \leq 0 & \left(sign(\tilde{I}_y) \neq sign(\tilde{I}_x)\right) \\ (b) \;:\; \vartheta > 0 & \left(sign(\tilde{I}_y) = sign(\tilde{I}_x)\right) \end{cases} \qquad (a),(b) \Rightarrow \Delta : \Delta x = \frac{1}{|\tan\vartheta|} \quad \left(\frac{|\tilde{I}_x|}{|\tilde{I}_y|}\right)$$

$$(c),(d) \Rightarrow \begin{cases} (c) \;:\; \vartheta \leq 0 & \left(sign(\tilde{I}_y) \neq sign(\tilde{I}_x)\right) \\ (d) \;:\; \vartheta > 0 & \left(sign(\tilde{I}_y) = sign(\tilde{I}_x)\right) \end{cases} \qquad (c),(d) \Rightarrow \Delta : \Delta y = |\tan\vartheta| \quad \left(\frac{|\tilde{I}_y|}{|\tilde{I}_x|}\right)$$

8.6.2 – FLOW-CHART

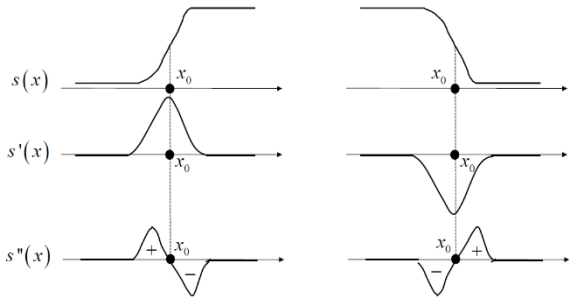
The overall flow-chart of an edge detector based on smooth derivatives and NMS is sketched below:



A final thresholding step typically helps pruning out unwanted edges due to either noise or less important details, so it's added at the end of the process.

ZERO-CROSSING OF THE SECOND DERIVATIVE

Edges may also be located by looking for zero-crossing of the second derivative of the signal.
There are many cases, but we study only two: on the left we have a positive edge, on the right we have a negative edge. We can consider the second derivatives: which threshold we have to use to localize this edge? We pick the maximum of the first derivative.
If we look at the second derivative, could we localize the edge precisely? **Zero-crossing: when the second derivative cross the zero**, this point is the one we're searching.



Pay attention: this is not the point where the signal is zero, but the point where second derivative is zero!

ZERO-CROSSING ALONG THE GRADIENT'S DIRECTION

Again, in the case of images (2D signals) we should look for zero-crossing of the second derivative along **gradient's direction**, but it requires a significant computational effort!

$$\frac{\partial^2 I}{\partial n^2}, \qquad \vec{n} = \frac{\nabla I}{|\nabla I|} \qquad \longrightarrow \qquad \frac{\partial^2 I}{\partial n^2} = \frac{\partial |\nabla I|}{\partial n} = \nabla \left(|\nabla I| \right) \vec{n}$$

$$\nabla \left(|\nabla I| \right) = \nabla \left(I_x^2 + I_y^2 \right)^{\frac{1}{2}} = \frac{\left(I_x I_{xx} + I_y I_{yx} \right) \vec{x} + \left(I_y I_{yy} + I_y I_{yx} \right) \vec{y}}{\left(I_x^2 + I_y^2 \right)^{\frac{1}{2}}}$$

$$\nabla \left(|\nabla I| \right) \vec{n} = \frac{I_x^2 I_{xx} + 2 I_x I_y I_{xy} + I_y^2 I_{yy}}{I_x^2 + I_y^2}$$

The first derivative along the gradient's direction is the **gradient magnitude.** How do we find the derivative of the gradient magnitude? We said that the gradient is the super derivative because in order to take derivatives, we can use the gradient in every direction.
Our function is not the magnitude because **the magnitude is a scalar and not a vector**.
So, the generalization of this idea finding zero-crossing in the second derivative to localize edges, means that we have to compute a lot of stuffs.
This edge detector, in fact, is very good, but it isn't the fastest method because it has to do significant computational effort. Hence, it is not very used because of its being slow.

8.6.3 - THE LAPLACIAN

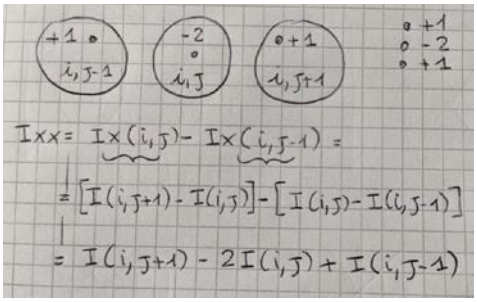
In their influential work on edge detection, **Marr & Hildreth** proposed to rely on the **Laplacian** as second order differential operator:

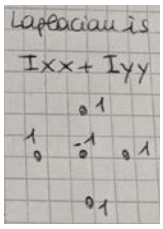
$$\nabla^2 I(x,y) = \frac{\partial^2 I(x,y)}{\partial x^2} + \frac{\partial^2 I(x,y)}{\partial y^2} = I_{xx} + I_{yy}$$

Using forward and backward differences to approximate, respectively, first and second order derivatives:

$$I_{xx} \cong I_x(i,j) - I_x(i,j-1) = I(i,j-1) - 2I(i,j) + I(i,j+1)$$
$$I_{yy} \cong I_y(i,j) - I_y(i-1,j) = I(i-1,j) - 2I(i,j) + I(i+1,j) \qquad \longrightarrow \qquad \nabla^2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Generally, we use a different type of derivative which is the **Laplacian** that is the sum of the second order partial derivatives. To compute the Laplacian, we need the two second order derivatives: this is much faster than the zero crossing and it is quite equivalent in most practical uses.





We use Backward approximation and we don't have to perform any differentiation on top of the image otherwise we get rubbish because there's noise. We have to deal with noise using a smoothing: e.g., we can use the Gaussian Filter.

8.6.4 - LAPLACIAN OF GAUSSIAN (LOG)

It can be shown that the zero-crossing of the Laplacian typically lays close to those of the second derivative along the gradient, being the former differential operator much faster to compute (i.e., just a convolution by a 3x3 kernel) than the latter.

As already discussed, a robust edge detector should include a smoothing step to filter out noise (especially in case second rather than first order derivatives are deployed). In their edge detector, Marr & Hildreth proposed to use a Gaussian filter as smoothing operator. Hence, their edge detector is referred to as **LOG (Laplacian of Gaussian) or Mexican Hat Filter.**

Edge detection by the LOG can be summarized conceptually as follows:

1. Gaussian smoothing: $\tilde{I}(x,y) = I(x,y) * G(x,y)$
2. Second order differentiation by the Laplacian: $\nabla^2 \tilde{I}(x,y)$
3. Extraction of the zero-crossing of $\nabla^2 \tilde{I}(x,y)$

Unlike those based on smooth derivatives, **the LOG edge detector allows the degree of smoothing to be controlled** (i.e. by changing the σ parameter of the Gaussian filter). This, in turn, allows the edge detector to be tuned according to the degree of noise in the image (i.e. higher noise -> larger σ).

Likewise, σ may be used to control the scale at which the image is analyzed:

- Larger σ typically chosen to extract the edges related to main scene structures
- Smaller σ to capture also small size details

Zero-crossing are usually sought for by scanning the image by both rows and columns to identify changes of the sign of the LOG.

Once a sign change is found, the actual edge may be localized:

1. At the pixel where the **LOG is positive** (darker side of the edge)
2. At the pixel where the **LOG is negative** (brighter side of the edge)
3. At the pixel where the **absolute value of the LOG is smaller** (the best choice, as the edge turns out closer to the "true" zero-crossing)

Again, to help discarding spurious edges, **a final thresholding step may be enforced** (usually based on the slope of the LOG at the found zero-crossing).

Example

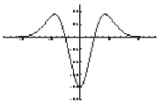
ZCs, σ=1 ZCs, σ=2 ZCs, σ=3

Applying thresholding based on the slope of the LOG (≥40) to the second image:

$ZCs,\ \sigma=1$

$T_h(ZCs),\ \sigma=1$

COMPUTATION OF THE LOG



$$\nabla^2\tilde{I}(x,y)=\nabla^2\left(I(x,y)*G(x,y)\right)=I(x,y)*\boxed{\nabla^2 G(x,y)}$$

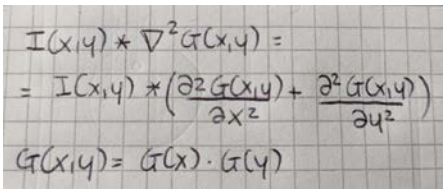
$$\boxed{\nabla^2 G(x,y)}=\frac{\partial^2 G(x,y)}{\partial x^2}+\frac{\partial^2 G(x,y)}{\partial y^2}=\frac{1}{2\pi\sigma^4}\left[\frac{r^2}{\sigma^2}-2\right]e^{-\frac{r^2}{2\sigma^2}},\ r^2=x^2+y^2$$

2D convolution by the Mexican Hat can be expensive in terms of computation, especially when the size of the filter is large. As it is the case of the Gaussian, **the size d of the filter must increase with σ.**
According to several studies: $\mathbf{3\omega \le d \le 4\omega}$ **with** $\omega = 2\sqrt{2}\cdot\sigma$
We have to choose the dimension of the kernel:

$$\begin{aligned}
\sigma=0.5 \quad d=4.24 &\Rightarrow 5x5\\
\sigma=1 \quad d=8.48 &\Rightarrow 9x9\\
\sigma=2 \quad d=16.97 &\Rightarrow 17x17\\
\sigma=3 \quad d=25.45 &\Rightarrow 26x26\\
\sigma=4 \quad d=33.94 &\Rightarrow 34x34\\
\sigma=5 \quad d=42.42 &\Rightarrow 43x43
\end{aligned}$$

However, thanks to separability of the Gaussian, computing the LOG boils down to four 1D convolutions, which is substantially faster (4d rather that d2 ops per pixel):

$$\begin{aligned}
I(x,y)*\nabla^2 G(x,y) &= I(x,y)*\left(G''(x)G(y)+G''(y)G(x)\right)\\
&= I(x,y)*\left(G''(x)G(y)\right)+I(x,y)*\left(G''(y)G(x)\right)\\
&= \left(I(x,y)*G''(x)\right)*G(y)+\left(I(x,y)*G''(y)\right)*G(x)
\end{aligned}$$



8.7 – CANNY'S EDGE DETECTOR

Without applying some denoising, we have bad results, so we have first to denoise and then to apply the calculation of derivatives. The amount of noise which can be dealt with is small, is limited by the size of smoothing filter. Smooth derivatives are typically small.
The LOG is something more specific and accurate: we calculate the second derivative along the gradient direction.
Precise localization is achieved by finding the Laplacian and this is used to find the direction derivative along the gradient direction and it's fast to compute.

Let's now introduce the best well-known edge detector: the Canny Edge Detector. This detector is guaranteed to be optimal.

Canny proposed to set **forth quantitative criteria to measure the performance of an edge detector** and then to find the optimal filter with respect to such criteria. Accordingly, he proposed the following three criteria:

- **Good Detection**: the filter should correctly extract edges in noisy images. This detector should be able to find edges also in noisy images, it has to be robust
- **Good Localization**: the distance between the found edge and the "true" edge should be minimum. This is about precision: if there's a true ideal edge at some pixel position, the detector should be able to localize that edge precisely
- **One Response to One Edge**: the filter should detect one single edge pixel at each "true" edge. We don't want thick edges, we want only one single point marked as edge, so no multiple responses

These are more or less the things we have been saying several times since we have started studying CV. In order to formalize these criteria in rigorous mathematical terms we lead to a demonstration.
How should we seek to find edges if we want to satisfy these criteria? Canny uses a simplified modelling: it studies 1D signals, more or less as we did.
The best edge detector is an algorithm that finds the maxima and the minima of the Gaussian derivative: a filter that is the first derivative of a Gaussian function. We find the Gaussian and we compute the first derivatives, then we compute the extrema. More or less, it is a way to reformulate a criterion we already talked about. To highlight changes and to localize them precisely with one response, we have to find the extrema.
What is striking in this approach? **We are able to demonstrate what we already knew, this is the most famous paper in CV.**

Extension: we filter a 2D signal by a Gaussian and then we look to find the extrema (derivatives are signed) or gradient magnitude (positive quantity) along the gradient direction.
We can do that in the most efficient way because we know that the Gaussian Filter enjoys some peculiar properties.

Addressing the 1D case and modelling an edge as a noisy step, he shows that the optimal edge detection operation consists in **finding local extrema of the convolution of the signal by a first order Gaussian derivative** (i.e. G'(x)).
This theoretical result can also be regarded as a proof of the optimality of the Gaussian as smoothing filter to detect noisy edges.
As usual, to end up with a practical 2D edge detector to be applied to images, we should look for local extrema of the directional derivative along the gradient. As such, **a straightforward Canny edge detector can be achieved by Gaussian smoothing followed by gradient computation and NMS along the gradient direction.**

As already pointed out, 2D convolution by a Gaussian can be slow and we can leverage on separability of the Gaussian function to speed-up the calculation.
The Gaussian is the product between horizontal Gaussian and vertical Gaussian. We take then the convolution along x and y and we obtain a 1D convolution. Now, we have the 2 components.

$$\tilde{I}_x(x,y) = \frac{\partial}{\partial x}\left(I(x,y) * G(x,y)\right) = I(x,y) * \frac{\partial G(x,y)}{\partial x}$$

$$\tilde{I}_y(x,y) = \frac{\partial}{\partial y}\left(I(x,y) * G(x,y)\right) = I(x,y) * \frac{\partial G(x,y)}{\partial y}$$

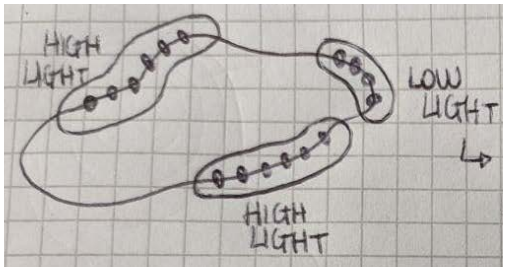
$$G(x,y) = G(x)G(y)$$

$$\tilde{I}_x(x,y) = I(x,y) * \left(G'(x)G(y)\right) = \left(I(x,y) * G'(x)\right) * G(y)$$

$$\tilde{I}_y(x,y) = I(x,y) * \left(G'(y)G(x)\right) = \left(I(x,y) * G'(y)\right) * G(x)$$

As already discussed, NMS is often followed by thresholding of gradient magnitude to help distinguishing between true "semantic" edges and unwanted ones. However, edge streaking may occur when magnitude varies along object contours.

Smart idea: whatever kind of edge detector we are applying, at the end there's always a threshold to take the final decision. Even if we look for extrema with the gradient magnitude or the zero process with the Laplacian, we need a threshold to decide.
What kind of quantity we're going to focus on? In case we use zero processing, we understood that the slope of the Laplacian is an indicator of how strong is the edge. If this quantity is strong enough, this pixel is an edge, otherwise it is not.
To find extrema along gradient direction we can do it dividing in sectors the pie or we can do that precisely moving along gradient direction and interpolating the 2 pixels. At the end we have several pixels candidate to be edge, but only some of those ones are really part of the edge.



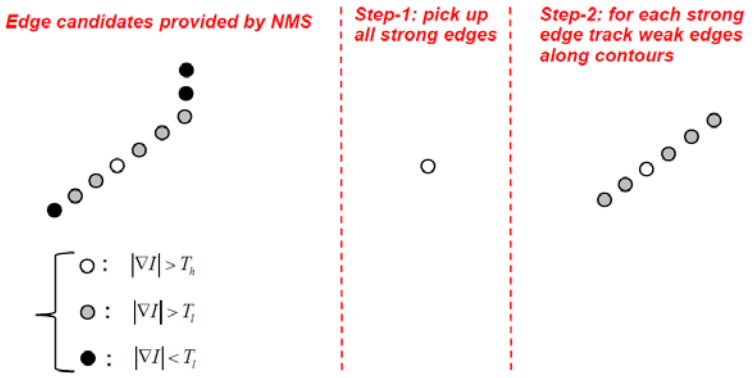
Sometimes we need multiple thresholds. We compare candidate pixels with a threshold and where there's low light, we have low gradient magnitude.
If we use a high threshold, we don't have the pixels in low light, otherwise, if we use a low threshold, we have spurious values.

How can we capture all the edges? We cannot use a single threshold, so we can first take the edges which are surely true (with high threshold) and then - because we know that picking an high threshold is edge streaking (some pieces are lost) - we would try to recover these missing parts and we do that adding to the initial strong contour those other candidates which are not as strong as the initial one but they are connected, they are neighbours. So, we compare these pixels to a threshold which is lower than the first.

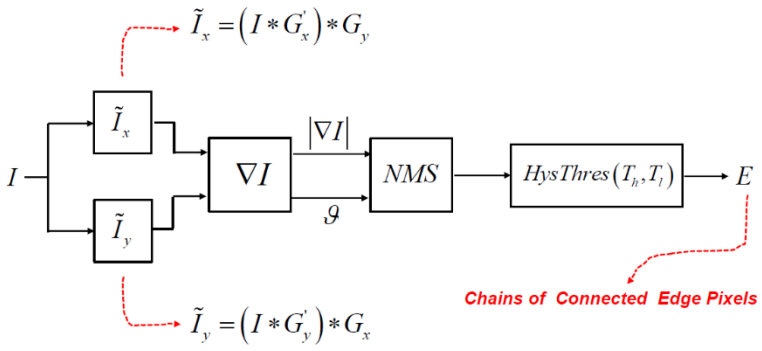
Formally, to address the above issue, Canny proposed a **"hysteresis" thresholding approach** relying on a **higher (Th) and a lower (Tl) threshold**. A pixel is taken as an edge if either the gradient magnitude is higher than Th **OR** higher than Tl **AND** the pixel is a neighbour of an already detected edge.

Hysteresis thresholding is usually carried out by tracking edge pixels along contours, which also brings

in the **"side-effect"** of Canny's providing as output **chains of connected edge pixels** rather than edge maps.

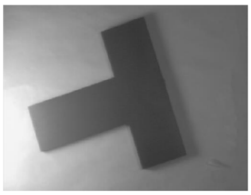


We have the computation of the smooth horizontal and vertical derivatives, we have the gradient (vector whose elements are the 2 component), we have the magnitude direction and finally rather than a single threshold, we apply the Hysteresis thresholds to obtain the final output which is a chain of connected pixels.
So, the overall **flow-chart of the Canny edge detector** is provided below:

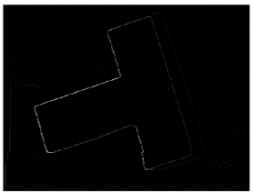


Example

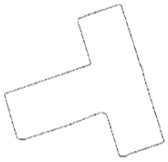
The lighting conditions are particular, so segmentation for this image doesn't work. However, if we look at this picture, we can distinguish the object region and the background region.
So, if we apply Canny, we have the NMS output (all non-candidate pixels, the non-maxima are shown in black. All the other pixels are shown in white) but none all pixels are equally white.
At the end, we apply the final step with multiple threshold and we obtain the correct contour.



Input Image



NMS Output with gradient magnitude represented by gray-scales

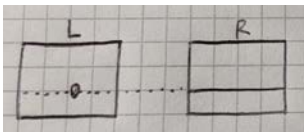


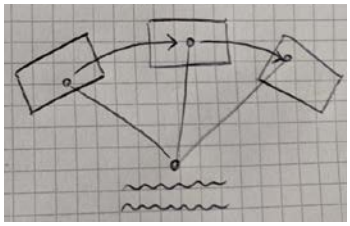
Final Output

9 - LOCAL INVARIANT FEATURES

9.1 – CORRESPONDENCES ARE KEY

Stereo is about finding depth and the key step to performing this estimation is to find the corresponding points. Stereo is kind of simplified set because we cast the problem of finding correspondences using the 2 images.
We find a depth map associated with the left image (which is the reference image) and then we find the corresponding point in the right image.





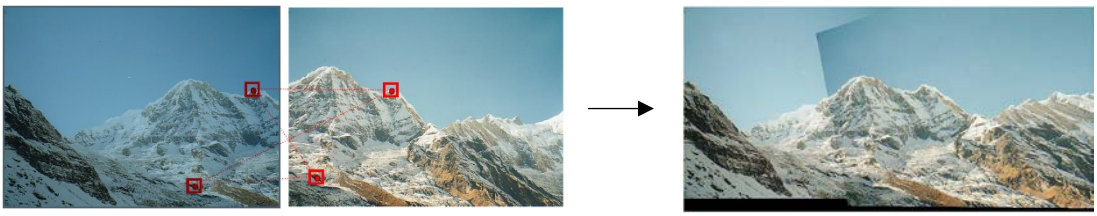
The 2 stereo cameras are related and we can scan the same row to find the corresponding point. That's the stereo correspondence problem. If we find the pixel in the same epipolar line, we can triangulate the 2 pixels and find the distance. With multiple features it's not so simple.

In general, setting of local invariant features is a very difficult problem because it's hard to find correspondences, but we have powerful tools to address the problem.
We take the image of the object in very different perspectives and with very different lighting conditions and this tool will work: very powerful.

A great variety of Computer Vision problems can be dealt with by finding corresponding points between two (or more) images of a scene. **Corresponding (aka homologous) points** are **image points which are the projection of the same 3D point in different views of the scene.**
Establishing correspondences may be difficult as homologous points may look different in the different views, e.g. due to viewpoint variations and/or lighting changes.

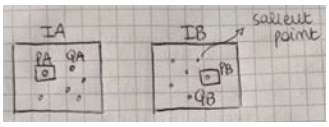
Example: Mosaicing
In order to create a larger image by aligning two images of the same scene, we need to:

- Align the two images by estimating a homography, which requires at least 4 correspondences (more is better)
- Find “salient points” independently in the two images
- Compute a local “description” to recognize salient points
- Compare descriptions to tell apart salient points



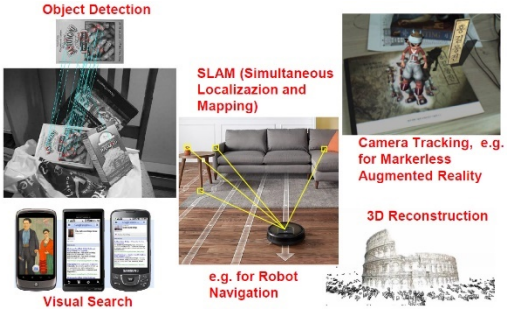
Let’s assume we find - according to some saliency functions - 4 salient points. Sometimes these are called the **fingerprint of the image** and they're used to recognise one image on the other image.
To perform the detection, the next step is matching: we need to find, given a salient point, the salient point correspondent in the other image.
The salient pixel in the left image maybe has a certain intensity, we can check if that pixel is in the right image. It isn’t a good approach because it is not good enough: maybe there are others pixel with the same intensity.

We establish matching based on the neighbourhood: we compare the neighbourhood of the salient point and we try to establish correspondences in the other image.
How do we compare the neighbourhoods? We don't compare the intensities, but **we compare functions of the neighbourhoods** which are called **local descriptors**.
So, we need a feature detector to find the salient points of the image and the local feature descriptors to compute a function on top of the image. Then, we find correspondences by doing the comparison in the descriptors space and not in the original image space.



N(PA)= neighbourhood of PA, N(PB) = neighbourhood of PB
Matching function called M_N(N(PA), N(PB)): we don't use this approach.
N(PA)→D(PA) and N(PB)→D(PB) where D(…) is the descriptor of …
We use the Matching function MD(D(PA), D(PB)) where D is the Domain of descriptors D:N→RK, a vector value function computer on top of the neighbourhood. E.g., in the SIFT, K=128.

If we compare directly the intensities, if the images have different light or different pov, we don't find any correspondence. On the contrary, if we describe the neighbourhood in a specific way, we can find the neighbourhood in the other image which have similar structure. So, we MUST use descriptors, not the intensities of pixels.



Other examples
In the Object Detection example, we have that the object is occluded: if an object is covered it is difficult to find salient/key points.
Also scaling changes and pov changes are relevant.

9.2 – THE LOCAL INVARIANT FEATURES PARADIGM

The task of establishing correspondences is split into 3 successive steps:

1. **Detection of salient points** (aka **key points**, interest points, features ...)
2. **Description/Computation of a suitable descriptor** based on a neighbourhood around a key point. This is a "magic function" that turns intensity into a K-dim space vector and throws away noise in large sense
3. **Matching descriptors between images**

The result should be **invariant (robust) to the many transformations that may relate images**.

9.2.1 – PROPERTIES OF GOOD DETECTORS/DESCRIPTORS

Properties of a good **detector**:

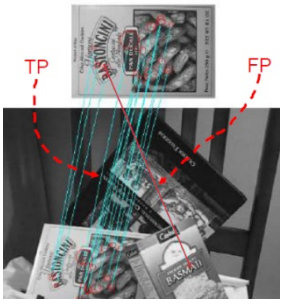
- **Repeatability**: it should find the same key points in different views of the scene despite the transformations undergone by the images
- **Distinctiveness/informativeness**: it should find key points surrounded by informative patterns of intensities, which would render them amenable to be told apart by the matching process. The points found by the detector should be really salient points and not normal points or uniform regions

Properties of a good **descriptor**:

- **Repeatability**: the descriptions computed at homologous points should be as similar as possible
- **Distinctiveness/informativeness**: the description algorithm should capture the salient information around a key point, so to keep important tokens and disregard changes due to nuisances (e.g., light changes) and noise
- **Compactness**: the description should be as concise as possible, to minimize memory occupancy and allow for efficient matching

We have two conflicting requirements: how much information about the neighbourhood has to be summarized in the descriptor? The more information we keep in, the more the descriptor is going to be a very due to nuisances. On one hand we want to throw away a lot of information to have a stable robust descriptor, but if we throw away information, we will not be able to match because the neighbourhood are too concise.

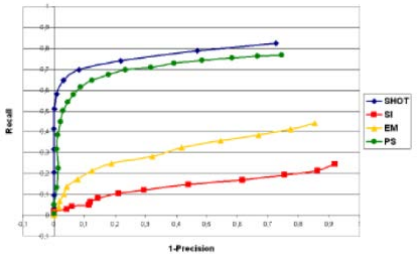
Speed is desirable for both, and **in particular for detectors**, which need to be run on the whole image, while descriptors are computed at key points only.



9.2.2 – PERFORMANCE OF THE MATCHING PROCESS

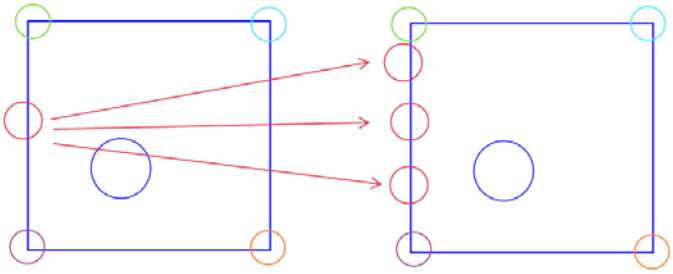
$$\text{Recall} = \frac{TP}{P}; \qquad \text{Precision} = \frac{TP}{TP + FP}$$

As long as one tries to gather more matches these become less precise.
Comparison across the whole range of operating points is achieved by **Precision (1-Precision) – Recall curves**.
Recall = number of true positives under the number of positives
Precision = true positive over all matches → how many of the matches are going to be connected
The ideal performance would be 100% recall and 100% precision. Obviously, this never happens and we have to trade between recall and precision, we cannot maximize both values.
In fact, MD(D(PA), D(PB)) → score, the higher the better. We compare with a threshold TH.
To optimize for Recall TH decreases, so the Recall increases. However, False Matches increase, so Precision decreases. **If Recall increases, Precision decreases and vice versa.**
These coloured curves take into account simultaneously precision and recall, two requirements we cannot maximize together.



INTEREST POINTS/CORNERS VS EDGES

Which one is the correspondent point? They all belong to edges. How can we distinguish them? We can consider the corners.



If we move in two perpendicular direction, we can find the non-ambiguous points.
In fact, in CV we use interest points and corner with the same meaning.

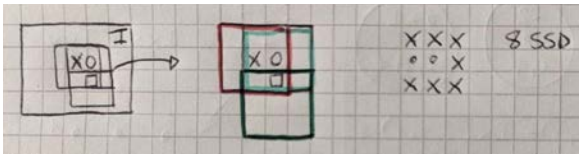
9.3 - MORAVEC INTEREST POINT DETECTOR

A major early proposal was due to Moravec:

$$C = min\left\{\sum_{i,j\in W} \left(I(i+m,j+n) - I(i,j)\right)^2\right\}; m,n \in \{-1,0,1\}, \neq (0,0)$$

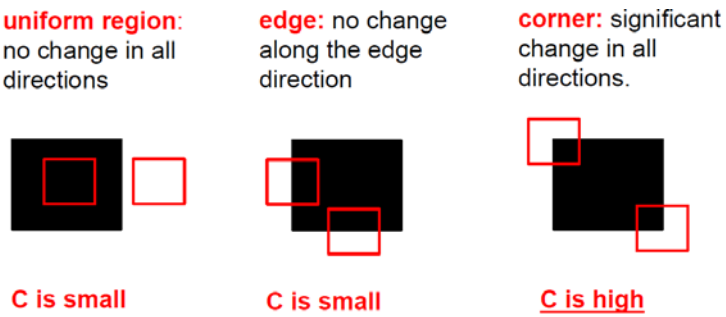
W: patch; $\left(I(i+m,j+n) - I(i,j)\right)^2$: SSD

First, let's do a consideration: if the min **SSD (Sum of Square Distances)** is high, the patch is dissimilar.
C = cornerness function: the min SSD(i,j) where i, j corresponds to each shift.
(In the formula, W is the patch we are considering.)



Based on the consideration, when the pixel will be an interest point? It has to be distinctive, so if we move around, we don't see similar patches, but different patches because it cannot be ambiguous. How can we validate this notion? We could pick another patch whose center is the pixel on the left with the X and the patch has the same size of the other one (7x7), but it has different properties. If the patches around are similar, we are in a uniform region so the point is not an interest point. There's a direction where patches are similar and there's a direction where patches are not similar.
Where this will be the case?

- **If the patch changes along one direction and not along the others directions, it's an edge**
- **If the patch doesn't change at all, it is in a flat region**
- **If the patch changes along two directions, it's a corner**



In the first image we can move the windows in all eight neighbouring patches and C is zero. That's why we are in a uniform region. In the second image C is still small because there's a direction where we don't find differences.

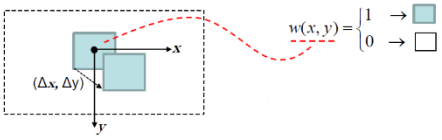
This approach is not so used because we have a quantization of many shifts, and if we rotate the image, that's not robust, it is not so good with noise. So, researchers find something better: the Harris corner detector.

9.4 – HARRIS CORNER DETECTOR

Harris & Stephens proposed to rely on a **continuous formulation of the Moravec's "error" function**. Denoted as **(Δx, Δy) a generic infinitesimal shift**, such a function may be written as:

$$E(\Delta x, \Delta y) = \sum_{x,y} w(x,y)\left(I\left(x+\Delta x, y+\Delta y\right) - I\left(x,y\right)\right)^2$$

The difference between patches is the error (E) when we shift by Δx, Δy. Rather than considering exactly patches, they consider the error function given globally.



Forgetting the w, the formula is I(...)-I: it's the whole image - the pixelwise. So, we have a window and we compute the difference sliding the window a little bit. This is really good because we have a global property of the image, but we looking at a local property. We are interested in assessing what happens around a pixel, so we insert the box w(x,y): if w is 7x7, then w(x,y) will be 1 in the 7x7 patch center (x,y), zero elsewhere.

$$I(x) \rightarrow I(x + \Delta x) = I(x) + Ix(x)\,\Delta x$$

where $Ix(x)\,\Delta x$ is the first derivative

Locally, we can approximate every complex function with Taylor Expansion.

$$I(x + \Delta x) - I(x) = Ix(x)\,\Delta x \rightarrow I(x, y) = I(x + \Delta x, y + \Delta y) = I(x, y) + Ix(x,y)\,\Delta x + Iy(x, y)\,\Delta y$$

where $Ix(x,y)\,\Delta x$ and $Iy(x, y)\,\Delta y$ are partial derivatives

$$I(x+ \Delta x, y + \Delta y)\, I(x, y) = Ix(x, y)\,\Delta x + Iy\,(x, y)\,\Delta y$$

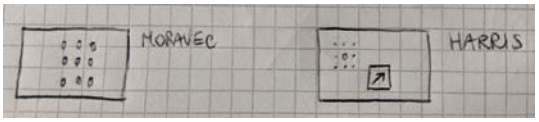
Due to the shift being infinitesimal, we can deploy Taylor's expansion of the intensity function at (x,y):

$$E(\Delta x, \Delta y) = \sum_{x,y} w(x, y)\left(I\left(x + \Delta x, y + \Delta y\right) - I\left(x, y\right)\right)^2$$

$$I\left(x + \Delta x, y + \Delta y\right) - I\left(x, y\right) \cong \frac{\partial I\left(x, y\right)}{\partial x}\Delta x + \frac{\partial I\left(x, y\right)}{\partial y}\Delta y = I_x\left(x, y\right)\Delta x + I_y\left(x, y\right)\Delta y$$

This is the most famous detector: this is used a lot, e.g., in OpenCV to find the corners of the chessboard when we calibrate. It is just an extension of Moravec's detector: it generalizes quite effectively the idea.

The key idea that improves performance significantly is to no longer consider a quantity of shifts, but to consider ALL shifts. So, it considers all possible infinite shifts: we cannot do every shift, but we formulate the problem in a continuous way.



To be more precise, if we take a pixel, we can consider only the 8 shifts around that pixel. Differently, in Harris, we consider a kind of infinitesimal shift and we formulate the cornerness function considering all these infinite shifts.

$$E(\Delta x, \Delta y) = \sum_{x,y} w(x, y)\left(I_x\left(x, y\right)\Delta x + I_y\left(x, y\right)\Delta y\right)^2 =$$

$$= \sum_{x,y} w(x, y)\left(I_x\left(x, y\right)^2\,\Delta x^2 + I_y\left(x, y\right)^2\,\Delta y^2 + 2\,I_x\left(x, y\right)I_y\left(x, y\right)\Delta x\Delta y\right)$$

$$= \sum_{x,y} w(x, y)\left(\begin{bmatrix}\Delta x & \Delta y\end{bmatrix}\begin{pmatrix} I_x\left(x, y\right)^2 & I_x\left(x, y\right)I_y\left(x, y\right) \\ I_x\left(x, y\right)I_y\left(x, y\right) & I_y\left(x, y\right)^2 \end{pmatrix}\begin{bmatrix}\Delta x \\ \Delta y\end{bmatrix}\right)$$

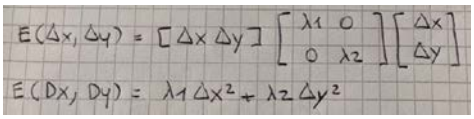
$$= \begin{bmatrix}\Delta x & \Delta y\end{bmatrix}\begin{pmatrix} \sum_{x,y} w(x, y)I_x\left(x, y\right)^2 & \sum_{x,y} w(x, y)I_x\left(x, y\right)I_y\left(x, y\right) \\ \sum_{x,y} w(x, y)I_x\left(x, y\right)I_y\left(x, y\right) & \sum_{x,y} w(x, y)I_y\left(x, y\right)^2 \end{pmatrix}\begin{bmatrix}\Delta x \\ \Delta y\end{bmatrix}$$

We are summing in a small window the square of the horizontal derivative (red) and the vertical derivatives (green).

At the end, we obtain:

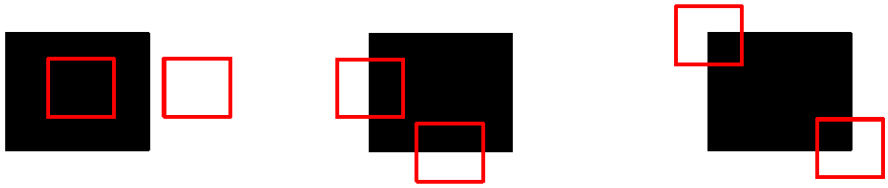
$$E(\Delta x, \Delta y) = [\Delta x \;\; \Delta y] M \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

M encodes the local image structure around the considered pixel. To understand why, let us hypothesize that M is a diagonal matrix:



$$M = \begin{pmatrix} \sum_{x,y} w(x,y) I_x(x,y)^2 & \sum_{x,y} w(x,y) I_x(x,y) I_y(x,y) \\ \sum_{x,y} w(x,y) I_x(x,y) I_y(x,y) & \sum_{x,y} w(x,y) I_y(x,y)^2 \end{pmatrix} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

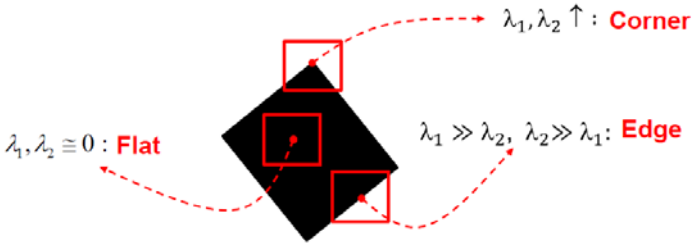
$\lambda_1, \lambda_2 \cong 0$: **Flat** $\quad \lambda_1 \gg \lambda_2$: **Edge** $\quad \lambda_1, \lambda_2 \uparrow$: **Corner**



- If λ1 and λ2 are small, whatever the x and the y, the error is small, so it doesn't matter Δx and Δy, whatever the direction: we are in a flat region
- If λ1 is large and λ2 is small, we have an edge
- If λ2 is large and λ1 is small, we have the perpendicular direction

Indeed, previous considerations have general validity as M is a real symmetric and thus can always be diagonalized by a rotation of the image coordinate system.
The columns of R are the orthogonal unit eigenvectors of M, λi the corresponding eigenvalues. RT is the rotation matrix that aligns the image axes to the eigenvectors of M.

$$M = R \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} R^T$$

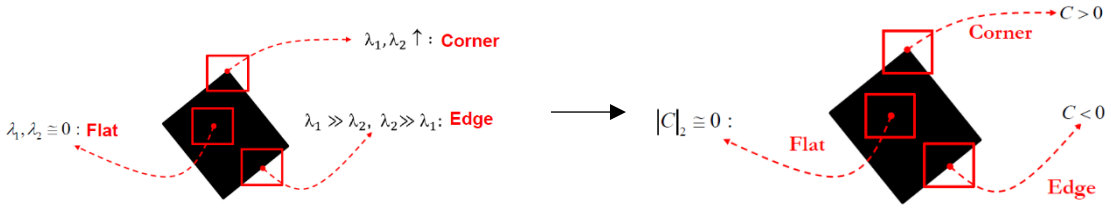


Computing the eigenvalues of M may be slow. Thus, Harris & Stephens propose to compute a more efficient "cornerness" function:

$$C = \det(M) - k \cdot \mathrm{tr}(M)^2 = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

This a function of the eigenvalues and the product of the eigenvalues is the det(M) and the sum is the same as the trace of the Matrix. The det of M is λ1* λ2, while the trace is λ1 + λ2.
They don't change if we apply a rotation: in Harris we compute det and tr(M) into this cornerness function.

Analysis of the above function would show that:

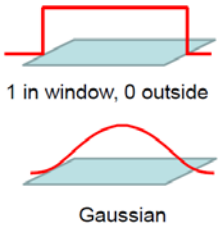


9.4.1 – HARRIS ALGORITHM

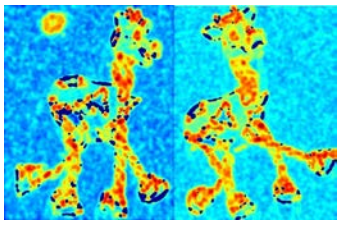
The **Harris corner detection algorithm** can thus be summarized as follows:

1. **Compute C at each pixel**
2. **Select all pixels where C is higher** than a chosen **positive threshold (T)**
3. **Within the previous set, detect as corners** only those pixels that are **local maxima of C**

It is worth highlighting that the **weighting function w(x,y)** used by the Harris corner detector is **Gaussian** rather than Box-shaped, so to **assign more weight to closer pixels** and less weight to those farther away.



Example



In the second image there's the cornerness function.

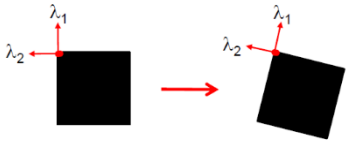
In the third image we applied thresholding to obtain only salient points.

In the last picture, we have fount the salient points of the original image.

9.4.2 - INVARIANCE PROPERTIES

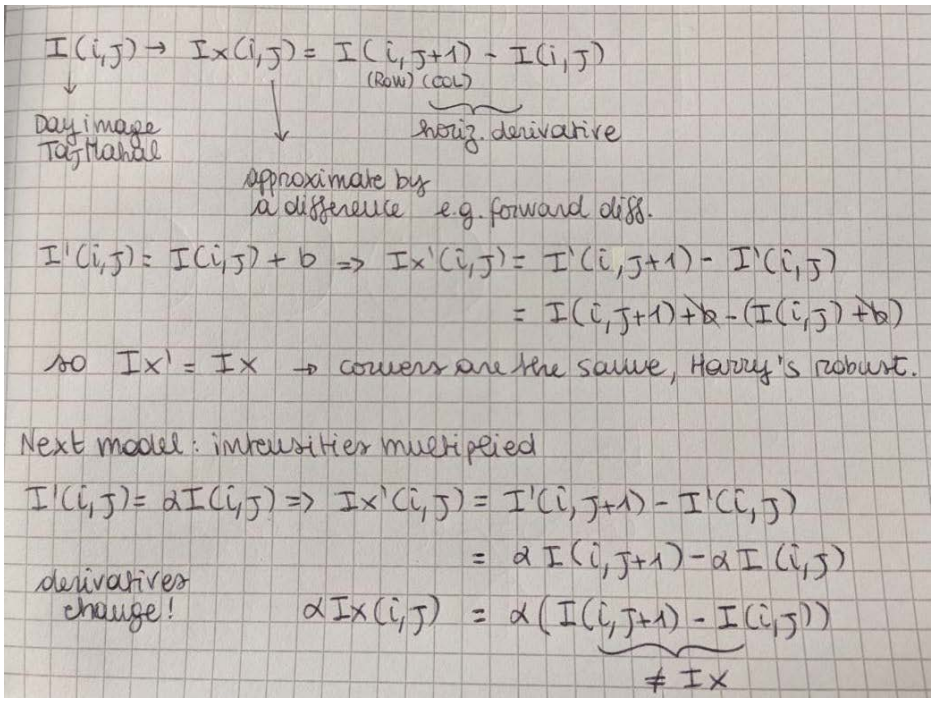
ROTATION

Rotation invariance? Yes, as the eigenvalues of M are invariant to a rotation of the image axes and thus so is Harris cornerness function.



Detectors try to find structures that are called interest points, that have a large variation along a perpendicular direction. These detectors find edges and corners.
Rotation, scale and invariance or at least robustness to brightness variation are important to consider.

The highest values don't change if we rotate the object. If we have intensity changes, is the Harris detector robust in this case? You may model the intensity changes in many different ways, so we cannot say yes or not. It depends on the model we use for the light change.
We cannot find a mathematical model of the intensity changes in reality world: the simple model is an additive bias [pron: baias] and in this case Harry is invariant.

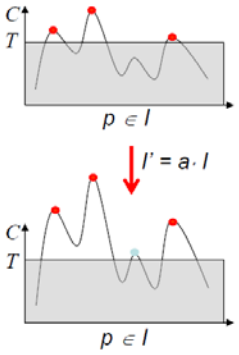


INTENSITY CHANGES

Invariance to Intensity changes? Limited.

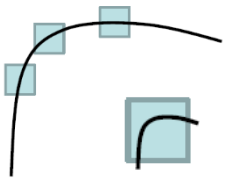
- **Yes**, to an additive bias (I' = I + b) due to the use of derivatives
- **No**, to multiplication by a gain factor (I' = aI), as derivatives get multiplied by the same factor

In case of multiplication by a gain factor for invariance to intensity changes, we can use the affine change model: I'(i,j) = a * I(i,j) + b
In practice, if we assume that the grey image on top is the 2D function, we have the corners above the T. If the image is multiplied by a scalar larger than 1, all intensities get higher and so we can find a corner which was not detected in the other image. The response of the Harris operator is a function of the intensities of the image.



SCALE INVARIANCE

Scale invariance? NO! Given the chosen size of the detection window, all the points along the border are likely to be classified as edges. However, should the object appear smaller in the image, use of the same window size would lead to detect a corner.

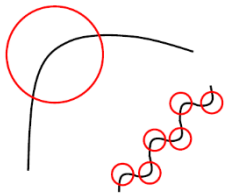


The use of a fixed detection window size makes it impossible to repeatably detect homologous features when they appear of different sizes in the images.

It took years to develop a real invariant feature to scale variance. Let's assume we have this object and we take three windows. In within the windows the derivatives are high, they are classified as

corners/interest points. However, these three windows are classified as edges because the variance is on one direction.
If we look instead at the same object but more far from the object or if we use a smaller focal lens, we have that the point could be classified as a corner.

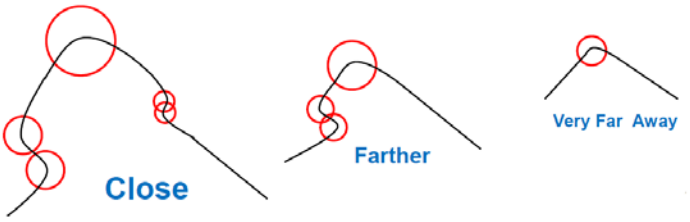
Typically, **an image contains features at different scales**, i.e., points that stand-out as interesting as long as a proper neighbourhood size is chosen to evaluate the chosen interestingness criterion.



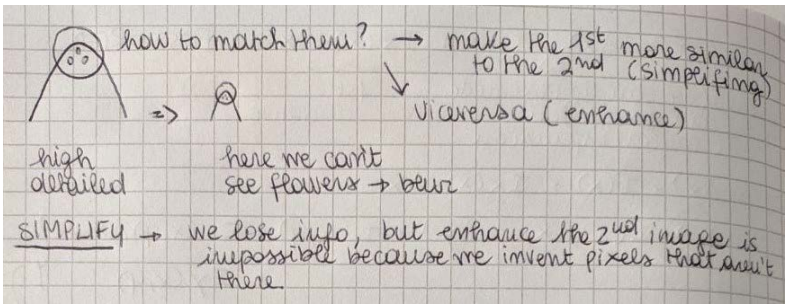
Thus, **detecting all features calls for a "tool" capable of analysing the image across the whole range of the scales deemed as relevant.**

Depending on the acquisition settings (distance and focal length) and object, it may look differently in the image, in particular it may exhibit more/less details (i.e. features).
Hence, features do exist within a certain range of scales.
The previously mentioned tool would thus allow for finding homologous features despite their showing-up at different scales: **the same feature would simply be detected at different steps within the multi-scale image analysis process**.

We need this tool to find all corners and to find correspondences in different images because we need to find the same features in many images.



Now, let's start reasoning also on matching the features. We have the three pictures of the mountain and we want the correspondence of the highest peak. We don't take the pixels and just compare the pixels: we compare the neighbourhood into a function called **descriptor** (which has a lot of value because it is a **vector function**). So, we compute a descriptor in every peak of the images and if they are similar, they are the same peak.



9.5 – GAUSSIAN SCALE-SPACE

9.5.1 – SCALE-SPACE

First of all, **the multi-scale tool should allow for representing the image at different scales**.
Several researchers, such as prominently, **Witkin and Koenderink**, have studied the problem and converged on the use of a function referred to as Scale-Space.
The Scale-Space is a one-parameter family of images created from the original one so that the structures at smaller scales are successively suppressed by smoothing operations. Moreover, one would not wish to create new structures while smoothing the images.

It can be shown that the **correct approach to obtain the Scale-Space** consists in smoothing the original image with larger and larger Gaussian kernels (or, equivalently, by solving the2D diffusion PDE over time starting from the original image).
So, Scale-Space is not one image, but many images obtained by changing the parameters. The first image would be the original, then we use a Gaussian σ, then a larger σ, then a larger σ, etc.

Representations dealing with larger and larger scales are obtained by convolution of the original image with larger and larger **Gaussian kernels**:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

Assumption: a scale space is understood to be a gaussian scale space, so we use always scale space and we intend Gaussian scale space.

The last image, the most blurred one, is the image that has the smallest derivative.

9.5.2 – FEATURE DETECTION & SCALE SELECTION

In itself, **the Gaussian Scale-Space** is only a tool to represent the input image at different scales. However, it neither **includes any criterion to detect features nor to select their characteristic scale**.
As features do exist across a range of scales, the latter concept deals with understanding at which scale a feature turns out maximally interesting and should therefore be described.
The fundamental research work on multi-scale feature detection and automatic scale selection is due to **Lindberg**, who proposed to compute suitable **combinations of scale-normalized derivatives of the Gaussian Scale-Space (normalized Gaussian derivatives) and find their extrema**.

Other authors are known as responsible of the Gaussian scale space, but also Lindberg did a great job in this research. The paper of Lindberg is a mix of theory and intuition, is not a truly mathematical paper. At the end, he was capable to proof that if we compute combinations of scale normalized derivatives, such as the Laplacian, on a scale-space, we obtain the multi scale features.
Scale normalized means that we have to multiply the derivative to a factor multiple to scale. If we move to highest scales, the derivatives are smaller and if we take derivatives in different scale, we'll have derivatives not comparable, that's why we normalize.

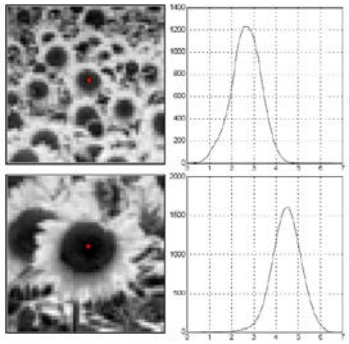
9.5.3 – SCALE-NORMALIZED LOG

Between Lindberg's proposed functions is the **scale-normalized Laplacian of Gaussian (LOG).**
LOG = Laplacian applied on a smoothed image with Gaussian. Now we apply Laplacian to the Gaussian scale-space. L(x,y,σ) is a Gaussian with a certain σ convolution the image, the inverse is the Laplacian of that. The σ^2 is the scale normalization factor.

$$F(x, y, \sigma) = \sigma^2 \cdot \nabla^2 L(x, y, \sigma) = \underbrace{\sigma^2 \cdot}_{\text{scale normalizer}} (\nabla^2 L(x, y, \sigma) * I(x, y))$$

Let's consider the center of the flower, the output of the scale normalized LOG for different σ is on the right: there's a maximum at a certain scale position. If we change the scale of the image, and we still look at the center of the flower, we find the same behaviour, but the maximum is on a larger scale.

The scale at which we find a maximum of the LOG is a function of the features in the image.
What's amazing is that **if we take the ratio between the 2 scales (looking at the peak) we get exactly the ratio between the size of the flowers, so the scale ratio between these 2 images.**

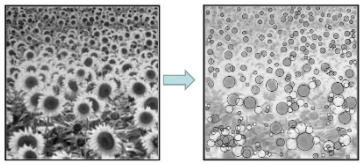


Considering the centers (red points) of the two dark blobs, it can be observed that the extremum of the scale-normalized LOG is found at a larger scale for the larger blob.

The ratio between the two characteristic scales is roughly the same as the ratio between the sizes (diameters) of the two blobs.

MULTI-SCALE FEATURE DETECTION

Features (blob-like) and scales detected as extrema of the scale-normalized LOG.



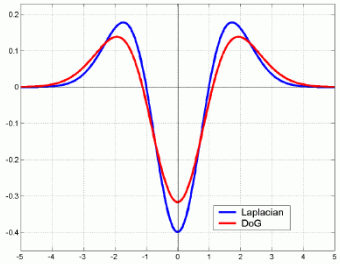
9.6 – DIFFERENCE OF GAUSSIAN (DOG) DETECTOR

Lowe proposes to detect key points by seeking for the extrema of the **DoG (Difference of Gaussian)** function across the (x,y,σ) domain:

$$DoG(x,y,\sigma) = \left(G(x,y,k\sigma) - G(x,y,\sigma)\right) * I(x,y) = L(x,y,k\sigma) - L(x,y,\sigma)$$

This approach provides a computationally efficient approximation of Lindberg's scale-normalized LOG:

$$G(x,y,k\sigma) - G(x,y,\sigma) \approx (k-1)\sigma^2\left(L_{xx} + L_{yy}\right)$$

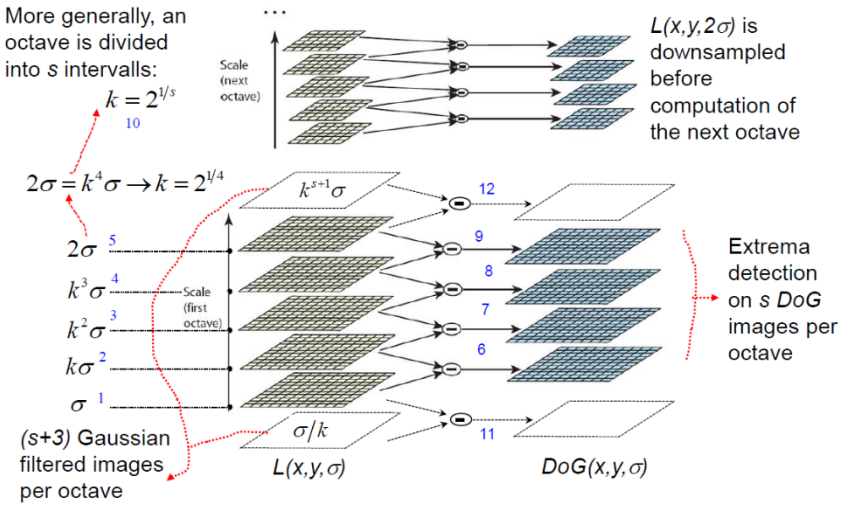


As **(k -1) is a constant factor**, it does not influence extrema location. As such, the choice of k is not critical.
Both detectors are rotation invariant (circularly symmetric filters) and find blob-like features.

Here there's the real practical implementation: the feature detector is called DOG and it's built in a way that is really easy and fast to compute on a given image. We find features not by using the scale normalized LOG, but by using another detector called DOG (Difference of Gaussian) that uses the difference between two gaussians related in that sigma by a constant factor key.
Lowe proved that **DOG is proportional to scale normalized LOG**, due to this k-1 factor. We don't care about the values of the LOG, we are interest on the extrema, so we can focus on these values.
This approach is intuitively better and more efficient because, assuming to apply Lindberg's theory, we find the features according to his method.
Indeed, we compute the descriptors and they have to be computed at the scale at which we have found the feature.
We should use the simplified image by Gaussian blur, not the original one. With the approach proposed by Lowe, on the contrary, we use the difference between nearby gaussians to find features and then we use the already computed images to find the descriptor.

9.6.1- COMPUTATION

Starting from the bottom, let's follow the numbers in blue:



1. We start with a certain sigma, then we smooth with another sigma up to a certain number of smoothed images. We do that for a range that goes from σ to 2σ and this range is called "**octave**". Looking at 6, 7, 8, 9, the light blue images are respectively the DOGs at $\sigma, k\sigma, k^2\sigma, k^3\sigma$.

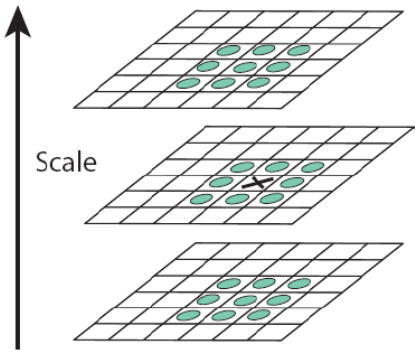
11. Now we can compute the DOG comparing pairs of images. To find extrema we are ok, but we cannot find extrema at σ and at 2σ... so, to find the extrema we should take another gaussian filtered image on the bottom and on the top. But what kind of sigma should we use? We use σ/k on the bottom and $k^{s+1}\sigma$ on the top.

10. We know how many scales we want to sample and then we can compute K, so we know how to smooth. **The more we have samples, the more k is smaller.**
With the stack of DOG images, we have a fast alternative to the LOG to do extrema selection. So, we look through the DOGs and we found the extrema. How many scales can we probe to find features with this set of gaussian-filtered images we have computed? Where could we find features?

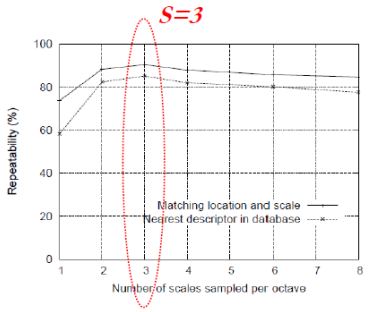
Another octave would go from 2σ to 4σ and σ begins to be very large. It means that we have a big filter, so lot of computation!
If we want to start a new octave at a double σ, we can resize the image and make it smaller. In fact, if we squeeze the image, we have the same stack of filters, but the image has changed.
So, in practice, **we have only one stack**! That's another way to simplify the calculation and to have a faster algorithm.

9.6.2 – KEY POINT DETECTION AND TUNING

Key point detection → Extrema detection: a point (x,y,σ) is detected as a key point iff its DoG is higher (lower) than that of the 26 neighbours (8 at the same scale and 18=9+9 at the two nearby scales) in the (x,y,σ) space.



Parameter Tuning: σ=1.6 (initial σ at each octave). The input image is enlarged by a factor of 2 in both dimensions.



9.6.3 - ACCURATE KEY POINT LOCALIZATION

To localize key points more accurately, the DoG function can be approximated around each extremum by its **second-degree Taylor expansion:**

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

Displacement wrt the (x,y,σ) position of the found extremum

DoG at the found extremum

Gradient of the DoG function at the found extremum

Hessian matrix of the DoG function at the found extremum

The new extremum (either maximum or minimum) can be localized by imposing the **gradient of the approximated function to be zero:**

$$\hat{\mathbf{x}} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1} \frac{\partial D}{\partial \mathbf{x}}$$

If the displacement $\hat{x}$ turns out large (>0.5 in at least one dimension), the extremum is re-assigned to the closest discrete position and the approximation computed again. Once the displacement is small, the procedure is stopped and the found displacement provides sub-pixel interpolation with respect to the last discrete position.

9.6.4 – PRUNING OF UNSTABLE KEY POINTS

Extrema featuring weak DoG responses turn out **scarcely repeatable**. They can be pruned by estimating the DoG at a found extremum $\hat{x}$ and thresholding its magnitude.

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2}\frac{\partial D}{\partial \mathbf{x}}^T \hat{\mathbf{x}}. \qquad |D(\hat{\mathbf{x}})| < T_{DoG}$$

Lowe also notices that unstable keypoint featuring a sufficiently strong DoG may be found along edges and devises a further pruning step based on the eigenvalues of the Hessian, which are proportional to the principal curvatures of the intensity surface.

Thus, **the higher is the ratio between the larger and smaller eigenvalues, r, the more similar to an edge rather than a blob is the considered pixel.** Similarly to Harris', explicit computation of the eigenvalues may be avoided.

larger eigenvalue

$$\frac{\mathrm{Tr}(\mathbf{H})^2}{\mathrm{Det}(\mathbf{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta^2} = \frac{(r+1)^2}{r} > T_{edge}$$

smaller eigenvalue

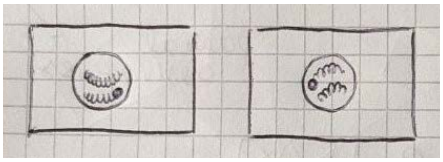
This increasing function of r can be thresholded to prune edges

Example

In the first image we have the DoG extrema, in the second one we have the keypoints after pruning weak responses and in the third one we have the final keypoints after pruning the majority of those located along edges, not all.

9.6.5 - SCALE AND ROTATION INVARIANT DESCRIPTION

Most keypoint detection algorithms follow an approach akin to Lowe's (i.e., finding extrema across a stack of images computed while increasing a scale parameter).
Then, **once each keypoint has been extracted, a surrounding patch is considered to compute its descriptor.** A desirable property concerns such descriptor being scale and rotation invariant.
To attain scale invariance, the **patch is taken from the stack image**, i.e., L(x,y,σ) in Lowe's, that correspond to the characteristic scale.
To attain rotation invariance, a characteristic (aka **canonical) patch orientation is computed**, so that the descriptor can then be computed on a canonically-oriented patch.



We compute descriptors at those points. The 2nd image is rotated, but rotation is invariant? We need to normalize neighbourhood to make descriptors compute the same function. We turn the axis according to the structure of that neighbourhood.

9.7 – SIFT (canonical orientation)

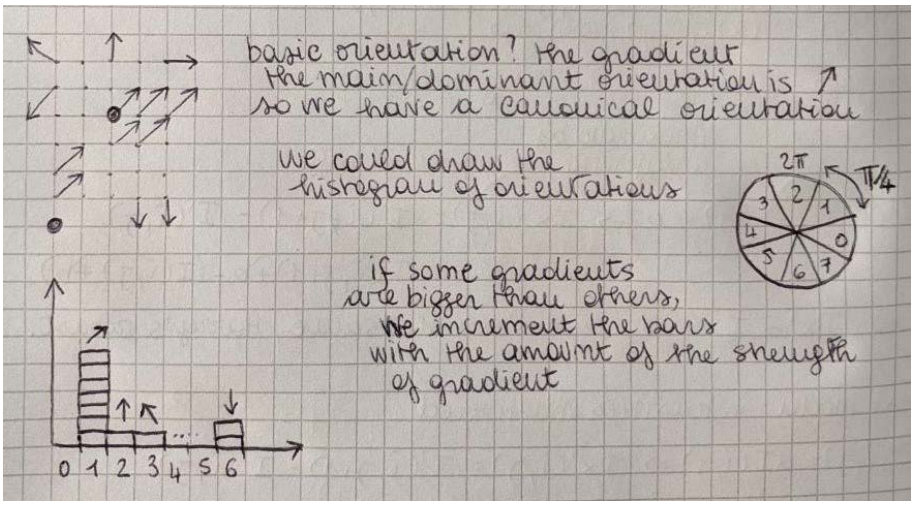
Lowe proposes to compute a descriptor referred to as **SIFT (Scale Invariant Feature Transform)** at each DoG keypoint.
In SIFT, the **characteristic orientation** is computed as follows:

- **Given the keypoint**, the **magnitude and orientation of the gradient are computed at each pixel** of the associated Gaussian-smoothed image, L:

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

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

- Then, **an orientation histogram** (with bin size equal to 10°) **is created** by accumulating the contributions of the pixels belonging to a neighbourhood of the keypoint location
- The contribution of each such pixel to its designated orientation bin is given by the **gradient magnitude weighted by a Gaussian** with $\sigma = 1.5 \cdot \sigma_s$, with $\boldsymbol{\sigma_s}$ denoting the scale of the keypoint

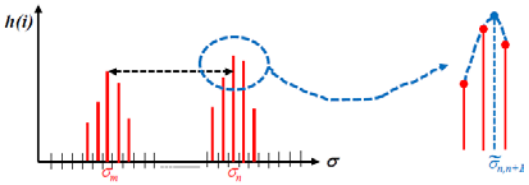


In reality we don't have 8 pieces of the pie, but more or less 36 because each bin is equal to 10°.

We know that if we have fond a key point at a specific scale, this point will appear at a lower scale but not a larger scale.

The characteristic orientation of the keypoint is given by the highest peak of the orientation histogram. Moreover, other peaks higher than 0.8 of the main one would be kept as well.
Accordingly, **a keypoint may have multiple canonical orientations** and, in turn, multiple descriptors sharing the same location/scale with diverse orientations. This has been found to occur quite rarely (~15% of the keypoints), though.
Finally, **a parabola is fit in the neighbourhood of each peak** to achieve a more accurate estimation of the canonical orientation. Purposely, the two adjacent bins to the found peak are considered.

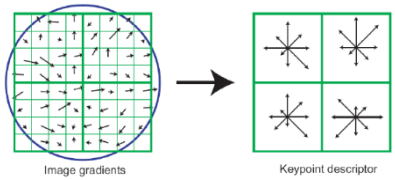
$\sigma_i = i \cdot \Delta\sigma = i \cdot (\pi/18)$



In the figure, we pick both highest points: we keep the two main orientations to calculate two descriptors.

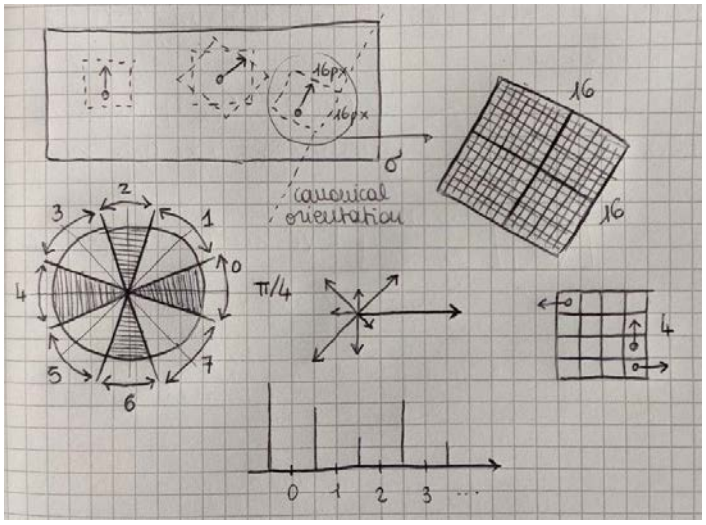
9.7.1 – SIFT DESCRIPTOR

Given a detected keypoint (x,y,σ), **a 16x16 pixel grid** around its location in the associated Gaussian-smoothed image, L(x,y,σ), is considered. **The grid is further divided into 4x4 regions** (each of size 4x4 pixels) and **a gradient-orientation histogram is created for each region**. Gradients are rotated according to the already computed canonical orientation of the keypoint.

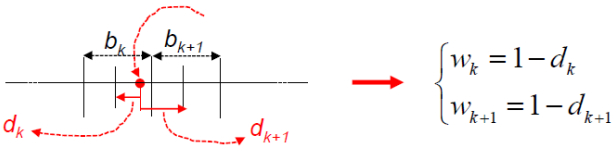


Each histogram has 8 bins (corresponding to an **angular step of 45°).** Each pixel in the region contributes to its designated bin according to the gradient magnitude as well as to a Gaussian weighting function centred at the keypoint location (with σ equal to half the grid size). Therefore, **the SIFT descriptor has size equal to the number of regions times the number of histogram bins per region**, i.e., 4 x 4 x 8 = 128 elements.

At the end, the whole region is divided in 16 pixels, 16 regions. For each of this 4x4 subregions, it computes a descriptor and a very good descriptor is again a histogram of gradients orientations. Gradient orientations capture the directions and that is the most essential information. There will be one orientation histogram for each of these 16 subregions (in the image it shows only 4, but we know they are 16).



SIFT is biologically inspired. There exist studies suggesting that neurons in the primary visual cortex (V1) do match gradient orientations robustly with respect to a certain degree of shift of the input pattern for recognition purposes.
To avoid boundary effects, a soft rather than hard assignment is employed in SIFT, whereby the contribution to two adjacent bins is weighted by the distance to the bin center.



This is done within a histogram as well as between regions (the contribution is spread bilinearly between 4 adjacent regions). Hence, **the overall scheme is referred to as trilinear interpolation**.

The descriptor is normalized to unit length to gain invariance wrt affine intensity changes. Then, to increase robustness to non-linear changes, all elements larger than 0.2 are saturated and the descriptor normalized again.

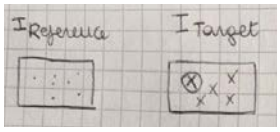
Let's assume that there's a gradient with a certain direction. According to a hard assignment, all amounts go to 0, but I can also assign a gradient which is between 0 and 1, half to 0 and half to 1.
So, soft assignment is preferable: we always quantize because we cannot have infinitive precision.

9.8 – MATCHING PROCESS

Descriptors (e.g. SIFT) are compared across diverse views of the same subject matter to find corresponding keypoints.
This is a classical **Nearest Neighbour (NN) Search problem**:

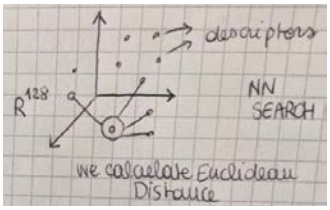
Given a set S of points in a metric space M and a query point q ∈M,
find the closest point in S to q.

Without loss of generality, we assume that we wish to match the local features computed at run time from an image under analysis, aka **target image (T)**, to those already computed from a **reference image(R)** or, equivalently, a set of reference images.



As for each feature in T we look for the most similar one in R:

- The **features in T represent query points, q**
- The **features in R provide set S**
- The **metric space, M**, is the **descriptor space endowed by a distance function**. When matching SIFT descriptors, $M=R^{128}$, the **distance function** being **typically the Euclidean distance**



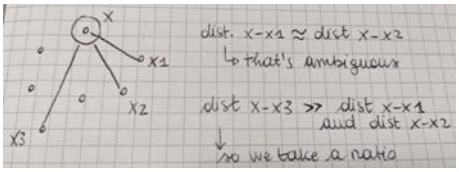
9.8.1 – VALIDATING MATCHES

The found NN does not necessarily provide a valid correspondence as **some features in T may not have a corresponding feature in R**.
Generally, this is due to clutter and/or viewpoint changes. Hence, it turns out **mandatory to enforce criteria to accept/reject a match found by the NN search process**.
In practice, we find the distance from the nearest neighbour and the second nearest neighbour.

Two such criteria are as follows:

1. $d_{NN} \leq T$ (NN distance)
2. $\frac{d_{NN}}{d_{2-NN}} \leq T$ (ratio of distances → Lowe)

Lowe shows that T=0.8 may allow for rejecting 90% of wrong matches while missing only 5% of those correct.

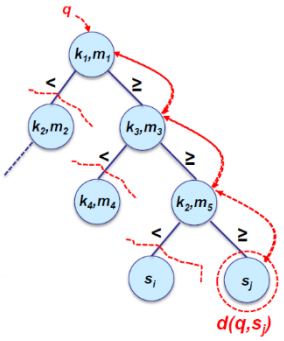


9.8.2 – EFFICIENT NN-SEARCH

Exhaustively searching for the NN of the query feature, q, has linear complexity in the size of S, which turns out often exceedingly slow.
Thus, **efficient indexing techniques** borrowed from database management and information retrieval **are deployed to speed-up the NN-search process**.
The **main indexing technique** deployed for feature matching is known as **K-D TREE**. In particular, the preferred approach is the **approximate variant referred to as BBF (Best Bin First)**. Indeed, unlike the basic k-d tree, the BBF formulation is efficient also in high-dimensional spaces such as descriptor spaces (e.g., R^{128} for SIFT).

K-D TREE

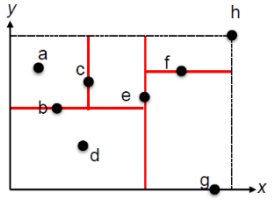
The **k-d tree generalizes the binary search tree** to the case of k-dimensional data. Each node defines a split of the data according to one of the k dimensions.
During the search, the query point, q, traverses the tree from the root down to a leaf according to the splits defined by nodes. The datum stored in the leaf is close to q, though not guaranteed to be the NN.

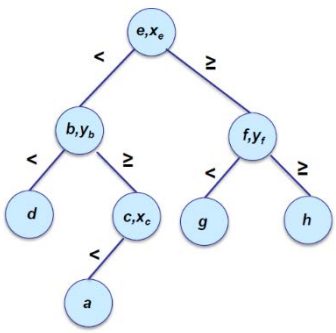


Therefore, **the tree is traversed back from the leaf to the root to refine the search**. During backtracking, only the sub-trees that may contain closer data are explored.
If the tree is balanced, the **initial search** requires **$\log_2$(size(S)) comparisons**, which yields a large speed-up as long as backtracking involves visiting relatively few nodes.

In practice, we build an index: each node defines a split of the data in two subspaces based on the value of one of the dimensions. E.g., I have 10 dim and I split the subspaces in under than 10 and more than 10.

The data are always split by choosing the dimension showing the highest variance. The datum taking the median value of such dimension defines the split.
Thus, **the root is created first**. This splits the initial set of data, S, into two sub-sets, S1 and S2, of, roughly, the same sizes. The process is then applied recursively.
The k-d tree partitions the data space into bins adaptively: **bins are smaller in higher density regions, larger in regions of lower data density**.

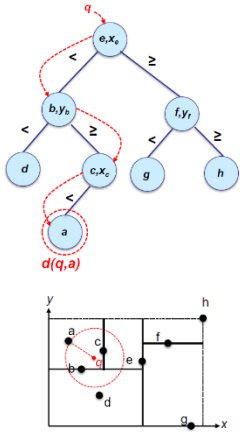




As the (hyper)sphere centred at q with radius d(q,a) does intersect the (hyper)planes separating the partition of the space associated with leaf a from its adjacent partitions, it may be necessary to explore these partitions (together with their parent nodes) to find the NN of q.

1. Given the query, *q*, the tree is traversed from the root to the closest leaf, denoted here as NN_{curr} , the associated distance given by $d(q, NN_{curr})$.

2. Backtracking:
   - The parent node to the found leaf is considered. If its distance to *q* is less than $d(q, NN_{curr})$, the parent node becomes the new NN_{curr} and $d(q, NN_{curr})$ gets updated accordingly.
   - A check is carried out to determine whether the other half-space defined by the current parent node may contain or not a closer point. This depends on whether or not the hypersphere of radius $d(q, NN_{curr})$ centred at *q* intersect the splitting hyperplane.
     - Should the former be the case, the sub-tree whose root is the parent node is traversed down to the next closest leaf.
     - Conversely, the parent node to the current one is considered.
   - The process ends upon getting to the root of the tree, with NN_{curr} turning out the NN to *q*.

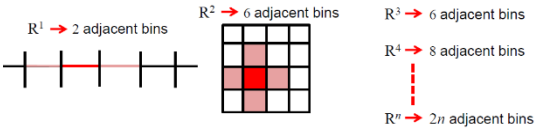


THE CURSE OF DIMENSIONALITY

A k-d tree may be thought of as partitioning the space into "bins". During backtracking, the bins adjacent to that containing the found leaf might be examined to find the NN.

However, **as the dimensionality of data increases, so does the number of bins adjacent to a given one**, which renders **k-d trees inefficient** (perhaps even slower than exhaustive search) **with high-dimensional data.**

The issue comes from the inherent structure of the R^n space, since it is hardly addressable.



9.8.3 – APPROXIMATE SEARCH (BEST BIN FIRST)

Beis & Lowe have proposed an approximate search algorithm, referred to as **Best Bin First (BBF)**, which deploys a **priority queue** and **limits the maximum number of reachable leaves (Emax).**

1. As with the standard algorithm, given the query, *q*, the tree is traversed down to the closest leaf, so to find the initial NN_{curr} and associated distance ($d(q, NN_{curr})$). Moreover, though, the visited nodes are ordered in a *priority queue* according to their distance to *q*.

2. Backtracking:
   - The first element in the *priority queue* is extracted so to traverse its unexplored branch down to a leaf. During the process, the *priority queue* is always kept updated.
   - The previous step is iterated until a fixed number of leaves (E_{max}) has been reached.

The found *approximate* NN is the closest data point (NN_{curr}) after Emax leaves have been visited.
So, the BBF is much faster and reasonably accurate.

9.8.4 – OTHER MAJOR PROPOSALS

A fast alternative to SIFT is the well-known **SURF (Speeded-Up Robust Features**) algorithm. Blob-like features are detected through efficiently computable filters inspired by Lindberg's Gaussian

derivatives. Likewise, the scale-space is achieved by efficient mean filtering rather than Gaussian smoothing and the descriptor is based on computing responses to fast **Haar filters**.

MSER (Maximally Stable Extremal Regions) detects interest regions of arbitrary shapes, in particular approximately uniform areas either brighter or darker than their surroundings. Description of MESR region may then be carried out by SIFT.

FAST (Features from Accelerated Segment Test) is a very efficient detector of corner-like features.

A variety of binary descriptors, such as **BRIEF, ORB and BRISK**, have been proposed to minimize memory occupancy and accelerate the matching step thanks to the use of the Hamming distance.

10 - OBJECT DETECTION

The object detection problem occurs in countless applications and can be formulated as follows:

Given a reference image (aka model image) of the sought object, determine whether the object is present or not in the image under analysis (aka target image) and, in case of detection, estimate the pose of the object.

Depending on the application, the pose may often be given by a translation, a roto-translation or a similarity (roto-translation plus scale).
We want to find, in a small reference image, a model of the object and control if it's found in the target image. It is the case of many industrial applications. Is this object part of the image? Yes, but we need other information: we need the pose, such as the translation, the rotation and the scale.
This has to be fast because is typically a step of a process and then other computation will follow.

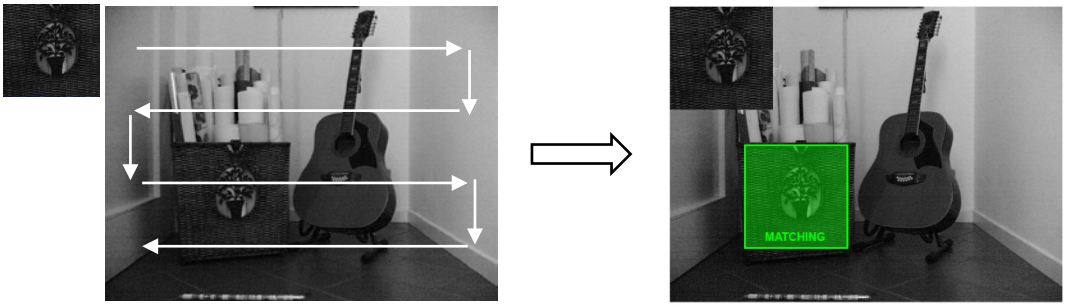
The problem has indeed a number of diverse facets, such as seeking to **detect a single/multiple instances of the model** as well as either a **single/multiple models**. For the sake of simplicity, and without much loss of generality, hereinafter we will refer mainly to the **basic "single-model/single instance" problem.** Generalization to any of the other variants turns out more often than not straightforward.

Typical nuisances to be dealt with are **intensity changes, occlusions and clutter**.
Moreover, **computational efficiency** is a major requirement in most practical applications. In particular, it is harder to achieve efficiency as long as the dimensionality of the pose space and/or the number of models gets larger.
Many different object detection approaches have been proposed in the computer vision literature. Here, we will focus on:

- **Template matching (aka pattern matching)**
- **Shape-based matching**
- **The Hough Transform**
- **Detection by local invariant features**

10.1 – TEMPLATE MATCHING

The model image is slid across the target image to be compared at each position to an equally sized window by means of a suitable (dis)similarity function.



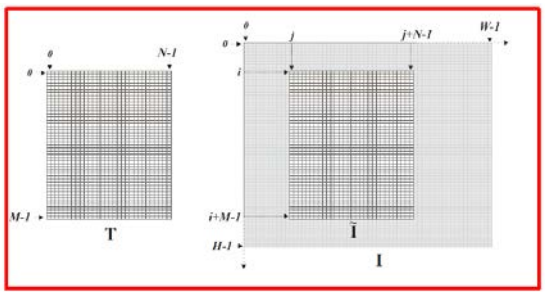
10.1.1 – DISSIMILARITY FUNCTIONS

We can maximize a similarity function or minimize a dissimilarity function.
Both $\tilde{I}$(i, j) (the window at position (i, j) of the target image having the same size as T) as well as T can be thought of as M·N-dimensional vectors. Accordingly, the **SSD (Sum of Squared Differences) represents the squared L2 (Euclidean) norm of their difference.**

$$SSD(i,j)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left(I(i+m,j+n)-T(m,n)\right)^2$$

$$SAD(i,j)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left|I(i+m,j+n)-T(m,n)\right|$$

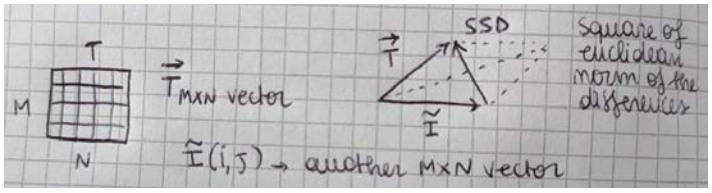
$$NCC(i,j)=\frac{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}I(i+m,j+n)\cdot T(m,n)}{\sqrt{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}I(i+m,j+n)^2}\cdot\sqrt{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}T(m,n)^2}}$$



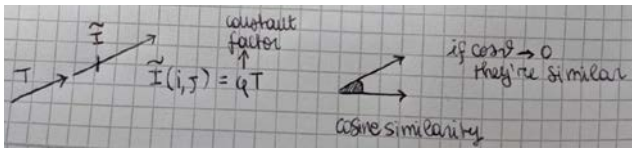
The **SAD (Sum of Absolute Differences)** represents the L1 norm of the difference between vectors $\tilde{I}$(i,j) e T, while the **NCC (Normalized Cross-Correlation)** represents the cosine of the angle between vectors $\tilde{I}$(i,j) e T.
The NCC is **invariant to linear intensity changes**:

$$NCC(i,j)=\frac{\tilde{\mathbf{I}}(i,j)\cdot\mathbf{T}}{\|\tilde{\mathbf{I}}(i,j)\|\cdot\|\mathbf{T}\|}=\frac{\|\tilde{\mathbf{I}}(i,j)\|\cdot\|\mathbf{T}\|\cdot\cos\theta}{\|\tilde{\mathbf{I}}(i,j)\|\cdot\|\mathbf{T}\|}=\cos\theta \longrightarrow \tilde{\mathbf{I}}(i,j)=\alpha\cdot\mathbf{T}$$

If we look at the image, the SAD is the same of the SSD, but if we change the template, the best match is given by a gain factor * T. What would be the better? How should we draw the $\tilde{I}$ vector?



We prefer to use this other function which is the cosine and it is called **NCC**. If $\tilde{I}$ and T, the numerator, is a sum of product of corresponding elements, it is a dot product between two vectors and the denominator is the product of the norms of two vectors.



So, it is **also called COSINE SIMILARITY** and it is good because it is slower, but it is invariant to linear intensity changes. It's a trade-off, it depends on what kind of conditions we have: if we don't have variances of intensity, we may use SAD or SSD.

The **ZNCC (Zero mean NCC)** is computed after subtraction of the means:

$$I(i+m,j+n)\rightarrow\left(I(i+m,j+n)-\mu(\tilde{I})\right) \qquad T(m,n)\rightarrow\left(T(m,n)-\mu(T)\right)$$

Where:

- $\mu(\tilde{I})$ = **Zero-Mean Normalized Cross-Correlation**
- $\mu(T)$ = **Correlation Coefficient**

$$\mu(\tilde{I})=\frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}I(i+m,j+n) \qquad \mu(T)=\frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}T(m,n)$$

Looking more in deep at the means used to compute the ZNCC:

$$I(i+m,j+n) \rightarrow \left(I(i+m,j+n)-\mu\left(\tilde{I}\right)\right) \qquad T(m,n) \rightarrow \left(T(m,n)-\mu(T)\right)$$

$$ZNCC(i,j)=\frac{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left(I(i+m,j+n)-\mu\left(\tilde{I}\right)\right)\cdot\left(T(m,n)-\mu(T)\right)}{\sqrt{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left(I(i+m,j+n)-\mu\left(\tilde{I}\right)\right)^2}\cdot\sqrt{\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left(T(m,n)-\mu(T)\right)^2}}$$

$$\rightarrow \tilde{\mathbf{I}}(i,j)=\alpha\cdot\mathbf{T}+\beta$$

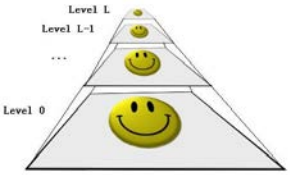
(Invariant to affine intensity changes)

Example

The pictures are made much brighter. Using those dissimilarity functions that change a lot when we have a light change, like the SSD and the SAD, we have a very wrong result and we cannot localize the template. So, the SSD and the SAD are not robust.
Conversely, if we use ZNCC or NCC, they are successful also when the lights become very different.
In the last very difficult case, in which we cannot see almost anything, SSD and SAD fail, ZNCC continues to find the template correctly and the NCC is not providing a wrong result.
So, based on experience, ZNCC is the most robust method.

10.1.2 – FAST TEMPLATE MATCHING

Template matching may turn out exceedingly slow whenever the model and/or target images have a large size (i.e., computational complexity is O(M×N×W×H)).
Development of fast template matching algorithms is therefore an active research topic. A **popular approach is to deploy an image pyramid**.



If we squeeze the image, the program will run very faster, so in general we squeeze the image in a smaller size: this is very effective and using this approach means that we're using a pyramid.
We create a stack with the parameter 2. At the top we have the more significant image, then we do the same with the template and we apply our algorithm.
Finally, we plan the result and we perform a search at the higher resolution (so with the largest template and the largest image).

Very fast approach, though the **number of levels needs to be chosen carefully** (and empirically) to avoid loss of information detrimental to detection:

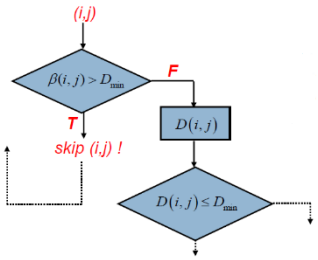
- Smoothing and sub-sampling (typically 1/2 on both sides at each level)
- Full search at top level and then local refinements traversing back the pyramid down to original image

Here's a taxonomy of the most famous fast template matching approaches:

- **Approximate Methods**: the **result is not guaranteed** to be the same as would be provided by a full-search.
  - **Image Pyramid**: Full-search at top level, then local refinement throughout successive levels down to the original full-size image
  - **Sub-Templates**: A relatively small sub-template (e.g., a salient region or a set of salient points) is sought first in order to highlight a set of candidate positions, then the full-size comparison is carried out at these positions only.
- **Exact (aka Exhaustive) Methods**: the **result is guaranteed** to be the same as would be provided by a full-search.
  - **Call-out with dissimilarity functions (e.g., SAD-SSD):** computation of the function at the current position is terminated as soon as its value gets higher than the current minimum (or a threshold).
  - **Matching in the Fourier domain (FFT).**
  - **Deployment of efficiently computable bounds of the (dis)similarity function** to skip candidates which cannot improve the current degree of matching.

10.1.3 – BOUND-BASED METHODS

Let D(i,j) be the dissimilarity function to be minimized to detect T and let us assume that there exists a **lower bound of this function**: β(i, j) ≤ D(i, j).
The final outcome is the same as would be provided by a full-search!



The approach is **faster than a full-search** provided that β(i,j) can be calculated much more efficiently than D(i,j) (efficient bound) and it turns out **enough accurate to skip the computation of D(i,j)** at a large number of image positions (effective bound).
In case of template matching based on similarity functions, β(i,j) must be an upper bound: β(i,j) ≥ S(i,j).
Efficient and effective bounds have been proposed for SAD, SSD, NCC and ZNCC.

10.1.4 - SUCCESSIVE ELIMINATION ALGORITHM (SEA)

SEA was originally proposed (Lee & Salari) for the **block matching step** carried out to **estimate motion in video compression**, then it has been **widely used also for template matching**.
An advanced and more effective approach has been proposed recently (Tombari, Mattoccia & Di Stefano).

$$\mathbf{X}=\begin{bmatrix}x_1\\ \vdots\\ x_N\end{bmatrix} \Rightarrow \|\mathbf{X}\|_p=\left(\sum_{i=1}^{N}|x_i|^p\right)^{1/p}$$

Triangle Inequality:

$$\left|\|\mathbf{X}\|_p-\|\mathbf{Y}\|_p\right|\le\|\mathbf{X}-\mathbf{Y}\|_p$$

$$p=1,\ \mathbf{X}=\tilde{I}(i,j),\ \mathbf{Y}=T \rightarrow \left|\|\tilde{\mathbf{I}}(i,j)\|-\|\mathbf{T}\|\right|\le\|\tilde{\mathbf{I}}(i,j)-\mathbf{T}\|$$

Box-Filtering Off-line

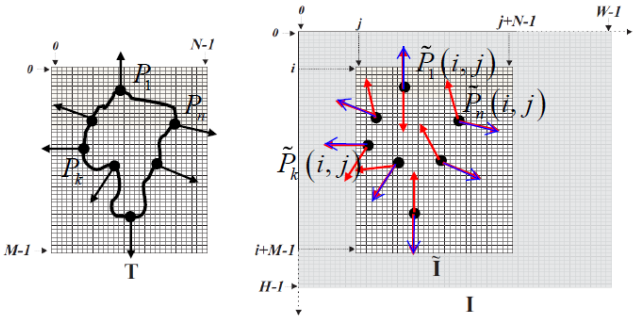
$$\left|\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}I(i+m,j+n)-\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}T(m,n)\right|\le\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left|I(i+m,j+n)-T(m,n)\right|$$

The first term is the difference of the norm of the current sub image and the template

$$\beta(i,j)\le SAD(i,j)$$

10.2 – SHAPE-BASED MATCHING

Shape-based Matching (Steger) can be thought of as an **edge-based template matching approach**.
First, a **set of control points, Pk, is extracted from the model image** by an Edge Detection operation (e.g., Sobel) and the gradient orientation at each Pk is recorded.
Then, at each position (i,j) of the target image, **the recorded gradient orientations** associated with control points **are compared to those at their corresponding image points**, $\tilde{P}_k$(i,j), in order to compute a similarity function.



Practically, we slide a window and compare at each position the points corresponding to the position of the edge pixel in the template. If we consider only the blue arrows, we don't consider all the positions. Assuming we compute these black gradients, we obtain the blue arrows and they are perfectly aligned, but we assess also the similarity with the red arrow: it is matching the shape of the contour and the current sub image.

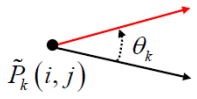
10.2.1 – SIMILARITY FUNCTION

The defined similarity function below spans the **interval [-1; 1]**. It takes its maximum value when all the gradients at the control points in the current window of the target image are perfectly aligned to those at the control points of the model image.
Choosing a detection threshold, Smin, can be thought of as **specifying the fraction of model points** which must be seen in the image to trigger a detection.

Template gradient → $\mathbf{G}_k = \mathbf{G}(P_k) = \left[G^x(P_k), G^y(P_k)\right]^T, \quad k = 1 \ldots n$

Gradient of the corresponding point in the image → $\tilde{\mathbf{G}}_k(i,j) = \mathbf{G}\left(\tilde{P}_k(i,j)\right) = \left[G^x\left(\tilde{P}_k(i,j)\right), G^y\left(\tilde{P}_k(i,j)\right)\right]^T, \quad k = 1 \ldots n$

$$S(i,j) = \frac{1}{n}\sum_{k=1}^{n} \frac{\mathbf{G}_k \cdot \tilde{\mathbf{G}}_k(i,j)}{\|\mathbf{G}_k\| \cdot \|\tilde{\mathbf{G}}_k(i,j)\|} = \frac{1}{n}\sum_{k=1}^{n} \cos\theta_k$$



The $\tilde{P}_k$ are unit vectors, so we're computing their cosine.
This function doesn't change with lighting changes because we have derivatives. We are not considering magnitude, so it is **perfectly invariant to light changes**.
The **tricky point** that makes this work very well is that **we extract edges in the template** and **we perform edge detection ONLY in the template and NOT in the image**.

10.2.2 – MORE ROBUST SIMILARITY FUNCTION

Certain application settings call for **invariance to global inversion of contrast polarity along object's contours**, as the object may appear either darker or brighter than the background in the target image.

This kind of invariance can be **achieved by a slight modification to the similarity function** defined previously:

$$S(i,j)=\left|\frac{1}{n}\sum_{k=1}^{n}\frac{\mathbf{G}_k\cdot\tilde{\mathbf{G}}_k(i,j)}{\|\mathbf{G}_k\|\cdot\|\tilde{\mathbf{G}}_k(i,j)\|}\right|$$

As a more robust similarity function it takes the absolute value. If the background is brighter than the foreground, we search the template on a black region. So, without the absolute value, we have that all gradients have opposite direction -1.
If we want to withstand the contrast polarity inversions, we have to use this absolute value or sometimes we can take the absolute value like in the second formula: there, we have LOCAL contrast polarity inversions and sometimes it is better.
Indeed, the following function is even more robust due to the **ability to withstand local contrast polarity inversions**:

$$S(i,j)=\frac{1}{n}\sum_{k=1}^{n}\frac{\left|\mathbf{G}_k\cdot\tilde{\mathbf{G}}_k(i,j)\right|}{\|\mathbf{G}_k\|\cdot\|\tilde{\mathbf{G}}_k(i,j)\|}=\frac{1}{n}\sum_{k=1}^{n}\left|\cos\theta_k\right|$$

10.3 – THE HOUGH TRANSFORM (HT)

The Hough Transform (HT) **enables to detect objects having a known shape** (e.g. lines, circles, ellipses…) **based on projection of the input data into a suitable space referred to as parameter or Hough space.**
The HT **turns a global detection problem into a local one** and it is usually **applied after an edge detection process** (i.e., the actual input data consist of the edge pixels extracted from the original image).
The HT is **robust to noise** and allows for **detecting the sought shape even though it is partially occluded** into the image (up to a certain user-selectable degree of occlusion).
It was invented to detect lines and later extended to other analytical shapes (circle, ellipses) as well as to arbitrary shapes (**Generalized Hough Transform**). The **GHT principle is widely deployed** also within state-of-the-art object detection pipelines relying on local invariant features such as, e.g., **SIFT**.

This is a pretty clever object detection method used to find certain type of shapes, basically analytics shapes, so shapes we can define with an equation. More recently, it has been used with local invariant feature paradigm, i.e., SIFT features, and it is really very good.

10.3.1 – BASIC PRINCIPLE

Let's consider first the HT formulation for lines: $y - mx - c = 0$
In the usual image space interpretation of the line equation, the parameters $(\hat{m}, \hat{c})$ are fixed, so that the equation represents the mapping $(1 \rightarrow \infty)$ from point $(\hat{m}, \hat{c})$ of the parameter space to the image points belonging to the line.

$$y - mx - c = 0 \qquad \rightarrow \qquad y - \hat{m}x - \hat{c} = 0$$

However, we may instead fix $(\hat{x}, \hat{y})$, so as to interpret the equation as the mapping $(1 \rightarrow \infty)$ from image point $(\hat{x}, \hat{y})$ to the parameter space providing all the lines through the image point.
The equation still represents a line, so all the parameter values representing lines through image point $(\hat{x}, \hat{y})$ lay on a line of the parameter space.

$$y - mx - c = 0 \qquad \rightarrow \qquad \hat{y} - m\hat{x} - c = 0$$

Accordingly, if we consider two image points P1, P2 and map both into the parameter space, we get **two lines intersecting at the parameter space point** representing the image line through P1, P2:

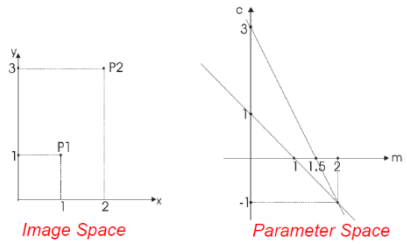
We get two equation of two lines and we obtain one m and one c. These lines give us all the lines through P1 and P2 and the intersection is the line through both.

$$\begin{cases} \hat{y}_1 - m\hat{x}_1 - c = 0 \\ \hat{y}_2 - m\hat{x}_2 - c = 0 \end{cases} \Longrightarrow \begin{cases} m = \frac{\hat{y}_2 - \hat{y}_1}{\hat{x}_2 - \hat{x}_1} \\ c = \frac{\hat{x}_2\hat{y}_1 - \hat{x}_1\hat{y}_2}{\hat{x}_2 - \hat{x}_1} \end{cases}$$

More generally, if we map n image points, we get as many intersections as

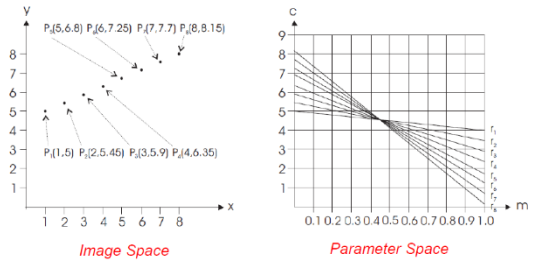
$$\frac{n(n-1)}{2}$$

i.e. the number of lines through the n image points.



Considering then n collinear image points, we can notice that **their corresponding transforms** (i.e. parameter space lines) **will intersect at a single parameter space point** representing the image line along which such n points lay:

We do that for every point in the image and if there are many curves in the parameter space, this means that there are many pixels compatible with the single instance of the shape we are looking for.
So, rather than looking for a line in the image, we look for a point in the parameter space.

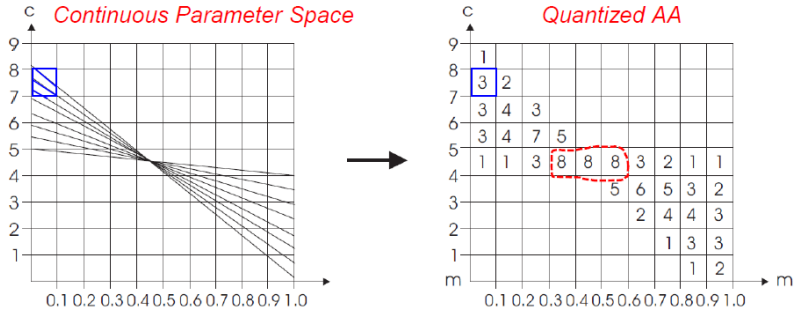


Therefore, **given a sought analytic shape represented by a set of parameters, the HT consists in mapping image points** (i.e. usually edge points) **so as to create curves into the parameter space of the shape.**
Intersections of parameter space curves indicate the presence of image points explained by a certain instance of the shape, the more the intersecting curves the higher the evidence of the presence of that instance in the image.
Detecting objects through the HT consists in finding parameter space points through which many curves do intersect, which is indeed a local rather than global detection problem.

To make it work in practise, the **parameter space needs to be quantized** and allocated as a memory array, which is often referred to as **Accumulator Array (AA)**.
The AA is initially empty and then it quantizes the parameter space and increment its cells and find the votes. If a shape get enough votes into the AA, we declare detection.

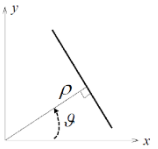
Formally, curves are "drawn" into the AA by a so-called **voting process**: the transform equation is repeatedly computed to increment the bins satisfying the equation. Accordingly, a high number of intersecting curves at a point of the parameter space will provide a high number of votes into a bin of the AA. Finding parameter space points through which many curves do intersect is thus implemented in practise by **finding peaks of the AA**, i.e. local maxima showing a high number of votes.

Exemplar AA for line detection



The AA highlights the presence of a line with $m \in [0.3, 0.6], c \in [4, 5]$. To detect the line more accurately, the AA should be quantized more finely.

The HT is robust to noise because spurious votes due to noise unlikely accumulate as coherently into a bin as to trigger a false detection. **A partially occluded object can still be detected** provided that the threshold on the minimum number of votes required to declare a detection is lowered according to the degree of occlusion to be handled.



10.3.2 – HT FOR LINE DETECTION

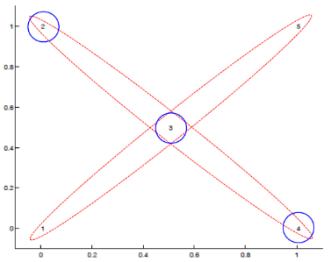
The usual line parametrization considered so far, i.e., y – mx – c = 0, is impractical due to m spanning an infinite range. The so-called **<u>normal parametrization</u>** is therefore adopted in the HT for lines:

$$\rho = xcos\vartheta + ysin\vartheta$$

It's hard to quantize the m because we have an infinite range, se often we apply HT and use another parametrization called normal in which we take the parameters ρ (ro) and ϑ.
Image points $(\hat{x}, \hat{y})$ are mapped into sinusoidal curves of the (ϑ, ρ) parameter space:

$$\rho = \hat{x}cos\vartheta + \hat{y}sin\vartheta$$

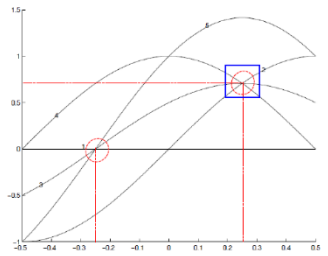
Both parameters have a finite range and each edge point in the image is mapped into a sinusoidal curve. So, in practice, we quantize ρ and ϑ and get the corresponding values.
With the normal parametrization:



$$\vartheta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right], \qquad \rho \in [-\rho_{max}, \rho_{max}]$$

while ρmax is usually taken as large as the image diagonal:

$$NxN\ pixels \rightarrow \rho_{max} = N\sqrt{2}$$

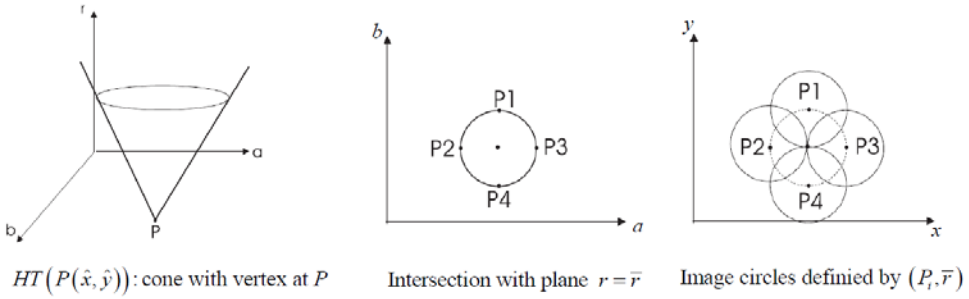


We have an image, so first edge detection and then we cast the votes into the parameter space and so we collect all these votes. Then, we perform non maxima suppression and pick the strongest peaks.
Noise can be a problem in this process during the edge detection. We don't find only the true strong edges, but we find also false edges (edges in uniform regions are symptom of noise).
Given that, are these spurious edges creating troubles in the process? They're going to vote quite randomly, so they just spread votes randomly into the AA and this is not causing any problem to us. So, this algorithm is robust to noise.

10.3.3 – HT FOR CIRCLE DETECTION

In the case of circular objects, the equation is interpreted as a mapping from the image space (x, y) to the 3-dimensional parameter space (a, b, r). Accordingly, the equation provides all the parameter triplets representing circles through image point $(\hat{x}, \hat{y})$.
(a, b) are in the center of the circle and r is the radius. Dimensionality of the parameter space = 3, so we draw cones in the 3D parameter space.

$$(x-a)^2 + (y-b)^2 = r^2$$

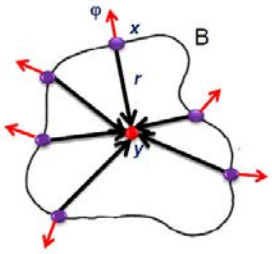


$HT(P(\hat{x}, \hat{y}))$: cone with vertex at P | Intersection with plane $r = \bar{r}$ | Image circles definied by $(P_i, \bar{r})$

10.3.4 – GENERALIXED HOUGH TRANSFORM (GHT)

OFF-LINE PHASE

The HT has been extended to detect arbitrary (i.e. non analytical) shapes:



Off-line Phase (to build the object's model)

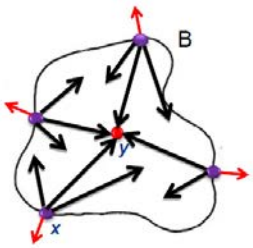
1. A reference point *y* is chosen (e.g. barycentre).
2. Gradient direction is quantized according to a chosen step $\Delta\varphi$
3. For each point *x* belonging to object's border *B*:
   1. Compute gradient direction $\varphi(x)$
   2. Compute vector *r* from *y* to *x* (*i.e.* $r = y\text{-}x$).
4. Store *r* as a function of $\Delta\varphi$ (R-Table)

What if we don't have an equation of the object? This is the typical object detection setting and it is very important. The first step is about extracting the edges of the shape. For example, we can start with the barycentre of something, maybe of the image, of the blob. We consider all edge pixels that have been found with edge detection and then we compute vectors which are the model of our shape. The position of the reference point in the image is the red one and the black arrows are the vectors that shall be the model of our shape.
Now, we compute gradients at all edge pixels: we already computed them because we did edge detection, so we have the red arrows that are the unit vector with according direction of the gradient. Then, we store all the joining vectors into a table which is call **R-Table** in a way that the table is indexed by quantized gradient directions.
If the red arrow has direction 45, we store it in the row of the table with 45: it is like the histogram we did, but inside the table.
This approach is also called **STAR MODEL**: the elements of the star model are stored into the rows of the R-table.

i	ϕ_i	R_{ϕ_i}
0	0	$\{r \mid y - r = x, x \in B, \phi(x) = 0\}$
1	$\Delta\phi$	$\{r \mid y - r = x, x \in B, \phi(x) = \Delta\phi\}$
2	$2\Delta\phi$	$\{r \mid y - r = x, x \in B, \phi(x) = 2\Delta\phi\}$
...	...	...

An entry in the R-Table can contain several r vectors.
At test time we acquire the target image, so we apply the edge detection on the target image.

ON-LINE PHASE



On-line Phase (object detection)

1. An image *A[y]* is iniatialized as accumulator array.

   For each edge pixel *x* of the input image :

   2. Compute gradient direction *φ*
   3. Quantize φ to index the R-Table. For each r_i vector stored into the accessed row:
      a) Compute the position the reference point *y*: $y=x+r_i$
      b) Cast a vote into the accumulator array: *A[y]++*
4. As usual with the HT, instances of the sought object are detected by finding peaks of the accumulator array.

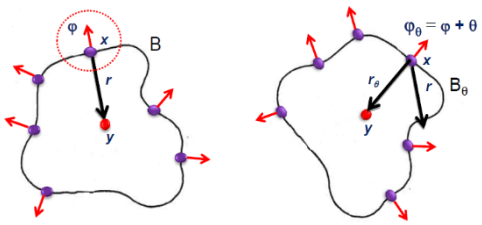
i	ϕ_i	R_{ϕ_i}
0	0	$\{r \mid y - r = x, x \in B, \phi(x) = 0\}$
1	$\Delta\phi$	$\{r \mid y - r = x, x \in B, \phi(x) = \Delta\phi\}$
2	$2\Delta\phi$	$\{r \mid y - r = x, x \in B, \phi(x) = 2\Delta\phi\}$
...	...	...

⇒

i	ϕ_i	R_{ϕ_i}
0	0	$\{r \mid y - r = x, x \in B, \phi(x) = 0\}$
1	$\Delta\phi$	r1,r2,r3
2	$2\Delta\phi$	$\{r \mid y - r = x, x \in B, \phi(x) = 2\Delta\phi\}$
...	...	...

for each pixel we look at the gradient direction and we index a row in the table in the row with the corresponding value. If the point is on the contour, we cast votes according to the stored joining vectors in the row. Now, we take a second edge pixel and so and so.
We declare a detection of the given shape in case many edge pixels in the targe image do vote coherently the position of the reference point in the image.
We quantize typically another image with the same size of the original one, initialized to zero, and we increment all pixels in this image which are AA as long as edges are going to cast their votes.
The AA is a space which encodes the pose of the object.

HANDLING ROTATION



If we use the same orientation as something plus theta, we have to subtract theta during accessing the table.
But we don't know there's a rotation… let's assume we know that: if the gradient is 80° but there's a rotation of 20° with respect to the canonical pose, we shall subtract the 20°. If we subtract 20, we correctly go to the right image.

$$T_s\left[R[\varphi]\right] = R\left[(\varphi_\theta - \theta) \bmod 2\Pi\right] \quad \Rightarrow \forall r : y = x + r_\theta = x + ROT(r,\theta),\ A[y,\theta]++$$

How to handle an unknown rotation? We have to try all rotations, there's nothing else we can do.
We quantize the rotation range and then we try all rotations: given a certain gradient, we vote all the hypothesized rotations. It means that the **AA is no longer an image**, but **is a stack of images called THE ROTATION.**
If we assume rotation from -10 to 10, we have 21 images, each corresponding to a certain rotation hypothesis.

HANDLING SCALE

$$\forall r : y = x + r_s = x + s \cdot r,\ A[y,s]++$$

HANDLING ROTATION AND SCALE

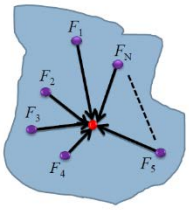
$$\forall r : y = x + s \cdot r_\theta,\ A[y,\theta,s]++$$

GEOMETRIC VALIDATION: STAR MODEL

Off-line phase:

$$F=\{F_1,F_2\dots F_N\},\ F_i=(\mathbf{P}_i,\varphi_i,S_i)$$

$$\mathbf{P}_C=\frac{1}{N}\sum_{i=1}^{N}\mathbf{P}_i \rightarrow \mathbf{V}_i=\mathbf{P}_C-\mathbf{P}_i$$



$$\forall F_i \in F \Rightarrow F_i=(\mathbf{P}_i,\varphi_i,S_i,\mathbf{V}_i)$$

For each of these blue points here we have a position in the template, a canonical orientation and a scale.

So, these points has a scale of 3, and orientation of X degrees.

We need also another parameter, like the barycentre of the image.

We compute joining vectors and then we create a richer set of features in which the keypoints.

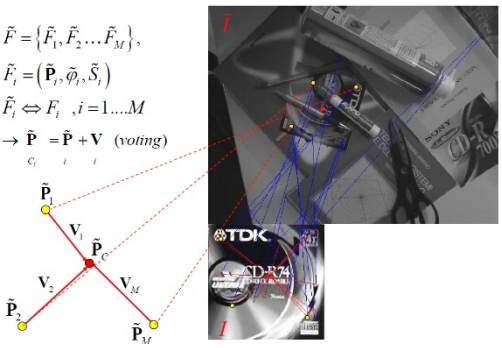
The R-table here is missing because we don't need it. We establish the potential matches based on the gradient direction which is the only information we can attach to an edge.

We don't need any R-table because we have others descriptors.

$$\tilde{F}=\{\tilde{F}_1,\tilde{F}_2\dots\tilde{F}_M\},\quad \tilde{F}_i=(\tilde{\mathbf{P}}_i,\tilde{\varphi}_i,\tilde{S}_i)$$

$$\tilde{F}_i \Leftrightarrow F_i\ ,i=1\dots M$$

$$\rightarrow \tilde{\mathbf{P}}_{C_i}=\tilde{\mathbf{P}}_i+\mathbf{V}_i\ \ (voting)$$



During the Online Phase, the reference point must be this because we found the joining vectors.

Now, this is quite similar to what we have done with edges.

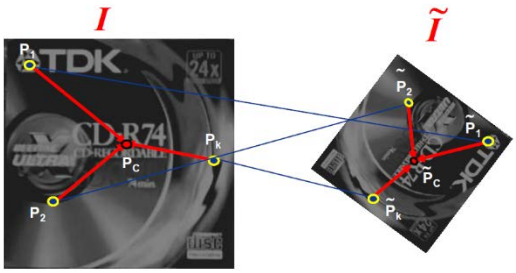
We match based on descriptors.

The red arrows on the left figure, i.e., the object in the reference image, are the joining vectors of the image I. The figure on the right is the the object in the target image.

If we apply the red arrows as they are, we don't find the barycentre, so we have to compute the rotation.

Then, if we apply the red arrows rotated but with their dimension, we again don't find the barycentre, so we need to scale them, in this case to reduce them.

Finally, after rotating and scaling the joining vector, we can find the ration between the scales.



$$F_i=(\mathbf{P}_i,\varphi_i,S_i)\Leftrightarrow\tilde{F}_i=(\tilde{\mathbf{P}}_i,\tilde{\varphi}_i,\tilde{S}_i)$$

$$\Delta\varphi_i=\varphi_i-\tilde{\varphi}_i,\ s_i=\frac{S_i}{\tilde{S}_i}$$

$$\tilde{\mathbf{P}}_{C_i}=\tilde{\mathbf{P}}_i+s_i\cdot\mathbf{R}(\Delta\varphi_i)\mathbf{V}_i$$

Example