UNIVERSITY OF PADUA

Departement of Information Engineering

MASTER'S DEGREE IN CONTROL SYSTEMS ENGINEERING

COMPUTER VISION AND ARTIFICIAL INTELLIGENCE IN INDUSTRIAL ENVIRONMENT

Supervisor: Prof. Giovanni Boschetti

Candidate: Dott. Leonardo Brigida

ACADEMIC YEAR 2023-2024

A Giulia

Contents

Sommario

La Computer Vision è una disciplina, ora più che mai, rilevante sia per usi civili, come le applicazioni di elaborazione grafica, che per usi industriali, come il riconoscimento e la localizzazione di oggetti nello spazio. È proprio in quest ultimo ambito che si possono ottenere maggiori vantaggi, sia dal punto di vista economico che di precisione. Lo scopo di questo elaborato è quello di analizzare, partendo dai fondamenti dei funzionamenti della Computer Vision e dell'Intelligenza Artificiale, le applicazioni in campo industriale. Vengono analizzati tre diversi scenari: controllo qualità, tracciamento di oggetti sopra ad un nastro trasportatore ed infine la localizzazione di un oggetto nello spazio tramite la visione stereoscopica. Sono riportati inoltre i risultati sperimentali per tutti gli scenari analizzati, attraverso prove svolte in laboratorio utilizzando telecamere industriali, un nastro trasportatore e un delta robot posizionato sopra di esso. In conclusione la positività dei risultati ottenuti dimostra quanto queste tecnologie siano utili per aumentare l'efficienza e la precisioni dei processi industriali.

Abstract

Computer Vision is a discipline, now more than ever, relevant both for civil uses, such as graphics processing applications, and for industrial uses, such as the recognition and localization of objects in space. It is precisely in this last area that greater advantages can be obtained, both from an economic and precision point of view. The aim of this paper is to analyse, starting from the fundamentals of the functioning of Computer Vision and Artificial Intelligence, the applications in the industrial field. Three different scenarios are analysed: quality control, tracking of objects on a conveyor belt and finally the localization of an object in space using stereovision. The experimental results are also reported for all the scenarios analyzed, through tests carried out in the laboratory using industrial cameras, a conveyor belt and a delta robot positioned above it. In conclusion, the positivity of the results obtained demonstrates how useful these technologies are for increasing the efficiency and precision of industrial processes.

Introduction

The aim of this thesis is to analyze some applications of the Computer Vision and Artificial Intelligence in the industrial field, starting from their fundamentals, then defining the experimental setups used and finally developing the complete applications. In industrial environment a lot of tasks requires a visual feedback to be completed, let's think about the quality control or object localization, these usually are performed by humans. This leads to some problem relates to the accuracy of the task, because the human precision is not high in locating a possible object in space. Furthermore the parameters for a quality control task are hard to be defined operating with humans, because the evaluation is person dependent, so different persons have different perception of "good" and "bad" samples. Automation is the key to obtain advantages from all the points of view: hard and alienating jobs are executed in autonomous way leaving the operator free to complete other tasks in parallel, this allows a faster and more precise production, making the lives of operators less tiring and increasing the overall economic incomes. My interest in analyzing the Computer Vision and Artificial Intelligence scientific disciplines is born not only to understand their functioning but also to discover their potential in the industrial field. In the first part of this research the Computer vision is analyzed, highlighting how the projection from the 3D space to the image plane works, exploring all the techniques used for the image analyses, from the low levels to the high levels ones. In the second part some principles of Artificial Intelligence are reported, exploiting the learning phase, passing through the feed forward neural networks, finally focusing on the field of the AI that is able to operate with images, so the convolutional neural net-

works, reporting one of the main CNN architecture used nowadays: the YOLO architecture. After this overview three application are analyzed in details: quality control, conveyor tracking and stereovision. For each application there is a different section in which the task to be solved is explained, analyzing the theory used to develop the practical solution. The quality control task was developed during an internship at the Vimar company, in complete autonomy I developed not only the setup but also the software to control the quality of the screen of all the video intercoms made, which until then were controlled by humans. The conveyor tracking task instead involves locating objects on the conveyor belt to be subsequently picked up and moved by a robot to substitute humans in this alienating job. The localization is made using a software that analyzed the images that came from a camera on the belt. Two approaches are used, the first that uses only computer vision low and mid levels techniques, the second one, more general, that makes the use of Artificial Intelligence, more specifically I used a YOLO CNN to detect different objects on the belt and control the robot so that it places the object in different desired final location based on the classes of the objects detected. Finally one very interesting field of study is to reconstruct the 3D position in space of a desired object from images that came from 2 or more cameras that frame it. This task is called stereovision, in my approach I build my own stereocamera system, in which the two cameras are not only translated but also rotated. I developed a Python program able to detect an object using the YOLO CNN used in the previous task and retrieve the 3D position with respect to a desired world reference frame. In conclusion for completeness all the codes developed and used for these projects are reported in the Appendix section.

Chapter 1

Development of Computer Vision

Extract useful information from images is, today more than ever, a topic of interest, it is imperative for advancing fields such as autonomous driving, medical diagnostics, security systems and industrial applications.

1.1 Origin of computer vision

The introduction of the Bartlane cable picture transmission system in the early 1920s is one of the earliest applications of digital images, used in the newspaper industry, when pictures were sent by submarine cable between London and New York. The first technique for improving image quality was implemented in 1964, when pictures of the moon transmitted by Ranger 7, a space probe began at the Jet Propulsion Laboratory (Pasadena, California), were affected by a distortion caused by the on-board camera, this were corrected by a Computer Vision process. From the 1960s until the present digital image processing techniques have improved and now they are used in a broad range of applications intended for human interpretation, for example in medicine to enhance the contrast or code the intensity levels into color for easier interpretation of X-rays. The Computer Vision is used also in another area of application that is the one related to solving problems dealing with machine perception. In this second case the goal is to extract information from an image in a form that is suitable for computer pro-

cessing, like applications in industries, for industrial machine vision for product assembly, and inspection and in military field, for recognition.[1]

1.2 Digital image acquisition

Nowadays we are in the digital age and all the modern camera acquires the images in a digital version, they are equipped with a sensor, typically a CMOS one. Let's now go into the details of the image formation model: an image is a two-dimensional functions of the form $f(x, y)$, it's value is a scalar quantity at spatial coordinates (x, y) and is proportional to the energy radiated by a physical source and consequentially absorbed from the camera sensor. So $f(x, y)$ is non-negative and bounded, for a monochrome image at any coordinates (x, y) is typically between 0 and 255, encoded in 1 byte of memory, for an RGB image instead the function has 3 channel, each one for each primary color. Since an image in the real world can be seen as a continuous function with respect to the x- and y-coordinates, to create a digital one a conversion into a digital format is needed: this is made using *sampling* and *quantization*. The CMOS sensor has it's own spatial resolution, enumerated with the number of pixels of the sensor, this digitize the coordinate values and is the sampling part, the output a continuous voltage waveform so a quantization of the amplitude values is needed to store the information in a digital memory. So the quality of a digital image is determined by the spatial resolution of the sensor and by the discrete intensity levels used in quantization, process that is not reversible. An image can be seen not only as a function but also as a matrix, for example in the following equations are reported the representation of an image of shape MxN: in the equation 1.1 as a function and in the equation 1.2 as a matrix.

$$f(x,y) = \begin{bmatrix} f(0,0) & f(0,1) & ... & f(0,N-1) \\ f(1,0) & f(1,1) & ... & f(1,N-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(M-1,0) & f(M-1,1) & ... & f(M-1,N-1) \end{bmatrix} \tag{1.1}$$

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,N-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M-1,0} & a_{M-1,1} & \dots & a_{M-1,N-1} \end{bmatrix} \tag{1.2}$$

From the previous two equations is possible to see that the convention to the center of the origin of an image is at the top left corner, the x axis goes downwards and the y axis goes to the right. For the image digitization knowing M, N and L, the number of discrete intensity levels for each channel, is necessary. There is no restriction regard the choice of N and M, they are chosen initially as the sensor resolution but they can be changed at will in the post processing, the only restriction is that they must be positive integers. Instead the L value, since is archived in a digital hardware, is usually an integer power of two:

$$L = 2^k \tag{1.3}$$

where the exponent k is an integer. All the discrete levels are equally distributed and are all the integers in range $[0, l-1]$. In the following figures are reported the image plotted as a surface, figure 1.1, the image displayed as a visual intensity array, figure 1.2 left, and the image reported as a 2-D numerical array, figure 1.2 right.

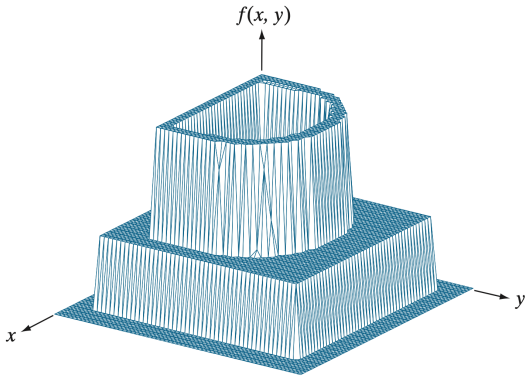


Figure 1.1: Image of a letter "D" plotted as a surface [1]

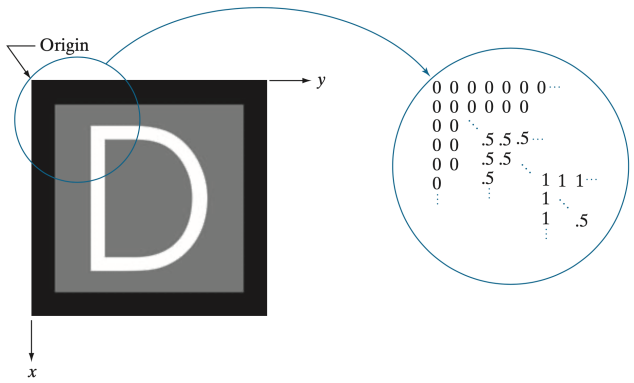


Figure 1.2: Image of a letter "D" displayed as a visual intensity array on the left and it's normalised intensity values on the right [1]

1.2.1 Pinhole camera model

The first model of a camera was the *Pinhole camera model* reported in the figure 1.3: [2]

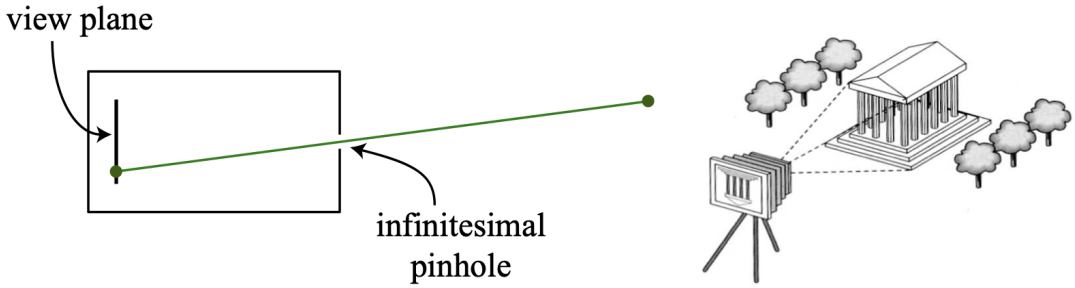


Figure 1.3: Pinhole camera model [2]

The light from a point travels on a straight path through a pinhole and the object is imaged upside-down on the image plane, only from a mathematical point of view we can consider another image plane between the hole and the object in which the image is not upside-down. The main elements of this model are the *optical center*, the location of the pinhole, the *image plane*, the *optical axis*, perpendicular to the image plane and passing through the optical center, the *principal point*, the intersection between the image plane and the optical axis, and the $focal\ length$, the distance between the image plane and the optical center. From this model is possible to describe the *perspective geometry*: the relation

between a 3D point in the world and a 2D point in the image plane. The following figure is a visual representation of the geometry of projection:

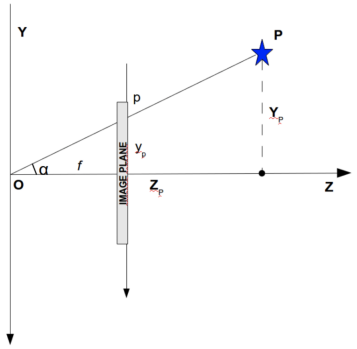


Figure 1.4: Visual representation of the perspective geometry [3]

In the figure 1.4 is possible to see the relation between two triangles, the one with cathetes Y_P and Z_P and the one with cathetes y_p and f. So using the similar triangle rule we obtain the following relation:

$$\frac{Y_P}{y_p} = \frac{Z_P}{f} \tag{1.4}$$

this equation is analogous for the X coordinates. Therefore we obtain the following relation:

$$x_p = f * \frac{X_P}{Z_P} \tag{1.5}$$

$$y_p = f * \frac{Y_P}{Z_P} \tag{1.6}$$

$$tan(\alpha) = \frac{Y_P}{Z_P} \tag{1.7}$$

where x_p and y_p are coordinates of the image plane and Y_P, X_P and Z_P are 3D coordinates. Since the projection is onto a 2D plane the distance information is lost, all the points in the line between the principal point and the point P are projected all in the same image point p. The equations above can be rearranged

in matrix form, using the homogeneous coordinates:

$$Z\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = Z\begin{bmatrix} \frac{fX}{Z} \\ \frac{fY}{Z} \\ 1 \end{bmatrix} = \begin{bmatrix} fX \\ fY \\ Z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{1.8}$$

where the 3x4 matrix in the last equation is called *Projection matrix*. Since in the image plane the coordinates used are the pixels, there is another conversion from (x, y) to (u, v) coordinates, this transformation can be defined using the principal point, at coordinates (u_0, v_0). The conversion between meters and pixels needs the pixels width and height, respectively w and h, and it uses the following conversion factors:

$$k_u = \frac{1}{w} \qquad k_v = \frac{1}{h} \tag{1.9}$$

So the mapping from (x, y) to (u, v) is reached by translation and scaling, as reported in the following equation:

$$u = u_0 + \frac{x_p}{w} = u_0 + k_u x_p \qquad v = v_0 + \frac{y_p}{w} = v_0 + k_v y_p \tag{1.10}$$

Combining the conversion from image plane to pixel of equation 1.10 with the projection equation 1.5, we obtain the following relation:

$$u = u_0 + k_u x_p = u_0 + \frac{f}{w}\frac{X_P}{Z_P} = u_0 + f_u \frac{X_P}{Z_P} \tag{1.11}$$

The same result is valid for the y coordinates. So the projection is now the following:

$$P = \begin{bmatrix} f_u & 0 & u_0 & 0 \\ 0 & f_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = K[I|\mathbf{0}] \tag{1.12}$$

where K is the *camera matrix*. It depends from the parameters k_u, k_u, u_0, v_0 and f that are called *intrinsic parameters*, that defines the projection characteristics of the camera.

Instead the position of the camera with respect to the world is defined by a rototranslation matrix, reported in the following equation:

$$T = \begin{bmatrix} R & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \tag{1.13}$$

The numbers of parameters involved in this matrix are 3 for translations and 3 for rotations, these are called *extrinsic parameters*.
So the overall projection become:

$$\tilde{m} \simeq PT\tilde{M} \tag{1.14}$$

where $\tilde{m}$ is the homogeneous column vector in the image plane and $\tilde{M}$ is the homogeneous column vector in the 3D space, the equation is true up to a scale factor Z. The overall process goes from the world coordinates to the camera coordinates, from these to the sensor plane and finally to the pixels coordinates.[3] This model brings with it a trade-off between *sharp image* and the *light intensity*, so the modern cameras uses a different model that is the *thin lens model*. The lens is used to focus the light onto the sensor surfaces, so that it can captures enough light in a sufficiently short period of time, this solved the problem of blurry images. The thin lens can be consider as a 2D plane where the deviation of the light occurs and his focal length, $|f|$ is defined as the distance from the lens which rays from an infinitely distant source converge in focus. In the following figure the thin lens model is reported:

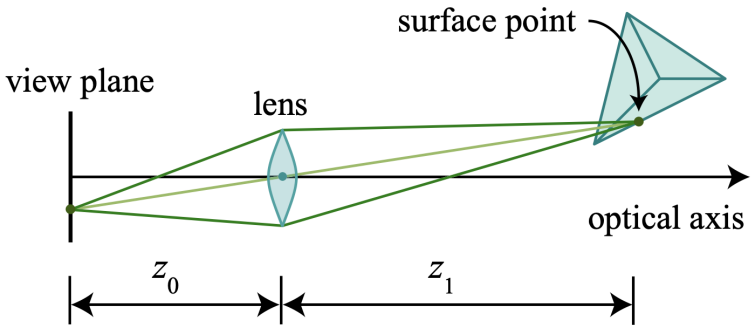


Figure 1.5: Thin lens model [4]

In the figure 1.5 is possible to see two distances: the first between the image plane and the lens, Z_0, and the second between the lens and the object, Z_1. The thin lens equation that relates this two distances with the focal length is the following:

$$\frac{1}{f} = \frac{1}{Z_0} + \frac{1}{Z_1} \tag{1.15}$$

All the equations of the perspective geometry are still valid. The real lenses are affected by additional effects among which the most relevant is the *distortion* of the image. The distortion can be of two type: *radial* and *tangential*; in the radial one the amount of the distortion depends on the distance of the point from the image center, instead in the tangential one the distortion is caused by the non-ideal alignment between the lens and the sensor. The radial distortion can be corrected using the following polynomial approximation:

$$x_{corr} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{1.16}$$

$$y_{corr} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{1.17}$$

where (x_{corr}, y_{corr}) are the corrected points of the distorted (x, y). The tangential one can be modeled using the following equations:

$$x_{corr} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \tag{1.18}$$

$$y_{corr} = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \tag{1.19}$$

In the real world there exist also another type of lenses: the wide lenses. This type is characterized by a FOV, field of view, which is typically greater with respect to a normal one, so it makes a distortion of the image because it fills in the same sensor size a wider image. The distortion is not only perspective but also radial. For the cameras with this type of lens the perspective geometry with respect to the one explained in this chapter changes.

1.2.2 Camera calibration

To measure the projection characteristics of a camera and it's distortion coefficients a camera calibration process is needed. This process estimates only the intrinsic camera parameters, the extrinsic ones can be found with another dedicated process.
The general camera calibration process follows the following steps:

- Takes an object of known shape
- Takes different pictures of it
- Analyzes the projection process

Instead of using a random object for this process, since it must be found in the image, an object with an easily recognisable shape is suggested. In practice the most used one is the checkerboard, in which the square corners are used as reference points for the calibration. So the calibration process in detail is the following:

- Collect N images of the checkerboard
- For each image:
  - List the M 3D corner positions in the pattern reference system
  - Find the corner positions in the image reference system
- Initialize the intrinsic parameters:
  - For K: f_u, f_v, u_0, $v_0 \rightarrow \theta_K$
  - For distortion: k_1, k_2, k_3, p_1, $p_2 \rightarrow \theta_d$
- Initialize the extrinsic parameters: usually the matrix T is set to be the identity matrix

- Solve the non-linear least squares problem:

$$\min_{\theta_K,\theta_d} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} ||K[I|\mathbf{0}]T_j\tilde{\mathbf{P}}_{\mathbf{i,j}} - \tilde{\mathbf{p}}_{\mathbf{i,j}}||^2 \tag{1.20}$$

Where $\tilde{P}_{i,j}$ is the 3D point and $\tilde{p}_{i,j}$ is the corresponding 2D point in the image plane in homogeneous coordinates. Each view of a planar object can be represented by a homography so only four points per view are free, more points are useful to measure distortion. Neglecting distortion there exist 5 intrinsic parameters so at least 2 images are needed, but since the calibration process is a minimization one, a larger number of images is needed. Details of the process in Python code is reported in Appendix A.

1.2.3 Spatial and Intensity resolutions

The sensor resolution and the choice of the L value influence two type of resolution in the output image from a camera: the *spatial* resolution and the *intensity* resolution. In practice the *spatial* resolution can be seen as the measure of the smallest detectable detail in an image, a common measure is the dots or pixels per unit distance. Image size by itself does not carry this information, for example an image with resolution of 1024 x 1024 pixels is not a meaningful statement without reporting the spatial dimensions covered by the image. The size of the image is useful in making comparison between two different camera sensors: a 20-megapixel sensor has a higher capability to notice details with respect to an 8-megapixel sensor, but assuming that both cameras are equipped with comparable lenses and the pictures are taken at the same distances from the framed subject. This last assumption is important because the distance of the object of interest from the camera and the camera lens make an important role to determine the effective spatial resolution that a camera has. There exist a lot of different type of camera lenses, each one with different focal length, that influences the distance of the plane from the camera in which the object is in focus. Let's make a practice example of spatial resolution to be more clear: consider three different cameras

that take a photo of an object of dimension 10 cm at a distance from the cameras of 20 cm. For this example let's take the following cameras which specifications are reported in the following table 1.1: RPi Camera (G), RPi Wide Angle Camera Module, OAK-1 camera.

	RPi Camera (G)	RPi Wide Angle Camera Module	OAK-1 Camera
FOV (deg)	160°(D) x 140°(H)	135°(D) × 120°(H) × 105°(V)	81°(D) × 69°(H) × 55°(V)
Sensor resolution	5MP (2592 × 1944 px)	12MP (4056 x 3040 px)	12MP (4056 x 3040 px)

Table 1.1: Specifications of three different cameras (D diagonal, H horizontal, V vertical

To execute this test we can use the geometry projection formula 1.5, but this relation can be applied only to the OAK-1 camera because the other two have a wide lens. For this cameras we can use another qualitative approach considering an uniform distortion for all the FOV of the camera, which normally doesn't happen. The relation that we consider is the following:

$$N_{pixels} = \frac{28,07^\circ}{FOV^\circ} * Resolution \tag{1.21}$$

where the output N_{pixels} is the number of pixels occupied by the object in the image, $28,07^\circ$ is the angle of circumference occupied by the object of 10 cm and *Resolution* is the horizontal resolution of the sensor. The results are reported in the following table 1.2:

	RPi Camera (G)	RPi Wide Angle Camera Module	OAK-1 Camera
N_{pixels}	520 px	949 px	1552 px

Table 1.2: Number of pixels occupied by an object of 10 cm at a distance of 20 cm from three different cameras

In this example the OAK-1 has the best spatial resolution at a distance of 20 cm. The *intensity* resolution instead is the smallest detectable change in intensity level, it is a common practice to consider it as the number of bits used to quantize the intensity level, is the k value of the equation 1.3.

1.3 Computer vision processing

An image can be transformed or analyzed in several ways, both for obtain a better image to be visualized and to extract useful information that can be used for further tasks. Let's now go into the details of the main computer vision processes, first analyzing the low-level image processing, with spatial operations, then moving on the mid-level, for example detecting lines and features, to finally reach the high-level vision processes, for example the object recognition one.

1.3.1 Low-level image processing: spatial operations

Spatial operations refer to the transformations performed directly on the pixels, these are classify into three categories: geometric transformations, single-pixel operations and local operations.

Geometric transformations

A geometric transformation, as is deductible, is an operation which is going to change the spatial relationship among pixels. It is made up of two steps:

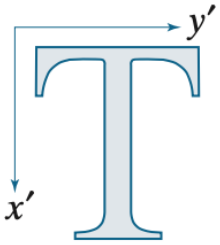
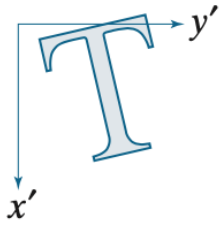
- Coordinate transformation: $(x', y') = T(x, y)$

- Image resampling

There exist different affine transformations, which key characteristic in 2D is that they preserve points, straight lines, and planes. The affine transformation can be written as an extension of the formula $(x', y') = T(x, y)$ using homogeneous coordinates in the following way:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbf{A} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1.22}$$

The main ones are scaling, translation, rotation, and shearing, reported in the following table 1.3:

Transformation name	Affine matrix, A	Coordinate equations	Example
Scaling or Reflection (for reflection, set one scaling factor to -1 and the other to 0)	$\begin{bmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = c_x x$ $y' = c_y y$	
Rotation (about the origin)	$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x\cos\theta - y\sin\theta$ $y' = x\sin\theta + y\cos\theta$	

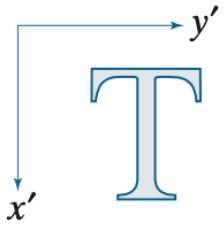
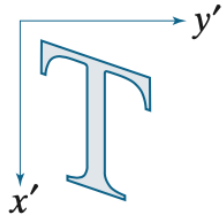
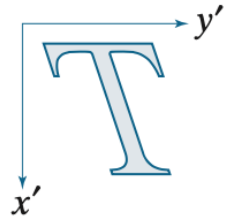
Translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + t_x$ $y' = y + t_y$	
Shear (vertical)	$\begin{bmatrix} 1 & s_v & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + s_v y$ $y' = y$	
Shear (horizontal)	$\begin{bmatrix} 1 & 0 & 0 \\ s_h & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x$ $y' = s_h x + y$	

Table 1.3: Examples of affine transformations

The second step, the image resampling, is needed because the geometrical transformations work on geometrical points, not pixels. The resampling from geometrical points to pixels is needed, and two different type of mapping exist: forward and backward mapping. The differences between these two mapping are reported in the following table 1.4:

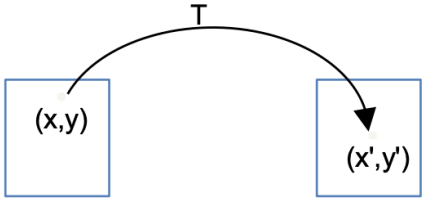
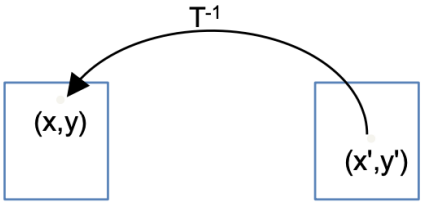
Forward mapping	Backward mapping
	
$(x', y') = T(x, y)$	$(x, y) = T^{-1}(x', y')$
For each (x, y) the mapping computes the corresponding (x', y')	For each (x', y') the mapping computes the corresponding (x, y)
Ambiguity: there may be multiple points on the same (x', y')	No ambiguity: find each (x', y') only once
May be missing pixels	Fills all pixels

Table 1.4: Differences between forward and backward mapping

Since the backward mapping fills all the pixels with no ambiguity is the best choice for mapping pixels in the image resampling phase.

Single-pixel operations

The simplest operation that can be performed on a digital image is to change the intensity of its pixels individually, using a transformation function, T, as follow:

$$s = T(z) \tag{1.23}$$

where z is the current intensity level of a pixel in the original image and s is the mapped one. There exist several operations, the main ones are:

- Negative
- Logarithm
- Gamma
- Contrast stretching

- Intensity slicing
- Histogram equalization

For a grayscale image the **negative** transformation switch dark and light using the following equation:

$$s = (L - 1) - r \tag{1.24}$$

where r is the current intensity level, an example is reported in the following figure:

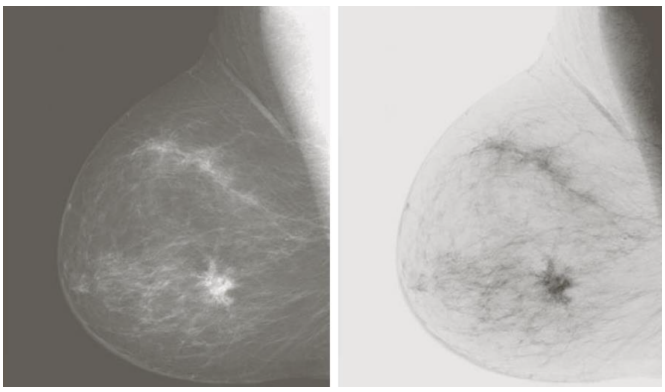


Figure 1.6: Example of negative transformation [1]

The **logarithm** operation highlights the differences among pixels, using the following equation:

$$s = c\ log(1 + r) \qquad c = \frac{L - 1}{log(L)} \tag{1.25}$$

A similar version of this last one is the **gamma** transformation, that is also tunable:

$$s = c\ r^{\gamma} \qquad c = (L - 1)^{1-\gamma} \tag{1.26}$$

In the following figure several examples of gamma transformation are reported, the logarithm one is similar to the gamma using a γ equal to 0.3:

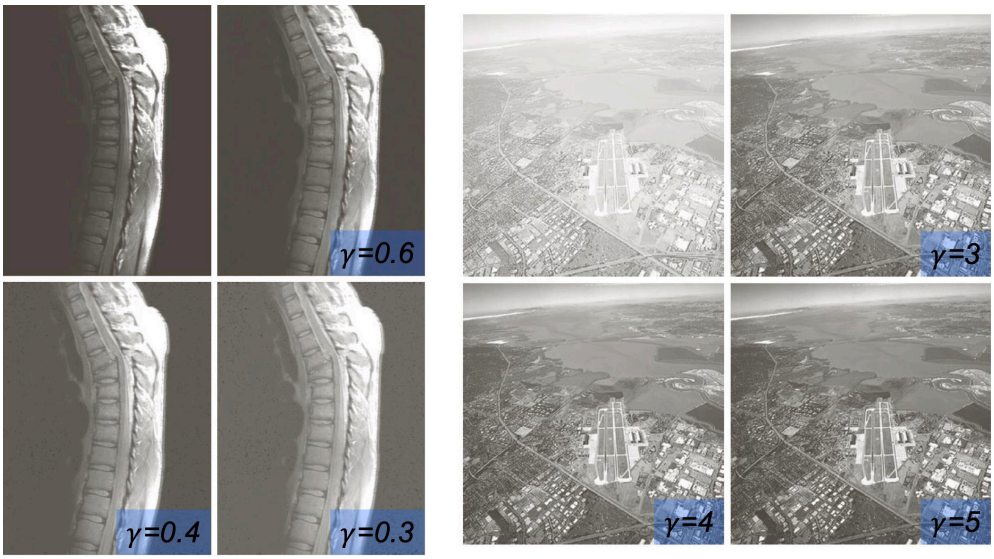


Figure 1.7: Example of gamma transformations [1]

The **contrast stretching** is useful for enhance the contrast, that is the difference between the highest and the lowest intensity level in the image, but can be also measured as the *root mean square contrast*:

$$RMSC = \sqrt{\frac{1}{MN}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}(I(i,j)-\bar{I})^2} \tag{1.27}$$

The **intensity slicing** consist in consider only a subset of gray level and set all the other to 0 or 255, to make evident only some interesting part of the image. Another technique consist in the image **thresholding** that set all the pixels below some threshold to 0 and the others to 255. The last important single-pixel operation is the histogram equalization. The histogram of an image is composed by the intensity level of gray in the horizontal axis and the number of pixels per level of intensity in the vertical axis. The equalization try to make flat this histogram, the function used to this process is the cumulative distribution function of the histogram:

$$s_k = (L-1)\sum_{j=0}^{k} p_r(r_j) \qquad p_r(r_j) = \frac{n_k}{MN} \tag{1.28}$$

where r is the original gray level and the round of s is the new one. If the image has different regions that have very different pixel distribution characteristics this

process can give better result if used for local neighborhoods.

Local operations

This last category of spatial operation make use of a *kernel/filter*, that is a neighborhood of the pixel taken into consideration. In the kernel a weight is associated to each pixel involved, this depends to the process applied and the filter can be linear or non-linear. The filter can be applied to the image in several ways, for example evaluating a correlation/convolution or taking the max/min values of all the pixel involved in the kernel. In the correlation operation the filter is superimposed on each pixel location and there is the evaluation of a weighted average. Considering a filter of dimension mxn:

$$\begin{aligned} m &= 2a + 1 \\ n &= 2b + 1 \end{aligned} \tag{1.29}$$

The *correlation* operation is defined as:

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) I(x + s, y + t) \tag{1.30}$$

Instead the equation of the *convolution* applied in the spatial domain is the following:

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) I(x - s, y - t) \tag{1.31}$$

Comparing these two last equations is possible to see that the only things that change are the two signs in the last parenthesis, and since a lot of filters are usually simmetric, convolution and correlation returns the same result. Usually these two operations are used as synonyms, in most of the cases convolution is used and the filters obtained by applying convolution are called convolutional filters. There exist a lot of different *spatial filters*:

- Linear filters (convolution)
  - Average

  - Derivative

- Non-linear filters
  - Min
  - Max
  - Media

If the sum of all the weights is 1 the brightness of the image is unchanged.

Linear filters

The *average* filter is a filter of size nxn, the bigger n is, the more it smooths the image, and all the weights of the kerner all equal to $\frac{1}{nxn}$. The *derivative* one can be of the first or the second order, the first is non-zero on the onset of a step and along the ramp, the second is non-zero on the onset and at the end of a step but is zero along ramps of constant slope. These operations can be implemented using specific filters, for example for the gradient computation the most famous ones are the *Sobel filters*, instead for the second ones are the *Laplacian filters*:

$$Sobel: \quad \frac{\delta f}{\delta y} = \begin{array}{|c|c|c|}\hline -1 & -2 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline \end{array} \qquad \frac{\delta f}{\delta x} = \begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \tag{1.32}$$

$$Sobel\ diagonal: \quad \begin{array}{|c|c|c|}\hline 0 & 1 & 2 \\ \hline -1 & 0 & 1 \\ \hline -2 & -1 & 0 \\ \hline \end{array} \qquad \begin{array}{|c|c|c|}\hline -2 & -1 & 0 \\ \hline -1 & 0 & 1 \\ \hline 0 & 1 & 2 \\ \hline \end{array} \tag{1.33}$$

$$Laplacian: \quad original = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{1.34}$$

$$Laplacian: \ with\ diagonal = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{1.35}$$

The example of the application of the sobel filters is reported in the figure 1.8 and the example of the result of the laplacian ones in the figure 1.9:

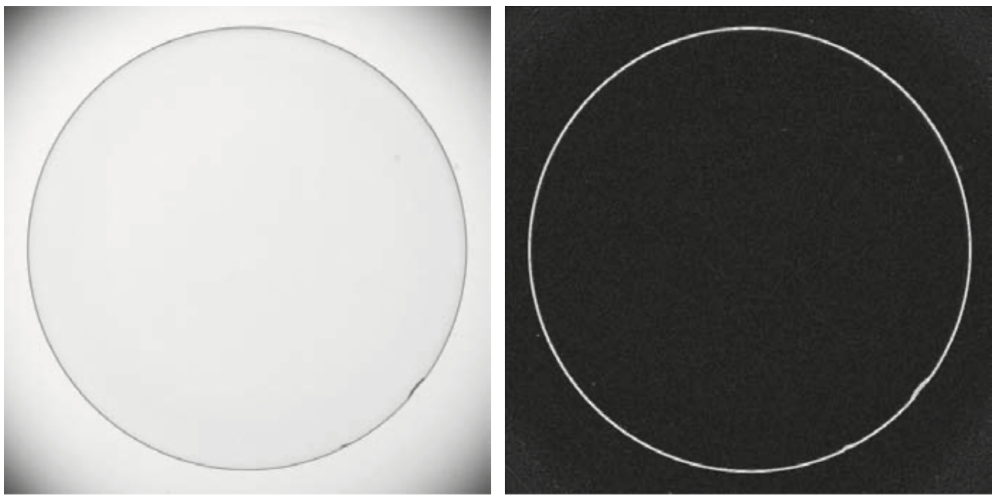


Figure 1.8: Example of the result of the application of the sobel filters (right image) of the original image reported on the left [1]

The laplacian image result can be usefull if we want to sharpening the original one, subtracting the two images.

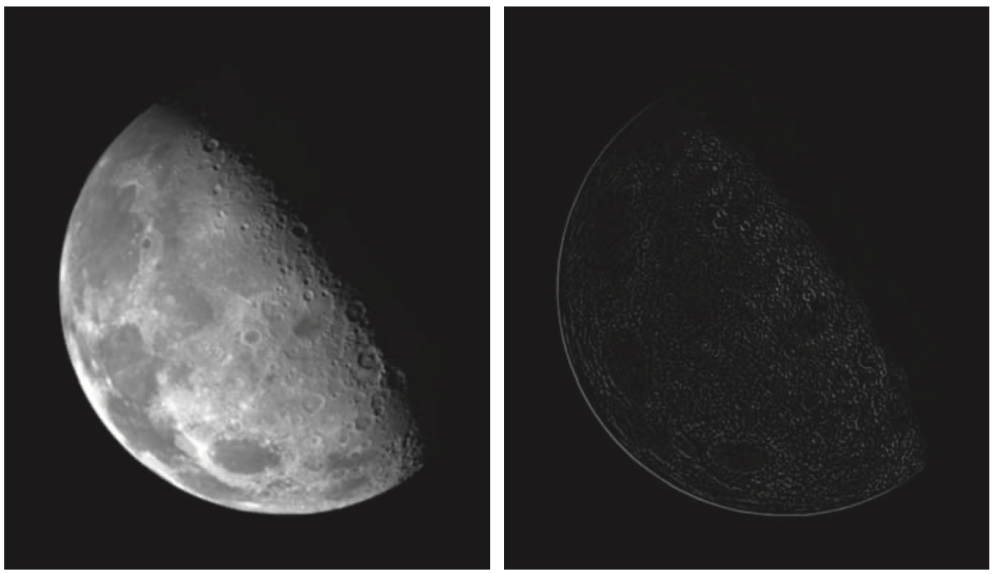


Figure 1.9: Example of the result of the application of the laplacian filters (right image) of the original image reported on the left [1]

Non-linear filters

The image acquired in the real world is composed by the ideal one plus some noise, so an important area of image analyses regard the *image restoration*. There exists several type of noise models, for example gaussian, salt & pepper and uniform noise. To remove the noise the median filter can be used, there exist several types of this category of filters, for example the *arithmetic* one:

$$g(x,y) = \frac{1}{mn} \sum_{s,t \in R} f(s,t) \tag{1.36}$$

This filter can remove gaussian noise but not salt & noise one, for this last type other filters can be used. For example the *max* filter highlights salt noise and removes the pepper one, instead the *min* filter highlights pepper noise and remove the salt one.

1.3.2 Mid-level image processing

Mid-level image processing consider the analysis of more complex concepts, the main ones are morphological operators, edges, lines, image segmentation and features.

Morphological operators

Morphological operators belong to the low-level image processing, but they operate on the shape in binary images. The logic beneath these operators is the membership or not of a set of pixels in the entire image. These operator add or remove pixels from the image, changing its shape. More in detail exist two main morphological operator: *erosion* and *dilation*, and two concatenation of them: *opening* and *closing*.
Considering two sets A and B the **erosion** operation is defined as:

$$A \ominus B = \{z|(B)_z \subseteq A\} \tag{1.37}$$

The explanation is the following: move the set B to the z point and keep z if and only if the whole B set is included in A. This process separates weakly connected components in the image or makes them thinner. In the figure 1.10 two examples of erosion operations are reported.
Considering two sets A and B the **dilation** operation is defined as:

$$A \oplus B = \{z|(B)_z \cap A \neq \emptyset\} \tag{1.38}$$

The explanation is the following: move the set B to the z point and keep z if and only if there is at least one overlapping pixel with A. This process connects weakly separated components in the image or makes them thicker. In the figure 1.11 two examples of dilation operations are reported.

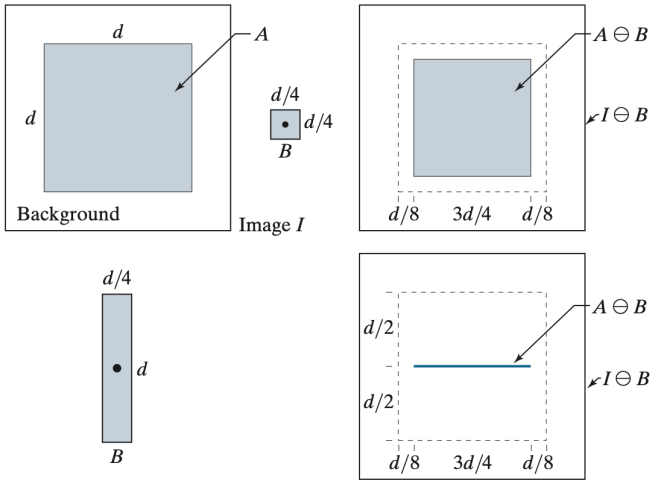


Figure 1.10: Top left: original image; top center: B set; top right: result of erosion between A and B; bottom left: another set B; bottom right: result of erosion between this last set and the original image [1].

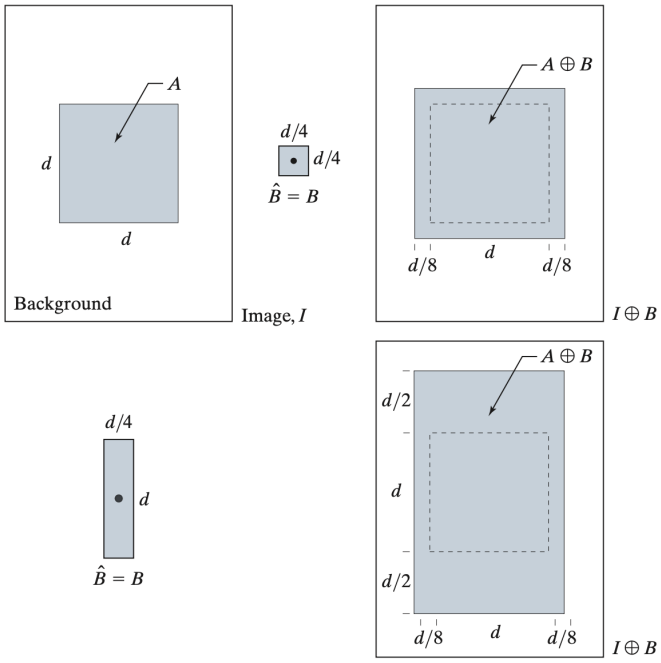


Figure 1.11: Top left: original image; top center: B set; top right: result of dilation between A and B; bottom left: another set B; bottom right: result of dilation between this last set and the original image [1].

The concatenation of an erosion and a dilation, in this order, gives the **opening** operation, considering two sets A and B this operation is defined as:

$$A \circ B = (A \ominus B) \oplus B \tag{1.39}$$

This process smooths the contours of the image and delete thin components without changing the element size. The concatenation of a dilation and an erosion gives the **closing** operation, considering two sets A and B this operation is defined as:

$$A \bullet B = (A \oplus B) \ominus B \tag{1.40}$$

This process connects weakly separated components in the image without changing the element size. In the figure 1.12 two examples of opening and closing operations are reported:

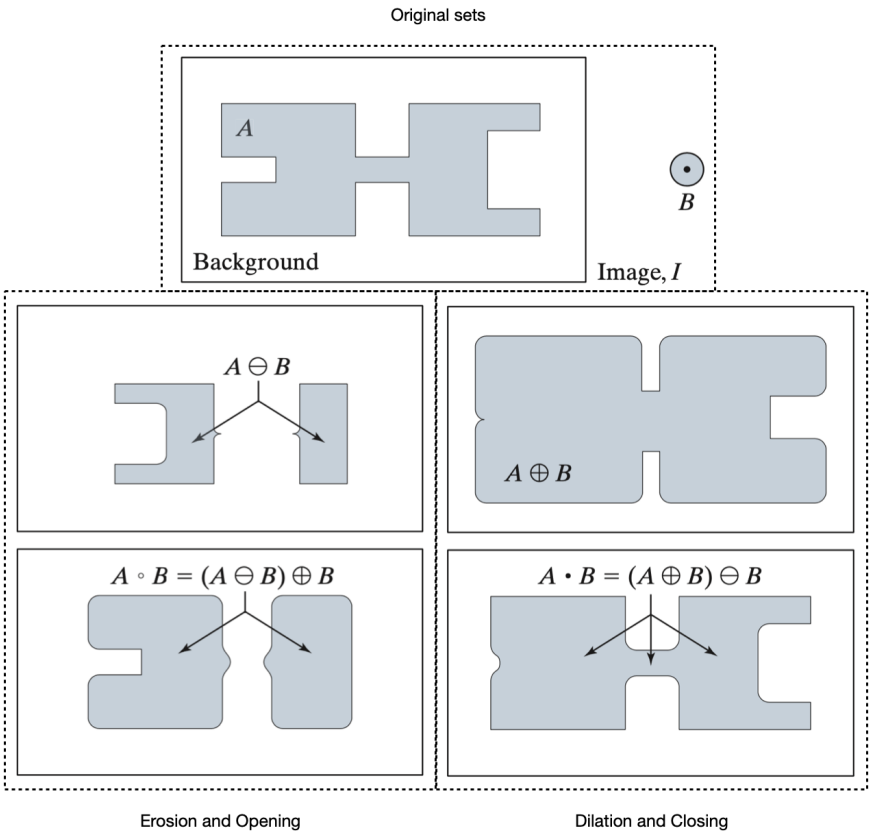


Figure 1.12: Examples of Opening and Closing on two sets A and B [1].

Edge detection

Edge detection is based on derivative filters, since these highlight edges in the image. The main edge detection algorithm is the **Canny** algorithm, it has the following features: well localization of edges points and return a single edge point as thickness. The Canny algorithm is composed by the following steps:

- Smooth the image with a Gaussian filter
- Compute the gradient, magnitude and angle images
- Apply non-maxima suppression to the gradient magnitude image
- Uses hysteresis thresholding to detect and link edges

Smoothing the image is necessary to reduce the noise, since the gradient computation is very susceptible to noise, the non-maxima suppression creates thin edges, that are desirable to obtain an accurate location, and finally the hysteresis thresholding connects weak edges with stronger ones. An example of the application of the Canny edge detection is reported in the following figure:

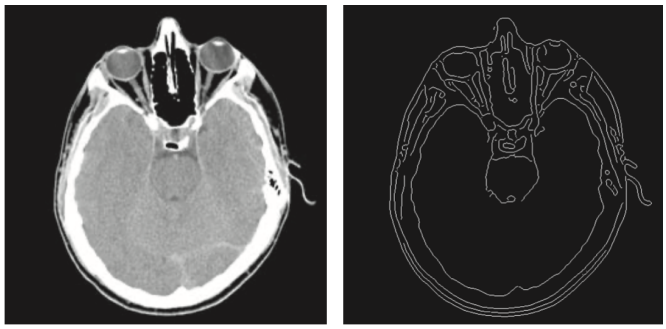


Figure 1.13: Result of the Canny edge detection algorithm [1].

Lines detection

The most used method to find lines in an image is using the **Hough transform**. Instead of analyse all the couples of edge points and evaluate the line passing

through them, is more efficient rewrite the equation of the line passing through (x_i, y_i) considering the *ab-plane*:

$$y_i = ax_i + b \quad \rightarrow \quad b = -x_i a + y_i \tag{1.41}$$

The ab-plane is called *parameter space*. To represent also vertical lines can be used a normal representation of this formula as follow:

$$\begin{gathered} x\ cos\theta + y\ sin\theta = \rho \\ y = \Big(-\frac{cos\theta}{sin\theta}\Big)x + \Big(\frac{\rho}{sin\theta}\Big) \end{gathered} \tag{1.42}$$

In the following figure an example of line detection using Hough transformation is reported.

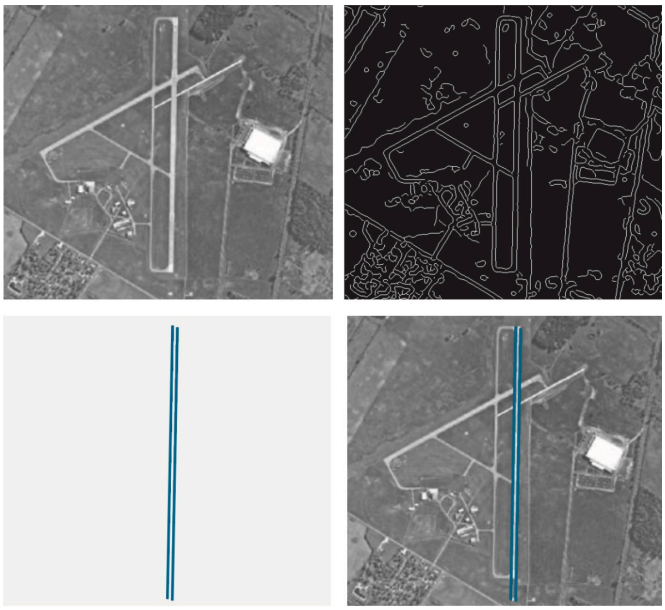


Figure 1.14: Top left: original image; top right: result of Canny edge detection; Bottom left: result of hough line detection; bottom right: overlap on the original image [1].

Image segmentation

One interesting process that can be applied to an image is to divide it into several regions, each one containing points belonging to the same category. Several categories can be identifies, for example the color category or the texture class. This

process is identify as **image segmentation**, to chose the regions through which the image is divided it follows two criteria: the similarity between pixel in the same region and the discontinuity between pixel in different regions. There exists a lot of different segmentation techniques for example segmentation by thresholding and watershed transformation. The **segmentation by thresholding** make use of the histogram of the image, choosing a threshold it divides the histogram into two regions. The choice of the threshold is free, but to automate it there exist a method, called Otsu's method. The *Otsu's method* is useful to find the best threshold to apply in the segmentation, it maximizes inter-class variance and minimizes intra-class variance. To help this method to give the best result some precautions can be applied to the original image, for example smoothing, combined it with edge detection and split the image in different regions and apply this method separately for each one. This method can be applied for multiple thresholding, finding n different regions instead of only two. In the figure 1.15 an example of segmentation by thresholding using Otsu's method is reported.

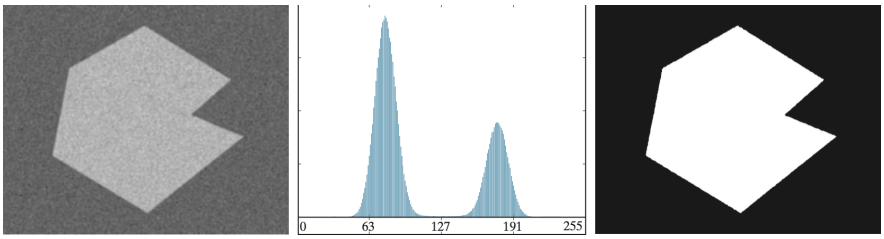
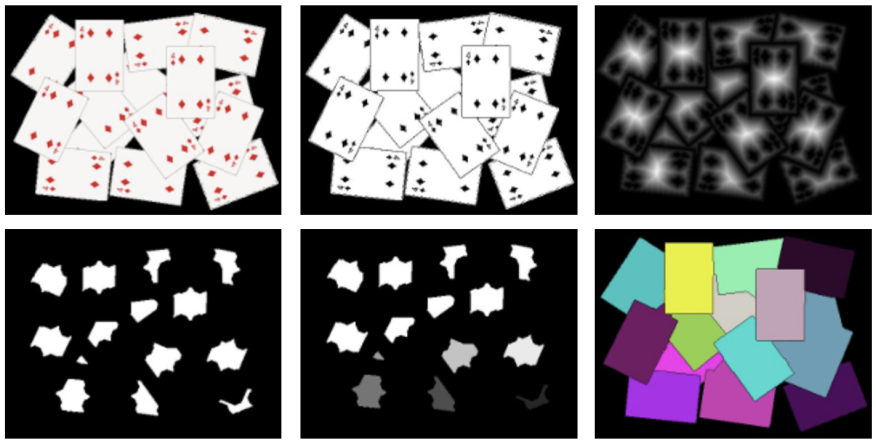


Figure 1.15: Left: original image; center: it's histogram; right: result of segmentation by thresholding using Otsu's method [1].

Another important method used for image segmentation is the **watershed algorithm**, it provides connected segmentation boundaries and components. The idea at the base of this algorithm is that a grayscale image can be seen as a topographic map, where the intensity is the height value. In this map there exist three different point categories: local minima, local maxima and middle points; the local maxima points are the salient ones because if we think of dropping some water on them, it could fall into two or more local minima. So the goal to the

algorithm is to find the local maxima and connects them in *watershed lines*. To segment an image using this algorithm the following steps they must be used: transform the original image into a binary one, apply the distance transform to find the distances from the edges, threshold it, assign a marker to each blob and finally apply the watershed algorithm that can be seen as a repetition of dilation operations. In the following figure an application of segmentation using watershed algorithm is reported:



A	B	C
D	E	F

Figure 1.16: A) original image; B) binary version; C) distance transform; D) threshold of distance transform; E) marker assignment; F) final segmentation using watershed algorithm [5]

Feature matching

The first method discovered to match the same object in different images is the **feature matching**. The idea is to detect salient points in the image, describe these elements, called *features*, and match them with the features of another image. The keypoints in general must be invariant to transformations and insensitive to illumination changes. To describe a keypoint a *descriptor* is used, it is based on different element of the image such as the color and orientation. The most used feature detection is the **SIFT** feature, it detects the keypoint and

makes the calculation of its descriptor. The SIFT feature, Scale Invariant Feature Transform, is invariant to translations, rotations and scales. To find a feature it follows four steps: scale-space detection, to find image locations that are invariant to scale change, detect keypoints, finding the maximums and minimums of the Gaussian filtered images, computes the orientation and calculates the descriptor, using a neighborhood of 16×16 pixels and values corresponding to the gradient of the image weighted by a gaussian function centered in the feature location. The matching of the feature can be done in several ways, one of the most used is the Nearest Neighbor Distance Ration, NNDR, that consider the ration between two different distance of nearby features in the features space; finally a threshold is used to consider only good matches. An example of feature matching using SIFT feature is reported in the following figure:

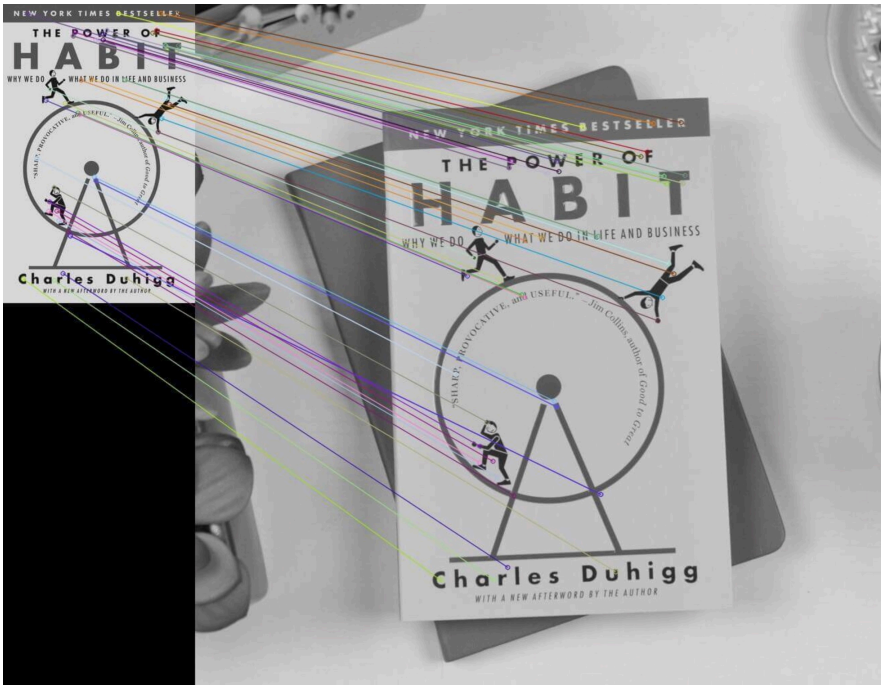


Figure 1.17: Left: object to be found; right: object in bigger and more complex environment [6]

1.3.3 High-level image processing

High-level image processing includes all the more complex tasks that can be applied to an image, like object recognition and detection. In the early 2000s the first algorithms that tried to solve these goals were born. For example for the face detection task one of the most famous developed algorithm is *Viola & Jones face detector*, that makes the use of a window moving over the image to find matching locations. This approach of sliding window over the whole image is slow and make the use of a lot of calculation. Another approach used to image classification is called *bag of words*, the idea is to divide a complex object in simpler features and collect all them in a codewords dictionary. Clustering all the feature in space is possible to find the centroid of each cluster of features. Evaluating each occurrence of each word, that is each feature find in the image, in the codeword is possible to classify the image based on the codeword that was most identified. Nowadays these techniques are outdated by the application of the so called *artificial intelligence*, which methods and results are better and also the execution is faster. This topic is so important that it is thorough in the next chapter.

Chapter 2

Principles of Artificial Intelligence

Artificial intelligence is the discipline that try to make an artificial system capable to simulate the human intelligence. This occurs through the optimization of mathematical functions inside computer programs. The branch of Artificial intelligence that is specialized to work with images is the *Neural Networks and Deep Learning*. A neural network in the general case is a network of artificial neurons through which some input signals are propagate to the outputs. It is composed by a layer structure, in the literature the nominative *deep learning* is used to specify that the network has 8 or more layer. To better understand how a neural network works the introduction to the learning problem and to the feed forward neural networks is necessary.

2.1 Learning phase

An human can make reliable prediction, for example in detecting a car in an image, thanks to his previous knowledge. Instead a computer program can make reliable prediction, considering the same example, only if it is trained during a training phase. The challenge that want to be achieved in this phase is to make a program able to output correct values beyond the previous knowledge acquired

in the learning phase. So make a program learn is different from let it memorize all the possible data in the training phase. There exists two different type of learning: supervised and unsupervised learning. The supervised learning is made of example of input-output pairs, in which the outputs are called labels, instead the unsupervised one makes use only of the input data. For the purposes of this master thesis the interest is addressed only on the supervised learning.

2.1.1 Supervised learning

The *supervised learning* problem assumes that the input data x be part of an input space, that contains all the possible input, and that exists input probability density function $p(x)$. The target values are generated as result of a function $f(\cdot)$. At the beginning of the learning phase p(x) and f(·) are unknown. To learn the function that relates the labels with the inputs a training dataset is used, it is composed by a finite number of labeled examples, independently sampled from the distribution $p(x)$. The estimate of the true function f is called $\widehat{f}$ and is called *learning model*. In general a learning model is a parametric function that depends on some trainable parameters tuned during the learning phase:

$$y \;=\; \widehat{f}(\mathbf{x}, W) \;\simeq\; f(x) \tag{2.1}$$

where y is the prediction of the model $\widehat{f}$ that depends on the parameters W. The training phase is the process of tuning of the model's parameters, usually done by mathematical optimization techniques. The goal of this phase is to produce an estimate of the true function that is able to make good prediction over the whole input distribution, not only for the inputs belonging to the training set. The *learning strategy* is called *empirical risk minimization (ERM)* that consist in minimize the empirical average of a particular function over the training dataset. This particular function is called *loss function*, that depends on the application and evaluates the *prediction error*: the error between the prediction and the true label considering a specific input. In the following equation the learning strategy

just illustrated is reported:

$$arg \min_{\widehat{f}} \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}\Big(\widehat{f}(x_n), t_n\Big) \tag{2.2}$$

where N is the dimension of the training set. After this phase the model is tested to check that it is able to fit in a good way also input outside the training set, if this is not the case, this situation is called *overfitting*[1] There exists different optimization method used for the function approximation, the most used ones are the *gradient descent* and its *stochastic* version.

Gradient descent

The **gradient descent** algorithm uses the training data to adjust the parameter of the learned function. The idea at the base of this algorithm is to modifying the function parameters in according to the gradient to find the minimum. This process is done because in general the functions are complex, may depends on thousands of parameters and usually are non-convex. The modification of the parameters is made following the steepest descend, the length of the step is tuned using a parameter η, called *learning rate*, usually small, e.g. 10^{-2}, to avoid the overshoot of the solution. There exist also another version of this algorithm, that is the **stochastic gradient descent**, it is the same algorithm but it uses for the computation of the gradient not the entire training set but only a subset of it, defined as batch, or a single data point. A sequence of the algorithm that covers the entire training dataset is called *epoch*, and on average the direction followed is exactly the steepest one. The advantages are that it helps to escape from local minima of the function and add a *regularization* factor, topic explained in a near future paragraph.

2.1.2 Regression and classification

There exist two type of problem that can be solved using supervised learning: *regression* and *classification*.

[1]This concept is illustrated in detail in the section 2.1.3

The **linear regression** problem consist in estimate a function that approximate the linear relation between a scalar, the output, and a generic vector, the input. The generic learned function is reported in the following equation:

$$y \;=\; \hat{f}(\mathbf{x}, \mathbf{w}) \;=\; \langle \mathbf{w}, \mathbf{x} \rangle \;+\; b \tag{2.3}$$

where y is the scalar predicted value, w is the parameter vector of dimension d, x is the input vector of dimension d and b is a scalar bias. The loss function used is the *mean squared error (MSE)* and the ERM for this problem becomes:

$$arg \; \min_{\mathbf{w},b} \; \frac{1}{N} \sum_{n=1}^{N} (\langle \mathbf{w}, \mathbf{x}_n \rangle + b - t_n)^2 \tag{2.4}$$

where t_n is the label corresponding to the input vector x_n. The closed form solution to this minimization problem is reported in the following equation:

$$w^* \;=\; (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t} \tag{2.5}$$

where w^* is the best solution, $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n]^T \in \mathbb{R}^{N \times d}$ and $\mathbf{t} = [t_1, ..., t_n]^T$.
The **classification** problem consist in estimate a function that can binary classify the input data into two sets. The generic learned function is reported in the following equation:

$$y \;=\; \hat{f}(\mathbf{x}, \mathbf{w}) \;=\; sign(\langle \mathbf{w}, \mathbf{x} \rangle \;+\; b) \tag{2.6}$$

where y is the scalar predicted value that can assume only $-1, +1$ as values, w is the parameter vector of dimension d, x is the input vector of dimension d and b is a scalar bias. The sign function helps to decide on which side of the hyperplane the point is located. Since the sign function makes the error flat almost everywhere, it is replaced with a smooth version of the unitary step, the *sigmoid* function. The solution of this problem can be found using GD and SGD methods already explained.
There exists also **non-linear learning problems** that can be solved mapping the problem into a *linear feature space*. Finding a good feature map is hard and also domain knowledge, a solution to this problem is using a neural network, that can learn the feature extraction method in its training phase.

2.1.3 Overfitting and regularization

Before see in detail the operation of the neural networks is important to explain better some concepts which have only been mentioned so far: *overfitting* and *regularization*.

The **overfitting** happens when the model instead of learning try to memorize the input data, this is a bad thing because it loses the powerful feature of generalize the input data distribution, that allows to make prediction outside the training set. One way to detect it during the training phase is to use another set, a validation set: after each epoch this dataset that contains unseen samples is used to evaluate the performances of the model outside the training data. If the loss on this dataset starts increasing most likely the model starts to overfit, in the following figure an example of training and validation losses are reported:

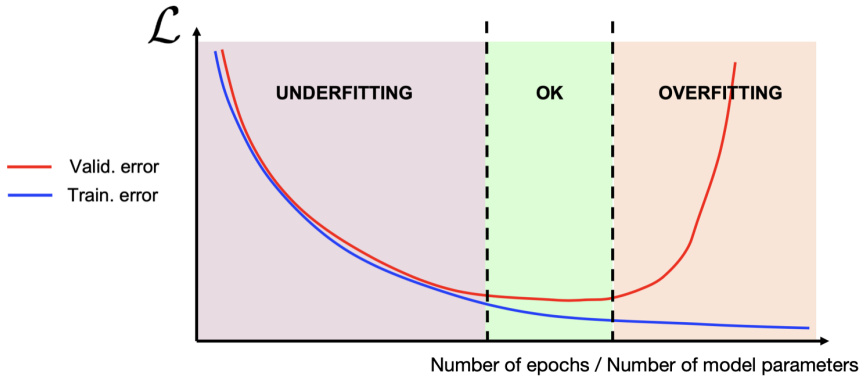


Figure 2.1: Example of training and validation losses during training [7]

So the overfitting happens both if the model is trained for too many epochs and if it has too many parameters. Another approach to solve this problem is adding a regularization term in the loss function.

Regularization helps to stabilize the model using a term that depends on the model's parameters, imposing constraints on them preventing overfitting. Overfitted models are unstable, so a small variation of the abscissas leads to a large variation of the ordinates, instead stable models do no overfit. The most used

therm is the *Tichonov* or *L2* regularization:

$$R(w) = \lambda ||w||_2^2 \tag{2.7}$$

so the process try to minimize not only the ERM but also the squared norm of the weights, making the overall function more convex, so more stable.

2.2 Feed forward neural networks

A Feed forward neural network is a network made by a series of nodes, called neurons, with one or more inputs and outputs, used both for regression and classification problems. Before going into the detail of this structure, to better understand how it functions an in-depth analysis of the *perceptron* algorithm is needed.

2.2.1 Perceptron algorithm

The **perceptron** algorithm is used for binary classification, so it's decision function is the same as the equation 2.6. The algorithm linearly separate the data using an hyperplane, parametrized by $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$, where w gives the orientation and b the distance from the origin. The explanation of the decision function is the following: if a point lies on the hyperplane the result of the equation is zero, otherwise the it represent the projection of $\mathbf{x}$ onto $\mathbf{w}$, in which the length is represented by the argument of the function and the direction is given by $\mathbf{w}$.

The perceptron *idea* to find the best $\mathbf{w}$ is to start from $\mathbf{w}^{(0)} = \mathbf{0}$ and adjust it any iteration of the algorithm until all the data are separated correctly. Denoting with y_i the output of the decision function corresponding to the input $\mathbf{x}_i$ and with t_i the true label associated to the input, is possible to see that for a correct classification the following relation must be true:

$$y_i t_i > 0 \qquad where \;\; y_i = (\langle \mathbf{w}^{(k)}, \mathbf{x}_i \rangle + b) \tag{2.8}$$

where k represent the current iteration of the algorithm. So to correctly classify all the training set the relation just illustrated must be true for all the data in the set: $y_i t_i > 0,\ \forall n$. The loss function used to find the updating rule of the weights is the reported in the following equation:

$$\mathcal{L}(\mathbf{w}, \mathcal{D}) \ = \ \sum_{n=1}^{N} l(\mathbf{w}, \mathbf{x}_n) \ = \ \sum_{n=1}^{N} max[0, -(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) t_n] \tag{2.9}$$

where $\mathcal{D}$ represent all the training dataset. If a point is already correctly classified the loss is zero, so only the uncorrected samples contributes to the weights update, using a gradient descend strategy to minimize the total loss function. The gradient of the loss is reported in the following equation:

$$\nabla \mathcal{L}(\mathbf{w}) \ = \ \sum_{n \in \mathcal{M}} -\mathbf{x}_n t_n \tag{2.10}$$

where $\mathcal{M}$ represent the set of misclassified points. The learning rule is to take a single misclassified point and update the weights using the rule reported in the following equation:

$$\mathbf{w}^{i+1} = \mathbf{w}^i + \mathbf{x}_n t_n \tag{2.11}$$

At any iteration the loss function is always improving on the misclassified sample.

2.2.2 Structure and training

The perceptron algorithm has some weak points: it fails if the data are not linearly separable and only uses halfspaces. As already mentioned for the non-linear learning problems, the solution is to make use of a feature map, but a good one is hard to find. The true solution for this problem is to let learn the feature map, this is possible thanks to the particular architecture of the neural networks: a composition of multiple simple function that can increase the abstraction make use of a multi-layered structure. A basic neural network architecture is the *feed forward neural networks*, *FFNN*, composed by small neurons, each one based on the Perceptron, in which instead of the sign function there is a non linear function, called *activation*, is used.

The input-output relation of an artificial neuron is reported in the following graph:

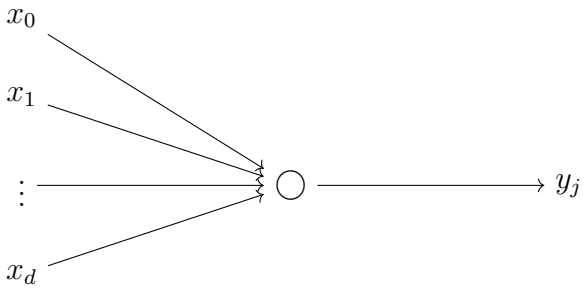


Figure 2.2: Representation of the input-output relation of an artificial neuron

where y_j represent the output of the neuron j, calculated using an activation function f, as reported in the following equation:

$$y_j = f\Big(\sum_{i=1}^{d} w_{i,j}x_i + b\Big) \tag{2.12}$$

In the following figure a basic structure of a FFNN with three inputs, one hidden layer with four neurons and two outputs is shown:

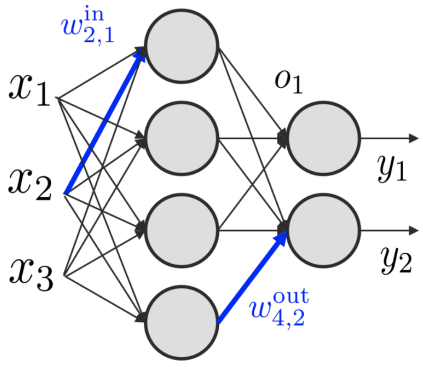


Figure 2.3: Example of FFNN with three inputs, x, two outputs, y, and one hidden layer with four neurons [7]

The final activation function, used in the nodes that are in the output layer, is different from the one used in all the other nodes of the network and it depends on the learning problem for which the network is trained: for example linear, relu, sigmoid or softmax. It has been demonstrated that a single-layer FFNN can

approximate any function given a suitable activation function, but the number of hidden neurons required is exponential with respect to the input space. A first solution to reduce the complexity of the resulting network is to use a layered structure with fewer neurons per layer. There are also some parameters called *hyper-parameters*, that are parameters that are not learned during the training phase: for example the learning rate, the mini-batch size and the size of the network in number of neurons and layers. These parameters in general are chosen based on the experience, but some consideration on the learning rate can be done: if it is too small, there is a slow convergence during training, vice versa if it is too high there could be a divergence. Also for the number of parameters of the network there exist some considerations: if a network is too small is probably that there is no convergence, this situation is called *underfitting*, instead if the number is too large, the training is slow, difficult and overfitting is probable.
Let's analyze how a FFNN is trained, this is done making the use of **backpropagation**, that is an efficient way of calculate the gradients. Since the gradient descend method needs the gradient of the loss function over all the parameters of the network, this calculation can be very expensive due to the potentially very large number of parameters. The idea is to calculate the gradient of the loss function with respect to each weight of the network and to use the chain rule of derivatives, this process is more efficient. So first there is a forward pass of the input through the network, then the results are used to backpropagate the gradient to the first layer of the network.

2.2.3 Limitations of FFNN

This basic formulation of the neural networks has some limitations: first of all a NN needs an huge number of parameters, especially operating with images. An example is a NN with a single hidden layer of 100 neurons that classify an image of shape 500x500 RGM image: the input is a vector of dimension $500 \cdot 500 \cdot 3 = 750000$, the number of weights only in the first layer is $750000 \cdot 50 = 37,5$ millions of parameters. This is computationally expensive beyond that memory expensive.

Another problem is that the structural information in the input is lost during the vectorization of the input, losing the spatial information typical of the image. The solution to these problem was found in the *convolutional neural networks*.

2.3 Convolutional neural networks

2.3.1 Structure and training

In a Convolutional neural network, CNN, the input structure is preserved, no modifications of any kind to the input are performed to preserve its structural information. Between two consecutive layers the operation that is performed is the *convolution*: one or more filters slide over the first layer executing the convolution operation with the corresponding values of the layer and finally the activation function is applied. These filters contain the learnable parameter that are modify during the training phase. The convolution can be executed in one, two or three dimensions, it depends on the input shape: for example in the case of an image as input the convolution is operates in two dimensions and the filters in the first convolution layer have a shape of the type [M $\times$ N $\times$ C], where C is the number of channels of the input image. The depth of the next layer of the network depends on the number of filters used for the convolution with the previous one; so if D filters are used for the convolution with a layer in the network the next one will have a depth equal to D. In this case of an input depth C the 2D convolution uses a filters with a shape of [M $\times$ N $\times$ C] and the movement of the filter is made in 2 direction, in the case of 3D convolution the filters depth is much smaller than the depth of the input and the movement is made in all 3 the dimensions. A visual representation of a convolution is reported in the following figure:

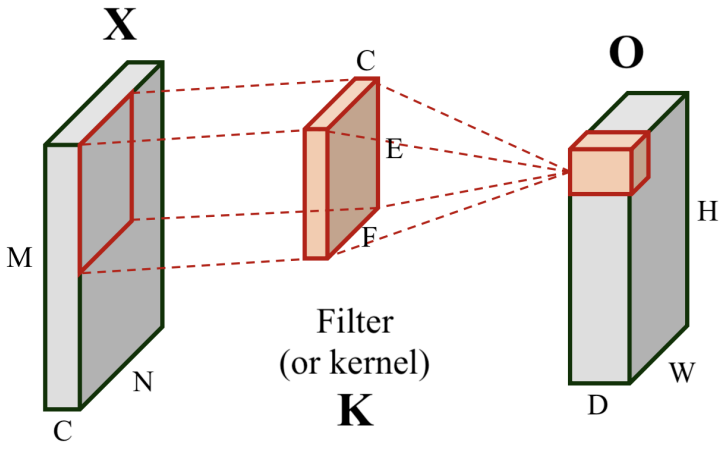


Figure 2.4: Example of a convolution between an input layer of shape [M × N × C] with D kernels of shape [E × F × C], the result layer has a shape of [H × W × D] [7]

For completeness the formulation just described is reported in the following equation:

$$o_{i,j} = f\Big(\sum_{c=1}^{C}(K_c * X_c)(i,j) + b\Big) \tag{2.13}$$

where $o_{i,j}$ is the output refering to the (i,j) input location, the convolution operation is made for all the input depth C, the bias b is added and the activation function f is applied. The equation of the convolution applied in the spatial domain is reported in the equation 1.31 in the first chapter, but to be more clear in the following equation the generic formula for the convolution in CNNs is reported:

$$(K_c * X_c)(i,j) = \sum_{f=0}^{F-1}\sum_{e=0}^{E-1} X_c(i+f, j+e)\cdot K_c(f,e) \tag{2.14}$$

The result of a convolution with one filter compose only one channel of the next layer, called activation map, so to obtain a depth D of the next layer the convolution operation needs to be repeat with D different filters, this number is an hyperparameters of each specific layer. The non linear map f, used element-wise to the activation map, gives a feature map.

To define all the **structure** of a network know the output dimension of a layer

is necessary, but before the definition of some terms is necessary:

- *Stride*: number of steps that the filter makes when is shifting on the layer, minimum 1 and maximum the size of the filter.
- *Padding*: dimension of the frame in the border add at the edge of the input tensor so that the filters used can fit an integer numbers of times.

Now we can define in the following equation the dimension of the output O using a stride S, a padding P, a filter size F and an input size d:

$$O = \frac{d - F + 2P}{S} + 1 \tag{2.15}$$

Since the learning of the feature is made in an hierarchical way, sometimes is useful to reduce the dimension of the tensors between two consecutive layers, this is made using Another type of layer in a CNN that is the *pooling* layer. The reduction of the tensors can be done in two ways: using a stride grater than one or using a pooling layer. In a pooling layer the function used are usually the following ones: minimum, maximum and mean functions. Most of the times the pooling layer map a region of dimension RxR with stride R, this introduce also invariance to small translation, very usefull for classification problems. In the following figure a generic CNN architecture is reported:

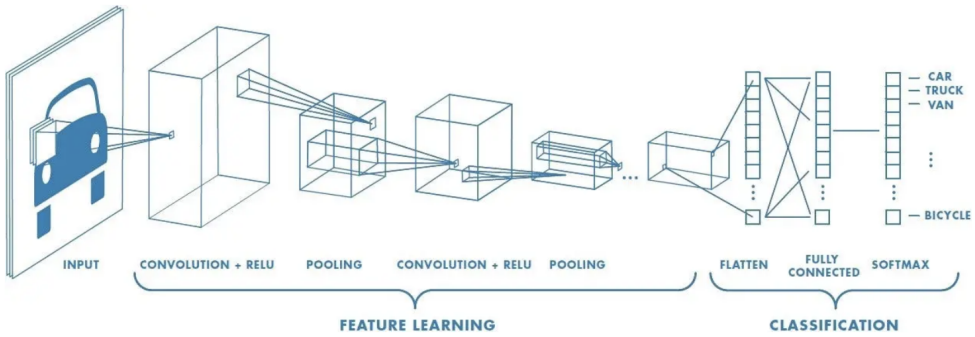


Figure 2.5: Example of a generic CNN [8]

Usually in a CNN, as is possible to see in the figure above, going in depth the numbers of channels tends to increase to learn as many features as possible. In the

convolutional layers feature are learned, instead in the final part the feature are linearized and a small FFNN is present, this is used to classify the feature in the correct number of output desire for the specific application. The **training** of the CNNs uses the technique of the backpropagation, as the FFNNs, but modified for the different structure used. Since this type of network is very powerful in the last years other optimizers with respect to the gradient descend have been discovered, below the most used optimizers will be listed, without explanation regarding their operation:

- SGD with momentum
- Nesterov momentum
- AdaGrad
- RMSprop
- Adam

2.3.2 Overfitting and batch normalization in CNNs

To prevent *overfitting* in a CNN there exist some techniques:

- Early stopping
- Data augmentation
- L2 regularization
- Dropout

Recalling the figure 2.1, if a network is trained too long, in terms of epoch, the overfitting occurs, because the network memorize the input-output relation of the limited data in the training set. The *early stopping* is a technique that uses another set, called validation set, and at the end of each epoch computes the loss on this set, to detect the overfitting and so stopping the training before it occurs.

Another technique is to expand the data in the training set, because the more data you have the less likely it is to overfit, because there are more data to fit. This augmentation is made in artificial way, for example working on images the augmentation can be done using some techniques exploited in the first chapter, for example adding noise, making rotation, translation or applying some filters. As for the FFNNs, a technique to reduce overfitting is the application of the regularization, but also another idea comes out: the *dropout* idea. the *dropout* consist in randomly remove some of the neurons at every epoch, excluding also its incoming and outgoing connections. This application allows all the neuron to learn features and better accuracies have been demonstrated using it.
One problem that the leyered structure brings with it is the internal covariate shift problem: this happens when the training data distribution do not represent the true distribution of the data. To be more clear let's make an example: training the network with data all of the same color and test it with data containing all colors. In this case the weights of the network are not optimized for the full input distribution, so they may need to be retrained. In CNNS this problem happens in every layer during the training phase, because every layer needs not only to learn but also to compensate for this shift of the previous one. To overcome this problem, the use of *batch normalization* is needed: it aims to control this phenomena normalizing the layers' inputs by re-centering and re-scaling with tunable parameters also learned during the training phase.

2.3.3 Residual networks

One of the last block discovered in the last years that can be used to build the structure of a convolutional neural network is the *residual block*. Going into deep networks, that can learn more in terms of feature, there is a problem related to the vanishing of the gradient because the regularization reduces the weights at any iteration by default. This problem was visible also in the performances of the networks, because deeper networks performed less well than their shorter versions. The solution was found using a *residual unit*, this block is a variation

of the standard layer already explained, that adds the idea of skip connection, as reported in the following equation:

$$\mathbf{o} = f(x) + x \tag{2.16}$$

since the x is known, the network needs to learn only the $f(x) = \mathbf{o} - x$, called residual, usually a small quantity. In the following figure a visualization of the residual block is reported.

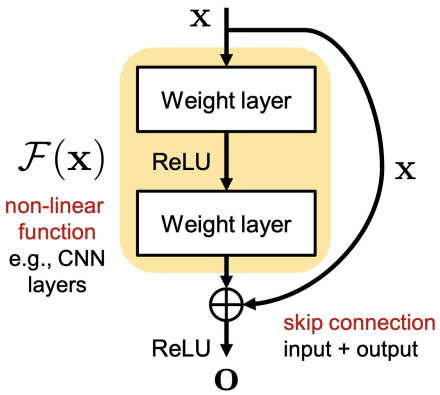


Figure 2.6: Example of a residual block [7]

So the residual networks allow to reach unprecedented accuracy on a variety of tasks and are almost the state of the art for the CNN architecture.

2.3.4 Benefits of CNNs

The CNNs have some advantages and features with respect to the FFNNs:

- *Local connettivity*: The size of the filters is much smaller than the input size and each neuron of a subsequent layer depends only on few neurons of the previous one, this characteristic allows to learn features in a hierarchical way, increasing the receptive field layer by layer, this is much more efficient and fast than dense FFNN. In the following figure two examples of perceptive field are reported:

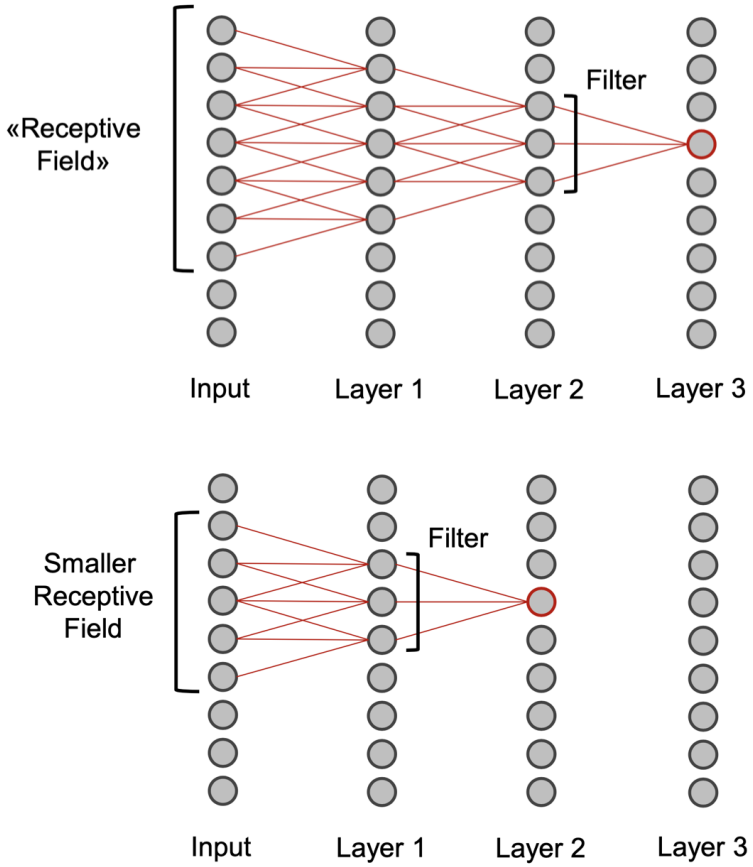


Figure 2.7: Example of the perceptive field of a neuron of the third layer, upper figure, and a the one of a neuron of the second layer

As is possible to see the perceptive field of a neuron in the third layer is bigger with respect to the one of a neuron in the second one, so the more layers the network has, the more the features learned concern a greater part of the input, learning global patterns.

- *Weight sharing*: each filter apply the same weights over the whole input, this allows a big save in terms of numbers of weights to learn with respect to FFNNs, also due to the using of spatial structure there is an higher efficiency in encoding information. This type of structure can go deeper in analyze the input patterns, requiring less memory with respect to a FFNN architecture.

- *Translation equivariance*: given two function f and g the equivariance means

that these two generate the same output, with the same input x, whatever the order of application of the two, the equation representation is:

$$f(g(x)) = g(f(x)) \tag{2.17}$$

In CNNs there is the equivariance between the convolution operation and the translation one, this means that if a pattern is present in another part of the input, the corresponding feature leaned is shifted too, but still present. This makes CNNs very robust to recognise the same pattern in any location of the input. However the convolution is not equivariant to rotation. [7]

2.3.5 Yolo architecture

One of the most famous and efficient convolutional neural network architecture used for object detection is the *YOLO* architecture. The object detection is a computer vision task in which the goal is to detect and classify one o more classes of objects in an image and localize them using a bounding box around each prediction. This architecture was presented in 2016 by Joseph Redmond et al., is faster with respect to other proposed architecture because it makes the prediction of the bounding boxes, the probability and the class to which it belongs in only one phase. The model divides the input image into an S $\times$ S grid, a grid cell is responsible for detecting that object if the center of an object falls into that grid cell. Each cell predicts B bounding boxes and assign to them a score, called *confidence score*. This score tells the probability that the box contains an object, *Pr(Object)*, and the accuracy of the dimensions and localization of the box with respect to the object, IOU_{true}^{pred}. Each bounding box consist in 5 prediction: x, y, w, h and the confidence; where x and y are the coordinate of the top left corner of the box, w and h the width and height, the confidence is $Pr(Object) * IOU_{true}^{pred}$. Each grid cell also predicts C conditional class probabilities, $Pr(Class_i|Object)$, that are conditioned on the grid cell containing an object. A non-maximal suppression can be used to fix detections of the same object by multiple cells. The overall *class-specific confidence scores* for each bounding box is described in the following

equation:

$$Pr(Class_i|Object) * Pr(Object) * IOU_{true}^{pred} = Pr(Class_i) * IOU_{true}^{pred} \quad (2.18)$$

This score encode both the probability that a specific class appears in the box and how well the predicted box fits the object. In the following figure a schematic representation of the operation of the YOLO neural network architecture is reported:

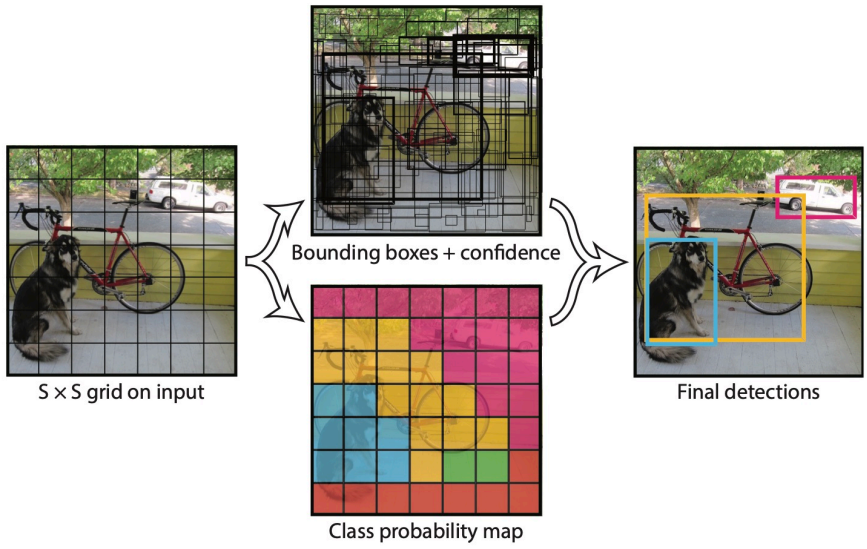


Figure 2.8: Schematic representation of the operation of the YOLO [9]

The overall *architecture* is reported in the following figure:

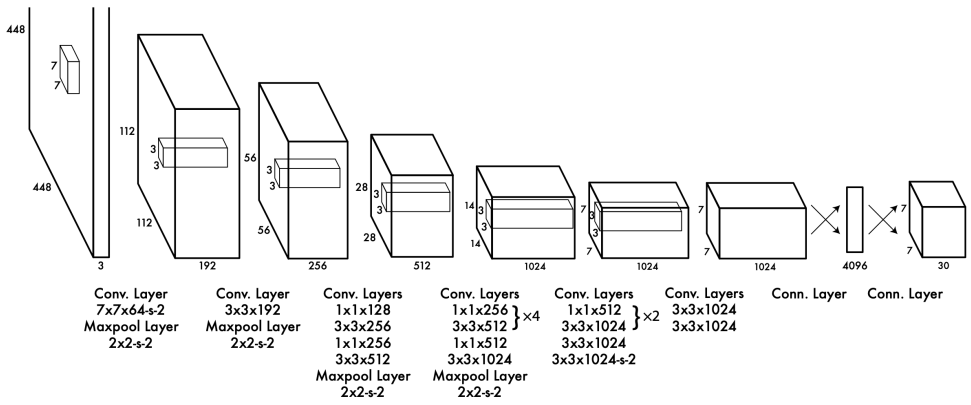


Figure 2.9: Representation of YOLO architecture [9]

The network takes an input of shape 448 x 448 x 3, it has 24 convolutional layers

and ends with 2 fully connected layers, it alternates 1 x 1 convolutional layers to reduce the feature space from preceding layers. The final output, the one of the last fully connected layer, has a shape that depends on the parameters chosen during the training, for example in the original paper they use the following parameters: S equal to 7, B equal to 2; so the final output has the following shape: $SxSx(B * 5 + C)$, where C is the number of classes in the training set.[9]
One of the last releases of the YOLO architecture is the YOLOv8, released in 2023 by Ultranalytics company. This version provided five scaled version: YOLOv8n (nano), YOLOv8s (small), YOLOv8m (medium), YOLOv8l (large) and YOLOv8x (extra large). It also can achieve not only task of object detection but also segmentation, pose estimation, tracking, and classification, using an input size of 640 pixels. This architecture can be used in personal projects without training it from zero, it is more convenient to download a pretrained version, in which all the feature are already learned, and train it on a personal training dataset.[10]

Chapter 3

Quality control on products

3.1 Quality control through image analyses

With today's level of competition between companies operating in the same sector, the *quality control* in industries is a really important aspect, not only to avoid product returns due to defects in the product itself, but also to ensure greater product competitiveness. All industries are trying to automate it, not only to reduce the cost of this operation, but also to make the control process more precise. Automate this task leads to some evaluation criteria that are the same for all the samples and that not depends on the light condition of the environment or the personal evaluation of the human that see it. The topic just explained in the previous chapters are used for this aim, for example using a camera: it can photographs the object to be controlled and check that some particular parameters are correct, like the dimensions, the shapes, the contours, the colors and so on.

3.2 Screen quality control

One example of application of computer vision for quality control purposes is the screen control. The following project was developed in Vimar spa at Marostica during an internship.

3.2.1 Motivation and setup

Motivation

For this company seeing the correct functioning of the screens is necessary because, for the purposes of this particular screen, the input signals is analog, so very subject to interference, and also to check the DAC inside the device itself.

Setup

The screen that were used in this project have a resolution of 800 x 480 and since the input signal to the device is analog, there is no way to check the correct image at the end of the communication, also verify the correct conversion from analog to digital signals is not possible, so there is no other way than see the result direct as image on the screen. The setup used is a completely darkened box in which a camera and a screen at 10 cm of distance are placed. The box was necessary because the resulting images captured by the camera were very subject to light reflections. The camera used is the OAK-1 camera, which has a resolution of 12Mp and a FOV of 81°(D)×69°(H)×55°(V).

3.2.2 Quality control approach

The screen quality control consisted in two phases: the first was to check a gradient color image and the second an EBU colour bar, used in the television sector. But before making comparison between the true images and the ideal ones, the localization of the screen in the image was needed.

Screen localization

The screen in the image was not perfectly aligned with the contours of the image itself and also was subjected to some distortion due to the no perfect calibration of the camera. A camera calibration process was needed to find the distortion coefficient to apply on the original image to find its undistorted version. The following figure shows the screen seen from the camera in the darkened box.

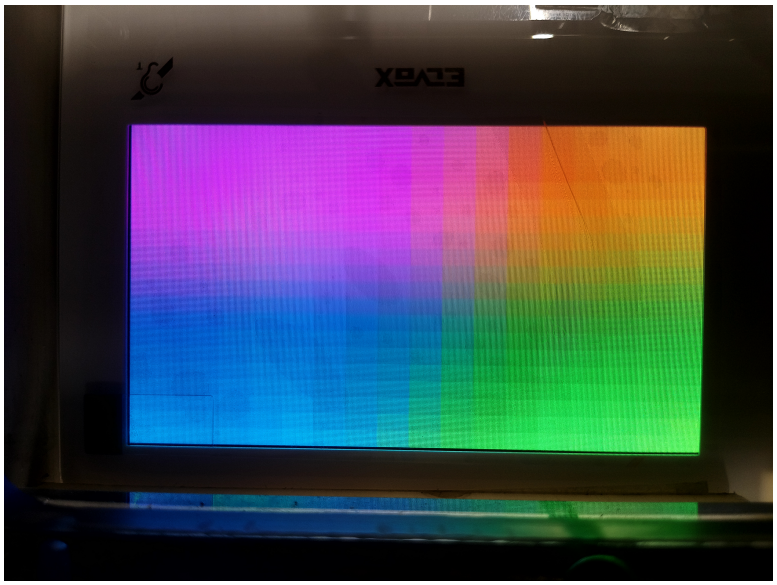


Figure 3.1: Screen visualized by the camera in the darkened box

A first approach I used for the screen detection was using some segmentation techniques, for example the watershed algorithm, but I obtained only bad results. This was due to the fact that the screen has a low resolution and the camera instead has a very good resolution, so the camera was able not only to detect each pixel with good precision but also the black space between each pixel, this leads to the presence of a black grid all inside the screen that don't allow the good segmentation. Resizing the image was not enough to solve the problem because, also using different techniques of interpolation, the grid was still present. The solution that I found was to use a thresholding technique of it's gray version to divide the screen and the plastic around it. Doing so also the reflex on the boundary of the image are selected. After I used some morphological operator to delete the grid inside the screen, the result is reported in the following figure:

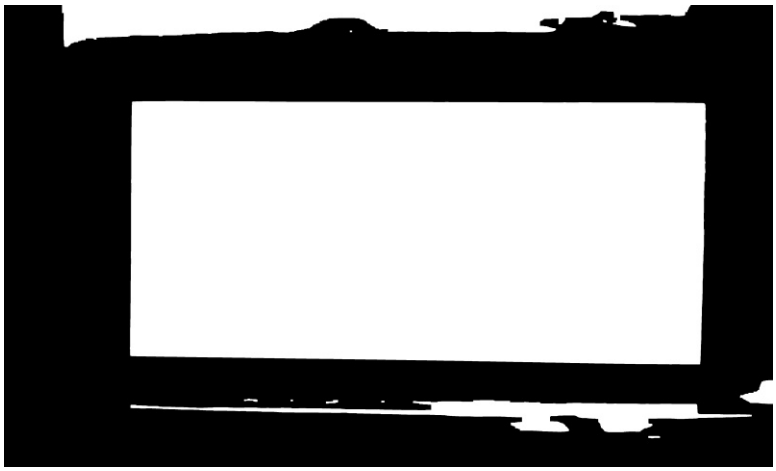


Figure 3.2: Image thresholded

At this point for an human eye the location of the screen is intuitive: is the main rectangle in the center of the screen. For a program the location is no intuitive so I created an approach to find the exact corner points: detect the lines in the image, check if they are vertical or horizontal, eliminate all the other lines, find the four lines closer to the image center, two verticals and two horizontals, and finally calculate all the four intersections, so the corners of the screen, of these lines. Using a geometric transformation I isolated the screen, resizing it to the real shape of 800 x 480; the result is reported in the following figure:

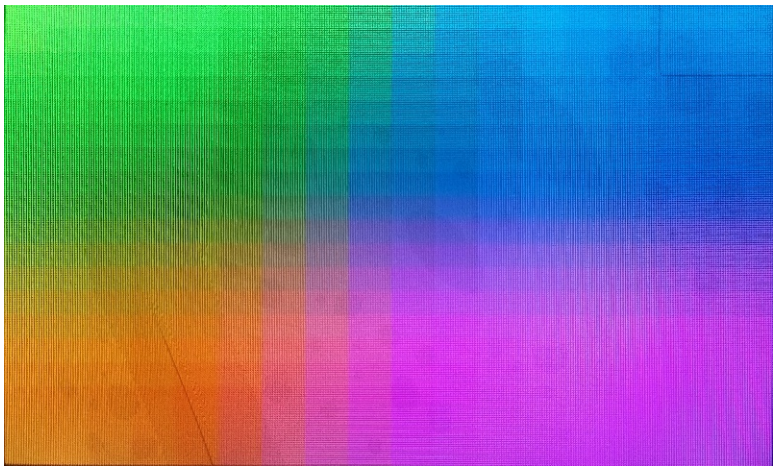


Figure 3.3: Screen detection visualizing the gradient image

Gradient image analyses

Once the screen was detected and segmented from the image the strategy that I found was to make the comparison between an ideal image of the gradient and the true one, since the true image changes a little bit from device to device. To compare the two I decided to convert the two images in HSV format, Hue Saturation Brightness, and to compare the H channels of the two; if the mean of the difference of the two H channels is below some threshold the test was passed. An example of true and real H channel is reported in the following figure:

Figure 3.4: Comparison between the H channels of the ideal image, on the left, and a sample of a real one, on the right

EBU colour bar

The second test I had to solve was the EBU colour bar control. This technique was used in the television sector especially during the analog transmission of the signal. Some color bar are projected on the screen to see if the DAC function in the proper way, avoiding the superposition of the bars. In the following image a sample of EBU bar on the screen is reported:

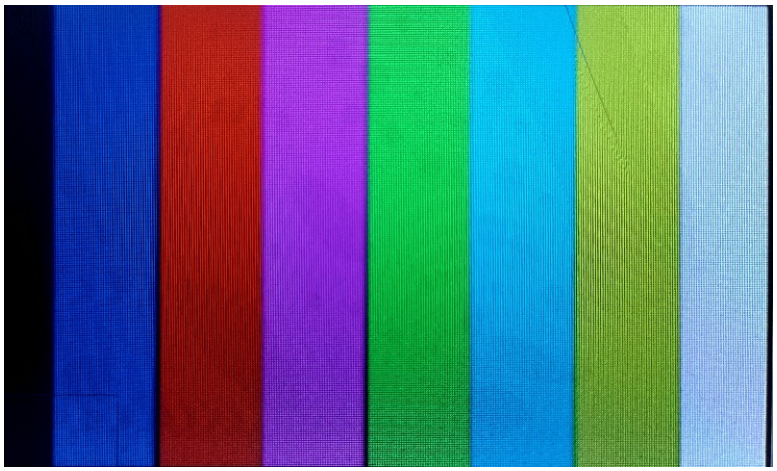


Figure 3.5: Screen showing the EBU color bar

In this test my approach was to operate, as in the first test, with the HSV images, by calculating a mask for each band and analyzing any line present between the two, if this had a thickness lower than a threshold then the test was passed. By joining the masks of each band together the following mask is obtained:

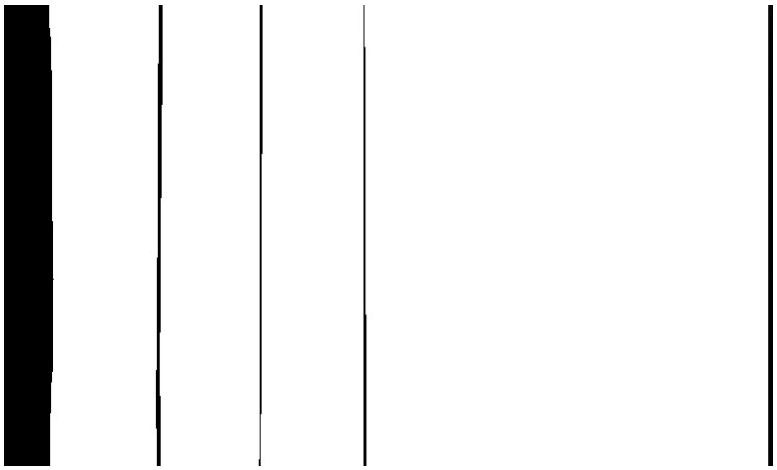


Figure 3.6: Mask of the color bars obtained

As is possible to see on the left there is the black bar and three lines are present in this particular sample. In this case a little bit of superposition is allowed, the thickness of each line is below the threshold, so the test was passed also in this case.

Chapter 4

Conveyor tracking

In industries there are multiples application of the robotics which aim to automatize repetitive and aliening jobs: especially moving objects from a place to another during the processing phase. To automatize the process of picking and placing objects from a conveyor belt to another place, a robot must be used. It is a powerful machine that has a defect: it is blind; so it needs at least one sensor that tell him where the object is located, to be able to take the object in the correct way. There exist a lot of sensors able to retrieve the position of the object in the conveyor belt: for example ultrasonic, inductive, capacitive and optical sensors. All these sensors are electronic ones, so they do not need an external program that process the information captured to understand if and where there is the object, but their output contains basic informations. Another type of sensor that can be used in an industrial environment is the camera sensor: the camera captures the light in the environment and gives as output an image. To retrieve the presence and the position of a specific object in the scene an ulterior program is needed. So, at first glance, the use of a camera seems to be worst with respect to the use of all the other sensor listed earlier, but this last choice can capture more informations with respect to linear sensors. Nowadays the use of cameras in industries is increasingly frequent, thanks to the use of CV and especially AI that help to retrieve a lot of informations on the objects in an image, like the number of objects found, their position, orientation and so on. Before going into

the details of the *Conveyor tracking* problem, an overview of the existing robots and the main control techniques used is carried out.

4.1 Robotics and control: general case

In robotics there exist two main categories of robots: mobile robots and robots manipulator. A mobile robot is an agent that can moves in the environment, most of the time used for transport materials over medium and long distances, whose content is inserted by an human operator or a robots manipulator. A robot manipulator instead has a fixed base and can move only in a limited space, but has the capability of picking and placing object very precisely. For the purposes of the conveyor tracking task the focus is limited on the category of robot manipulator.

4.1.1 Manipulator robots

Referring to an industrial environment an *industrial robot* is an automatically controlled, reprogrammable multipurpose manipulator, programmable in three or more axes for use in industrial automation applications. The whole *industrial robotic system* in addition to the robot also includes the end-effector and any external auxiliary devices that support the robot task.[11]

Mechanism: coupling and degrees of freedom

A robot is a mechanical structure composed by a series of parts, called links. A set of links form a *kinematic chain* if they are connected by *kinematic pairs*, this type of connection allows relative motion between links. In the space an unlinked rigid body has 6 degrees of freedom, 3 for rotations and 3 for translation, a coupling between two links by a kinematic pair reduces the number of degrees of freedom of the relative motion. The kinematic pairs can be classify using the number of degrees of freedom allowed by the coupling: for example the two most used pairs of class one are the revolute pair, that allows one rotational dof, and the prismatic pair, that allows a translational dof. There exist a lot of different kinematic pairs

from class one to class five, all used to form a kinematic chain, if the frame is defined it is called *mechanism*. A kinematic chain can be open or closed and it has a number of degrees of freedom that depends on the kinematic pairs used to build the chain, there exist a general equation that can be used to find the dof of a spatial mechanism. This equation is called *Kutzbach equation* and has the following formulation:

$$n = 6(m-1) - 5c_1 - 4c_2 - 3c_3 - 2c_4 - c_5 \tag{4.1}$$

where n is the number of dof, m the number of links, that includes also the frame, and c_i is the number of kinematic pairs that belongs to the class i. However this formulation could fail in case of wrong application, redundant constraints and hyperstatic mechanism.

Robots classification

There exists two type of robot categories: serial and parallel robots, the difference is that a serial robot has only one kinematic chain from the frame to the flange, instead the parallel one has more than one kinematic chain. The structure of a **serial robot** is composed by an arm, which is responsible for the position, a wrist, which instead manages the orientation, and a end effector, which allows the interaction with the environment. The kinematic pair used in this type of robots are the revolute and prismatic pairs. If a serial robot has more than six degrees of freedom is said to be redundant, specifically is said intrinsically redundant, because the dimension of the operational space, six, three for rotation and three for translation, is smaller than the dimension of the joint space, dof of the robot. The classification of the robotics arms is based on the first 3 dofs of the mechanism, composed using revolute (R) and prismpatic (P) joints:

- Cartesian (PPP)
- Cylindric (RPP)
- SCARA (RRP)

- Polar or spherical (RRP)
- Articulated or anthropomorphic (RRR)

The structure of a **parallel robot** is composed by more kinematic chains, responsible for the position and orientation, and a end effector to interact with the environment. The kinematic pair used in this type of robots are the spherical pair and universal joint. One of the most used parallel robots in industries is the *Delta* robot, can be composed by three or four kinematic chains and for each one is composed by one revolute and two universal joints. An example of Cartesian and Parallel robots is reported in the following figure:

Figure 4.1: Example of Omron cartesian robot, on the left, and a parallel Delta robot, on the right [12]

Regarding the end effector there exist infinite many different types of grippers and tools that can be used. The gripper allows to pick and place objects in the workspace and usually is designed ad-hoc for the specific application.[13]
The **evaluation criterias** for the choice of a robot are the following ones:

- Precision, defined with different specifications:
  - Accuracy: denotes how much the result of the movement is near to the ground true desired;
  - Repeatability: tells how much the same desired movement will gives the same result between one movement and another one, with the same

target;

- Sensibility: the minimum movement that can be carried out;

- Payload: total weights that can be moved, formed by the tool plus the object to pick;

- Workspace: is the region described by the origin of the end effector frame, when the manipulator executes all possible motions of its joints; it is also divided into two workspaces: reachable workspace, all the position that can be reached with at least one orientation, and dexterous, all the position that can be reached with any orientation;

- Performance: described in terms of velocity, acceleration and cycletime.

4.1.2 Robot pose

Each robot link is a rigid body, in which the distance between any pair of points remains the same regardless of the forces applied on it. A rigid body in space can be completely described by its pose: position and orientation.

Rotation matrices

To representing the orientations the **rotation matrices** are used. Let's consider two frames, one rotated relative to the other, and a point P in the space. The described setting is reported in the following figure:

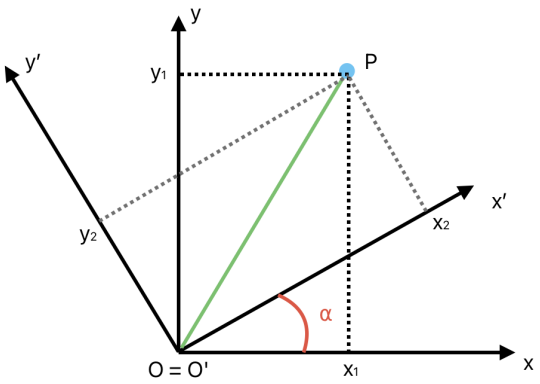


Figure 4.2: Example of a point P visualized from two rotated frames

The point P in the space has coordinates x_1 and y_1 with respect to frame one and x_2 and y_2 with respect to frame two. The generic relation between these two points is reported in the following equation:

$$P_1 = [R_{2,1}]P_2 \tag{4.2}$$

where $[R_{2,1}]$ is the rotation matrix, matrix of direction cosine, that represent the rotation between frame two with respect to frame one, defined as follow:

$$[R_{2,1}] = \begin{bmatrix} cos(x_2, x_1) & cos(y_2, x_1) & cos(z_2, x_1) \\ cos(x_2, y_1) & cos(y_2, y_1) & cos(z_2, y_1) \\ cos(x_2, z_1) & cos(y_2, z_1) & cos(z_2, z_1) \end{bmatrix} \tag{4.3}$$

The rotation which involve rotation only around one of the three axes are defined as *elementary rotations*, that are characterized by the following elementary

rotation matrices:

$$
\begin{aligned}
R_x(\alpha) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) \\ 0 & sin(\alpha) & cos(\alpha) \end{bmatrix} \\
R_y(\alpha) &= \begin{bmatrix} cos(\alpha) & 0 & sin(\alpha) \\ 0 & 1 & 0 \\ -sin(\alpha) & 0 & cos(\alpha) \end{bmatrix} \\
R_z(\alpha) &= \begin{bmatrix} cos(\alpha) & -sin(\alpha) & 0 \\ sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}
\end{aligned}
\tag{4.4}
$$

The rotation matrices have some properties: can be inverted and composted. The inversion matrix of a rotation is its transposed, as reported in the following equation:

$$
\begin{aligned}
R^T \cdot R = I \\
R^{-1} \cdot R = I
\end{aligned}
\tag{4.5}
$$

so R^T is equal to R^{-1}. To describe the composition of a rotation matrices let's assume another frame rotated in the figure 4.2, the relation between P_1, P_2 and P_3 can be written as follow:

$$
\begin{aligned}
\{P\}_1 = [R_{2,1}]\{P\}_2 \qquad \{P\}_2 = [R_{3,2}]\{P\}_3 \\
so: \quad \{P\}_1 = [R_{2,1}][R_{3,2}]\{P\}_3 \\
since: \quad \{P\}_1 = [R_{3,1}]\{P\}_3 \\
then: \quad [R_{3,1}] = [R_{2,1}][R_{3,2}]
\end{aligned}
\tag{4.6}
$$

This relation is true if the rotation are made with respect to a current frame, post-multiplying the matrices. If the rotations are made with respect to a fixed frame the overall rotation is obtained by pre-multiplication of the rotation matrices in the order of the given sequence. A rotation in the space, according to the Euler's

rotation theorem, can be described by a sequence of rotation, no more than three, about coordinate axes, where no two successive rotations can be about the same axis. There exist at maximum twelve triplets of angles, each one represent a triplet of Euler angles. One of the most used triplet is the Roll-Pitch-Yaw angles, that represent the rotations around the Z, Y and X axis and they represent rotations defined with the respect to a fixed frame. Given these three angles γ, β and α, the overall rotation is given by pre-multiplication, as reported in the following equation:

$$R = R_z(\alpha)R_y(\beta)R_x(\gamma) \tag{4.7}$$

Transformation matrices

If the two frames are not only rotated but also translated one with respect to the other, there is also an offset to consider. In the following figure an example of this case is reported:

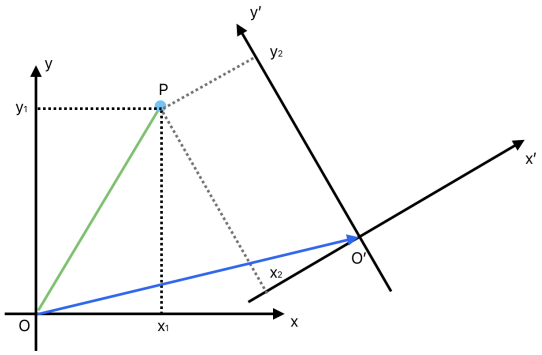


Figure 4.3: Example of a point P visualized from two rotated and translated frames

In this case the relationship between the point P visualized in the frame one, $\{P\}_1$, and the same point visualized in the frame two, $\{P\}_2$, is described in the following equation:

$$\vec{OP} = \vec{O'O} + \vec{O'P} \qquad so: \quad \{P\}_1 = \{O'\}_1 + [R_{2,1}]\{P\}_2 \tag{4.8}$$

Using an homogeneous representation of P: $\{\tilde{P}\}_1 = \begin{bmatrix} \{P\}_1 \\ 1 \end{bmatrix}$ the following relation can be obtained:

$$\{\tilde{P}\}_1 = [T_{2,1}]\{\tilde{P}\}_2$$

$$\text{where: } [T_{2,1}] = \left[\begin{array}{ccc|c} & \mathrm{R}_{2,1} & & \{O'\}_1 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{4.9}$$

The matrix $[T]$ is called *transformation* matrix. The properties of composition and inversion are still valid for this type of matrices, but with some changing. The following equation shows the composition of transformation matrices between a point P seen from 2 different frames:

$$\begin{aligned} \{\tilde{P}\}_1 = [T_{2,1}]\{\tilde{P}\}_2 \qquad & \{\tilde{P}\}_2 = [T_{3,2}]\{\tilde{P}\}_3 \\ so: \quad & \{\tilde{P}\}_1 = [T_{2,1}][T_{3,2}]\{\tilde{P}\}_3 \\ since: \quad & \{\tilde{P}\}_1 = [T_{3,1}]\{\tilde{P}\}_3 \\ then: \quad & [T_{3,1}] = [T_{2,1}][T_{3,2}] \end{aligned} \tag{4.10}$$

Some changes are adopted in the formulation of the inversion property, as reported in the following equation:

$$\begin{aligned} &\{\tilde{P}\}_1 = [T_{2,1}]\{\tilde{P}\}_2 && \{\tilde{P}\}_2 = [T_{2,1}]^{-1}\{\tilde{P}\}_1 \\ &\{P\}_1 = \{O'\}_1 + [R_{2,1}]\{P\}_2 && \{P\}_2 = -[R_{2,1}]^T\{O'\}_1 + [R_{2,1}]^T\{P\}_1 \\ &so: \quad [T_{2,1}]^{-1} = \left[\begin{array}{ccc|c} & \mathrm{R}_{2,1}^T & & -R_{2,1}^T\{O'\}_1 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \end{aligned} \tag{4.11}$$

So the inversion of a transformation matrix is not its transpose.

End-effector pose

The description of the end-effector pose is part of the kinematic problems, more specific is a problem of direct kinematic. The **direct kinematic** problem consists in describe the end-effector pose as function of the joint variables of the robot. The ingredients of the direct kinematic are two: the base frame O_0 and the end-effector frame O_e. The direct kinematic is described by the the transformation matrix $T_e^0(q)$ that depends by all the joint variables q. To find this matrix there exists two ways: find it by geometric inspection or in a systematic way. For complex kinematic chains the easier way is the systematic one. This method consist in find the description of the kinematic relationship between consecutive links and use the composition property of the transformation matrices to find the overall relation:

$$[T_e^0] = [T_{1,0}][T_{2,1}] \cdots [T_{e,n}] \tag{4.12}$$

The direct kinematic can also be written in the following formulation:

$$\mathbf{x}_e = k(\mathbf{q}) \tag{4.13}$$

where $\mathbf{x}_e$ is the end-effector pose and k is the representation of the direct kinematics, that depends on the joint variables vector $\mathbf{q}$. To chose where place frames in each link is convenient to use a convention, called Denavit-Hartenberg, not described in this work.

4.1.3 Robot tasks and control

In the use of robots there are also other tasks that are useful, such as finding the joint variables that are needed to reach a desired pose, this task is solved by the *inverse kinematics*. The **Inverse kinematics** goes from the cartesian space, in which the manipulator task is defined, to the joint space. It is typical a non-linear problem that can have no solution, a finite number of solution or infinitely many. To solve this problem there exists two types of solutions: analytical or numerical solutions. The analytical solution uses a geometric inspection of the kinematic

chain and is preferred when it can be found, instead the numerical one is slower with respect to the analytical but is easier, is needed in redundant robot and uses the *Jacobian* matrix. For the numerical solution there exists different method, like the Gradient and Newton methods. The Newton method uses the inverse of the Jacobian matrix in its formulation instead the Gradient uses the transpose, the first is faster but has problems near to the singularities condition of the Jacobian, in which the matrix lose rank and it is not invertible. The Jacobian matrix is the matrix of the first order derivatives of the direct kinematic with respect to the joint variables, reported in the following equation:

$$\mathbf{J} = \frac{\delta \mathbf{k}}{\delta \mathbf{q}} = \begin{bmatrix} \frac{\delta k_1}{\delta q_1} & \frac{\delta k_1}{\delta q_2} & \dots & \frac{\delta k_1}{\delta q_n} \\ \frac{\delta k_2}{\delta q_1} & \frac{\delta k_2}{\delta q_2} & \dots & \frac{\delta k_2}{\delta q_n} \\ \vdots & \vdots & & \vdots \\ \frac{\delta k_n}{\delta q_1} & \frac{\delta k_n}{\delta q_2} & \dots & \frac{\delta k_n}{\delta q_n} \end{bmatrix} \tag{4.14}$$

where k represent the direct kinematic in the formulation 4.13. The Gradient and Newton methods are reported in the following equation:

$$\begin{aligned} Newton: \qquad & \mathbf{q}^{k+1} = \mathbf{q}^k + \mathbf{J}_k^{-1}(\mathbf{q}^k)[\mathbf{x}_e^d - k(\mathbf{q}^k)] \\ Gradient: \qquad & \mathbf{q}^{k+1} = \mathbf{q}^k + \alpha \mathbf{J}_k^T(\mathbf{q}^k)[\mathbf{x}_e^d - k(\mathbf{q}^k)] \end{aligned} \tag{4.15}$$

where $\mathbf{x}_e^d$ is the vector of the desired pose and α is a scalar step size grater than zero. The numerical solution to the inverse kinematics problem just illustrated are used as basic control strategy to control a robot. There exists also other types of the kinematics and also the dynamics that allows to achieve a better model description of the robot to obtain a better control in terms of performances. Below are some of the possible analyzes that can be carried out on a robot:

- differential kinematics: it analyses the relation between joint velocities and cartesian velocities;
- statics: it analyses the relation between the forces applyed to the end-effector and the ones applied to the joints;
- Dynamics: it analyses the relation between the forces acting on the robot and the robot motion.

Motion control

To control the motion of the robot to execute tasks of regulation or trajectory tracking, the general control scheme is based on feedback and feedforward. The feedforward part applyes to the robot the command to follow the ideal trajectory, instead the feedback part compensate for unmodeled dynamics and the errors in the tracking of the trajectory or the error in the final pose, taking information from the robot sensors. The following figure represent the generic control scheme just explained:

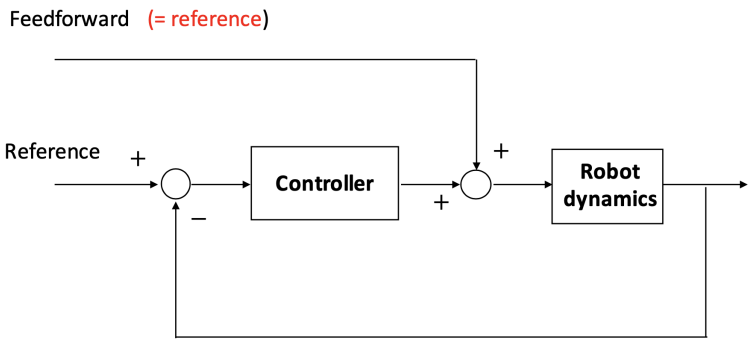


Figure 4.4: Example of generic robot control scheme [11]

The current feedback control in industries is a low level one, because it gives as control input some kinematic commands, in general velocities, forgetting the dynamics of the system. To achieve better performances, in terms of end-effector velocity and acceleration, some improving in the model or in the control are needed, for example including the dynamics of the robot model or changing the controller in an adaptive one. The controller can retrieve information, as already introduced at the beginning of this chapter, using one or more cameras. A controller based on feedback on visual measurement is called visual servoing. There exists two types of visual servoing: position-based visual servoing (PBVS) or image-based visual servoing (IBVS); PBVS uses the reconstructed 3D pose from the images to drive the robot, instead IBVS work on the 2D image plane to elaborate the motion to gives to the robot. Since the conveyor tracking problem uses the cameras but it does not need a feedback during the motion based on the informations that came

from the cameras all these control techniques are not explained in the details. From the user point of view, if the robot is already built and its controller is already developed, the interest is to give commands to the robot to be able to complete custom tasks, like the pick and the place of objects in the workspace. This is done using a GUI developed by the company producing the specific robot. For example for the Omron's robots there exist a free software, *Automation Control Environment (ACE)*, within which it is possible to set and code tasks to be executed by Omron's robots, the code is developed using its specific language called *Adept V+*. This environment is used for the development of the conveyor tracking task addressed in the next section.

4.2 Conveyor tracking

The conveyor tracking is an important task in industries, it is a task whose objective is to detect and localize one or more objects in the the conveyor belt so that these objects are subsequently taken by a robot and placed in a desired position. I personally solved this task, in the DII University laboratory, detecting the objects developing two different Python programs: the first program is made using only computer vision techniques instead the second one using artificial intelligence.

4.2.1 Experimental setup

The **setup** is composed by a conveyor belt, an Omron delta robot with four kinematic chains, two cameras, one RGB and one grayscale, an industrial controller, that controls the robot, and finally a PC, necessary to code in ACE an Adept V+ code, sent to the industrial controller. The conveyor belt is approximately 4 meters long, its velocity is controlled by a potentiometer regulated by the user and its position is measured by an encoder. The delta robot is located exactly above of the conveyor belt and towards its end. After a first part of the conveyor belt dedicated to the positioning of objects, the two cameras are present, which

take pictures of the conveyor belt from a bird's-eye view from above. The PC is used as GUI, it is necessary to code in ACE the V+ program, and also can extract the information of all the sensors and signals inside the robotic system. In the following figure the experimental setup of the laboratory is reported:

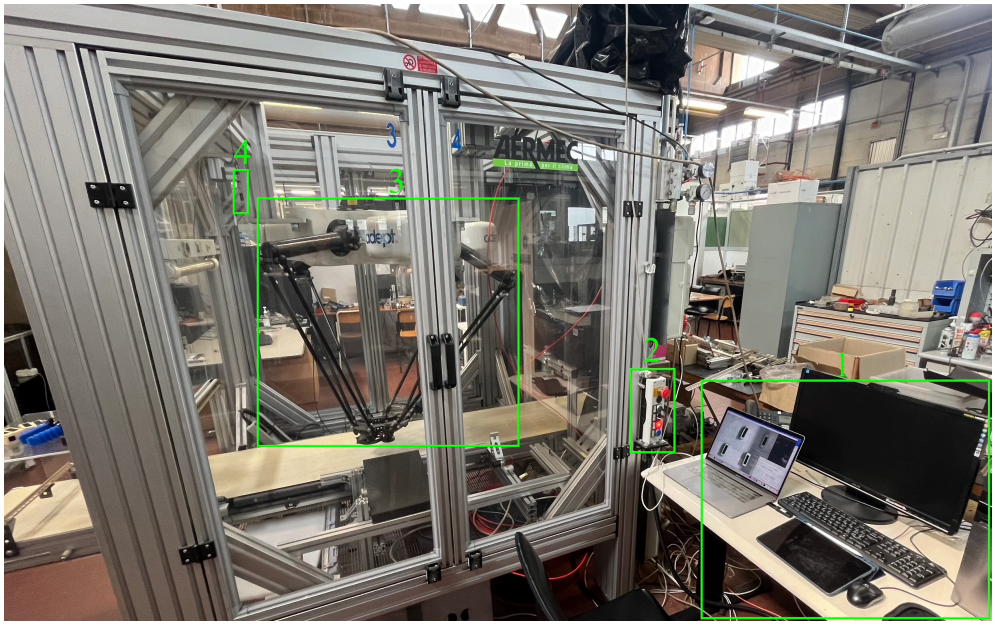


Figure 4.5: Experimental setup of the conveyor tracking

In the figure is possible to see the PC station, labeled with number one, the controller of the conveyor belt, number two, the Omron delta robot, number three, and the camera, number four, the robot controller is not visible from this prospect because is located behind the robot outside the grey structure that contains the robot. This structure is necessary by law because if the user opens the door of the cabin the power is turned off to prevent accidents due to the impact with the robot. This cabin is not needed for the collaborative robots, that are special robot able to sensing the environment and stops if a collision is detected. In ACE all the parameters are under control, the user can control the position of the robot, the signal from the encoder and also the trigger of the camera, that tells when the picture is captured, so that is possible to retrieve the distance travelled by the frame captured by the camera of the conveyor belt during time.

4.2.2 Experimental approach

For the setup already explained, my personal **approach** to the task is the following one: during the movement of the conveyor belt the camera captures frames that are sent to a Python server, that detects objects in the image captured, analyzing if there are objects in the image and where are located, the server communicates this information with the client, that is the controller and, using the trigger information from the camera, calculates the distance travelled by the objects on the conveyor belt, and finally controls the robot in such a way to pick each object detected when it enters in the workspace of the robot, placing each of them in a different place, based on the shape and the color of each of them. In the following figure the flowchart of the experimental approach is reported.

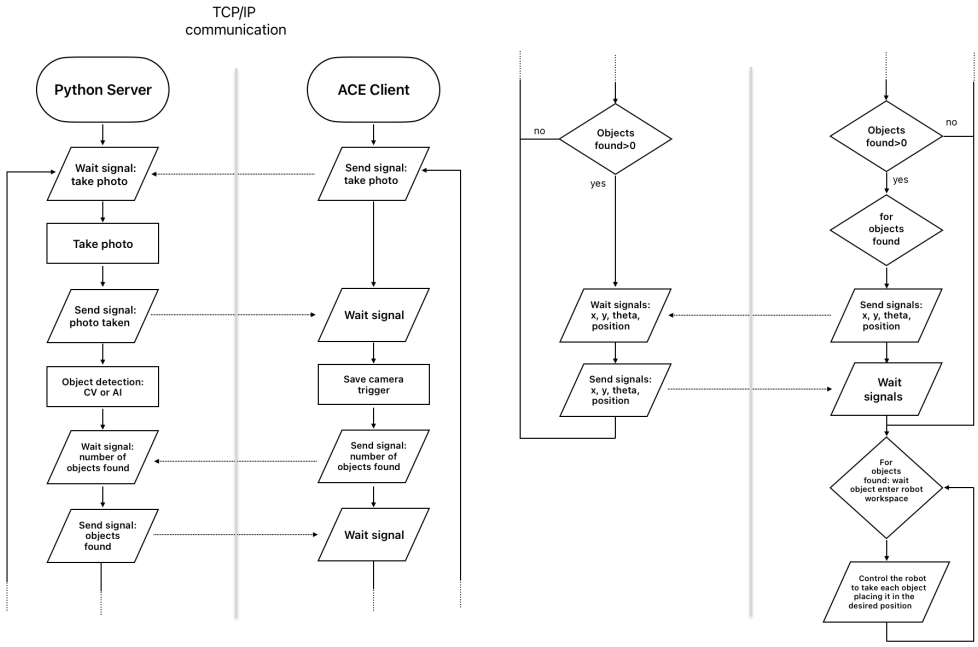


Figure 4.6: Flowchart of the experimental approach divided into two columns

Camera calibration

The specification of the industrial cameras that are used are considered unknown, but already calibrated, so the fist step is to find the focal length of the camera, because is needed to obtain the projection from 3D coordinate and pixels coordinate, as described in the equation 1.11. Since the camera can retrieve only information in 2D coordinate and it's fixed and perpendicular to the conveyor belt, the formulation can be rewritten as reported in the following equation:

$$from: \quad u = u_0 + f_u \frac{X_p}{Z_p} \qquad to: \quad u = u_0 + f_{uZ_p} X_p \tag{4.16}$$

where the parameter f_{uZ_p} incorporate not only the focal length in pixels but also the term Z_p, unknown. This equation is valid in both the axes, x and y. To find this parameter is sufficient to take a photo of a known object positioned on the conveyor, so at a distance Z_p from the camera, and use the reverse formula below:

$$f_{uZ_p} = \frac{\Delta u}{\Delta X} \tag{4.17}$$

where Δu is the size of the object in the image plane, so in pixel coordinates, and ΔX is the size of the object in the real world, in meters. In the following figure a sample used for the camera calibration is reported:

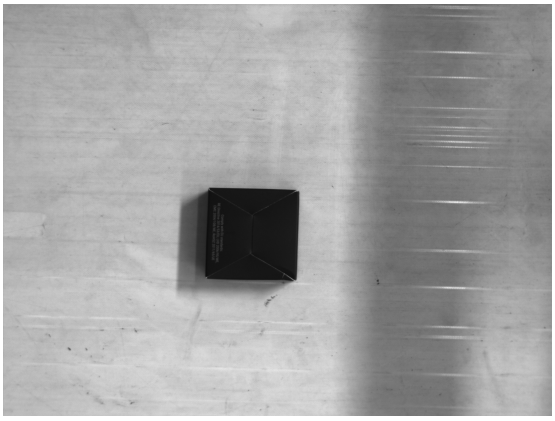


Figure 4.7: Sample for the f_{uZ_p} calculation

The object in the real world has a size of 0.064 meters, in the image plane has a size of 427 pixels, the f_{uZ_p} obtained is 6677. The camera has a resolution of

2592x1944, so values of u_0 and v_0 used in the equation 4.16 are 972 and 1296. So the inverse of the equation 4.17 can be used to retrieve the distance of an object from the center of the image plane situated on the conveyor belt, given the f_{uZ_p} just found and the position of the object in the image plane in pixels coordinates.

Transformation matrix camera-robot

Once the object in the 3D world with respect to the camera is found, the transformation matrix between the camera and the robot is needed, because it describes the relative pose, position and orientation, between these two so that the 3D point can be reported to the robot frame. To better understand in the following figure a visual description of the problem is reported:

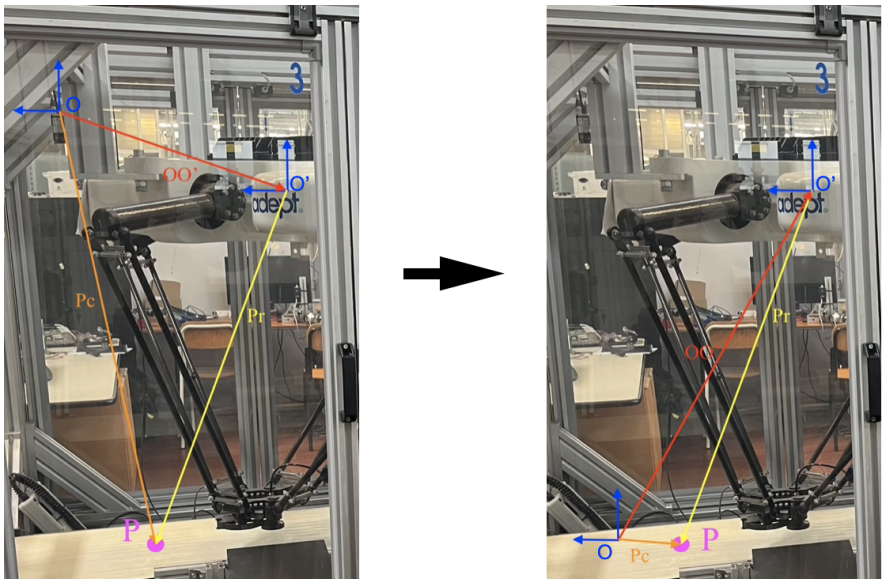


Figure 4.8: Graphics visualization of the ideal frames and distances on the left figure and the assumed true ones on the right

So the equation of the transformation matrix, explained in the equation 4.8, in my case becomes the following relation:

$$\{P\}_c = \{OO'\}_c + [R_{r,c}]\{P\}_r \tag{4.18}$$

where $\{P\}_c$ is the 3D point reported in the camera reference frame, $\{OO'\}_c$ is the vector from the camera frame to the robot frame seen from the camera frame,

$[R_{r,c}]$ is the rotation matrix between the camera and the robot and finally $\{P\}_r$ is the 3D point reported in the robot frame. Since the overall transformation matrix, which description is described in the equation 4.9, has rank four, is possible to retrieve the matrix given at least four couples of 3D point, each defined both in the robot frame and in the camera frame, not aligned and not all in the same plane perpendicular to the camera. The relation that map the points of the robot to the camera frames using the transformation matrix is reported in the following equation:

$$\begin{bmatrix} x_r \\ y_r \\ z_r \\ 1 \end{bmatrix} = [T_{r,c}] \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} \tag{4.19}$$

$$so: \quad \mathbf{y} = T\mathbf{x} \qquad that \;\; is \;\; equal \;\; to: \quad \mathbf{y}^T = \mathbf{x}^T T^T$$

The transpose version is useful only for coding reasons because it is more convenient store each point in a row vector instead of a column one, as in the case of a list of points. So the inverse of the relation just explained is intuitive:

$$T^T = (\mathbf{x}^T)^{-1}\mathbf{y}^T \tag{4.20}$$

But since the points found by the camera are all in a plane, all with the same height, the matrix $\mathbf{x}^T$ is not invertible, so to overcome this issue the use of the pseudoinverse is needed, and the resulting matrix T is not exactly a transformation matrix because it has the third rows empty, since no information on the height of the objects can be retrieved by the camera. Also since the measures in both the robot frame and in the camera frame are noisy, the formulation of the solution can be revisited using the solution of a least square problem:

$$T^T = [\mathbf{x}\mathbf{x}^T]\mathbf{x}\mathbf{y}^T \tag{4.21}$$

This solution can give better result only using more then four couple of points. The following figure shows the experimental approach used in the laboratory: on the

left there is the robot moving on the objects and on the right the corresponding image captured from the camera.

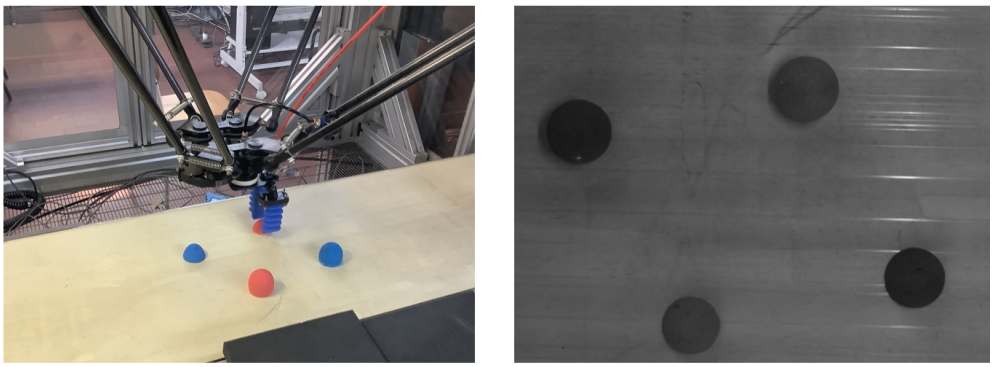


Figure 4.9: Experimental approach used in the laboratory to find the transformation matrix between the robot and the camera frames

Since the camera is not able to see the robot workspace, the measure of the distance between the object seen in the robot workspace and the object seen in the image frame must be taken from the encoder of the conveyor belt. The simple code is reported in the Appendix B, in which is possible to see that the offset, which depends on each calibration, is subtracted in the measures taken in the robot frames. The resulting transformation matrix map the points seen from the camera frame into points in the robot frame, considering the translation and rotation of the two different frames, without a manual user analysis of the physical setup.

ACE client: software and V+ code

Automation Control Environment (ACE) is an OMRON software platform which provides an effective way to implement industrial automation applications, it allows users to easily configure robots and vision systems. I used the code of another Control Systems colleague, Ettore Santoiemma, who already defined a way to code in Adept V+ language the delta robot. Since his code makes the use of an integrated system in ACE that finds blobs in the images taken from the camera, I applied some changes in such a way that it functions with my Python server,

explained in the next section. The code is made by five modules, two main modules and three secondary ones; the two main modules use the secondary ones inside their code, so only these two must be inserted in the tasks of the controller during operation. The first main module is called "robot_main", through which the robot is controlled, it regulates all the location in the workspace, the movement of the robot, if a new object is inserted in the list of objects to be picked and remove them from the list as soon as they are placed in the final desired locations. The second main module is called "communication_main", it creates the communication as client to the server, managing the requests and the answers, like when to take an image and if any object is detected. In case of a new object detected it adds the information relative to the position, orientation and final desired location in the list. The movement of the robot, the addition and removal of the information in the list are managed by the three secondary modules. The flowchart of the overall operation of the controller is reported in the figure 4.6 already inserted at the beginning of the experimental approach. The complete code is reported for completeness in the Appendix C.

Python server: two approaches to object detection

I decided to developed the code in Python language because it makes the access to the world of artificial intelligence easier. I state that both the two approaches and the codes that i developed can be implemented also in other codes languages, for example C++ language, exporting the convolutional neural networks from Pytorch to ONNX or TorchScript formats, and translating the remaining language from Python to C++. Now let's go into the details of my two approaches: the first one makes the use of only computer vision techniques, explained in the first chapter of this thesis, the second one instead uses artificial intelligence, more precisely a YOLO CNN, to detect and localize objects in the image. The structure of both the two codes is the same and it is reported in the following flowchart, which is a more detailed version of the one reported in the figure 4.6:

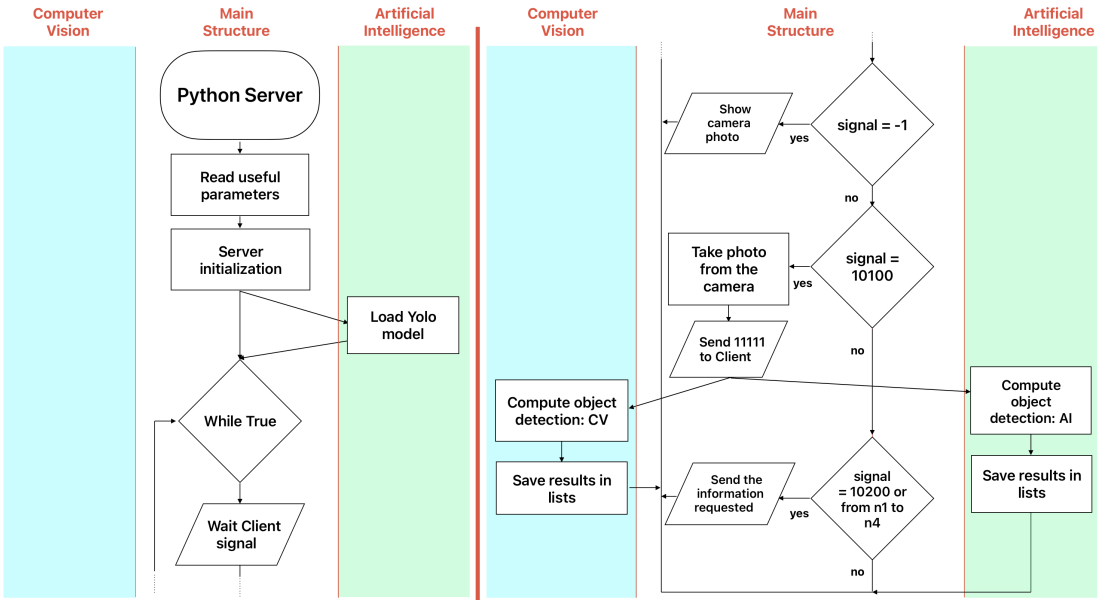


Figure 4.10: Flowchart of the main structure of the Python server codes

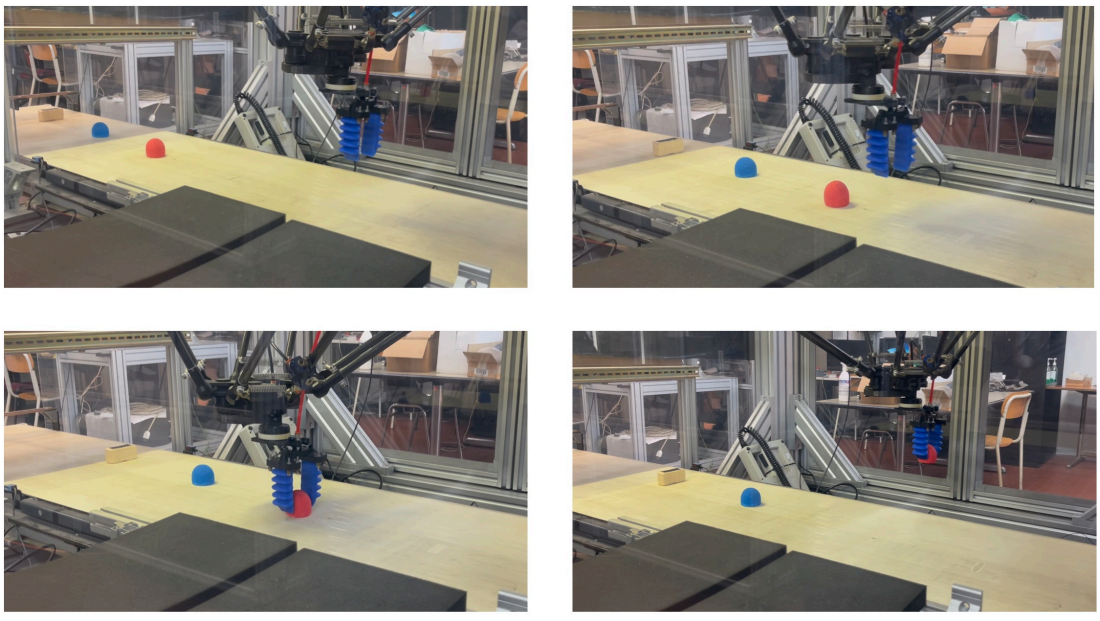
Now that the main structure is defined let's go into the details of the differences of the two object detection analyses. There are few but important differences between the two programs: the computer vision one makes the detection and localization of the objects through a written by hand dedicated function, that depends on the problem that I am trying to solve, instead the artificial intelligence loads, at the beginning, a neural network model and detect the object through model inference of the input image. The **computer vision** object detection function takes as input a resized version of the image captured by the camera, of shape 640x480 pixels, this process reduces the details and keeps the bigger blobs. A threshold is made to divide the sure objects to the sure background, which value is problem depending, and a series of morphological operation is made, first close then open, to isolate the shaded areas from the objects. Contours are found to isolate each objects, comparing the areas false detection are excluded, for example small blob of the background or the shadow of the true object. The center of the contour is assumed as the center of the detected object and is saved in a list. If lines are detected in the contour of the object it is assigned to be placed in another final desired position and the orientation of the object is measured. Using the RGB

camera another control can be made on the color of the object: for example blue and red objects with the same shape can be placed in different desired positions. I choose not to use mid-high level techniques, like segmentation or blob detection, because with this particular setup the objects are easily recognisable also with a combination of low-mid level techniques. For completeness I report that as first test I tryed to detect the object making the subtraction between the image from the camera and a saved background, in the resulting difference image the objects were very good detected, but this method function only if the conveyor belt is completely uniform in the color and also if there are not illumination changes. Instead for the **artificial intelligence** program a trained YOLO neural network is loaded, that is used to detect and localized the objects in the images captured from the camera. So the image is given as input in the model and the output is composed by the location of the bounding box around the detection and the class of the object detected, in my case I trained the CNN to detect blue and red balls so the output classes are in $\{0, 1\}$. To detect other type of objects, like object with rectangular shape, a new training set and a new training phase are needed before the detection. The complete Python code used as server and the one used for training are reported in the Appendix D. For both the codes the region of interest for the detection is the centered 4/5 of the image, because objects detected in the edges may be bad localized so can be captured again in the next frame. The time interval between frames depends by the controller, because it communicates to the server to take a picture only if the conveyor belt has moved by a certain amount of space, defined in the code for this experiment as 20cm, since the camera can see a window of around 30cm in the direction of movement of the conveyor. The maximum velocity of the conveyor belt is 31,5cm/s, the codes can execute the detection in less that 250ms, so these codes can be used also at the maximum velocity of the conveyor belt.

4.2.3 Results and future work

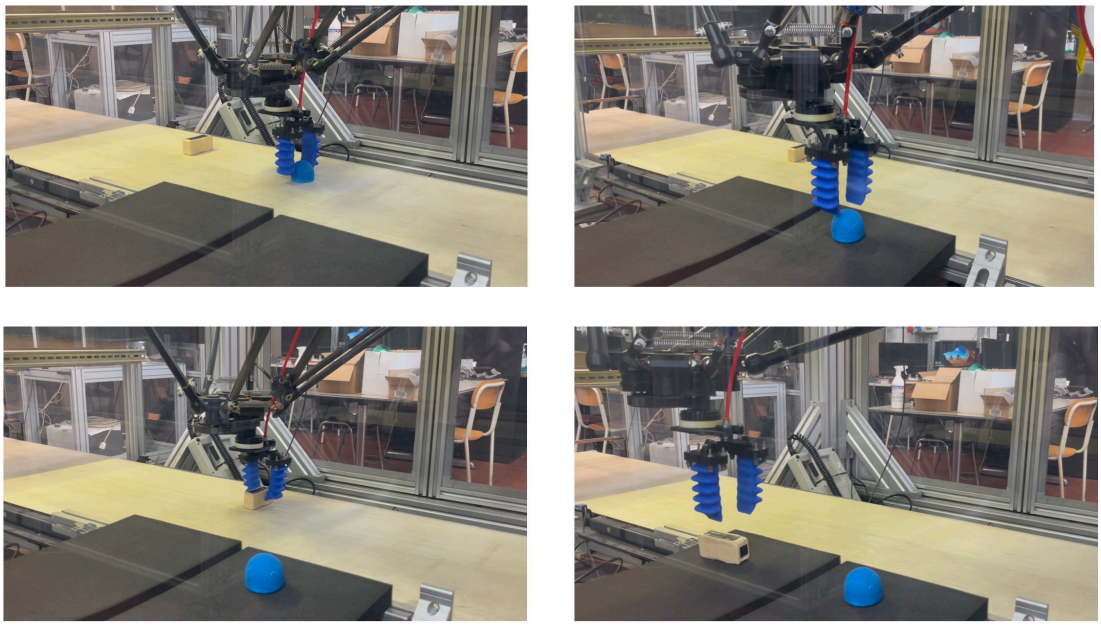
Results

The goal of object detection for conveyor tracking is reached in different ways, the computer vision approach is designed ad-hoc for the task and may be modified to be adapted for different tasks and setups, with complex background only high level computer vision approaches can complete the task. Instead the artificial intelligence approach is more generic and can be used with any background or setup without retraining the model, also the benefit is that it can be used without a previous study of the final setup and it function very well also using a generic training dataset. The following figures show the highlights of the task in which the robot must pick and place three different objects in the corresponding defined places:



A	B
C	D

Figure 4.11: A) object seen from the camera; B) robot approaching first object; C) robot picking the object; D) robot placing in the desired final position for red objects



A	B
C	D

Figure 4.12: A) robot picking the second object; B) robot placing it in the desired final position for blue objects; C) robot picking the object with rectangular shape rotating the gripper; D) robot placing it in the desired final position for objects with rectangular shape

Difficulties

The difficulties that I encountered are related to the specific cameras used in the experimental setup. The trigger sent from the camera is captured by the controller, so the synchronism between the frame captured by the Python server and the signal received by the controller is absent. A way to find this synchronism is to set the fps, frames per second, of the camera with a low value. In my personal experiments I used a fps value lower than 10, so that the frame captured by the Python server and the trigger captured from the controller, client of the communication, after the server communicated to it that a picture was saved, are synchronised. For example, since Python wait for a new frames before capture

the picture, using an fps value of 10 the time available by the communication of the information that tells that the picture has been saved is 100ms, and since the communication requires less that 10 times 100ms, the syncronism can be reached. This low value of fps is used also because I noticed that the controller was not able to update the trigger value of the camera for higher value of fps, so this led to the failure of the task. For the RGB camera the fps is reduced to 4 and also the resolution is decreased from 2590x1940 to 2000x1940 otherwise the Python server received a completely black image, probably due to an issue related to the physical USB communication.

Future works

As future work I suggest to solve the issue related to the syncronism between the camera and the PC using a different camera or trying to control the camera in a software way, maybe using the pylon libreries that are designed to control the industrial cameras present in the DII laboratory. Also the YOLO network describe the detection in a parallel way to the image, so it does not rotate according to the object shape, so once the object is detected the optional rotational component must be found using some computer vision techniques on the contour of the area of the image containing the object. To solve this issue using only artificial intelligence there exist a new version of the YOLO neural network architecture, that is called YOLO OBB, Oriented Bounding Boxes, which output the oriented bounding boxes, following the shape of the object detected.

Chapter 5

Stereovision

In the first chapter the projection of a 3D point into the 2D image plane was already illustrated in the equation 1.14. Since a single camera can not reconstruct the depth of an object seen, two or more cameras can be used to reconstruct, using *multiple view geometry*, this third dimension, combining the information coming from each camera. This approach can be used in industrial environment since the images brings a lot more information with respect to the others measure systems like a lidar. The combination of this technique with artificial intelligence can opens new frontiers in the field of industrial automation and can brings the processes at another performance levels. Now let's go into the details of the theory behind the stereovision before enter into the experimental setup.

5.1 Multiview geometry

Let's consider two cameras that capture the same 3D point in space, this set-up is typical of the stereo cameras, and is visualized in the following figure:

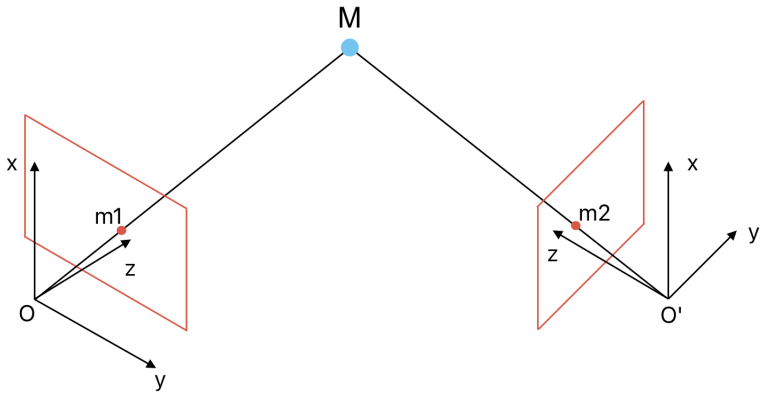


Figure 5.1: General stereo camera set-up

Since the equation 1.14 reported in the first chapter is true up to a scale factor, is possible to write that equation for the points m_1 and m_2 in the following way:

$$\begin{aligned} \lambda_1 \tilde{m}_1 &= P_1 \tilde{M}_1 \qquad \text{where:} \quad \tilde{M}_1 = T_1 \tilde{M} \\ \lambda_2 \tilde{m}_2 &= P_2 \tilde{M}_2 \qquad \text{where:} \quad \tilde{M}_2 = T_2 \tilde{M} \end{aligned} \tag{5.1}$$

The relation between M_1 and M_2 can be found exploiting the following equations:

$$\begin{aligned} M_1 &= T_1 M \\ M_2 &= T_2 M \\ \text{Combining the two we obtain:} \quad M_2 &= T_2 T_1^{-1} M_1 \\ \text{So:} \quad M_2 &= T_{2,1} M_1 \end{aligned} \tag{5.2}$$

Given m_1 and m_2 as projections of the same M, find $T_{2,1}$, so $R_{2,1}$ and the vector OO', is called *pose reconstruction problem*. Looking for a pure geometric relation, we can assume the matrix P as the identity matrix, considering a normalized camera with focal length equal to one and no other intrinsic parameters. The equations 5.1 can be rewritten in the following way:

$$\begin{aligned} \lambda_1 \tilde{m}_1 &= \tilde{M}_1 \\ \lambda_2 \tilde{m}_2 &= \tilde{M}_2 \end{aligned} \tag{5.3}$$

We can rewrite the equation 5.2 in the following way:

$$M_2 = R_{2,1} M_1 + O_{2,1} \tag{5.4}$$

where $O_{2,1}$ is the is the column vector representing the distance OO'. Removing the use of homogeneous coordinates the following system of equation can be found, using the equations 5.3 and 5.4:

$$\begin{cases} \lambda_1 m_1 = M_1 \\ \lambda_2 m_2 = M_2 \\ M_2 = R_{2,1} M_1 + O_{2,1} \end{cases} \tag{5.5}$$

Solving the system the following relation is obtained:

$$< m_2, [O_{2,1}]_x R_{2,1} m_1 >= 0 \tag{5.6}$$

that represent the scalar product between the vector m_2 and the vector resulting by the cross product between $O_{2,1}$ and $R_{2,1}$, computed as the matrix multiplication between the skew-symmetric matrix $[O_{2,1}]_x$ and the vector $R_{2,1}$. The overall product is zero if the volume of the parallelepiped having the three vectors as sides is zero, this is true only if the three vectors are coplanars. The equation 5.6 is called *Longuet Higgins equation* and represent a constraint, called *Epipolar contraint*, represented in the following figure:

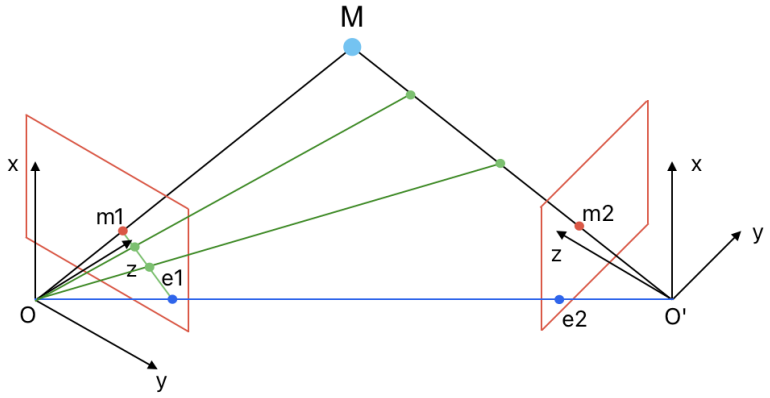


Figure 5.2: Representation of the Epipolar contraint

where the line OO' is called baseline and the points e_1 and e_2 are called epipoles. This contraint implies that all the points in the *epipolar line*, the line between m_1 and e_1, are projected in the point m_2 in the second image plane.

The matrix $[O_{2,1}]_x R_{2,1}$ is called **Essential** matrix, E.

There exist also another important matrix, called **Fundamental** matrix, defined in the following equation:

$$F = (K_2^{-1})^T E K_1^{-1} \tag{5.7}$$

that is used as the essential one in a calibrated vision for the epipolar contraint: $\tilde{m}_2^T F \tilde{m}_1$. The matrix E is a 3x3 matrix but characterized by the following SVD decomposition:

$$\begin{aligned} E &= U\Sigma V^T \\ \text{where:} \quad \Sigma &= \begin{bmatrix} \sigma & 0 & 0 \\ 0 & \sigma & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{5.8}$$

So the matrix E can be found through minimization, there exist different type of algorithm that solve this problem, for example the 8-points algorithm, which output a matrix E already projected in the *Essential space*, because given a noisy approximation of E, projecting it onto the Essential space is needed, this is made making the average of σ_1 and σ_2 to find the σ value and placing σ_3 to zero in the SVD decomposition. Now that the matrix E is found by the algorithm, it is possible to retrieve the matrix $R_{2,1}$ and the vector $O_{2,1}$ from E, as reported in the following equation:

$$\begin{aligned} [O_{2,1}]_x &= U R_z\left(\pm\frac{\pi}{2}\right)\Sigma U^T \\ R_{2,1} &= U R_z^T\left(\pm\frac{\pi}{2}\right) V^T \end{aligned} \tag{5.9}$$

So there exists four possible pairs of $(O_{2,1}, R_{2,1})$ such that $E = O_{2,1}, R_{2,1}$. The choices valid for the pinhole camera model are $R_z\left(+\frac{\pi}{2}\right)$, for both the equations in the formula 5.9.[14]

5.2 Pose reconstruction

Now that all the notions about the stereovision and the transformation matrix between two cameras are explained, let's go into the details of the pose recon-

struction problem. At first let's consider the simpler case, in which the cameras are parallels, then exploiting the general case adding a rotation between them.

5.2.1 Parallel cameras

Let's start thinking at the human eyes, looking straight ahead the depth perception comes from the "disparity" of the projections of a given 3D point in your right and left retinal images. This concept can be used to reconstruct the pose in the case where the two cameras are parallels, not rotated but only shifted in the x axis, so in horizontal way, as represented in the following figure:

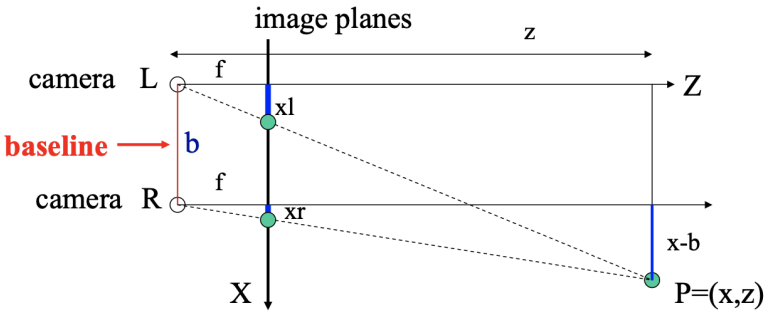


Figure 5.3: Stereovision representation with parallel cameras

where b, baseline, is the distance between the two cameras, f the focal length of the two cameras, P the 3D point in space, x_r and x_l the distance of the point projected in the image planes of the right and left cameras. Recalling the equation 1.4 in the first chapter, the following relations can be exploited and put in a system:

$$\begin{cases} Z = f\frac{X}{x_l} \\ Z = f\frac{X-b}{x_r} \end{cases} \tag{5.10}$$

where the coordinates of the 3D point are X, Y ans Z. So the resulting Z, depth of the point, is reported in the following equation:

$$\begin{aligned} Z &= f\frac{b}{x_l - x_r} = f\frac{b}{d} \\ X &= x_l\frac{Z}{f} = x_r\frac{Z}{f} + b \\ Y &= y_l\frac{Z}{f} = y_r\frac{Z}{f} \end{aligned} \tag{5.11}$$

where d is the disparity and the X and Y coordinates derives directly from the equation 5.10. So the depth of a 3D point is inversely proportional to disparity d. To solve this problem we need a priory knowledge of f and b, and the current knowledge of the correspondences of the same 3D point into the two images, this can be done using AI, like YOLO, to make the detection or using some other CNN architecture that are trained to find the disparity map of the two images, like ModuleNet.

5.2.2 Not parallel cameras

For not parallel cameras the reconstruction of the $O_{2,1}$ and $R_{2,1}$ matrices reported in the equation 5.9 must be done. To reconstruct the 3D point from the 2D image coordinates and the matrices $O_{2,1}$ and $R_{2,1}$ the *Direct Linear Transform*, DLT, can be used.

Direct Linear Transform

DLT is a method for calculating a matrix equation of the form $Ax = 0$, where A is a matrix and x is an unknown vector that we want to find. Let's now consider the stereovision problem, we already said in the equation 1.14 that $\tilde{m} = \lambda P\tilde{M}$, where the matrix P and the vector $\tilde{m}$ are known. Since $\tilde{m}$ and P $\tilde{M}$ are parallel vectors the cross product of these must be zero:

$$\begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix} \times \begin{bmatrix} \vec{p_1}\tilde{M} \\ \vec{p_2}\tilde{M} \\ \vec{p_3}\tilde{M} \end{bmatrix} = \begin{bmatrix} v_1\vec{p_3}\tilde{M} - \vec{p_2}\tilde{M} \\ \vec{p_1}\tilde{M} - u_1\vec{p_3}\tilde{M} \\ u_1\vec{p_2}\tilde{M} - v_1\vec{p_1}\tilde{M} \end{bmatrix} = \begin{bmatrix} v_1\vec{p_3} - \vec{p_2} \\ \vec{p_1} - u_1\vec{p_3} \\ u_1\vec{p_2} - v_1\vec{p_1} \end{bmatrix} \tilde{M} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.12}$$

where $\vec{p_i}$ are the row vector of the matrix P, four dimensional, and u_1 and v_1 are the pixel coordinates of the point in the image plane. The reported equation is in the form $Ax = 0$ but its third row is a linear combination of the other two, as expected using only a single camera. Since we are using two cameras we can add in the equation the points of the second view, as reported in the following equation:

$$A\tilde{M} = \begin{bmatrix} v_1\vec{p_3} - \vec{p_2} \\ \vec{p_1} - u_1\vec{p_3} \\ v_2\vec{p_3} - \vec{p_2} \\ \vec{p_1} - u_2\vec{p_3} \end{bmatrix} = 0 \tag{5.13}$$

The solution of the previous equation is a vector $\tilde{M}$ that is into the null space of A, this space is spanned by the right singular vectors corresponding to zero singular values of the matrix A. Since in the real world the measurements are noisy, the problem $A\tilde{M} = 0$ can be structured in the following way

$$A\tilde{M} = \vec{w} \tag{5.14}$$

finding the $\tilde{M}$ that minimized the $\vec{w}$ quantity. Now let's go into a step by step solution, first of all let's make the SVD decomposition of the matrix A, as reported in the following equation:

$$A\tilde{M} = USV^T\tilde{M} \tag{5.15}$$

Now let's take the dot product of $\vec{w}$ to find a scalar quantity, as reported in the following equation:

$$\vec{w}^T\vec{w} = \left(\tilde{M}^T VSU^T\right) \cdot \left(USV^T\tilde{M}\right) = \tilde{M}^T VS^2V^T\tilde{M} \tag{5.16}$$

whew U and V are orthonormal matrices and S is a diagonal matrix, which entries are decreasing and represent the singular values of the matrix A. Since V is orthonormal, if we select $\tilde{M}$ to be one of the column vectors of V^T, $\vec{v_i}$, we obtain the following relation:

$$\vec{v_i}^T VS^2V^T\vec{v_i} = s_i^2 \tag{5.17}$$

where s_i is the i'th entry of the diagonal of the matrix S. Since the goal is to minimize $w^T w$, this is equivalent to choose the last diagonal entry of the matrix S^2, selecting as $\tilde{M}$ the last column vectors of V^T.

5.3 Approach and result

5.3.1 Hardware

I tried to solve this task using two identical phone cameras but I found several problems: first of all the setup must be well fixed, as a minimal movement is enough to make changes in the translational and rotational components of the transformation matrix between the two cameras, and also I discover that the cameras that I tried to use are wide cameras, so the projection theory explained in the first chapter are not applicable. These camera may be used only if the map between the 3D points and the corresponding 2D mapped points in the image plane is found. So I build my own stereo camera system using 2 identical cameras, with resolution of 1280x720 pixels, a declared FOV of 50° and manual adjustable focus. I placed these cameras at a distance of about 60cm on a wooden beam, each with a rotation of 10 degree rotation towards the center of the beam. The final setup is reported in the following figure:

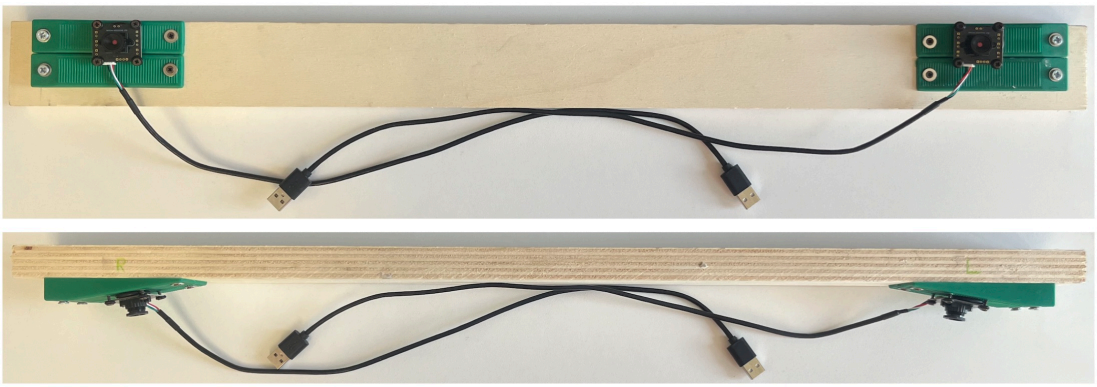


Figure 5.4: Stereovision setup

The choices of the distance between the two cameras and their rotations with

respect to the beam influence the parameters of the transformation matrix between the two cameras. With this setup is possible to see in both cameras the same object at a distance greater than 60cm. Furthermore even if each camera has a sensor resolution of only 1MP, the spatial resolution one meter away from an object is less that 1mm. One important note regards the overall precision of the system, this depends not only from the cameras spatial resolution but also from the camera calibration results, the precision of the transformation matrix given by the stereo camera calibration and finally from the precision of the localization of the object or feature in both the images of the two cameras. So to retrieve a better estimate of the distance of some object of interest from the stereo camera system is advisable to choose two camera with high sensor resolution, build a robust system, so that the transformation matrix between the two cameras remains the same in time, execute a good single camera calibration and stereo calibration of the stereo system, for example using a chessboard of which the squares are of known size, and finally develop a software able to detect feature or object in a precise and reliable way. Also the relative pose of the two camera helps in the precision factor, for example in the following figure three stereo camera scenarios are reported: in the first the two cameras are rotated one with respect the other of an angle greater of 90°, in the middle figure the relative angle is of 90°, in the third one instead they are parallels.

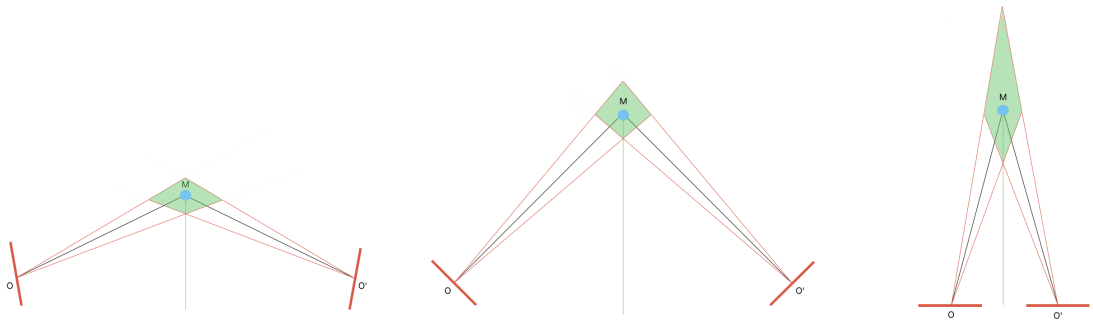


Figure 5.5: Three scenarios of relative poses of two cameras in a stereovision application

So is possible to see that, given the same angle error in the detection, which

boundaries are described as the red lines, the corresponding area of the mismatch is wider in the direction of the epipolar line in the first case, in the second is more or less uniform, instead in the last case is much greater in the perpendicular axis with respect to the epipolar line. Given all these information to reach the goal with a very high precision is advisable to use more than two cameras, in case of only two cameras is better to rotate one with respect to the other in such a way that the area of interest in which the object is probable to be detected is around the intersection of the two optical axis of the two cameras, like the central scenario of the figure 5.5.

5.3.2 Software implementation

I took inspiration from the project of Temuge Batpurev, PhD and researcher scientist at Kyushu Institute of Technology, Japan, who analysed the stereovision problem in the case of not parallel cameras.[15]
The structure of the entire code is divided into two parts: the pre-processing and the execution. In the first process the single camera and the stereo camera calibration are made, these produces the camera matrices and the distortion coefficients of both cameras, retrieving also the transformation matrix between the right and left cameras, setting the origin of the system frame on the right one. In the following figure two photos captured by the stereocamera during the calibration process are reported:

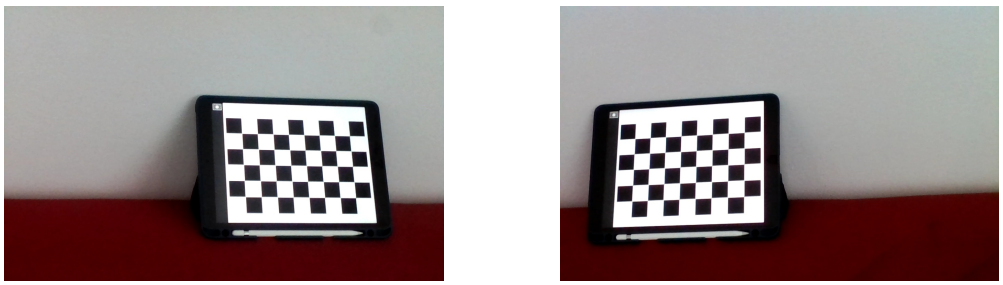


Figure 5.6: Stereocamera photos of the left and right cameras during the calibration phase

The second process is the real execution of the task, in which using the data founded in the pre-processing, the triangulation is made to retrieve the depth of the desired object. To found the desired object I used two possible choices: find the object by hand in a post-processing way, for example using the click of the mouse in the captured pictures, or in real time, using an automatic detection algorithm. This last option can be used in real time implementing the techniques already explained, for example using a YOLO neural network. There exist another option: let a CNN lean the disparity map for parallels cameras, as already mentioned previously, or let it learn all the passages already explained, from the stereo calibration to the triangulation, but is more complex because a train dataset must be crated, composed by the pairs of pictures from the two cameras and the 3D reconstructed environment. To completness the overall structure used is reported in the following figure: on the left the pre-processing and on the right the two possible approaches to the points triangulation.

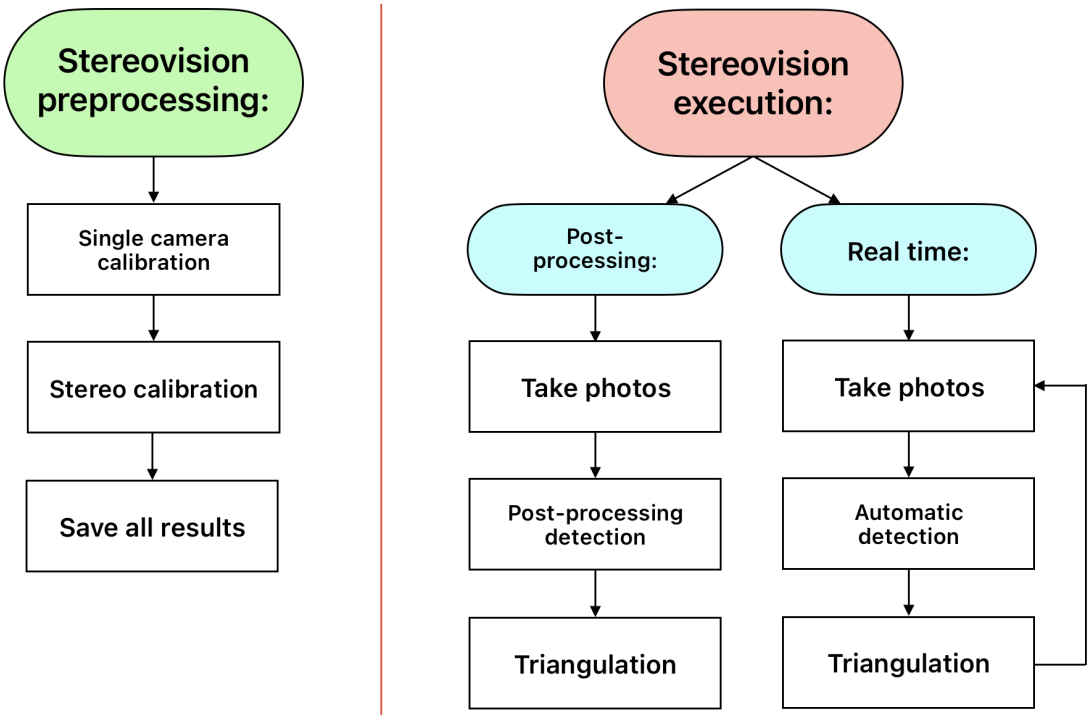


Figure 5.7: Structure of the Stereovision code: on the left is reported the pre-processing part and on the right the two possible approaches used

The triangulation is made using the DLT method explained in the section 5.2.2. The complete code used is reported in the appendix E

5.3.3 Results

Difficulties

The main difficulties I encountered developing this task were related to the cameras, because these must have normal and not wide lenses, and their order, namely which camera was the right one and which one was the left one. This is a very important aspect because swap the two cameras in the software leads to a failure of the triangulation of the points, so at each connection of the camera to the pc the test to verify the correct camera order is needed.

Results

I combined the described stereovision cameras with the YOLO network, the resulting system is able to detect any blue or red ball from a distance of at least 60 cm from the cameras. To reduce the distance of detection the length between the two cameras must be reduced or the rotation of one camera with respect to the other must be increased. This system can be used in an industrial environment in all the application where the pose of the objects of interest is described in the space and not on a specific plane. In the following figures the example of pictures from the stereocamera are reported:

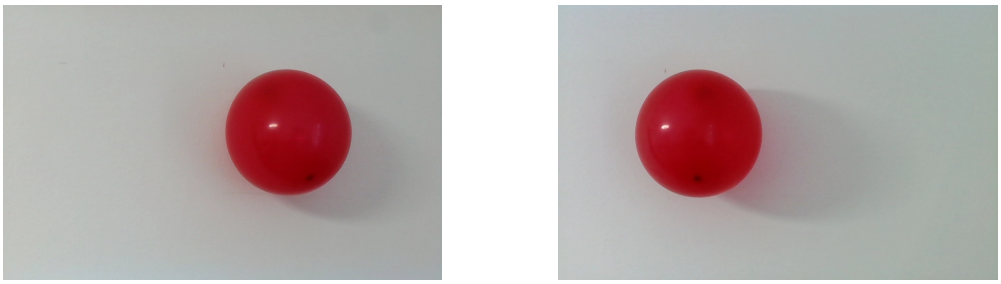


Figure 5.8: Stereocamera photos of the left and right cameras capturing a balloon at a distance of 95cm

Figure 5.9: Stereocamera photos of the left and right cameras capturing a balloon at a distance of 171cm

Figure 5.10: Stereocamera photos of the left and right cameras capturing a balloon at a distance of 302cm

The same object was captured at three different distances and in the following table the true and the retrieved distances are reported:

	True distance	Retrieved distance
Figure 5.8	95 cm	97 cm
Figure 5.9	171 cm	174 cm
Figure 5.10	302 cm	305 cm

Table 5.1: Comparison between the true and the retrieved distances in the three setups reported in the figures 5.8, 5.9 and 5.10

The little mismatch between the true and measured distances in the three setups are due to the fact that the detection retrieves the center of the object, instead the laser meter calculates the distance between the camera and the surface of the balloon. So the goal of reconstruct the 3D pose in the world reference frame given a stereo camera system is reached with high accuracy.

Conclusions

The aim of analyze some applications of the Computer Vision and Artificial Intelligence in the industrial field is reached, the ones explained in this thesis are only few examples of the infinite applications that can be developed. These technologies help make processes faster and more precise, with not only economic benefits but also support for working staff. In the quality control task automation leads to some evaluation criteria that are the same for all the screens and that not depends on the light condition of the environment or the personal evaluation of the human operator. In the conveyor tracking task the automatic localization of the objects on the belt and the result robot control allows the operator to carry out less alienating work in parallel with the robot, with benefits for both the worker and the entrepreneur. In the stereovision task the aim of reconstruct the 3D position of the object seen from two cameras is reached, this allows to obtain a lot more information of the environment with respect to other distance sensor, like lidars, data useful for further tasks. In conclusion the scientific disciplines of Computer Vision and Artificial Intelligence have enormous potential in the industrial fields, especially in the automation field they raising the level of difficulty that can be achieved in the tasks.

Appendix A

Camera calibration Python code

In this section the code used for the camera calibration task is reported, there exists a lot of open source codes that solves this task, for example I used the code of Temuge Batpurev. To make the calibration a set of photos is needed, each showing the same chessboard in space, and insert all the pictures in a folder. An example of picture is reported in the following image:

Figure A.1: Example of photo used for the camera calibration process

In the following code a function that solve the camera calibration task is reported, using a chessboard with 6x9 squares of size 2.1cm, where $images\_folder$ is the folder that contains all the images.

```
def calibrate_camera(image_folder):
    images_names = glob.glob(image_folder)
    images = []
```

```
for imname in images_names:
    im = cv.imread(imname, 1)
    images.append(im)
criteria = (cv.TERM_CRITERIA_EPS +
    cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

rows = 5 #number of checkerboard rows.
columns = 8 #number of checkerboard columns.
world_scaling = 0.021 #real world
    square size

#coordinates of squares in the checkerboard world space
objp = np.zeros((rows*columns,3), np.float32)
objp[:,:2] = np.mgrid[0:rows,0:columns].T.reshape(-1,2)
objp = world_scaling* objp

#frame dimensions. Frames should be the same size.
width = images[0].shape[1]
height = images[0].shape[0]

#Pixel coordinates of checkerboards
imgpoints = [] # 2d points in image plane.

#coordinates of the checkerboard in checkerboard world space.
objpoints = [] # 3d point in real world space

for frame in images:
    gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    #find the checkerboard
```

```
        ret, corners = cv.findChessboardCorners(gray, (rows,
            columns), None)

        if ret == True:

            conv_size = (11, 11)

            corners = cv.cornerSubPix(gray, corners, conv_size,
                (-1, -1), criteria)
            cv.drawChessboardCorners(frame, (rows,columns),
                corners, ret)
            cv.imshow('img', frame)
            cv.waitKey(500)

            objpoints.append(objp)
            imgpoints.append(corners)

    ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(
        objpoints, imgpoints, (width, height), None, None)
    print('rmse:', ret)
    print('camera matrix:\n', mtx)
    print('distortion coeffs:', dist)

    return mtx, dist
```

In the function just reported the number of rows and columns of the checkerboard refer to the corners, that are the the numbers of squares minus one. The function returns the camera matrix and the vector of distortion coefficients, the *world_scaling* variable scales the squares in the chessboard world space so that the algorithm is able to find an approximation of the real camera matrix and no one with a scaled focal length in pixels unit.

In the main code the function is called in the following way:

```
mtx, dist = calibrate_camera(image_folder)
```

Appendix B

Transformation matrix Python code

The approach described in the section 4.2.2 to find the transformation matrix from the camera frame to the robot frame is implemented in a simple Python code, shown below.

```
#camera
X=np.array([[-93.88,-11.68,73.09,135.62],
            [77.31,-69.5,73.84,-72.03],
            [0,0,0,0],
            [1,1,1,1]])
X_T=np.transpose(X)

#robot
offset_belt=575.5 #millimeters
Y_T= np.array([[110.89-offset_belt,49.34,-1000,1.],
               [33.73-offset_belt,-96.91,-1000,1. ],
               [-52.33-offset_belt,39.31,-1000,1.],
               [-110.74-offset_belt,-103.66,-1000,1.]])
```

```
A_T=np.dot(np.dot(np.linalg.pinv(np.dot(X,X_T)),X),Y_T)
```

In the code is possible to see the matrix X, which columns contain the position vectors in homogeneous coordinates of the object seen from the camera, and the Y_T matrix, which rows contain the position vectors in homogeneous coordinates of the same points seen from the robot. So the experimental approach is to place four object in random position in the robot workspace, measuring their exact position with respect to the robot frame, inserting these values in the matrix Y_T, moving the belt until all the objects are inside the camera view, take a photo of the object and measure the distances of the objects with respect to the center of the camera frame on the belt, inserting all the values in the matrix X. The resulting A_T is a 4x4 matrix that relates the object seen in the camera frame to the ones in the robot frames, more precisely: $Y\_T = X\_T A\_T$. With this approach the final transformation matrix is not said that it is only a rototranslation one, because there is no restriction on the determinant, that for transformation matrix that describes a rotation and translation must be equal to one, and the relation between coefficients of the rotational components, because each row and column must have unitary module and be orthonormal to each other. In the real setup all the measure incorporates noise, so it is very likely that the matrix found is not only a rototranslational one, but only an approximation of the true one. To find a better estimate of the real transformation matrix more couples of points are needed. The transformation matrices found, for both the RGB and grayscale cameras, are closed to represent a 180° rotation on the y axis plus the translation, for example in the following equation the transformation matrix for the RGB camera is reported:

$$A^T = \begin{bmatrix} -1.01 & 0.01 & 0 & 0 \\ 0 & 1.005 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -512 & 43 & 1050 & 1 \end{bmatrix} \tag{B.1}$$

This result is consistent with respect to the ideal chosen frames, as reported in the following figure:

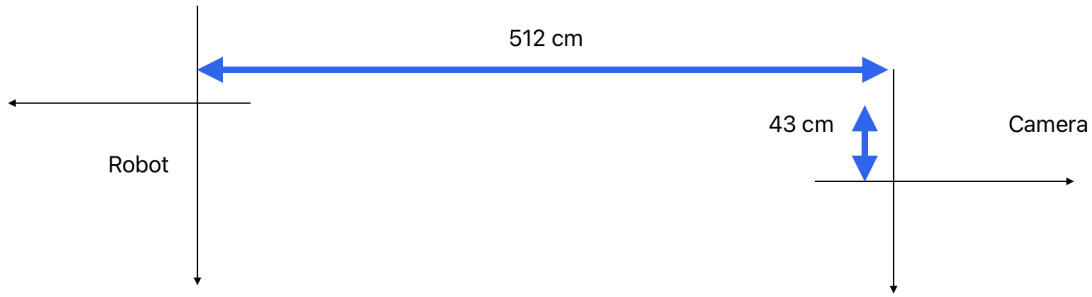


Figure B.1: Representation of the ideal chosen frames

Appendix C

Adept V+ code in ACE environment

The ACE Adept V+ code used for control the Delta robot was inspired by the code of the colleague Ettore Santoiemma, with some changes. The "communication_main" code is reported below:

```
.PROGRAM communication_main()
    ATTACH (lun, 4) "TCP"
    TYPE "lun: ", lun
    status = IOSTAT(lun) ;Check status of ATTACH
    IF status < 0 THEN
        TYPE "Error from ATTACH: ", $ERROR(status)
    END

    ;IP of Python server and the port opened
    FOPEN (lun, 0) "192.168.0.25 /REMOTE_PORT 20000
        /BUFFER_SIZE 4096"

    status = IOSTAT(lun) ;Check status of FOPEN
    IF status < 0 THEN
```

```
    TYPE "Error from FOPEN: ", $ERROR(status)
END

;to make the buffer empty
WHILE LATCHED(-1) DO
END

;variables definition
piece_list = 0
piece_new = 0
trig_pos = 0
do_wait = 1

WHILE TRUE DO

    ;request a picture every 20cm
    WAIT ABS((DEVICE(0,0,,1)-trig_pos)*sf) >= 200

    $take_photo = "10100"
    WRITE (lun) $take_photo ;Send to the server
    READ (lun, do_wait) $read_str ;Get reply from server

    ;update the values of interest of the controller like
        the trigger
    event = LATCHED(-1)

    ;read trigger
    trig_pos = DEVICE(0,0,,4)

    $number_detection = "10200"
```

```
WRITE (lun) $number_detection
READ (lun, do_wait) $read_str2 ;Get reply from server

piece_new = VAL($read_str2)
IF piece_new > 0 THEN
    IF DEVICE(0,0,,2) <= 0 THEN
        FOR i = 1 TO piece_new
            ;ask x coorinate piece number "i"
            $variable = $ENCODE(/i1,i)+$ENCODE(/i1,1)
            WRITE (lun) $variable ;Send to the server
            READ (lun, do_wait) $read_str_x ;Get reply
                from server
            $variable = $ENCODE(/i1,i)+$ENCODE(/i1,2)
            WRITE (lun) $variable ;Send to the server
            READ (lun, do_wait) $read_str_y ;Get reply
                from server
            $variable = $ENCODE(/i1,i)+$ENCODE(/i1,3)
            WRITE (lun) $variable ;Send to the server
            READ (lun, do_wait) $read_str_theta
            $variable = $ENCODE(/i1,i)+$ENCODE(/i1,4)
            WRITE (lun) $variable ;Send to the server
            READ (lun, do_wait) $read_str_location
            x_new = VAL($read_str_x)
            y_new = VAL($read_str_y)
            alpha_new = VAL($read_str_theta)
            location = VAL($read_str_location)

            ;adding the new pieces detected
            CALL def_add(x_new, y_new, alpha_new,
                trig_pos, location)
```

```
            END
        END
    END
END
.END
```

In the "communication_main" code the connection to the server is made, using the IP and port of the Python server, some variable are defined and the rest of the code is inside a WHILE loop. Into the WHILE loop the client ask to the server every times that the belt moves of 20cm to take a picture of the belt and there is a exchange of messages to obtain information regarding the number of object detected, their position, orientation and final desired place location. All these information are inserted in a list using the "def_add" function.
The "def_add" function is reported in the following code:

```
.PROGRAM def_add(x, y, alpha, trigger, location)
    threshold = 30
    check = TRUE

    x_temp = x
    y_temp = y
    alpha_temp = alpha
    location_temp = location

    IF piece_list == 0 THEN
        ;adding the next element in the list
        list[piece_list+1,0] = x_temp
        list[piece_list+1,1] = y_temp
        list[piece_list+1,2] = alpha_temp
        list[piece_list+1,3] = trigger
        list[piece_list+1,4] = location_temp
```

```
    ;updating the number of element in the list
    piece_list = piece_list+1
ELSE
    FOR k = 1 TO piece_list
        IF (y_temp <= list[k,1]+threshold) AND
            (y_temp >= list[k,1]-threshold) THEN

            IF (x_temp <= list[k,0]+((trigger-list[k,3])*sf)+
                threshold) AND (x_temp >= list[k,0]+
                    ((trigger-list[k,3])*sf)-threshold) THEN
                check = FALSE
            END
        END
    END

    IF check THEN
    ;adding the next element in the list
        list[piece_list+1,0] = x_temp
        list[piece_list+1,1] = y_temp
        list[piece_list+1,2] = alpha_temp
        list[piece_list+1,3] = trigger
        list[piece_list+1,4] = location_temp
    ;updating the number of element in the list
        piece_list = piece_list+1
    END
  END
.END
```

In the code just reported a threshold is set, useful to identify the same object that is detected in two subsequent photos. If the list is empty the coordinates of

the object are added in the list, otherwise there is a control in the position of the object detected: if the new detection is close to one already present in the list the new object is not added in the list, as it is considered the same object already present.

The "robot_main" code is reported below:

```
.PROGRAM robot_main()
        ATTACH ()    ;attach to the robot
        WAIT 0.5

        ;define relevant locations
        SET lochome = TRANS(0,0,-950,0,180,180)
        SET locway = TRANS(100,-200,-920,0,180,180)
        SET loccamera = TRANS(-512,-43,-1050,0,180,180)
        dz = 50

        ;define belt parameters
        sf = -0.25
        ENABLE BELT
        DEFBELT %locbelt = loccamera, 1, 1, sf
        WINDOW %locbelt = TRANS(500,500,0), TRANS(-500,-500,0)

        ;resetting the position and speed
        SPEED 10 ALWAYS
        MOVE lochome
        BREAK
        OPENI
        DELAY 0.5
        WHILE TRUE DO

            ;robot movement (pick the first object of the list)
```

```
        ;checks if there are pieces in the list
        IF piece_list >= 1 THEN

        ;waiting for the object to reach a safe distance
        ;i.e. enters the robot working space
            WAIT (list[1,0]+(DEVICE(0,0,,1)-list[1,3])*sf)>
                -300

        ;pick the first object of the list and put it in
            the "list[1,4]" place location

            CALL def_move(list[1,0]+512, list[1,1]+43,
                list[1,2], list[1,3], list[1,4])

        ;removing the first object of the list
            CALL def_rem()
        END
    END
    DETACH ()
.END
```

The program starts with the connection to the robot and defining the location of interest. There is the connection to the BELT, defining it as located on the camera frame with a "sf" conversion factor, with a windows of one square meter below the robot. The "sf" factor is -0.25 and it depends on the encoder used in the experimental setup, this converts the encoder values in millimeters. Before the "WHILE" cycle there is the movement of the robot to the "home" location and the opening of the gripper. Inside the "WHILE" cycle only if a new piece is present in the "list" of object detected the controller waits since the object enters the belt windows and control the robot using a function called "def_move". Once the movement ends the object is removed from the list using a function

called "def_rem". The "loccamera" location has as parameters the translational components of the transformation matrix between the camera and the robot and is needed because the robot must wait until the object enters the workspace of the robot before picking it. These parameter are then summed in the input of the "def_move" function, because the controller already takes in consideration the belt location moving in time, so we have to pass into the function the object in the camera coordinates and not in robot coordinates. So the calibration process that was used to found the transformation matrix between the camera and the robot is needed not only to defining the camera location but also to defining the location in the belt to the pick movement. Another approach may be to pass directly, from the Python server to the client ACE, the location of the objects with respect to the camera frame without adding the translational components of the transformation matrix camera-robot in the inputs of the move function, but this does not take in consideration any distortions.
The "def_move" code is reported below:

```
.PROGRAM def_move(x, y, alpha, pos_start, pos_end)

    IF pos_end == 1 THEN
        SET final_locplace = TRANS(100,-400,-950,0,180,180)
    END
    IF pos_end == 2 THEN
        SET final_locplace = TRANS(-20,-400,-950,0,180,180)
    END
    IF pos_end == 3 THEN
        SET final_locplace = TRANS(-20,400,-950,0,180,180)
    END

    ;setting the belt initial position as the position when
        the object has been individuated
    SETBELT %locbelt = pos_start
```

```
    APPRO %locbelt:TRANS(x,y,0,alpha,0,0), dz
    BREAK
    MOVE %locbelt:TRANS(x,y,0,alpha,0,0)
    BREAK
    CLOSEI
    DELAY 0.5
    DEPART dz
    BREAK

    ;placing object
    MOVE locway
    APPRO final_locplace, dz
    BREAK
    MOVE final_locplace
    BREAK
    OPENI
    DELAY 0.5
    DEPART dz
    BREAK
    MOVE locway
    MOVE lochome
    BREAK
.END
```

In the first part the final place location is set, based on the pos_end value, the location of the belt is set to the position when photo was made, because the controller sum in the x axis the distance that the belt travels in the time between when the photo was taken and when the robot is commanded. The robot approaches the object, moving in a parallel way with respect to the belt, picking the object and placing in the desired final location.

The "def_rem" code is reported below:

```
.PROGRAM def_rem()
        FOR l = 1 TO piece_list-1
            ;shifting up the list
            list[l,0] = list[l+1,0]
            list[l,1] = list[l+1,1]
            list[l,2] = list[l+1,2]
            list[l,3] = list[l+1,3]
            list[l,4] = list[l+1,4]
        END
        ;updating the number of element in the list
        piece_list = piece_list-1
.END
```

The code just reported show how the elements of the list are shifted when an object is placed by the robot.

Appendix D

Conveyor Tracking Python code

In this appendix the Python code used as server for the object detection task is reported, in the first section there is the computer vision function developed for object detection, in the second one there is the complete code that used YOLO for the object detection and finally in the third section there is an in-depth analysis on the code used for the training phase.

D.1 CV function detection

This function uses as input the image, called "frame" and outputs the lists x,y,theta and desired place of the objects detected, already explained in the section 4.2.2.

```
#gray image
frame_gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
#threshold it using a value found by hand
ret,thresh= cv2.threshold(frame_gray, 175, 255, cv2.THRESH_
    BINARY_INV)
#use some morphological operators
for u in range(5):
    kernel = np.ones((1,3),np.uint8)
    thresh = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)
```

```
    kernel = np.ones((3,1),np.uint8)
    thresh = cv2.morphologyEx(thresh, cv2.MORPH_CLOSE, kernel)
for u in range(5):
    kernel = np.ones((3,1),np.uint8)
    thresh = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel)
    kernel = np.ones((1,3),np.uint8)
    thresh = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel)
#find contours
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE,
    cv2.CHAIN_APPROX_SIMPLE)
#define some variables
grandezza_oggetto=120
numero_oggetti_trovati=len(contours)
x_oggetto_trovato=[]
y_oggetto_trovato=[]
area_oggetto_trovato=[]
theta_oggetto_trovato=[]
place_oggetto_trovato =[]
#iterate foe the numbers of contours
for i in range(len(contours)):
    #approximate area
    epsilon = 0.03*cv2.arcLength(contours[i],True)
    approx = cv2.approxPolyDP(contours[i],epsilon,True)
    area=cv2.contourArea(approx)
    #if area is too big or too small continue
    if area>20000 or area<200:
        print("Area: ",area)
        oggetto_good=False
        continue
    oggetto_good=True
```

```
#create black image
img_black=np.zeros((frame_gray.shape[0],frame_gray.shape[1]))
#draw contour
cv2.drawContours(img_black, [contours[i]], -1, (255), 3)
#find the center
x_max_contour=0
y_max_contour=0
x_min_contour=img_black.shape[1]
y_min_contour=img_black.shape[0]
for j in range(len(contours[i])):
    if y_min_contour >= contours[i][j][0][1]:
        y_min_contour=contours[i][j][0][1]
    if y_max_contour <= contours[i][j][0][1]:
        y_max_contour=contours[i][j][0][1]
    if x_min_contour >= contours[i][j][0][0]:
        x_min_contour=contours[i][j][0][0]
    if x_max_contour <= contours[i][j][0][0]:
        x_max_contour=contours[i][j][0][0]
y=int(y_min_contour+(y_max_contour-y_min_contour)/2)
x=int(x_min_contour+(x_max_contour-x_min_contour)/2)
#if it is close to the edges it is discarded
if y<frame_copy.shape[0]/10 or y>frame_copy.shape[0]*9/10
    or x<frame_copy.shape[1]/10 or x>frame_copy.shape[1]*
        9/10:
    continue

# test if it is a circle:
img_black = np.uint8(img_black)
linesP = cv2.HoughLinesP(img_black, 1, np.pi / 180, 70,
    None, 70, 10)
```

```
if linesP is None:
    place = 1
    theta=0
    if frame[y,x,0]<frame[y,x,2]: #check if it is red
        place=2
else:#  it has a rectangular shape:
    lunghezza=0
    linea=0
    place=3

    for k in range(0, len(linesP)):
        l = linesP[k][0]
        lung=math.sqrt(pow(l[0]-l[2],2)+pow(l[1]-l[3],2))
        if lung>=lunghezza:
            lunghezza=lung
            linea=k
    for k in range(0, len(linesP)):
        if k==linea:
            l = linesP[k][0]
            theta=math.atan2(abs(l[3]-l[1]),abs(l[2]-l[0]))

if oggetto_good==True:
    oggetto_da_sostituire=False
    oggetto_da_cancellare=False
    sostituzione=0
    if len(x_oggetto_trovato)>0:
        for h in range(len(x_oggetto_trovato)):
            if abs(x-x_oggetto_trovato[h])<grandezza_oggetto
                and abs(y-y_oggetto_trovato[h])<grandezza_
                    oggetto:
```

```
                    if area>area_oggetto_trovato[h]:
                        #replace i with h
                        oggetto_da_sostituire=True
                        sostituzione=h
                    else:
                        oggetto_da_cancellare=True

        if oggetto_da_sostituire==True:
            x_oggetto_trovato[sostituzione]=x
            y_oggetto_trovato[sostituzione]=y
            area_oggetto_trovato[sostituzione]=area
            theta_oggetto_trovato[sostituzione]=theta
            place_oggetto_trovato[sostituzione]=place
        elif oggetto_da_cancellare==False:
            x_oggetto_trovato.append(x)
            y_oggetto_trovato.append(y)
            area_oggetto_trovato.append(area)
            theta_oggetto_trovato.append(theta)
            place_oggetto_trovato.append(place)
```

D.2 Python server

In this section the full server code is reported, using YOLO for the object detection task:

```
def robot_position( x_px_camera, y_px_camera):
    global f_u_v_zp, A_T

    ux0=int(frame.shape[1]/2)
    uy0=int(frame.shape[0]/2)
```

```
    ux=x_px_camera
    uy=y_px_camera
    xp = (ux-ux0)/f_u_v_zp*1000
    yp = (uy-uy0)/f_u_v_zp*1000 #millimeters

    X_T = np.array([xp,yp,0,1])
    Y_T=np.dot(X_T,A_T)
    X_r=Y_T[0]
    Y_r=Y_T[1]
    return X_r, Y_r

def ricevi_comandi():
    global conn
    richiesta = conn.recv(4096)
    if richiesta.decode()!="":
        return richiesta.decode()
    else:
        return -1

def invia_comandi(data):
    global conn
    conn.sendall(str(data).encode())

def sub_server(indirizzo, backlog=1):
    #backlog = num max connections
    try:
        s = socket.socket()
        s.bind(indirizzo)
        s.listen(backlog)
    except:
```

```
        sub_server(indirizzo, backlog=1)
    conn, indirizzo_client = s.accept() #conn=socket_client
    return conn, s

def chiudi_server():
    global s
    s.close()
    sys.exit()

f_u_v_zp=6677
grandezza_oggetto=120
A_T = np.array([[ -0.9887,    -0.0427,   0.,   0. ],
                [-0.0194,   0.9625,    0.,    0. ],
                [ 0.,    0.,    0. ,  0. ],
                [ -554.9859,    -28.3367,  -1000,    1. ]])

model = torch.hub.load(r'.','custom',path=r'./best_blu_red.pt',
    source='local')  # custom trained model

conn, s= sub_server(("", 20000)) # IP is implied as ""
factor_img=4
dimension=[0,0]
flag_frame=False
webcam = cv2.VideoCapture(0)
width = 2000
height = 1940
webcam.set(cv2.CAP_PROP_FRAME_WIDTH, width)
webcam.set(cv2.CAP_PROP_FRAME_HEIGHT, height)
while True:
    start_time = time.time()
```

```
comando = ricevi_comandi()
comando=int(comando)
if comando!=-1:
    print("comando= ",comando)
if flag_frame == True and comando==-1:
    dimension[0]=int(frame.shape[1]/factor_img)
    dimension[1]=int(frame.shape[0]/factor_img)
    frame_resized = cv2.resize(frame, dimension,
        interpolation= cv2.INTER_AREA)
    cv2.imshow("frame", frame_resized)
    key = cv2.waitKey(10)
    if key == ord("q"):
        chiudi_server()
        break
if comando!=-1:
    if comando == 10100:

        ret, frame = webcam.read()

        if ret==True:
            invia_comandi(11111)
            flag_frame = True
            dimension[0]=int(frame.shape[1]/factor_img)
            dimension[1]=int(frame.shape[0]/factor_img)
            frame_resized = cv2.resize(frame, dimension,
                interpolation= cv2.INTER_AREA)
            cv2.imshow("frame", frame_resized)
            cv2.waitKey(1)

            frame_rgb = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
```

```
results = model(frame_rgb)

tensor_result = results.xyxy[0]
numpy_result = tensor_result.numpy()
#print(numpy_result)
#print(len(numpy_result)) #es len 2
#print(len(numpy_result[0]))



print("Numero di oggetti trovati: ",
    len(numpy_result))
X_robot=[]
Y_robot=[]
place=[]
for f in range(len(numpy_result)):
    x1=numpy_result[f][0]
    y1=numpy_result[f][1]
    x2=numpy_result[f][2]
    y2=numpy_result[f][3]
    acc=numpy_result[f][4]
    classe=numpy_result[f][5]
    x_mean=int((x1+x2)/2)
    y_mean=int((y1+y2)/2)
    if y_mean<frame.shape[0]/10 or
        y_mean>frame.shape[0]*9/10 or
            x_mean<frame.shape[1]/10 or
                x_mean>frame.shape[1]*9/10:
        continue
    place.append(classe+1)
    theta=0
```

```
            print("Oggetto trovato in x: ",x_mean,", y:
            ",y_mean,", theta: ",theta, ", place: ",place[f])
            if classe==0:
                cv2.circle(frame,(x_mean,y_mean),5,
                    (0,0,255),3)
            else:
                cv2.circle(frame,(x_mean,y_mean),5,
                    (255,0,0),3)
            X_r, Y_r = robot_position(x_mean,y_mean)
            X_robot.append(X_r)
            Y_robot.append(Y_r)
        dimension[0]=int(frame.shape[1]/factor_img)
        dimension[1]=int(frame.shape[0]/factor_img)
        frame_resized = cv2.resize(frame, dimension,
            interpolation= cv2.INTER_AREA)
        cv2.imshow("frame", frame_resized)
        cv2.waitKey(1)
        print("punti trovati=: ",len(X_robot))
        print("X_robot: ",X_robot)
        print("Y_robot: ",Y_robot)



        end_time=time.time()
        print("time: ",end_time-start_time)

    if (comando)==10200:
        invia_comandi(len(X_robot))
    if (comando%10)==1:
        invia_comandi(int(X_robot[comando//10-1]))
    if (comando%10)==2:
```

```
            invia_comandi(int(Y_robot[comando//10-1]))
        if (comando%10)==3:
            invia_comandi(int(0))
        if (comando%10)==4:
            invia_comandi(int(place[comando//10-1]))
```

At the beginning some function are reported, for example the "robot_position" function transforms the coordinates from the camera frame to the robot frame and the "sub_server" creates the connection. The camera settings are defined, like the resolution. If the command received is different from -1, default value that identifies no messages received, the server act as request by the client. If the command is "10100" the server takes a picture and detect the objects using the YOLO network, saving the information in lists. The other commands can be a two digits number: the first digit on the left represents the number of the object of which the server want to know information, the last digit instead refers to the information to send, one for the "x" coordinates, two for the "y" coordinates, three the position and four the desires place location.

D.3 YOLO training

In this section the code used on COLAB site is reported, through which the training of the network is made, the "—" represent a different section in the ".ipynb" file:

```
---
%cd /content
---
#Clone repo and install dependencies
!git clone https://github.com/ultralytics/yolov5  # clone
%cd yolov5
%pip install -qr requirements.txt  # install
```

```
import torch
import utils
---
#import the dataset, for example from Roboflow
---
%cd /content/yolov5/
!python train.py --data dataset_folder/data.yaml --epochs 10 --batch-size 2
```

This is an example of training code for the YOLOv5 CNN, a similar version was implemented for the YOLOv8 CNN. The dataset is task dependent and some examples can be found in the "Roboflow" website. In the "runs" folder is possible to find the trained model with ".pt" extention.

Appendix E

Stereovision code

To solve the stereovision task I modified the code of Temuge Batpurev. The code is divided into two parts: the pre-processing and the execution of the task. In the pre-processing part the single camera calibration for both the camera and the stereo camera calibration are made: for the single camera one the code used is reported in the Appendix A, retrieving the two camera matrices and the two vectors of distortion coefficients, used as input in the stereo camera calibration process. For the stereo camera calibration pairs of photos reporting the same chessboard in 3D space are needed, these are inserted into two folders, each one referring to photos captured with a single camera. Furthermore the pictures need to have the same name order in both the folders so that is possible to reconstruct, iterating through the names within the folder, the pairs of photos captured. For the stereo camera calibration the following function is used, that takes as input the two camera matrices, the two vector of distortion coefficients and the two names of the folder containing the photos of the chessboard already described:

```
def stereo_calibrate(mtx1, dist1, mtx2, dist2,
    folder_1,folder_2):
    #read the synched frames
    images_names = glob.glob(folder_1)
    images_names = sorted(images_names)
```

```
    c1_images_names = images_names

    images_names = glob.glob(folder_2)
    images_names = sorted(images_names)
    c2_images_names = images_names

    c1_images = []
    c2_images = []
    for im1, im2 in zip(c1_images_names, c2_images_names):
        _im = cv.imread(im1, 1)
        c1_images.append(_im)

        _im = cv.imread(im2, 1)
        c2_images.append(_im)

    criteria = (cv.TERM_CRITERIA_EPS +
        cv.TERM_CRITERIA_MAX_ITER, 100, 0.0001)

    rows = 5 #number of checkerboard rows.
    columns = 8 #number of checkerboard columns.
    world_scaling = 0.021

    #coordinates of squares in the checkerboard world space
    objp = np.zeros((rows*columns,3), np.float32)
    objp[:,:2] = np.mgrid[0:rows,0:columns].T.reshape(-1,2)
    objp = world_scaling* objp

    #frame dimensions. Frames should be the same size.
    width = c1_images[0].shape[1]
height = c1_images[0].shape[0]
```

```
#Pixel coordinates of checkerboards
imgpoints_left = [] # 2d points in image plane.
imgpoints_right = []

#coordinates of the checkerboard in checkerboard
    world space.
objpoints = [] # 3d point in real world space

for frame1, frame2 in zip(c1_images, c2_images):
    gray1 = cv.cvtColor(frame1, cv.COLOR_BGR2GRAY)
    gray2 = cv.cvtColor(frame2, cv.COLOR_BGR2GRAY)
    c_ret1, corners1 = cv.findChessboardCorners(gray1,
        (5,8), None)
    c_ret2, corners2 = cv.findChessboardCorners(gray2,
        (5,8), None)

    if c_ret1 == True and c_ret2 == True:
        corners1 = cv.cornerSubPix(gray1, corners1,
            (11, 11), (-1, -1), criteria)
        corners2 = cv.cornerSubPix(gray2, corners2,
            (11, 11), (-1, -1), criteria)

        cv.drawChessboardCorners(frame1, (5,8), corners1,
            c_ret1)
        cv.imshow('img', frame1)

        cv.drawChessboardCorners(frame2, (5,8), corners2,
            c_ret2)
        cv.imshow('img2', frame2)
```

```
            cv.waitKey(500)

            objpoints.append(objp)
            imgpoints_left.append(corners1)
            imgpoints_right.append(corners2)

    stereocalibration_flags = cv.CALIB_FIX_INTRINSIC
    ret, CM1, dist1, CM2, dist2, R, T, E, F =
        cv.stereoCalibrate(objpoints, imgpoints_left,
            imgpoints_right, mtx1, dist1, mtx2, dist2,
                (width, height), criteria = criteria,
                    flags = stereocalibration_flags)

    return R, T
```

In this function the corners of the chessboard are found for each pairs of images, in the same way as for the single camera calibration code, but the list of the corners of the left and right images are used as input of the Opencv function "stereoCalibrate" that solves the problem of pose reconstruction between cameras, giving the rotational matrix and translational vector of the left camera with respect to the right, since I defined the reference frame of the stereo camera system on the right camera.

The second part of the code is the true execution of the task: object 3D pose recontruction given the two images captured from the stereo camera. This part can be executed in two ways: post-processing or real-time. In both cases the steps to obtain the 3D pose of the desired object are the same: find the locations of the object in the two images and reconstruct the 3D pose in space through triangulation. For the real-time case the detection can be done using a YOLO CNN, like in the case of the red and blue balls used in the conveyor tracking task, so let's focus on the triangulation part. The following code is the triangulation function used to retrieve the 3D pose of the desired object, this makes the use of

the DLT method explained in the section 5.2.2. The function takes as input the coordinates of the object seen in the two images, the two camera matrices, the rotation matrix and translational vector founded during the stereocalibration; it outputs the 3D coordinates of the object. It function also in the case of multiple detection, passing as input the lists of coordinates of all the objects seen in the two images.

```
def triangulate(mtx1, mtx2, R, T, points_1, points_2):
    global ax, fc, fig
    uvs1 = np.array(points_1)
    uvs2 = np.array(points_2)

    #RT matrix for C1 is identity.
    RT1 = np.concatenate([np.eye(3), [[0],[0],[0]]],
        axis = -1)
    P1 = mtx1 @ RT1 #projection matrix for C1

    #RT matrix for C2 is the R and T obtained
        from stereo calibration.
    RT2 = np.concatenate([R, T],
        axis = -1)
    P2 = mtx2 @ RT2 #projection matrix for C2

    def DLT(P1, P2, point1, point2):

        A = [point1[1]*P1[2,:] - P1[1,:],
             P1[0,:] - point1[0]*P1[2,:],
             point2[1]*P2[2,:] - P2[1,:],
             P2[0,:] - point2[0]*P2[2,:]
            ]
        A = np.array(A).reshape((4,4))
```

```
        B = A.transpose() @ A
        from scipy import linalg
        U, s, Vh = linalg.svd(B, full_matrices = False)

        print('Triangulated point: ')
        print(Vh[3,0:3]/Vh[3,3])
        return Vh[3,0:3]/Vh[3,3]

    p3ds = []
    for uv1, uv2 in zip(uvs1, uvs2):
        _p3d = DLT(P1, P2, uv1, uv2)
        _p3d=world_position(_p3d)
        p3ds.append(_p3d)
    zero = np.array([0,0,0,1])
    p3ds = np.array(p3ds)
    return p3ds
```

The following code reports the main code used for the stereovision task:

```
#import of: mtx1 mtx2 R T

#load model
model_yolo = torch.hub.load(r'.','custom',path=r'./best_blu_red.pt',
    source='local')  # custom trained model

webcam_l = cv2.VideoCapture(1)
webcam_r = cv2.VideoCapture(2)

while True:
    ret_r, frame_r = webcam_r.read()
```

```
ret_l, frame_l = webcam_l.read()
if ret_l==True and ret_r==True:
        x1,y1,x2,y2=0,0,0,0
        results_r = model_yolo(frame_r)
        results_l = model_yolo(frame_l)

        tensor_result_r = results_r.xyxy[0]
        numpy_result_r = tensor_result_r.numpy()
        tensor_result_l = results_l.xyxy[0]
        numpy_result_l = tensor_result_l.numpy()


        if len(numpy_result_r)==1 and len(numpy_result_l)==1:
            x1=numpy_result_r[0][0]
            y1=numpy_result_r[0][1]
            x2=numpy_result_r[0][2]
            y2=numpy_result_r[0][3]
            x_mean_r=int((x1+x2)/2)
            y_mean_r=int((y1+y2)/2)

            x1=numpy_result_l[0][0]
            y1=numpy_result_l[0][1]
            x2=numpy_result_l[0][2]
            y2=numpy_result_l[0][3]
            x_mean_l=int((x1+x2)/2)
            y_mean_l=int((y1+y2)/2)

            points_1[0][0]=x_mean_r
            points_1[0][1]=y_mean_r
            points_2[0][0]=x_mean_l
```

```
points_2[0][1]=y_mean_l

p3ds=triangulate(mtx1, mtx2, R, T,points_1,points_2)
```

Given the two camera metrices, the rotation matrix and translational vector, the yolo model is loaded, the initialization of the two cameras is made, the rest of the code is inside a while loop. In the while loop the frames are captured from the two cameras and the object detection is made using YOLO, the result is trasformed into the array "numpy_result". For simplicity I consider the case of single object detection, so only if only one object is detected, from the "numpy_result" the information about the center of the detection are extracted and used in the "triangulate" function already described. The resulting point is a vector containing the coordinates with respect to the right camera frame of the 3D point in space. This can be also transformed, with another transformation matrix, to a point with respect to a world reference frame, finding the transformation matrix like the conveyor tracking case.

Bibliography

[1] G. R. C. and W. R. E., *Digital Image Processing.* England: Pearson Education Limited, 2018.

[2] http://www.cs.toronto.edu/~kyros/courses/418/Notes/Camera.pdf.

[3] S. Ghidoni, *Computer Vision course*, University of Padua, 2021.

[4] M. Martinello, "Coded aperture imaging," Ph.D. dissertation, 05 2012.

[5] https://docs.opencv.org/4.x/d2/dbd/tutorial_distance_transform.html.

[6] https://thepythoncode.com/article/sift-feature-extraction-using-opencv-in-python?utm_content=cmp true.

[7] J. Pegoraro, *Neural Networks and Deep Learning course*, University of Padua, 2022.

[8] https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5 way/.

[9] https://arxiv.org/pdf/1506.02640.

[10] https://arxiv.org/pdf/2304.00501.

[11] R. Carli, *Robotics and control I course*, University of Padua, 2021.

[12] https://omron.it/it/home.

[13] G. Boschetti, *Industrial Robotics course*, University of Padua, 2022.

[14] A. Cenedese, *Robotics and Control II course*, University of Padua, 2022.

[15] https://temugeb.github.io/opencv/python/2021/02/02/stereo-camera-calibration-and triangulation.html.