People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University Amar Thelidji- Laghouat

FACULTY : Technology

DEPARTMENT : Electronic

MASTER MEMORY

Presented by :

KACEMI Abdelbadiaa
and
ABDESSELAM Sofiane

FIELD : SCIENE AND TECHNOLOGY

FILIERE : AUTOMATION AND INDUSTRIAL INFORMATIC

OPTION : AUTOMATION

Theme

Positioning a unicycle mobile robot by 2D visual servoing

Defence jury :

Last Name and First Name	Grade	Quality
Mrs. CHOUIREB Fatima	Professor at UATL	President of the jury
Mrs. BENKOUIDER Fatiha	MCB at UATL	Examiner
Mrs. FEKNOUS Safia	MAA at UATL	Supervisor

Promotion: September- 2020

ملخص

المهمة التي كان علينا تحقيقها عن طريق المحاكاة في هذا المشروع هي قيادة روبوت متحرك أحادي الدراجة مزود بكاميرا على متنه ، إلى موقع محدد بالنسبة إلى معلم موجود في بيئة الروبوت. يتم التحكم في محكات الروبوت بحيث يصل إلى الموضع المطلوب من خلال تحكم مؤازر بصري ثنائي الأبعاد. تم إجراء المحاكاة باستخدام المحاكي 'كوبيلياسيم' 'CppeliaSim' الذي يوفر ميزات مهمة مثل إمكانية إنشاء بيئة افتراضية ثلاثية الأبعاد من اختيارنا ، باستخدام نموذج من روبوت متحرك وكاميرا جاهزة للاستخدام في برنامج كوبيلياسيم ، والتقاط صور للبيئة باستخدام الكاميرا الافتراضية . في نظام التحكم هذا تأتي معلومات المستشعر الوحيدة من الكاميرا. لتحديد المعلومات المرئية المطلوبة في مخطط التحكم ، يتم تنفيذ معالجة الصور الملتقطة بالكاميرا باستخدام تحويل دائرة هوف نظرًا للشكل المميز للمعلم الذي يتكون من أربعة دوائر وهذا باستخدام مكتبة 'أوبنسيفي' 'OpenCV' . تمت كتابة برنامج التحكم بالكامل بلغة C ++.

الكلمات المفتاحية : مؤازرة بصرية ثنائية الأبعاد ، روبوت متحرك أحادي الدراجة ، تحويل دائرة هوف ، محاكي كوبيلياسيم .

Abstract

The task that we had to achieve by simulation in this project is the positioning relative to a landmark placed in the environment, of a unicycle mobile robot equipped with an on-board camera. The control of the robot's actuators so that it reaches the desired position is achieved with a 2D visual servo-control. The simulation was made with the CppeliaSim simulator which offers very interesting features such as the possibility of creating a three-dimensional virtual environment of our choice, using ready to use CoppeliaSim models of mobile robot and camera. In such a control system, the only sensor information comes from the camera. To determine the visual information required in the control scheme, and because of the characteristic shape of the landmark (four circles), a processing of the images captured with the camera is performed with the Hough Circle Transform using OpenCV library. The entire control program was written in C ++.

Keywords: 2D visual servoing, Unicycle mobile robot, Circle Hough transform, CoppeliaSim simulator.

Résumé

La tâche que nous avions à réaliser dans ce projet est la simulation du positionnement d'un robot mobile unicycle équipé d'une caméra embarquée, par rapport à un « amer » placé dans l'environnement. Le contrôle des actionneurs du robot pour qu'il atteigne la position souhaitée est réalisé grâce à un asservissement visuel 2D. La simulation a été réalisée avec le simulateur CppeliaSim qui offre des fonctionnalités très intéressantes, comme la possibilité de créer un environnement virtuel tridimensionnel de notre choix, utiliser un modèle de robot mobile et un modèle de caméra prêts à l'emploi. Dans un tel système de contrôle, les seules informations capteur proviennent de la caméra. Pour déterminer les informations visuelles requises dans le schéma de contrôle, et en raison des formes choisies pour l'amer (quatre cercles), un traitement des images capturées est effectué avec la transformation de Hough pour la détection de cercles à l'aide de la bibliothèque OpenCV. L'ensemble du programme de contrôle a été écrit en C ++.

Mots clés : Asservissement visuel 2D, Robot mobile unicycle, Transformé de Hough pour la détection des cercles, Simulateur CoppeliaSim.

Table of contents

Chapter II
Simulation and programming tools

Chapter III
Simulation and results

Figures table

Chapter I: Mobile Robot and Visual Servoing

Chapter II: Simulation and programming tools

Chapter III: Simulation and results

General introduction

Over the past thirty years, algorithmic and technological advances have made possible to link the aspects of perception and action more closely, integrating directly the measurements provided by a vision system into closed loop control laws on the visual information extracted. This approach, called visual servoing, joins the work on the referenced sensor control and is the subject of our project.

With a vision sensor, providing in basis 2D information, the nature of the potential measurements is extremely rich, since we can consider in visual servoing both 2D measurements, such as coordinates of characteristic points in the image for example, and 3D measurements, provided by a location module exploiting the 2D information extracted. From this wealth comes the major difficulty of visual servoing, namely, among all the potential information, how to select those that will provide satisfactory behavior to the system.

Our aim is to place a robot with an embedded camera somewhere in its environment according to an image captured from this place. To realize this objective we tried the technic of 2D visual servoing on a simulated unicycle mobile robot and environment by using the CoppeliaSim simulator.

Before giving the results of this simulation in the chapter 3, we speak on chapter 1of the unicycle mobile robot cinematic model, the camera’s geometric model, and the 2D visual servoing principals and equations.

In the chapter 2 we present the tools used for making this realization, as CoppeliaSim simulator, C++ openCV the C++ library of Computer Vision, and the library Armadillo for matrix computation.

Finally, in the conclusion we give our remarks about the simulation and results.

Chapter I

Mobile Robot and Visual Servoing

I.1. Introduction

In this chapter we will present the theoretical concepts and mathematical models of the different parts of our control system which is composed of a unicycle mobile robot, a camera, the image features extraction algorithm part, and the controller. For each part we will present only the details that must be known for realizing the robot control by simulation.

I.2. Mobile robots

I.2.1. Definitions

A mobile robot can be defined as a machine equipped with perception, decision-making and action capabilities that enable it to act autonomously in its environment depending on its perception. In particular, an autonomous mobile robot is a complex mechanical, electronic and computer system that implements:

- A set of **sensors** (exteroceptive and proprioceptive): Exteroceptive sensors provide information about the environment close to the robot. Such as the distances between the robot and nearby obstacles, or detecting a collision with an obstacle, or providing 2D images of the explored environment. Proprioceptive sensors provide information of the robot's internal condition, such as its speed or position.
- A set of **actuators**: The actuators allow the robot to move in its environment. The most common are wheeled robots, but there are also walking robots, flying robots, underwater robots and crawling robot.

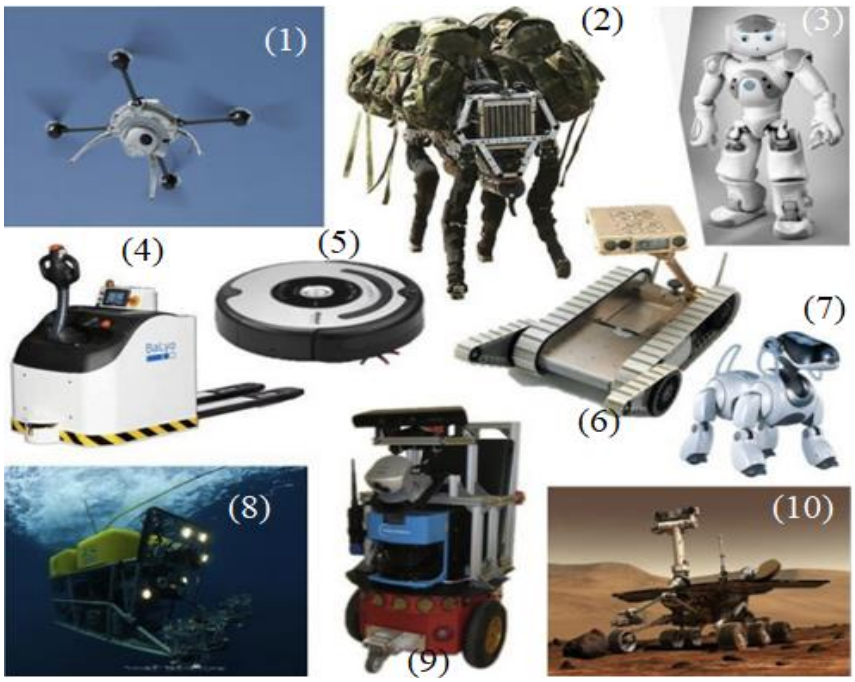


Figure I.1. Different types of mobile robots

(1) Drone - (2) Quadrupedal dog military - (3) Aibo - (4) AGV/AGC - (5) The Roomba household robot – (6) The Packbot - (7) Asimo - (8) Submarine - (9) Pioneer 3DX mobile robot - (10) The Rover (NASA)

The type of locomotion defines two types of constraints:

- **Cinematic constraints** are related to geometry possible movements of the robot.
- **Dynamic constraints** are related to the effects of movement (limited accelerations, bounded speeds, inertia and friction)

I.2.2. Wheeled mobile robots

Wheels are the most common means of robotic locomotion. In fact, mobile wheeled robots are easy to make and have great possibilities for movement and manoeuvrability with a speed significant acceleration. The type of wheels (fixed, orientable or Swedish wheels) and their arrangement defines the type of the robot locomotion. There are several types of wheeled mobile robots. The main ones are the unicycle robot, the car robot and the omnidirectional robot.

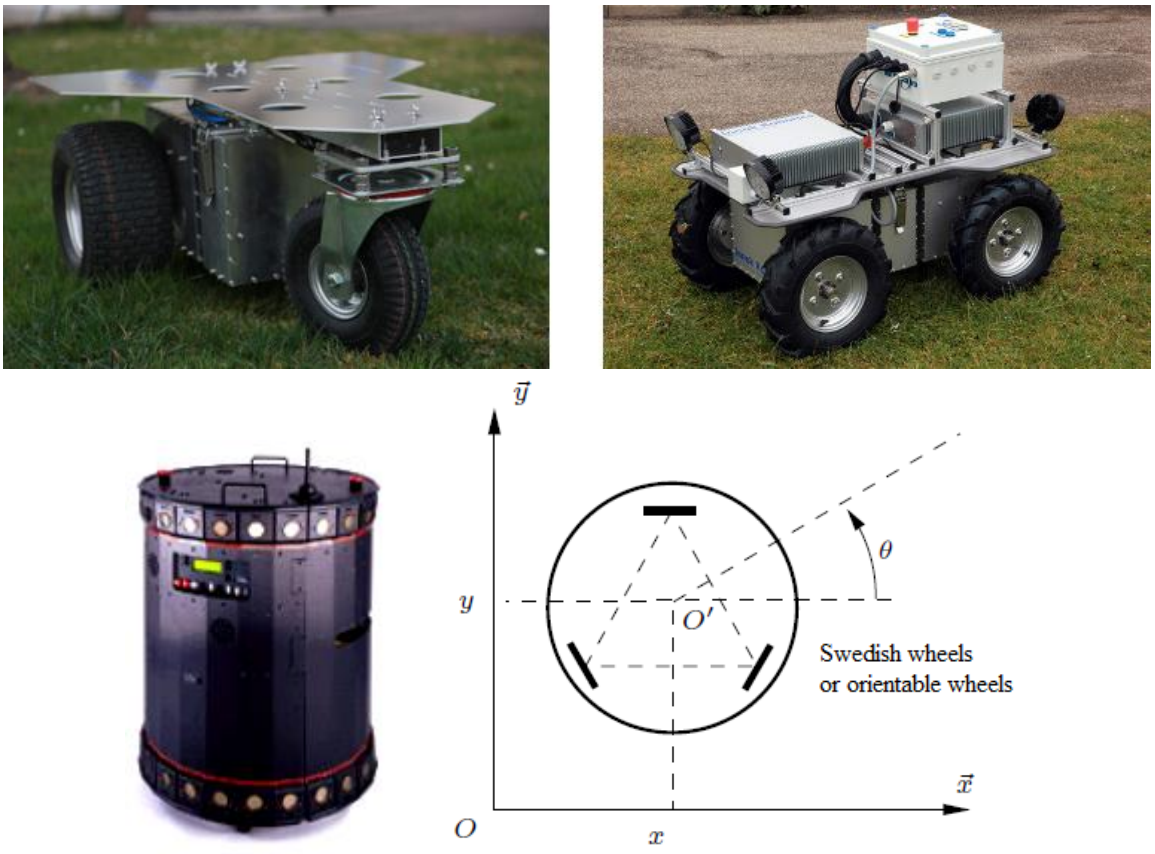


Figure I.2. Unicycle, car and omnidirectional mobile robots

I.2.3. Unicycle mobile robot

Unicycle robot is actuated by two independent wheels and possessing eventually a number of freewheels (casters) to ensure its stability. The diagram of a unicycle type robot is given in figure I.3.

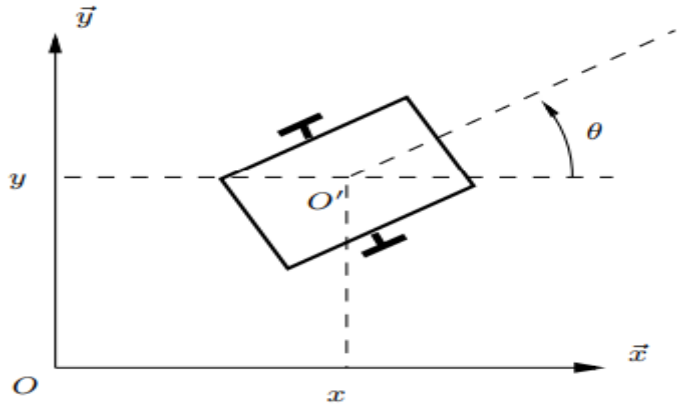


Figure I.3. Unicycle mobile robot posture (Position and orientation) in the coordinate system $(O,\vec{x},\vec{y},\vec{z})$

I.2.4. Cinematic model of unicycle robot

Suppose the robot is moving according to an arbitrary trajectory see figure I.4. The robot wheels have radius r. L is the distance between the two wheels. A point CIR is the Instant Rotation Centre. R is the curvature radius of the robot's trajectory. ω is the robot rotation speed around the CIR. v_d and v_g are respectively the speeds of the right and left wheels. ω_d and ω_g are respectively the rotation speeds of the right and left wheels. (x,y) and θ define the position and orientation of the robot in the coordinate system $(O,\vec{x},\vec{y},\vec{z})$. We call (x,y,θ) the posture of the robot.

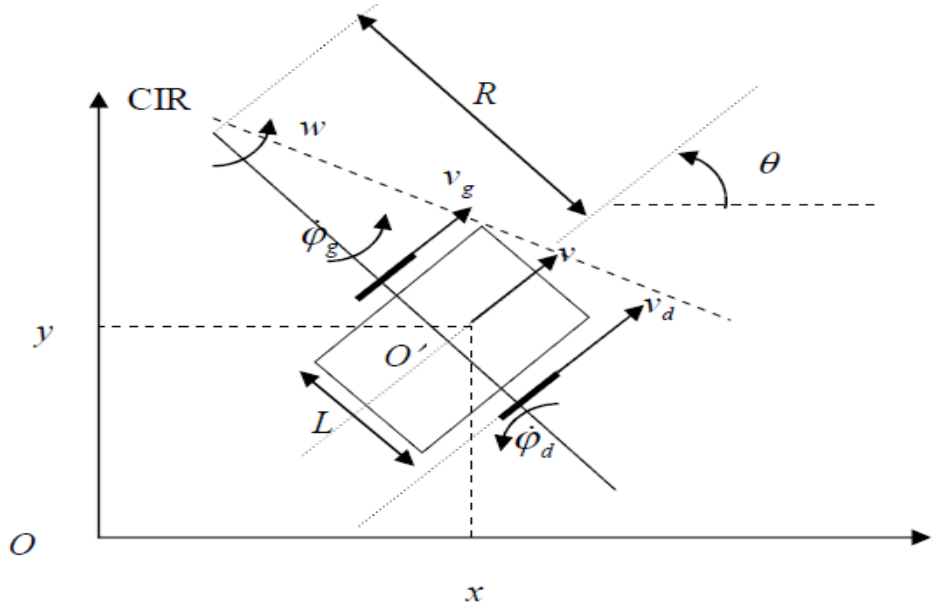


Figure I.4. Instant Rotation Centre (CIR) of a unicycle-type robot

The speeds of the right and left wheels are calculated by the equations :

$$\boldsymbol{v_d = \left(R + \frac{L}{2}\right)\omega = r\omega_d} \qquad \textbf{(I.1)}$$

$$\boldsymbol{v_g = \left(R - \frac{L}{2}\right)\omega = r\omega_g} \qquad \textbf{(I.2)}$$

$$\boldsymbol{v = R\omega} \qquad \textbf{(I.3)}$$

From equations **(I.1)**, **(I.2)** and **(I.3)** we deduce:

- The robot longitudinal and rotation speeds (v and ω) as function of the right and left speed:

$$\boldsymbol{v = \frac{r}{2}\left(\omega_d + \omega_g\right)} \qquad \textbf{(I.4)}$$

$$\boldsymbol{\omega = \frac{r}{L}\left(\omega_d - \omega_g\right)} \qquad \textbf{(I.5)}$$

- The rotation speed of the right and left wheels as function of v and ω and the robot dimensions:

$$\boldsymbol{\omega_d = \frac{2v + L\omega}{2r}} \qquad \textbf{(I.6)}$$

$$\boldsymbol{\omega_g = \frac{2v - L\omega}{2r}} \qquad \textbf{(I.7)}$$

It is also shown that the robot rotation speed is equal to its rotation speed around the CIR:

$$\omega = \dot{\theta} \qquad \textbf{(I.8)}$$

At the lowest level (hardware level) we can consider $\left(v_d, v_g\right)$ as the command of the unicycle robot, however, we generally prefer to express it by (v, ω).

The **cinematic model** of unicycle mobile robot is given by the relation between the derivative of the posture $\left(\dot{x}, \dot{y}, \dot{\theta}\right)$ and the command. It is simply determined geometrically :

$$\begin{cases} \dot{x} = v\cos\theta \\ \dot{y} = v\sin\theta \\ \dot{\theta} = \omega \end{cases} \qquad \textbf{(I.9)}$$

I.3. Pioneer P3DX robot

I.3.1. Definitions

The Pioneer P3-DX is one of several models of mobile robots manufactured by Adept Mobile Robots, LLC. The robot is widely used as a platform for robotics education and research. It is a ground-based robot that has two driven wheels and one free caster wheel at its rear.
The robot is equipped with an array of sonars for sensing obstacles, wheel encoders for odometry, and bumpers to detect collisions. At the heart of the Pioneer robot is the Advanced Robot Control and Operations Software (ARCOS), which runs on the robot's installed microcontroller. ARCOS handles all of the low-level control of the robot, such as operating the motors, tracking wheel encoder signals, reporting sonar data, and communications with a connected client. ARCOS has a client–server architecture where the user is required to develop his own client application to communicate with the ARCOS server. Since the ARCOS server manages the low-level control of the robot, this architecture allows the user to focus on the higher-level functionality of the robot control application.
The Pioneer 3 DX is fully capable of mapping its environment, finding its way home and performing other sophisticated path-planning tasks.

Figure I.5. Pioneer 3DX mobile robot

I.3.2. Technical specifications of the P3-DX robot

- Traversable terrain: indoor, wheelchair accessible
- 1.6 mm lacquered aluminium body - Foam-filled rubber tires
- 252Wh batteries, 2 DC motors with encoders
- Processor : Hitachi HS-8 microcontroller
- Sensors : Odometer, 8 US sensors in front + options (bumpers, laser telemeter, gyroscope)
- Max. speed: 1.2 m/s (peaks up to 1.6 m/s) - Rotation speed: 300°/s
- Autonomy: 8-10 hours with 3 batteries (with no accessories)
- Dimensions (L x l x h) : 44cm x 38 x 22cm
- Weight: 9 kg (Maximum payload: 17 kg)

I.3.3. **Optional** Accessories

- Laser-range finders
- Mono- and stereo-vision cameras
- Rear SONAR
- Wireless serial to Ethernet for remote operation
- Robotic arms and grippers
- Gyroscope
- Segmented bumper arrays
- Speakers and microphones
- Joystick

Figure I.6. Pioneer P3-DX sensors and actuators

Figure I.7. Pioneer P3-DX sensors and actuators alternative configuration

I.4. Camera

I.4.1. Definitions

The camera is one of the most essential tools in computer vision. It is the device by which we can record images of the world around us for using them for various applications.

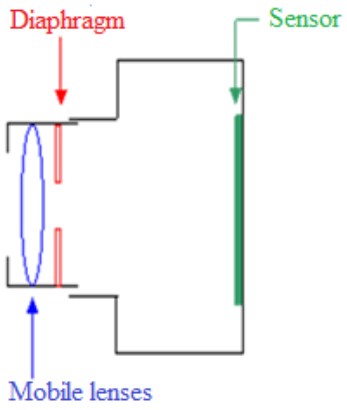


Figure I.8. Schema of a camera

The camera consists of three parts: Optical system, image sensor and an electronic card (figure 1.9).

Figure I.9. Camera components

1. **Optical system** (the Objective): It is composed of a set of lenses, diaphragm and mirrors.
   - **Lenses**: A lens is a transparent medium of glass or plexiglass, limited by one or two spherical surfaces. The set of lenses of a camera is equivalent to unique convex lens. The rays emitted by an object point on a convex lens intersect at one point out of the lens forming the image point (figure I.10)

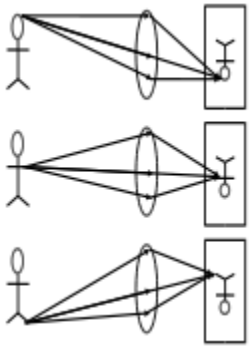


Figure I.10. Getting an image point with a convex lens

- **Diaphragm**: A diaphragm is a mechanical element which is composed of overlapping metal blades (the iris) that open and close to change the size of the opening. It conditions the amount of light transmitted and the depth of field of the camera.

2. **Image sensor**: An image sensor comes in the form of an integrated-circuit with a photosensitive surface. This photosensitive surface is a matrix of photodiodes called photosites. Each photosite corresponds to an image element (pixel) and has a size of 3 µm to 30 µm, and converts the amount of light to which it is exposed into a number of electrons (electric charge). CCD and CMOS are the most common technology of image sensor. The image sensor is placed at a distance equal to the focal length from the objective equivalent convex lens.

3. **Electronic card** : It is dedicated to digitizing the image, processing it with the integrated demosaicing and compression programs (jpg, png, etc.), and storing it in a storage memory.

The **shutter** is used to hide the sensor image at the end of the image capture.

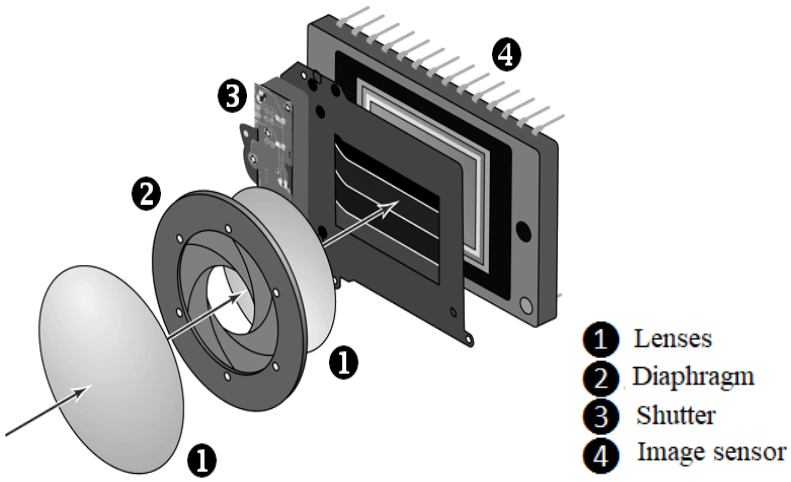


Figure I.11. Exploded view of a camera

I.4.2. Camera characteristics

1. **Focal length**
2. **Vision angle**: The Field of View (FOV) is characterized by horizontal and vertical viewing angles. The vision angle depends on the focal length and the sensor dimensions (height and width). (figure I.12 and figure I.13)
3. **Image resolution**: Image resolution is the detail an image holds. It is described by two numbers, the number of pixel columns (width) and the number of pixel rows (height), for example as *1024 × 512*.
4. **Frame rate**: It is the frequency (rate) at which consecutive images in a video (called frames) appear on a display. It is expressed in frame/second.

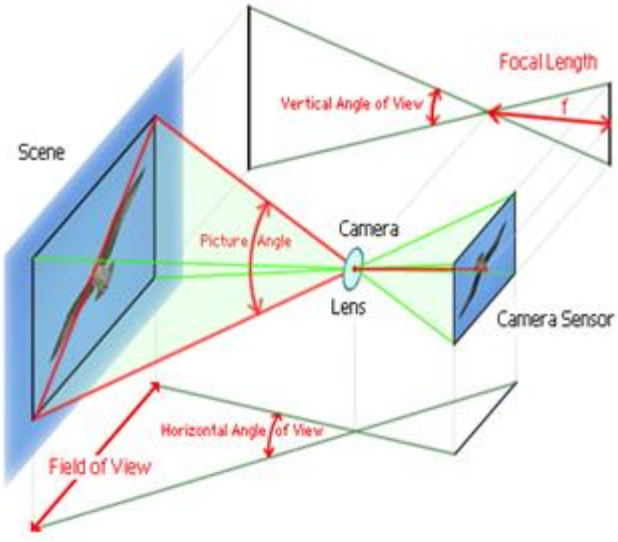


Figure I.12. Field of view of a camera

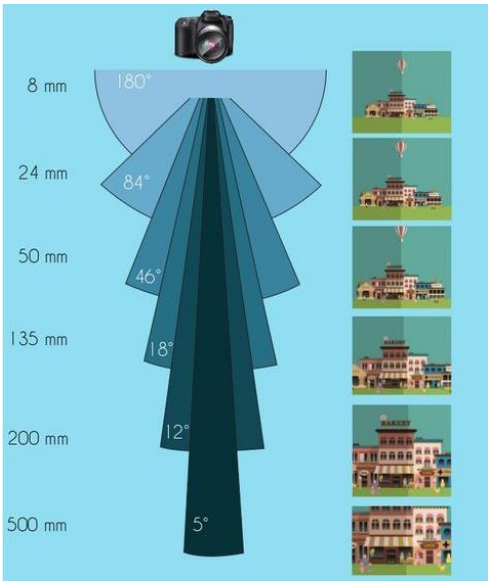


Figure I.13. Effect of the FOV on the image acquired

I.4.3. Camera's mathematical model

It is equivalent, in mathematical terms, to a projection of the scene with respect to the lens optical center (called projection center too) on the sensor surface. This type of projection is called perspective projection. The image obtained by the perspective projection model is reversed. To obtain an image in the right side up, the geometric model is modified by placing the image plane in front of the optical center. (figure I.14)

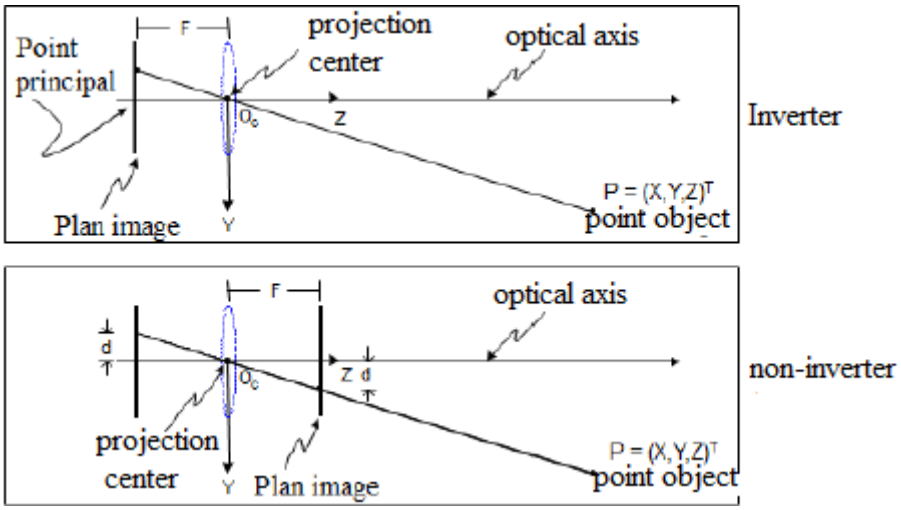


Figure I.14. Inverted and non-inverted camera model

Also three coordinates systems must be defined: (figure I.15)

- Scene (or world) coordinates system (Arbitrary defined in the scene).
- Image coordinate system (u, v, w) (as shown on the figure).
- Camera coordinates system(x, y, z): Where the origin of the coordinate system is confused with the optical center and the Z axis is confused with the optical axis in the direction of the visualized scene.

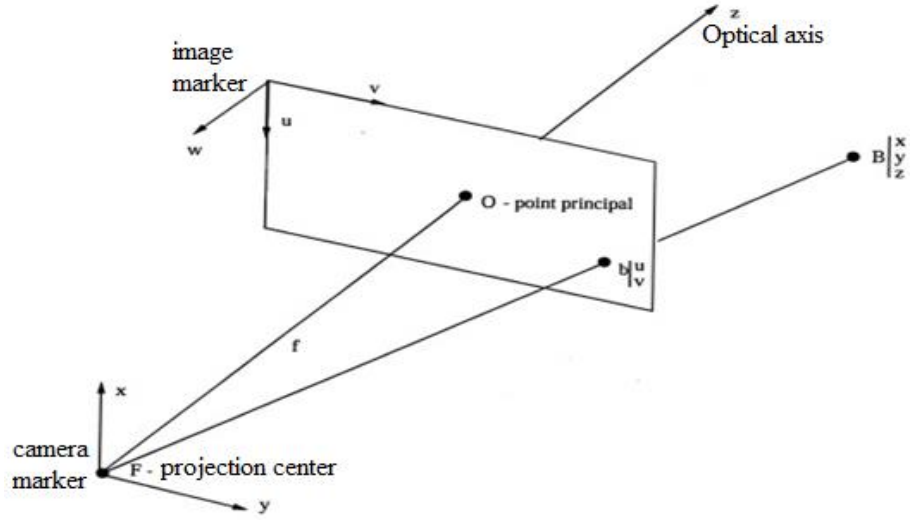


Figure I.15. Camera's geometrical model

Let be an object point B and its image b. There coordinates in the camera coordinates system:

$$B=\begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad , \quad b=\begin{pmatrix} \acute{x} \\ \acute{y} \\ f \end{pmatrix} \qquad \textbf{(I.10)}$$

Points B and its image b are linked by the following relationships:

$$\frac{x}{\acute{x}} = \frac{y}{\acute{y}} = \frac{z}{\acute{z}} = s \quad , \quad z' = f \qquad \textbf{(I.11)}$$

The coordinates of image points must be given in pixels in the image coordinates system (O,u,v). To convert distances into pixels we use horizontal and vertical scale factors ku and kv (given in pix/mm).

$$x_{pix} = K_v \times x_{mm} \qquad \textbf{(I.12)}$$

$$y_{pix} = K_u \times y_{mm} \qquad \textbf{(I.13)}$$

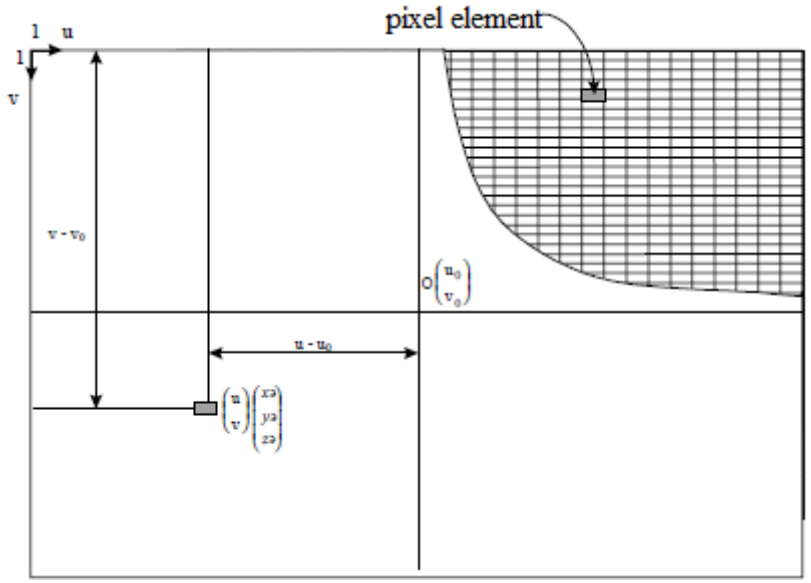


Figure I.16. Image system of coordinates

I.5. The circle Hough Transform

I.5.1. Introduction

A commonly faced problem in computer vision is determining the location, number or orientation of a particular object in an image. It could be for example to determine the straight roads on an aerial photo, this problem can be solved using Hough transform for lines. Often the objects of interest have other shapes than lines, it could be parables, circles or ellipses or any other arbitrary shape. The general Hough transform can be used on any kind of shape, although the complexity of the transformation increases with the number of parameters needed for describing the shape. Therefore, the Hough transform in general is only considered for simple shapes with parameters belonging to R^2 or at most R^3**.** In the following we will look at the Circular Hough Transform

I.5.2. Circle Hough Transform principal (CHT)

The equation of a circle is:

$$(\mathbf{x} - \boldsymbol{a})^2 + (\boldsymbol{y} - \boldsymbol{b})^2 = \boldsymbol{r}^2 \quad \textbf{(I.14)}$$

As it can be seen the circle got three parameters r, a, and b. Where a and b are the coordinates of the circle center, and r is the radius. The parametric representation of the circle is :

$$\begin{cases} \boldsymbol{x} = \boldsymbol{a} + \boldsymbol{r}\,\boldsymbol{cos}(\boldsymbol{\theta}) \\ \boldsymbol{y} = \boldsymbol{b} + \boldsymbol{r}\,\boldsymbol{sin}(\boldsymbol{\theta}) \end{cases} \quad \textbf{(I.15)}$$

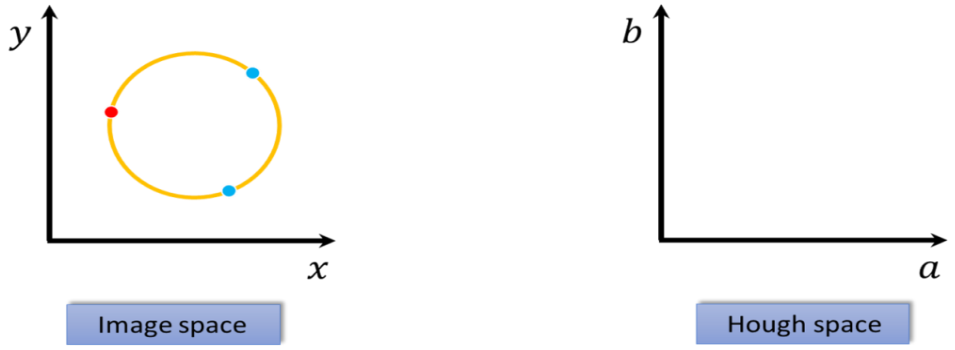


Figure I.17. Image and Hough spaces

Let take an example where we have three points $P_1(x_1, y_1)$, $P_2(x_2, y_2)$ and $P_3(x_3, y_3)$ that belong to a circle (Figure I.17) of center (a', b') and radius r. How can we prove they belong to a circle and determine the center and radius of that circle?

Suppose first that we know the radius r of the circle. According to equation (I.15) the center of this circle belongs to the circle in the parameter space (Hough space) which the coordinates of its center are (x_1, y_1) (Figure I.18).

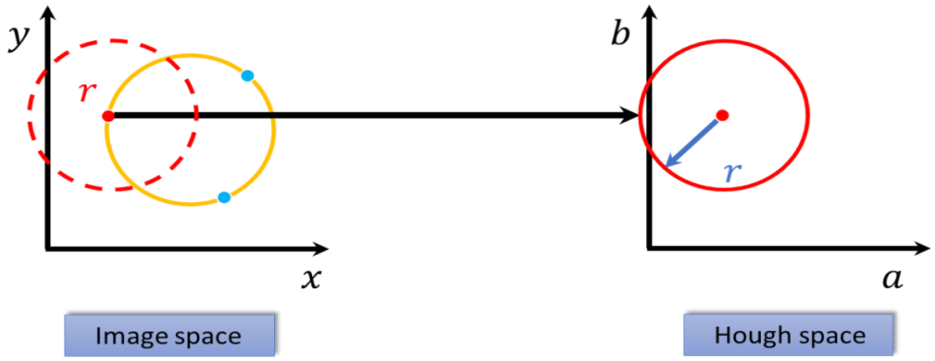


Figure I.18. A point in the image space is transformed to a circle in Hough space (r const.)

We do the same with points P2 and P3 (Figure I.19). The center of the circle to which belong P1, P2 and P3 is the common point of the three circles in the parameter space.

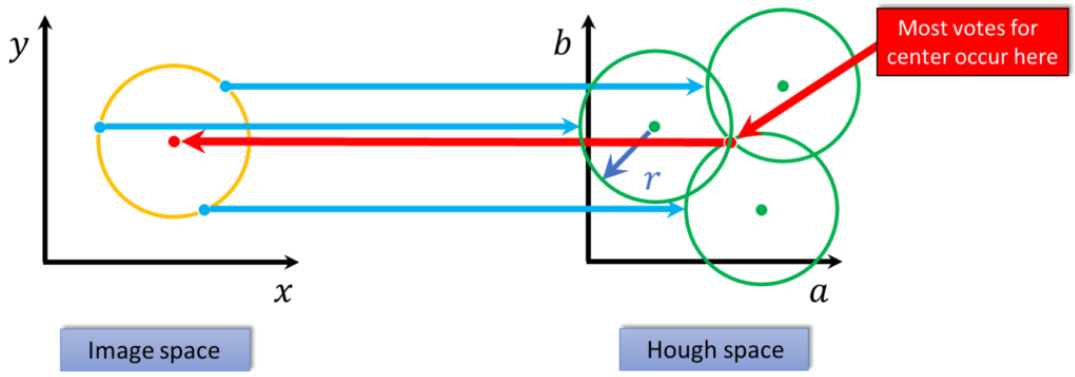


Figure I.19. The image space to Hough space transformation of three points

To find this common point (the circle's center) we will use an accumulator M. This accumulator is a matrix where it is saved how many times each point in the parameter space is drawn. So, $M(l, m)$ is the number of times the point with coordinates (l, m) in the parameter space is drawn. The local maxima value in the accumulator M indicates the center of a circle with radius r.

In fact, the circle radius length is unknown. Therefore, in CHT the procedure described above is repeated for different values of the radius (from r_{min} to r_{max} by defined step).

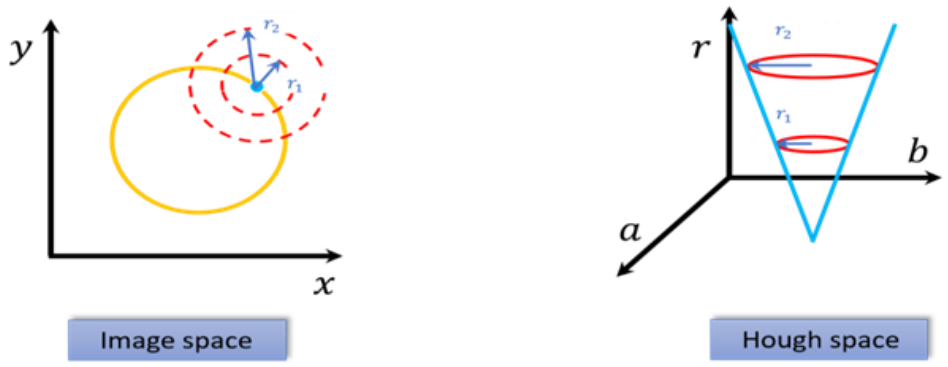


Figure I.20. A point in the image space is transformed to a cone in Hough space (r variable)

For an unknown radius r, known gradient direction

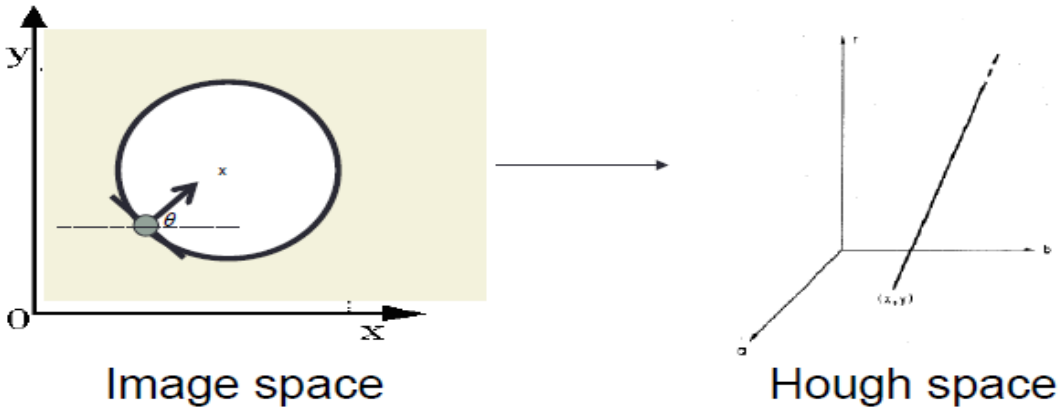


Figure I.20 bis. When the gradient direction is known, a point in the image space is transformed to a line in Hough space

I.5.3. Circle Hough Transform algorithm

For detecting circles in an image using CHT first we have to detect all the edges in the image. Any edge detection technique can be used. The algorithm of CHT is the following:

```
1. Quantize the parameter space
     M[a_min , …. , a_max , b_min , …. , b_max , r_min , …. , r_max].
2. For each edge pixel (x,y):
      For each possible radius value r.
        For each possible gradient direction Θ:
           %% or use estimated gradient
           a = x - r cos(θ)
           b = y - r sin(θ)
           M[a,b,r] = M[a,b,r] + 1.
        End
      End
3. Find the local maxima in the parameter space.
```

I.6. Visual servoing

I.6.1. Definitions

Visual servoing, also known as vision-based control, is a control in which the feedback information is an image. The visual servoing is used to control the motion of a robot (manipulator or mobile robot). Images are provided by a vision sensor (a camera generally) that can be fixed or embedded on the robotic system. The visual servoing can be used to perform positioning task of a robot in relation to its environment (Figure I.21), or a target tracking task (Figure I.22) .

Figure I.21. A target tracking example using a pan/tilt camera

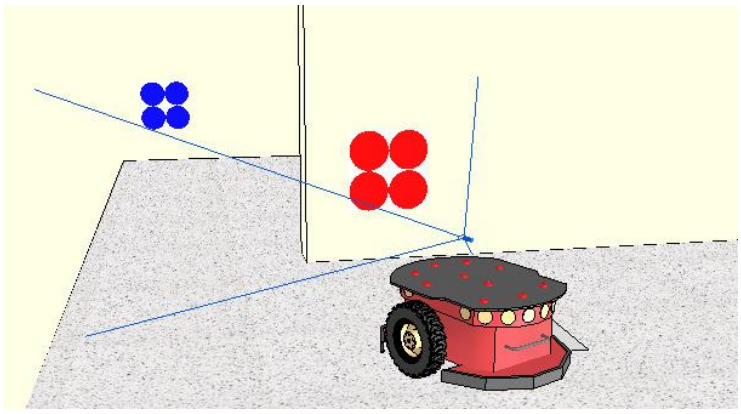


Figure I.22. Positioning task of a mobile robot with an onboard camera

Different servoing architecture classifications are offered in the literature. The architecture type is based on the nature of the error signal (task function). The error signal can be defined in either 3D workspace coordinates, or directly in terms of image features or combination of them . This leads us to three types of architectures:

- 2D visual servoing, also called “Image-based visual servoing” (IBVS)
- 3D visual servoing, also called “Position-based visual servoing” (PBVS)
- 2D ½ Visual Servoing also called « Hybrid Visual Servoing »

I.6.2. 2D Visual servoing

In our project we used the image-based visual servoing (IBVS), or 2D visual servoing architecture. That's why we will focus thereafter only on that type of visual servoing.

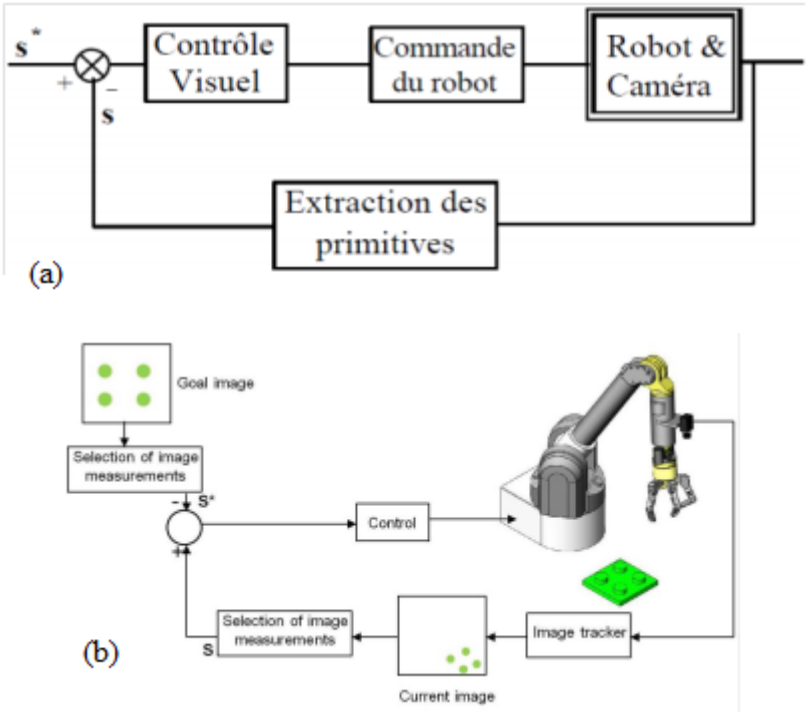


Figure I.23. -a- Image-based visual servoing control scheme
-b- Visual servoing scheme represented with an example

In this architecture (Figure I.23), the desired robot position is characterized by the image we get at this position (the goal image). Information "S" from the image provided by the camera at this instant is compared to the information in goal image S*. Information S and S* are extracted from the image by image processing. From error between S and S* the controller gives the robot's joints velocities. This process continue until the error S-S* becomes zero. Then, the robot reaches the goal position.

I.6.3. The control low

In visual servoing control schema we consider S, a set of visual information (example, coordinates (x,y) in the case of a point primitive). These visual information S depends of r(t), the relative position between the camera and the object of interest. r(t) is function of time when the camera or the object is moving. This is expressed mathematically by:

$$\boldsymbol{S = s\big(r(t)\big)} \quad \textbf{(I.16)}$$

So, the variation of the visual information is:

$$\boldsymbol{\dot{S} = \frac{ds}{dr}\,\dot{r}} \qquad \textbf{(I.17)}$$

$\boldsymbol{L_s = \frac{ds}{dr}}$ is the Jacobian image, also called the interaction matrix.

$\boldsymbol{\tau = \dot{r}}$ is the relative kinematic torsor between the camera and the object.

$$\boldsymbol{\tau = [V_x V_y V_z \omega_x \omega_y \omega_z]^t} \quad \textbf{(I.18)}$$

τ has 3 components of translation velocities, and 3 components of rotation velocities (the 6 degrees of freedom). So, we write the expression of the visual information variation as:

$$\boldsymbol{\dot{S} = L_s . \tau} \qquad \textbf{(I.19)}$$

We have "Task function" (the error in the servoing scheme):

$$\boldsymbol{e = (S(t) - S^*)} \qquad \textbf{(I.20)}$$

The variation of the task function $\boldsymbol{e}$:

$$\boldsymbol{\dot{e} = \dot{S}(t)} \qquad \textbf{(I.21)}$$

Because $\boldsymbol{\dot{S}^* = 0}$

We choose to have an error changing according to a decreasing exponential function. This allow to have an error that goes to zero after a while, and thus $\boldsymbol{S(t)}$ converges to $\boldsymbol{S^*}$

$$\boldsymbol{e(t) = e(0)e^{-\lambda.t}} \qquad \textbf{(I.22)}$$

The relation between the error and its variation is:

$$\boldsymbol{\dot{e} = -\lambda e} \qquad \textbf{(I.23)}$$

The convergence takes less time as $\boldsymbol{\lambda}$ is much bigger.

By substituting the equations (I.21) and (I.23) in the equation (I.19) we get :

$$\boldsymbol{\lambda e = -L_s \tau} \qquad \textbf{(I.24)}$$

The control low is therefore:

$$\tau = -\lambda L_S^+ e \qquad \textbf{(I.25)}$$

$\boldsymbol{L_s{}^+}$ is the pseudo-inverse of the interaction matrix $\boldsymbol{L_s}$

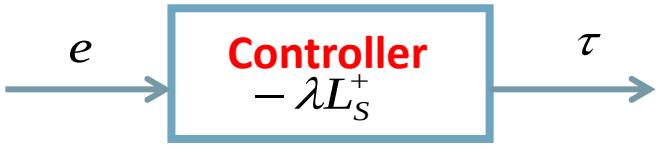


Figure I.24. The 2D visual servoing controller

I.6.4. Interaction matrix

We recall the relation between the coordinates of an object point $\boldsymbol{P}(\boldsymbol{X},\boldsymbol{Y})^{\boldsymbol{T}}$ and the coordinates of its image $\boldsymbol{p}(\boldsymbol{x},\boldsymbol{y})^{\boldsymbol{T}}$ with respect to the camera coordinate system:

$$\boldsymbol{p} = \frac{\boldsymbol{f}}{\boldsymbol{Z}}\boldsymbol{P} \quad \Leftrightarrow \quad \begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{y}\end{bmatrix} = \frac{\boldsymbol{f}}{\boldsymbol{Z}}\begin{bmatrix}\boldsymbol{X}\\ \boldsymbol{Y}\end{bmatrix} \qquad \textbf{(I.26)}$$

f is the focal length of the camera and Z the distance between the object point P and the camera

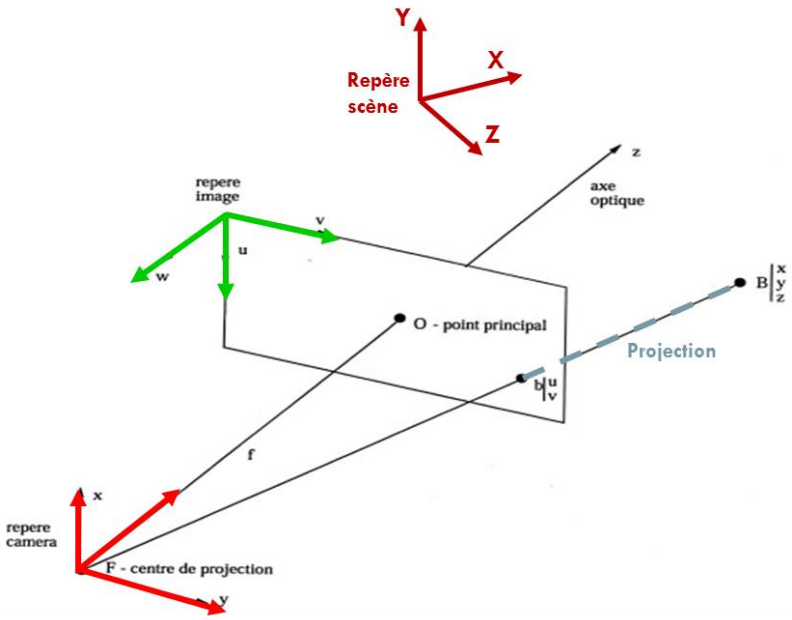


Figure I.25. Object point and its projection on the image plane.
Camera, World and Image system of coordinates

Either the camera is fixe or embedded, the object point P is in motion relative to the camera . According to the fundamental equation of kinematics (I.27) we have the relation between the variations of the object P coordinates and the kinematic torsor $[\boldsymbol{V} \quad \boldsymbol{\omega}]$ for 6 DOF :

$$\dot{\boldsymbol{P}} = -\vec{\boldsymbol{\omega}} \times \vec{\boldsymbol{P}} - \vec{\boldsymbol{V}} \qquad \textbf{(I.27)}$$

$$\begin{bmatrix}\dot{\boldsymbol{X}}\\ \dot{\boldsymbol{Y}}\\ \dot{\boldsymbol{Z}}\end{bmatrix} = -\begin{bmatrix}\boldsymbol{\omega_x}\\ \boldsymbol{\omega_y}\\ \boldsymbol{\omega_z}\end{bmatrix} \times \begin{bmatrix}\boldsymbol{X}\\ \boldsymbol{Y}\\ \boldsymbol{Z}\end{bmatrix} - \begin{bmatrix}\boldsymbol{V_x}\\ \boldsymbol{V_y}\\ \boldsymbol{V_z}\end{bmatrix} \qquad \textbf{(I.28)}$$

By substituting the equation (I.26) in the equation (I.28) we get:

$$\dot{\boldsymbol{x}} = -\boldsymbol{f}\cdot\boldsymbol{\omega_y} + \boldsymbol{y}\cdot\boldsymbol{\omega_z} - \frac{\boldsymbol{f}}{\boldsymbol{Z}}\cdot\boldsymbol{V_x} + \frac{\boldsymbol{x}.\boldsymbol{y}}{\boldsymbol{Z}}\boldsymbol{\omega_x} - \frac{\boldsymbol{x}^2}{\boldsymbol{f}}\boldsymbol{\omega_y} - \frac{\boldsymbol{x}}{\boldsymbol{Z}}\cdot\boldsymbol{V_z} \quad \textbf{(I.29)}$$

$$\dot{\boldsymbol{y}} = \boldsymbol{f}\cdot\boldsymbol{\omega_x} - \boldsymbol{x}\cdot\boldsymbol{\omega_z} - \frac{\boldsymbol{f}}{\boldsymbol{Z}}\boldsymbol{V_y} + \frac{\boldsymbol{y}^2}{\boldsymbol{Z}}\boldsymbol{\omega_x} - \frac{\boldsymbol{y}\cdot\boldsymbol{x}}{\boldsymbol{f}}\boldsymbol{\omega_y} + \frac{\boldsymbol{y}}{\boldsymbol{Z}}\cdot\boldsymbol{V_z} \qquad \textbf{(I.30)}$$

In matrix writing we have:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\end{bmatrix}=\begin{bmatrix}\frac{-f}{z} & 0 & \frac{x}{z} & \frac{xy}{f} & \frac{-f^2-x^2}{f} & y\\ 0 & -\frac{f}{z} & \frac{y}{z} & \frac{f^2+y^2}{f} & -\frac{xy}{f} & -x\end{bmatrix}\begin{bmatrix}V_x\\ V_y\\ V_z\\ \omega_x\\ \omega_y\\ \omega_z\end{bmatrix} \qquad \textbf{(I.31)}$$

From equations (I.19) and (I.31) we deduce the expression of the interaction matrix L_s :

$$L_s=\begin{bmatrix}\frac{-f}{z} & 0 & \frac{x}{z} & \frac{xy}{f} & \frac{-f^2-x^2}{f} & y\\ 0 & -\frac{f}{z} & \frac{y}{z} & \frac{f^2+y^2}{f} & -\frac{xy}{f} & -x\end{bmatrix} \qquad \textbf{(I.32)}$$

We can see in equation (I.32) that interaction matrix depends on the camera's focal length, the image point coordinates, and the depth Z.

If the robotic system used in 2D visual servoing has less DOF than 6 we can reduce the interaction matrix (I.32) by omitting some colons, keeping just the colons relative to the DOF of the robot.

In practice we take a set of points as primitives, generally four points. In that case the interaction matrix will be:

$$L_s=[L_1 \quad L_2 \quad L_3 \quad L_4]^t \qquad \textbf{(I.33)}$$

$$L_s=\begin{bmatrix}L_1\\ L_2\\ L_3\\ L_4\end{bmatrix}=\begin{bmatrix}\frac{-f}{z} & 0 & \frac{x_1}{z} & \frac{x_1y_1}{f} & \frac{-f^2-x_1^2}{f} & y_1\\ 0 & -\frac{f}{z} & \frac{y_1}{z} & \frac{f^2+y_1^2}{f} & -\frac{x_1y_1}{f} & -x_1\\ \frac{-f}{z} & 0 & \frac{x_2}{z} & \frac{x_2y_2}{f} & \frac{-f^2-x_2^2}{f} & y_2\\ 0 & -\frac{f}{z} & \frac{y_2}{z} & \frac{f^2+y_2^2}{f} & -\frac{x_2y_2}{f} & -x_2\\ \frac{-f}{z} & 0 & \frac{x_3}{z} & \frac{x_3y_3}{f} & \frac{-f^2-x_3^2}{f} & y_3\\ 0 & -\frac{f}{z} & \frac{y_3}{z} & \frac{f^2+y_3^2}{f} & -\frac{x_3y_3}{f} & -x_3\\ \frac{-f}{z} & 0 & \frac{x_4}{z} & \frac{x_4y_4}{f} & \frac{-f^2-4^2}{f} & y_4\\ 0 & -\frac{f}{z} & \frac{y_4}{z} & \frac{f^2+y_4^2}{f} & -\frac{x_4y_4}{f} & -x_4\end{bmatrix} \qquad \textbf{(I.34)}$$

We see that the interaction matrix is calculated with image points coordinates with respect to camera coordinates system.

The relation between $(\boldsymbol{x}, \boldsymbol{y})$ and $(\boldsymbol{u}, \boldsymbol{v})$, respectively the image point coordinates in respect to the camera and image coordinate systems are given by equations (I.35) and (I.36) (see figure I.26 for the demonstration):

$$\boldsymbol{x} = \frac{\boldsymbol{H}}{\boldsymbol{2}} - \boldsymbol{u} \qquad \textbf{(I.35)}$$

$$\boldsymbol{y} = \boldsymbol{v} - \frac{\boldsymbol{W}}{\boldsymbol{2}} \qquad \textbf{(I.36)}$$

H : Height of the image

W : Width of the image

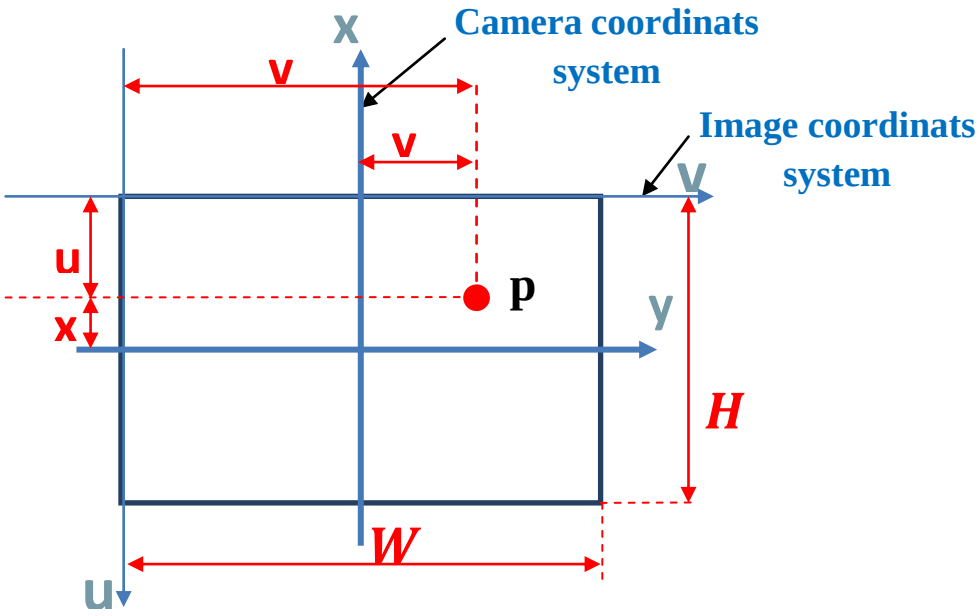


Figure I.26. Coordinates of an image point p in respect of camera and image systems of coordinates

When the focal length of a camera isn't given we can deduce it from other camera parameters (see equation (I.37)). As in our case, the camera parameters in the simulation tool CoppeliaSim was given only by the resolution and the angle of vision of the camera.

$$\tan\left(\frac{\alpha}{2}\right) = \frac{W}{2f} \qquad \text{(I.37)}$$

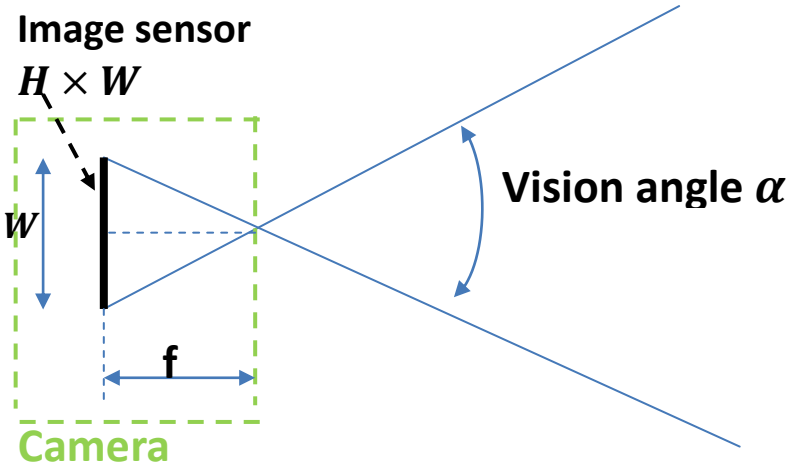


Figure I.27. The focal length,
the angle of vision, and the image resolution

I.6.5. The depth calculation

A primitive is an elementary geometric shape (point, right segment, portion of ellipse, etc.). In the majority of cases, primitive dots are used.

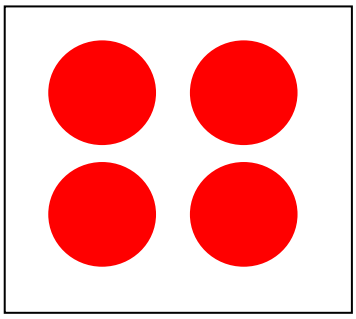


Figure I.28. The landmark used in our project

The landmark we used has four circles. The centers of these circles are placed on vertices of a square. The length of the square sides must be known, the primitive points are the centers of these circles, which can be detected by the circle Hough transform processing we explained in the previous section (section 4 in the same chapter).

Z is the depth of the landmark relative to the camera .There is a relation that can help to estimate Z from information from the image and the real world**.**

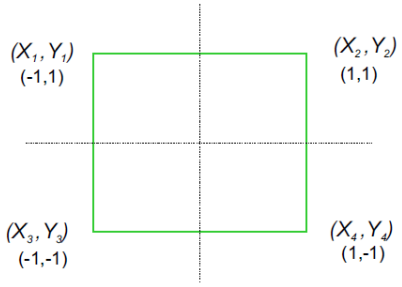


Figure I.29. Coordinates of real circles centers

Let be $(\boldsymbol{u_1}, \boldsymbol{v_1})$, $(\boldsymbol{u_2}, \boldsymbol{v_2})$, $(\boldsymbol{u_3}, \boldsymbol{v_3})$, and $(\boldsymbol{u_4}, \boldsymbol{v_4})$ the coordinates of the four circles centers in the image. And $(\boldsymbol{X_1}, \boldsymbol{Y_1})$, $(\boldsymbol{X_2}, \boldsymbol{Y_2})$, $(\boldsymbol{X_3}, \boldsymbol{Y_3})$, and $(\boldsymbol{X_4}, \boldsymbol{Y_4})$ the coordinates of the four circles centers in the real world.

The coordinates v1 and v3 can be obtained by the perspective projection relationship:

$$\boldsymbol{v_1 = f\frac{Y_1}{Z}} \quad \boldsymbol{and\ v_3 = f\frac{Y_3}{Z}} \qquad \textbf{(I.38)}$$

Therefore we find the relation between the image and the real square length sides:

$$\boldsymbol{v_1 - v_3 = f\frac{(Y_1 - Y_3)}{Z}} \qquad \textbf{(I.39)}$$

For more accuracy we can take the 4 points (not only two):

$$\boldsymbol{Z = \frac{f}{2}\left(\frac{(Y_1 - Y_3)}{v_1 - v_3} + \frac{(Y_2 - Y_4)}{v_2 - v_4}\right)} \qquad \textbf{(I.40)}$$

$$\boldsymbol{Z = \frac{f}{2}\left(\frac{\Delta Y_l}{\Delta v_l} + \frac{\Delta Y_r}{\Delta v_r}\right)} \qquad \textbf{(I.41)}$$

$\boldsymbol{\Delta Y_l}$ and $\boldsymbol{\Delta v_l}$ are respectively the length of the left side of the real and the image square

$\boldsymbol{\Delta Y_r}$ and $\boldsymbol{\Delta v_r}$ are respectively the length of the right side of the real and the image square

I.7. Conclusion

We have presented in this chapter the main parts of the servoing system we are studying; the mobile robot, the camera, the controller, and the image primitives extraction part. Also, We have given the mathematical equations of each part.

Chapter II

Simulation and programming tools

II.1. Introduction

To simulate the robot behavior we had to use different software tools. The main one is CoppeliaSim simulator which allowed us to build the virtual environment containing a mobile robot, an onboard camera and a landmark, and allowed us to execute our C++ program.

In the program we used tow libraries, OpenCV for detecting circles in the images captured, and Armadillo library for matrix calculation.

In this chapter we give an overview on the libraries openCV and Armadillo and their use in our program, and the simulator coppeliaSim and some of its programming functions.

II.2. Circle detection with OpenCV library

OpenCV (Open source computer vision) is a library of programming functions mainly aimed at real-time computer vision. Originally developed by Intel (officially launched in 1999, it was later supported by Willow Garage then Itseez (which was later acquired by Inte). The library is cross-platform and free for use under the open-source BSD license.

One of OpenCV's goals is to provide a simple-to-use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly. The OpenCV library contains over 500 functions that span many areas in vision, including factory product inspection, medical imaging, security, user interface, camera calibration, stereo vision, and robotics. Because computer vision and machine learning often go hand-in-hand, OpenCV also contains a full, general-purpose Machine Learning library (ML module) .

Hough circle transform function `cv::HoughCircles()`

In openCV, circles in an image can be detected by the function **`cv::HoughCircles()`** that uses the Hough circle transform algorithm. The synopsis of this function is:

```
void cv::HoughCircles(
cv::InputArray image,     // Input single channel image
cv::OutputArray circles, // N-by-1 3-channel or vector of Vec3f
int method,               // Always cv::HOUGH_GRADIENT
double dp,                // Accumulator resolution (ratio)
double minDist,           // Required separation (between lines)
double param1 = 100,      // Upper Canny threshold
double param2 = 100,      // Unnormalized accumulator threshold
int minRadius = 0,        // Smallest radius to consider
int maxRadius = 0 );      // Largest radius to consider
```

The input **`image`** is an 8-bit image (grayscale image).

The result array, **`circles`**, will be either a matrix-array or a vector, depending on what you pass to the function. If a matrix is used, it will be a one-dimensional array of type CV::F32C3; the three channels will be used to encode the location of the circle and its radius. If a vector is used, it must be of type std::vector<Vec3f>.

The **`method`** argument must always be set to cv::HOUGH_GRADIENT.

The parameter **`dp`** is the resolution of the accumulator image used. If dp is set to 1, then the resolutions will be the same; if set to a larger number (e.g., 2), then the accumulator resolution will be smaller by that factor (in this case, half).

The parameter **`minDist`** is the minimum distance that must exist between two circles in order for the algorithm to consider them distinct circles.

The two arguments **`param1`** and **`param2`**, are the edge (Canny) threshold and the accumulator threshold, respectively. The threshold value is the value in the accumulator plane that must be reached for the algorithm to report a circle**.**

Recall that the Canny edge detector takes two different thresholds itself. When cv::Canny() is called internally, the first (higher) threshold is set to the value of **`param1`** passed into **`cv::HoughCircles()`**, and the second (lower) threshold is set to exactly half that value.

The parameter **`param2`** is the one used to threshold the accumulator.

The final two parameters are the minimum and maximum radius of circles that can be found.

II.3. CoppeliaSim

Figure II.1. CoppeliaSim logo

II.3.1. Introduction

CoppeliaSim is a powerful cross-platform robot simulator which has a free educational version. CoppeliaSim evolved from V-REP, which was discontinued in late November, 2019. CoppeliaSim's strength comes from several features:

- CoppeliaSim provides a framework which includes dynamic simulation engines, forward/inverse kinematics tools, collision detection libraries, vision sensor simulations, path planning, GUI development tools, and built-in models of many common robots.
- You can embed Lua scripts directly into a simulation scene for processing simulated sensor data, or running control algorithms. A remote API allows one to develop standalone applications in many programming languages (C/C++, Python, Java, Lua, Matlab) that are able to pass data in and out of a running CoppeliaSim simulation .

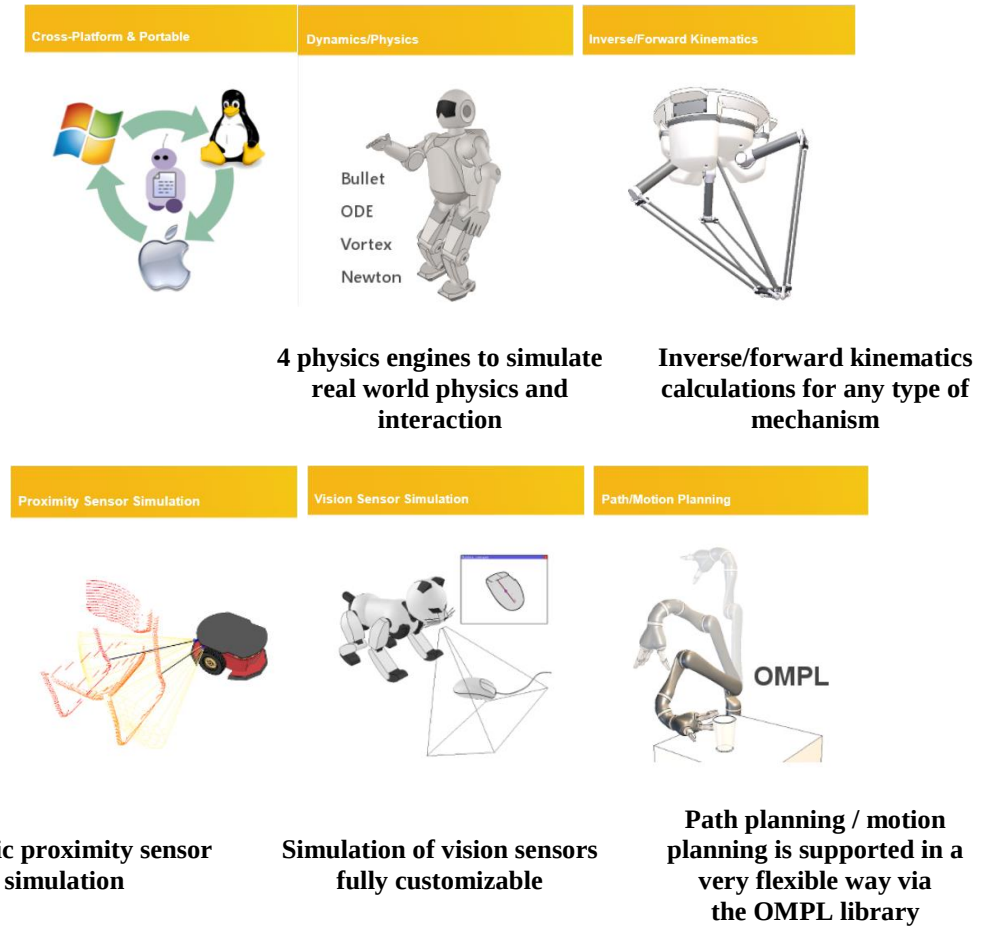


Figure II.2. Ilustration of some of CoppeliaSim characteristics

II.3.2. CoppeliaSim graphical user interface

The figure II.3 illustrates a typical view of the CoppeliaSim application.

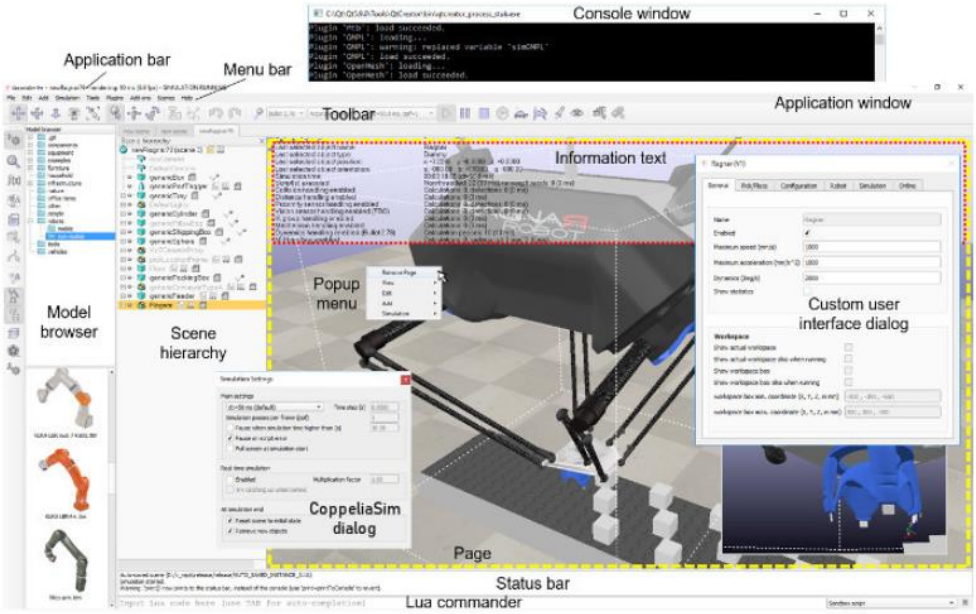


Figure II.3. CoppeliaSim user interface

The principal windows elements of CoppeliaSim user interface are:
- Toolbars
- Model browser
- Scene hierarchy
- Scene view

The model browser (figure II.4) displays in its upper part a CoppeliaSim model folder structure, and in its lower part, thumbnails of models contained in the selected folder. Thumbnails can be dragged-and-dropped into the scene to automatically load the related model.

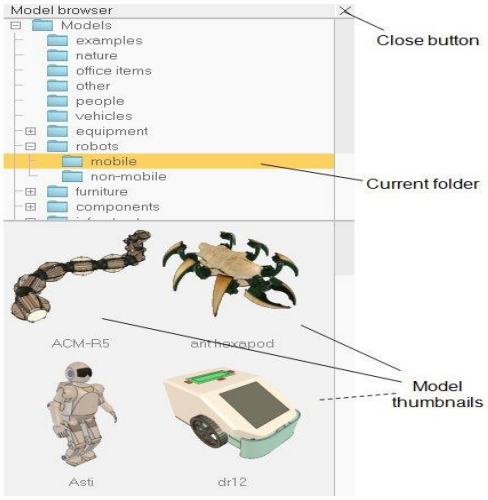


Figure II.4. Model browser

When creating a new simulation, the default scene will contain:

- Several camera objects
- Several light objects used to illuminate the scene.
- Several views: a view is associated with a camera and displays what the camera sees.
- The floor.
- The default main script: allow running minimal simulations, without the need of child scripts.

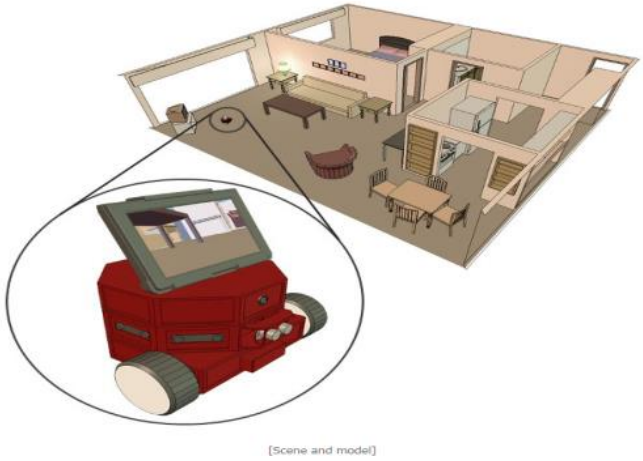


Figure II.5. Example of CoppeliaSim scene

The scene hierarchy (figure II.6) displays the content of a scene. Since scene objects are built in a hierarchy-like structure, the scene hierarchy displays a tree of this hierarchy. Objects in the scene hierarchy can be dragged and dropped onto another object, in order to create a parent-child relationship

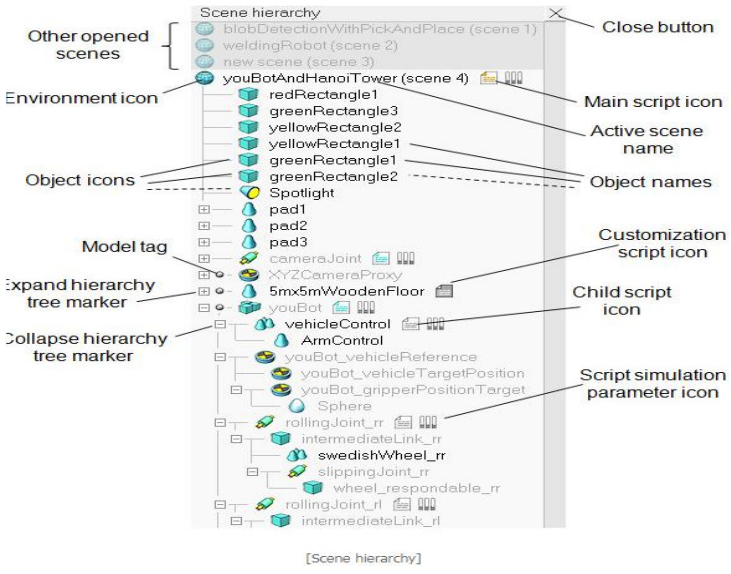
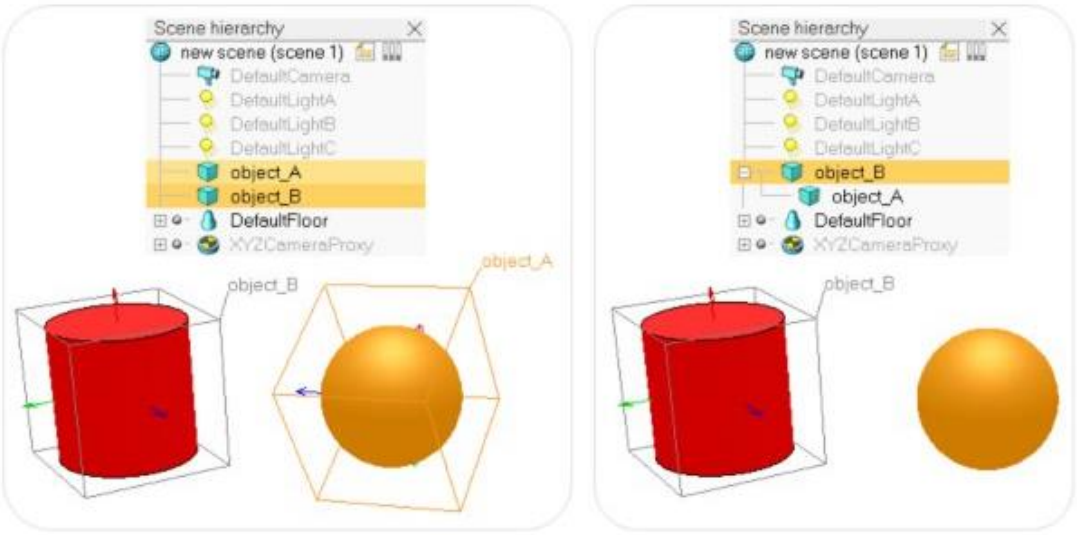


Figure II.6. Scene hierarchy

If object A in the scene hierarchy is built on top of object B (figure II.7), object B is the parent and object A is the child. So when object B moves, object A will automatically follow, since object A is attached to object B.

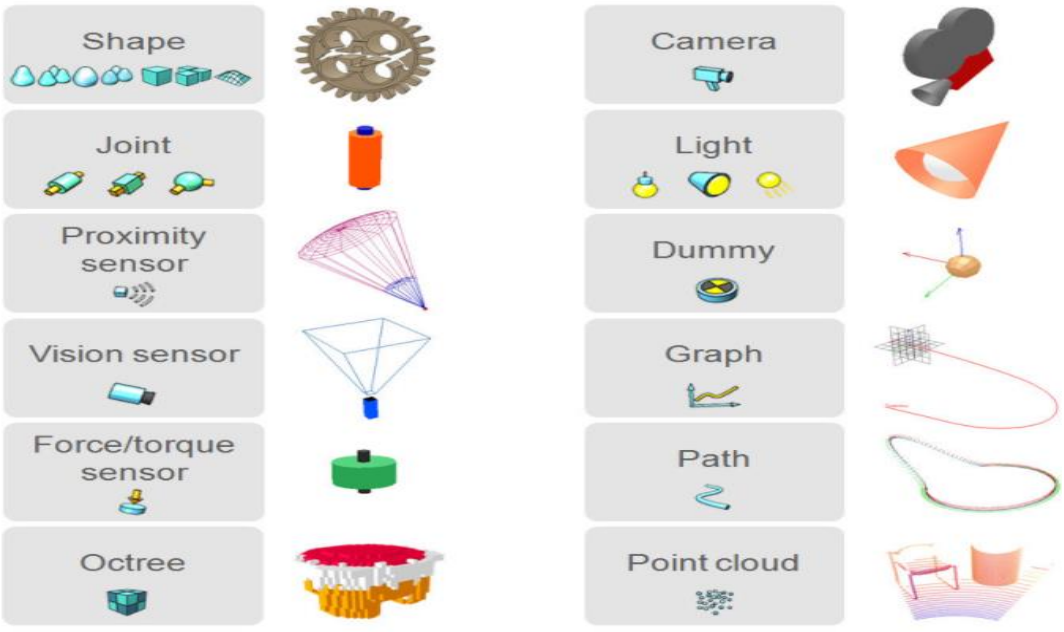


[(1) Before attaching object A to object B, (2) after attaching object A to object B]

Figure II.7. Child/Parent relationship between objects

II.3.3. Scene objects

Scene objects (figure II.8) are the elements that are used for building a simulation scene. Objects are visible in the scene hierarchy and in the scene view.



[Object types in CoppeliaSim and their three dimensional representation]

Figure II.8. Scene objects

The figure II.9 displays the 5 primitive shapes (plane, disc, cuboid, sphere and cylinder).

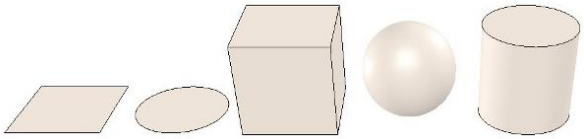


Figure II.9. Primitive shapes

A joint is an object that has at least one intrinsic Degree of Freedom (DoF). Joints are used to build mechanisms and to move objects, Joint (or actuators) are four types supported: revolute, prismatic, screws and spherical.

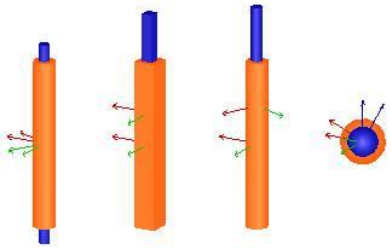


Figure II.10. Revolute, prismatic, screw and spherical joints

Vision sensors come in 2 different types and can be adjusted for different purposes:

- Orthographic projection-type
- Perspective projection-type

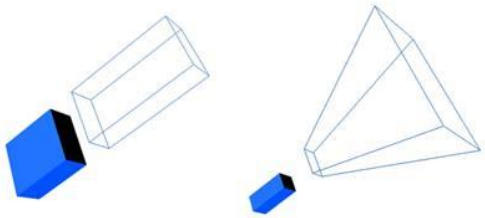


Figure II.11. Vision sensors, orthogonal projection-type and perspective projection-type

The dialog box in figure II.12 displays the settings and parameters of the vision sensor. Among these parameters we have the perspective angle and the image resolution.

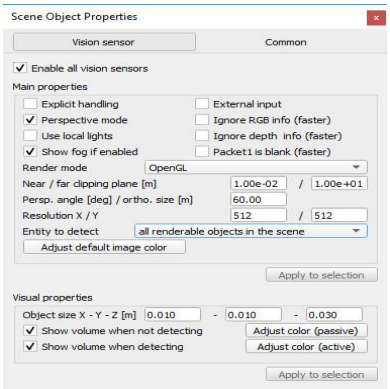


Figure II.12. Vision sensor settings

II.3.4. Programing with CoppeliaSim

The remote API is part of the CoppeliaSim API framework. It allows communication between CoppeliaSim and an external application. It supports C/C++, Java, Python, Matlab, Octave and Lua languages.

To use the remote API functionality in your C/C++ application, you should include the following files (located in CoppeliaSim's installation directory) in your project:

- extApi.h
- extApi.c
- extApiPlatform.h
- extApiPlatform.c

And in your IDE configuration, set as a preprocessor definition:

- **NON_MATLAB_PARSING**
- **MAX_EXT_API_CONNECTIONS=255**
- **DO_NOT_USE_SHARED_MEMORY)**

In our project we used C/C++ remote API for writing the simulation code because we are using the C++ OpenCV library in our program. The code was written in Microsoft Visual Studio IDE. CoppeliaSim provides more than 100 remote API functions. We present here some of them, the ones we used in our C++ code.

Synopsis	simxStart(connectionAddress, connectionPort, waitUntilConnected, doNotReconnectOnceDisconnected, timeOutInMs, commThreadCycleInMs)
Description	Starts a communication thread with the server (i.e. CoppeliaSim). This should be the very first remote API function called on the client side. This function returns the client ID, or -1 if the connection to the server was not possible.
Parameters	• *connectionAddress*: the ip address where the server is located (i.e. CoppeliaSim) • *connectionPort*: the port number where to connect. Specify a negative port number in order to use shared memory, instead of socket communication. • *waitUntilConnected*: if ≠0 the function blocks until connected (or timed out). • *doNotReconnectOnceDisconnected*: if ≠0 the communication thread will not attempt a second connection if a connection was lost. • *timeOutInMs* • *commThreadCycleInMs*

Synopsis	simxStartSimulation(clientID, operationMode)
Description	Requests a start of a simulation (or a resume of a paused simulation).
	• *clientID*: the client ID. Refer to simxStart • *operationMode*: Recommended operation mode is simx_opmode_oneshot.

Synopsis	simxGetObjectHandle(clientID, objectName, handle, operationMode)
Description	In CoppeliaSim you have to reference the objects you are working with.
Parameters	• *clientID*: the client ID. Refer to simxStart • *objectName*: name of the object. • *handle*: pointer to a value that will receive the handle • *operationMode*: a remote API function operation mode.

Synopsis	simxGetVisionSensorImage(clientID, sensorHandle, resolution, image, options, operationMode)
Description	Used to get image from vision sensor.
Parameters	• *clientID*: the client ID. Refer to simxStart • *sensorHandle*: handle of the vision sensor • *resolution*: pointer to 2 values receiving the resolution of the image • *image*: pointer to a pointer to the image data. • *options*: image options. Each image pixel is a byte (greyscale image), or rgb byte-triplet • *operationMode*: a remote API function operation mode

Synopsis	simxSetJointTargetVelocity(clientID, jointHandle ,targetVelocity ,operationMode)
Description	Used to send velocity to Joint.
Parameters	• *clientID*: the client ID. Refer to simxStart. • *jointHandle*: handle of the joint • *targetVelocity*: target velocity of the joint (linear or angular velocity depending on the joint-type) • *operationMode*: a remote API function operation mode

II.4. Armadillo library

4.1. Definition

Armadillo is a linear algebra library for the C++ language. It provides high-level syntax and functionality similar to Matlab aiming towards a good balance between speed and ease of use. The development of the Armadillo library is done by Conrad Sanderson and Ryan Curtin.

Figure II.13. Armadillo C++ library logo

In this project, we used this library in our C++ simulation program for calculating the command array of equation (I.25) which needs matrix computations (matrix product and pseudo-invers matrix).

4.2. Example of programs with Armadillo library

We can find the complete documentation of the Armadillo library in its website []. We give here some programing examples on only the operations we needed in our program (matrix declaration and affectation, matrix product, pseudo-invers matrix function)

The mat class is used to create a 2D matrix, with the size specified via template arguments.

Example 1 : Filling a matrix

```
mat A(2 ,2);          //Empty matrix with 2 rows and 2 colons
A<<1<<0<<endr<<0<<1; //The elements of the 1st raw: 1 and 0. The 2nd
raw: 0 and 1
cout<<"A="<<A;
```

The result is:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Example 2: Matrix product

```
mat A(2 ,2); A<<1<<2 endr<<2<<1;          // matrix A=[1 2; 2 1]
mat B(2 ,3); B<<1<<2<<3<<endr<<3<<2<<1;  // matrix B=[1 2 3; 3 2 1]
mat C(2 ,3);
C=A*B;            //Product of matrices A and B
cout<<"C="<<C;    //Display the matrix C
```

The result is:

$$C = \begin{bmatrix} 8 & 6 & 8 \\ 5 & 6 & 7 \end{bmatrix}$$

Example 3: Calculating B^+ the pseudo invers of matrix B

The manual calculation of B^+ gives:

$$B^+ = B^t[BB^t]^{-1}$$

$$B^+ = 2\begin{bmatrix} 1 & 3 \\ 2 & 2 \\ 3 & 1 \end{bmatrix}\begin{bmatrix} 7 & 5 \\ 5 & 7 \end{bmatrix}^{-1}$$

$$B = \frac{1}{3}\begin{bmatrix} -2 & 1 & 4 \\ 4 & 1 & -2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{6} & \frac{1}{3} & \frac{1}{12} \\ \frac{1}{12} & \frac{1}{3} & -\frac{1}{6} \end{bmatrix}$$

With Armadillo library we use the function **pinv()** to calculate de pseudo inverse of a matrix B.

```
mat B(2 ,3); B<<1<<2<<3<<endr<<3<<2<<1;  // matrix B=[1 2 3; 3 2 1]
mat V = pinv( B );  //Matrix V is the pseudo invers of B.
                    //Matrix V has automatically the same size as B
cout<<" V ="<< V;   //Display matrix V
```

The result is double type :

$$B^+ = \begin{bmatrix} -1,66 & 0.33 & 0.083 \\ 0.083 & 0.33 & -1,66 \end{bmatrix} = \begin{bmatrix} -\frac{1}{6} & \frac{1}{3} & \frac{1}{12} \\ \frac{1}{12} & \frac{1}{3} & -\frac{1}{6} \end{bmatrix}$$

II.5. Conclusion

In this chapter, we have given an overview of two libraries in c ++, OpenCV for computer vision and Armadillo for matrix computing. We also gave an overview of the CoppeliaSim simulator in its two parts: construction of the virtual environment and programming of the robot's behavior.

Chapter III

Simulation and results

III.1. Introduction

We present in this chapter a simulation with CoppeliaSim of positioning a Pioneer unicycle mobile robot in a simple environment (a room), without obstacles in his close neighborhood. The Pioneer robot is equipped with a camera on his top. A landmark is stuck to one wall at the height of the camera. The positioning task is achieved by 2D visual servoing.

We will see in this chapter the construction of the environment, the simulation flowchart, the simulation C++ code, and finally the results

III.2. Construction of the robot environment

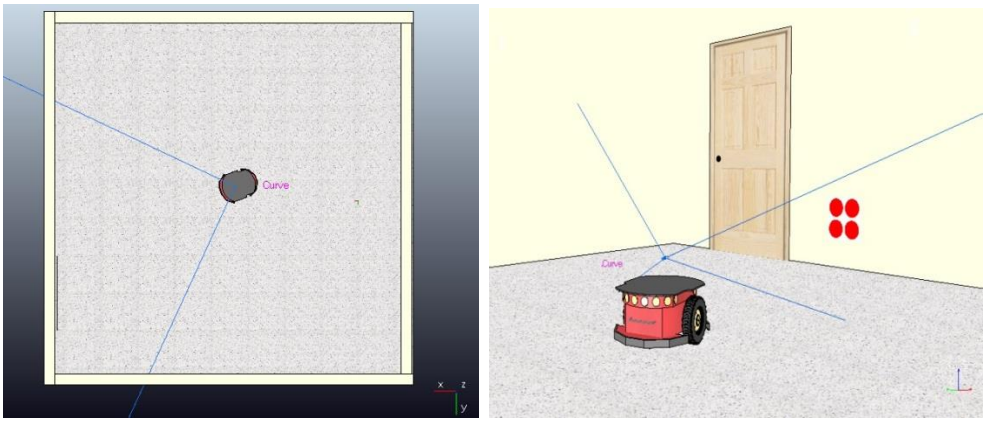


Figure III.1. Photos of the virtual 3D environment constructed
(left: Top view, right: Side view)

The virtual environment we built (figure III.1) consists of an empty room with an area of 5m×5m with one door. We placed in this environment the model of the "Pioneer" mobile robot existing in CoppeliaSim. We placed a landmark (figure III.1) that we made with simple plane chaps.

The walls: 4 walls made of textured cuboid chaps of 2.50 m high and 15cm Thickness
The robot: The robot we chose is Pioneer. It is a collection of two wheels fixed to chasse with two revolute joints. We don't need to build it because it is available in Coppelia model's list. Just we get it from the model browser than place it in the scene.

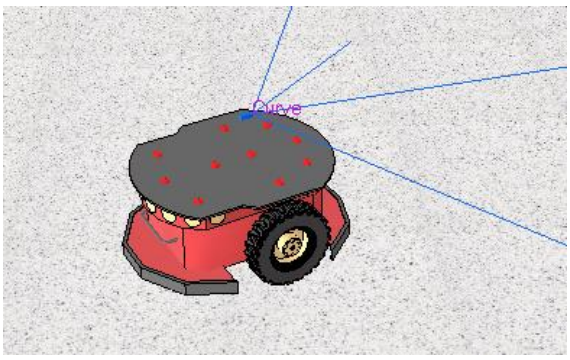


Figure III.2. Pioneer model with a camera placed on top

From the geometric properties of the robot Pioneer model we get two information that we need in the program to calculate the angular velocity of the left and right wheels, the radius of the wheels and the distance between the two wheels. We find from the coordinates of the left and right wheels:

- Wheel radius: `0.0975 m`
- Distance between wheels: `3.300e-01 m`

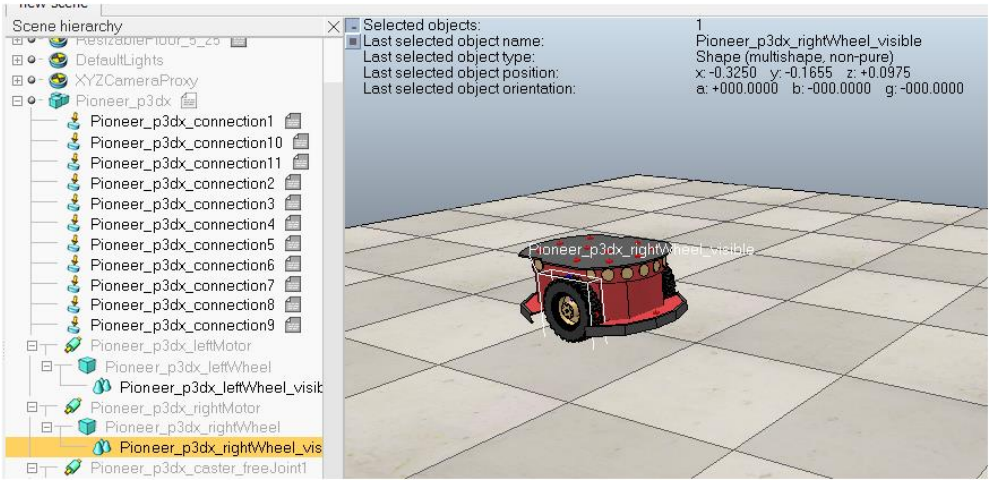


Figure III.3. Pioneer right wheel coordinates

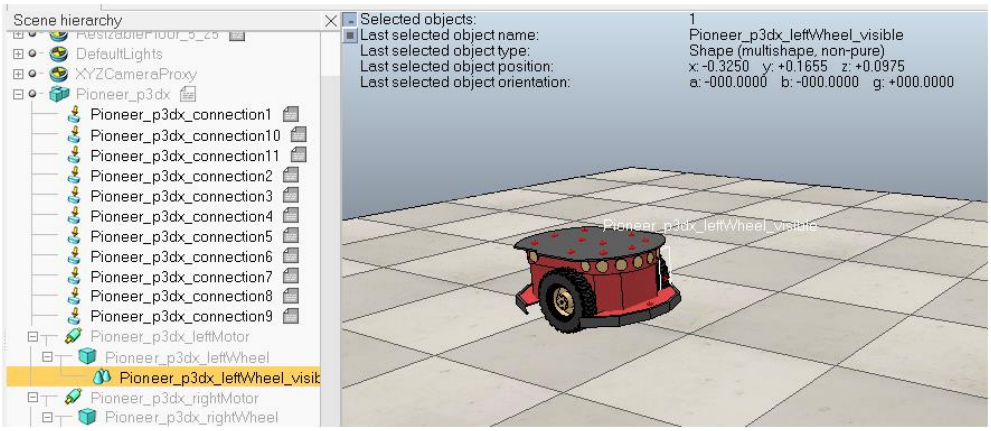


Figure III.4. Pioneer left wheel coordinates

The camera: We add perspective vision sensor object. To embed the camera sensor on the robot, in the scene hierarchy pic the vision sensor object and add it to the robot as a child. Thus the camera moves when camera moves. The camera is placed +3.4000e-01m from floor (at the height of the landmark). We set the camera properties in the scene object properties window.

Resolution of the image sensor: 1024x512
Angle of vision: 46°

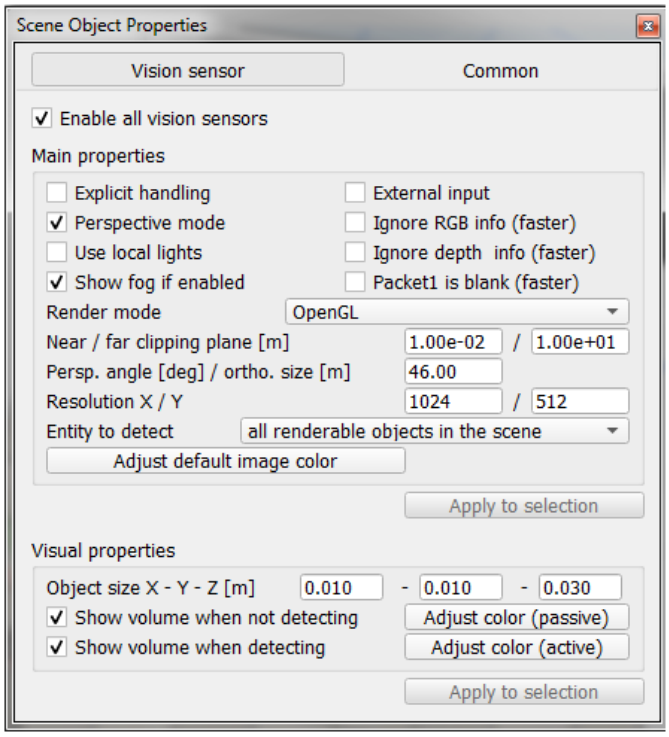


Figure III.5. vision sensor parameters

The landmark we made it with plane chaps (4 discs and a plane) that we group together. The dimensions of the landmark are shown in the figure III.6. The landmark is placed on the wall at 3.4000e-01m Height from the floor.

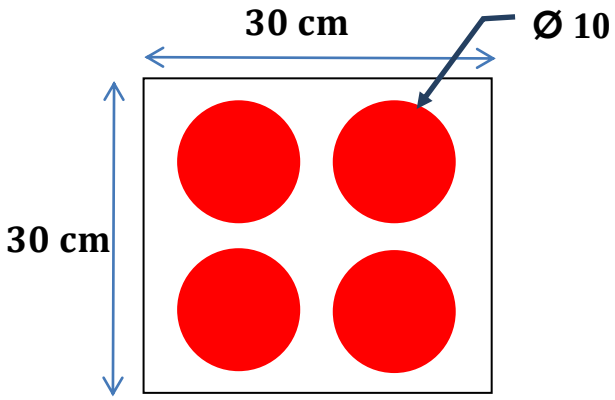


Figure III.6. Landmark dimensions

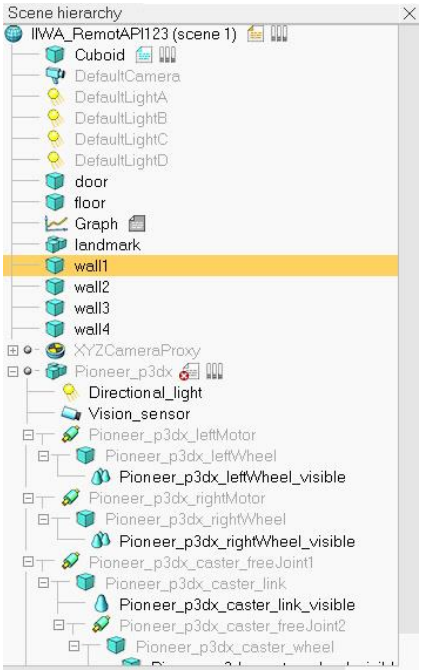


Figure III.7. The scene hierarchy

We add the object "Graph" for drawing the path of the robot in 3D. The child script of the robot must be disabled to avoid running the default code attached to it.

III.3. The Simulation flowchart

First of all, we take the robot to the desired position and we capture an image with the embedded camera from this position. This image is our goal image. We store its image file in the storage disk.

The task of the program is to take the robot from its actual position to the desired position.

We load the goal image from the disk and process it with the CHT (circle Hough transform) to detect the coordinates of the four primitives. It is the desired features coordinates.

Then, a new image is captured and processed with the Circle Hough Transform to determine the features coordinates. These coordinates are compared to those of the image goal. If the difference (the error) is yet higher than a limit e_lim we calculate and apply a new command to the right and left motors wheels (equations 25, 6 and 7). Else, the program is stopped.

The unicycle mobile robot has 3 DOF, 2 translations along Y and Z axes (According to the camera system of coordinates), and one rotation around the X axis. The interaction matrix expression in equation (32) is reduced to 3 colons.

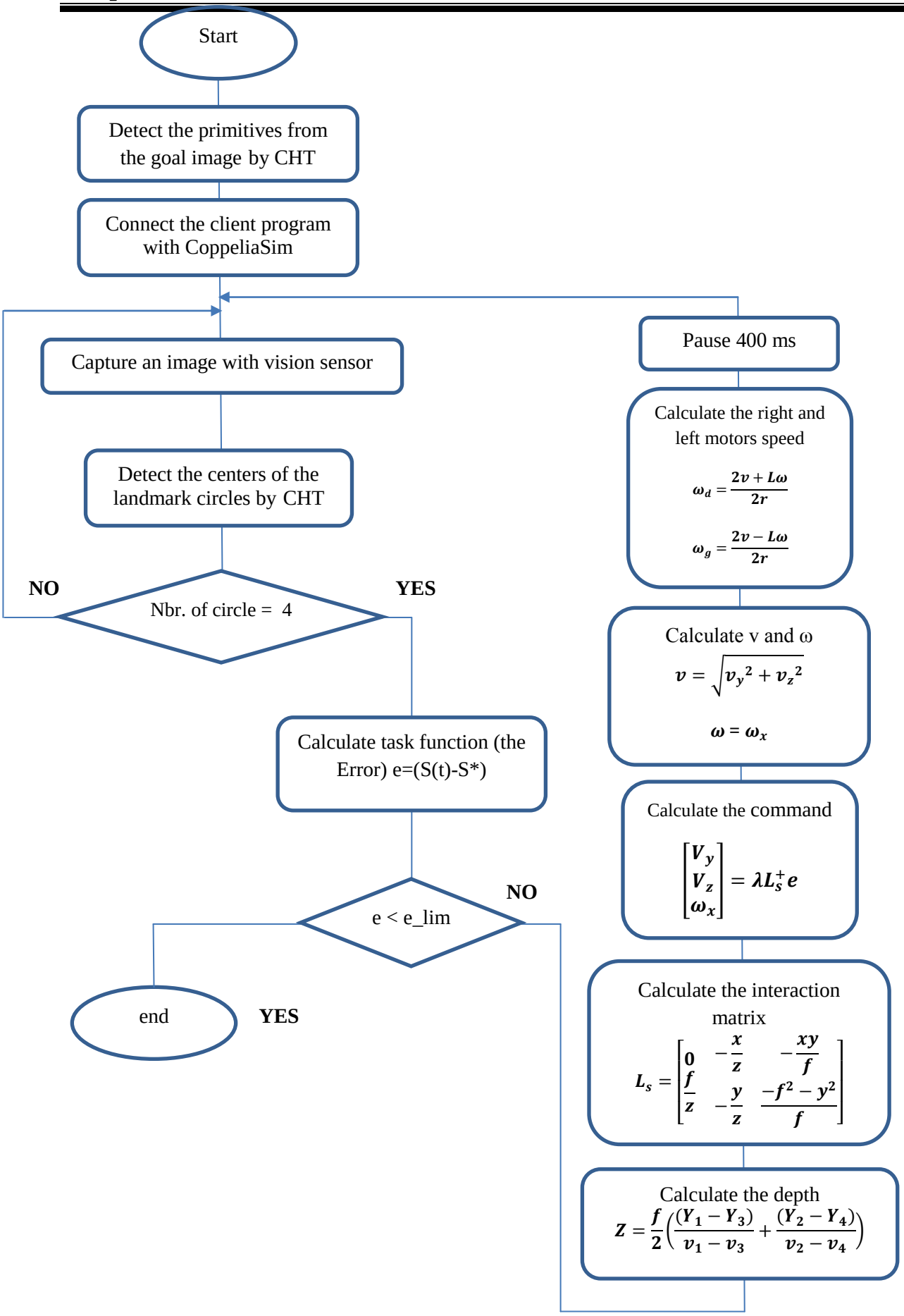


Figure III.8 Simulation flowchart

III.4. Results

At the desired position (figure III.9) the robot captures an image (the goal image figure III.10). We put on the floor a mark (yellow) indicating the desired pose (position and orientation).

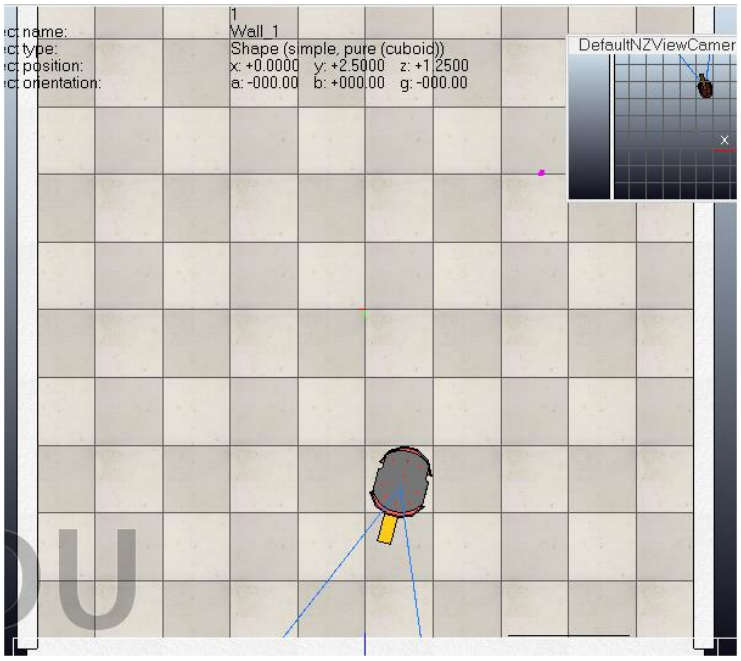


Figure III.9. Robot at the desired position (top view)

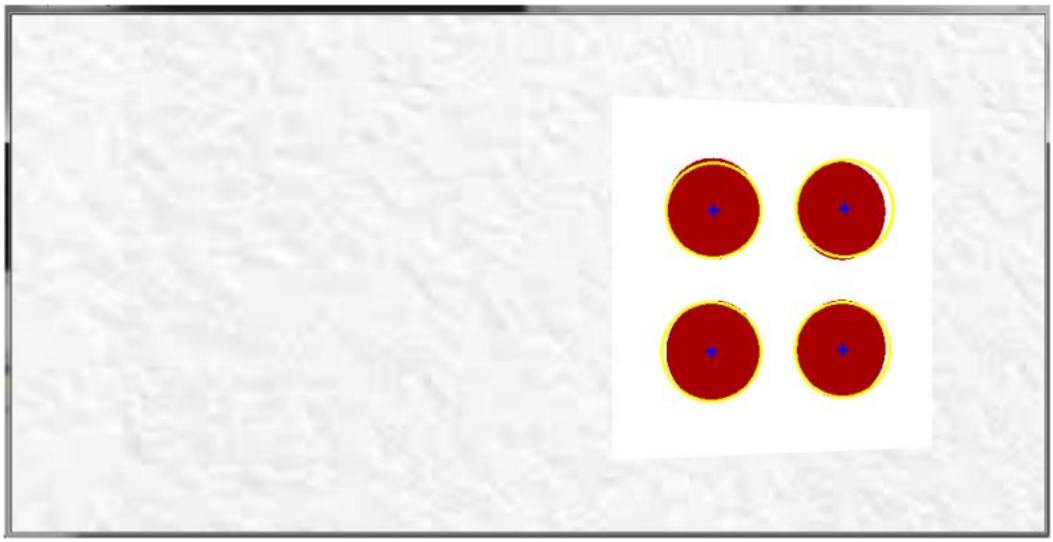


Figure III.10. The desired image with detection of the landmark 4 circles

The figure III.11 shows the robot placed at an arbitrary position in the environment and the image captured by the vision sensor at that place (figure III.12).

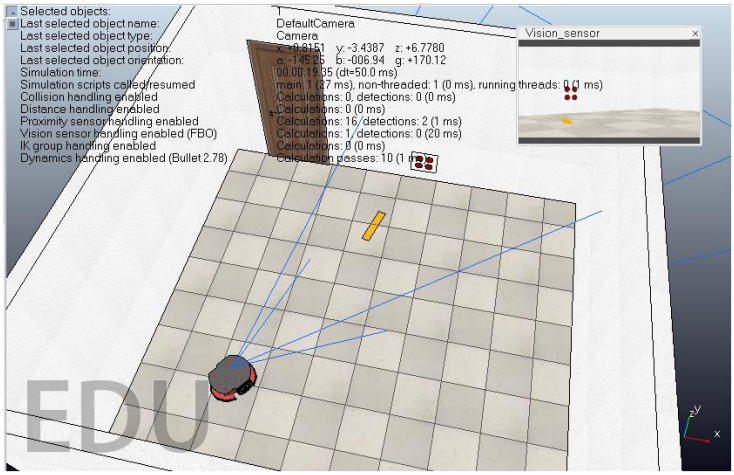


Figure III.11. Robot initial position and initial captured image

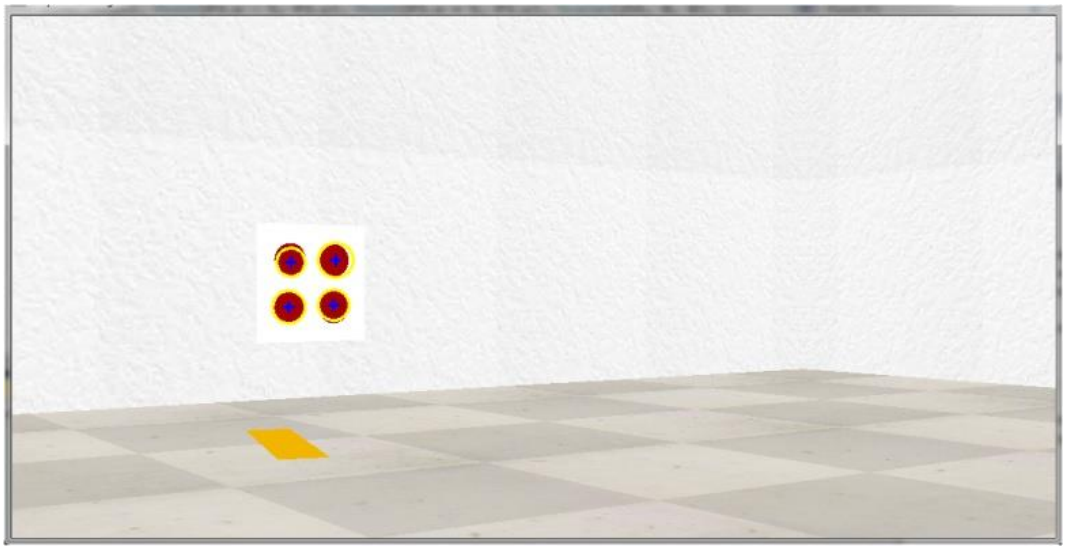


Figure III.12. Image captured from the initial position with circle detection

We can see in figure III.13 that the robot achieves the desired pose since its pose coincides with the mark's pose. The path followed by the robot in its way to the goal position is smooth and direct.

Also, by comparing the final captured image and the goal image in figure III.14 we see that they coincide with acceptable accuracy.

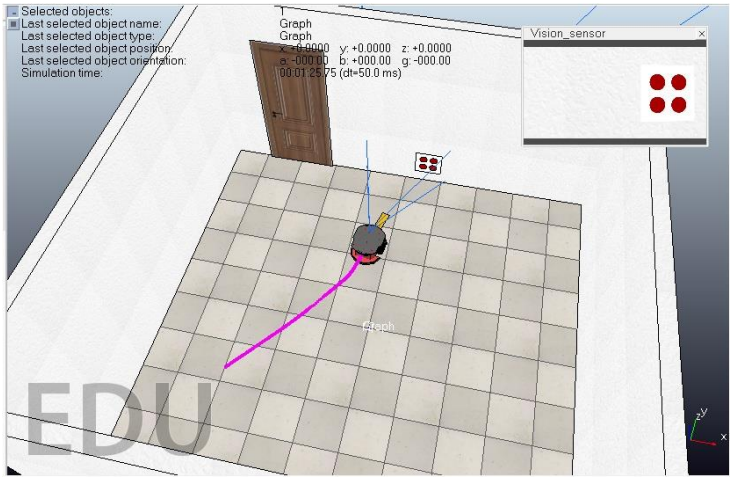


Figure III.13. Robot final position and final captured image

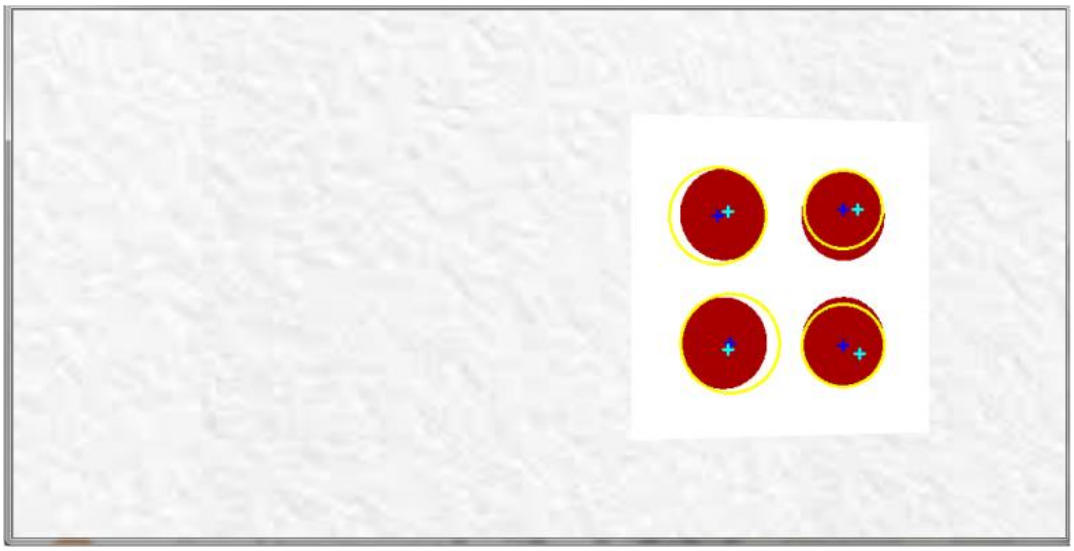


Figure III.14. Features in image captured at the final position (Blue crosses) compared to the features in the goal image (crosses in cyan)

III.5. The program

We use in this code the C++ programming language, openCV library for image processing, and armadillo library for matrix calculations.

```
#include  <Windows.h>
#include <iostream>
#include <stdio.h>
#include <opencv2\opencv.hpp>
#include <armadillo>
#include <cmath>

extern "C" {
#include "extApi.h"
}

using namespace std;
using namespace cv;
using namespace arma;

int main()
{
	simxInt Vsensor = 0;
	simxInt leftmotorHandle = 0, rightmotorHandle = 0;
	Point P1, P2, P3, P4; // Circles centers
	cv::Mat grayframe;
	double omega_R, omega_L; // Right and left wheels rotation speeds
	vector<Vec3f>  circles;
	int resolution[2];
	simxUChar *image = 0;
	int v_sensor1;
	vec e(8); // Error
	double E; // Error norm
	vec C1(2), C2(2), C3(2), C4(2); // Circles centers in respect to camera system
	int Z; // Depth
	mat L(8, 3); // Interaction matrix
	vec vitesse(3); // Speeds vector
	vec k(3); mat K;
	double omega_x, V; // Rotation and translation motor's speeds
	double l = 0.33, r = 0.1; // Distance between the robot two wheels and wheel's
	radius

// Start new connection with the server(CoppeliaSim)
	simxFinish(-1);   // Close any previously unfinished business
	simxInt clientID = simxStart("127.0.0.1", 19999, true, true, 5000, 5);
	Sleep(1000);
	if (clientID != -1)
	{
		cout << " Connection status to VREP: SUCCESS" << endl;
		simxSynchronous(clientID, 1);
		// Synchronization
		simxStartSimulation(clientID, simx_opmode_oneshot);                //
Start simulation

    // Get object's handle (Vision sensor, left and right motors)
		simxGetObjectHandle(clientID, "Vision_sensor", &Vsensor,
		simx_opmode_blocking);
		simxGetObjectHandle(clientID, "Pioneer_p3dx_leftMotor", &leftmotorHandle,
		simx_opmode_blocking);
		simxGetObjectHandle(clientID, "Pioneer_p3dx_rightMotor",
		&rightmotorHandle, simx_opmode_blocking);
```

```
// Detect features from goal image by CHT:
		cv::Mat image1 = imread("d:/GoalImage.png", CV_LOAD_IMAGE_UNCHANGED);
//Load image goal file from disk
		//namedWindow("Goal image", CV_WINDOW_AUTOSIZE);
		//imshow("Goal image", image1);
		cvtColor(image1, grayframe, CV_BGR2GRAY);

		vector<Vec3f>  Goal_circles;
		HoughCircles(grayframe, Goal_circles, CV_HOUGH_GRADIENT,
			2,	// accumulator resolution (size of the image / 2)
			20,	// minimum distance between two circles
			55,// 80,// 55,	// Canny high threshold
			45,	// minimum number of votes
			10, 58); // min and max radius
		/*while (Goal_circles.size() != 4)
		{
			cout << "Different de 4" << endl;
			HoughCircles(grayframe, Goal_circles, CV_HOUGH_GRADIENT, 2, 20,
			55, 45, 10, 58);
		}*/
		cout << endl << "Sorti de la boucle" << endl;

// Class the four centers of circles
		vec V1, V2;
		V1 << cvRound(Goal_circles[0][0]) << cvRound(Goal_circles[1][0]) <<
		cvRound(Goal_circles[2][0]) << cvRound(Goal_circles[3][0]);
		V2 << cvRound(Goal_circles[0][1]) << cvRound(Goal_circles[1][1]) <<
		cvRound(Goal_circles[2][1]) << cvRound(Goal_circles[3][1]);

		// Drowing the 4 circles:
		for (int i = 0; i < Goal_circles.size(); ++i)
			circle(image1, Point(V1(i), V2(i)), cvRound(Goal_circles[i][2]),
			Scalar(0, 255, 255), 2);

		uvec ind = sort_index(V1);
		V1 = sort(V1);
		V2 = V2.elem(ind);

		if (V2(0) < V2(1))
		{
			P1 = Point(V1(0), V2(0));
			P3 = Point(V1(1), V2(1));
		}
		else
		{
			P3 = Point(V1(0), V2(0));
			P1 = Point(V1(1), V2(1));
		}

		if (V2(2) < V2(3))
		{
			P2 = Point(V1(2), V2(2));
			P4 = Point(V1(3), V2(3));
		}
		else
		{
			P4 = Point(V1(2), V2(2));
			P2 = Point(V1(3), V2(3));
		}

    // Drowing blue crosses on the 4 centres:
```

```
            line(image1, Point(P1.x - 10, P1.y), Point(P1.x + 10, P1.y), Scalar(255,
            0, 0), 2);
            line(image1, Point(P1.x, P1.y - 10), Point(P1.x, P1.y + 10), Scalar(255,
            0, 0), 2);
            line(image1, Point(P2.x - 10, P2.y), Point(P2.x + 10, P2.y), Scalar(255,
            0, 0), 2);
            line(image1, Point(P2.x, P2.y - 10), Point(P2.x, P2.y + 10), Scalar(255,
            0, 0), 2);
            line(image1, Point(P3.x - 10, P3.y), Point(P3.x + 10, P3.y), Scalar(255,
            0, 0), 2);
            line(image1, Point(P3.x, P3.y - 10), Point(P3.x, P3.y + 10), Scalar(255,
            0, 0), 2);
            line(image1, Point(P4.x - 10, P4.y), Point(P4.x + 10, P4.y), Scalar(255,
            0, 0), 2);
            line(image1, Point(P4.x, P4.y - 10), Point(P4.x, P4.y + 10), Scalar(255,
            0, 0), 2);
            //namedWindow("Goal image", CV_WINDOW_AUTOSIZE);
            //imshow("Goal image", image1);
            //waitKey();

    // Coodinates of the 4 features in respect to camera's system of coordinates

            vec C1_Goal(2); C1_Goal << (512 / 2) - P1.y << P1.x - (1024 / 2);
            vec C2_Goal(2); C2_Goal << (512 / 2) - P2.y << P2.x - (1024 / 2);
            vec C3_Goal(2); C3_Goal << (512 / 2) - P3.y << P3.x - (1024 / 2);
            vec C4_Goal(2); C4_Goal << (512 / 2) - P4.y << P4.x - (1024 / 2);

            // Compute the focal lenght
            float angVision = 46 * 3.14 / 180;
            float f = 1024 / (2 * tan(angVision / 2));     // focal

            while (1)
            {
       // Capture an image
                    v_sensor1 = simxGetVisionSensorImage(clientID, Vsensor,
                    resolution, &image, 0, simx_opmode_oneshot_wait);
                    if (v_sensor1 != simx_return_ok)
                    {
                           cout << endl << "Capture not OK" << endl << endl;
                           //waitKey();
                    }
       // Transform in openCV image
                    cv::Mat image2(resolution[1], resolution[0], CV_8UC3, image);
                    cvtColor(image2, image2, COLOR_RGB2BGR);
                    flip(image2, image2, 0);

       // Detect features from goal image by CHT:
                    cvtColor(image2, grayframe, CV_BGR2GRAY);

                    HoughCircles(grayframe, circles, CV_HOUGH_GRADIENT,
                           2,       // accumulator resolution (size of the image / 2)
                           30,      // minimum distance between two circles
                           68,      // Canny high threshold
                           60,      // minimum number of votes
                           10, 60); // min and max radius
                    cout << circles.size() << endl;
                    /*while (circles.size() != 4)
                    {
                           cout << circles.size() << endl;  // waitKey;
                           HoughCircles(grayframe, circles, CV_HOUGH_GRADIENT, 2, 30,
                           75, 50, 10, 60);// 2, 30, 60, 45, 10, 100);
                    }*/
```

```
                cout << endl << "Sorti de la boucle" << endl;

// Class the four centers of circles
                V1 << cvRound(circles[0][0]) << cvRound(circles[1][0]) <<
                cvRound(circles[2][0]) << cvRound(circles[3][0]);
                V2 << cvRound(circles[0][1]) << cvRound(circles[1][1]) <<
                cvRound(circles[2][1]) << cvRound(circles[3][1]);

                // Drowing the 4 circles in image2
                for (int i = 0; i < circles.size(); ++i)
                        circle(image2, Point(V1(i), V2(i)), cvRound(circles[i][2]),
                        Scalar(0, 255, 255), 2);

                //namedWindow("1iere image", CV_WINDOW_AUTOSIZE);
                //imshow("1iere image", image2); waitKey();

                ind = sort_index(V1);
                V1 = sort(V1);
                V2 = V2.elem(ind);

                if (V2(0) < V2(1))
                {
                        P1 = Point(V1(0), V2(0));
                        P3 = Point(V1(1), V2(1));
                }
                else
                {
                        P3 = Point(V1(0), V2(0));
                        P1 = Point(V1(1), V2(1));
                }

                if (V2(2) < V2(3))
                {
                        P2 = Point(V1(2), V2(2));
                        P4 = Point(V1(3), V2(3));
                }
                else
                {
                        P4 = Point(V1(2), V2(2));
                        P2 = Point(V1(3), V2(3));
                }

// Drowing blue crosses on the 4 centres:
                line(image2, Point(P1.x - 10, P1.y), Point(P1.x + 10, P1.y),
                Scalar(255, 0, 0), 2);
                line(image2, Point(P1.x, P1.y - 10), Point(P1.x, P1.y + 10),
                Scalar(255, 0, 0), 2);
                line(image2, Point(P2.x - 5, P2.y), Point(P2.x + 10, P2.y),
                Scalar(255, 0, 0), 2);
                line(image2, Point(P2.x, P2.y - 10), Point(P2.x, P2.y + 10),
                Scalar(255, 0, 0), 2);
                line(image2, Point(P3.x - 10, P3.y), Point(P3.x + 10, P3.y),
                Scalar(255, 0, 0), 2);
                line(image2, Point(P3.x, P3.y - 10), Point(P3.x, P3.y + 10),
                Scalar(255, 0, 0), 2);
                line(image2, Point(P4.x - 10, P4.y), Point(P4.x + 10, P4.y),
                Scalar(255, 0, 0), 2);
                line(image2, Point(P4.x, P4.y - 10), Point(P4.x, P4.y + 10),
                Scalar(255, 0, 0), 2);
                //namedWindow("Goal image", CV_WINDOW_AUTOSIZE);
                //imshow("Captured image", image2);
```

```
                    //waitKey();

    // Coordinates of the four centers in respect to the camera system
                    C1 << (512 / 2) - P1.y << P1.x - (1024 / 2);
                    C2 << (512 / 2) - P2.y << P2.x - (1024 / 2);
                    C3 << (512 / 2) - P3.y << P3.x - (1024 / 2);
                    C4 << (512 / 2) - P4.y << P4.x - (1024 / 2);

       // Error calculation
                    e << C1(0) - C1_Goal(0) << C1(1) - C1_Goal(1) << C2(0) -
                    C2_Goal(0) << C2(1) - C2_Goal(1) << C3(0) - C3_Goal(0) << C3(1) -
                    C3_Goal(1) << C4(0) - C4_Goal(0) << C4(1) - C4_Goal(1);
                    E = norm(e);
                    cout << "E = " << E << endl;

       // The depth Calculation
                    Z = (f / 2)*(((0.14) / (C1(0) - C3(0))) + ((0.14) / (C2(0) -
                    C4(0))));

       // The interraction matrix L:
                    L << 0 << C1(0) / Z << C1(0)*C1(1) / f << endr
                           << -f / Z << C1(1) / Z << f + pow(C1(1), 2) / f << endr
                           << 0 << C2(0) / Z << C2(0)*C2(1) / f << endr
                           << -f / Z << C2(1) / Z << f + pow(C2(1), 2) / f << endr
                           << 0 << C3(0) / Z << C3(0)*C3(1) / f << endr
                           << -f / Z << C3(1) / Z << f + pow(C3(1), 2) / f << endr
                           << 0 << C4(0) / Z << C4(0)*C4(1) / f << endr
                           << -f / Z << C4(1) / Z << f + pow(C4(1), 2) / f;

       // Speeds calculation
                    k << 0.003 << 0.003 << 0.003; K = 1 * diagmat(k);
                    vitesse = -pinv(L)*e; vitesse = K*vitesse;

                    omega_x = vitesse(2);
                    V = sqrt(pow(vitesse(0), 2) + pow(vitesse(1), 2));  // m/s

                    omega_R = V + (l*omega_x / (2 * r)); omega_R *= 180 / 3.14;    //
                    in degree/s
                    omega_L = V - (l*omega_x / (2 * r)); omega_L *= 180 / 3.14;

       // Command the robot
                    simxSetJointTargetVelocity(clientID, leftmotorHandle, omega_L,
                    simx_opmode_streaming);
                    simxSetJointTargetVelocity(clientID, rightmotorHandle, omega_R,
                    simx_opmode_streaming);

                    Sleep(200);
             }
       }
       else
             cout << "Connetion failed, clientID = " << clientID << endl;
       return 0;
}
```

III.6. conclusion

In this chapter, the concepts presented in the previous chapters have been used to write a program that simulates the positioning of a mobile robot with an on-board camera, in its environment with respect to a landmark.

General conclusion

Conclusion

Above all, this project allowed us to discover the powerful CoppeliaSim robot simulator and to learn how to work with it. CoppeliaSim is a powerful simulator since it contains the dynamic models of almost all types of robots (manipulator robots, flying robots, bipeds, etc.) and almost all types of sensors (US sensor, laser, vision sensor, etc.). The construction of the virtual 3D environment in CoppeliaSim is very simple and offers several possibilities (such as for example the placement of static or mobile obstacles in the vicinity of the robot).

This project also allowed us to understand the principles of visual servoing and to apply it to a simulation of unicycle-type robot.

There are several parameters to set in the program, such as the parameters of the Hough transform function, the lambda parameter in the command calculation and finally the time interval between two successive commands. The results of the servo-control greatly depend on the choice of these parameters. This choice of these parameters is not always easy to make since it is done by trial and error and requires several tests.

Problems: The OpenCV Circle Hough Transform function doesn't give always the same results with the same image. In our captured images we should find four circles, but sometimes the CHT function detects less or more circles. This is why in the program we should execute the CHT function many times in a loop until the number of circles detected equal four.

The error remaining between the target position and the real position is due to the non-holonomi constraint of the unicycle mobile robot

The four circles of the landmark aren't perfectly circular in the image because of perspective. That's why there detection isn't accurate, and this has impact on the servoing process.

Bibliography

[1] Bernard BAYLE, « Robotique mobile », Ecole Nationale Supérieur de Physique de Strasbourg & Université de Strasbourg. 2010

[2] James Calusdian · Xiaoping Yun, "A simple and highly portable MATLAB interface for learning robotics", SN Applied Sciences journal, july 2019

[3] Amanda Whitbrook, "Programming Mobile Robots with Aria and Player", Springer 2010

[4] S. FEKNOUS, "Vision intelligente" course for students of M2 automation and industrial computing, university Amar Telidji of Laghouat. Electronic department, 2019-2020

[5] S. Chetan More and Anita P. Patil, "Circular Hough Transform For Detecting And Measuring Circles of Object", International Journal on Recent and Innovation Trends in Computing and Communication. Volume 3, Issue 2. February 2015

[6] Adrian Kaehler & Gary Bradski, "Learning OpenCV 3", O'Reilly edition, 2017

[7] F. Chaumette, Chapter 3 "Asservissement visual" of the book « La commande des robots manipulateurs » of W. Khalil, Hermès Editions, 2002

[8] Seth Hutchinson, Gregory D.Hager, Peter I.Corke, "A tutorial on visual servo control". IEEE TRANSACTION ON ROBOTIC AND AUTOMATION, VOL 12 NO 5, October 1996

[9] Armadillo C++ library website address: http://arma.sourceforge.net/. Last visited in August 2020

[10] Tufa TEPE, "Mobile robot navigation using visual servoing", M.Sc. internship, TU/e University of technology, Eindhoven

[11] Alper Yilmaz, Mubarak Shah, "Hough Transform", Lecture-17, CAP 5415 Computer Vision, Fall 2013.

[12] CoppeliaSim user manual Version 4.1.0, https://www.coppeliarobotics.com/helpFiles/. Last visited in August 2020.

[13] Leopoldo Armesto, Profesor de Robótica, Universitat Politécnica de Valencia Espagne, « Introduction to CoppeliaSim » video courses on youtube, Last visited in September 2020.

[14] M. Kevin Lynch, "CoppeliaSim introduction", Northwestern Mechatronics Wiki, http://hades.mech.northwestern.edu/index.php/CoppeliaSim_Introduction. Last visited in September 2020.