UNIVERSITÄT SIEGEN

Embedded Control Lab 3: Road Following in Coppelia Sim: An Introduction to Robotic Navigation

by:
Group 8:
Davin Edison
Hatim Sadliwala
Ishita Singh
Oussema Kassebi

Embedded Control Laboratory

Davin Edison, Ishita Singh, Oussema Kasebi, Hatim

December 2023

Contents

List of Figures

1 Introduction

Lane following has been a standard for automotive industry for the past few years now. Started with Honda Accord 2003 which featured a LKAS (Lane-Keeping Assist System). This early system could provide steering input to help keep the car in the center of the lane, albeit with limitations in terms of capabilities and the need for frequent driver intervention.

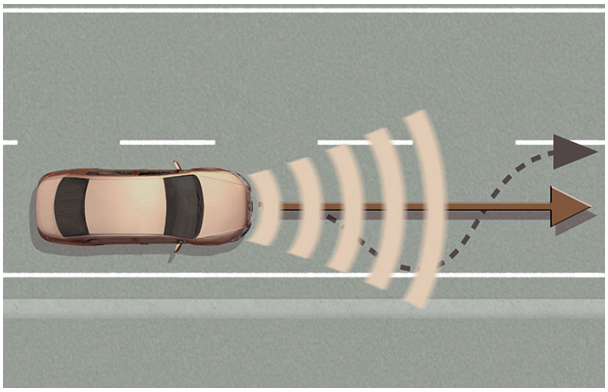

Figure 1.1: Lane Keep Assist Technology

LKAS acceptance by the public was somewhat mixed, primarily due to several factors:

1. **Technology Limitations** As an early version of lane-keeping assist technology, the LKAS in 2003 Honda Accord was designed to work in relatively simple driving scenarios, primarily on highways, and required clear lane markings to function correctly.
2. **Market Availability** The LKAS feature was initially available only in certain markets, such as Japan and not globally, thus this limits the exposure of 2003 Honda Accord LKAS feature.
3. **Consumer Awareness and Trust** In the early 2000s, consumer awareness and trust in autonomous driving features were still developing. Many drivers were not familiar with or entirely comfortable relying on automated driving assistance technology.

Fast forward to 2013-2014, Mercedes Benz launched its 2014 S-Class which featured an advanced version of lane-keeping assist as part of its suite of driver assistance technology. This system was more sophisticated than earlier versions found in other vehicles and included features like:

- Active steering interventions to help keep the car centered in its lane.
- The ability to detect and follow lane markings more accurately.
- Integration with other driver-assistance features for a more cohesive driving experience.

The S-Class's lane-keeping assist was part of a larger package of innovative features that included adaptive cruise control, blind-spot detection, and a pre-collision system, which together offered a near-autonomous driving experience under certain conditions.

2 Robotic Navigation

Robotic navigation, a critical component in the fields of robotics, revolves around the ability of a robot to determine its position and plan its movement within an environment. Fundamentally, it incorporates sensor technologies such as LIDAR, ultrasonic sensors, and cameras to perceive surroundings, coupled with sophisticated algorithms for mapping, localization, and path planning. The development of robotic navigation has been propelled by advances in computational power, sensor accuracy, and artificial intelligence, particularly in machine learning and computer vision.

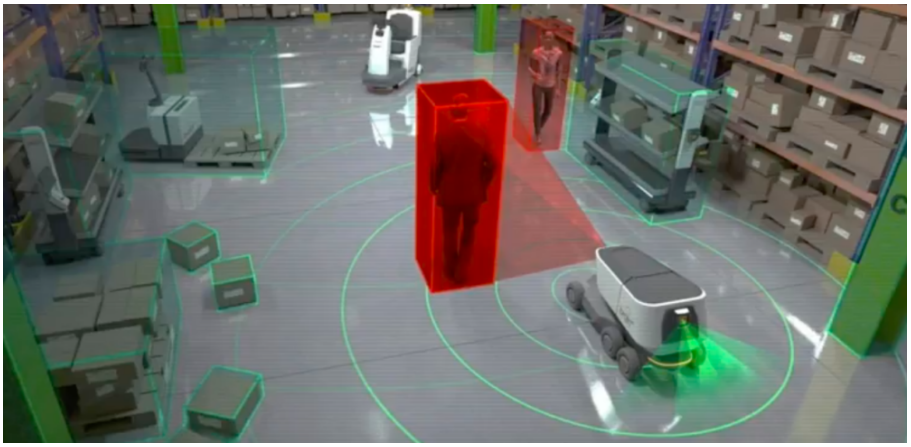

Figure 2.1: Robotic Navigation

The advantages of advanced robotic navigation include increased efficiency and precision in tasks, reduced need for human intervention and the ability to operate in hazardous or inaccessible environments.

3 CoppeliaSim

CoppeliaSim, formerly known as V-REP, is a versatile and powerful 3D robot simulation software used extensively in robotics for simulation, educational purposes, and research. It operates as a virtual robotic experimentation platform, enabling users to create, customize, and control a wide range of robotic models and scenarios. The core of CoppeliaSim's functionality lies in its ability to provide a highly detailed and accurate physics simulation environment, supporting various physics engines. This allows for realistic modelling of complex robotic mechanisms, sensor behaviors, and interactions with environments.

One of the primary advantages of CoppeliaSim is its flexibility and extensibility. It supports numerous programming languages and APIs, including remote API functionalities for integration with external software like MATLAB or Python. This makes it an ideal tool for prototyping a nd testing robotic algorithms without the need for physical hardware, reducing development costs and time. Additionally, its user-friendly interface and rich set of features make it accessible to both beginners and advanced users.

CoppeliaSim is known for its versatility and power in robotics simulation, offers a broad range of features that makes it a popular choice among researchers, educators, and developers in the field of robotics and automation. Some of its key features include:

1. **Advanced Physics Simulation** CoppeliaSim, supports multiple physics engines like Bullet, ODE, and Vortex, allowing for accurate and realistic physics simulations. This includes the simulation of rigid body dynamics, soft body dynamics, fluid dynamics and collision detection.
2. **3D Modeling and Rendering** The software provides detailed 3D modeling capabilities along with high-quality rendering. Users can create, import, and modify complex robot models and environments.
3. **Robust Scripting Support** CoppeliaSimallows users to write control scripts in several programming languages, including Lua, Python, and C++. This scripting flexibility enables complex algorithm implementation and automation within the simulation environment.
4. **Sensor Simulation** A wide range of sensor types can be simulated, including proximity sensors, vision sensors, force sensors, and more, which are crucial for developing and testing perception algorithms.
5. **Multi-Robot Simulation** CoppeliaSim can simulate multiple robots simultaneously, allowing for study of swarm robotics, cooperative tasks, and multi-robot interactions.
6. **Remote API Functionality** It offers remote API capabilities, enabling the simulation to be controlled from external programs or scripts, thereby providing integration flexibility with other software tools.

7. **Real-Time Simulation Control** It allows for real-time control and adjustment of the simulation, including pause, speed-up or slow-down of the simulation time.

8. **Path Planning and Navigation** CoppeliaSim includes features for path planning and navigation, which are essential for autonomous robots.

3.1 CoppeliaSim in Engineering Development

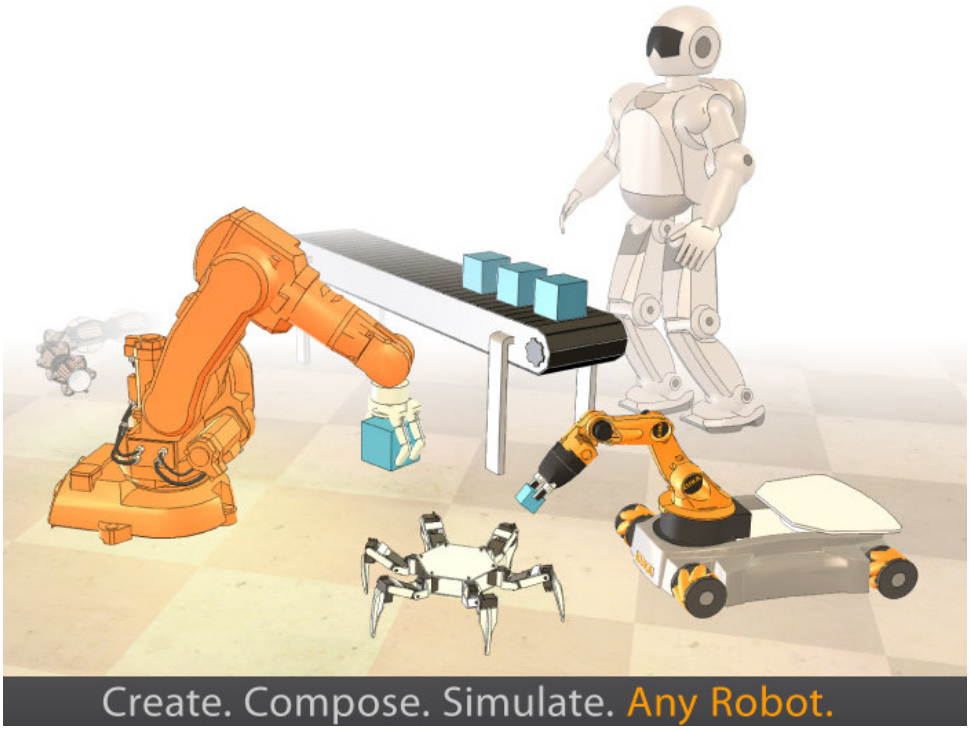

Figure 3.1: CoppeliaSim Vast Robot Libraries

CoppeliaSim serves as an invaluable asset for engineering, particularly in the fields of robotics and automation. Its comprehensive simulation environment allows engineers to design, test, and refine robotic systems and components in a virtual space before physical prototypes are built. This capability is crucial in engineering, where iterative design and testing are key to developing efficient and reliable systems. By using CoppeliaSim, engineers can simulate real-world physics, including dynamics and kinematics, ensuring that their designs are not only theoretically sound but also piratically feasible. The software's ability to model complex interactions with various environments and its support for a wide range of sensors and actuators enables engineers to closely mimic real-world conditions, thus reducing the time and cost associated with physical prototyping and testing.

Additionally, CoppeliaSim's scripting capabilities in languages like Python and Lua allow for the integration of custom algorithms and control systems, making it a versatile tool for research and development. Whether it's for designing industrial robots, autonomous vehicle, or advanced automation systems, CoppeliaSim provides a robust, flexible, and cost-effective solution for engineers to bring their innovative ideas to life in a controlled and safe virtual environment. This not only accelerates the development process but also enhances the overall quality and reliability of engineering solutions.

4 Lab Environment Setup

4.1 Setting Up CoppeliaSim

CoppeliaSim can be installed to user's system by downloading the edu version from their official site - https://www.coppeliarobotics.co
To verify that the installation is successful, an instance of CoppeliaSim can be launched and explored its basic features.

For this Lab Task, a custom scene with multiple robots has been prepared for this lab. Upon starting CoppeliaSim, load the provided scene file. This file contains predefined paths and robot models setup for the road following task.

4.2 Provided CoppeliaSim Scene

Here are the overview of the provided CoppeliaSim Scene for this task

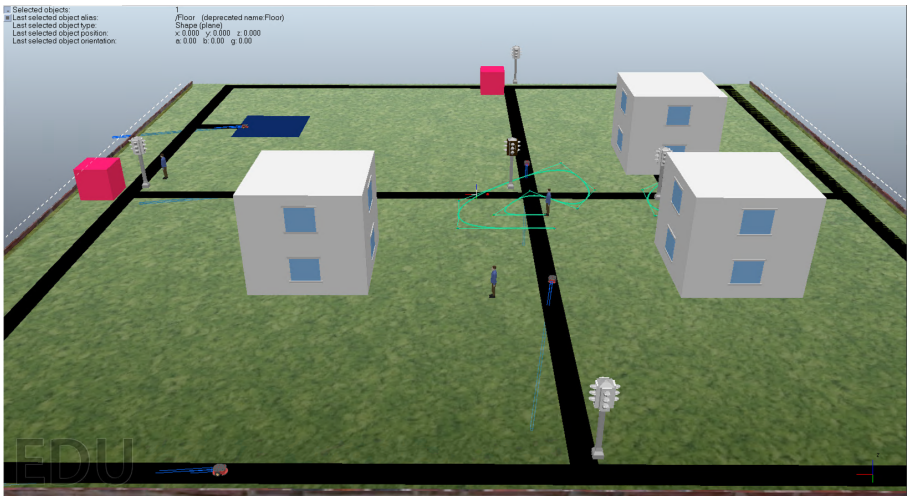

Figure 4.1: Top View of CoppeliaSim Scene

In this task, the provided CoppeliaSim Scene has 5 robots that are on the road. Each robot is equipped with cameras which can be seen in Fig 4.2

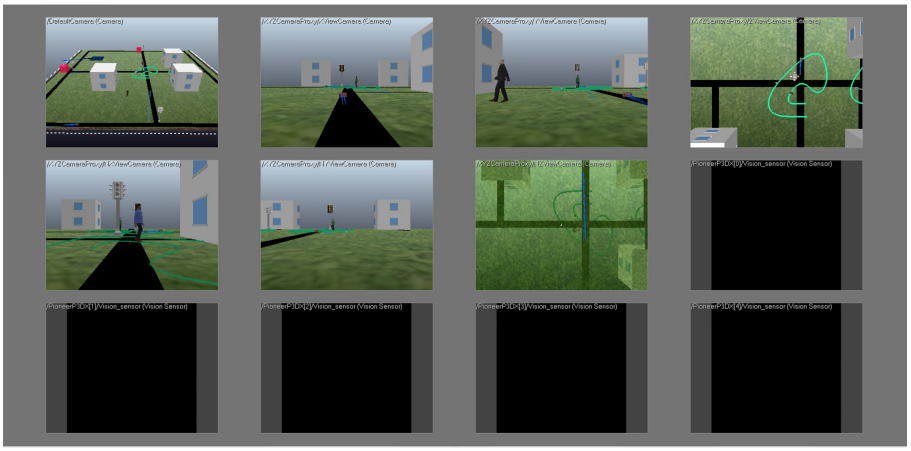

Figure 4.2: Several Cameras in the provided CoppeliaSim scene

4.3 Python Setup for Robot Control

If not already installed, Python has to be installed to user's system. Python 3.x is recommended for compatibility. Ensure that user's Python environment is working correctly by running a simple test script.

CoppeliaSim Python API is crucial for interfacing with CoppeliaSim from Python scripts. It can be installed by using this syntax:

```
python3 -m pip install coppeliasim-zmqremoteapi-client
```

User will also need additional libraries like numpy for numerical operations and opencv for image processing. Install these using a package manager like pip or conda.

```
conda install conda-forge::opencv
```

To test that CoppeliaSim Python API is working properly, a basic script that interacts with the simulation, such as staring and stopping the simulation has to be run. This setup provides the foundation for user to start developing and testing your road following algorithms in a controlled simulation environment. The next section will guide user through the development of the road following algorithm.

5 Road Following Algorithm Development

5.1 Algorithmic Foundation

The core of this taks is to develop an algorithm that enables a simulated robot to follow a designated path. User may opt for various approaches such as sensor-based line following, computer vision techniques, or even basic AI algorithms.

5.2 Sensor Integration

Utilize the simulated vision sensors in CoppeliaSim to detect the path.

5.3 Control Logic

Develop the control logic that translates sensor input into motor commands. This could involve simple steering mechanisms or more complex control systems like PID controllers.

5.4 Programming

User algorithm should be written in Python, ensuring it interfaces effectively with CoppeliaSim through the RemoteAPI. User should focus on writing clean, well-documented code for the ease of understanding and modification. Key concepts and approach for the code are:

- **Computer Vision**
  User should explore image processing techniques like color detection, edge detection or even neural networks for more advanced implementations.
- **Proportional Controller**
  For simple paths, a proportional controller might suffice to adjust the robot's steering based on the deviation from the path.
- **Debugging and Iteration** User should regularly test their algorithm in the simulation. Be prepared to iterate and debug based on the robot's performance.

5.5 Code Explanation

The core of this taks is to develop an algorithm that enables a simulated robot to follow a designated path. User may opt for various approaches such as sensor-based line following, computer vision techniques, or even basic AI algorithms.

The code is divided into several files in order to make it robust, clean and improve readability. Those files are:

5.5.1 constant.py

constant.py file consists of constant variables that will be used for the entirety of the program and PID control Logic. Which can be seen below.

```
#Setting up the constant variables for the entire program and PID Control Logic
res = 128.0 #For resolution initialization
center = res/2.0 #To find the center of the image

#PID Gains
kp = 0.05   #Setting up P Value in PID Controller
ki = 0.01   #Setting up I Value in PID Controller
kd = 0.0    #Setting up D Value in PID Controller
scale_factor = 0.5
```

5.5.2 utils.py

utils.py file consists function for image processing process. This is needed in order to read, process and make decision from the image that is taken from robot camera. Which can be seen below.

```
#File for image processing function
import numpy as np              #Importing NumPy to process matrixes
import cv2 as cv                #Importing OpenCV for image processing
import constant                 #Improting constant.py file to load all of the con
from constant import center     #Importing variable center from constant.py file


def processimage(image):
    image = list(image) #Read Image Array
    image = np.array(image, np.uint8)  #Changing image array to uint8 format
    image = image.reshape(constant.res[1], constant.res[1], 3) #reshaping the arra
    gimage = cv.cvtColor(image, cv.COLOR_RGB2GRAY) #Converting color image to Gray
    ret, thresh = cv.threshold(gimage, 20, 225, 0) #Finding the road inside the im
    # Calculating the centeroid using moments
    M = cv.moments(thresh)
    if M["m00"] == 0.0:
        cx = 0.0
        cy = center
    else:
        cx = float(M["m10"] / M["m00"])
        cy = float(M["m01"] / M["m00"])
    return cy
```

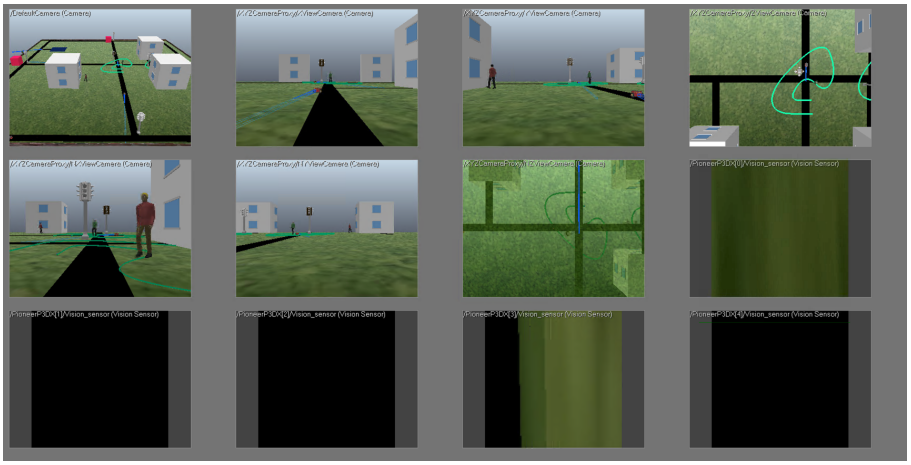

Figure 5.1: Direction Decision

As its explained in the paragraph above, the steering angle of the robot is calculated accroding to the input image from the robot camera. As it can be seen in Fig 5.1 a frame in the 3rd row and 3rd column shows that the black line is not in the center, so the author calcualte based on the center of the frame and tweak the robot steering angle according to the distance of the black pixel to the center of the frame.

5.5.3 robot.py

robot.py file consists a class for creating instances of robot object. It includes the initialization of the robot, steering angle calculation function, robot velocity calculation function, and update function to update the steering angle, and robot velocity according to its current location and orientation. Which can be seen below.

```
    #Script for creating a class for the robots with complete algorithm and genera
from coppeliasim_zmqremoteapi_client import RemoteAPIClient #Importing ZMW Remote
import constant     #Importing constant.py file to load all of the constant variab
import utils        #Improting utils.py file to load the image processing function
import numpy as np  #Importing NumPy to process matrixes
import cv2 as cv    #Importing OpenCV for image processing
from constant import res, center, kp, ki, kd, scale_factor  #Importing Variables f

#Creating a new instance of CoppeliaSim using Remote API Client from ZMQ and assig
client = RemoteAPIClient()
sim = client.require('sim')

#Create a class called Robots in which all of the functions of the robot can be ma
class Robots:
    motor_r = "null"
    motor_l = "null"
    vision = "null"
    targetVelocity_r = 5.0
    targetVelocity_l = 5.0
    speed = 5.0
    integral = 0
    error_prev = 0

#Constructor to set the values of some class variables. Also to do object selectin
    def __init__(self, num):
        self.motor_r = sim.getObject(f"/PioneerP3DX[{num}]/rightMotor")
        self.motor_l = sim.getObject(f"/PioneerP3DX[{num}]/leftMotor")
        self.vision = sim.getObject(f"/PioneerP3DX[{num}]/Vision_sensor")
        self.speed = 5.0
        self.integral = 0
        self.error_prev = 0

#Function for Calculating the steering angle using Image Processing and PID Contro
    def steeringcalc(self, cy):
        error = constant.center - cy
        self.integral = self.integral + error
        derivative = error - self.error_prev
        # Calculate PID output
        if error:
            pid_output = constant.kp * error + constant.ki * self.integral + const
        else:
            pid_output = 0.0
        # Save current error for the next iteration
        self.error_prev = error

        # Steering Angle Value
        steering_angle = constant.scale_factor * pid_output
        return steering_angle
```

```
#Function for calculating final velocity of the robot.
    def velocityangle(self, angle):
        self.targetVelocity_l = self.speed - (angle / 180) * self.speed
        self.targetVelocity_r = self.speed + (angle / 180) * self.speed

#Function that will update the condition of the robot in which will update the ste
    def update(self):
        image, constant.res = sim.getVisionSensorImg(self.vision)
        cy = utils.processimage(image)
        ang = self.steeringcalc(cy)
        self.velocityangle(ang)
        sim.setJointTargetVelocity(self.motor_r, self.targetVelocity_r)
        sim.setJointTargetVelocity(self.motor_l, self.targetVelocity_l)
```

5.5.4 **main**$_{c}$*ontroller.py*

main_controller.py file consists the main sequence of the code for this project, which are: Create Robot Objects -> Start Simulation -> Updating each robot steering angle and velocity through out the simulation. Which can be seen below.

```
#Main Program
#Create robot objects -> Start simulation -> updating each robot throughout the si
from coppeliasim_zmqremoteapi_client import RemoteAPIClient #Importing Communicati
client = RemoteAPIClient() #Create a new instance of CoppeliaSim Client in Python
sim = client.require('sim') #Create a new instance of CoppeliaSim called sim.
import robot #Importing robot.py file
from robot import Robots #Improting Robots class from robot.py file

def main():
    r0 = Robots(0) #Create an instance of Robot 0
    r1 = Robots(1) #Create an instance of Robot 1
    r2 = Robots(2) #Create an instance of Robot 2
    r3 = Robots(3) #Create an instance of Robot 3
    r4 = Robots(4) #Create an instance of Robot 4
    sim.setStepping(True)
    sim.startSimulation() #Starting the Simulation
    while (t := sim.getSimulationTime()) < 30: #Limiting the Simulation time to le
        print(f'Simulation time: {t:.2f} [s]') #Print out simulation time
        r0.update() #Updating the condition of the robot 0, steering angle, and ve
        r1.update() #Updating the condition of the robot 1, steering angle, and ve
        r2.update() #Updating the condition of the robot 2, steering angle, and ve
        r3.update() #Updating the condition of the robot 3, steering angle, and ve
        r4.update() #Updating the condition of the robot 4, steering angle, and ve
        sim.step() #Updating the simulation, take a new step in the simulation
    sim.stopSimulation() #Stop the simulation after t >= 30s
if __name__ == '__main__':
    main()
```

6 Summarize

In essence, this project offers the authors an opportunity to gain expertise in robot control through image processing. The methodology involves employing OpenCV alongside NumPy and Python for effective implementation. CoppeliaSim serves as a simulation platform, replicating real-world conditions for experimentation. Within the provided CoppeliaSim scene, the challenge is to control five robots with a singular controller. This task is streamlined by creating a dedicated class for each robot, facilitating cohesive and efficient management of the robotic entities.

The control mechanism for the robots involves utilizing input images from each robot, which are then processed into arrays using NumPy. To determine the current orientation of the robot, the author identifies the center of the image. This information is crucial for subsequent calculations that dictate the speed of the right and left motors in the robot. By carefully manipulating motor speeds, the system ensures the robot remains in the middle of the road, effectively controlling its direction. This sophisticated approach allows for precise navigation and alignment of the robots within their environment.

Leveraging the capabilities of CoppeliaSim proves to be instrumental in this project, providing the author with a dynamic platform to simulate real-world conditions. This simulation not only enhances the author's comprehension of robot behavior but also enables a comprehensive exploration of potential flaws in the design. The virtual environment serves as a crucial testing ground, offering insights into how the robots might react in various scenarios, ultimately contributing to the refinement of their functionality. One notable advantage of utilizing CoppeliaSim lies in its ability to significantly reduce costs associated with running non-ready robots in the actual physical world. This simulation-driven approach not only streamlines the development process but also mitigates potential risks and expenses by allowing the author to identify and rectify issues before deploying robots in real-world settings. In essence, the use of CoppeliaSim emerges as a strategic and cost-effective means to attain a nuanced understanding of robot behavior and optimize the design for practical applications.