UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI INGEGNERIA

Corso di Laurea Magistrale in
Engineering and Computer Science

Industrial Automation And Robotics B

Project:

BEER PRODUCTION AND SELF-DRIVE CAR

Students:

Allegra Davide Giuseppe

Miano Alberto

Musmeci Edoardo

ANNO ACCADEMICO 2023-2024

Contents

Overview

Our project regarding the first part of the course focuses on the implementation of a PLC system to control the brewing process. The brewing process is characterized by a number of complex processes that can be depicted through schemes such as SFC, the creation of a Ladder on the TIA Portal program, and so on. This system includes the management of the various stages of production, such as malt milling, fermentation, filtration, and bottling. The main objective is to improve production efficiency and ensure continuous monitoring of production conditions.

For the second part, concerning the robotics part , we realized an autonomous guided vehicle with obstacle detection using CoppeliaSim, an advanced robotics simulation environment. Innovation in the field of autonomous driving represents one of the most advanced frontiers in engineering and robotics. To achieve this we used the Dijkstra algorithm, which by means of a list of points, each representing coordinates, calculates the shortest path , in case the car during this path encounters a wall, then an obstacle, it recalculates the path looking for an alternative way to get to the destination.

Chapter 1

PLC Programming Part

1.1 Beer Process Description

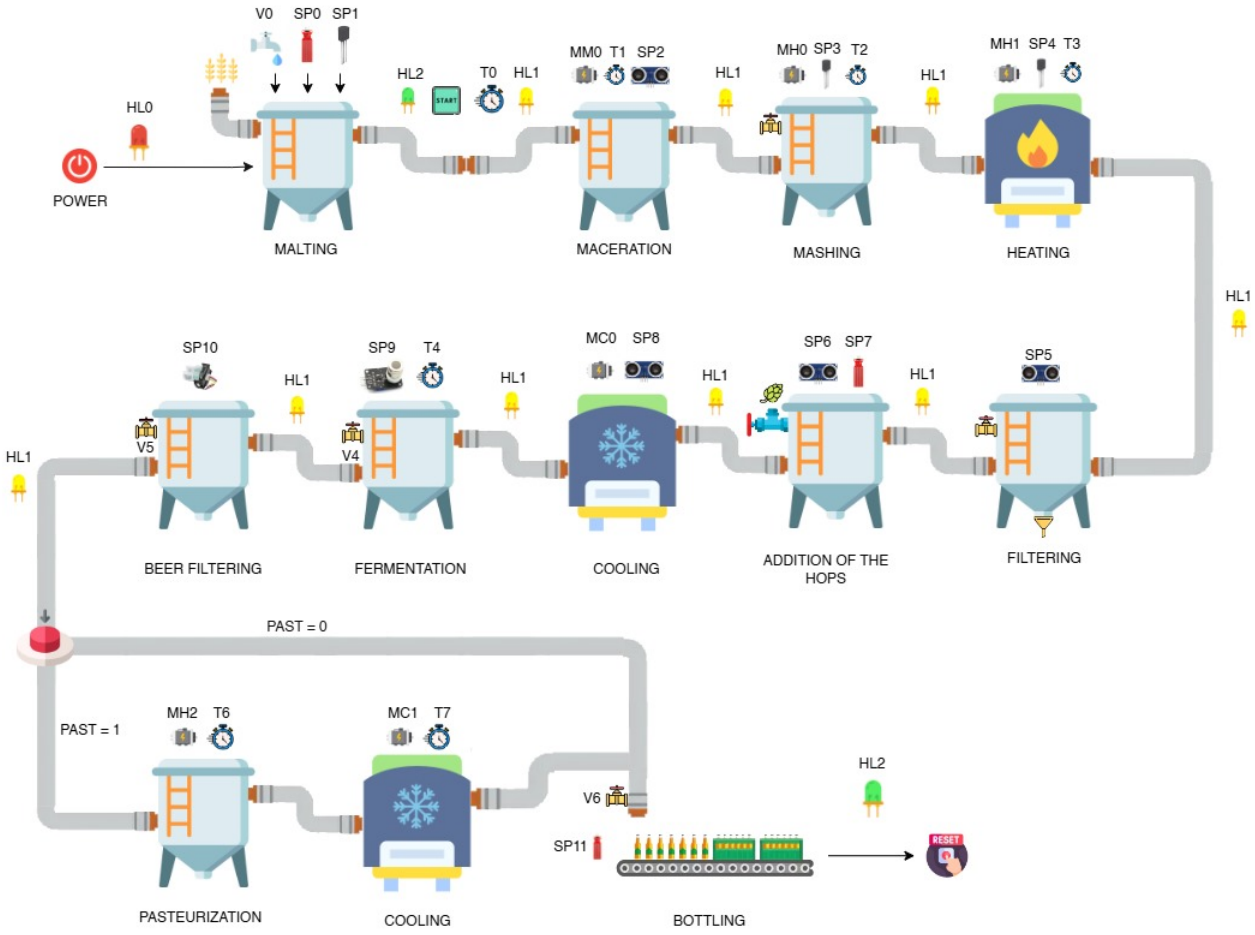

Figure 1.1: Phases of the process

- **Malting:** Barley grains are soaked in water to allow germination.
- **Maceration:** The germinated barley (malt) is mixed with water to activate enzymes that convert starches to sugars.

- **Mashing:** The mixture is heated to further the conversion of starches to fermentable sugars.
- **Heating:** The mash is heated in a kettle.
- **Filtration:** The mash is filtered to separate the liquid wort from the solid grain residues (spent grains).
- **Hop Addition:** Hops are added to the wort at various stages to impart bitterness and aroma.
- **Cooling:** The wort is cooled down to fermentation temperature.
- **Fermentation:** Yeast is added to the cooled wort to ferment the sugars, producing alcohol and carbon dioxide.
- **Beer Filtration:** The beer is filtered to remove any remaining yeast and particulates before packaging.

1.1.1 Schematization

Following the schematization which describes the production process is explained:

- State 0
  Empty maceration tank
  Red Light HL0
  IN: Press POWER button
- State 1
  Red Light HL0
  Activate valve V0
  Fill the maceration tank using a sensor SP0
  IN: SP0 = 1
- State 2
  Red Light ON
  Check if water 12 °C and 15 °C using SP1
  IN: SP1 = 1

- State 3
  Green Light HL2 ON
  Operator Press START button
- State 4
  Yellow Light ON
  Starting Timer T0
  IN : T0 = 10s
- State 5
  Yellow Light
  At the end of the timer, activate the motor maceration MM0 to macerate the malt
  Starting Timer T1
  IN: T1 = 30 s
- State 6
  Yellow Light
  Malt check by sensor SP2
  IN: SP2 = 1
- State 7
  Yellow Light
  Activate the Heading Motor (70 °C)
  Check if the water is on 70 °C using the sensor SP3
  IN: SP3 = 1
- State 8
  Yellow Light
  Activate the V1 valve to stir the malt
  Starting Timer T2
  IN: T2 = 30 s
- State 9
  Yellow Light
  At the end of timer T2, heading motor will be activated (Heading Motor MH1)
  Check if the water is 99 °C-104 °C using the sensor SP4
  IN: SP4 =1

- State 10
  Yellow Light
  Activate T3 timer for wort cooking
  IN: T3 = 10 s

- State 11
  Yellow Light
  Activation of valve V2 for threshing filtering
  Check via the SP5 sensor if filtering has been carried out
  IN: SP5 = 1

- State 12
  Yellow Light
  Activate valve V3 to add hops
  Check via the SP6 and SP7 sensors if the hops has been added
  IN: SP6 = 1 AND SP7 = 1

- State 13
  Yellow Light
  Activate a cooler motor to cool the wort
  Check if the wort has been cooled via the SP8 sensor
  IN: SP8 = 1

- State 14
  Yellow Light
  Activate valve V4 to begin fermentation
  Activate Timer T4 and sensor CO2(SP9) to monitor fermentation
  IN: T4 = 30 s AND SP9 = 1

- State 15
  Yellow Light
  Activate valve V5 To filter the beer
  Check if the beer is filtered using SP10
  IN: SP10 = 1

- State 16
  Yellow Light
  Check if press button PAST

Start Timer T5
IN: PAST = 1 AND TIMER ¡ 30

- State 17: Pasteurization
  Yellow Light
  Activate Heading Motor MH2 to pasteurize
  Start Timer T6
  IN: T6 = 30s

- State 18: Cooling
  Yellow Light
  Activate cooler motor MC1
  Start Timer T7
  IN: T7 = 30 s

- State 19: Bottling
  Yellow Light
  Activate valve V6 for bottling beer
  Check if the bottling level has been reached by means of a level sensor SP11
  IN: SP11 = 1

- State 20
  Green Light
  IN: RESET

1.1.2 Tables

Following the tables obtained from the schematization are shown:

INPUT

Label	Input Address	Comments
POWER	I 0.0	Power
START	I 0.1	Start
SPO	I 0.2	Water level sensor
SP1	I 0.3	Temperature sensor
SP2	I 0.4	Proximity sensor(tipo ottico o capacitivo)
SP3	I 0.5	Temperature Sensor
SP4	I 0.6	Temperature Sensor
SP5	I 0.7	Proximity Sensor
SP6	I 1.0	Proximity Sensor
SP7	I 1.1	Level Sensor
SP8	I 1.2	Temperature Sensor
SP9	I 1.3	CO2 Sensor
SP10	I 1.4	Sensori di torbidità o di chiarezza
SP11	I 1.5	Level Sensor
PAST	I 1.6	Pastorizzazione(bool)
RESET	I 1.7	Reset
EMERGENCY	I 2.0	Emergency

Figure 1.2

OUTPUT

Label	Output Address	Comments
HL0	Q 0.0	Red Light
HL1	Q 0.1	Yellow Light
HL2	Q 0.2	Green Light
V0	Q 0.3	Valve
V1	Q 0.4	Valve
V2	Q 0.5	Valve
V3	Q 0.6	Valve
V4	Q 0.7	Valve
V5	Q 1.0	Valve
V6	Q 1.1	Valve
MM0	Q 1.2	Maceration Motor
MH0	Q 1.3	Heading Motor
MH1	Q 1.4	Heading Motor
MH2	Q 1.5	Heading Motor
MC0	Q 1.6	Cooler Motor
MC1	Q 1.7	Cooler Motor

Figure 1.3

STATE

Label	Address
State 0	M 3.0
State 1	M 3.1
State 2	M 3.2
State 3	M 3.3
State 4	M 3.4
State 5	M 3.5
State 6	M 3.6
State 7	M 3.7
State 8	M 4.0
State 9	M 4.1
State 10	M 4.2
State 11	M 4.3
State 12	M 4.4
State 13	M 4.5
State 14	M 4.6
State 15	M 4.7
State 16	M 5.0
State 17	M 5.1
State 18	M 5.2
State 19	M 5.3
State 20	M 5.4

Figure 1.4: States of SFC

TIMER

Label	Memory Address
T0	M 2.0
T1	M 2.1
T2	M 2.2
T3	M 2.3
T4	M 2.4
T5	M 2.5
T6	M 2.6
T7	M 2.7

Figure 1.5: Timer

1.1.3 SFC

SFC stands for Sequential Function Chart, and it is a graphical programming language commonly used in the field of industrial automation. SFC is part of the IEC 61131-3 standard, which defines several programming languages for PLCs. Here are key points about SFC:

- Graphical Representation: SFC uses a graphical representation to model the sequential control of processes. The chart is organized into steps, actions, and transitions, making it visually intuitive for understanding and designing complex control sequences.
- Elements of SFC: - Steps: These represent individual operations or states in a process. Each step may have associated actions. - Actions: Actions are specific tasks or operations performed within a step. - Transitions: Transitions indicate the conditions under which the system moves from one step to another. They define the logical flow of the sequence.
- Parallel Execution: SFC allows for parallel execution of steps, which is particularly useful for modeling systems with multiple concurrent processes.
- State-Based Control: SFC is well-suited for representing systems with different states. The chart helps define the conditions under which a system transitions from one state to another.
- Structured Programming: SFC promotes structured programming practices, making it easier to design, understand, and maintain complex control logic.
- Used in Combination: SFC is often used in combination with other IEC 61131-3 languages, such as ladder logic (LADDER) or structured text (ST), to create comprehensive and efficient control programs for PLCs.

Following, in Figure 1.6 , the Sequential Function Chart, obtain from the schematization, is shown:

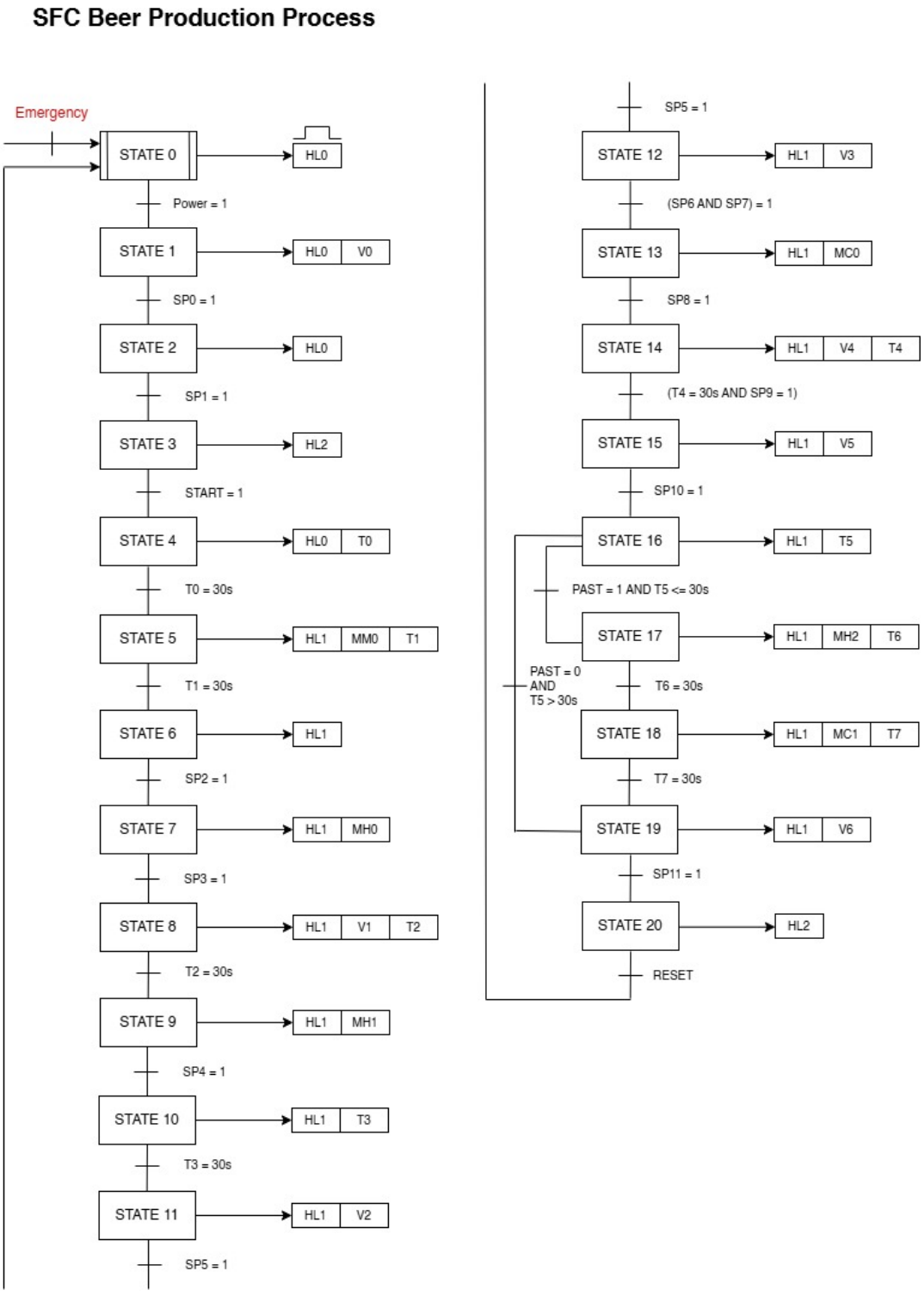

Figure 1.6: States of SFC

1.1.4 Ladder Diagramm

The Ladder Diagram (LADDER) is a graphical programming language widely used in industrial automation, specifically for programming programmable logic controllers (PLCs). It is one of the programming languages defined by the IEC 61131-3 standard. Here are key features and concepts related to the Ladder Diagram:

- Graphical Representation: The Ladder Diagram is graphically represented as a ladder-like structure, with horizontal and vertical lines forming the "rungs" of the ladder. Symbols and graphical elements represent various electrical and control elements.
- Logical Control Elements: - Contacts: Represent input conditions or sensors. They can be normally open (NO) or normally closed (NC), indicating whether the associated condition is true or false. - Coils: Represent output devices or actions. They are energized or de-energized based on the logical conditions represented by the contacts.
- Rungs: Each horizontal row in the ladder diagram is called a rung. A rung typically represents one control operation or logic statement.
- Power Rails: The vertical lines on the sides of the diagram are the power rails. They represent the electrical power source for the control system.
- Common Elements: - Timers and Counters: Ladder Diagrams often include timers and counters for handling time-dependent or counting operations. - Comparators and Math Operations: Some PLCs support mathematical operations and comparators for more advanced control logic.
- Application Areas: Ladder Diagrams are commonly used in manufacturing and industrial settings for programming PLCs to control various processes, machinery, and equipment.
- Ease of Understanding: Ladder Diagrams are known for their simplicity and ease of understanding, especially by individuals with a background in electrical engineering or traditional relay control systems.

- Programming Style: Ladder programming style is derived from relay logic control systems, making it familiar to those who have experience with relay-based control systems.

Following the Ladder Diagram, obtain from the Sequential Function Chart, is shown:

First segment(Inizialization)

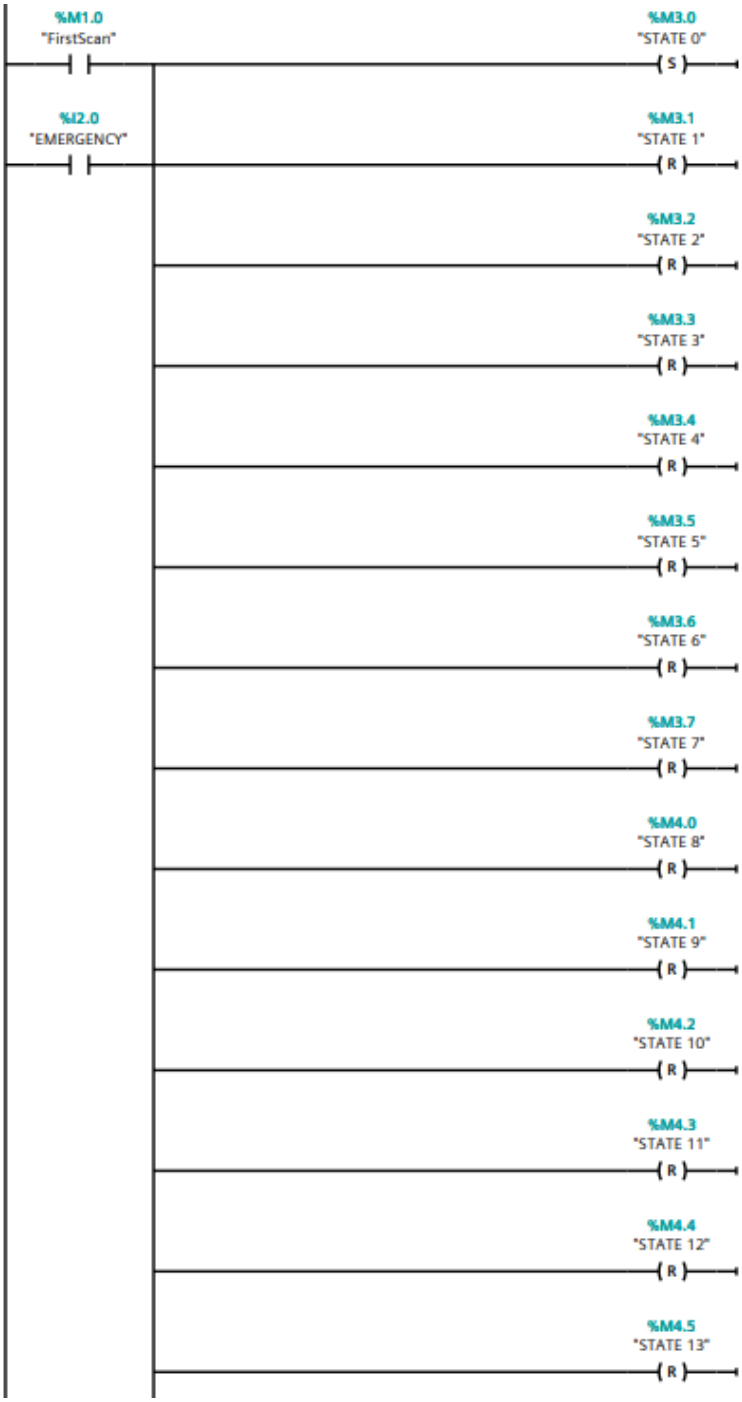

Figure 1.7

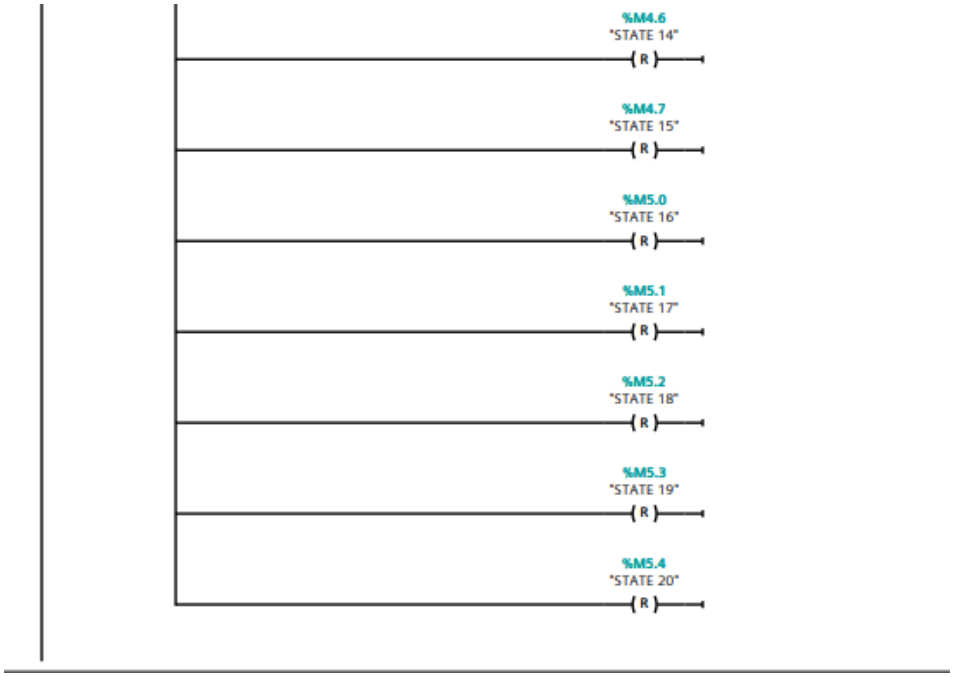

Figure 1.8

Second segment(State transitions)

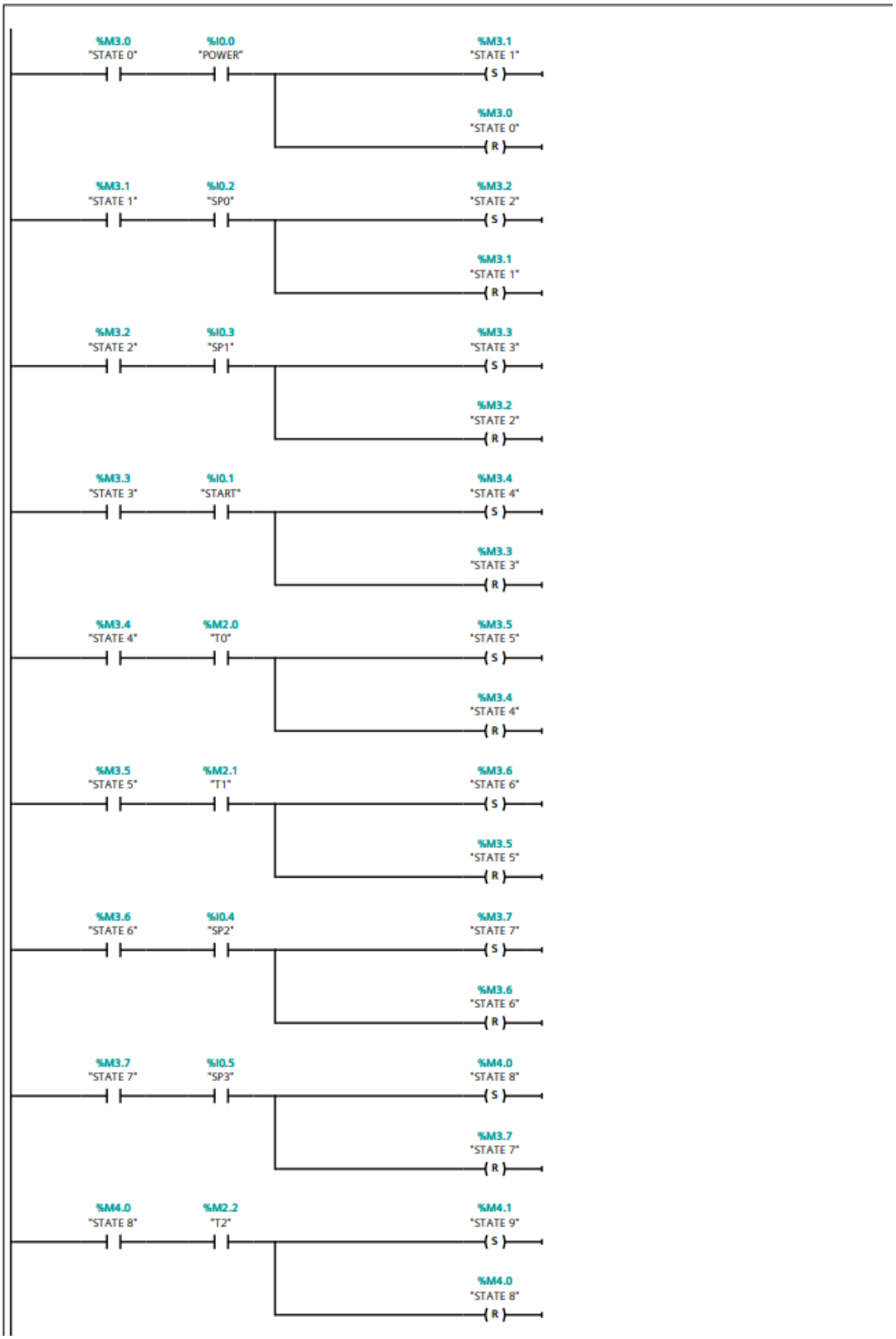
%M3.0
"STATE 0"
%I0.0
"POWER"
%M3.1
"STATE 1"
S
%M3.0
"STATE 0"
R
%M3.1
"STATE 1"
%I0.2
"SP0"
%M3.2
"STATE 2"
S
%M3.1
"STATE 1"
R
%M3.2
"STATE 2"
%I0.3
"SP1"
%M3.3
"STATE 3"
S
%M3.2
"STATE 2"
R
%M3.3
"STATE 3"
%I0.1
"START"
%M3.4
"STATE 4"
S
%M3.3
"STATE 3"
R
%M3.4
"STATE 4"
%M2.0
"T0"
%M3.5
"STATE 5"
S
%M3.4
"STATE 4"
R
%M3.5
"STATE 5"
%M2.1
"T1"
%M3.6
"STATE 6"
S
%M3.5
"STATE 5"
R
%M3.6
"STATE 6"
%I0.4
"SP2"
%M3.7
"STATE 7"
S
%M3.6
"STATE 6"
R
%M3.7
"STATE 7"
%I0.5
"SP3"
%M4.0
"STATE 8"
S
%M3.7
"STATE 7"
R
%M4.0
"STATE 8"
%M2.2
"T2"
%M4.1
"STATE 9"
S
%M4.0
"STATE 8"
R

Figure 1.9

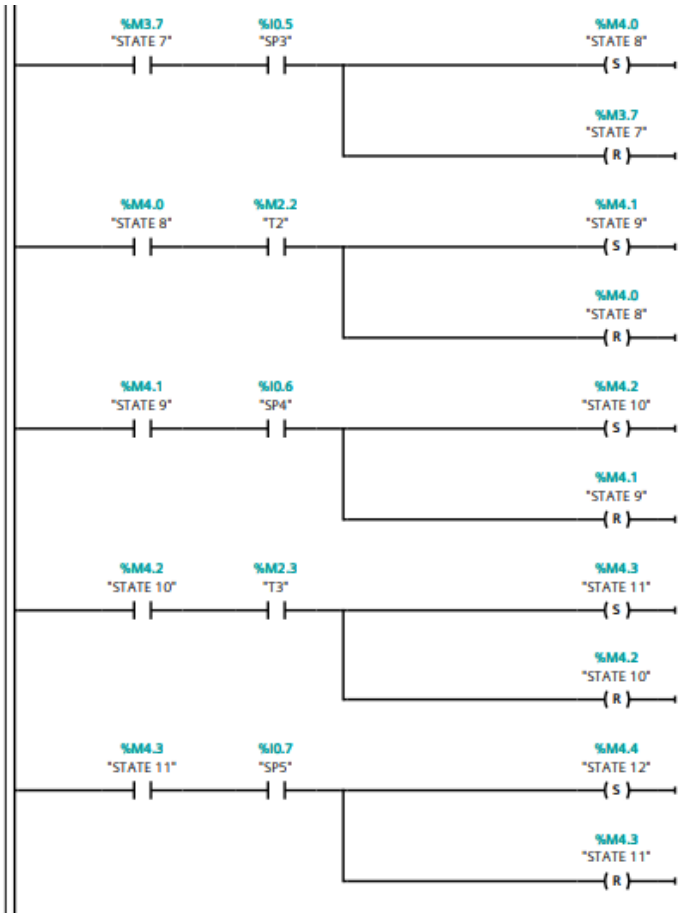

Figure 1.10

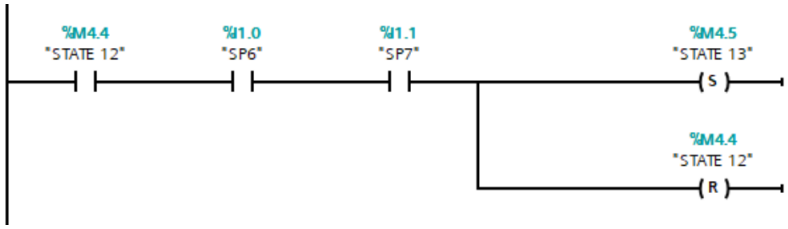

Figure 1.11

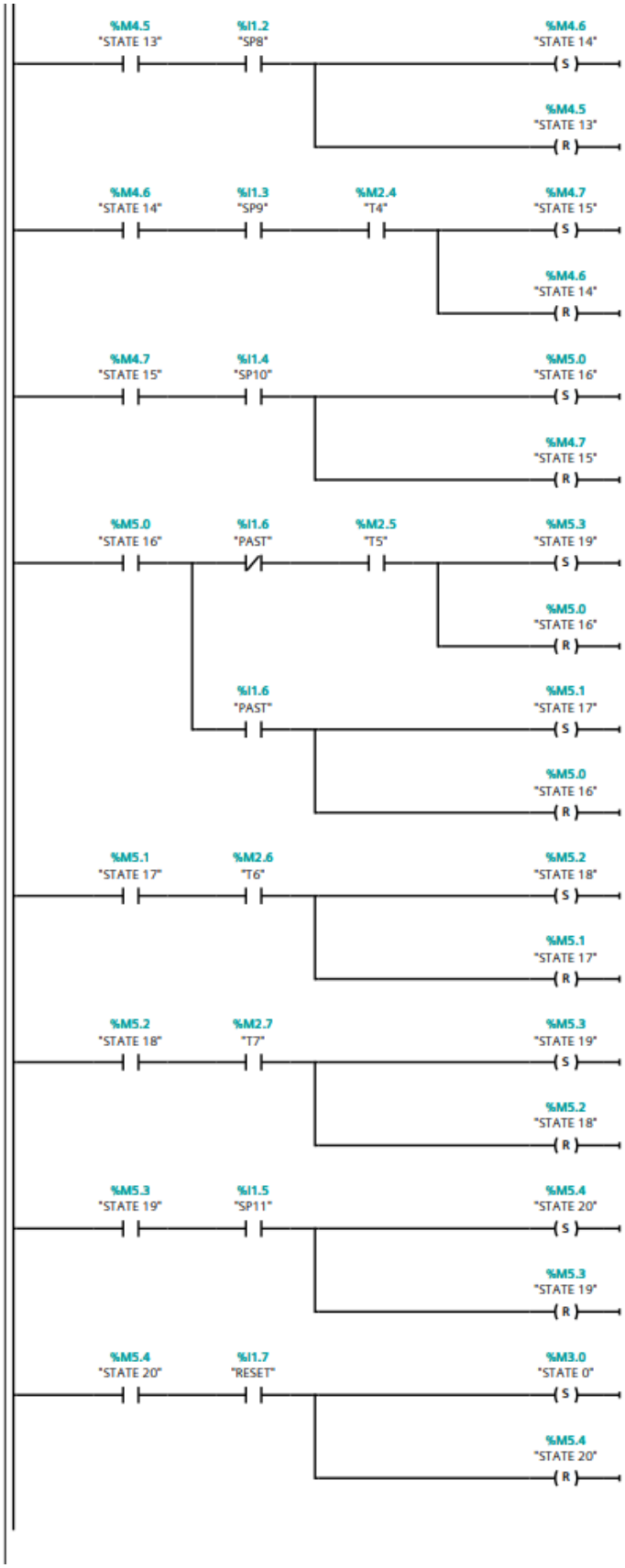
%M4.5
"STATE 13"
%I1.2
"SP8"
%M4.6
"STATE 14"
S
%M4.5
"STATE 13"
R
%M4.6
"STATE 14"
%I1.3
"SP9"
%M2.4
"T4"
%M4.7
"STATE 15"
S
%M4.6
"STATE 14"
R
%M4.7
"STATE 15"
%I1.4
"SP10"
%M5.0
"STATE 16"
S
%M4.7
"STATE 15"
R
%M5.0
"STATE 16"
%I1.6
"PAST"
%M2.5
"T5"
%M5.3
"STATE 19"
S
%M5.0
"STATE 16"
R
%I1.6
"PAST"
%M5.1
"STATE 17"
S
%M5.0
"STATE 16"
R
%M5.1
"STATE 17"
%M2.6
"T6"
%M5.2
"STATE 18"
S
%M5.1
"STATE 17"
R
%M5.2
"STATE 18"
%M2.7
"T7"
%M5.3
"STATE 19"
S
%M5.2
"STATE 18"
R
%M5.3
"STATE 19"
%I1.5
"SP11"
%M5.4
"STATE 20"
S
%M5.3
"STATE 19"
R
%M5.4
"STATE 20"
%I1.7
"RESET"
%M3.0
"STATE 0"
S
%M5.4
"STATE 20"
R

Figure 1.12

Third segment(Commands)

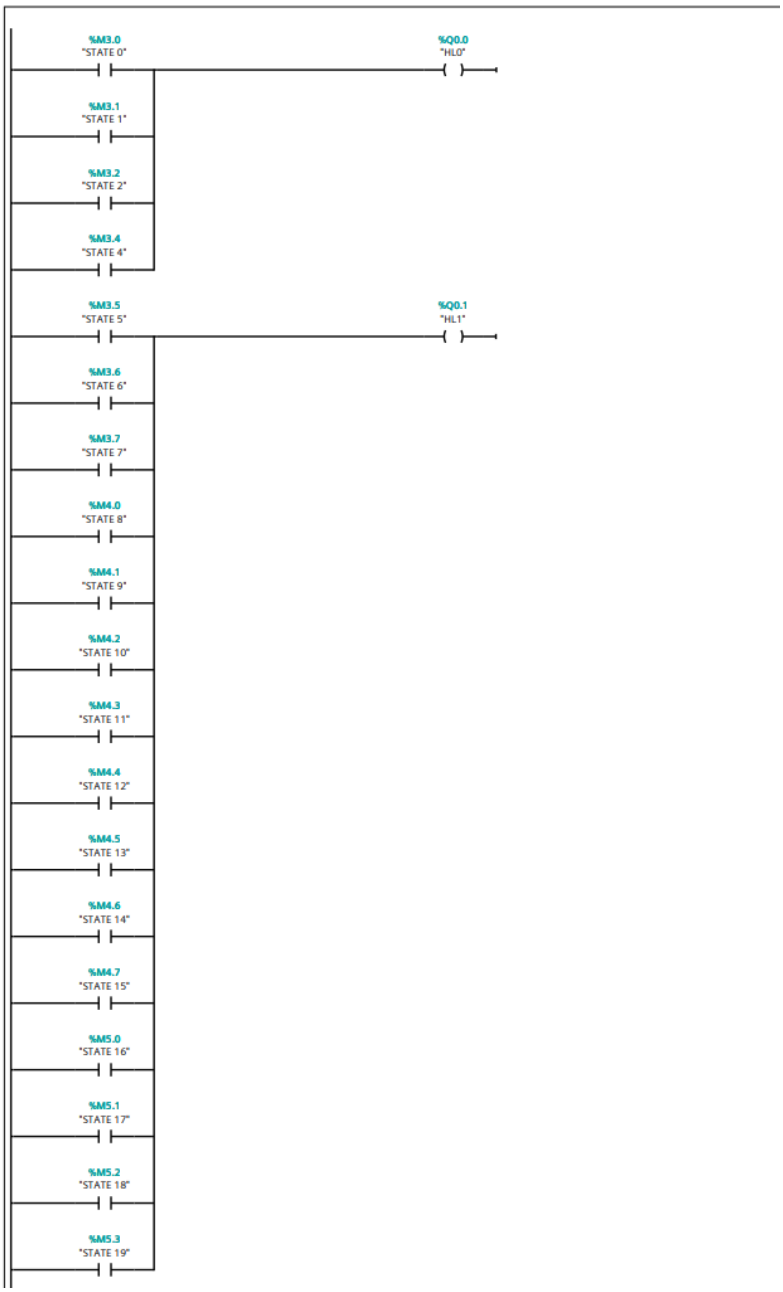

Figure 1.13

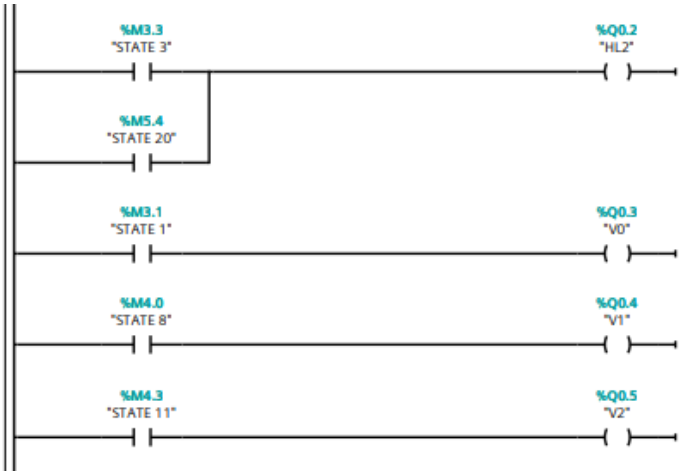

Figure 1.14

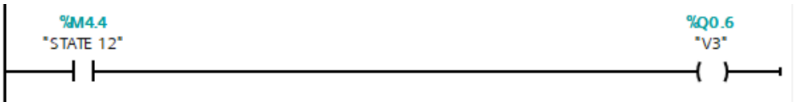

Figure 1.15

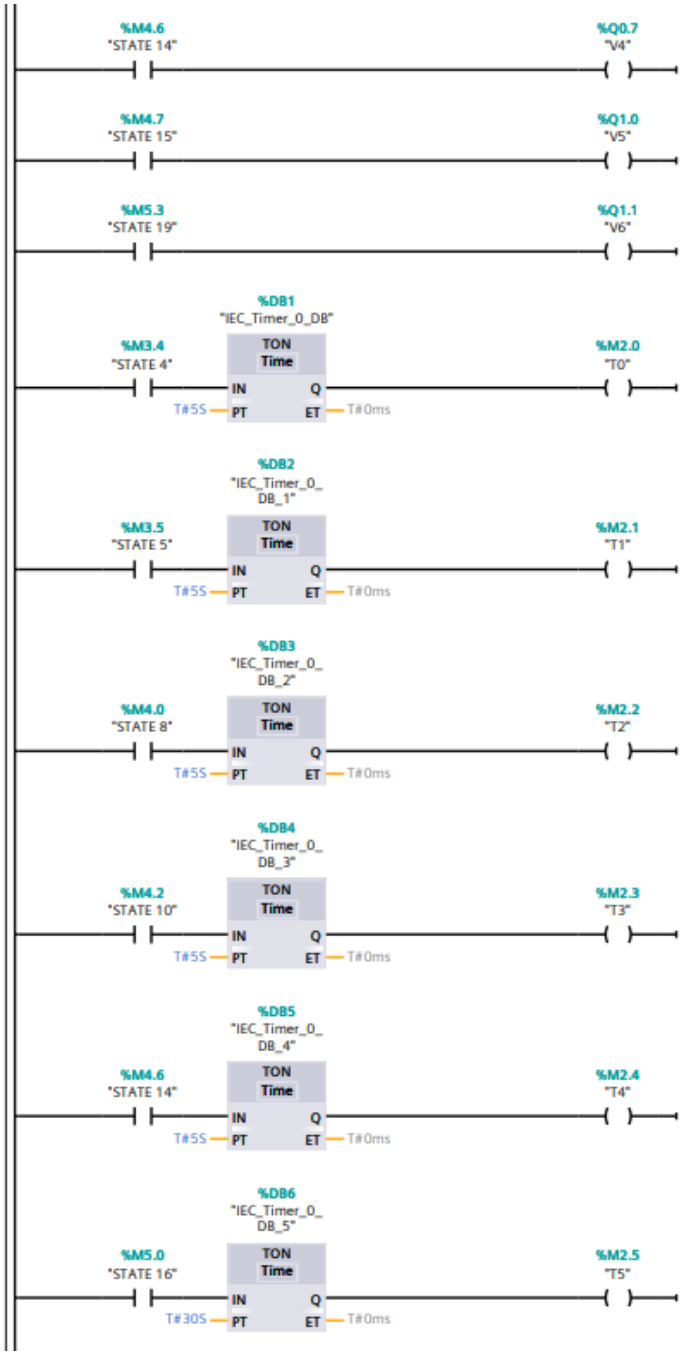
%M4.6
"STATE 14"
%Q0.7
"V4"
%M4.7
"STATE 15"
%Q1.0
"V5"
%M5.3
"STATE 19"
%Q1.1
"V6"
%DB1
"IEC_Timer_0_DB"
%M3.4
"STATE 4"
TON
Time
IN
Q
T#5S
PT
ET
T#0ms
%M2.0
"T0"
%DB2
"IEC_Timer_0_DB_1"
%M3.5
"STATE 5"
%M2.1
"T1"
%DB3
"IEC_Timer_0_DB_2"
%M4.0
"STATE 8"
%M2.2
"T2"
%DB4
"IEC_Timer_0_DB_3"
%M4.2
"STATE 10"
%M2.3
"T3"
%DB5
"IEC_Timer_0_DB_4"
%M4.6
"STATE 14"
%M2.4
"T4"
%DB6
"IEC_Timer_0_DB_5"
%M5.0
"STATE 16"
T#30S
%M2.5
"T5"

Figure 1.16

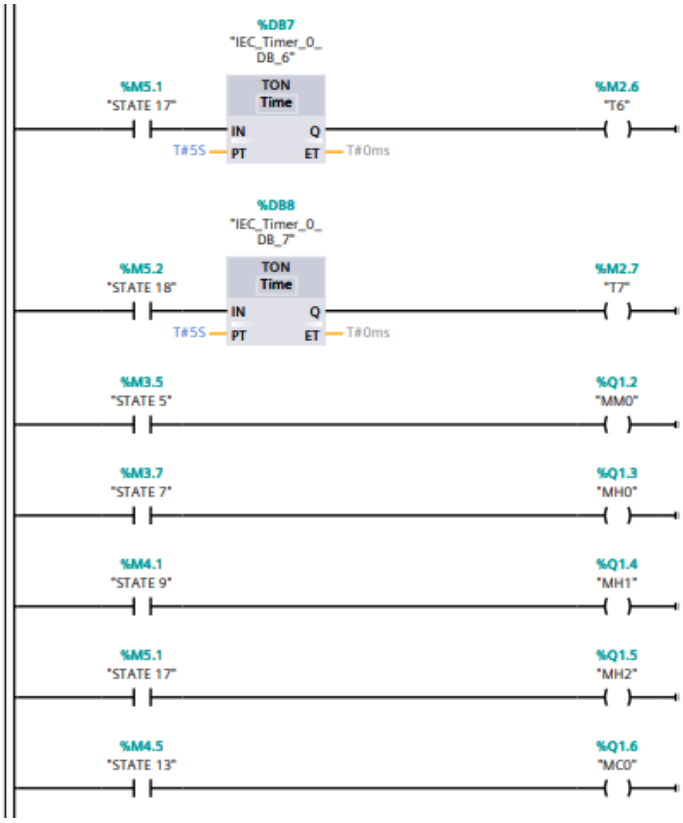

Figure 1.17

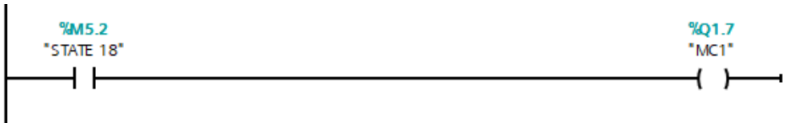

Figure 1.18

1.1.5 Component's List

The following tables contains the list of all the hardware components used for this project:

Label	Type	Quantity	Price	Description	Image
SP0 SP7 SP11	Level Sensor	x3	5.18€ x1	Water Sensor water level sensor is an easy-to-use, cost-effective high level/drop recognition sensor, which is obtained by having a series of parallel wires exposed traces measured droplets/water volume in order to determine the water level. Easy to complete water to analog signal conversion and output analog values can be directly read Arduino development board to achieve the level alarm effect.	
SP2 SP5 SP6	Proximity Sensor	x3	7.99€ x1	A proximity sensor detects the presence or absence of an object without physical contact using inductive, capacitive, optical, or ultrasonic technologies. The detection range varies from millimeters to meters, and the accuracy and response time depends on the technology used. They are designed to withstand environmental interference such as dust and temperature variations, and offer analog, digital and relay outputs.	
SP1 SP3 SP4 SP8	Temperature Sensor	x4	5.13€ x4	The LM35 is an analog temperature sensor that provides an output voltage proportional to the temperature in degrees Celsius. Work over a temperature range from -55°C to +150°C. Output voltage: 10mV for each degree Celsius. Power supply: 4V to 30V.	
SP9	CO2 Sensor	x1	51€ x1	A CO2 sensor measures the concentration of carbon dioxide in the air using primarily non-dispersive infrared (NDIR) technology or chemical-electrochemical sensors. These sensors have a measurement range that can be up to several thousand parts per million (ppm) and offer an accuracy of about ±30 ppm or ±3% of the reading. Response time varies from a few seconds to minutes, and long-term stability ensures minimal drift.	
SP10	Turbidity Sensor	x1	27.09 €	The turbidity sensor adopts the optical principle to determine the turbidity of the liquid solution by combining the transmittance and the dispersion rate of the liquid solution. The sensor module adopts high-quality electronic components to ensure the accuracy of turbidity measurement.	

1.1.6 Programmable Logic Controller

PLCs (Programmable Logic Controllers) are fundamental devices in the field of industrial automation. A PLC is a real-time microprocessor-based system that implements functions such as logic, sequencing, timing, counting, and arithmetic to perform control tasks. It continuously monitors the status of input devices and makes decisions based on a customized program to control the status of output devices. A PLC contains a microcontroller, or microprocessor-based control, with special interface circuitry designed to operate in harsh industrial environments. These circuits allow the PLC to interact with a wide range of industrial sensors and actuators. PLCs are economical for controlling complex systems and are flexible, allowing them to be easily expanded and reapplied to control other systems quickly and simply. This flexibility is essential to adapt to changes in production processes. Thanks to their computational capabilities, PLCs can perform sophisticated controls, managing complex operations that require precision and reliability. Programming PLCs is relatively simple and reduces downtime. Ease of use allows engineers to develop and modify control programs without requiring lengthy training sessions. PLC components are highly reliable, making them capable of operating for years without failure. This reliability is crucial in industrial environments where downtime can be very costly. Early PLCs were programmed using a technique based on relay logic wiring diagrams, known as ladder diagrams. This programming method is intuitive for electrical engineers and eases the transition from relay-based controls to PLC-based systems. An input can come from a switch or any other type of sensor. An output will be a device outside of the PLC that is turned on or off, such as lights or motors.

1.1.7 TIA Portal

TIA Portal, which stands for Totally Integrated Automation Portal, is a software platform developed by Siemens for the design, programming, and commissioning of industrial automation systems. This platform is primarily used with PLCs but also supports other Siemens automation devices and technologies.

Here are some key features of TIA Portal:

- Complete Integration: TIA Portal provides an integrated development environment that covers the entire life cycle of an automation project, from design to commissioning. This means that engineers can work on all project phases within a single platform
- Support for Various Devices: TIA Portal is not limited to PLCs but also supports other Siemens automation devices such as drive systems, human-machine interfaces (HMIs), sensors, and more. This simplifies the integration and programming of a wide range of automation components
- Programming Languages: TIA Portal supports various programming languages commonly used in the field of industrial automation. These include ladder logic (LADDER), structured text (ST), function block diagram (FBD), sequential function chart (SCL), and others
- Simulation and Diagnostics: TIA Portal provides simulation tools that allow engineers to test and verify the operation of their program before implementing it physically in the field. Additionally, it offers advanced diagnostic features to facilitate issue resolution
- Intuitive User Interface: The platform is designed with an intuitive user interface that simplifies navigation and use by engineers and operators
- Integrated Communication: TIA Portal facilitates communication between different devices and systems within an industrial installation, promoting process integration and operational optimization

TIA Portal is widely used in the industry for its ability to simplify and expedite the development, programming, and maintenance of automation systems

1.1.8 PLC Variables

Below, all the tables provided by TIA Portal containing the variables used in the PLC will be displayed.

	Nome	Tabella delle variabili	Tipo di dati	Indirizzo	Ritenz...	Acces...	Scrivi...	Visibil...
1	POWER	Tabella delle variabi..	Bool	%I0.0	☐	☑	☑	☑
2	START	Tabella delle variabi..	Bool	%I0.1	☐	☑	☑	☑
3	SP0	Tabella delle variabi..	Bool	%I0.2	☐	☑	☑	☑
4	SP1	Tabella delle variabi..	Bool	%I0.3	☐	☑	☑	☑
5	SP2	Tabella delle variabi..	Bool	%I0.4	☐	☑	☑	☑
6	SP3	Tabella delle variabi..	Bool	%I0.5	☐	☑	☑	☑
7	SP4	Tabella delle variabi..	Bool	%I0.6	☐	☑	☑	☑
8	SP5	Tabella delle variabi..	Bool	%I0.7	☐	☑	☑	☑
9	SP6	Tabella delle variabi..	Bool	%I1.0	☐	☑	☑	☑
10	SP7	Tabella delle variabi..	Bool	%I1.1	☐	☑	☑	☑
11	SP8	Tabella delle variabi..	Bool	%I1.2	☐	☑	☑	☑
12	SP9	Tabella delle variabi..	Bool	%I1.3	☐	☑	☑	☑
13	SP10	Tabella delle variabi..	Bool	%I1.4	☐	☑	☑	☑
14	SP11	Tabella delle variabi..	Bool	%I1.5	☐	☑	☑	☑
15	PAST	Tabella delle variabi..	Bool	%I1.6	☐	☑	☑	☑
16	RESET	Tabella delle variabi..	Bool	%I1.7	☐	☑	☑	☑
17	EMERGENCY	Tabella delle variabi..	Bool	%I2.0	☐	☑	☑	☑

Figure 1.19: Inputs variables table

18	HL0	Tabella delle variabi..	Bool	%Q0.0	☐	☑	☑	☑
19	HL1	Tabella delle variabi..	Bool	%Q0.1	☐	☑	☑	☑
20	HL2	Tabella delle variabi..	Bool	%Q0.2	☐	☑	☑	☑
21	V0	Tabella delle variabi..	Bool	%Q0.3	☐	☑	☑	☑
22	V1	Tabella delle variabi..	Bool	%Q0.4	☐	☑	☑	☑
23	V2	Tabella delle variabi..	Bool	%Q0.5	☐	☑	☑	☑
24	V3	Tabella delle variabi..	Bool	%Q0.6	☐	☑	☑	☑
25	V4	Tabella delle variabi..	Bool	%Q0.7	☐	☑	☑	☑
26	V5	Tabella delle variabi..	Bool	%Q1.0	☐	☑	☑	☑
27	V6	Tabella delle variabi..	Bool	%Q1.1	☐	☑	☑	☑
28	MM0	Tabella delle variabi..	Bool	%Q1.2	☐	☑	☑	☑
29	MH0	Tabella delle variabi..	Bool	%Q1.3	☐	☑	☑	☑
30	MH1	Tabella delle variabi..	Bool	%Q1.4	☐	☑	☑	☑
31	MH2	Tabella delle variabi..	Bool	%Q1.5	☐	☑	☑	☑
32	MC0	Tabella delle variabi..	Bool	%Q1.6	☐	☑	☑	☑
33	MC1	Tabella delle variabi..	Bool	%Q1.7	☐	☑	☑	☑

Figure 1.20: Outputs variables table

56		STATE 0	Tabella delle variabi..	Bool	%M3.0	☐	☑	☑	☑
57		STATE 1	Tabella delle variabi..	Bool	%M3.1	☐	☑	☑	☑
58		STATE 2	Tabella delle variabi..	Bool	%M3.2	☐	☑	☑	☑
59		STATE 3	Tabella delle variabi..	Bool	%M3.3	☐	☑	☑	☑
60		STATE 4	Tabella delle variabi..	Bool	%M3.4	☐	☑	☑	☑
61		STATE 5	Tabella delle variabi..	Bool	%M3.5	☐	☑	☑	☑
62		STATE 6	Tabella delle variabi..	Bool	%M3.6	☐	☑	☑	☑
63		STATE 7	Tabella delle variabi..	Bool	%M3.7	☐	☑	☑	☑
64		STATE 8	Tabella delle variabi..	Bool	%M4.0	☐	☑	☑	☑
65		STATE 9	Tabella delle variabi..	Bool	%M4.1	☐	☑	☑	☑
66		STATE 10	Tabella delle variabi..	Bool	%M4.2	☐	☑	☑	☑
67		STATE 11	Tabella delle variabi..	Bool	%M4.3	☐	☑	☑	☑
68		STATE 12	Tabella delle variabi..	Bool	%M4.4	☐	☑	☑	☑
69		STATE 13	Tabella delle variabi..	Bool	%M4.5	☐	☑	☑	☑
70		STATE 14	Tabella delle variabi..	Bool	%M4.6	☐	☑	☑	☑
71		STATE 15	Tabella delle variabi..	Bool	%M4.7	☐	☑	☑	☑
72		STATE 16	Tabella delle variabi..	Bool	%M5.0	☐	☑	☑	☑
73		STATE 17	Tabella delle variabi..	Bool	%M5.1	☐	☑	☑	☑
74		STATE 18	Tabella delle variabi..	Bool	%M5.2	☐	☑	☑	☑
75		STATE 19	Tabella delle variabi..	Bool	%M5.3	☐	☑	☑	☑
76		STATE 20	Tabella delle variabi..	Bool	%M5.4	☐	☑	☑	☑

Figure 1.21: States variables table

48		T0	Tabella delle variabi..	Bool	%M2.0	☐	☑	☑	☑
49		T1	Tabella delle variabi..	Bool	%M2.1	☐	☑	☑	☑
50		T2	Tabella delle variabi..	Bool	%M2.2	☐	☑	☑	☑
51		T3	Tabella delle variabi..	Bool	%M2.3	☐	☑	☑	☑
52		T4	Tabella delle variabi..	Bool	%M2.4	☐	☑	☑	☑
53		T5	Tabella delle variabi..	Bool	%M2.5	☐	☑	☑	☑
54		T6	Tabella delle variabi..	Bool	%M2.6	☐	☑	☑	☑
55		T7	Tabella delle variabi..	Bool	%M2.7	☐	☑	☑	☑

Figure 1.22: Timer variables table

Chapter 2

Robotic's part

2.1 CoppeliaSim

In the second part of the project, we focused on the implementation of an autonomous driving system. It was therefore decided as the robotics program to use CoppeliaSim, because it seemed the most suitable for our needs. CoppeliaSim, previously known as V-REP (Virtual Robot Experimentation Platform), is a versatile and powerful robot simulation software. It's widely used in research, education, and industrial applications for simulating robotic systems and environments.

2.2 Self-driving car

A self-driving car system, also known as an autonomous vehicle (AV) system, is a sophisticated technological setup designed to enable a vehicle to operate without human intervention. These systems rely on a combination of hardware and software components to perceive the environment, make decisions, and control the vehicle's movement.

2.2.1 Design Phase

First, the self-drive problem was schematized in a model drawn in the shape of a checkerboard. Therefore, the outer boundary, characterized

by the wall surrounding the perimento, was first included. Secondly, "20cmHightWall50cm" 50 centimeters in length and "20cmHightWall100cm" 100 centimeters in length were chosen as object walls. Therefore, different routes that the car can take were identified and diversified.

The main auto object chosen is the LineTracer:

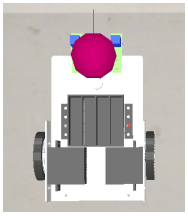

Figure 2.1: LineTracer Vehicle

The LineTracer vehicle in CoppeliaSim is a simulated robot designed to follow a line, typically used in educational and research contexts to demonstrate basic robotic control and sensor integration. This vehicle is composed of several key components that work together to allow it to detect and follow a line on the ground. Here's a detailed breakdown of its composition:

- 2 Wheels
- 3 Front sensors
- 1 Front proximity sensor (added on top)

The LineTracer mobile vehicle, unlike other vehicles used, such as the Pioneer, does not suffer from motor instability. Thanks to its two fixed side wheels, it enjoys considerable stability during the simulation without crashing into walls. A proximity sensor was later added above the vehicle that is directed towards the front and detects obstacles at 0.10 metres.

2.2.2 Self-drive logic

In the programme's init, the lane remains stationary, waiting for the user, via the graphical interface, to set the start and end point. When "start

simulation" is clicked, the simulation will begin. The car will then be positioned at the starting point with a consistent direction in search of the next point. The shortest path is then calculated, using the Dijkstra algorithm, which will return a list of points to be travelled. So the car, knowing the current point and the successful one, verifies the direction it must take. The Dijstra algorithm returns the points of the shortest route knowing the adjacencies between the various points. A green-coloured path is also drawn, indicating the current route the machine is travelling.

We set up two walls along the route. If the car, via its proximity sensor, detects a wall in front it will act accordingly:

1. Retro to previous point
2. Removes the adjacencies between the two points where there is the middle wall
3. Recalculate Dijkstra
4. Draw the new path

Rotation wheels logic

To create the rotation wheels logic when the machine has to turn, we used some basic parameters, such as the current and next point, which inserted in a function called "getNextOrientation", allowed us to derive the next orientation of the machine in terms of angles and labels. From here by comparing the label with the current and next orientation, we were able to understand where the machine had to turn, whether to the right or left. If the machine turn to the right, the speed of the wheels will be:

- left wheel = 0.8
- right wheel = 0.1

If the machine turn to the left, the speed of the wheels will be:

- left wheel = 0.1
- right wheel = 0.8

Once the turning process has started, we compare the current orientation of the machine, in terms of angles, with the next one previously calculated with the "getNextOrientation" function and as soon as the orientation is approximately equal, with a margin of error of 0.55, via a function called "checkOrientationRange", we will set the wheels again at the same speed so that the machine advances in the next direction.

Wall detected logic

To implement the wall detected logic, we have added, as previously introduced, a proximity sensor in order to detect a possible wall in front of its path. Through an API called "sim.handleProximitySensor", we get a binary value that tells us whether or not the wall has been detected. If the first case occurs, we change the mark of the wheels, the car in reverse goes back to the previous point and once detected, it eliminates the adeciences where the wall is located and calculates the new path. Once the new path is calculated, we apply the logic of wheel rotation explained above.

2.2.3 Simulation

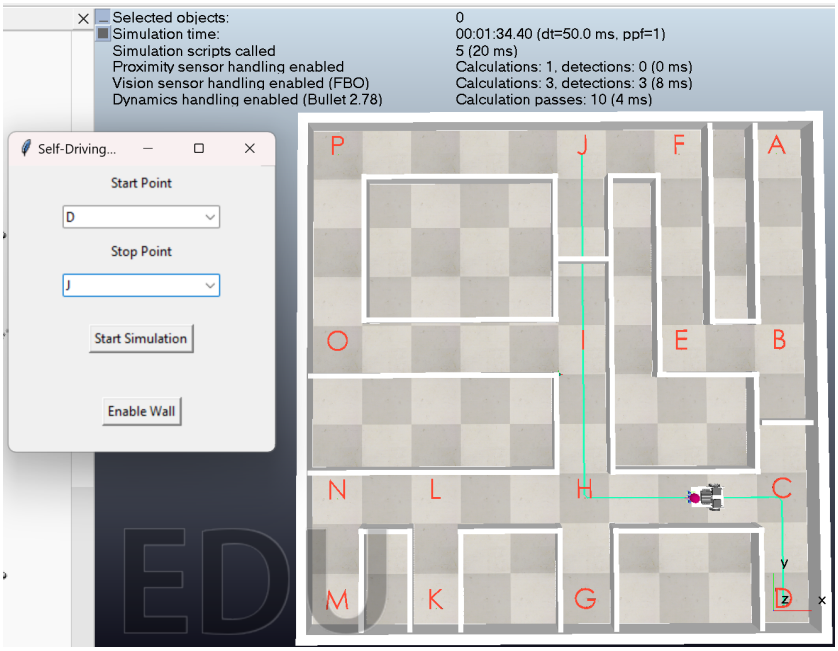

Figure 2.2: Initial path from "D" to "J"

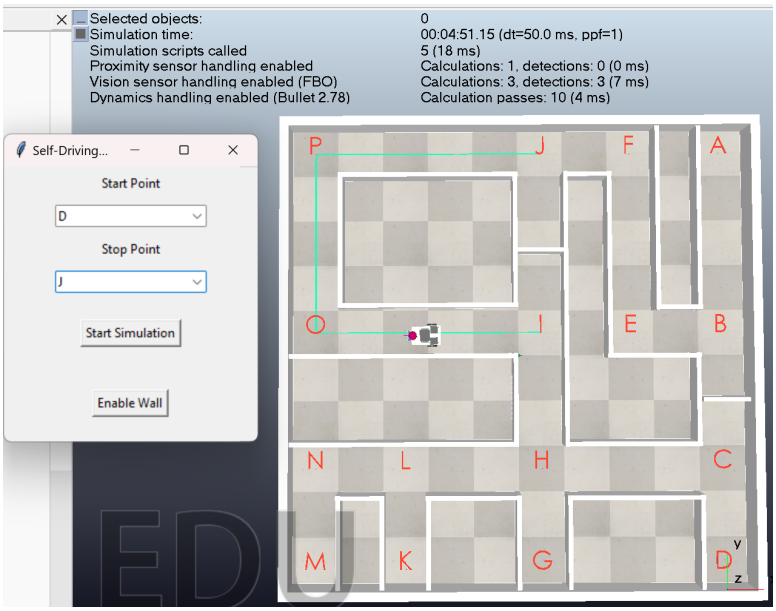

Figure 2.3: Recalculated path from "D" to "J" due to a wall detection between "I" and "J"

2.3 Data saving

During the simulation, useful informations are saved in a CSV file.
The data that are stored concerns:

- The precedent point
- The next point to be reached
- Car position coordinates
- Car orientation coordinates
- Car next orientation
- Event triggered

	PrecPoint	NextPoint	Position	Orientation	NextOrientation	Event
1	D	C	[2.25064, -1.31721, 0.05414]	[1.57369, -0.00048, 1.57015]	[90.0, 0, 90.0]	Point acrossed
2	C	B	[2.25054, -0.71438, 0.05418]	[1.57313, 0.00105, 1.57026]	[90.0, 0, 90.0]	Wall detected
3	C	H	[0.32954, -1.27153, 0.05417]	[-1.94382, 1.56382, -1.19741]	[90, 90, 90.0]	Point acrossed
4	H	I	[0.2472, 0.17866, 0.05414]	[1.57363, -0.00255, 1.57087]	[90.0, 0, 90.0]	Point acrossed
5	I	J	[0.24855, 0.9083, 0.05418]	[1.57309, -0.0017, 1.57096]	[90.0, 0, 90.0]	Wall detected
6	I	O	[-2.1806, 0.18819, 0.05412]	[-1.75125, 1.55363, -1.39043]	[90, 90, 90.0]	Point acrossed
7	O	P	[-2.26303, 2.18129, 0.05415]	[1.57258, -0.00503, 1.57036]	[90.0, 0, 90.0]	Point acrossed
8	P	J	[0.17929, 2.25951, 0.05421]	[-2.1352, -1.56709, -2.1353]	[90, -90, 90]	Point acrossed
9	J	F	[1.17858, 2.25652, 0.05421]	[0.0, -1.56801, -0.00012]	[90, -90, 90]	Point acrossed
10	F	E	[1.27285, 0.32523, 0.05408]	[-1.57359, -0.00288, -1.57108]	[-90.0, 0, -90.0]	Point acrossed
11	E	B	[2.17053, 0.23545, 0.0541]	[-2.28054, -1.56705, -2.28073]	[90, -90, 90]	Point acrossed
12	E	B	[2.17053, 0.23545, 0.0541]	[-2.28054, -1.56705, -2.28073]	[90, -90, 90]	Final destination reached

Figure 2.4: CSV file generated at the end of simulation

These data allow us to draw statistical evaluations of vehicle behaviour given a start and end point chosen by the user. In this way we can understand possible improvements to the movement algorithm.

2.4 Python Code

The following code was placed inside an "associated child threaded" file, so the code is executed asynchronously to the main simulator code. This makes it possible to have a GUI that is separate from the code, i.e. that does not wait for the main code to be executed. So, asynchronism is useful for not blocking the code. This code implements two drop-downs containing the dot letters "A" through "P". Before the simulation starts, the user is made to choose the start and end point (a check is made to ensure that the letters are not the same). When the 'Start Simulation' button is pressed, the actuation function in the main file is enabled to run, so the simulation begins. A second button has been added, namely "randomise wall", which makes a random wall visible but between the path chosen by the user. This last function has not been implemented, we will talk about it in future work section 3.1

LineTracer customized child threaded python script:

```
#python

import tkinter as tk
from tkinter import ttk

combo1_value = ''
combo2_value = ''


def startSimulation():
    global combo1_value, combo2_value
    if combo1_value != combo2_value:
        sim.setStringSignal("startPoint", combo1_value)
        sim.setStringSignal("stopPoint", combo2_value)
        sim.setStringSignal("startSimulation", "True")
        print(">> Simulation started")
        print(f">> [StartPoint: {combo1_value} | StopPoint: {
   combo2_value}]")
    else:
        print(">> [!] You cannot choose same start and stop points")


def enableWall():
    sim.setStringSignal("enableRandomWall", "True")
    print(">> Random wall inserted")


def update_combo1_value(*args):
```

```
28      global combo1_value
29      combo1_value = combo1_var.get()
30
31
32  def update_combo2_value(*args):
33      global combo2_value
34      combo2_value = combo2_var.get()
35
36
37  def loadInterface():
38      global combo1_var, combo2_var
39      root = tk.Tk()
40      root.title("Self-Driving Tool")
41      root.geometry("400x260")
42
43      label1 = tk.Label(root, text="Start Point")
44      label1.pack(pady=5)
45
46      combo1_var = tk.StringVar()
47      combo1 = ttk.Combobox(root, textvariable=combo1_var, values=[chr(i)
        for i in range(ord('A'), ord('P') + 1)])
48      combo1.pack(pady=5)
49      combo1_var.trace('w', update_combo1_value)
50
51      label2 = tk.Label(root, text="Stop Point")
52      label2.pack(pady=5)
53      combo2_var = tk.StringVar()
54      combo2 = ttk.Combobox(root, textvariable=combo2_var, values=[chr(i)
        for i in range(ord('A'), ord('P') + 1)])
55      combo2.pack(pady=5)
56      combo2_var.trace('w', update_combo2_value)
57
58      button1 = tk.Button(root, text="Start Simulation", command=
       startSimulation)
59      button1.pack(pady=20)
60      button2 = tk.Button(root, text="Enable Wall", command=enableWall)
61      button2.pack(pady=20)
62
63      root.mainloop()
64
65
66  def sysCall_init():
67      sim = require('sim')
68      sim.setStringSignal("startSimulation", "False")
69      sim.setStringSignal("enableRandomWall", "False")
70
71
72  def sysCall_thread():
73      loadInterface()
```

LineTracer customized associated non-threaded python script:

```python
#python

import math
import heapq
import csv


def shortest_paths(adjacencies, dummies, start, stop):
    def dijkstra(start, stop):
        distances = {point: float('inf') for point in adjacencies}
        distances[start] = 0
        queue = [(0, start)]
        previous = {point: None for point in adjacencies}

        while queue:
            current_distance, current_point = heapq.heappop(queue)
            if current_point == stop:
                break

            for neighbor in adjacencies[current_point]:
                new_distance = current_distance + distance(dummies[current_point], dummies[neighbor])
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
                    previous[neighbor] = current_point
                    heapq.heappush(queue, (new_distance, neighbor))

        path = []
        current_point = stop
        while current_point is not None:
            path.insert(0, current_point)
            current_point = previous[current_point]

        return path, distances[stop]

    # Shortest path calculate in both directions
    path_start_to_stop, distance_start_to_stop = dijkstra(start, stop)
    path_stop_to_start, distance_stop_to_start = dijkstra(stop, start)

    if distance_start_to_stop <= distance_stop_to_start:
        return path_start_to_stop, distance_start_to_stop
    else:
        return path_stop_to_start[::-1], distance_stop_to_start


def distance(point1, point2):
    return ((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)**0.5


def loadGlobalVariables():
    # From/to point
    self.startPoint = ""
    self.stopPoint = ""

```

```
    # Vehicle properties
    self.coordZ = 0.08328
    self.velocity = 6.0

    # Static global variables
    self.pointRangeValue = 0.08

    # Global variables
    self.index = 0
    self.initFinished = False
    self.changeDirection = False
    self.destinationReached = False
    self.orientation = []
    self.newOrientation = []
    self.orientationName = ''
    self.newOrientationName = ''
    self.isChanging = False
    self.retro = False
    self.pathAlreadyRemoved = False
    self.proximitySensorEnabled = True
    self.dati = []

    # Get object handles
    self.carObject = sim.getObject(".")
    self.motorLeft = sim.getObject("./DynamicLeftJoint")
    self.motorRight = sim.getObject("./DynamicRightJoint")
    self.proximitySensor = sim.getObject("./Proximity_sensor")

    self.alpha = {
        "0": "A",
        "1": "B",
        "2": "C",
        "3": "D",
        "4": "E",
        "5": "F",
        "6": "G",
        "7": "H",
        "8": "I",
        "9": "J",
        "10": "H",
        "11": "L",
        "12": "M",
        "13": "N",
        "14": "O",
        "15": "P"
    }

    self.adjacencies = {
        "A": ["B"],
        "B": ["C", "E"],
        "C": ["B", "D", "H"],
        "D": ["C"],
        "E": ["B", "F"],
        "F": ["E", "J"],
        "G": ["H"],
        "H": ["C", "G", "I", "L"],
        "I": ["H", "O", "J"],
        "J": ["F", "I", "P"],
```

```
        "K": ["L"],
        "L": ["H", "K", "N"],
        "M": ["N"],
        "N": ["M", "L"],
        "O": ["I", "P"],
        "P": ["O", "J"]
    }

    self.dummies = {
        "A": [2.250, 2.250],
        "B": [2.250, 0.250],
        "C": [2.250, -1.250],
        "D": [2.250, -2.350],
        "E": [1.250, 0.250],
        "F": [1.250, 2.250],
        "G": [0.250, -2.350],
        "H": [0.250, -1.250],
        "I": [0.250, 0.250],
        "J": [0.250, 2.250],
        "K": [-1.250, -2.350],
        "L": [-1.250, -1.250],
        "M": [-2.250, -2.350],
        "N": [-2.250, -1.250],
        "O": [-2.250, 0.250],
        "P": [-2.250, 2.250]
    }

    # Initial conditions
    setVelocity(0.0, 0.0)


def calculateDijkstra():
    self.points, self.distances = shortest_paths(self.adjacencies, self.dummies, self.startPoint, self.stopPoint)

    # Initial vehicles properties (position and velocity)
    initial_x = self.dummies[self.points[0]][0]
    initial_y = self.dummies[self.points[0]][1]
    first_point_x = self.dummies[self.points[1]][0]
    first_point_y = self.dummies[self.points[1]][1]

    self.initialPosition = [initial_x, initial_y, self.coordZ]
    setPosition(self.carObject, self.initialPosition)
    self.orientation = getNextOrientation([initial_x, initial_y], [first_point_x, first_point_y])[0]
    self.orientationName = getNextOrientation([initial_x, initial_y], [first_point_x, first_point_y])[1]

    setOrientation(self.carObject, self.orientation)


def getPosition(object):
    [x, y, z] = sim.getObjectPosition(object)
    return [x, y, z]


def setPosition(object, positions):
    sim.setObjectPosition(object, positions)
```

```
167
168
169 def getVelocity(object):
170     velocity = sim.getJointTargetVelocity(object)
171     return velocity
172
173
174 def setVelocity(vLeft, vRight):
175     sim.setJointTargetVelocity(self.motorLeft, vLeft)
176     sim.setJointTargetVelocity(self.motorRight, vRight)
177
178
179 def getOrientation(object):
180     orientation = sim.getObjectOrientation(object, -1)
181     return orientation
182
183
184 def setOrientation(object, angle):
185     orientation = getOrientation(object)
186     new_orientation = [math.radians(angle[0]), math.radians(angle[1]),
    math.radians(angle[2])]
187     sim.setObjectOrientation(object, -1, new_orientation)
188
189
190 def checkOrientationRange(coords, orientation):
191     if(coords[1]+0.550 > orientation[1] and coords[1] -0.550 <
    orientation[1]):
192         return True
193     else:
194         return False
195
196
197 def getNextOrientation(point, newPoint):
198     x = newPoint[0] - point[0]
199     y = newPoint[1] - point[1]
200     if(x == 0):
201         if(y > 0):
202             return [90.00, 0, 90.00], "top" # Sopra
203         else:
204             return [-90.00, 0, -90.00], "bottom" # Sotto
205     elif(y == 0):
206         if(x > 0):
207             return [90, -90, 90], "right" # Destra
208         else:
209             return [90, 90, 90.00], "left" # Sinistra
210
211
212 def oppositeOrientation(orientation):
213     opposite = {
214         "top": [-90.00, 0, -90.00],
215         "bottom": [90.00, 0, 90.00],
216         "right": [0.00, 90, 180.00],
217         "left": [0.00, -90, 0.00]
218     }
219     return opposite[orientation]
220
221
222 def checkRange(x, y, point):
```

```
223     if(x <= point[0]+self.pointRangeValue and x >=point[0]-self.
    pointRangeValue and y <= point[1]+self.pointRangeValue and y >=point
    [1]-self.pointRangeValue):
224         return True
225     else:
226         return False
227 
228 
229 def checkRotationWheels(currentOrientation, nextOrientation, velocity,
    retro):
230     if nextOrientation == currentOrientation:
231         return setVelocity(self.velocity, self.velocity)
232     elif currentOrientation == "top":
233         if nextOrientation == "left":
234             if retro:
235                 return setVelocity(-velocity, velocity)
236             else:
237                 return setVelocity(0.1, velocity)
238         else:
239             if retro:
240                 return setVelocity(velocity, -velocity)
241             else:
242                 return setVelocity(velocity, 0.1)
243     elif currentOrientation == "right":
244         if nextOrientation == "top":
245             if retro:
246                 return setVelocity(-velocity, velocity)
247             else:
248                 return setVelocity(0.1, velocity)
249         else:
250             if retro:
251                 return setVelocity(velocity, -velocity)
252             else:
253                 return setVelocity(velocity, 0.1)
254     elif currentOrientation == "bottom":
255         if nextOrientation == "right":
256             if retro:
257                 return setVelocity(-velocity, velocity)
258             else:
259                 return setVelocity(0.1, velocity)
260         else:
261             if retro:
262                 return setVelocity(velocity, -velocity)
263             else:
264                 return setVelocity(velocity, 0.1)
265     elif currentOrientation == "left":
266         if nextOrientation == "bottom":
267             if retro:
268                 return setVelocity(-velocity, velocity)
269             else:
270                 return setVelocity(0.1, velocity)
271         else:
272             if retro:
273                 return setVelocity(velocity, -velocity)
274             else:
275                 return setVelocity(velocity, 0.1)
276 
277 
```

```
278 def appendPoint(coordinates, pathPoints):
279     pathPoints.append(coordinates[0])
280     pathPoints.append(coordinates[1])
281     pathPoints.append(0)
282     pathPoints.append(0)
283     pathPoints.append(0)
284     pathPoints.append(0)
285     pathPoints.append(1)
286     return pathPoints
287 
288 
289 def generatePath():
290     pathPoints = []
291     for element in self.points:
292         coordinate = self.dummies[element]
293         appendPoint(coordinate, pathPoints)
294     self.pathHandle = sim.createPath(pathPoints, 0, 100, 0)
295     self.pathAlreadyRemoved = False
296 
297 
298 def removePath():
299     sim.removeObjects([sim.getObject('../Path')])
300     for k in reversed(range(0, len(self.points))):
301         nodeName = f"../ctrlPt[{k}]"
302         sim.removeObjects([sim.getObject(nodeName)])
303     self.pathAlreadyRemoved = True
304 
305 
306 def appendDataRow(precPoint, nextPoint, position, orientation,
    nextOrientation, event):
307     position = [round(position[0], 5), round(position[1], 5), round(
    position[2], 5)]
308     orientation = [round(orientation[0], 5), round(orientation[1], 5),
    round(orientation[2], 5)]
309     self.dati.append({
310         "PrecPoint": precPoint,
311         "NextPoint": nextPoint,
312         "Position": position,
313         "Orientation": orientation,
314         "NextOrientation": nextOrientation,
315         "Event": event
316     })
317 
318 
319 def saveDataCSV():
320     CSVFileName = 'selfDriveAutoData.csv'
321     headers = ["PrecPoint", "NextPoint", "Position", "Orientation", "
    NextOrientation", "Event"]
322 
323     CSVFileNotExists = not os.path.exists(CSVFileName) or os.path.
    getsize(CSVFileName) == 0
324     with open(CSVFileName, mode='a+', newline='') as file_csv:
325         writer = csv.DictWriter(file_csv, fieldnames=headers)
326         if CSVFileNotExists:
327             writer.writeheader()
328         writer.writerows(self.dati)
329 
330 
```

```
def sysCall_init():
    sim = require('sim')


def sysCall_actuation():
    sim = require('sim')

    startSimulation = sim.getStringSignal("startSimulation").decode("utf-8")

    # Init
    if startSimulation == 'True' and self.initFinished == False:
        self.startPoint = sim.getStringSignal("startPoint").decode("utf-8")
        self.stopPoint = sim.getStringSignal("stopPoint").decode("utf-8")
        calculateDijkstra()
        generatePath()
        setVelocity(self.velocity, self.velocity)
        self.initFinished = True

    # Actuation
    if startSimulation == 'True':
        if self.destinationReached == False:
            precPoint = self.points[self.index]
            nextPoint = self.points[self.index+1]

            coords_prec = self.dummies[precPoint]
            coords_next = self.dummies[nextPoint]

            [x, y, z] = getPosition(self.carObject)
            [xPos, yPos, zPos] = getOrientation(self.carObject)
            [a, b, c] = getNextOrientation(coords_prec, coords_next)[0]
            oppositeDirection = getNextOrientation(coords_prec, coords_next)[1]

            # Proximity Sensor
            wallDetected,_,_,_,_ = sim.handleProximitySensor(self.proximitySensor)
            if wallDetected == 1 and self.proximitySensorEnabled:
                print(">> Wall detected")
                self.proximitySensorEnabled = False
                if self.pathAlreadyRemoved == False:
                    removePath()
                setVelocity(-self.velocity, -self.velocity)
                appendDataRow(precPoint, nextPoint, [x, y, z], [xPos, yPos, zPos], [a, b, c], "Wall detected")
                self.retro = True


            # Retro
            if self.retro and checkRange(x,y, coords_prec):
                self.proximitySensorEnabled = True
                setVelocity(0.0, 0.0)
                self.retro = False
                self.index = 0

                if nextPoint in self.adjacencies[precPoint]:
```

```
383                     self.adjacencies[precPoint].remove(nextPoint)
384                 if precPoint in self.adjacencies[nextPoint]:
385                     self.adjacencies[nextPoint].remove(precPoint)
386                 self.points, self.distances = shortest_paths(self.
    adjacencies, self.dummies, precPoint, self.stopPoint)
387                 print(self.points)
388                 generatePath()
389 
390                 precPoint = self.points[self.index]
391                 nextPoint = self.points[self.index+1]
392 
393                 coords_prec = self.dummies[precPoint]
394                 coords_next = self.dummies[nextPoint]
395 
396                 self.orientation = self.newOrientation
397                 self.orientationName = self.newOrientationName
398                 self.newOrientation = getNextOrientation(coords_prec,
    coords_next)[0]
399                 self.newOrientationName = getNextOrientation(
    coords_prec, coords_next)[1]
400                 checkRotationWheels(self.orientationName, self.
    newOrientationName, 0.78, True)
401                 self.isChanging = True
402 
403             # A dummy was reached
404             if checkRange(x, y, coords_next):
405                 print(f">> Intersection: ({precPoint}, {nextPoint})")
406                 self.orientation = getNextOrientation(coords_prec,
    coords_next)[0]
407                 self.orientationName = getNextOrientation(coords_prec,
    coords_next)[1]
408                 self.index += 1
409                 self.changeDirection = True
410                 appendDataRow(precPoint, nextPoint, [x, y, z], [xPos,
    yPos, zPos], [a, b, c], "Point acrossed")
411 
412                 # Is destination reached?
413                 if self.index == len(self.points)-1:
414                     print(">> Final destination reached")
415                     setVelocity(0.0, 0.0)
416                     self.destinationReached = True
417                     if self.pathAlreadyRemoved == False:
418                         removePath()
419                     appendDataRow(precPoint, nextPoint, [x, y, z], [
    xPos, yPos, zPos], [a, b, c], "Final destination reached")
420                     saveDataCSV()
421                     sim.pauseSimulation()
422 
423             elif self.changeDirection == True:
424                 self.newOrientation = getNextOrientation(coords_prec,
    coords_next)[0]
425                 self.newOrientationName = getNextOrientation(
    coords_prec, coords_next)[1]
426                 checkRotationWheels(self.orientationName, self.
    newOrientationName, 0.8, False)
427                 self.isChanging = True
428                 self.changeDirection = False
429 
```

```
430             if self.isChanging and checkOrientationRange([xPos, math.
    degrees(yPos), zPos], self.newOrientation):
431                 setVelocity(self.velocity, self.velocity)
432                 self.isChanging = False
433
434
435 def sysCall_nonSimulation():
436     pass
437
438 def sysCall_beforeSimulation():
439     loadGlobalVariables()
440
441 def sysCall_afterSimulation():
442     if self.pathAlreadyRemoved == False:
443         removePath()
444
445 def sysCall_cleanup():
446     pass
447
448 def sysCall_cleanup():
449     pass
```

Chapter 3

Conclusions

In conclusion, the projects presented for industrial automation and robotics were the beer production process and the development of a robotics program (self-drive system) using the Coppeliasim simulator.

- **PLC Part: Beer production**
  The brewing process is a process that requires time and in particular a lot of attention to obtain excellent quality beer. Through sensors, actuators, valves and motors, the process can be optimized via PLC programming. The use of the Ladder language and PLC programming offers several advantages compared to other types of programming, especially in the context of industrial automation: a visual and intuitive language (even for those with little experience), maintenance, reliability, scalability, robustness, integration with different systems and deterministic execution minimizing errors.

- **Robotic's Part: Self-drive car**
  CoppeliaSim offers numerous advantages for the development and simulation of self-drive car systems, making it a useful choice for such applications. It contains a realistic physics engine (for example the bullet 2.78 used), quality 3D rendering, flexibility, modularity, integration with ROS and provides a complete environment of models and complex sensors such as GPS, Lidar and radar. Thanks to coppeliasim we have created an excellent environment where we can simulate the autonomous driving of a vehicle thanks to the sensors inserted and the programming logic developed.

3.1 Future works

- In the interface there is a button that, when pressed, must randomly enable a wall within the path of the machine. What you can do is insert a series of walls inside the scene, changing their properties to make them, at the beginning, invisible and non-collidable and once the key is pressed one of them is made visible.
- Another possible future work is to fix the part related to the turn of the machine, as, with a speed greater than the set, the values of the orientation we get, between one loop and another, have a greater gap than the preset range and therefore the machine would not find the point where to stop the turn. The range could be increased, but, by doing so, the machine would stop the turn too early and consequently we would have a wrong trajectory.