UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería Electrónica Industrial y Automática

# Universal Robots® UR5. Desarrollo de Programación

Autor:

Rodriguez Vidriales, Carlos

Eusebio de la Fuente López

HAMK University

Valladolid, septiembre de 2020.

TFG REALIZADO EN PROGRAMA DE INTERCAMBIO

|  |  |
|---|---|
| TÍTULO: | Universal Robots® UR5 Program development |
| ALUMNO: | Carlos Rodriguez Vidriales |
| FECHA: | Julio 2020 |
| CENTRO: | Hammë Ammatikorkeakoulu (Valkeakoski) |
| UNIVERSIDAD: | HAMK University |
| TUTOR: | Juhani Henttonen |

ABSTRACT:

Nowadays, Robotics has plenty of industrial applications. From basic mechanical operations to massive production executions, any robotic system must have the capacity to solve them so, in order to know how they work, robots must be analyzed using theory studies and results given by different kind of simulations. In this thesis, UR5 will be analyzed by putting it in a controlled laboratory and a virtual industrial environment to see how it responds when different tasks are needed to be done. Since both real and simulated robot are being studied, different software will be needed to control them, so programming is also an important factor to know about. Visual Components, which is the simulation software, will give us the opportunity to know about Python language, and UR5 software, Polyscope language.

KEY WORDS: Robotics, Automation, UR5, Visual Components, Python

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

2020

# Universal Robots® UR5 Program development

Carlos Rodriguez

HAMK University

20-7-2020

# TABLE OF CONTENTS

# INTRODUCTION

A robot is, by definition, a mechanical system which can be moved in the space so it can accomplish a certain number of objectives following some orders given by a human being. Either the term or its development´s beginning as we know nowadays, start during the 20th century. This doesn´t mean that the robot´s history is a short tale. In fact, there were plenty of machines which, being manipulated by a mechanical control, they could imitate certain movements that helped them to represent the shapes in which they were made. Those machines are called *automates* because they could only make one type of action, inherent to their fabrication process, and it can´t be selected, which is the opposite thing of the robots. Starting from *King-su Tse*, who invented a wood-made bird in the 500 b.C. until *Leonardo da Vinci*, with its mechanical lion which could open its chest so it could show the royal emblem of the family of *Francisco I.*

The word *robot* was invented by the Czech author Karel Capek to its theatrical rehearsal "Rossum´s Universal Robots" in 1922 (*robot* comes from the word *robota*, which means servitude´s work). This is a very important historical event because it was this rehearsal the first one which shows that is possible to create specific human shaped systems that can make slavery activities and heavy work for the humankind (nowadays this concept is defined by androids and gynoids). Since then, either in a cultural or technological way everything that we know related to robotics have been developed.

One of the first steps of this development was the invention of the Turing´s machine in 1936, which lead to the famous robotic laws in 1942 and other important events, until the creation of the Industry 4.0 and the collection and manipulation of resources using *Big Data.* Those robotic laws are the cornerstone of the modern robotics. They were made up by the writer Isaac Asimov and were first published in 1942 in the novel *Runaround* and are described like this:

1.- A robot may not injure a human being, or, through inaction, allow a human being to come to harm.

2.- A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.

3.- A robot must protect its own existence as long as such protection does Dot conflict with the First or Second Laws.[11]

Those are the rules in which every protocol is based and all the physics are implemented into the robotic systems, in order to guarantee the workers´ security and to minimize as long as it is possible the incidents that can happen. Those can appear as alarm commands which leads to a safer environment, or the creation de isolated spaces such as security cells where we put the robots so their movements can´t be blocked by any spontaneous actions or by any human. They can even appear as morphic developments so nobody can get hurt even if the robots are programed to, because that protection is inherent to the components of the machine, like the collaborative robots, or *cobots*, made to move in hostile environments where human workers and robots can work together safely even if they get hit.

In a commercial way, the start of the robotics begins in 1954, when George Devol develops the first programmable robot, called Unimate[2], which was patented as a "programmed device to transfer articles". Officially considered as the first industrial robot, it could transfer pieces which were previously melted cast into an assemble process room, so they could be welded into a factory car. However, the sales in the very first attempts were failed. It wasn´t a successful product, until the patent was bought by the entrepreneur *Joseph Engelberger*, whose idea with the robot was to sell it from a brand-new company called *Unimation* making advertisements showing that specific robot making different services like preparing drinks. Thanks to that decision, nowadays we call him as the "Father of the Robotics"

After that, *Unimate* stopped being just a robot, and it started to be an entire group of different products, being developed in other countries, like Japan, where a conference was organized for almost 700 Japanese executives and manufacturing engineers with Engelberger as speaker during the 60´s.

That event made that a lot of different companies from other countries started to develop their own products so they could compete against them in this new commerce. Those companies were mainly European, Japanese and north American, and also some universities started to research for new technologies in different laboratories. MIT, Stanford University and Edinburgh University are some examples of the different investigating centers, but it´s the Stanford University who has the first achievement, creating a little robotic arm with electrical actuations only in 1971.

In the company world, the Swedish company called ASEA (which is not actually an independent company, but a holding company from ABB group) created de IRB series, with its most successful product, the IRB6 machine: the first robot controlled and processed entirely by electrical signals, which could make plenty of functions, like welding, assemble, transport, varnished, supporting and leveling. In Asia, on the other hand, the JIRA (Japanese Industrial Robotics Association) is created by the vehicle´s factory Nisan, which is known for being the first association in history. Other companies, like Kawasaki create less important robotic systems.

As soon as computer technology is growing faster, making possible the development of the first digital electronics processes, the discover and manufacturing of the transistors and the LED technology, or the investigation about the use and invention of the first microprocessors since 1968, research areas start to look for the creation of the first robot computer-controlled which could execute without any problem Real Time commands, achieved in 1974. The idea of controlling a robot in an electronical way was normalized in the industry society, letting new companies to use this technology in new areas of investigation; even in the space (NASA used for the first time an arm shaped robot in 1976 to be used in the space[2].

Since then, plenty of new branches were opened in the era of the modern robotics, developing their products depending on the morphology, technology, application aim, environment…One of the most used examples of that, which is the one that we´re referring to this project, is the educational oriented "cobots", manufactured by Universal Robots.

Universal Robots is a Danish company dedicated to production and research of industrial robots which are specified to the small and medium companies. Although is a small company, it makes the biggest exportation of university educational robots.

Universal robots was founded in 2005 by Esben Østergaard, Kasper Støy, and Kristian Kassow, which were three classmates from the university. In the beginning, the idea of the company was to create new technology which could make little industrial task for small companies. Three years after, they successfully sell the first collaborative robot: UR5. Considered as the first big winning in the market, either its electronics or morphology become paradigmatic to the next series of products to be sold[4]. All the next generations (UR3, UR10 and UR16) are made in a shape of a human arm and are controlled by a digital screen where the user can directly interact called the Teach Pendant.

The physical similarities between those series and an arm are quite simple to see. Imagine a person standing up, with its arms extended. Due to its boned structure, it is completely rigid except for some joints which help us to move. When the study is focused on the torso and the arms, we can see that, from the waist to the wrist, there are 6 fundamental movements that let us reach all the points in the affordable space work. Starting from the waist, there´s one movement that we can do, which is rotational around the vertebral column. That movement can guide our arm in some specific direction. Then, we have the vertebral column, which with the help of all vertebras working as one joint (we can´t move one specific vertebra without moving the others) inclines all the body to the front and backwards. After that, we have the shoulder, which rotates all the arm in half spherical movement. Continuing with the description, the elbow, which is a rotational joint, that moves in the plane created by the arm and forearm, describing half of a circle. The last two joints are located in the wrist. Both joints, which are rotational, can describe a last hemisphere with the hand. Those joints are all described by these *cobots*, as we can see in the pictures in Figure 1:

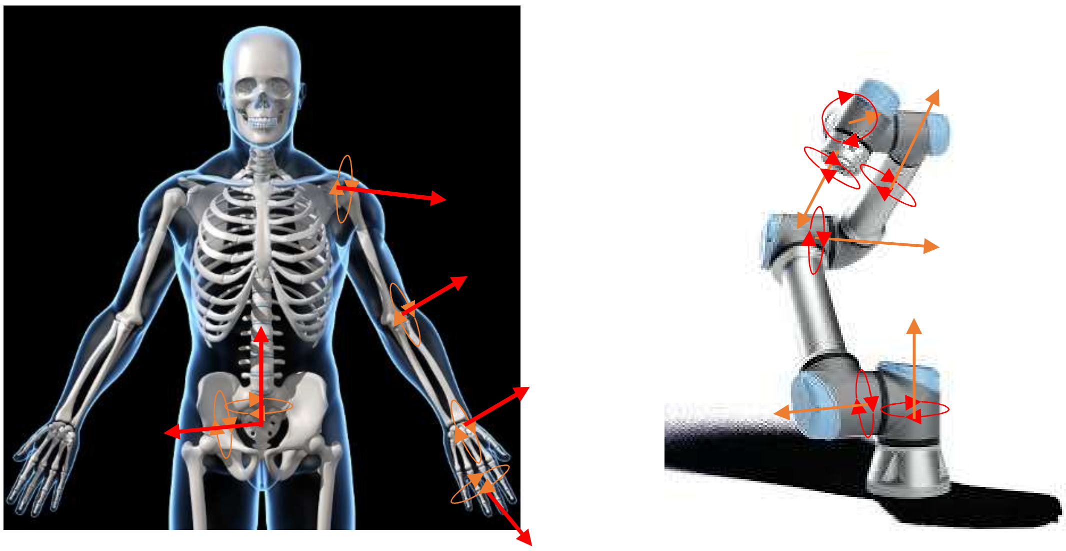


Figure 1. Similarities between human joints and UR5 joints

As, in the same way that we can reach most points in the workspace volume with a specific rotation of our joints, the *cobot* also can. Due to the necessity of the robot to use 6 joints to reach each point, is the reason that we say that the robot has 6 degrees of Freedom (6-DOF) For simplicity, each joint is numbered from 1 to 6, starting from the joint where is screwed to a flat surface, to the point that we´re going to use to do the functions given. The first joint (called base as well), is the one where absolute coordinate frames are usually set. Is common that in the last joint, in order to make different operations, drills are made to attach different types of instruments, so the robot is properly equipped. However, this configuration makes the user to set other coordinate frame that can give the robot relative movements. This frame, is called TCP (Tool Center Point).

The program that controls the UR series, is called URSim[10]. URSim lets the connection to the robot by using the interface, where we can set several options from the configuration of the robot, set the properties and features of the TCP, store the data collected during an execution, or set an entire script to be done once or repetitively. The interface that we´re talking about, is the Teach Pendant. The Teach Pendant is, as shown in Figure 2, a digital screen which can be used with a stylus given by the company, to control the instant states of the robots, with its Inputs and Outputs, and gives the user the complete control over the robot. To ensure the communication between user and robot, both are connected via wire. It consists of one tactile screen, one handle under the interface, so it can be safely held without minimum probability of fall, one shutdown button, one emergency stop button, which, if you press it, it cuts the supply to the robot safely so it can't be moved until certain checks are done, one freedom button, which is used to unlock the joints the robot so you can easily move it with your hands, and an USB port to import and export the programs written.

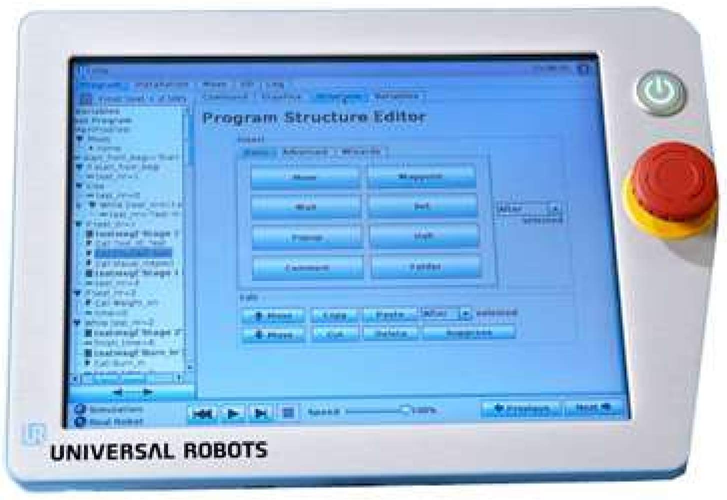


Figure 2 Teach Pendant™

The language given by this device, is called PolyScope[9]. This language is a sequential-commands-based language, with the possibility of setting threads and alarms, in order to get the variables of the robot and transform them into voltage and current levels inside the motors. Since the program was to make thing understandable to the user that is monitoring it, the commands are simplified in functions of set, reset variables, different movements of the robot, and the basic operations. One experienced user can use more commands, even it can write a complete script and import it at certain line. This language uses the normal operators and variables, such as the Booleans, the integer or the string variables, but also need alternative variables, like the waypoints or the confdata (which we saw before). However, the waypoints and the joint variable are the main variable used here. Those variables are the ones which set in the 3d world, takes the information of the position and orientation of a point to be reached, depending if you set by using direct kinematics or indirect kinematics. On one hand, the set points have 13 sub variables: 3 of position, 4 of orientation and 6 of external axis coordinates. On the other hand, joint variables only have 6: one for each joint that the robot has . The interface offers three main groups of commands to be chosen in the main program: The Basic functions, The Advanced Functions, and the Wizard Functions:

Basic Functions

Those are the main functions that the robot can do to execute a movement program, with changes in variables. The commands given are:

1. MoveJ (Waypoint): With a waypoint or a group of waypoints given, the program sets a path in which the robot will go in order to reach those waypoints. The type of movement that the robot does, is a joint movement, so the robot will take the easiest movement warrantying that the robot won´t hit any part of himself, when there´s a possibility to do it that way. In most cases, the waypoints can be reached. The only exceptions that can´t make the robot move are:
    a. The point to be reached is outside the workspace of the robot
    b. The point given is inside or near the surface of one of the arms of the robot
    c. The path calculated makes the robot surpass the value of one of the arm angles.

In terms of efficiency and energy, this movement is the one that wastes the less, but is one of the slowest movements. So is good to ensure the joint configuration of the robot, but is not if the movement given must be critical.

2. MoveL (Waypoint): Is the same type of command than MoveJ, with the exception that the movement that the robot does like this is linear. This type of command is usually used when the robot needs to draw in the air some patron, or when it gets a point which is near the actual position, so it can get it with the same orientation and in a fast way. In other cases, this command is not recommended, since the robot recalculates the movement for each joint every time that it has to get a new point, which is different to the previous one, which only calculates the necessary joints to be moved. On the other hand, it has the least distance to move, which is an advantage in some cases (when, for example, the workspace is smaller than the reachable space that the robot has).
3. Wait and Halt: Those are commands that stops the robots in two special circumstances. If we want to stop the robot for a specified amount of time, and then we want the robot to continue with the program, we will use the wait command. On the other hand, if we want to end the program and stop the robot immediately, we will use the command Halt. The las one is used as an alarm when we want to finish earlier the program or if we

have an error and is highly not recommended to move the robots because it can hurt someone or itself.

4. Set: as the command says, it sets the value of certain variables depending on its nature. It can set digital and analog Outputs a value, or it can set a function, it can set pulses to external variables, or increment by one any kind of variables. Exceptionally, it can set a new TCP and it can reconfigure the total weight of the payload.
5. Popup: This command shows up a message below the screen written by the user, or shows a keyboard so the user can modify the value of a variable. Once that the variable is changed, or the message is shown, and it´s dismissed, the program will continue normally. There are three types of popup: a message type, a warning type, and an error type, and it can be chosen to Halt the program execution once that the popup appears.
6. Comment and Folder: Comment and Folder are string type commands, where it contains a text to be shown. In the comment command, it´s written a non-functional text, which is used to explain the script that the robot will execute. The folder command, on the other hand, holds a collection of program lines that executes when the robot gets that line. Is the equivalent in normal programming languages as subroutines or functions.

## Advanced Functions

Those are the functions that controls the logic of the sequence in which the robot must execute the program lines. Those break with the linearity of a normal execution, and creates different paths so the robot can control more situations. Its commands are the loop command, the SubProg command, the Assignment command, the If-else command, Script Code command, the event command, the thread command, the timer command, the switch-case command, and the direction command:

1. If-else and switch-case: Those are the commands which sets conditions during the execution depending on logical statements given. Usually, on the value of variables. Both structures, make the if and else, and the different cases in switch, respectively, exclusive one from another, being impossible to execute both program lines in the same try. Switch-case structure is a special case of if-else. It´s used when there are more than three statements that can happen to the logical variables, or is used to classify a value

given. Once that those structures finish, the program continues, and they can be only executed once.

2. Loop structures: The loop structures execute a group of program lines several times until a statement is done. The statement can be an expression that becomes true, it can be a counting of a certain amount of times, or it can even be an infinite loop, which is executed always. Once that the loop is finished and the script exits the loop, it can´t be returned to it.
3. SubProg and Script Code structures: They set up and executes a group of lines in order to make the script easier to read it. It´s similar to the Folder command, with the difference that those ones call the subfunctions from other part of the program, or even in other file of the system, and those lines can´t be fit there without making things harder than expected.
4. Event command: Those are one type of interrupt commands, where it gets the highest preference in alarms. The difference between the interrupt and event, is that, when a statement becomes true and both are executed, the interrupt blocks the normal execution of the program, until the interruption is done.
5. Timer: Is a specified type of event command, where, one that the program lines get to them, starts, stops or resets the measurement of the time that the robot can do. It can either just show it, or store it in a variable to be used. Is typical to use those variables as logical statements in loops and if-else commands
6. Direction: Is a MoveL command where, instead of setting a waypoint to move the robot, it makes a relative calculus from the actual position to the point to be reached. It's really useful to draw some patterns or get some geometrical points in the space
7. Thread commands: Those commands set two parallel scripts that are executed at the same time, so the robot and the outputs can what they are told to do at the same time, in order to save time or scape from infinite loops. Those kinds of commands are prepared to optimize the time and resources of the processor so, even if in a microscopical way the processor is executing sequential lines, macroscopically is doing two tasks at the same time.

Wizard Functions

The wizard functions described the optional commands that the robot can handle, when its related to inputs and outputs from devices that are not part of the robot. There isn´t a specific amount of commands, because those depends on the external device that we use in the process and are usually imported from the original page of the distributor. There are 4 main wizard commands, which are general for these devices:

1. Pallet: The pallet operation allows the robot to perform the same sequence of motions and actions at several different positions. It's used for palletizing or similar operations. A pallet operation consists of the following steps:
    a. A Program Sequence
    b. A Pattern
    c. An optional "Before start" Sequence
    d. An optional ”After start“ Sequence
2. Seek: Is a main function which holds two different operations: Stacking and Destacking operations. With given parameters such as the distance from the first stock to the last one, and the pointer of the stock which indicates the starting position, the robot can pick and place several objects in the workspace, and stack them making a pile, or can destack them placing them in other places or just picking one of them.
3. Force: This function lets the *cobot* to be free for a certain amount of time, so it can move, pick and push objects with a certain amount of force. This is useful when the robot must do some operations where fragile objects and weak stuff must have some operations done, and depending on the force given can easily break, so the situation requires that the robot must lower its parameters only in those operations. In some cases, where the robot has an external armor which gives it more force, the force command can measure the big operations which requires harder work.
4. Conveyor tracking: This robot is able to connect external conveyors so it can be synchronized with the fabrication process. That means that the program is prepared to follow the actual information of the conveyor and make an actual track of it. The program is up to control two different conveyors at the same time
5. Gripper actions: This kind of commands, which are imported from the gripper module (We´re talking about the Robotiq® 2F-85 gripper) collects several different commands that can control the gripper that we are using. The group of commands includes grasp and release, memorize object and if-else detection object (this gripper includes a wrist

camera to use artificial vision so the robot can understand its environment and act in consequence).

As we have just said, the robot is able to communicate with the exterior by receiving signals from different sensors. Its idea is to simulate that it has some of the 5 senses that the humans have. In fact, we can relate that the robot that we are using contains at least two of the 5 senses that we have, and it’s possible to include one more.

The first sense that we can see, is in the robot´s nature. Since this robot is a specific type of robot called *cobot*, and can detect if any joint is blocked in the environment and can set an interruption if it gets touched by anything, we can say that the robot has the touch sense. This sense is inherited to the robot, so it cannot be disabled. However, its reaction can be programmed so instead of stop the program lines, the user can program it to avoid the obstacle or even display a message to the Teach Pendant.

The second sense that the *cobot* has, is the sense of sight. Since the robot has attached to its TCP a camera that helps the robot to identify objects that have different colors or different shapes, it´s able to classify the products, or select one specific which is different to the others, and do different task that can be only done to those figures. The technology of Artificial Vision has been so developed, that nowadays they can combine it to artificial intelligence and can make robots to play famous games against the most experienced players and win, as we can remember the famous battle between Kasparov and the Deep Blue machine, or recently, with the machine called AlphaGo, which is the actual leader of the game called “Go”, defeating the best players in quick matches.

It’s possible to mention a third one, the hearing, so we could attach to the robot some vibration sensors so it could understand some situations depending on what´s going on, but here it´s nonsense to use it for an industrial life (in industries is common to have loud noise and yelling, so those can easily make an interferences with the actual orders or wanted stimulus).

All similarities will be necessary if we want to achieve the objectives that we´re following so we can demonstrate the hypothesis (which we´ll discuss later). During those objectives, there aren´t only programming steps, but programming and organization of the environment, knowing that either knowledge in VC program or Python language are needed to show the results as real as possible according to what can happen in real life. Despite that, the most important part will be the results, in a conceptual way, which will show if the robot is able to process indefinitely the exercises proposed.

The TCP that we´re using, as mentioned before, is a gripper, that catches the objects and releases in other places. The specific model is the Robotiq® Wrist camera and gripper 2F-85. This gripper is able to pick objects with a maximum diameter of 85 mm and has measurements in angles and mm of gap between the claws. This model uses servomotors to control the speed and position of the claws and gets Input-Output signals to control both parameters. Thanks to the series connection, it can quickly send and receive the signals. Using the commands, the gripper can control how much open do we want the gripper, how fast do we want to do it, and with which force do we want to grab them, so it can take fragile objects. This gripper is modular with a wrist camera that it has, so we can see at any moment what the robot sees, and we can set several parameters to make the robot more efficient.

## PROJECT HYPOTHESIS

If the program *Polyscope* is able to develop several program lines and convert them into simple joints movement, then the composition of different groups of program lines makes a composition of different joints movements leading to complex process tasks completing an entire manufacturing process.

## OBJECTIVES

In order to demonstrate the Hypothesis, we´ll have to achieve the following objectives:

1. Design and develop several environments of the robot to behave in different situations

2. Design and develop the environment props to interact with the robot

3. Program the external devices needed to cooperate with the robot

4. Program the tasks using the robot language, starting from easy tasks to complete groups of programs

5. Evaluate the final result by executing the tasks

# MATERIAL AND METHODS

## MATERIAL

- Laptop MSI GF63 Thin 95C
- Processor Intel® Core™ i7-9750H CPU with 2.60 GHz as frequency and 16 GB RAM
- Nvidia GeForce™ GTX 1650 with Max-Q Design
- Operative system Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- SSD SAMSUNG 500 GB
- External HDD Intenso GmbH 1TB
- Microsoft Office 365 for Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- Microsoft Word 2020 Professional for Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- Notepad2 4.2.25 for Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- VMware Workstation 15 Player for Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- Visual components Premium 4.2 for Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- AutoDesk 2019 Student´s License for Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- AutoCAD 2019 Student´s License for Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- Operative system Ubuntu 18.04 unknown version and unknown processor
- Operative system Ubuntu 19.10 version 64 bits and x64 processor
- Operative system flavour Ubuntu´s flavour "Lubuntu" version 64 bits and x64 processor
- URSim_VIRTUAL-3.13.0.10253 Lubuntu based
- URSim_VIRTUAL-3.13.0.10253 Ubuntu based
- UR5 System for URSim_VIRTUAL-3.13.0.10253 Lubuntu based
- Python 3.8 version 2019 for Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- Python 2.7 for Microsoft Windows 10 Pro 2019 version 64 bits and x64 processor
- Python module pathlib for Python 3.8 version 2019
- Pip module Python 3.8 version 2019.

- Universal Robots® Teach Pendant module with IP20 classification, 1.5 kg, 4,5 m cable length made in aluminum and PP, with emergency stop button, digital screen, Hands-Free button and handle.
- Universal Robots® Control Box module with IP20 classification, in 6 level ISO class Cleanroom, less than 65 dB of noise, 4 type of I/O ports (Digital input, Digital Output, Analog Input, Analog Output), I/O power supply of 24V, 2A, Communication via TCP/IP, 100 Mbit of speed, Modbus communication, Profinet communication and Ethernet IP communication, Power source of 100-240 VAC, 50-60 Hz, with necessary functioning temperature in range between $0^o$ and $50^oC$, configurated in a 475x423x268mm Protection Box, with 15 kg and made of steel
- Universal Robots® UR5 robot with repeatability of ±0.1 mm, ambient temperature operating range between $0^o$ and $50^oC$, Typical 150W of Power Consumption, 15 advanced adjustable safety functions, TüV NORD Approved Safety Function Tested in accordance with EN ISO 13849:2008 PL d, Payload of 5kg, 850 mm of reach, 6 Degrees of Freedom, Polyscope graphical user interface on 12 inch touchscreen with mounting Teach Pendant Programming, Working range on each joint $\pm 360^o$ , and speed of $\pm 180^o/s$, and speed of TCP 1m/s, IP54 classification, 5 level in ISO Class Cleanroom required, 72dB of noise, with any kind of robot mounting, 6 I/O ports, including 2 Digital Inputs, 2 Digital Outputs and 2 Analog Inputs, I/O power supply in TCP of 12V/24V and 600mA, round footprint with 149mm diameter, made in Aluminum, PP and other plastics, tool connector M8 type, 6m of cable length for the robot arm, and a total weight (with cable) of 18.4kg.
- Robotiq® 2F-85 Gripper, with opening range of 85mm, minimum diameter for encompassing of 43mm, maximum height of 162.8mm, maximum width of 148.6mm, 925g of weight, grasp force of 20N, up to 235N, Finger speed of 20 to 150mm/s, repeatability of position of 0.05mm, ±10% of Force repeatability, position resolution of 0.4mm, and grasp force resolution, referred to the Force control, followed by:

  $$W = \frac{2F \cdot C_f}{S_f}$$

  With $C_f$ Coefficient of friction, $S_f$ Safety factor, W weight force of the object grasped and F actuation force.

  Center of mass (including the camera) in x, y and z respectively, -0.7mm, 1.2 mm, 57.1mm (considering the axis in Denavit-Hartenberg parameters method), with TCP located in (0,0,175.5)mm (relative position from coupling joint), and a total mass of 975g. Moment of Inertia with fingers fully open:

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} = \begin{bmatrix} 4180 & 0 & 0 \\ 0 & 5080 & 0 \\ 0 & 0 & 1250 \end{bmatrix} kg \cdot mm^2$$

Force limits in x, y and z of 50, 50, 50 N, moment limits in x, y and z of 5, 5, 3 Nm, with Operating supply voltage of 24 VDC ±10%, and absolute maximum supply voltage of 28 VDC, having minimum power consumption of less than 1 W, and maximum peak current of 1A.

- Robotiq® Wrist Camera, maximum load of 10 kg, Weight with tool plate 230g, added height of 23.5mm, global thickness of 29.5mm, Center of mass located in (0,1,58)mm, 975 g with 2F-85 of total mass, approximate moment of inertia:

  $$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} = \begin{bmatrix} 111 & 0 & 0 \\ 0 & 70 & 3 \\ 0 & 3 & 165 \end{bmatrix} kg \cdot mm^2$$

  Operating supply voltage of 24 VDC ±20%, minimum power consumption of 1W, maximum power of 22 W, Communication interface with other modules via USB 2.0, Camera with maximum resolution of 5Mpx at 2fps (2560 X 1920), maximum frame rate of 30 fps at 0.3 Mpx (640x480), Active array size of 2592 x 1944, Focus range from 70 mm to infinity, 6 LED diffuse white light Integrated lightning, Liquid sense technology of autofocus, accuracy of vision in UR5 of ±3mm, minimum board distance in UR5 of 34 cm in calibration board position, maximum board distance of 70 cm in calibration board position, minimum board distance in UR5 of 64x48 cm in field of view, maximum board distance of 10x7.5 cm in field of view, and recommended background using HSV color cone[9].

- 250 x 250 mm white board made of rigid polymer; CNC mechanized with four 10 mm diameter steel bars located in each corner, 5 grey pipes used as pieces collector, making a row inside the board, and 20 holes of 20mm diameter grouping them in a 4x5 form. Hole and pipe sizes adapted to the pieces with +2 mm tolerance
- 25 black cylinder pieces with 20 mm diameter and 10 mm height.

## Methods

Due to the circumstances during the months that the Final Project was made, there are two main environments in which we program the robot. The first part, which was made during the time that I was able to work in the laboratory, is made with the actual robot and stuff from the laboratory, which are the Operative system Ubuntu 18.04 unknown version and unknown processor, URSim_VIRTUAL-3.13.0.10253 Ubuntu based, Universal Robots® Teach Pendant module , Universal Robots® Control Box module, Universal Robots® UR5 robot, Robotiq® 2F-85 Gripper, Robotiq® Wrist Camera, 250 x 250 mm white board and 25 black cylinder pieces. The other part, once that the university was closed, and those things weren´t available anymore, so all that was supposed to be made in the next exercises, were changed by simulation, using the program Visual components Premium 4.2 to simulate the animation, AutoCAD 2019 Student´s License to model all the props that the simulation program can´t offer or was needed to make things similar to the situation in laboratory, VMware Workstation 15 Player, Operative system Ubuntu 19.10 version 64 bits and x64 processor, Operative system flavor Ubuntu´s flavor "Lubuntu" version 64 bits and x64 processor, URSim_VIRTUAL-3.13.0.10253 Lubuntu based and UR5 System for URSim_VIRTUAL-3.13.0.10253 Lubuntu based so the robot control and the teach pendant can be simulated, and both Python 3.8 and 2.7 as programming language to setup the script of the behaviors from the different conveyors, feeders and robot, respectively.

All the development of the project can be summed up with the next requested tasks:

### Basic Level Task:

Using the Teach Pendant, use one of the black pieces from the board and place it in all the holes, with robot, table and board positions not variables in the space.

The basic level task was made with the real robot, in the laboratory of the university, and using the first group of material described earlier.

First of all, in order to know the real workspace of the robot, we check the maximum and the minimum angle of each joint (in the worst of cases) so we´ll program it according to those ranges. Those values will be different to the datasheet of the robot, because objects in the environment will affect in the operations. The first direct problem that the robot has, is its own base support.

Once that the work space is described, the robot can be programmed with the Polyscope language. The idea is the following steps:

First we get sure that the robot will be in the origin point and the gripper will be in "open" state. Once that we have that position, the robot will aproach without obstacle to the piece. Then the robot will grasp it and it will move around the table with two loops that will add to the actual position the distance between holes: one will be for x and the other will be for y. Being in the last position, the robot will leave the piece in its first position, and will finish in the same position that we wanted the robot to be: facing uopwards and with the gripper open.

The steps described were developped in the next only script:

```
PROGRAM
   ROBOT PROGRAM
#HERE WE SET  THE DISTANCE BETWEEN HOLES´ CENTERS
     X_DISTANCE := P[0.04,0,0,0,0,0]
     Y_DISTANCE := P[0,0.04,0,0,0,0]

#SPECIFIC COMMAND FROM THE GRIPPER, WHICH IS CONNECTED TO PORT 4
AND SET DIGITAL INPUT IN "OPEN" VALUE
     GRIPPER OPEN (4)
#ORI_BOT (ORIGIN OF ROBOT) AND OP_POINT (OPERATIVE POINT) ARE,
RESPECTIVELY, THE POINT WITH ANGLES IN DEFAULT SETTINGS, AND THE
AUXILIAR POINT IN WHICH THE ROBOT WILL USE AS REFERENCE FROM THE
PIECE THAT WILL PICK
     MOVEJ
       ORI_BOT
       OP_POINT
#SPECIFIC COMMAND FROM THE GRIPPER, WHICH IS CONNECTED TO PORT 4
AND SET DIGITAL INPUT IN "CLOSE" VALUE
     GRIPPER CLOSE (4)
#USING POINTS IN HALFWAY, THE ROBOT WILL PLACE THE SAME BLACK
PIECE IN ALL THE FREE SPOTS FROM THE BOARD. SINCE THERE´S 4 ROWS
AND 5
COLUMNS, WE´LL START THE LOOPS WITH INITIAL POSITION ONE ROW
BEFORE THAN EXPECTED. OTHER SOLUTION, COULD BE STARTING FROM THE
INITIAL POSITION AND THEN ADD THE NEXT ROWS, MAKING THE LOOP ONE
TIME LESS
     AUX_POS:=POSE_SUB(OP_POINT, X_DISTANCE)
     MOVEL
```

```
          LEV_POINT
        LOOP 4 TIMES
          CALL TRANSLATE_X
          LOOP 4 TIMES
            CALL TRANSLATE_Y
          AUX_POS:=POSE_SUB(AUX_POS, P[0,0.16,0,0,0,0])
        MOVEL
          OP_POINT
          GRIPPER OPEN (4)
        MOVEJ
          ORI_BOT

#AUXILIAR FUNCTION. EVERYTIME THAT WE USE IT, THE ROBOT WILL
CHANGE TO NEXT HOLE LOCATED IN THE SAME COLUMN (X DIRECTION)
   TRANSLATE_X
      AUX_POS := POSE_ADD(AUX_POS, X_DISTANCE)
      MOVEL
        AUX_POS
        LEV_POINT
#AUXILIAR FUNCTION. EVERYTIME THAT WE USE IT, THE ROBOT WILL
CHANGE TO NEXT HOLE LOCATED IN THE SAME ROW (Y DIRECTION)
   TRANSLATE_Y
      AUX_POS := POSE_ADD(AUX_POS, Y_DISTANCE)
      MOVEL
        AUX_POS
```

The script is commented in green with the descriptions of all parts. An important characteristic is that in the grasp and release actions of the gripper, we haven´t considered the size of the piece. The reason is that the robot uses a really low contraction force (about 5N) and the gripper have two rubber claws, so the friction is really high between the gripper and the object. In the end, as long as it has a non-slippery surface, the robot will be able to rasp any kind of object with binary options.

## Intermediate Level Task:

Draw at least two characters. Both characters should have some curvature.  Characters can be either numbers or letters. At this level Visual Components software is used for programming.

From now on, the material and environment turn into a simulation, using the program described before. Everything that we did before doesn´t affect to these tasks. Is important to mention that the project wasn´t programmed originally like this, and included a couple of tasks more that involved the real UR5 and a camera, so the study of the robot wasn´t going to be useless.

In first place, in order to have a similar environment, the props and the structures that were going to be used were modelled and imported to the simulation environment. The following pictures will show everything that was modified from AutoCAD and then imported to the simulation, so UR5 and the wrist camera don´t appear.

Taking some simple drawings from the real objects, the base support, the boards and the pen with the cube were created with the program. The table was imported from a similar model and then was escalated according to the height of the real table.

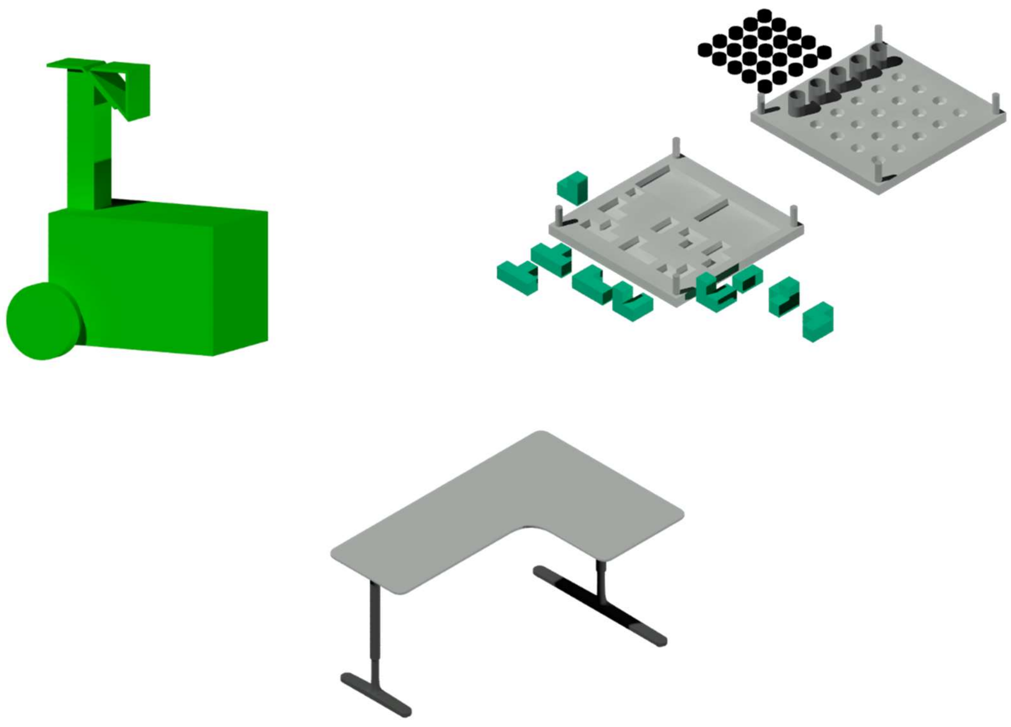


Figure 3. 3D Model of the boards, the base support and the table

With all the things virtually created, the objects from Figure 3 are imported into the Visual Components program and placed properly so the UR5 has a working environment as shown in Figure 4.

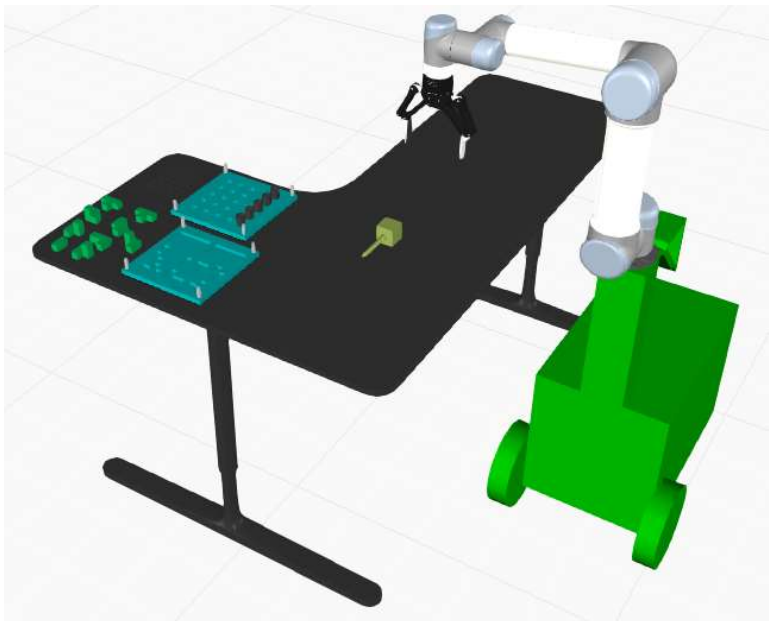


Figure 4. Working environment of the UR5 in the Intermediate level task

With the environment created, next step to be taken is the setup of the frames that the robot should take as reference to the moves that will do.

The program doesn´t automatically attach the gripper, so it must be set up manually. In order to do that, it´s necessary to complete several steps so once that we can set points in the space to be reached by the robot, we will be sure that those will be reached by the desired TCP frame.

First of all, by selecting the option “Snap” in the “Tools” subgroup from the “Modelling” option, we can pick the grip and we can move it in the space. If the gripper is close enough to the last joint of the robot, the program will consider that as a tool that we want to attach to it, so in preview mode we will see a shadow showing the final result of the griped “physically” attached to the robot, as if we screwed it in real life. Next thing that we should do is to set up the I/O label: both from the robot and from the gripper.

To configure the I/O of the gripper, we simply go to the robot´s properties by selecting it in the 3d view and having the “Process” option selected. On the right we will have the Component´s Properties, and in the Input / Output option, we set the desired Output port, since is an actuator that depends on the robot to close and open its claws. This specific robot has only 2 Digital Output ports, so we can choose either of them to connect them. Is important to set as well what kind of actions will make the gripper either if the value is set to False or is set to True. In our case, we can set that when the value is False, the gripper will release, and when is True, the gripper will grasp. We could set the opposite thing as well, since we´re doing it in a simulation, but in order to represent as good as possible the reality, we set it up like that, due to security reasons: if the robot had a problem and couldn´t send a signal properly to the tool, and UR5 does that using positive logic, is reasonable that in that situation the gripper should be open, so anything is blocked by it. Besides it, we will also set the location of the frame in the desired position by selecting the gripper having the "Program" option active. Since we´re using a gripper in which we already know its recommended TCP position (0 mm in x, 0 mm in y, 175.5mm in z), we can simply use that to the simulation. However, although x and y are set in the same position, z was set manually and there´s some distance between the recommended and the actually set (the z coordinate has a value of 183.021mm). In the end, since the gap is small enough (less than 8mm), and we can consider the objects grasped with a weight small enough (that distance starts to be a problem from weights that the robot can´t handle) it won´t affect the result of the simulations, as well as it could happen in real life if those tasks were made with a real robot.

On the other hand, in order to configure the I/O port from the robot, we select the signals from the subgroup "Connect" in "Home" option, selecting the UR5. In the 3d model view we will see a table with arrows pointing, from the external ports in the 2F_140 (we can see that there´re 5 options: IN_J1_Action, IN_J1_Close, IN_J1_Open, OUT_J1_ClosedState and OUT_J1_OpenState) to the control box that the robot has (it has 3 memory locations for Input and 4 memory Locations for Output). This, represents the connections that the robot considered that are plugged from its external ports, and reconverts them into signals that the control box can understand. In real life, this is NOT a task made by a normal user, but an internal connection that the robot automatically does, but in this case, it must be made manually, because those ports, when they are set in TRUE-FALSE values, gives the gripper the movement of the claws. The Input, on the other hand, gives the information to the robot that the gripper is fully closed or fully open[8]. The sequence could be the following:

The robot, in which its memory has value of the gripper at 10% of its capacity (fully closed), wants to open at 80% of its capacity. However, the user opened the gripper releasing the breaks and left it to a 50 %. The robot calculates (since it doesn´t know that the gripper is not physically completely closed) that it has to open the claws 70 % only. In theory, the final result is that the claws will be open at 120% of its capacity, so when the claws reach the 100%, the Out_J1_ClosedState will send a signal to the robot, so it will automatically stop trying. This is a useful way to reset the true values of the gap between claws during simulation. In this part, we will observe that IN_J1_Action is connected to port 100 (is the variable that gives the open and close animation to the gripper).

The description of the task is the following text:

With the position of the robot in operative mode (in operative mode, the angles of the joints are: 0, -90, 90, -90, -90, 0), the robot will first grasp a pen with cubical base that is laying horizontally in the table, and will be used to draw the letters "U, R" and the number "5" in the table. Then the robot will leave the pen in the same position and finally the robot will stay in the operative position until the program reactivates again. As the Basic level task, this one is sequential too, and doesn´t depend on external variables, so everything can be described with fixed variables. Both languages are similar in the nature of their programs.

The Program editor of Visual Components is a modular menu of basic programs that are based in the python libraries of VC. The idea is simple: the menu offers you a blank space where you can put some specific and basic commands. The menu also contains a part to choose between subroutines and external functions that can interact with them.

The basic commands are:

·Point-to-point motion statement: The robot sets a path using joint movements from origin point to destiny point.

·Linear motion statement: Does the same function as the P2P statement, using a linear movement.

·Path statement: Sets, besides the origin and destiny point, several points halfway in which calculates a path using spline function.

·Define tool and Base statement: Both statements change their Tool and Base frames during the program in case that the robot will use a different tool or if the robot has the possibility to work from a more comfortable origin frame.

·Touch up command: Updates instantly the values of the variables and states of the robot so everything is up-to-date once that the command is executed.

·Call sequence statement: The robot stops temporally the actual sequence so it starts another subsequence. Once that the subsequence is finished (unless there´s a stop alarm or a halt before), the robot will return to the line that stopped.

·Assign variable statement: Gets a variable that already exists and changes its value. The variable can be integer, real, string, Boolean or complex variables from other VC libraries.

· While, break and continue statement: Statements of logic sequence commands. While creates loop statements. Break makes the current control command to be ended and makes the robot continue the script after the control that was blocking it. Continue statement forgets the current line that can block the robot from doing anything and continues in the next line.

· If and return statement: statements of logic sequence commands. If will create a condition statement. Return will halt the actual sequence and saves in memory the value of the variable that will be returned to the current function.

· Program synchronize statement: Statement that synchronizes the external devices of the robot with the current states of the variables before it continues.

· Delay statement: stops the pointer from continuing executing the script for a specific amount of time.

· Halt Statement: stops everything that can be executed in the simulation and stops it.

· comment and Print statement: Statements used in show string variables. Comment statement sets a non-functional string text in the script to describe anything. Print statement makes a popup in the computer screen when the program  reaches this line.

· Wait for Binary Input: The robot will stop until a specific binary Input is set to the desired value.

·Set Binary output: The robot will continue with an external variable having a desired value.

```
#LIN IS THE LINEAR MOVEMENT COMMAND USING THE TOOL "TOOL1" AND
BASE "BASE1" FROM THE ACTUAL POSITION TO SEVERAL POSITIONS AND
MOVING AT 1000MM/S. THE AIM POINTS ARE "ORIGIN" ,"AUX" AND
"PICKUPTOOL" (ORIGIN IS DESCRIBED EARLIER, PICKUPTOOL IS THE
POINT USED TO RELOCATE THE ROBOT IN A COMFORTABLE POSITION TO
GRASP THE PENCIL TOOL, AND AUX IN AN AUXILIAR POSITION WHICH IS
USED SO THE ROBOT WON´T HAVE ANY PROBLEMS REACHING THAT POINT.
LIN ORIGIN TOOL1 BASE1 1000MM/S
LIN AUX TOOL1 BASE1 1000MM/S
LIN PICKUPTOOL TOOL1 BASE1 1000MM/S
#SET IS THE COMMAND THAT GIVES THE BOOLEAN VALUE TO THE INPUT-
OUTPUT OPTION IN THE PORT SELECTED WITH NUMBERS. PORT 1 IS THE
FUNCTIONAL PORT THAT SETS THE GRASP AND RELEASE OPTIONS. PORT
100 GIVES PERMISSION TO ANIMATE THE ACTION OF THE GRIPPER IN
THAT MOMENT. SINCE THAT OPERATION LASTS AROUND A SECOND, WE MAKE
THE ROBOT WAIT 1s SO THE ENTIRE ANIMATION IS COMPLETE BY USING
THE COMMAND "DELAY 1S".
SET OUT[1] == TRUE
SET OUT[100] == TRUE
DELAY 1S
#NEXT COMMANDS REPRESENT THE DRAWING OF THE SYMBOLS U, R AND 5
ALONG THE TABLE. EACH PATH COMMAND IS A SYMBOL, AND LIN COMMANDS
MOVE THE ROBOT TO AUXILIAR POSITIONS SO THE ROBOT CAN CHANGE THE
SYMBOL TO BE DRAWN WITHOUT WRITING ANYTHING ELSE.
LIN P2 TOOL1 BASE1 1000MM/S
LIN AUX TOOL1 BASE1 1000MM/S
LIN P4 TOOL1 BASE1 1000MM/S
PATH PP1 TOOL1 BASE1 (POS 21 FIELDS 12 EXT 0)
     P1
     P21
LIN P1 TOOL1 BASE1 1000MM/S
LIN P5 TOOL1 BASE1 1000MM/S
PATH PP2 TOOL1 BASE1 (POS 21 FIELDS 12 EXT 0)
     P1
     P25
LIN P6 TOOL1 BASE1 1000MM/S
LIN P7 TOOL1 BASE1 1000MM/S
```

```
PATH PP3 TOOL1 BASE1 (POS 21 FIELDS 12 EXT 0)
      P1
      P51
#ONCE FINISHED, THE ROBOT USES AUX AND LEAVE POSITIONS TO
RELEASE THE PEN IN THE SAME ORIENTATION AS BEFORE. THEN IT
RELEASES THE PEN AND FINALLY IT GOES TO ITS ORIGIN POSITION
LIN AUX TOOL_TCP BASE1 1000MM/S
LIN LEAVE TOOL_TCP BASE1 1000MM/S
SET OUT[1] == FALSE
SET OUT[100] == FALSE
DELAY 1S
LIN P3 TOOL_TCP BASE1 1000MM/S
LIN ORIGIN TOOL_TCP BASE1 1000MM/S
HALT
```

The script is commented in green with the descriptions of all parts.

## Advanced Level Task:

Using two conveyors (one for incoming objects and the other for pallets where objects are fit inside) the robot UR5 will take, using two raycast sensors, 3D cubes that are randomly placed in different orientations and different distances from the center, and will be placed in a 6-hole pallet where, once that all holes are filled, the pallet will be moved and everything will start again. The raycast sensors will give the distance data to the robot via text file, and the robot will calculate the angle and distance from the center using trigonometry equations. All process will be made using Python coding.

Since the proposed task needs any other type of environment, we must change completely the 3d view according to everything that is described above. Fortunately, there´s no need to create new models, because the catalogue of VC offers all we need to create the simulation (conveyors, feeders and pallets). In the end, there´s no need to create anything more complex than the environment that we can see in Figure 5, so modelling and rendering are not a problem anymore.

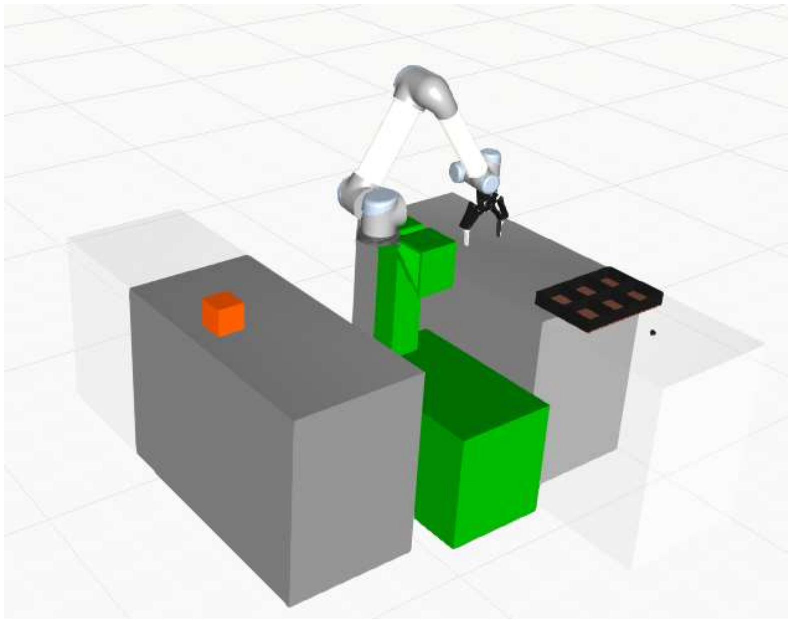


Figure 5. Working environment of the UR5 in the Advanced level task

All this task is made using Python language; even the creation of some 3d cubes, with VC libraries prepared for this language.

Python is a programming language created to make program lines more legible and comfortable, either to read and to write. It was created by the Netherlander Guido van Rossum. Nowadays, this language is one of the most used languages worldwide, and is used in all kind of different situations, including 3d modelling, videogames designs and CAD programming.

Visual Components offers the possibility to get and use all the different features that the program has. Using both VC libraries and other that are imported, any programmer is able to combine between the basics of programming and features from other programs and the ones that the program already gives. Since the script are Python-based, libraries can be easily imported in the normal way, which is locating the library folder in the path that the language uses to import different libraries to the script. Let´s give an example:

Visual Components libraries don´t offer the possibility to export values of variables into text files, so it´s necessary to import a python library which is previously downloaded. The library that we´re using is called “pathlib”. The only difference between python inside the program, and python downloaded from a 3rd party, is that “inside” python doesn´t have the option to interact with other programs, so first we install a second version of python, and we install from the command shell the executable “pip”, using the command “pip-install”. Once installed, we use this new program to download the library into the new version of python using again the command shell (let´s suppose that the pathlib library doesn´t belong to the basic libraries of

Python). Now that we have the folder, we simply move the folder from the library folder inside the new version of python to the one that is inside the program. The advantage is that both paths are really similar, so there´s no problem locating both library folders:

Python VC: C:\Program Files\Visual Components\Visual Components Premium 4.2\Python\lib

Python: C:\Users\actual_user\AppData\Local\Programs\Python\Python38-32\Lib

The structure of the VC programming is level based. In order to get a component in a certain level and to copy into a variable, each superior level must be assigned too to other variables. That structure defines the relation between the visual menu and the command lines that are written underneath it. Every option in the visual menu, like the bar menu or each 3d option, is translated into language classes assigned to general variables. The main structure is defined by the following hierarchy (Figure 6):

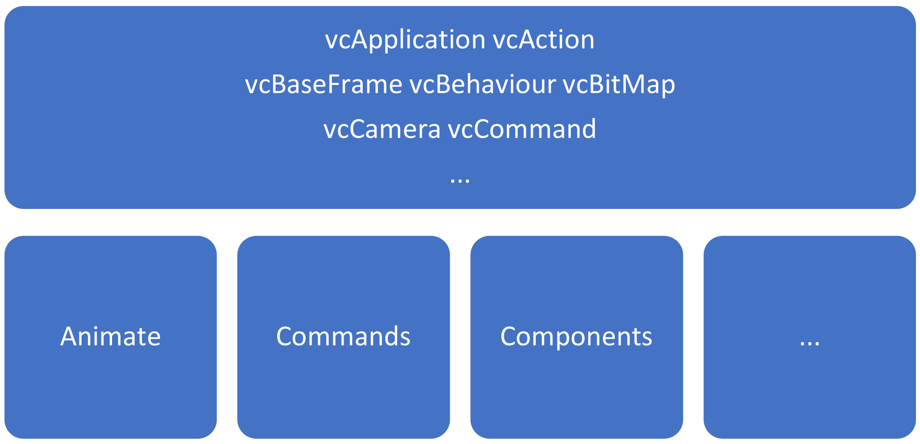


Figure 6 Hierarchical structure of functions in VC libraries

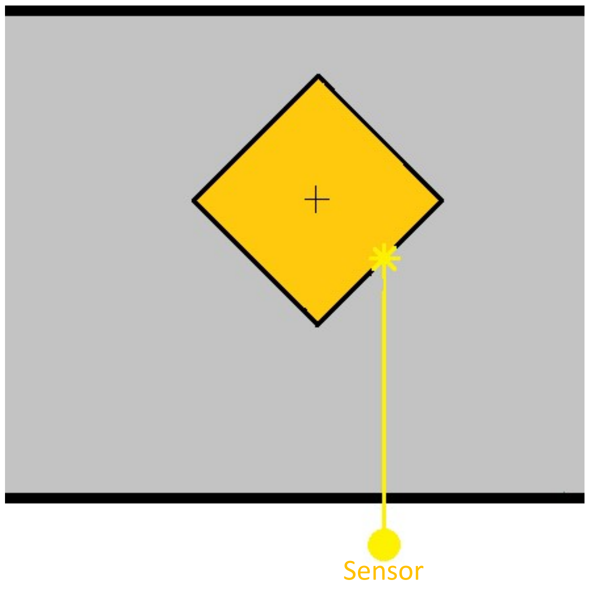
There are 4 different steps that describe the task:

First, the feeder template will create cubes at a constant period. Those cubes must be done with the same size, but in different positions and rotations. In order to get that, those cubes must be templates designed using the programming language, due to the need that both parameters must be different in each cube[7].

```
from vcScript import *
import vcMatrix
import random
app = getApplication()
#ProdBlock is the name of the cube
name_feature = "ProdBlock"
#Blockk is the name of the general component that apperars in the
Process tab
name = "Block"
#Before we create any cube, we must be sure that there´s no other cube
that´s already created. If there´s one, we delete it.
if app.findComponent(name) !=None:
  probe =app.findComponent(name)
  app.deleteComponent(probe)
def OnRun ():
  prod = app.findComponent("Feeder Template")
  creator = prod.findBehaviour("ComponentCreator")
  while app.Simulation.IsRunning == True:
    #This creates a block which size is 100 X 100 X 100
    brick = app.createComponent()
    brick.Name = name
    transform = brick.RootFeature.createFeature(VC_TRANSFORM,
"Transform")
    #Both parameters a and b defines random numbers inside the gap. a
represents the distance from the center of the conveyor, and b
represents the angle of rotation (in degrees)of the cube from its
original rotation (0°). "b" could have a gap between 0 and 360°, wich
could be logical, but since we´re using a cube, every 90° the rotation
can be reset to 0°, and the distribution from 0 to 90° has two
specular images from 0 to 45° and from 90 to 45°, so there will be no
difference using that gap.
    a=random.randint (-200, 100)
    b=random.randint (0,45)
    #The transforming expressions are text lines that describes in the
space modifications in scales, translations and rotations. Those
expressions are organized in the following structure: A letter symbol,
attached to the desired direction and the value in brackets,
everything in string variables. In case that you want to write more
expressions to the same object, those must be separated by a dot.
    transform.Expression = 'Ty('+str(a)+'). Rz('+str(b)+')'
    block = transform.createFeature(VC_BLOCK, name_feature)
    creator.TemplateComponent = brick
    part = creator.create()
    #Once created, the program will wait some predefined time until it
deletes the cube as pattern and creates a new one.
    delay(creator.Interval)
    app.deleteComponent(brick)
```

All the script is commented in green, with the description, step by step, that the program does in order to send cubes in a regular period of time. One thing to be noticed is that, during the simulation, the program creates the cube, but there´s no written command to hide the one that is used as pattern (everything that we do to that cube, the cube in the conveyor will be modified as well), so there will be two cubes in the 3d view for each time that we create them, but there will be only one that cross the feeder template.

Then, once that those cubes are created, they are moved into the conveyor to be collected. In the conveyor, there´s a raycast sensor that detects if there´s an object blocking its way. If an object is detected, then the raycast will return a real value with the distance measured. At first, the raycast won´t calculate final results, because in real life a distance sensor doesn´t have any processor to calculate any geometrical relations, so it must be the robot that will get the angle and center position depending on the text file that will receive from the raycast (more realistic). This script interacts too with the script of the other conveyor and the script from the UR5, since those must synchronize showing a sequential process. First the cubes will be moved to a comfortable place to the robot so it can be grasped easily. Then, with the information given by the raycast, the robot will pick the cube properly and it will place it in the pallet in the other conveyor, which is already placed before the robot needs it. Since the pallet has 6 holes, the robot will fill all the places with the cubes until the pallet is complete, which is the moment that the pallet will keep going and a new pallet will replace it. Is important to highlight Figure 7: The zenith view of a cube being read by the sensor



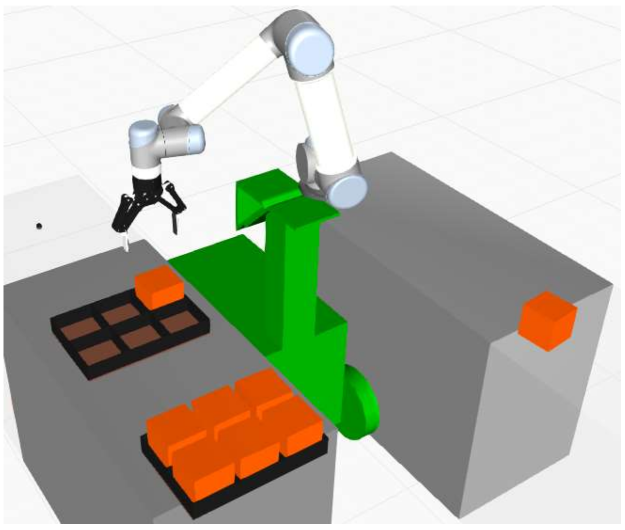


Figure 7 and 8. Zenith view of the first conveyor and 3d view of the operation

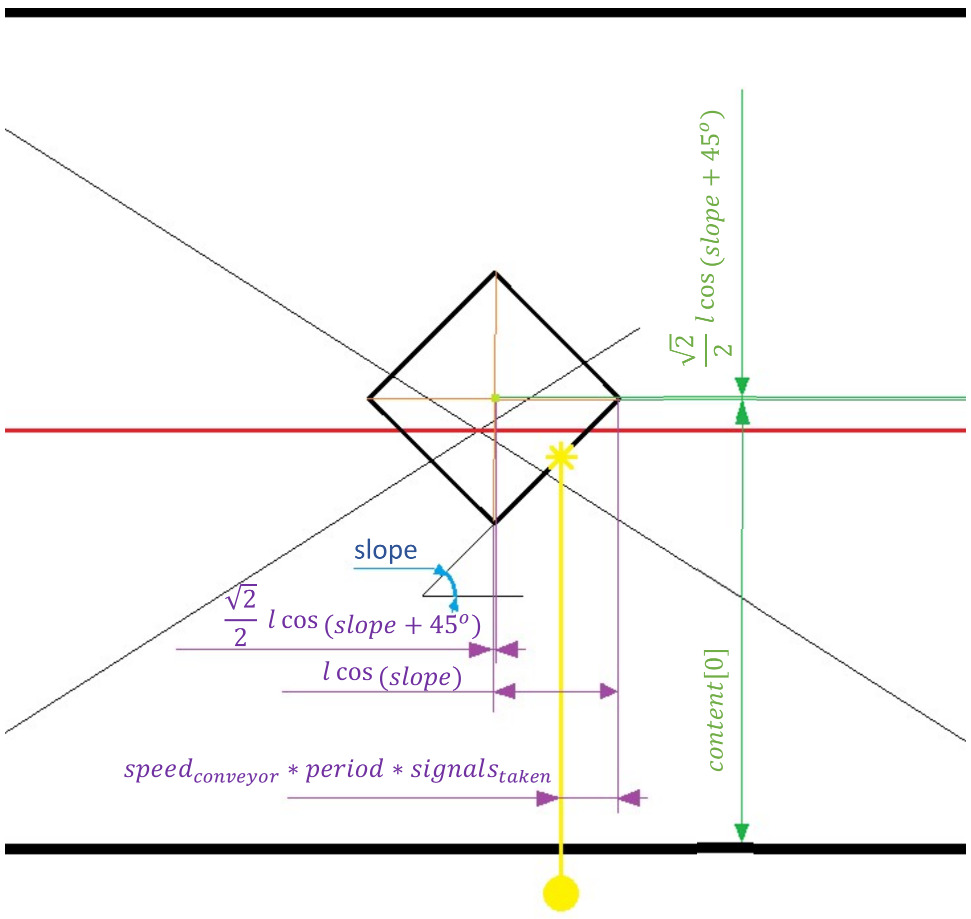
```
from vcScript import *

import os
import pathlib

#This command (pathlib.Path.cwd()) gets the path where the program is
executed and is return in variable type WindowsPath. An example can be
the desktop of the user, so the variable will have the value of
"C:\Users\actual_user\OneDrive\Desktop\Proyecto
Final\Proyecto\data.txt"
path = str(pathlib.Path.cwd()) + "data.txt"
```

```
comp = getComponent()
#Since the things that we want to modify is from the Raycast sensor
called RaycastSensor, and the script is already in
StraightConveyorTemplate, there´s no need to get application and then
go to the component StraightConveyorTemplate (the program understands
that, if the script is located in a component, that component must be
included in the script automatically)
sensor = comp.findBehaviour("RaycastSensor")
a = 0
#OnSignal(signal) is a function that is executed only if in the
simulation there´s one signal sent in the same component here. That
signal must be sent during the execution of OnRun located in the same
script
def OnSignal( signal ):
  global a
  global path
#When the signal sent (in this case is the signal of the raycast
sensor, which returns distances) is real type, then we´ll get 10
samples of different distances from the raycast to the cube, which
will be sent to the robot exporting it in a text file called
"data.txt" (as we can see from the path variable). Then we´ll stop the
conveyor so the cube can wait until is grasped by the robot, and we
will repeat with next cube
  if signal.Type == VC_REALSIGNAL:
    if sensor.DetectionContainer.Enabled == True:
      pos = signal.Value
      file = open(path,"a")
      file.write(str(pos) + " " +"\n")
      file.close()
      a = a+1
      if a % 10 == 0:
        sensor.DetectionContainer.Enabled = False
#OnRun() is the main function that is executed during the execution of
the simulation. Is executed only once, and is ideal to set first
values to the global variables and leave them to be modified later. In
this script, the function will check if the text file already exists,
and (just in case) is removed so it can be created again and clear.
Then, every 0.01 seconds the Raycast will send a distance signal using
the boolean variable "Pulse_Raycast"
def OnRun():
 global path
```

```
if os.path.exists(path):
   os.remove(path)
app = getApplication()
pulse = comp.findBehaviour("Pulse_Raycast")
while app.Simulation.IsRunning == True:
  pulse.signal(True)
  delay(0.01)
```

All the script is commented in green describing step by step the process with the raycast sensor

Now that the cube is ready to be picked, the robot receives the text file with the distances that the raycast took from the cubes, assigned to a vector variable called content and those will be transformed into position and desired orientation (slope). Figure 9 shows the trigonometrical relations between a cube with known side "l" and the parameters needed. With those equations, we can get the x and y position of the center of the cube, and the angle that is rotated (having as reference the red line).



$\frac{\sqrt{2}}{2}\, l \cos(slope + 45^o)$
slope
$\frac{\sqrt{2}}{2}\, l \cos(slope + 45^o)$
$l \cos(slope)$
$content[0]$
$speed_{conveyor} * period * signals_{taken}$

Figure 9. Known parameters and variables

$$x = \left(x_{raycast} + \ speed_{conveyor} * period * signals_{taken}\right) - lcos(slope) + \frac{\sqrt{2}}{2}\, l\cos\left(slope + 45^{o}\right)$$

$$y = \left(y_{raycast} + content[0]\right) + \frac{\sqrt{2}}{2}\, l\cos\left(slope + 45^{o}\right)$$

$$slope = \arctan\left(\frac{content[0] - content[9]}{speed_{conveyor} * period * signals_{taken}}\right) + Error$$

The measurements that the raycast does are not as accurate as we wanted, because the values that it collects values with only 3 decimals, so there will be an error attach to lack of information. However, between 0 and 45º there´s not much difference if we add to the result 2 more degrees.

```
from vcScript import *
from vcHelpers.Robot2 import *

import math as mt
import os
import pathlib

content = [0,0,0,0,0,0,0,0,0,0]
#path must be the same as in the previous script, since the robot is
the one that reads that file
path = str(pathlib.Path.cwd()) + "data.txt"
#Here we get all the information of the components that we need to be
controlled: the UR5 robot, both conveyors, and the Raycast Sensor
comp = getComponent()
robotExecutor = comp.findBehavioursByType("rRobotExecutor")[0]
robotProgram = robotExecutor.Program
mainRoutine = robotProgram.MainRoutine
app = getApplication()
conveyor = app.findComponent("StraightConveyorTemplate")
sensor = conveyor.findBehaviour("RaycastSensor")
conveyor_2 = app.findComponent("StraightConveyorTemplate #2")
sensor_2 = conveyor_2.findBehaviour("RaycastSensor")
```

```
robot = getRobot()
#switcher is a simple function that returns the value of the new row
(fil) and column (col) depending on the state of the fill of the
pallet
def switcher(state, fil, col):
  a = 0
  b = 0
  if state == 1:
    a = 0
    b = 0
  elif state == 2:
      a = 0
      b = 1
  elif state == 3:
      a = 0
      b = 2
  elif state == 4:
      a = 1
      b = 0
  elif state == 5:
      a = 1
      b = 1
  elif state == 6:
      a = 1
      b = 2
  return  a, b

def OnRun():
  global content
  global path
  sum = 0
  col = 0
  fil = 0
  state = 0
 #Routine refers to the selection of the different operations that the
robot can make during the simulation. Every operation with that suffix
makes different modifications either to the main structure or the
subroutines, including creation, removal and skipping of the execution
of those parts.
  mainRoutine.clear()
  subroutine = robotProgram.findRoutine("Grasp")
  if subroutine:
    subroutine.clear()
  if not subroutine:
    subroutine = robotProgram.addRoutine ("Grasp")

#the command .addStatement attach at the end of a subroutine a new
command which is characteristic of the robot. the different statements
created are grasp and release actions.
  statement = subroutine.addStatement(VC_STATEMENT_SETBIN)
  statement.OutputPort = 100
  statement.OutputValue = True
  statement = subroutine.addStatement(VC_STATEMENT_SETBIN)
  statement.OutputPort = 1
  statement.OutputValue = True
  subroutine = robotProgram.findRoutine("Release")

  if subroutine:
    subroutine.clear()

  if not subroutine:
    subroutine = robotProgram.addRoutine ("Release")

  statement = subroutine.addStatement(VC_STATEMENT_SETBIN)
  statement.OutputPort = 100
  statement.OutputValue = False
  statement = subroutine.addStatement(VC_STATEMENT_SETBIN)
  statement.OutputPort = 1
  statement.OutputValue = False
  robot.driveJoints(0, -90, 0, -90, 0, 0)

  while app.Simulation.IsRunning == True:
    delay(0.1)
    if sensor.DetectionContainer.Enabled == False:
      file = open(path,"r")
      for x in xrange(10):
        content[x] = float(file.readline())
      file.close()
      os.remove(path)
      sum = content[0]-content[len(content)-1]
      slope_r = mt.atan(sum/(200*0.01*len(content)))
      slope_d = -(180-(mt.degrees(slope_r)+2))
      x_object = 682.669 - 200*len(content)*0.01+100*mt.cos(slope_r)-
100*mt.cos(slope_r+mt.radians(45))/mt.sqrt(2)
      y_object = 1550.964 - content[0]-
100*mt.cos(slope_r+mt.radians(45))/mt.sqrt(2)
      robot.jointMoveToPosition(x_object,y_object,1000,0,0,-
slope_d,"world")
      robot.linearMoveToPosition(x_object,y_object,750,0,0,-
slope_d,"world")
      robot.callSubRoutine("Grasp")
      delay(1)
      robot.jointMoveToPosition(x_object,y_object,1000,0,0,-
slope_d,"world")
      robot.driveJoints(0, -90, 0, -90, 0, 0)
      state = state + 1
      fil, col = switcher(state,fil,col)
      x_release = 562.836 + fil * 118.8
      y_release = 379.594 - col * 118.8
      if sensor_2.DetectionContainer.Enabled == False:

robot.jointMoveToPosition(x_release,y_release,1000,0,0,90,"world")

robot.linearMoveToPosition(x_release,y_release,750,0,0,90,"world")
        robot.callSubRoutine("Release")
        delay(1)

robot.jointMoveToPosition(x_release,y_release,1000,0,0,90,"world")
        robot.driveJoints(0, -90, 0, -90, 0, 0)
        sensor.DetectionContainer.Enabled = True
        if state == 6:
          sensor_2.DetectionContainer.Enabled = True
          state = 0
```

All the script is commented in green describing step by step the process with the UR5.

Last script is the one that the second conveyor has to control where to stop and when to run, once that each pallet is full.

```
from vcScript import *

comp = getComponent()
sensor = comp.findBehaviour("RaycastSensor")
```

```
b = 0
c = 0
sbs = True
def OnSignal( signal ):
  global b
  global c
  global sbs
  if signal.Type == VC_REALSIGNAL:
    if sensor.DetectionContainer.Enabled == True:
      if sbs == True:
        if signal.Value < 500:
          b = b+1
          if b % 13 == 0:
            sbs = False
            sensor.DetectionContainer.Enabled = False
      elif sbs == False:
        c = c+1
        if c % 70 == 0:
          sbs = True
          c = 0

def OnRun():

 app = getApplication()
 pulse = comp.findBehaviour("pulse_Raycast")
 while app.Simulation.IsRunning == True:
   pulse.signal(True)
   delay(0.1)
```

Since this script is quite similar to the first one, there´s no need to comment it.

## RESULTS

Since our results are the simulation videos showing the behavior of the robot, Figure 10 and Figure 11 represents a frame of those videos. Underneath them, there´s a link that connects them to two pdf files, which are the real results.

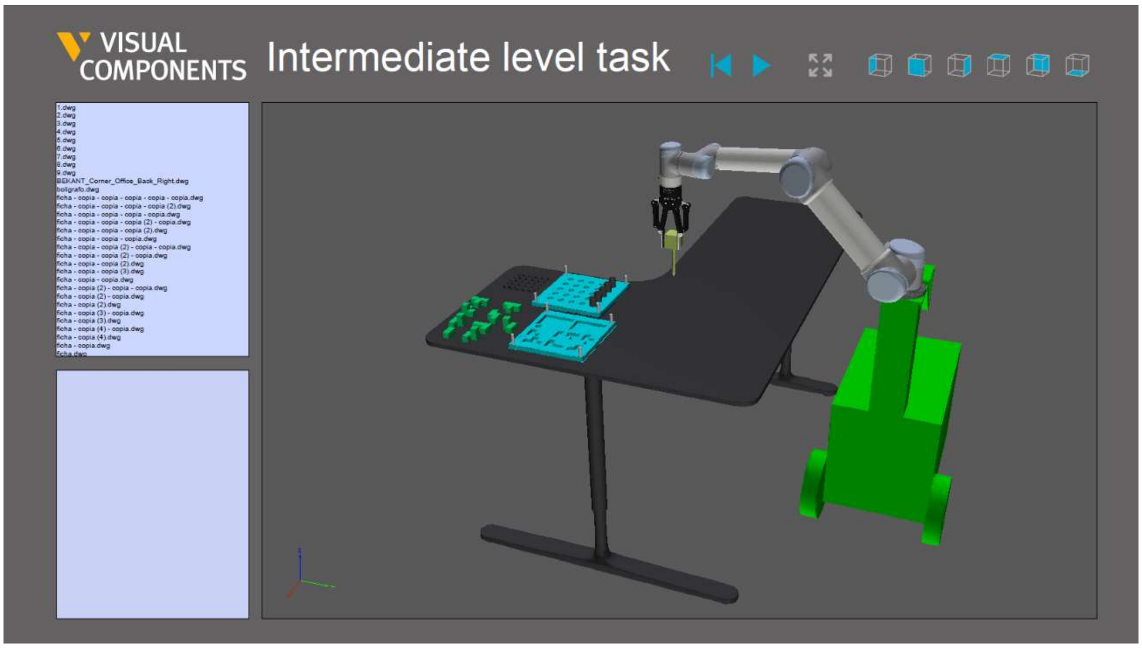

VISUAL COMPONENTS
Intermediate level task
1.dwg
2.dwg
3.dwg
4.dwg
5.dwg
6.dwg
7.dwg
8.dwg
9.dwg
BEKANT_Corner_Office_Back_Right.dwg
boligrafo.dwg
ficha - copia - copia - copia - copia - copia.dwg
ficha - copia - copia - copia - copia (2).dwg
ficha - copia - copia - copia - copia.dwg
ficha - copia - copia - copia (2) - copia.dwg
ficha - copia - copia - copia (2).dwg
ficha - copia - copia - copia.dwg
ficha - copia - copia (2) - copia - copia.dwg
ficha - copia - copia (2) - copia.dwg
ficha - copia - copia (2).dwg
ficha - copia - copia (3).dwg
ficha - copia - copia.dwg
ficha - copia (2) - copia - copia.dwg
ficha - copia (2) - copia.dwg
ficha - copia (2).dwg
ficha - copia (3) - copia.dwg
ficha - copia (3).dwg
ficha - copia (4) - copia.dwg
ficha - copia (4).dwg
ficha - copia.dwg
ficha.dwg

Figure 10. Intermediate level task pdf (link)

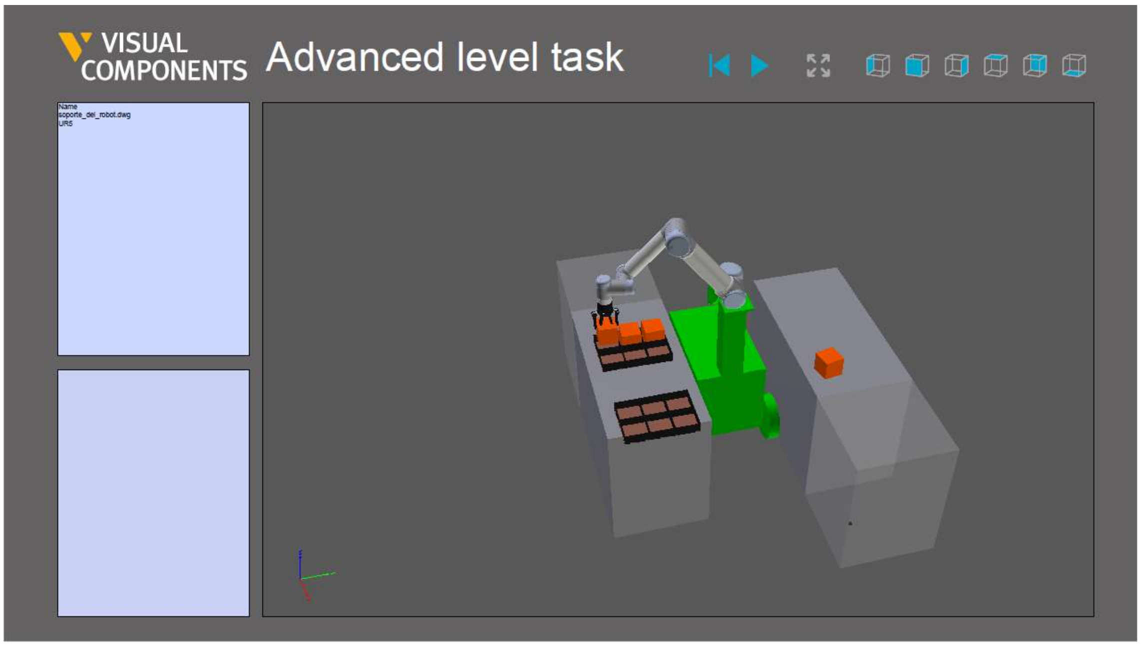


Figure 11. Advanced level task pdf (link)

# DISCUSSION

The process in which the project was done was previously optimized to get the best results in the most efficient way. Not referring in time only, but in resources and accuracy As we said in the introduction, the UR5 series is perfect to study this kind of statements, because we´re using a robot with high degree of safety, which is prepared too to work in an industrial environment, and is easy to program in order to get good results.

## ROBOT SELECTION

It is provided several robot models with different characteristics and similarities in some functions: One *KUKA* robot, one *ABB* robot, and a Dobot Magician, besides the UR5 *cobot* that we finally chose. Although those weren´t a good option if we wanted to elaborate several tasks in the university laboratory, in some aspects they could be better than the chosen one: Dobot Magician, for example, is a small robot with 4 DOF which SO can be programmed using either an Arduino board or a Raspberry Pi. This robot, made by Shenzen Yuejiang Technology, is made for high accurate tasks in a small workspace, such as CNC operations, laser engraving, automatic witting…, as long as the objects that it grasps (if it has to grasp anything) are light, due to its low joint resistance. In other words, the size and weight holding of the robot makes it impossible to do any of the tasks. However, in the beginning this robot was planned to be used in one of the tasks as an auxiliar robot that places different dimensional objects into a conveyor that, lately, they will be processed by the UR5.

The big problem appeared when the tasks had to be changed, keeping as much as possible into the simulation world. In the end, Dobot magician was replaced by the Feeder Templates that made the same operations. On the other hand, both robots (*KUKA* and *ABB* robots) are big and can lift a big weight without problem. They even have a similar programming language as Polyscope for UR5 (RAPID language, for ABB, KRL, for KUKA, and Polyscope are based in Pascal language, making them structurally similar). However, they weren´t rejected due to their capability, but to the working environment. The working environment where tasks were going to be made was in the laboratory of the university. This laboratory, is a non-isolated space where you could make your own researches related to electronics and automation. This affirmation implies that anyone can access to the workspace of the robot without any kind of security so, in order to minimize any risk, we had to choose a robot that can work in a risky space if there´s any incident while executing any task. In other words, we need a *cobot*, but those robots aren´t. So, all in all, the UR5 robot is the only robot that can achieve those requirements. If the environment was different, we wouldn´t have chosen it, and we would have used either of the other two robots.

The tasks required that the robot in every moment was still, screwed to a position that can´t be pushed, so things like stability or external information about the speed of the robot is not important, and all the characteristics in the datasheets must be what makes us choose. UR5, although is perfect for an educational environment, it isn´t the most prepared for industrial tasks in an isolated space. When that situation happens, Characteristics like speed, repeatability, force and precision are decisive so we can get the best results in the minimum of time: Comparing between three common robots from the different companies with same joints and sizes, like UR5, IRB120 and KR120 R2700-2 (from Universal Robots, ABB and KUKA, respectively), UR5 can´t take any weight that the other ones can. Is not prepared for big power wasted in move big weights. Both IRB120 and KR120 R2700-2 are quite similar, in terms of efficiency. Both robots are quite common to use in big industries, where the processes made are related to big objects that must be modified, or must be stocked in packages. Of course, those robots aren´t the same. IRB120 is better prepared to do some kind of tasks that the KR120 isn´t, and vice versa, due to their morphology and software. Their programming languages and status devices are ideal to control all different states in the variables during execution, acting quickly by, for example, pressing the emergency robot if anything goes wrong.

Is important to say that nowadays, due to the competition between the big robot companies, the new products offered are becoming more and more versatile, so they can adapt themselves properly in any kind of working environment, so choosing between some series or others is slowly more difficult. On one hand, that´s a good new for customers: general tasks require general robots, and since there are possible options, you can get the cheapest one without losing quality. On the other hand, in the industry new special and specific tasks are appearing, so there´s more and more need in buying robots that abandon generic tasks and focus on be prepared in specific tasks, like mechanical operations.

## ACCESORY SELECTION

Here, you have either the option to buy a robot that only can do that, or a robot that can adapt its TCP to different tools, so you can buy only one robot with several different tools that can be attached automatically during the process, minimizing times in changing the manufacturing lot. Due to its space, time and money saving, this option is nowadays the best one to make different operations to the same material in order to improve the chain mounting manufacturing style. Luckily, our final project doesn´t have to change its tool, since all the tasks must be made with a gripper. Other tool could be useless to make them, so it was important to take a gripper small enough not to interrupt the process and be handy, but big enough to take normal pieces. In this case, due to lack of options, Wrist camera and 2F-85 Robotiq® Gripper were chosen to make the grasp and release actions, although it has been a very good and complete accessory, since it was available to be programmed and simulated in Visual Components. Needless to say, that those aren´t the only devices that have helped in the thesis. However, the rest of them weren´t used in real life, but in simulation, like the Feeder templates, the Conveyors and the Raycast.

## PROGRAM SELECTION

The main advantage of Visual Components, is that is very comfortable to use and is very complete to offer any kind of action that the user can ask for. This is due to its base programming language. Visual Components, as we said earlier, is a program which is based in Python, and is prepared to offer all kind of options, shown in a well-structured and intuitive interface. Since it´s programmed in this language, it can take all the features from other languages with the advantage that programming in it becomes much easier to a normal user. Since it´s a high-level program, the basic tasks in which is programmed are quite complex seen by the computer, but simpler seen by the user. The main reason is that all functions, compilation and programming structure are modified so they can assign to the same group a bunch of atomic tasks in the same command.

As a user, is more comfortable to write each time one line instead of a group every time that wants to write a big function, and since is well structured, all modifications, corrections and comments can be quickly located during programming. Besides, due to the dynamic compilation, possible errors that can happen are warned during the execution, so everything that happened before that point, can be seen if it´s working correctly. However, for the computer the language is not fully optimized. Since the commands are structured in basic actions, is common to see that there are several lines inside them that are useless or aren´t the best option to solve the task. That makes this language slower than others, like C.

Time is not only the only defect, because the language doesn´t save object lines, making it wasting a lot of resources for nothing. In simple programs, since nowadays processors are powerful enough to manage them without breaking any deadline, Real Time process that requires critical tasks can be impossible to be well processed by some processors. In those situations, other languages that have low-level commands are better to be used. In our case, the tasks programmed don´t require short deadlines and can access to enough resources without assuming big delays, so executing both program and simulation isn´t a problem using it. If Visual Components was based in other language, like C++, it would have been different. C++, for example, is a programming language made to program functions in the quickest and simplest way, from the computer perspective. The interface that could have been created would have been much simpler in terms of options in the menu, or the capacity of different features would have been poorer, or, if the interface had the same interface as with Python, the program would be much larger in space. Memory would begin to be an important aspect in the program installation, due to the need of writing more lines for the same result comparing to Python language.

On the other hand, the lines would have been fully optimized so the computer can have the possibility of executing more tasks at the same time, or the program can even be noticeable faster than expected, improving other characteristics in its aspect. However, C++ is not prepared to develop programs like VC, so the large amount of command lines and the lack of dynamic compilation would make things impossible to develop new versions of the program and fix bugs and issues that the customers could have along the time. In these situations, is much better to have dynamism that to have stability.

The combination of an easy language program and a user-oriented interface are the best option to program simulations of robotic tasks. In fact, Visual Components is one of the few simulation programs that offer that number of features. From simple CAD modelling to real life connectivity, Visual Components give to the customer the experience of managing an industrial building with workers and different areas (can be either working areas or recreational areas) to represent as close as possible a normal industrial building so good decisions can be made in every kind of situations even before they can happen in real life. This type of programs are often used to design critical or emergency situations which can´t be done in real life due to the consequences that can appear afterwards. For example, having designed your entire industrial plant, you can set different events like an area burning, or breaking a machine, to see what could be the best idea to do so the workers are in the safest possible way or the process is interrupted the less possible amount of time, and redesign the actual place so, saving time and money, the building is prepared to that situations.

Sometimes, you would need any external machine, because it can´t be simulated due to its nature or is inefficient to do it, and can be physically connected using Ethernet or TCP/IP connection. That proves that this program can even set the necessary parameters based on a real machine with real data. In fact, is even possible to control an entire installation from a single computer, (in certain situations, where machines can make simple tasks and they don´t need to be made in critical deadlines), so the owner doesn´t have to buy an external automaton, like a PLC.

Programs like Visual Components aren´t as common as we could think, because they are usually prepared only to a specific type of environment. Some of them, they don´t offer an open 3d view to import CAD files, and the only environments that you can import are the ones that the program can offer you. Those programs are usually prepared for structural simulations where is not important how the process is done, but if the process is done. They usually calculate statistics parameters based on the nature of the environment and all the attempts render with different strategies and different values on variables. ARENA Simulation is a good example of this group. This program, created by Rockwell Automation, creates any kind of industrial process by the actions that have to be made in order to get a final result. The program is completely oriented to get a final table with different parameters describing how good was the attempt, rejecting any kind of detailed simulation and rearranging basic movements of different machine and workers in average times and delays. This, as well, has its pros and cons, because if the simulation is well prepared, the program offers really accurate results related to time and resources processing, average money needed to change the production, and efficiency, assuming that, for example, all kind of events are contemplated and controlled, including random ones, but it is not capable to get the information of everything that happens between the beginning or the end. When, where and how questions aren´t included in the study, which are sometimes useful in some aspects.

Other programs are oriented to CAD simulation, where objects can have their physics properties and can be simulated like it happened in real life. Those programs are programmed to simulate that based on physics equations that are demonstrated to work in those simulations with almost any error. However, those programs are just prepared for specific situations were there aren´t any interference or interactions with non-controlled elements: Everything that is connected, we know where and how they are connected. With this idea, they are perfect programs to get relations that can help studying any mathematical hypothesis. If we had, for example, a pipe where a gas is flowing across it and we want to study its speed profile, the program can simulate it by using, for example the Reynolds transport equation combined with the P,V,T relation that the gases have. Adding other parameters like friction coefficient from the material that the pipe is made for, we can set a diagram showing the different values of the speed depending on the position, pressure and geometry of the pipe, in perfect conditions. If we wanted to study what could happen if the gas leaked at any point of the pipe, the program should must know about the properties of the hole that is letting the gas to be leaked, including more equations and parameters that are needed if the program have to keep simulating it, as a controlled constraint.

That generates two big problems studying different interferences that affect to the system: first, is that those interferences imply big equations to be solved and, in consequence, much more space to be used, and second is that the program can be only useful if we know all the information about them, so if we only know that there´s a hole, and we don´t know what´s it shape or how big is it, the program is not able to generate the equation results, so, in the end, it provides a lot of information as long as everything is controlled. For simulations where things are usually random or can´t be controlled and the information needed is not that complete, these types of programs are completely useless. For an industrial process where is important how to describe the process in terms of production, is necessary to look for a program that has features from both worlds. A program that has a 3d world where the actual states can be seen in real time, but both interface and calculus behind it must be oriented into production.

## RESOLUTION METHOD

The effectiveness of a program is defined by its capability of problem resolutions, giving acceptable results in the fastest possible way. In order to solve robot movements and waypoints, the program (and the robot) must have an affordable option to get the solutions during the time between information is shown and the critical moment when next task must start in the processor. There are several ways to solve how a robot must solve the angles of the joints depending on the desired waypoints, or vice versa. In order to determine which one is better, is necessary to talk about the theory behind it.

Physically, a robot system is a group of different joints, in which origin points are placed in sequence, and the distance from one joint to the next is defined by a specific function, which can be either constant (which has a rigid connection) or a function from the joints described before. It always must have known the morphology of the robot described as equations, and a fixed origin frame where all joints are related to. The objective from that group will be always to describe the movement of the final joint, using the position, speed and acceleration from the previous ones. In order to get it, the equations can be either described as parametric equations defining the x, y and z position of the TCP from the base frame, or a matrix, which elements can describe different aspects from the actual joint and the previous ones. The generic matrix that describes position and orientation of the actual joint is called Transformation matrix. That matrix can be modified using translation, rotation and scale so the numeric result is identical as if we made the same variations in the 3d world. The Transformation Matrix is never only one, in which for one system, can be described in different matrixes and all of them can be right, since those matrixes can be defined from different frames which don´t have to be orthonormal. When a point can be only modified by a rotation and translation, only if we are using one specific orthonormal frame, the matrix is named Homogeneous Transformation Matrix. This matrix is a 4x4 matrix where can be separated in 3 main groups:

$$
{}^{A}T_{B} = \begin{array}{c}
\begin{array}{cccc} & 1 & & 2 \end{array} \\
\left[\begin{array}{ccc|c}
x_x & x_y & x_z & d_x \\
y_x & y_y & y_z & d_y \\
z_x & z_y & z_z & d_z \\
\hline
0 & 0 & 0 & 1
\end{array}\right] \\
3
\end{array}
$$

The first group, located in a 3x3 “submatrix”, represents the rotation given to the frame of a point in the space seen from the previous frame. The number $a_b$ represents the b coordinate of the a-axis. Due to the orthonormal condition indicated before, each row of this subgroup must be normalized, so:

$$\sqrt{a_x^2 + a_y^2 + a_z^2} = 1 \text{ (With a as x, y or z)} \qquad (1)$$

Second group, seen as a 3x1 “submatrix”, defines the position of the origin of the frame from the previous frame, in coordinates.

Last group is one auxiliar row that we use just to be able to make the respective operations of rotation and translation. Technically, the third group can be divided in two subgroups: the first one, which is the first three columns, that indicates the perspective of the sight, and the fourth column, which shows the scale. Due to the usage of math for robotics, where we´re using only points and there´s no change in perspective, we´ll always set those values to 0 0 0 1.

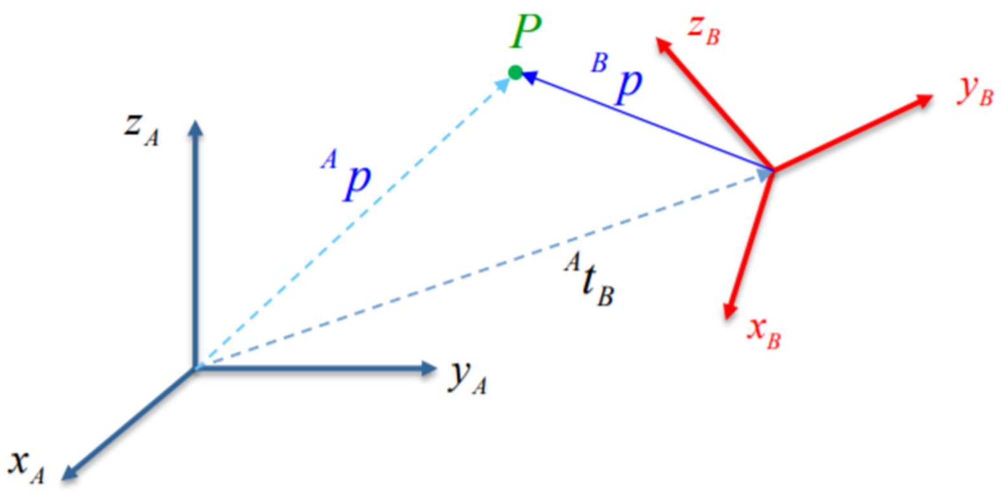


Figure 12. Different coordinates for the same point

So, in order to set one point in the space from two different frames like in Figure 12:

$$^{A}p = {}^{A}T_B \cdot {}^{B}p \qquad (2)$$

Where $^{A}p$ and $^{B}p$ are the vectors of position-scale of point P from the A/B frame. As we can see, the change of frame doesn´t depend on the direction where we are changing the frame, so, if always exists (2), then:

$$^{B}p = {}^{B}T_A \cdot {}^{A}p \qquad (3)$$

Will always exists. So, replacing $^{B}p$ in (2)

$$^{A}p = {}^{A}T_B \cdot {}^{B}T_A \cdot {}^{A}p$$

That can be only possible if:

$$^{A}T_B \cdot {}^{B}T_A = I_4 \rightarrow {}^{A}T_B = {}^{B}T_A{}^{-1}$$

So, knowing that both matrixes always exist at the same time, for each transformation matrix there´s always an inverse of that matrix. We can demonstrate that showing that

$$|{}^{A}T_B| \neq 0$$

Let´s use Chio´s rule in the last row:

$$\begin{vmatrix} x_x & x_y & x_z & d_x \\ y_x & y_y & y_z & d_y \\ z_x & z_y & z_z & d_z \\ 0 & 0 & 0 & 1 \end{vmatrix} = -0 \cdot \begin{vmatrix} x_y & x_z & d_x \\ y_y & y_z & d_y \\ z_y & z_z & d_z \end{vmatrix} + 0 \cdot \begin{vmatrix} x_x & x_z & d_x \\ y_x & y_z & d_y \\ z_x & z_z & d_z \end{vmatrix} - 0 \cdot \begin{vmatrix} x_x & x_y & d_x \\ y_x & y_y & d_y \\ z_x & z_y & d_z \end{vmatrix} +$$

$$+1 \cdot \begin{vmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{vmatrix} = \begin{vmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{vmatrix} \text{(4)}$$

First thing that we can see from this, is that the determinant of a transformation matrix doesn’t depend on the distance from the origin frame when there´s no change in perspective (which is completely logical because only if we change perspective, the dimension of the objects and positions are modified according to certain deformations). Due to our environment, which is robotics, that never changes perspective (we can assume that we´re always in a constant 3D world), distances will never affect the system.

Before we continue, let´s remember the properties of our frames. The frames that we´re using must be orthonormal, which are defined as vectors orthogonal to each other one by one, and each vector must have module 1. In other words:

$$\begin{vmatrix} i & j & k \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{vmatrix} = x_x \cdot i + x_y \cdot j + x_z \cdot k$$

So, using Chio´s rule for the first row:

$$i \cdot \begin{vmatrix} y_y & y_z \\ z_y & z_z \end{vmatrix} - j \cdot \begin{vmatrix} y_x & y_z \\ z_x & z_z \end{vmatrix} + k \cdot \begin{vmatrix} y_x & y_y \\ z_x & z_y \end{vmatrix} = x_x \cdot i + x_y \cdot j + x_z \cdot k$$

In conclusion, all adjugated (with sign) are equal to their component of the vector (the same thing done before can be done for each vector):

$$\begin{vmatrix} y_y & y_z \\ z_y & z_z \end{vmatrix} = x_x; \ -\begin{vmatrix} y_x & y_z \\ z_x & z_z \end{vmatrix} = x_y; \ \begin{vmatrix} y_x & y_y \\ z_x & z_y \end{vmatrix} = x_z \text{ (5)}$$

Developing (4) with Chio´s rule in the first row and substituting (5) in (4):

$$\begin{vmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{vmatrix} = x_x \begin{vmatrix} y_y & y_z \\ z_y & z_z \end{vmatrix} - x_y \begin{vmatrix} y_x & y_z \\ z_x & z_z \end{vmatrix} + x_z \begin{vmatrix} y_x & y_y \\ z_x & z_y \end{vmatrix} = x_x^2 + x_y^2 + x_z^2 = 1 \neq 0$$

So, transformation matrix in robotics will have always determinate equal to 1, showing that will always have an inverse. Other property is that the inverse of the rotation matrix is always the rotation transposed.

Let´s inverse the rotation matrix:

$$\begin{pmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{pmatrix}^{-1} = \frac{1}{\begin{vmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{vmatrix}} \cdot \begin{pmatrix} \begin{vmatrix} y_y & y_z \\ z_y & z_z \end{vmatrix} & -\begin{vmatrix} x_y & x_z \\ z_y & z_z \end{vmatrix} & \begin{vmatrix} x_y & x_z \\ y_y & y_z \end{vmatrix} \\ -\begin{vmatrix} y_x & y_z \\ z_x & z_z \end{vmatrix} & \begin{vmatrix} x_x & x_z \\ x_z & z_z \end{vmatrix} & -\begin{vmatrix} x_x & x_z \\ y_x & y_z \end{vmatrix} \\ \begin{vmatrix} y_x & y_y \\ z_x & z_y \end{vmatrix} & -\begin{vmatrix} x_x & x_y \\ z_x & z_y \end{vmatrix} & \begin{vmatrix} x_x & x_y \\ y_x & y_y \end{vmatrix} \end{pmatrix}$$

$$= \begin{pmatrix} \begin{vmatrix} y_y & y_z \\ z_y & z_z \end{vmatrix} & -\begin{vmatrix} x_y & x_z \\ z_y & z_z \end{vmatrix} & \begin{vmatrix} x_y & x_z \\ y_y & y_z \end{vmatrix} \\ -\begin{vmatrix} y_x & y_z \\ z_x & z_z \end{vmatrix} & \begin{vmatrix} x_x & x_z \\ x_z & z_z \end{vmatrix} & -\begin{vmatrix} x_x & x_z \\ y_x & y_z \end{vmatrix} \\ \begin{vmatrix} y_x & y_y \\ z_x & z_y \end{vmatrix} & -\begin{vmatrix} x_x & x_y \\ z_x & z_y \end{vmatrix} & \begin{vmatrix} x_x & x_y \\ y_x & y_y \end{vmatrix} \end{pmatrix}$$

With the property (5) and knowing that the inverse of a matrix is the transpose matrix with all elements changed by their adjugated, divided by its determinant (which we already saw that equals to 1), the result is:

$$\begin{pmatrix} \begin{vmatrix} y_y & y_z \\ z_y & z_z \end{vmatrix} & -\begin{vmatrix} x_y & x_z \\ z_y & z_z \end{vmatrix} & \begin{vmatrix} x_y & x_z \\ y_y & y_z \end{vmatrix} \\ -\begin{vmatrix} y_x & y_z \\ z_x & z_z \end{vmatrix} & \begin{vmatrix} x_x & x_z \\ x_z & z_z \end{vmatrix} & -\begin{vmatrix} x_x & x_z \\ y_x & y_z \end{vmatrix} \\ \begin{vmatrix} y_x & y_y \\ z_x & z_y \end{vmatrix} & -\begin{vmatrix} x_x & x_y \\ z_x & z_y \end{vmatrix} & \begin{vmatrix} x_x & x_y \\ y_x & y_y \end{vmatrix} \end{pmatrix} = \begin{pmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{pmatrix} = \begin{pmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{pmatrix}^T$$

Visually, it´s quite simple to see: first, we´ll analyze two different frames from a 2D world, to simplify the method, like in Figure 13:

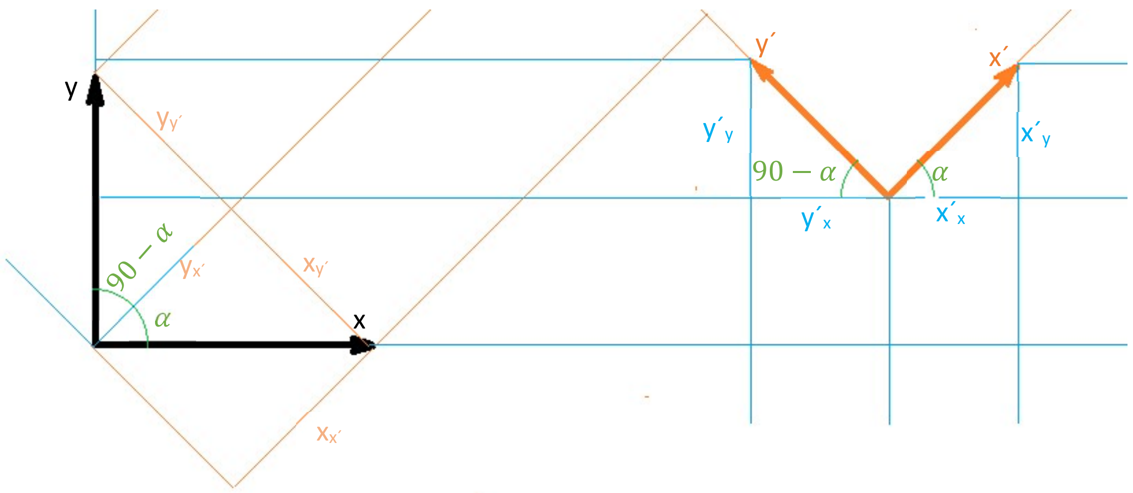


Figure 13. Visual demonstration of the transpose of coordinates

Using the trigonometric relations, we can see that:

$$x'_y = x' * sin\,(\alpha);\ x'_x = x' * \cos(\alpha)\,;$$

$$y'_x = -y' * \cos(90-\alpha) = -y' * sin\,(\alpha);\; y'_y = y' * sin\,(90-\alpha) = y' * \cos(\alpha)$$

$$x_{y'} = -x * sin\,(\alpha);\; x_{x'} = x * \cos(\alpha)\,;$$

$$y_{x'} = y * \cos(90-\alpha) = y * sin\,(\alpha);\; y_{y'} = y * sin\,(90-\alpha) = y * \cos(\alpha)$$

Knowing that x´, y´, x and y have the same module equal to 1, then we can say that:

$$x'_y = y_{x'}\,; x'_x = x_{x'}\,; y'_x = x_{y'}\,;\; y'_y = y_{y'}$$

The subgroups of each transformation matrix can be affected by the operations of Rotation and Translation:

The operation of Rotation is the matrix operation made for the rotation subgroup where represents the change by turning from one original frame to other a certain angle in the direction of one specific axis. It is represented as *R<sub>a</sub>(ϴ)*, where a is the axis x, y or z, and ϴ is the angle given of the rotation.

The general operation matrixes are:

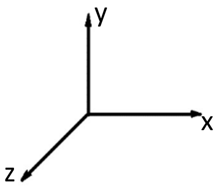
$$R_x(\Theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix};\; R_y(\Theta) = \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix};$$

$$R_z(\Theta) = \begin{pmatrix} \cos\,(\theta) & -\sin\,(\theta) & 0 \\ \sin\,(\theta) & \cos\,(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

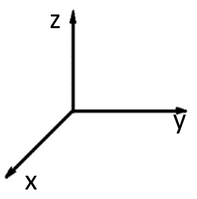
This rotation always is done by the post-product of the original frame, and for each rotation there can be only one angle:

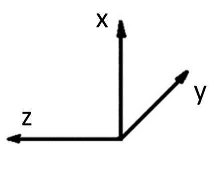
If we´re turning one frame in two different angles, the operations must be done in the correct order, because the change of one operation gives another different frame:

Let´s say that we have a normal frame with no rotation, and we want to rotate first 90 degrees around the z-axis, and then 90 degrees in the x-axis. If we do both operations in different orders:



$$T_{f,1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$





$$T_{f,2} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \end{pmatrix}$$

We will have as result two completely different frames.

The operation of Translation is the vector operation that is made for the subgroup vector Translation, and represents the movement made between origins of two different frames, describing a single linear translation. This operation is represented as $T_{x,y,z}$ ($d_x$ , $d_y$ , $d_z$ ) with $d_i$ distance made along i-axe. The general operation is:

$$T_{x,y,z}(d_x, d_y, d_z) = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$$

The translation operation can be done by the addition of the original frame, with the possibility of the translation of multiple directions, just by adding their distance coordinates:

$$\sum_{i=0}^{n} T_{i_{x,y,z}}(d_x, d_y, d_z) = \sum_{i=0}^{j} T_{i_{x,y,z}}(d_x, d_y, d_z) + \sum_{i=j}^{n} T_{i_{x,y,z}}(d_x, d_y, d_z)$$

$$\sum_{i=0}^{j} T_{i_{x,y,z}}(d_a) + \sum_{i=j}^{n} T_{i_{x,y,z}}(d_a) = \sum_{i=j}^{n} T_{i_{x,y,z}}(d_a) + \sum_{i=0}^{j} T_{i_{x,y,z}}(d_a)$$

When we want to use a robot to place some objects, and the robot (like the UR5) has some rotational joints, the only way to control it is by measuring the angle in which every joint moves, due to its electronics so, in order to reach points represented in coordinates in the workspace, we need to know how to convert a group of angles given for each joint (which we will call them $q_i$), into space coordinates. The main structure will be like this:

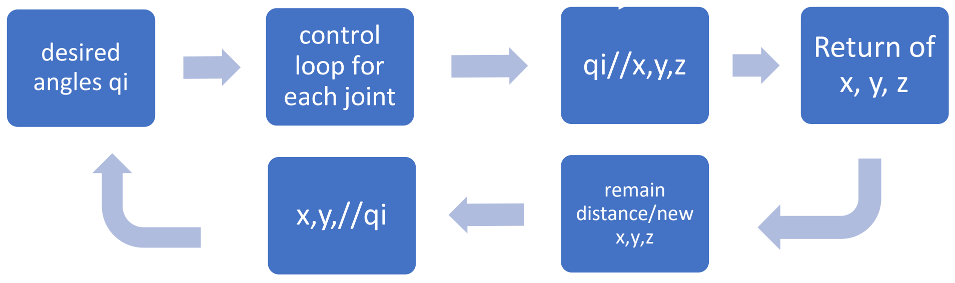


Figure 14. Fundamental robotic control loop

Both transformations that we can see in Figure 14, define the kinematics of the robot. The evolution of angles into coordinates, is called direct kinematics, and the opposite one, the inverse kinematics. There are different methods to determine the kinematics, but we will focus in three methods: the Denavit-Hartenberg parameters (for direct kinematics), the Homogenic Transformation Matrix development method (for inverse kinematics) and the geometrical method:

Geometrical method

Not suitable for all systems. The idea with this method is to describe, using the trigonometric relations, the wanted parameters in explicit equations. The wanted variables define the kinematics. Let´s see one example each:

1.-Direct kinematics

With the system given, define the x and y coordinates of the TCP knowing the angles $q_1$ and $q_2$:

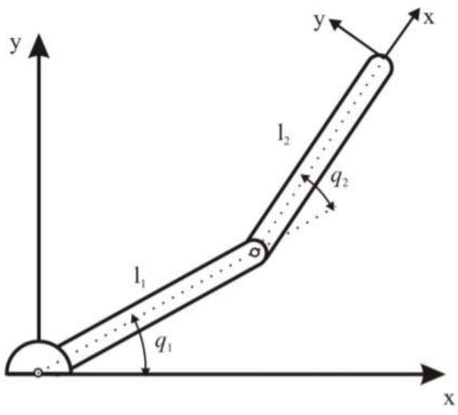


Figure 15. Graphical description of a biplanar robot

This exercise is quite simple: the idea is to add one by one the coordinates for each joint, dividing the problem in smaller pieces, so we separate Figure 5 in two smaller parts as in Figure 6.

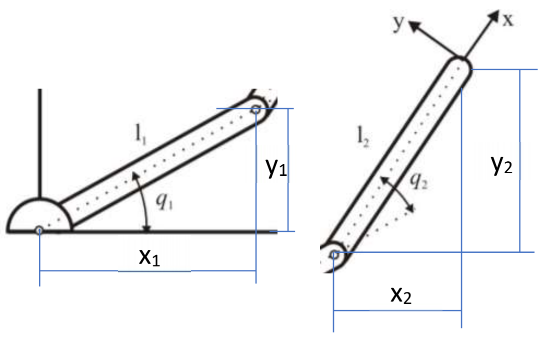


Figure 16. Statement of coordinates seen from different joints

we can determine that:

$$x_1 = l_1 \cos(q1)\,;y_1 = l_1 \sin(q_1)\,;x_2 = l_2 \cos(q_1 + q_2)\,;y_2 = \ l_2 \sin(q_1 + q_2)$$

$$x = x_1 + x_2 = \ l_1 \cos(q1) + l_2 \cos\,(q_1 + q_2)$$

$$y = y_1 + \ y_2 = \ l_1 \sin(q_1) + \ l_2 \sin(q_1 + q_2)$$

2.- Inverse Kinematics

This part is more difficult, because we need the direct kinematics to convert it again.

$$x = x_1 + x_2 = \ l_1 \cos(q1) + l_2 \cos\,(q_1 + q_2)$$

$$y = y_1 + \ y_2 = \ l_1 \sin(q_1) + \ l_2 \sin(q_1 + q_2)$$

$$x^2 + y^2 = l_1^2 + l_2^2 + 2l_1l_2(\cos(q_1)\cos(q_1 + q_2) + sin(q_1)sin(q_1 + q_2) =$$

$$l_1^2 + l_2^2 + 2l_1l_2\big(\cos^2(q_1)\cos(\mathrm{q}_2) - \cos(q_1)\,sen(q_1)sen(q_2) + sen^2(q_1)\cos(q_2) + \cos(q_1)\,sin(q_1)\sin(q_2)\big) =$$

$$= l_1^2 + l_2^2 + 2l_1l_2\big(\cos(\mathrm{q}_2)\big);\ q_2 = \arccos\,(\frac{x^2 + y^2 - l_1^2 - l_2^2}{2l_1l_2})$$

$$x = l_1 \cos(q_1) + l_2 \cos(q_1)\cos(q_2) - \mathrm{l}_2\sin(q_1)\sin(q_2)$$

$$x = \cos(q_1)(l_1 + l_2\cos(q_2)) - \mathrm{l}_2\sin(q_1)\sin(q_2)$$

$$y = y_1 + \ y_2 = \ l_1 \sin(q_1) + \ l_2 \sin(q_1)\cos(q_2) + l_2 \cos\,(q_1)\sin\,(q_2)$$

$$y = y_1 + \ y_2 = \ \sin(q_1)\,(l_1 + l_2\cos(\mathrm{q}_2)) + l_2\cos\,(q_1)\sin\,(q_2)$$

$$x + y = (l_1 + l_2\cos(q_2))\,(\cos(q1) + \sin(q_1)) + l_2 \sin\,(q_2)(\cos(q_1) - \sin(q_1))$$

$$x - y = (l_1 + l_2\cos(q_2))\,(\cos(q1) - \sin(q_1)) + l_2 \sin\,(q_2)(\cos(q_1) + \sin(q_1))$$

$$x + y + x - y = 2x = 2(l_1 + l_2\cos(q_2))\cos(q_1) + 2l_2\sin(q_2)\cos(q_1)$$

$$2x = \cos\,(q_1)\,(2(l_1 + l_2\cos(q_2)) + 2l_2\sin(q_2))$$

$$q_1 = \arccos\left(\frac{2x}{(2(l_1 + l_2\cos(q_2)) + 2l_2\sin(q_2))}\right)$$

Since both variables are the result of inverse trigonometrical operations (to get $q_2$ we need to use an arccosine, and $q_2$ depends on $q_2$), there can be more than one possible solution. This only happens in inverse kinematics, but, unless we have physical or digital restrictions (as we will see later), that will always happen. Figure 17 shows both graphical solutions for each waypoint.

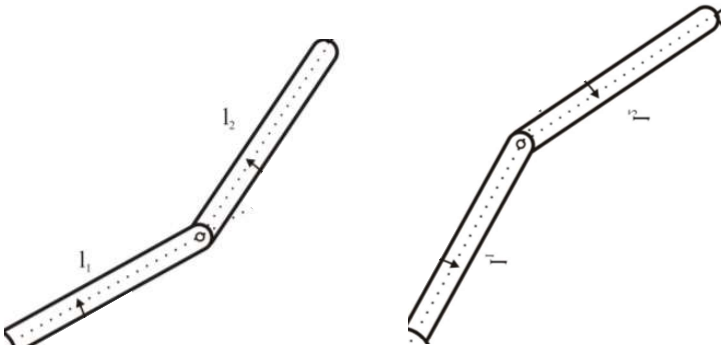


Figure 17. Two different angles reach the same position

Denavit-Hartenberg parameters method

This method, created by Jacques Denavit and Richard Hartenberg, is a convention of the Homogenic Transformation Matrix where, following some specific steps, we end with a table of parameters that we will use for various HTM, so, using the post-product, we can set a general HTM that depends entirely on the joints of the robot. In order to get those parameters, we need to do:

1.- Numerate the joints starting from the nearest joint to base with 1 until the last joint with n. The base of the robot will be numerated as 0.

2.- Locate and place the axis of each joint. If the joint is rotative, the axis will be its own turn axis. If its linear, the axis will be the axis with the direction in which we do the translation.

3.- We set the z-axis, for i from 0 to n-1, we place $z_i$ in the joint i+1.

4.-For the coordinate system $\{S_0\}$, we´ll put the origin at any point in $z_0$. Then, we place $x_0$ and $y_0$ using the dextrogyre system.

5.- For the rest of coordinate systems, in order to place $\{S_i\}$:

-If $z_{i-1}$ and $z_i$ don´t intersect, $\{S_i\}$ will be placed in the intersection of $z_i$ with the normal common line to $z_{i-1}$ and $z_i$.

-If $z_{i-1}$ and $z_i$ intersect, $\{S_i\}$ will be placed in the intersection point.

-If $z_{i-1}$ and $z_i$ are in parallel, $\{S_i\}$ will be placed in joint i

6.- For the rest of x-axis:

-If $z_{i-1}$ and $z_i$ are not in parallel, x-axis will be placed in the direction of the normal common line to $z_{i-1}$ and $z_i$, either looking from $z_{i-1}$ to $z_i$ or vice versa.

$$x_i = \pm(z_{i-1} \times z_i)$$

-If $z_{i-1}$ and $z_i$ are in parallel, x-axis will be placed in the plane that defines $z_{i-1}$ and $z_i$ and will be perpendicular to both.

7.- For the rest of y-axis, they will be place using the dextrogyre system.

8.- $\{S_n\}$:

- x-axis will be perpendicular to $z_{n-1}$ and will have to intersect it

- z-axis will be in the same direction of $z_{n-1}$, and looking outside the robot

- y-axis will complete the system using the dextrogyre system.

Once that every coordinate system is defined in the robot, we will be able to get a table with the parameters for each joint. That parameter table looks like Table 1:

Table 1. Denavit-Hartenberg parameters

| i-joint | $d_i$ | $\Theta_i$ | $\alpha_i$ | $a_i$ |
|---|---|---|---|---|
| 1 | $d_1$ | $\Theta_1$ | $\alpha_1$ | $a_1$ |
| … | … | … | … | … |
| n | $d_n$ | $\Theta_n$ | $\alpha_n$ | $a_2$ |

Where:

- $\Theta_i$ is the angle that we must turn $x_{i-1}$-axis around $z_{i-1}$ so $x_{i-1}$ and $x_i$ are in parallel.

- $d_i$ is the distance in $z_{i-1}$ axis in order to move $\{S_{i-1}\}$ so $x_{i-1}$ and $x_i$ are the same.

- $a_i$ is the distance in $x_{i-1}$ axis that we move $\{S_{i-1}\}$ so its origin fits with $\{S_i\}$

- $\alpha_i$ is the angle that we must turn $x_i$-axis so $\{S_{i-1}\}$ entirely fits with $\{S_i\}$

The resulting HTM will be the post production of all the small HTM between joints, which general composition is:

$$^{i-1}A_i = \begin{bmatrix} \cos(\theta_i) & -\cos(\alpha_i)\sin(\theta_i) & \sin(\alpha_i)\sin(\theta_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\alpha_i)\cos(\theta_i) & -\sin(\alpha_i)\cos(\theta_i) & a_i sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$^0A_n = \prod_{i=1}^{n} {}^{i-1}A_i \,; \boxed{\overrightarrow{x_0} = {}^0A_n \cdot \overrightarrow{x_n}}$$

There´s an alternative version of this method, proposed by John J Craig, which changes the steps, but makes the same method. Both are valid for every system.

Since all the positions are only composed by continuous functions, there will be only one solution for each group of angles, and the result will be direct.

Let´s see a practical example of a SCARA robot.

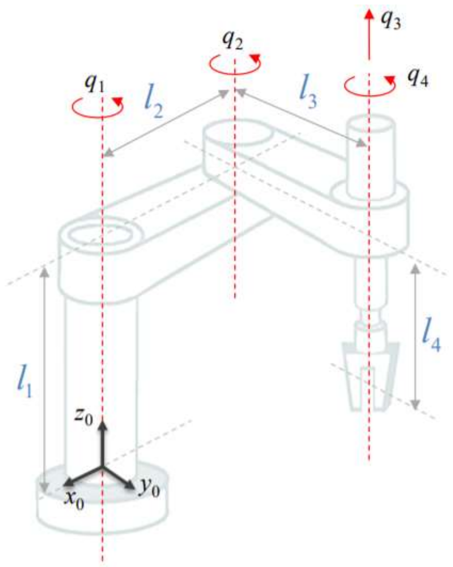


Figure 18. SCARA robot

The idea is to obtain the general HTM of Figure 18 using the D-H parameters. To get that, we´ll follow the steps given.

First, we put all the coordinate systems, knowing that the rotative axis are in parallel, so all origins must be in all joints. The z-axis, will be pointing to each rotative axis, and the x-axis will be perpendicular to the actual z-axis and the one before. The y-axis will complete each frame.

However, as we saw before, the TCP, which is the last frame, will be in the opposite direction, since the z-axis will have to be pointing outside the robot.

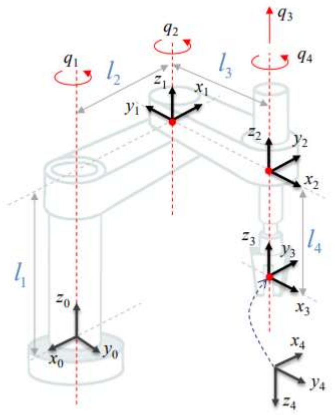


Figure 19. Position of the frames

If we look closely to Figure 19, we can see that the lengths given ($l_i$) will define the parameters of $d_i$ and $a_i$, and the angles of the joints ($q_i$) will do the same with $\Theta_i$ and $\alpha_i$, with the exception of $q_3$, which is linear

Filling the spaces in the parameters table, we finally get Table 2:

Table 2 Denavit-Hartenberg parameters in a SCARA robot

| i-joint | $d_i$ | $\Theta_i$ | $\alpha_i$ | $a_i$ |
|---|---|---|---|---|
| 1 | $l_1$ | $180+q_1$ | 0 | $l_2$ |
| 2 | 0 | $-90+q_2$ | 0 | $l_3$ |
| 3 | $-l_4+q_3$ | 0 | 0 | 0 |
| 4 | 0 | $90+q_4$ | 180 | 0 |

With their following local HTMs:

$$ {}^{0}A_1 = \begin{bmatrix} -\cos(q_1) & \sin(q_1) & 0 & -l_2\cos(q_2) \\ -\sin(q_1) & -\cos(\mathrm{q}_1) & 0 & -l_2\sin(q_2) \\ 0 \quad 0 & & 1 & l_1 \\ 0 \quad 0 & & 0 & 1 \end{bmatrix} $$

$$ {}^{1}A_2 = \begin{bmatrix} \sin(q_2) & \cos(q_2) & 0 & -l_3 \sin(q_2) \\ -\cos(q_2) & -\sin(q_2) & 0 & -l_3 \cos(q_2) \\ 0 \quad 0 & & 1 & 0 \\ 0 \quad 0 & & 0 & 1 \end{bmatrix} $$

$$ {}^{2}A_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & q_3 - l_4 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{3}A_4 = \begin{bmatrix} -\sin(q_4) & \cos(q_4) & 0 & 0 \\ \cos(q_4) & \sin(q_4) & 0 & 0 \\ 0 \quad 0 & & -1 & 0 \\ 0 \quad 0 & & 0 & 1 \end{bmatrix} $$

The global HTM is:

$$ {}^{0}A_4 = \begin{bmatrix} -c_{124} & -s_{124} & 0 & -l_3 s_{12} - l_2 c_1 \\ -s_{124} & c_{124} & 0 & l_3 c_{12} - l_2 s_1 \\ 0 \quad 0 & & -1 & l_1 - l_4 + q_3 \\ 0 \quad 0 & & 0 & 1 \end{bmatrix} $$

Where $c_{ijk}$= cos(i) cos (j) cos(k) and $s_{ijk}$= sin(i) sin(j) sin(k)

Homogenic Transformation Matrix development method

The HTM method uses the direct kinematic by the Denavit-Hartenberg parameters to find 12 non-linear equations with as many variables as joints the system has. For example, from the last exercise, we can set the direct kinematics like this:

$$ {}^{0}A_4 = {}^{0}A_1 \cdot {}^{1}A_2 \cdot {}^{2}A_3 \cdot {}^{3}A_4; $$

However, we can set other matrixes according to the properties of the matrix operations:

$$ {}^{0}A_1{}^{-1} \cdot {}^{0}A_4 = {}^{1}A_2 \cdot {}^{2}A_3 \cdot {}^{3}A_4 $$

$$ {}^{1}A_2{}^{-1} \cdot {}^{0}A_1{}^{-1} \cdot {}^{0}A_4 = {}^{2}A_3 \cdot {}^{3}A_4 $$

And it could continue, getting new easier and equivalent equations to get the results. As it happens with the geometrical method, this won´t have an only answer. In fact, there will be as many possible answers as DOF has the robot[3]. For example, the robot in Figure 20, since it has 6 DOF, it can have 6 possible solutions for the same problem.

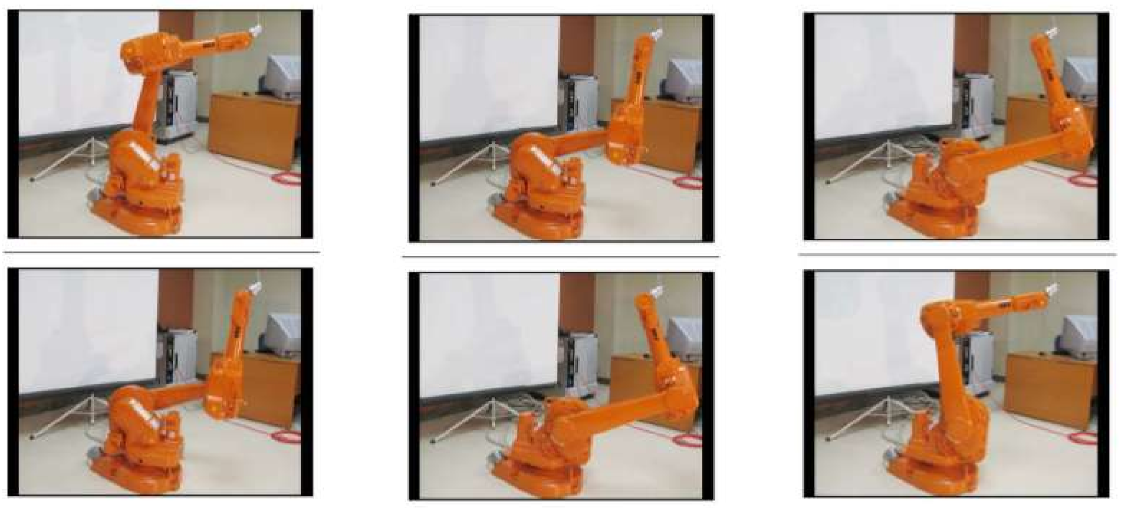
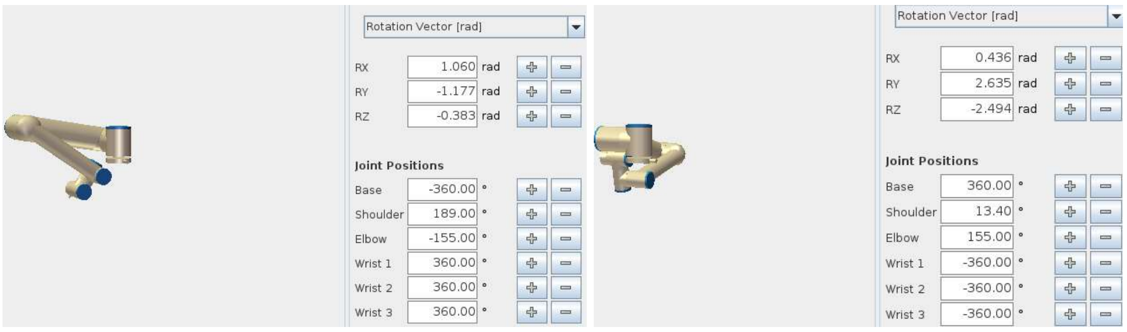


Figure 20. Possible configurations for an IRB120 to reach a point

In order to solve this, there are many robot languages that have some special variable, called confdata which, using an integer number, can set the quadrant where the joints will have a restrain to move, forcing the joints to have one specific result of angles. Other programs, calculate the distance between the point to reach and the movements of the joints to get it, and try to choose the most comfortable and fastest way to get it.

Is easy to realize that, even the different methods are quite useful to solve in different situations the problem given, computer programs can´t use any method that requires intuition, like the direct and the indirect kinematics method. Factors like Figure 4, clearly shows a need in using fixed steps to be sure to complete properly the control loop and get to next state with better results in $q_i$, x, y and z. Due to that, even if the other methods were easier and faster to make, computers can only solve this type of problems using a combination of D-H, HTM and control loop of resolution.

Using the teach pendant to manipulate the control of the robot, in order to get the workspace, we can check the maximum and minimum of the angles that the joints so it doesn´t hit anything that isn´t moving:


Rotation Vector [rad]

RX 1.060 rad
RY -1.177 rad
RZ -0.383 rad

Joint Positions

Base -360.00 °
Shoulder 189.00 °
Elbow -155.00 °
Wrist 1 360.00 °
Wrist 2 360.00 °
Wrist 3 360.00 °

Rotation Vector [rad]

RX 0.436 rad
RY 2.635 rad
RZ -2.494 rad

Joint Positions

Base 360.00 °
Shoulder 13.40 °
Elbow 155.00 °
Wrist 1 -360.00 °
Wrist 2 -360.00 °
Wrist 3 -360.00 °

Figure 21. Comparison between minimum and maximum angles

Figure 21 shows us the robot maximum workspace limits that the robot has including the base support before it gets hit with any part.

From Table 3 we can see the differences between the robot with base support and without base support:

Table 3 Comparision of angle ranges with and without constraints

| | Angle without base (º) | | Angle with base (º) | |
|---|---|---|---|---|
| Joint | minimum | maximum | minimum | Maximum |
| 1 | -360 | 360 | -360 | 360 |
| 2 | 360 | 360 | 189 | 13.40 |
| 3 | -360 | 360 | -155 | 155 |
| 4 | 360 | -360 | 360 | -360 |
| 5 | 360 | -360 | 360 | -360 |
| 6 | 360 | -360 | 360 | -360 |

The values of some of the joints don´t change at all, since their movement (like joint 5) doesn´t make a great change in the state of robots, or are made not to change position but the orientation (like the wrist), but the joints marked in orange (Joint 2 and Joint 3), since are the joints that have the parts that actually can hit the support, those are the ones that will have less range. We can think, in first place, that the table and the prop are also obstacles that can be hit by the robot. However, since the base support has wheels to move the robot, the restrictions will change deppending on the distance between the robot and the table. Even if we let the robot stay still, instead of recalculating the maximum joint angles, we can assure the movements from the robot that can go from the origin point (with the arm in vertical pointing upwards) to the piece that must be taken and viceversa, since the table is completely flat and is always under the piece.

Executing Basic level task, in the laboratory, shows us the movements of the robot making the operations given withou any problem of braking in speed or hitting hardly anythi ng. Besides, the piece taken doesn´t receive any damage repeating the task in loop. However, the structure of the program written in that language implies waypoints as constants (at least the initial ones), so is not possible to move anything at all. The program is prepared to have a fixed environment, with the board where we place the piece screwed to the table so it can´t be moved during the simulation. Due to that, everytime that the program was executed, the waypoints were modified everytime according to the actual position. If the board didn´t slip, everything could be easier. Other problem that we can find is that, since the holes are too tiny to place both piece and claws, the claws must grasp only half of the piece, having the problem that, since the piece is a cilinder, it can easily rotate in the axis perpendicular to the claws.

The last problem that we had executing it is that the robot itself had some problems reaching certain places, halting the execution due to the “impossible” solution that the robot had getting numbers as angles for the joints using inverse kinematics.

As long as the props didn´t move, and assigning waypoints to the robot that won´t give any problem calculating them, the robot is able to execute the program in a loop permanently (initial and final positions are the same during execution)

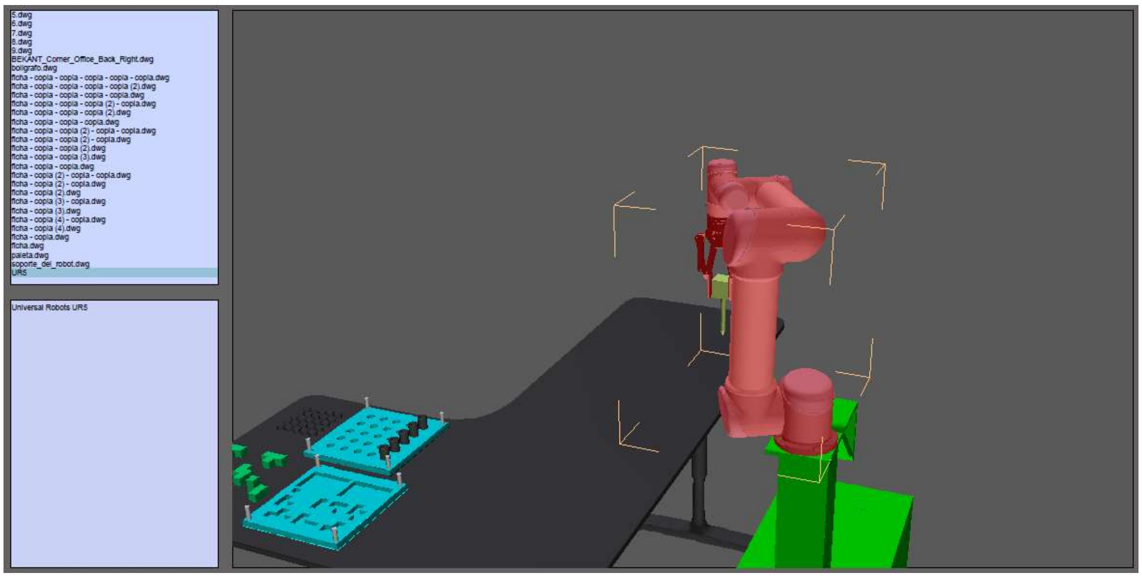
The first problem won´t appear in the following tasks, since the physics of the objects consider that the friction between objects are high enough so they won´t slip. However, the other problem, since is a problem that refers to a mathematical solution using loop control, it will appear too in the simulations and will make us to be careful at the moment that we choose a proper waypoint.

Next part to execute is the intermediate level task.

Figure 19 is a preview of the animation, which is linked to the pdf with the whole animation. Inside, you´ll see the whole animation of the robot making the task as if you started inside the VC program. There´s a small menu to see the process in a way more comfortable.

First, we can see both lists on the left.  The first one, on top, describes all the objects that have been imported, either from the catalogue of the program or externally, in alphabetical order. The second list, under the first one, in the beginning is always empty. This list shows the subcomponents that every part of the list has. If you choose any file from the first file, and it has a small description in its properties, it will be shown in the second list. For example, selecting "UR5", which is the last object in the list, it will appear a small description saying "Universal Robots UR5".

Besides, once that you click any object in the list, the 3d view will show you the animation seen from a camera focusing on the object selected, from the actual point (the selected object changes the color itself to red, like in Figure 22). It only changes the direction of the camera.



```
5.dwg
6.dwg
7.dwg
8.dwg
9.dwg
BEKANT_Corner_Office_Back_Right.dwg
boligrafo.dwg
ficha - copia - copia - copia - copia - copia.dwg
ficha - copia - copia - copia - copia (2).dwg
ficha - copia - copia - copia - copia.dwg
ficha - copia - copia - copia (2) - copia.dwg
ficha - copia - copia - copia (2).dwg
ficha - copia - copia - copia.dwg
ficha - copia - copia (2) - copia - copia.dwg
ficha - copia - copia (2) - copia.dwg
ficha - copia - copia (2).dwg
ficha - copia - copia (3).dwg
ficha - copia - copia.dwg
ficha - copia (2) - copia - copia.dwg
ficha - copia (2) - copia.dwg
ficha - copia (2).dwg
ficha - copia (3) - copia.dwg
ficha - copia (3).dwg
ficha - copia (4) - copia.dwg
ficha - copia (4).dwg
ficha - copia.dwg
ficha.dwg
paleta.dwg
soporte_del_robot.dwg
UR5
```

```
Universal Robots UR5
```

Figure 22. Object selection from the list in the pdf

On top of the pdf, we can see, on the right of the VC logo, the title of the task and several buttons that modifies the simulation in different aspects are placed forming a row, as shown in Figure 23.

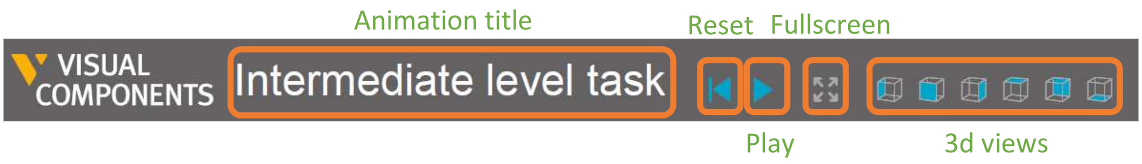


Figure 23. Top menu

Depending on the button, the action is different to be executed:

1. Reset: Halts the process and restarts it from the initial positions when the simulation was recorded. Is different from the reset button inside the program, because that resets the whole animation from the beginning of the process.
2. Play: Executes the process from the point it was stopped.
3. Fullscreen: Continues the animation modifying the aspect ratio to full screen. Depending on the processor, the simulation can work worse lowing fps during the video if this button is pressed.
4. 3d views: Sets the animation camera from different positions, all orthogonal. In order, from left to right, those views are: left, front, right, top, back and bottom.

Last part of the pdf is the 3d view: is the window with the 3d world where things are modified if options are selecting according to their descriptions.

Last part, the advanced level task, is executed in the same way. Is recommended to see the original file to get the whole animation without errors. The big visual issue with this is that in the pdf file most of the cubes that should be processed don´t appear. That, is an issue due to the recording, not to the task, so the fact that there wasn´t a possible solution with the computer that was recording doesn´t imply that the task is not well done. Despite that, the pdf is still available to see so there´s no need to open the program (is faster to see from the pdf and more comfortable, because with the program before the first simulation the script of the UR5 must be recompiled).

One difference between this task and the previous ones, is that this one isn´t executed making a loop, because the initial situation (no pallets and no cubes) only happens once (the creation of new pallets and cubes are faster than their disappearing through the conveyor. Due to that, and assuming that this task is made to make a continuous and "infinite" process, the recording is longer than it should be seen (there are about 4 periods of filling pallets, in order to see that there´s no problem if there are pallets or cubes waiting in the beginning).

Another issue which is from the animation (Figure 24), is that the gripper, when it takes a cube, its claws don´t collide with the surfaces and go through them. This, is because the cubes haven´t been modelled using the CAD of the program, but created entirely using Python program. In order to produce that collision, first we must be sure that the gripper of the robot must have enabled the collision with objects (ours is already enabled), and second, inside the Python coding we should add a command including physics properties of the components, enabling volumes as physical mass and not only as 3d volumes in the space. If those were changed, the simulation would look fancier, but it won´t change a thing in the execution.

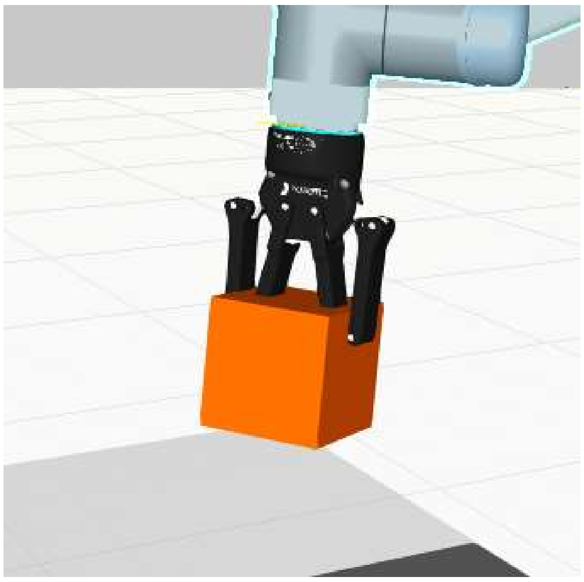


Figure 24. Here we can clearly see that the claws are inside the cube.

Even with all those visual issues, in the animation its noticeable that the process made is sequential, with steps controlled by the UR5, as it would be in real life.

It´s curious in this simulation, because the program detects that the support of the UR5, the UR5 and the cubes (which are called "Name" once that they appear but in the script are removed) appear in the simulation. However, the objects with no animation and the ones that come from the catalogue, like the pallet, the conveyors and the feeder templates don´t. Is something predefined from the program: everything that is labeled as "inanimate" from the catalogue, it´s useless to put them in the process list during recording, so those are hidden.

Although this project was done from scratch, this is not the first one that has been developing the hypothesis proposed. Researchers like Pablo Horno Perez[5] have been trying to write its thesis focusing in the robot properties and how does it work making several different tasks using the 2F-85 and wrist camera devices. Victor Hugo[4], on the other hand, oriented its project in using ROS (Robot Operating System) so mechanical operations can be made depending on the robot and working environment. In the end, both descriptions and results become reasonably similar between ours when methods and materials are described (the procedure to initiate a robot and to make it work in a factory or any kind of working environment are the same with every robot).

# CONCLUSION

In the end, the hypothesis proposed was a statement that could be reasonably demonstrated if we consider that every possible process is the result of the set of different actions that imply atomically movements, decisions and calculus, which are defined by the basic actions that the processor of the UR5 can do, which is included in the definition of the microprocessors and microcontrollers in digital electronics theory, easily demonstrated, using the Karnaugh maps and Boolean laws. However, those disposals, even if they prove that the hypothesis is true, they don´t show the potential and the final results that the robot is able to do. As engineers, is the most important thing to consider if we want to improve its technology or if we decide that the robot is up-to-date enough to keep it using in industry, and not if we can take the development of the methods that prove it using them as corollaries and extensions of the basic instructions.

It´s true that the whole project doesn´t depend entirely on the robot, because if the computer is not good enough, the time that is calculated and the simulations done could show different results, including bad states where the robot can´t process certain instructions that are easily done in real life. In fact, even if the computer is good enough, if the software is not the ideal to execute the programs made, there could be several problems including issues that aren´t related to the simulation (pathlib library is a library that only works well with Python software with versions over 2.7, and isn´t included until 3.2 version). However, assuming that the user has the ideal hardware and software (the material used in this project by his researcher is good enough to set properly the constraints in the environments of the simulations), there will be always good results with the scripts written.

From the basic level, which is just a group of sequential instructions including only grasping and movements instructions, until the advanced level, that is included besides the ones said previously, an entire group of conditional statements, various threads executed at the same time, and exports and imports of files and their interactions with the processors, we have been able to set different real processes which are not a problem to the robot to be done without real-time execution problems, as long as the environment that surrounds the robot is well prepared (referring this as an environment that is close enough in distance to the robot so the positions and waypoints where the robot must go through are reachable by it).

All in all, we can definitely say that the hypothesis proposed is demonstrated, and the results given shows us that the robot UR5 is capable to make industrial processes that imply a robotic arm, with the advantage that is safe to human workers that must collaborate in a short range.

# BIBLIOGRAPHY

[1] Asimov, I. Runaround. Astounding stories of Super-Science. 1942: 88. Pages 11-12.

[2] Engelberger J., Historical Perspective and Role in Automation in Shimon Y. NOF, Handbook of industrial robotics, Ed. John Wiley & Sons Inc. 1999. Pages 1-10

[3] Fraile Marinero J.C., Introducción a la robótica OK1 , Fraile Marinero J.C., Introducción a la robótica OK2, Fraile Marinero J.C., Cinemática directa y Parámetros DH OK8, Fraile Marinero J.C., Cinemática inversa OK9 in Fraile Marinero J.C., Herreros López A., García González J., Teoría de la asignatura Sistemas Robotizados, 2019, pages 1-1058

[4] Gomez Tejada, V. Operación de remachado mediante robot manipulador. 2015

[5] Horno Perez, P. Robot UR5 guiado por visión artificial. 2018

[6] Polyscope manual, seen in https://www.usna.edu/Users/weaprcon/kutzer/_files/documents/Software%20Manual,%20UR.pdf

[7] Python documentation, seen in https://docs.python.org/dev/

[8] Universal Robots, Our History, seen in https://www.universal-robots.com/about-universal-robots/our-history/

[9] UR5, Control box and Teach Pendant datasheet, seen in: https://www.universal-robots.com/media/50588/ur5_en.pdf

[10] UR5 installation manual, seen in: https://www.usna.edu/Users/weaprcon/kutzer/_files/documents/User%20Manual,%20UR5.pdf

[11] Sánchez Martín, F.M. Millán Rodríguez, F. Salvador Bayarri, J. Palou Redorta, J. Rodríguez Escovar, F. Esquena Fernández, S. Villavicencio Mavrich, H. Historia de la robótica: de Arquitas de Tarento al robot Da Vinci (Parte I). ACTAS UROLÓGICAS ESPAÑOLAS. 2007: 31. Pages 69-76

[12] Visual Components Guide, seen in https://www.visualcomponents.com/wp-content/uploads/2018/10/Experience_Guide-1.pdf