TÉCNICO LISBOA

Robotics

Electrical and Computer Engineering

Biomedical Engineering

Lab1 Report

Students:

Alexandre Reis (100123) alexandre.leite.reis@tecnico.ulisboa.pt
Sebastián Márquez (108665) sebastian.marquez@tecnico.ulisboa.pt
João Dias (100202) joao.v.dias@tecnico.ulisboa.pt
Carlos Marqués (108419) carlos.marques.tatay@tecnico.ulisboa.pt

Group - 18

2023/2024 – First Semester, Q2

Contents

1 Contribution of each member

1.1 Code contributions

The programming work was divided between the group members, in the following way:

- **Alexandre Reis (100123)**: Joystick interface with the computer ("joystick_functions.py", "main_pc1.py" and "robot_classes_manual.py"); Object oriented programming, representing the robots as objects (classes) in order to use those to define the state of the robots and communicate with them ("robot_classes_manual.py"); Serial communication between the python script and the robots ("robot_classes_manual.py"); Threads to be able to process the joystick commands, and communicate with serial to both robots, all at the same time ("main_pc1.py"); Method to control the robots' position and trajectory ("robot_classes_manual.py"); Scalpel's position prediction ("robot_classes_manual.py").
- **Carlos Marques (108419)**: Collision detection between the camera and the Scalpel ("colision_detection.py"); Scalpel's position prediction ("robot_classes_manual.py"); Code to find the rotation and translation matrix between the base of the 2 robots using the camera calibration (was not used in the final version of the code) ("calibration.py").
- **João Dias (100202)**: Real time communication (using Sockets) between the 2 robots ("communication_client.py", "communication_server.py", "main_pc2.py","main_pc1.py"); Foward Kinematics ("foward_kinematics.py"); Method to control the robots' position and trajectory ("robot_classes_manual.py"); Threads that parallelize the GUI with a communication server that receives the displayed information ("main_pc2.py")
- **Sebastián Márquez (108665)**: Graphical Interface ("graph_interface.py"); Detection of the Scalpel using the camera feed and subsequent distance calculation between the 2 objects (was not used in the final version of the code) ("ObjectTracking.py"); Depth estimation between camera and scalpel using monocular depth estimation (was not used in the final version of the code) ("monocularDepth.py").

1.2 Report contributions:

- **Alexandre Reis (100123)** Wrote Sections: 3. "Scorbot-ER VII Manipulator arm - Movement"; 6. "Controller inputs", part of 4."The program's sequence (modes)", 9. "Conclusions"
- **Carlos Marqués (108419)** Wrote Sections 8.1 "Camera calibration"; 8.3 "Direct distance measurement"
- **João Dias (100202)**: Wrote Sections 2. "Objectives and requirements", 5. "Arquitecture", Section 8.2 "Direct Kinematics"
- **Sebastián Márquez (108665)**: Wrote sections: 7. "Graphical Interface", Part of 4. "The program's sequence (modes)"

2 Objectives and requirements

The **requirements** defined for this project are:

- Receive inputs from user with a Joystick controller.
- Teleoperating 2 robots in order to make cut through a gelatin block. (One of the manipulators holds a scalpel and the other a camera)
- The user can only operate the robots using the camera's video feed.

In order to fulfill these requirements, some **objectives** were defined:

- The robots must be as easy to control by the user as possible. Their movement should also be predictable so the user can control them with minimal feedback (only camera feed).
- Features should be implemented to give confidence to the user while moving the robots. One of them should be preventing the user from colliding the 2 robots.
- The movement of the robots should be responsive (in real-time) to inputs given by the user.
- In order to implement the system in a teleoperating surgical procedure, the doctor and GUI should be operating in a different computer from the computer that controls the robots. These computers need to be able to be connected wirelessly and communicate without visible latency.
- More feedback (besides camera's video feed) needs to be given to the user about the robot's position and state.

3 Scorbot-ER VII Manipulator arm - Movement

The robots that are used in this project are the Scorbot-ER VII Manipulator arm. These have 5 Degree's of Freedom, operate with ACL commands (Advanced Control Language) and are controlled via serial communication. This implies that the python script that is running on the student's computer sends encoded ACL commands (in the form of strings) to the robot's buffer. The robot also responds to the computer with serial communication to the computer's serial port's buffer.

The robot has 2 main modes of operation:

- **Direct mode** - This executes commands given to the robot, immediately
- **Edit mode** - Is used to create ACL programs that make the robot run by itself

For this application, as we want to control the robot in real-time, based on input from the user, **Direct Mode** is the chosen option.

Inside Direct mode, the robot can be moved in 2 ways: Defining one or more positional vectors and later moving through them; or using **Manual Mode**, which moves the robot in real-time in one direction (defined be real-time commands).

3.1 Tests: Manual Mode vs Direct Move mode

Moving with direct mode and without Manual mode has plenty of advantages. Some of them are:

- Movement occurs in more than one joints/ direction at the same time
- More complex movements are achieved
- Robot can communicate with computer (when asked about it's position and other parameters).

Several test were done with this mode. These boiled down to the user pressing a button on the Joystick, the program defining a position that is some (small) distance from the original position and finally the robot moving to it.

The problem was that a lot of commands needed to be sent via serial communication to the robot. As these need to be sent in time intervals (so the robot can process all of them) the program was not at all responsive.

The wait times between sending commands were adjusted to minimize the response time of the program, but the best case achieved was not responsive enough and user's did not find it easy to control the robot at all. (The best time achieved from the moment an input is given to the moment the robot moves was between 0.5s and 1s).

The group tested Manual mode. With this mode, the response of the robot was immediate (in real time with the user input). As the responsiveness of the system is an objective that was defined at the beginning of the project, the chosen mode is **Manual Mode**.

3.2 Manual mode Tests: Joints vs XYZ

Manual mode has 2 ways of operating. Either the user can change the **joints positions** (one at a time), or the user **(in "Manual XYZ Mode")** can change a spacial coordinate (X,Y,Z,Pitch or roll) (one at a time).

To decide what was the best mode to use, some tests were done with a user unfamiliar with the system.

The test consist on: Starting from a top position; Reaching a position, with the scalpel, near the table and finally moving horizontally to another position in the table. All these movements were timed.

Disclaimer: The Manual Joints mode was using speed=16 and the Manual XYZ mode was using speed=10. The reason for this is that joints mode can move smoothly and without errors with higher speed values.

	Time to reach table	Time to move to point 2 (horizontal plane)
All Joints manual mode	23s	49s
All XYZ manual mode	40s	35s
Joints manual mode in 1st part; XYZ manual mode in 2nd part	30	40s

Table 1: Test results - Manual mode Joints vs XYZ

The user that performed these tests also stated that:

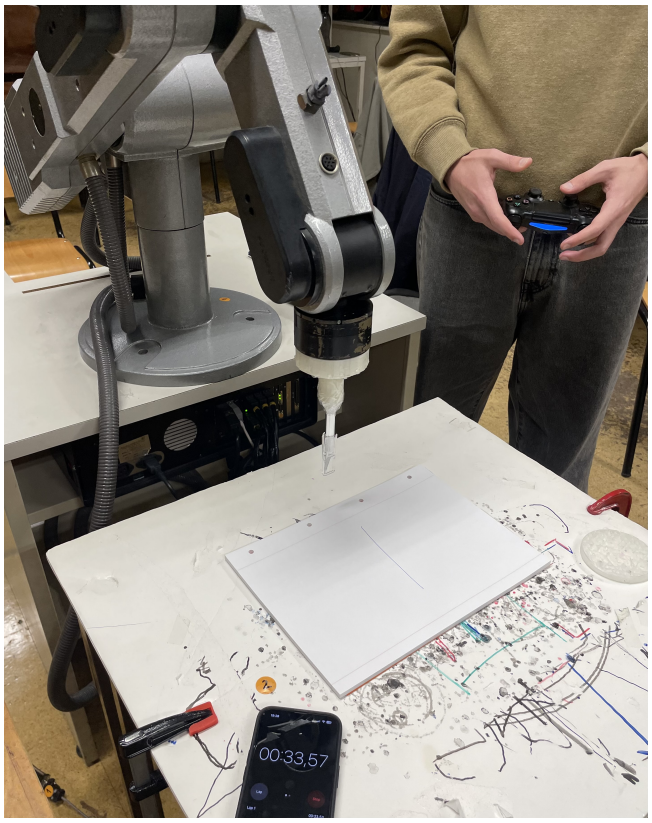

Figure 1: Time Test - Manual mode Joints vs XYZ

- The only intuitive way to move the robot, using the GUI to view it (as intended), is with XYZ mode.
- To move the robot with precision, XYZ mode is the best choice
- To move the robot quickly and further, Manual Joints mode is the best

The results suggest that **"Manual joints mode" are not very easy to control**, mainly when looking at the system through the camera feed. This means that, most of the time, the user should control the robot in Manual XYZ mode.

The only moment when Manual Joints mode is better at controlling the robot is to get it fast and continuously from any far away position to a position near the operation zone. To avoid needing to do this, the robot moves itself to a good position at the beginning of the program (figure 3). From this position, it is easy to control the scalpel's position in XYZ Manual mode.

Despite what was said before, in this program, the user also has the possibility to move the robot in Joints mode if needed. (for example if the operation zone changes position and the robot needs to rotate with the base joint).

3.3 Manual Mode Constraints

In order to know the current position of the robot's end-effector, in Direct Mode, the computer sends the command: "LISTPV POSITION \r". After this, the robot responds with a message containing Joint and Cartesian coordinate values for the current position of the

end-effector. A function "calculate_pos()" was implemented to do this and parse the output message, in order to get the joints and Cartesian coordinate values. This function was optimized to be as fast as possible.

The main problem with controlling the robot in Manual mode is that the LISTPV command cannot be executed inside this mode. This means that there is no easy way to know were the robot currently is.

This is very much a problem as one of the defined objectives was to not let the user collide the 2 robots. To implement this, we need to know where both robot are in real time.

A solution to this problem is to exit manual mode, calculate the position of the robot and, afterwards, enter Manual mode again. This takes some time (1,5s), and if done very frequently, it makes the program not responsive.

An alternative solution is to do this once in a while and, in the mean time, predict the position of the robot. This should make the program very responsive, most of the time.

In order to predict the position of the robot in real time, data was taken from the robot (relating the commands sent in manual mode and the distance traveled by the robot in a time interval). With this data, a linear regression was made. This linear regression predicts the end-effector's position with some error margin, which means that from time to time, the "calculate_pos()" function should be ran (so error does not accumulate).

In the final implementation of the system, when 7 seconds have passed since the last "LISTPV" command was done, the "calculate_pos()" function is ran. In order to not interrupt the user, the function waits for the user to stop inputting controls in the joystick, before executing.

3.4 Movement of the Camera robot

The camera-robot's movement does not need to be very responsive. Besides this, it's movement must be as easy to control as possible.

In order to not complicate the movement of this robot, not all 5 DOFs are given as controls to the user.

First of all, due to this robot's movement constraints, there is no need to move the roll of the camera (if it is in the correct initial orientation). This means that the initial position should have a good roll value and no more roll adjustments need to be done.

As we don't need this robot to be very responsive, it is possible to move it's position in normal Direct mode. This enables us to know where the robot's end-effector is with precision, at any time.

This movement can be done only in x and z coordinates, which are the most important in order to enable the user to get a good viewing angle.

Finally, the most important part of the movement of the camera is it's pitch angle. This is very powerful at enabling the user to look around.

In order to make the pitch angle motion as smooth as possible, it was decided to leave this robot in manual Joints mode for most of the time (to enable easy and quick pitch angle adjustments). And, when an end-effector position movement is needed, the robot will come out of manual mode and perform the movement.

This setup was tested with a user and he mostly used the pitch angle to move the camera. He sometimes also moved the end-effector's position, but as each movement was set-up to cover some centimeters, he did not complain about the latency that occurred.

4 The program's sequence (modes)

In this section, a walk-trough of the program actions and states will be given.

The program starts by homing the 2 robot's positions. This ensures that the encoders are reading correct values and that some predefined positions are in the right place.

At this point in time, the 2 computers connect (one being a server and the other being a client) and PC2 starts showing a GUI.

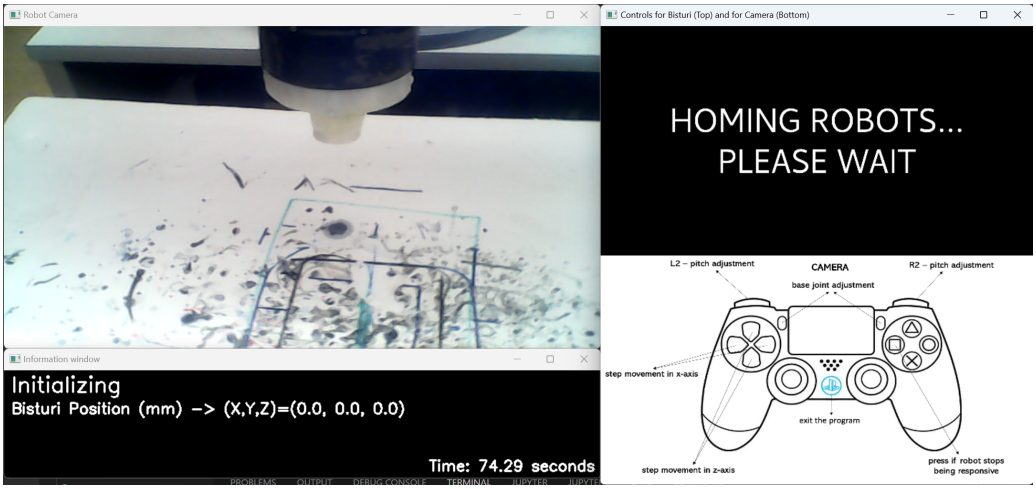

Figure 2: GUI in Homing state

After the homing process has been complete, the robots move to a predefined initial position. The position of the scalpel's robot was defined ensuring easy movement to the Gelatine block, in XYZ manual mode. The position of the camera was defined ensuring correct roll position and in order to get a good viewing angle while also being able to move in any direction with ease.

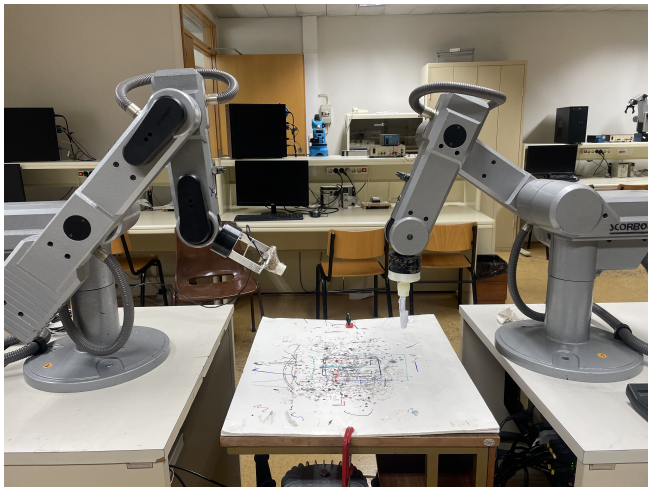

Figure 3: Initial Position of robots

4.1 Moving to the incision position

After the setup process has finished, the user can move the robots as he wishes. This can be done in Manual XYZ mode or in Manual Joints mode. The user can change between the two modes using the X button on the controller.

In order for the user to have an easy learning experience, the button layouts that control both robots are shown on screen.

The objective of this part of the program is for the user to move the robot's end-effector to the point where the cut will start.

At this point in time, the user can see the current coordinates of the Bisturi/Scalpel's position.

If the user wishes to, he can Zero the coordinates shown on screen (using the circle button) in one spot. This enables to see how much distance has been covered in each axis, from when he zeroed..

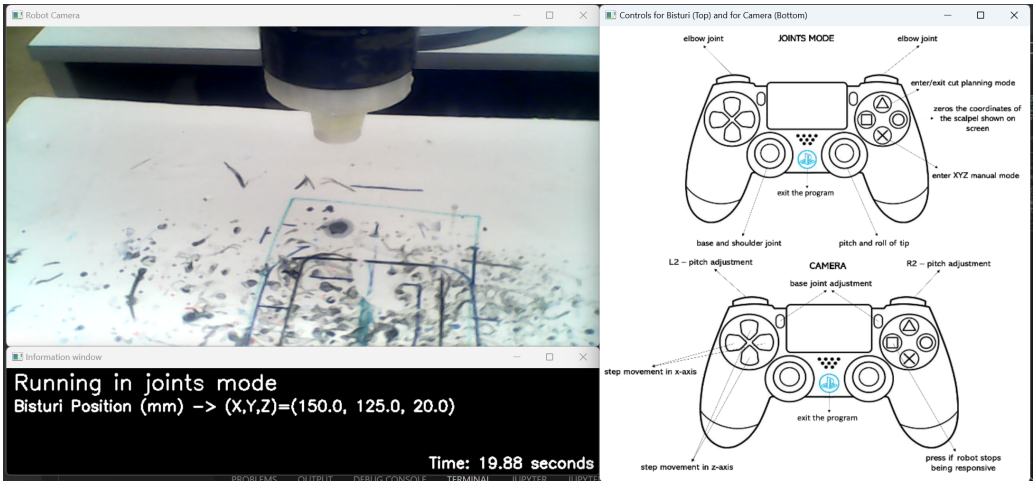

Figure 4: GUI in Manual Joints mode

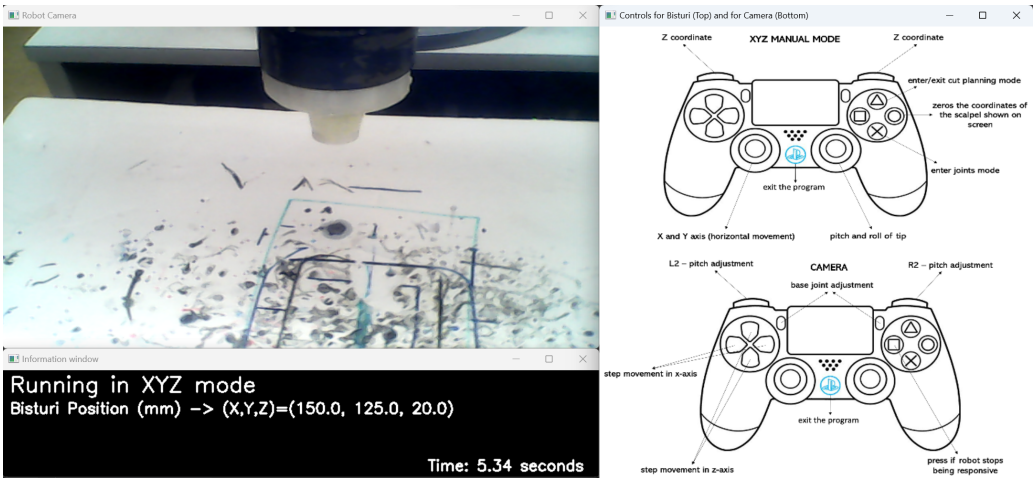

Figure 5: GUI in Manual XYZ mode

4.2 Cutting

To perform the cut on the gelatin block, the user has 2 options:

- Use the Manual XYZ mode to move the Scalpel and perform the cut manually
- Use the Cutting mode that is built-in to the program

If the user wants to cut using the second option, he needs to click the triangle button to enter prepare to cut mode. In this mode, the camera can move but the Scalpel cannot. This mode is used to define the 2 parameters of the cut (Depth of Cut and Length of cut).

Once again, there is a scheme that automatically appears on the GUI that explains how to operate this mode.

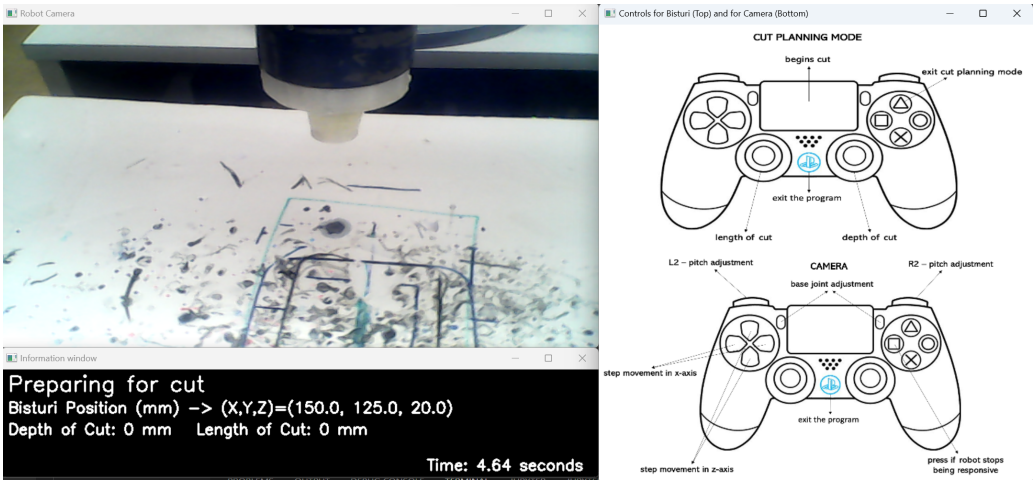

Figure 6: GUI in cut planning mode

Finally, when the user has selected the Depth of Cut and Length of cut values that he wants to use, he can click the Touchpad button on the controller, and the Scalpel will slowly trace a path. This path starts off by changing the pitch angle of the Scalpel, in order to make the incision with it's sharp edge. Then it perforates the Gelatin in an amount equal to the wanted depth. Then it changes the pitch angle of the Scalpel to be vertical (better for cutting horizontally). After this, it moves in the x axis (an amount equal to the wanted length of cut). Finally it moves up and leaves a margin between the blade and the Gelatin.

After this, the program returns to the normal movement in manual mode.

To end the program, the user has to press the Playstation button.

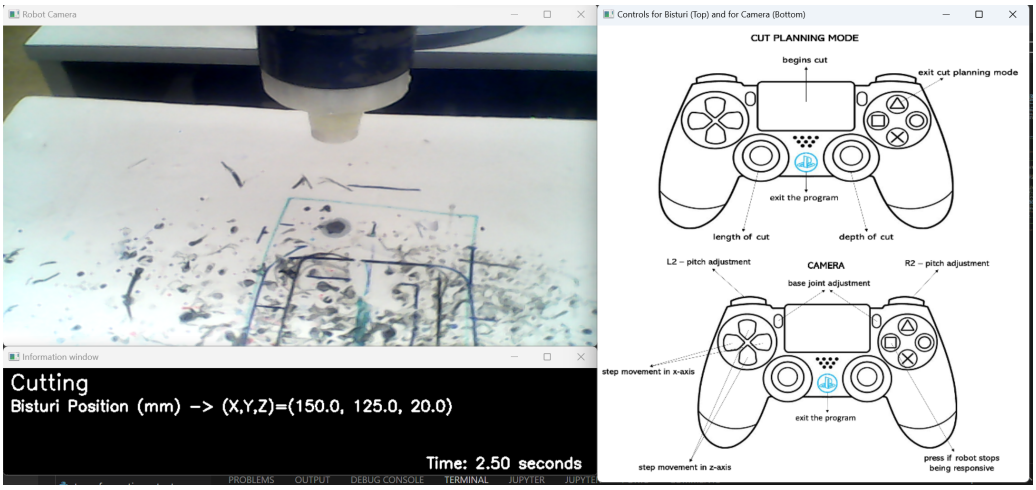

Figure 7: GUI while performing the cut

5 Architecture

The hardware architecture of this system was designed to accomplish all the defined objectives:

- As the movement of the robots must be responsive/in real time, both the joystick and the robots serial ports were connected to 1 computer (this is PC1). This ensured that this computer can be fast at processing user input and sending it to the robots.
- In order for the user to teleoperate in a different location than where the robots are at: the GUI must be shown in a different computer than PC1. For this, a wireless sockets communication was implemented between the 2 computers (this GUI computer is referred to as PC2). This communication between computer enabled the GUI to show a lot of information about the robots' state.
- As explained before, the Joystick controller is connected to the computer via Bluetooth, in order to enable more freedom for the user to move in the lab. This communication is done in 2 directions, as the computer also gives inputs to the controller, when it needs to vibrate.

In a real-world scenario, where the doctor had to perform surgery in a remote environment, the joystick would have to be connected to PC2 and it's input's should also be part of the sockets communication between computers. In this case that was no needed and it increased the delay in the system.

In that same real-world scenario, the video feed should also be sent to the remote computer via some videoconferencing software.

5.1 Communication between computers

As mentioned above, one of the main objectives of this work was to be able to have the GUI with relevant information related to the operation in a different computer, creating an environment where is possible to do an operation remotely.

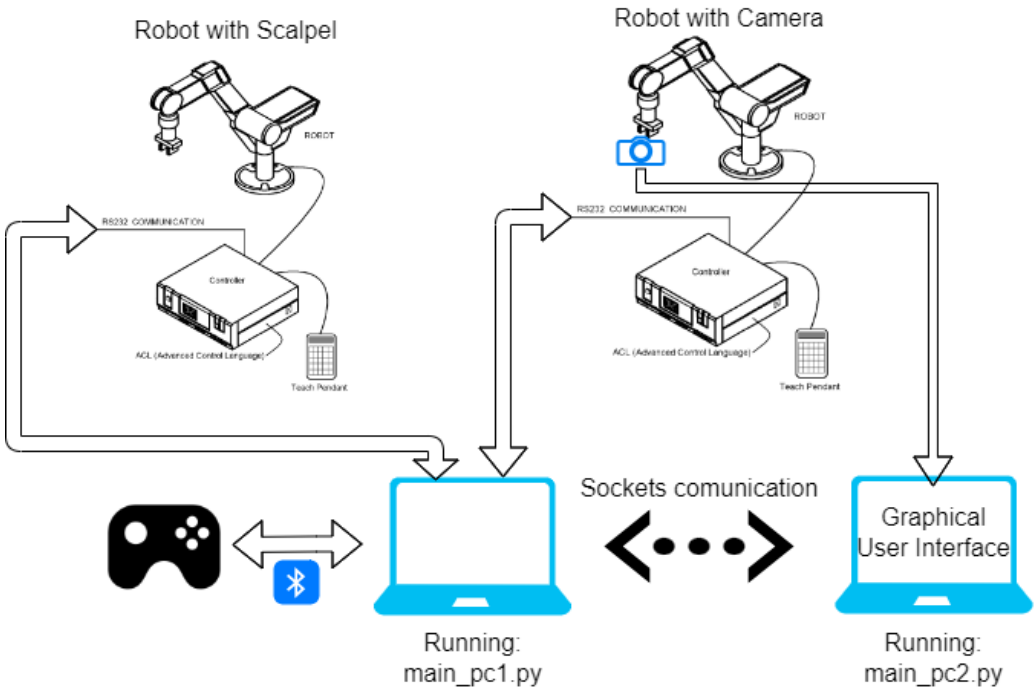

Figure 8: Hardware architecture of the system

To do so, it was needed to have a way to effectively transfer data between the two laptops. In the end, the built-in *Python* library was used, where the laptop where the graphical interface is shown acts as the server, while the laptop that is connected to the manipulators acts as a client. As long as both machines are connected to the same network, it is possible to exchange information between them.

It should also be mentioned that, in this case, the data flow is only in one way, from the client to the server, since with the current project's architecture is believed to be enough. Something that could be changed in the future, with the graphical interface laptop being able to do some type of computations, like the distance to a object, and then sending it to the other laptop, for example.

5.2 Threads

In each of the 2 computers, there are several processes running in parallel. For example, PC1 has to receive joystick inputs, process those inputs (predict the pose of the robot, check for collisions, etc.), send/receive serial commands to each of the 2 robots and communicate with sockets to the other computer.

In order for one of these steps to not delay the loop of another step, a thread was used for each one (taking care to share important information between Threads, via queues).

Threads are ideal for paralellizing serial communications to several serial ports. The reason for this is that serial communication tasks take time (to read and write to buffer + time.sleep() between operations), which gives time to other threads to run. Besides this, each of these tasks are done in loops, where the iterations per second are capped to a maximum of 40 (inducing wait time in them), which once again liberates time for other tasks to run. This essentially makes it look like all tasks are being done at the same time, when in practise, python only runs

one of them at any point in time.

This solution worked great and, the user cannot detect any delays that are caused by Threads.

An equal approach is done with the 2 tasks in PC2 (communicating with sockets and presenting the GUI).

5.3 Serial communication and Debug mode:

This project heavily relies in serial communication with the robot. That makes it hard to debug the project at home, when not connected via a serial port to the robots. The same applies to when there is no colleague with a computer nearby.

In order to debug the code in these situations, a variable was introduced to the "main_pc1.py" script. This variable is called *atHome*, it is False when testing in the lab and True when testing at home/outside the lab.

When this variable is True, no serial port is opened or closed and, instead of doing "serial.write" of a given command, the script just prints it to the terminal.

The same applies to the sockets communication thread on PC1: If $atHome = True$, instead of connecting to a server and sending data, the script just prints the same information to the terminal.

This turned out to be vital in the development of the program and it saved a lot of time debugging python code while connected to the robots.

6 Controller inputs

For this project, a PS4 controller was used instead of the one in the lab. There are several reasons that made the group choose that option and the main ones were:

- It has more buttons, which gives more freedom when creating new features and assigning buttons to them.
- The group could test with the controller at home, which enabled for a faster and easier debugging process.
- This controller has Bluetooth, which means that the computer can retrieve inputs from it without using a USB cable. This is very important in the lab when the user wants freedom to move.
- This controller can vibrate, which is another way of sending feedback to user about the state of the program. This can be done via Bluetooth and was implemented quite successfully.

Rule: The most easy-to-access buttons should be the ones that controll the most important parts of the robots' movement.

In order to discover which were the "most easy-to-access buttons", an analysis was made on the resting position of a human holding the controller. The group concluded that the human's thumbs rest on the analogical joysticks and his indexes rest on either L1/R1 or L2/R2. (If you don't know which buttons are L1, R1, L2 and R2, see Figure 9)

The most frequent actions performed by the user are:

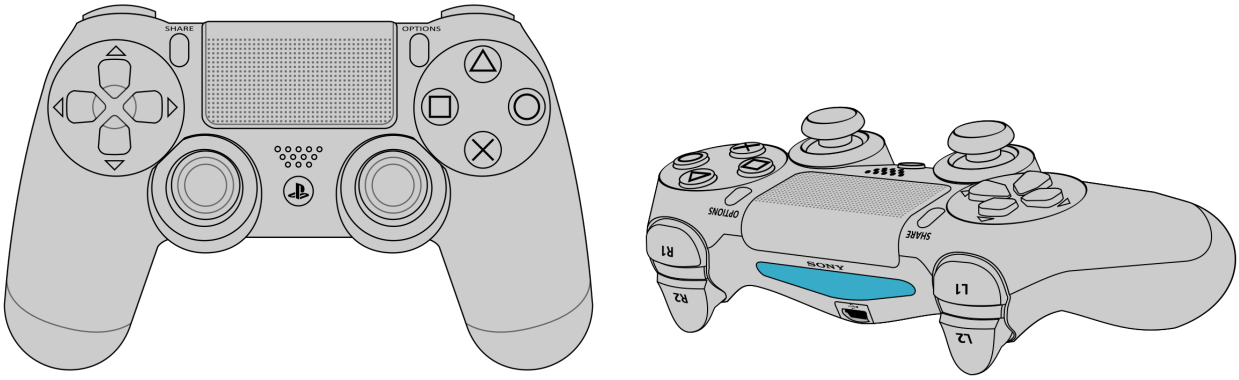

Figure 9: PS4 Dualshock 4 Controller

Figure 10: Resting position of a human holding the controller

- Move/controll all DOF of the Scalpel's robot
- Adjust the pitch angle of the camera

Because of these finding, the following were defined (In manual XYZ mode):

- **R2/L2** - Adjust pitch angle of camera
- **Right joystick** - Adjust pitch and roll angles of scalpel
- **Left Joystick** - Adjust position of scalpel in an horizontal plane (change x and y coordinates of scalpel)
- **R1/L1** - Adjust z coordinate of scalpel

The direction of movement (positive or negative) from each of these inputs from the controller was then adjusted using student that did not have experience using the system beforehand.

For "Manual Joints mode" control of the scalpel, a similar approach was used and the resulting scalpel controlling layout is:

- **Right joystick** - Adjust pitch and roll angles of scalpel
- **Left Joystick** - Adjust base joint and should joint of scalpel's robot
- **R1/L1** - Adjust elbow joints of scalpel's robot

Using the antitheses of the last thought-process, the **Playstation button** is the hardest button to access and there is no circumstance where a user miss-clicks it. For this reason, it was assigned to quit/finish the program (equivalent to closing all serial communication and stopping the running loops).

6.1 Controls for Camera Robot

The camera robot should start the program already in a favourable position (Figure 3). As explained in 3.4, not every DOF of the camera's robot needs to be controlled (only pitch angle, X and Z axis).

As explained before, the pitch angle is the most common of these adjustments and is assigned to **R2/L2** buttons (which are easy to reach).

The group's first idea was that each Z and X axis movement should be a step (2-5 cm) in that direction. After testing some approaches, the arrows were the best option, as the user involuntarily waits some time between arrow-button clicks which is exactly equivalent to the movement we desire from the camera robot.

We tested this layout with a user that was not familiar with the system. He helped to adjust the direction of each arrow. The main demand he had was to switch our original left arrow with the right arrow. The final product was:

- Up arrow increases the z coordinate of the camera by 50mm
- Bottom arrow decreases the z coordinate of the camera by 50mm
- Left arrow increases the x coordinate of the camera by 40mm
- Right arrow decreases the x coordinate of the camera by 40mm

The user also pointed out that it was fine to not have roll control (if initial position has good roll angle), but that he needed base-joint adjustment. The main reason for this is that he needs to follow the Scalpel when it is moving in the y coordinate (besides the fact that the initial position did not guarantee that the camera was aligned in the y axis with the scalpel).

This last suggestion was implemented. As the base joints adjustment mainly occurs at the beginning of the program or, at most, 1 - 2 times per operation, some harder to reach buttons were selected.

The selected buttons were the options and share buttons (figure 9) that are conveniently placed on the left and right of the controller. These make the camera robot base joint rotate to the right and to the left respectively.

6.2 Feature-operating buttons

The final buttons to assign were the right keypads (Triangle, Circle, Square and X). These were assigned to perform different features along the program (these features are explained inside the GUI, so that the user always knows what each one does). These features are:

- **X button** - changes manual move mode of scalpel's robot (between Joints mode and XYZ mode)
- **Circle button** - Zeros/Taras the current position of the scalpel (shown on the GUI). This is very useful when trying to measure distances while moving the scalpel.
- **Triangle** - Enter/exit "plan to cut mode", where the user can define the parameters of the cut and begin the path movement.
- **Square** - Button to press if camera robot stops being responsive (most of the time this is not needed)

To make sure that the robots do not perform the same action twice, some logic was implemented in order for a click of these buttons to perform a task if the state of the button was previously unclicked.

In order to give more feedback to the user about what mode he is in and what changed when pressing each of these buttons, controler vibrations were used.

Different low range and high range frequencies were used for each feature. This made the change in operating mode very tactile and improved one of the defined objectives on the begining of the report (giving more feedback to the user than just the camera's video feed).

7 Graphical Interface

The camera robot includes features that allow the operator to achieve the desired objective. The graphical interface is mainly based on a python dictionary which updates at real time, providing a friendly and active interaction between the robots and the user. This interface is divided into three windows: the video itself, a window containing images of the controller (with the function of each button) and a last one where crucial information is printed.

7.1 Controller Images for the Scalpel and Camera Robots

The first important feature of the graphical interface is a window located at the right side of the screen which includes a series of images that show the function of the buttons in the controller. In total, there are four images. The first three correspond to the controls of the scalpel robot (joints mode, XYZ normal mode and cut planning mode), while the fourth one corresponds to the controls of the camera robot. The window itself is divided into top and bottom sides. The top side contains the images of the scalpel's robot modes with their respective controls. A message saying "Homing Robots... Please Wait" appears on the same side when the program is being initialized. The bottom side contains the images of camera's robot controls. These images change in real time when the user switches between modes. The

goal of this feature is to help the user control the robots smoothly, owing to the fact that the controls may not be well known yet. See figures 11 and 12.

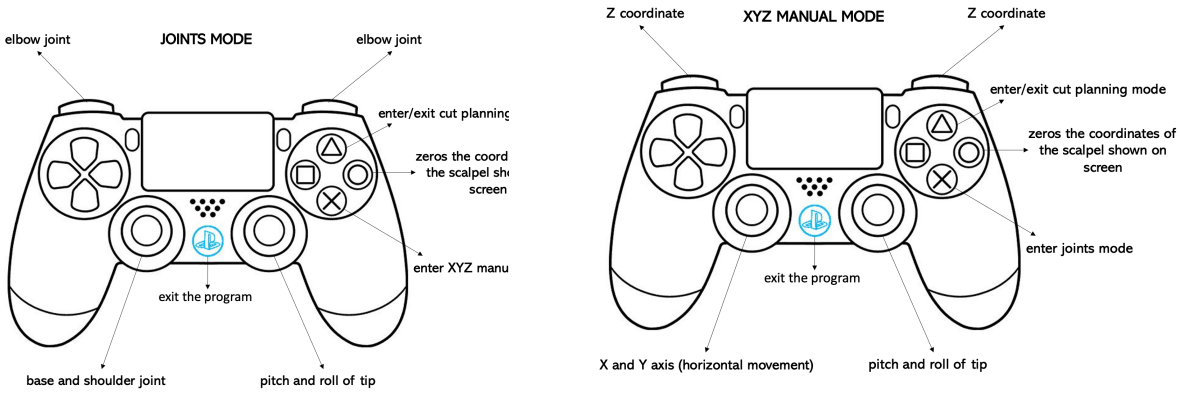

Figure 11: Controls for the Scalpel Robot: Joints Mode (Left) and XYZ Mode (Right)

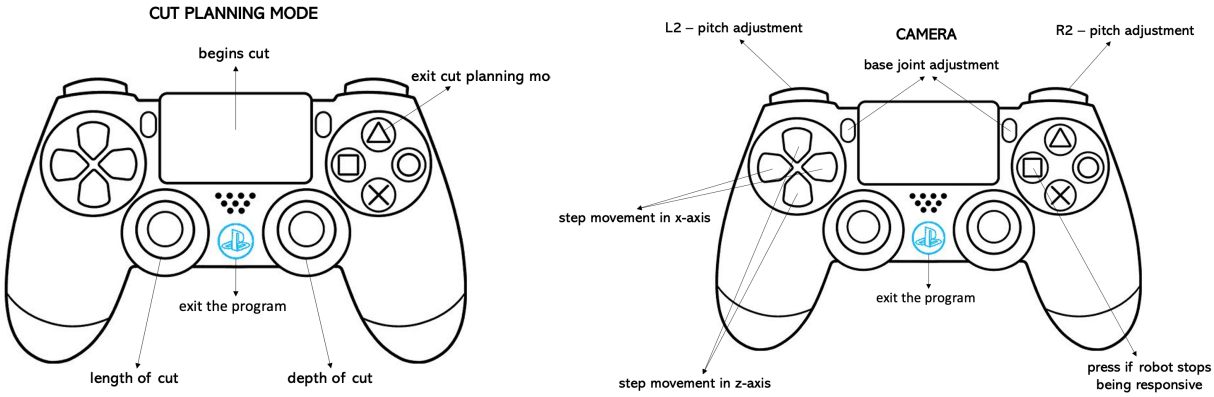

Figure 12: Controls for the Scalpel Robot: Cutting Mode (Left) and the Camera Robot (Right)

7.2 Initialization, Scalpel Robot's Current Mode and Finalization Messages

Besides the images, another window located at the bottom left corner of the screen shows six different text messages:

- Initializing
- Running in joints mode
- Running in XYZ mode
- Preparing for Cut
- Cutting
- Finished running

When the program is being started, the message "Initializing" is shown. The message of each of the modes of the scalpel's robot is updated in real time when the operator switches between modes. When the program is terminated, the message "Finished running" is shown. The idea of this is to help the operator know the current mode of the program and the scalpel's robot to avoid confusions when controlling the system. See the bottom left corner (black rectangular window) of Figures 2, 4, 5, 6 and 7.

7.3 Scalpel's Last Position, Length of Cut and Depth of Cut

The program prints on the aforementioned window, and under the previous messages, the last calculated position of the scalpel in X,Y,Z coordinates. This is crucial to know how much the robot moves in each direction after zeroing the coordinates. Since the operator can only look at the graphical interface, knowing the approximate position of the table in the Z coordinate can help them know the moment the scalpel is touching the gel (if they pay attention to the scalpel's position printed on screen).

Another message appears under the scalpel's position when the operator is ready to perform the cut, which indicates the length and depth of the cut (in millimetres) that is about to be done. This means that before cutting, the operator is able to pre-define and know how long and deep the robot is going to cut, avoiding issues if this step was manually done (these values change in real time as the user inputs them in the controller). See the bottom left corner (black rectangular window) of Figures 2, 4, 5, 6 or 7.

7.4 Timer

At the bottom right corner of the same window, a timer appears a soon as the program is started, allowing the operator to know how much time has passed from the beginning to the end of the procedure. This feature could be critical in a real life scenario, where time limitations to perform a specific surgery are set. See the bottom left corner (black window) of Figure 2.

7.5 Collision Warning Message

A red collision warning message "Imminent Collision" appears at the window containing the video itself, specifically at the top left corner. This message pops-up when both robots are about to collide, due to the fact that the distance between them is below a fixed security distance. At the same time that this message pops-up, the robot's movement is stopped for a few seconds, in order for a collision to be avoided. After the movement is reestablished, the message will still be visible until the robot's end-effectors are farther away from each other. This message allows the operator to know at any time that the robots will not collide, unless he is warned about it (giving confidence to the user). See Figure 13.

It was intended to develop a way to find the distance between the end-effectors of the scalpel and camera robots using a tracker included in python's OpenCV library, which allowed to track the scalpel and subsequently, find the distance between the robots (to manually avoid a collision). Another approaches were also tested to find the depth between the camera and the scalpel, which involved using python's MiDas library, specifically monocular depth estimation.

At the end, both approaches were discarded due to the fact that they did not provide good results when tested, since it is really challenging to find a distance or depth between a 2D

camera and an object. In the next section, it will be seen how the group managed to avoid collisions between the two robots.

Figure 13: Red "Imminent Collision" text message that warns the user of a possible collision.

8 Collision Avoidance

8.1 Camera calibration

In the previous sections (3.3), we explained how the positions of the end effectors of both robots were obtained. However, these coordinates are related to its corresponding reference frame, so we needed to get the relation in position and orientation between the reference frame of the camera robot and the one of the scalpel robot. This task seemed difficult as we would need to get this information only from the data we get from the robots and the camera image, ideally without looking or interacting with the tables that hold the robots.

In order to do that, the initial approach was to try to get the position of the scalpel robot relative to the camera by making use of the OpenCV library. The idea was to place a checkerboard pattern over the base of the scalpel robot, so that the position of this pattern relative to the reference frame of this robot would be the same during all the experiments.

The code to perform this task, included in the *camera_calibration.py* file, would work as follows, even though it wasn't fully developed for reasons that will be explained later. First, we would enter the calibration mode, where we would take pictures of the checkerboard pattern from different angles. In this way, we would obtain the intrinsic and distortion parameters of the camera. With this information, we would take a new image from a known position and orientation of the camera, and we would perform the pose estimation, getting the transformation matrix (rotation and translation) from the chessboard reference frame to the camera reference frame. In the following pictures, we can observe some test pictures that we obtained of the checkerboard pattern after performing the calibration.

Finally, additionally to this transformation matrix, we would obtain the following transformation matrices: from the scalpel end effector frame to the robot reference frame; from the scalpel robot reference frame to the chessboard reference frame; and from the camera reference frame to its robot reference frame. Thus, we could get the position of the scalpel relative to the camera robot reference frame, and we could get the distance between the two end effectors even though the configuration of the tables that support the robots changed.

Nevertheless, after some weeks of work we decided to discard this approach for various reasons. In the first place, the quality of the images taken with camera is not good at all, so

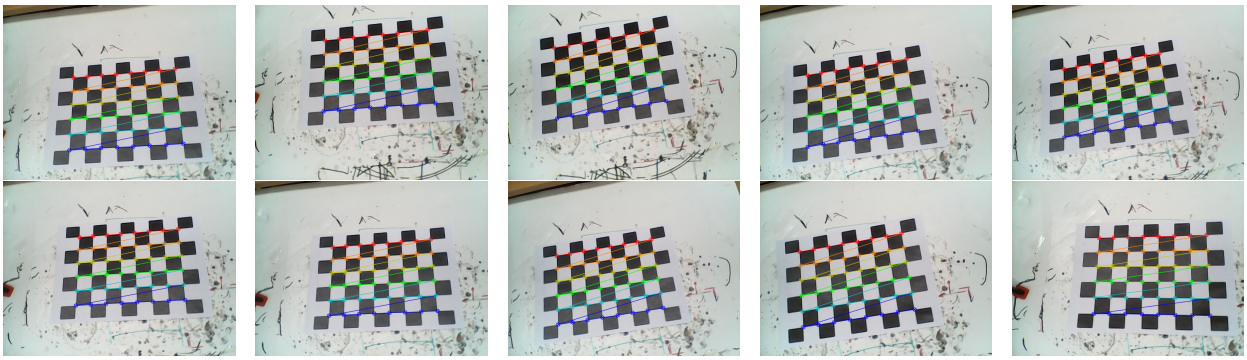

Figure 14: Calibrated pictures

the quality of the results obtained from the calibration could be affected. Also, the position and orientation information we obtain from the robots relate to their end effectors, and not to the tip of the scalpel and the camera. This was the main reason why we decided to abandon this methodology, as it was really difficult for us to precisely measure how much the camera and scalpel were translated and rotated with respect to their pertinent end effectors. This was specially hard for the camera, as the way it was attached to the robot arm was not really stable and it was impossible to get its actual position. For these reasons, we decided that it was not safe to use this method to perform the collision avoidance, as there would not have been any warranty of safety when operating the system.

8.2 Direct distance measurement

After realizing getting the relation between the two reference frames with the previous method was not possible, we decided to go for a simpler way.

The methodology we finally used for this tasks, which is included in the final code, is the following: Prior of the start of the robots control, the user makes sure the x axis of both robots are aligned, and then he measures the distance between the bases of the two robots. Once the user starts running the program, he gets asked to input this distance.

With this information, the program will be continuously check if there is danger of collision between the two robots. The way this is achieved is by getting the transformation matrix that relates the position of the scalpel given with respect to its reference frame to the camera robot reference frame. This way, the position of both end effectors is given in a common reference frame, so the function easily calculate the distance between the two points and return a collision message if this value is below a safety distance value.

This safety distance was obtained by running some tests in which we obtained x, y, z data of the end effectors in various configurations. Then, we assessed if there was risk of collision in each of those configurations in order to later then run the collision_avoidance function and see what the distance between the end effectors was in those risky positions. After these tests, we concluded that a safety distance of 200 mm ensured the correct functioning of the procedure.

Even though this way of getting the relative position between the two robots is not ideal, as it requires directly interacting with them and therefore doesn't allow a fully remote operation, it ensures a way of safely handling the robots with the available resources.

Figure 15: Robot's reference position for direct kinematics

8.3 Direct Kinematics

Another idea regarding creating a collision avoidance system was based around using direct kinematics to calculate the XYZ coordinate of the robot's end-effector, utilizing each of the joint values. Since the robots are both moved mainly on manual mode, and exiting it to retrieve the current joint values was proved as inefficient (resulting in a stiff robot movement), the joint values used as inputs in the direct kinematics matrices were predicted using a linear model with data of the joint movement relative to joystick's inputs, as mentioned in 3.3.

Once it was possible to obtain XYZ coordinates of the robots' end-effectors using joint values, the only thing needed to do collision avoidance would be to shift and rotate one of the robot's origin reference to the others, leading to having both end-effectors position in the same reference, where they could be compared and the distance between them computed.

After some discussion, it was believed that the easiest way to compute the robot's direct kinematics would be to assume a position where it would be easy to obtain the Homogeneous Transformations matrices and then, experimentally, see what were the robot's encoder values in that position, using them has offsets for each parameter. In the end, it was decided to use the position in image 15.

Taking as reference the position in 15, the D-H parameters used are shown in table 2.

It should be noted the presence of each joint offset (values in table 3), represented by δ_i, and EF corresponds to the value from the beginning of the wrist to the end-effector that changes between the two robots. Also, all the measurements were as shown in [1], unless the horizontal distances between the body and upper arm (0.135 m) and between the upper arm and forearm (0.1 m). Both were measured manually on-site.

Table 2: D-H parameters

i	a_{i-1} (m)	α_{i-1} (rad)	d_i (m)	θ_i (rad)
1	0	0	0	$\theta_1 + \delta_1$
2	0.05	0	0.3585	0
3	0	$\frac{-\pi}{2}$	0	$\theta_2 + \delta_2$
4	0	0	-0.135	$\frac{-\pi}{2}$
5	0.3	0	0	$\theta_3 + \delta_3$
6	0	0	0.1	0
7	0.35	0	0	$\theta_4 + \delta_4$
8	0	0	0	$\pi/2$
9	0	$\frac{\pi}{2}$	0	0
10	0	0	0.05	$\theta_5 + \delta_5$
11	0	0	EF	0

Table 3: Joints' offsets in encoding values

i	δ (encoder value)
1	0
2	-11350
3	8704
4	809
5	0

Using the values on 2 is possible to fill out the general transformation matrix between each reference frame (seen in [2]), resulting in 11 total matrices. The multiplication between all of them results in a homogeneous matrix with both the rotation matrix and the position vector inside it. For this case, only the position vector would be used. The computed matrices are as follows.

$$ {}^{0}_{1}T = \begin{bmatrix} \cos(\theta_1 + \delta_1) & -\sin(\theta_1 + \delta_1) & 0 & 0 \\ \sin(\theta_1 + \delta_1) & \cos(\theta_1 + \delta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^{1}_{2}T = \begin{bmatrix} 1 & 0 & 0 & 0.05 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.3585 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{2}_{3}T = \begin{bmatrix} \cos(\theta_2 + \delta_2) & -\sin(\theta_2 + \delta_2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin(\theta_2 + \delta_2) & -\cos(\theta_2 + \delta_2) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^{3}_{4}T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0.135 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{4}_{5}T = \begin{bmatrix} \cos(\theta_3 + \delta_3) & -\sin(\theta_3 + \delta_3) & 0 & 0.3 \\ \sin(\theta_3 + \delta_3) & \cos(\theta_3 + \delta_3) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^{5}_{6}T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{6}_{7}T = \begin{bmatrix} \cos(\theta_4+\delta_4) & -\sin(\theta_4+\delta_4) & 0 & 0.35 \\ \sin(\theta_4+\delta_4) & \cos(\theta_4+\delta_4) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^{7}_{8}T = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{8}_{9}T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^{9}_{10}T = \begin{bmatrix} \cos(\theta_5+\delta_5) & -\sin(\theta_5+\delta_5) & 0 & 0 \\ \sin(\theta_5+\delta_5) & \cos(\theta_5+\delta_5) & 0 & 0 \\ 0 & 0 & 1 & 0.05 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{10}_{11}T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & EF \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

In the end, and after a lot of effort, it wasn't possible to correctly implement the forward kinematics of the robot. While testing function *forward_kinematics()*, it wasn't possible to correctly obtain the end-effector position, when comparing to the XYZ output from the command *LISTPV*. This was probably due to an error in the translation between encoder values to radians or in the D-H parameters. Either way, the group tried to fix the error, but it wasn't possible to do so in time for the demonstration, which lead to creating another strategy to do collision avoidance, mentioned below, involving the method to get the end-effector's XYZ position mentioned 3.3 with a simple reference transformation.

Even though the direct kinematics function wasn't implemented in the final project, it was thought that it should be mentioned in detail due to the amount of effort put into it.

9 Conclusion

After testing the completed code several times, the group concluded that the implemented system is close enough to a working model, as a normal person can complete the wanted task almost every time (cutting an incision on a piece of gelatin).

With this said, in order to implement it on a real-life situation, some things needed to be fixed:

- **More visual feedback** about where the Scalpel is in relation to the Gelatin is essential for a secure operation. This is because it is very hard to know with precision where the Scalpel is, visually. This could be solved with: the addition of a second camera, positioned to the side; a 6th DOF on the camera robot, that would enable it to look at the Gelatin from the side; or if no hardware modification could be made, the better option would be to print some distance vectors on the GUI, that show distance to the Gelatin, using computer vision.

- **Accurate predictions of the Scalpel's position**: The current system performs a "LISTPV" command to get the current position of the robot approximately every 7 seconds. In the mean time, some linear position prediction are made based on the inputs given to the robot. These intermediate predictions are not completely accurate and that is why a "LISTPV" command needs to be done frequently. For a final working system,

these predictions would need to be optimized in terms of accuracy, so no interruptions of movement would need to be made in order to perform the "LISTPV" command. This completely continuous type of movement would improve the trust that the user can have while operating the robots.

- **Precision in the movement**: For a medical application, the robots would need a lot more precision while moving.
- **Fixed robots**: For a real application, both robots would need to be bolted to a rigid structure. Their exact locations would need to be known, which would help predicting the position of the Scalpel and performing colision avoidance. This would not only ensure precision but also repeatability.

Some good implementation that are already working in the code and that approximate it a lot to a system that is ready to be used in surgical circumstances are:

- **Remote operation**: A situation in which it would be desirable to perform a surgical operation using this system would be if the doctor is not in the same place as the patient. For this reason, our system implements a 2 computer architecture. One of the computers is located with the robots/patient and controls the movement of the robots. The other computer should be with the doctor and displaying the video feed (besides information about the robots' states). These 2 computers communicate via sockets, so they can be as far away as needed, if they are connected to Ethernet. The joystick is also wireless and can be located with the doctor.
- **Collision detection**: This is already implemented and should give confidence to the user to move the robots without fear of causing a collision. In a later prototype, it would also be essential to expand this feature for collision detection with surrounding objects (table, walls, etc.). To do this, we would need to know all the coordinates of the surroundings and implement boundaries so the robot's do not collide with these objects.
- **Responsiveness**: For the most part, the program is responsive in real time. It also displays information on the GUI without the user picking up on any visible delays. This is essential on any real-world use case.

Overall, the implemented system works well and is not too far away of a working prototype that could be tested in real world scenarios.

References

[1] SCORBOT-ER VII User's Manual (2nd edition) User Manual, Eshed Robotics, (1996, reprinted 1998).

[2] Sequeira, J., (2023). *Lecture 5*, Instituto Superior Técnico, https://fenix.tecnico.ulisboa.pt/downloadFile/1970943312409305/lecture_5.pdf

[3] Tech with Tim. (2020, April 5). Python Socket Programming Tutorial. Youtube. https://www.youtube.com/watch?v=3QiPPX-KeSc