Group Description:
This assignment was written by - Afonso Araújo (*96138*), Diogo Mucharrinha (*96179*), João Santos (*96237*) for the Robotics Course of the 1^{st} Semester, P2, of 2022/2023, taught by professors João Sequeira, Alberto Vale and Zhiqi Tang.

1 Introduction

The following report guides the reader through the **Background** behind this Serial Manipulators assignment, the **System Architecture** developed by the group in order to achieve as good an output as possible, the **Main Challenges** the group faced throughout this 5 week assignment, the groups's **Experimental Results** and, to conclude, **Scalability and Future Work**.

2 Background

This assignment came into existence with the purpose of allowing the students of the Robotics course to better understand the concepts of Kinematics, such as Direct and Inverse Kinematics, and to gain a sensibility towards hardware with actuators, sensors and serial manipulators.

The goal of this assignment was to draw an image, without lifting "the pen off the paper", with the Scorbot VII, the manipulator to control on this task. However, in order for it to draw such image, some image processing techniques were applied, with the aid of Python, so the Scorbot VII was able to fulfill its task.

As it will become clearer with the next chapters, specifically 3.2.2, there was a main focus on using concepts such as Inverse Kinematics in order to move the Scorbot VII.

Several sessions later the group was able to draw the selected images, as we can see here 5.

3 System Architecture

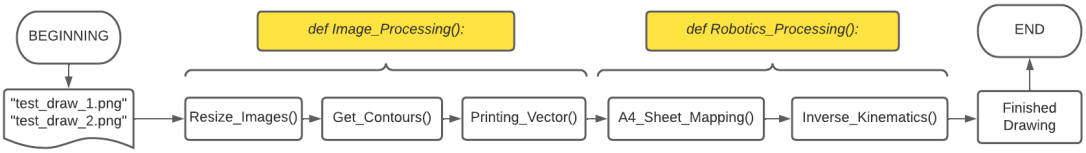


Figure 1: System Architecture

3.1 Image Processing Architecture

Image processing constituted a large part of this assignment because the raw images that were given needed some modifications before they could be analyzed. Afterwards, it was necessary to remove the contours of the image to obtain arrays that the robot could read. To do so, we used an *OpenCV* function and then applied several filters to get a near-perfect array that reduced the number of points needed to complete the drawing.

3.1.1 Resize

As mentioned above, it was necessary to apply some changes to the images.

We first started using a black and white Google image of a fish, but the lines of such an image were so thick that after it was processed by the *findContours()* function, it led to a disfigured drawing. Such technique was successfully replaced after discovering the test images had thinner lines.

With this in consideration, there was left the problem of the test image's sizes. This was due to the fact that such images were too large to be processed. To solve this issue, we rescaled the images to a fraction of their size by using an *OpenCV* resize function and then applying their original aspect ratio.

3.1.2 Obtaining Contours

To obtain a vector of coordinates that could be passed to the robot, we decided to use the function *findContours()* from *OpenCV* since this provided us with a vector containing the coordinates of the points belonging to the image traces.

This pre-made algorithm made our work easier since these points were already ordered, so the robot could create a continuous line when running the vector. Although the contours were easily obtained, it was necessary to reduce the number of lines, the number of points and to optimize the pathfinding of the pen when drawing.

The function *findContours()* returns multiple contours having some lines in common, thus overlapping certain parts of the drawing.

To solve this issue, we created a function (*get_new_contours()*) that stored the most extensive outline intact and removed all the points of the second one near the already stored ones. After, we replaced the second contour with all the new coordinates. This process was iterative until there were no unseen contours on the list.

Algorithm 1 Get_New_Contours() Design

```
for i = 1, 2, .., N do                                  ▷ N = number of contours
    my_idx ← largest_contours_index
    max_list ← contours_list[my_idx]
    for j = 2, 3, .., M do                              ▷ M = len(contours_list[my_idx])
        for k = 1, 2, .., O do                          ▷ O = len(max_list)
            if distance(i, j) ≤ 3 then
                Flag == True
        if Flag == True then
            max_list[ ] ← point
            min_list[ ] ← point
contours_list[my_idx] ← min_list
```

Once eliminated all the unnecessary points of all contours, but the first, we removed all empty vectors from the main list and started working on the first contour.

When analyzing the outcome of the *findContours()* function, we noticed that some parts of the outer contour were repeated because the outline extended around the lines of the image as

if it were a bidimensional object. So any loose ends would have the kind of contour shown in Figure 2

Knowing that each contour has at least one point for each variation of one unit in each coordinate and the corresponding vector is sorted, we remove all the points within a distance of two units that were outside an interval of three indices from the analyzed point.

This way, we could ensure that all the loose ends of the outer shape of the drawing would only have half of their outline.

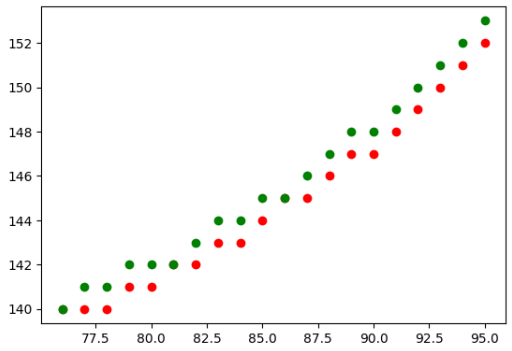
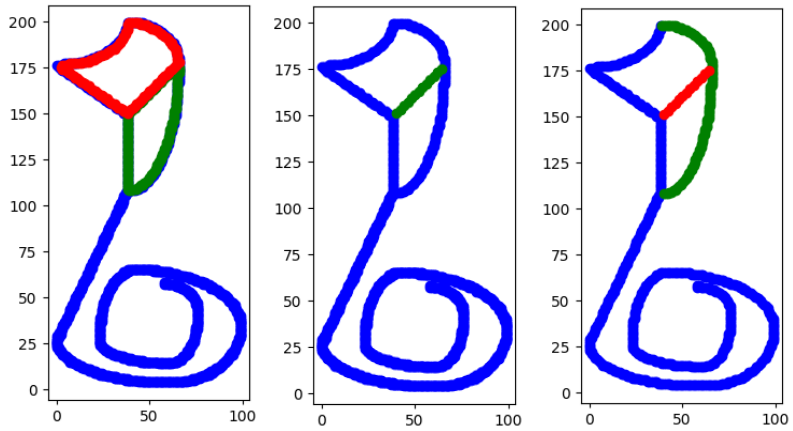


Figure 2: Two sides of the same contour of a loose end

Taking the Figure 2 as an example, one of the two sets of points was eliminated depending on the beginning of the contour. The function that performed this refinement was called *get_mask_first()*.



(a) (b) (c)

Figure 3: (a) Vector contours_list (b) Vector new_list (c) Vector connected_list

However, when there are points that are removed from the contours array, there will be a discontinuity in the outlines of the images. For this, a function was created that detects any interruption in those vectors. So *get_mask()* calculated the distance between each point in the analyzed array, and if this value was greater than a defined threshold, the contour was divided in two.

Once all the contours were processed, to reduce the number of points in each one, we started connecting them so that the robot could make the drawing in one fell swoop.

3.1.3 Printing Vector

In order for the robot to make the drawing smoothly it was necessary for our program to be capable of providing the robot with a single array that contained all of the contours and the points between them (backtracking between the contours). To do so, we implemented the function *connec_cnt()*.

Algorithm 2 Connec_Cnt(countours) Design

```
new_countour ← countours[0]
first_point, last_point ← −1
for i = 1, 2, .., N do                                  ▷ N = number of contours
    for j = 0, 1, ..., S do                             ▷ S= size of the contour
        if distance(new_countour[j], countours[i][0]) < 4 & first_point == −1 then
            first_point = j
        if distance(new_countour[j], countours[i][−1]) < 4 & last_point == −1 then
            last_point = j
    if first_point > last_point then                    ▷ Closer to end of new_countours[]
        new_countour ← Inverse(new_countour[first_point : len(new_countours) − 1])
        new_countour ← countours[i]
    else
        new_countour ← Inverse(new_countour[last_point : len(new_countours) − 1])
        new_countour ← Inverse(countours[i])
```

After getting the single array that was going to be printed, we then passed the array through a mask in order to obtain a smaller vector that maintains the information needed.

3.2 Robotics Architecture

Regarding the Robotics part of this project, the group tackled the problem in two fronts - the mapping of the drawing to an A4 sheet of paper, and the Inverse Kinematics programming of the Scorbot VII.

3.2.1 A4 Sheet of Paper Mapping

Since we are to draw our image into an A3 sheet of paper, glued to the table, the group decided to map out a window inside an A4 sheet of paper in order to have "wiggle" room for mistake. Such window would be a (15×23.7) cm window inside a common (21×29.7) cm A4 sheet of paper.

In order to illustrate the Linear Transformation done for every received image we can take, as an example, a simple square mapped in $\mathbb{R}$,

where,

$$(square\_vector_{5\times2})^T = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix} \tag{1}$$

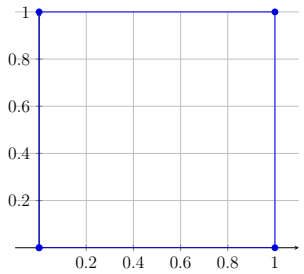


Figure 4: Square in $\mathbb{R}$

whilst having the following Linear Transformation,

$$coef = \begin{cases} \frac{(picture\_resolution \times 23.7)}{max(square\_vector[:,1])}, & max(square\_vector[:,1]) \geq max(square\_vector[:,0])) \\ \frac{(picture\_resolution \times 15)}{max(square\_vector[:,0])}, & max(square\_vector[:,0]) \geq max(square\_vector[:,1]) \end{cases} \tag{2}$$

$$square\_vector = coef \times square\_vector \tag{3}$$

where *picture_resolution* is the comparative size of the image regarding its maximum possible size., i.e, $picture\_resolution = 75\%$ means the image will be at 75% of the window size.

From here we can go to our mapping in $\mathbb{R}^2$, whose z-axis is fixed, representing our non-moving paper above the table on which the drawing is created,

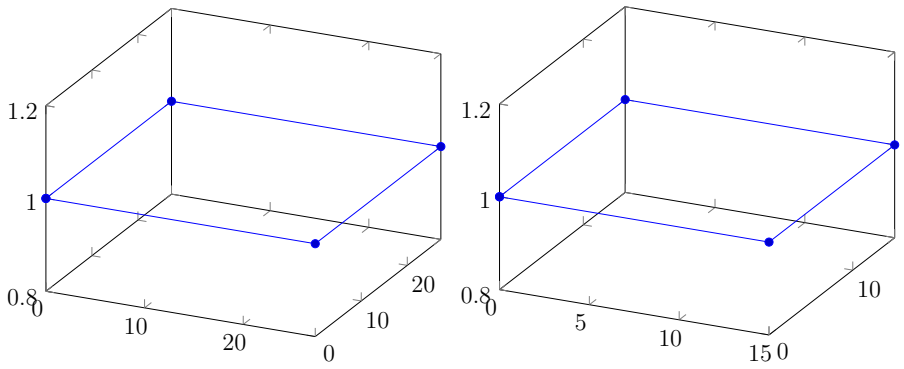


Figure 5: Left: Case where our square is greater on the y-axis. Right: Case where our square is greater on x-axis

leading to,

$$square\_vector_{5\times3} = \begin{pmatrix} 0 & 0 & table\_z \\ 0 & 23.7 & table\_z \\ 23.7 & 23.7 & table\_z \\ 23.7 & 0 & table\_z \\ 0 & 0 & table\_z \end{pmatrix}, or, square\_vector_{5\times3} = \begin{pmatrix} 0 & 0 & table\_z \\ 0 & 15 & table\_z \\ 15 & 15 & table\_z \\ 15 & 0 & table\_z \\ 0 & 0 & table\_z \end{pmatrix} \tag{4}$$

In a real scenario, $square\_vector[N]$ would be our $image\_proc\_vector[N]$, or $new\_countour[N]$ as stated before, making a segway into our next chapter, Inverse Kinematics.

3.2.2 Inverse Kinematics Programming

For our Inverse Kinematics block, we jump into the Scorbot VII hardware and our "direct-communication" with it. Below are the two main algorithms responsible for the drawing of our images, using the $image\_proc\_vector[N]$ from before.

Algorithm 3 SVECT[N] Creation

$P_X = current\_marker\_pos()$
$P_1 \leftarrow P_X$
$P_{1(z)} \leftarrow P_{X(z)} + 500$
$svect = empty\_vec(N)$
for $i = 1, 2, ..., N$ **do**
    $svect[i] \leftarrow P_X$
    $svect_x[i] \leftarrow image\_proc\_vector_x[i-1]$
    $svect_y[i] \leftarrow image\_proc\_vector_y[i-1]$
    $i \leftarrow i + 1$
end for

Starting with the Algorithm 3 analysis, P_X represents the point where the marker is touching the paper, manually set by a member of the group in order to bypass the need to go from "Home" to a point near the paper through software and joint manipulation.

Therefore, P_X will be outside the range of our drawing, creating the need to lift the marker (5 cm above the table) with the point P_1, in order to create a drawing as clean as possible.

Following this we have our transfer of information from the $image\_proc\_vector[\,]$ to our $svect[\,]$.

Algorithm 4 Scorbot VII Movement Instructions

$move\_scorbot(P_1, std\_speed)$
$P_{Init} \leftarrow svect[1]$
$P_{Init(z)} \leftarrow P_{1(z)}$
$P_{Final} \leftarrow svect[N]$
$P_{Final(z)} \leftarrow P_{1(z)}$
$move\_scorbot(P_{Init}, std\_speed)$
for $i = 1, 2, ..., N$ **do**
    $move\_scorbot(svect[i], speed\_coef)$
    $i \leftarrow i + 1$
end for
$move\_scorbot(P_{Final}, std\_speed)$

To analyze our Inverse Kinematics programming we can jump inside our $move\_scorbot()$ function.

Very early on, the group realized Inverse Kinematics would be a smoother and easier method to implement and process our code through. The reason behind this was the fact that if we were able to process our image into cartesian coordinates, as seen in 4, we could simply program the joint-system of the Scorbot to reach such coordinates on the piece of paper, instead of having to program every single individual joint to do so.

Such, reveals the inner-workings of the *move_scorbot*() function, where we command the Scorbot, coordinate pair by coordinate pair to do anything in its capability with its joints to reach them, at a certain speed. Where any pre-determined point, P_X, P_1, P_{Init}, P_{Final}, moves at a standard speed of 10%, $svect[]$ is drawn at $speed\_coef$, explained later at 4.2.1.

However it is to note that a Direct Kinematics approach was also taken into consideration, when the certainty of placing the marker "manually" with the Scorbot remote control was but a possibility. Then, there would be a need to program joint by joint the Scorbot to go from "Home" to P_X.

To conclude the Algorithm 4 analysis, the need for both P_{Init} and P_{Final} came about the fact that in order for the drawing to be as clean as possible, it was necessary that the marker began exactly above the first point of the drawing and lifted off the table exactly above the last point of the drawing.

4 Analysis on the Main Challenges Faced

4.1 Main Image Processing Challenges

The image processing challenges encountered were mainly because of the need to avoid overfitting the program to the given figures. They can be divided into two: conflicting images and tunning mask thresholds.

4.1.1 Conflicting Images

Since we only had two image examples as a point of reference, provided one week after the assignment's release, we started using other kinds of images that needed different processing. Although it's easy to distinguish which type of image we are working with, it is more difficult to generalize the algorithm to obtain consistent results for different inputs. This is the reason why, later at 5 we see some backtracking being done with "test_draw_1.png".

On the other hand, if we only focus on the provided images, we tend to overfit the code for them; given that the number of image samples is reduced, paying attention to some unique characteristics of these same samples is necessary.

For example, the function that chooses the beginning of the main vector based on the distance between the start/end of the first contour and all other starts/ends of the secondary contours is prone to giving errors in some misleading shapes.

4.1.2 Regularization of Thresholds

The other great challenge we had to face in the Image Processing side, was the regularization of thresholds we used in order to apply the various masks to get the final ready-to-print vector. To solve this challenge we had to calibrate very carefully the value we introduced because in the end, any problem would make our program not work as intended.

4.2 Main Robotics Challenges

The Main Robotics Challenges can be focused on a single physical system, beyond the Scorbot VII, the **marker**.

4.2.1 The Marker

The marker at the tip of the Scorbot´s "head" would sometimes become an unstable system. The aluminium protection that surrounded the marker in order for the Scorbot to have a tighter grip on it, was connected through a fine and thin wire bolted to the end of the protection, and the marker itself.

Such system would give the marker an unpredictable spring-like action that, in times of drawing, could lift the marker of the table, or "shake" the marker in a different axis than the one it was used to being drawn with, leading to irregular lines on the drawing.

The group thought of making use of the Roll property of the Scorbot VII in order to counter-act the "shakiness" of the marker. However such strategy became too time consuming.

Therefore, the group simply increased the amount of points of the drawing, increasing the time required by the Scorbot to draw,

$$speed\_coef = 50 \times len(svect) \tag{5}$$

but increasing the stability of the drawing, where $speed\_coef$ equals the total amount of time to draw.

5 Experimental Results

As for the experimental results, the following are behind the link: https://youtu.be/ddvADxQlvGo.

In the video it is shown how the stability increased with the amount of points, as stated before 4.2.1, with an increase in time, and how the physical system behind the marker could be an agent behind the disruption of the drawing decreasing the likeness to the original.

6 Scalability and Future Work

As for scalability and future work, it is easy to see how this contained experiment can become useful for larger automation tasks in several industries, such as the automobile or aerospace industries.

Either through Inverse or Direct Kinematics, the ability to automate a robot to perform a task is extremely important when dealing with large-scale productions. A more concise example of direct application of our work, can be the automation behind soldering in the metallurgical industry.

Since we have the image processing done, we could "simply" swap our marker with a laser and exact precise soldering on a metal sheet.

7 Bibliography

[1] Robotics (Lectures 1-4) - Slides of Theoretical Classes 2022/2023, 1st Semester (MEEC), by Professor João Sequeira