ENGINERING IN INDUSTRIAL TECHNOLOGIES

DESIGN OF PROGRAMS FOR LOCALIZATION, DIRECT AND INVERSE KINEMATIC FOR A CLIMBING ROBOT.

FINAL DIPLOMA WORK

Tutor:
Dr. Riko Šafarič

Author:
JAVIER MAQUIRRIAIN ANTOÑANZAS

Public University of Navarre
Higher Technical School of Industrial Engineers and Telecommunications

University of Maribor
Faculty of Electrical Engineering and Computer Science

INDEX

1. Abstract.....2
2. Key words.....3
3. Introduction (2-3 pages) 4
    3.1 Motivation.....4
    3.2 Structure 6
4. Problem presentation 6
  4.1 Description of robot's mechanism.....6
  4.2 Description of algorithm 7
    4.2.1 What we want to do 8
    4.2.2 Goals 9
    4.2.3 Diagrams 9
5. Software description.....16
  5.1 Documentation of algorithm in detail 16
    5.1.1 Relative distances' calculations – Part 1 16
    5.1.2 Representation of the map of holes – Part 4.1.....19
    5.1.3 Extra checking function.....20
    5.1.4 Distance's comparison and hole's search – Part 2.....20
    5.1.5 Calculate centre's position – Part 3 32
    5.1.6 Representation of the robot's configuration – Part4.2 34
    5.1.7 Inverse program.....39
6. Conclusion.....42
  6.1 Result 42
  6.2 Advantages and disadvantages.....43
    6.2.1 Advantages.....43
    6.2.2 Disadvantages 44
  6.3 What could be improved 44

1. Abstract

This project is part of a group of project destined to the creation of a **climbing robot**, and is aimed at **programing** the microcontroller with the function needed to perform its function.

This project is focused on the **design of an algorithm for the localisation** of the robot's position in a map of a surface with the information provided by the sensors of its components. With this aim in the project, two programs will be developed, one solves the problem and the **equations of the direct kinematic** (main program) and the other the **inverse kinematic** (auxiliary program) of the robot.

In the main program apart from solving the direct kinematic of the robot, a function used to **triangulate position** will be implemented.

With these programs the robot must be capable of fulfilling the following specific technical requirements:

- To know exactly the robot's positions in the surface in which it is affixed with the help of a map and the sensors of the robot servo motors.

- To be able to determine the coordinates of the robot's centre with the measures provided by the sensors. In other words, based on the information of the links' configuration, achieve the position of the robot's centre (direct kinematic).

- To be able to determine the position of the robot servo motors when the robot arrives to a desired position. In other words, based on the desired position of the robot's centre, achieve the configuration needed for the robot's links (inverse kinematic).

Due to the experimental character of the project, the programs will be developed first in the software package Matlab.

The climbing robot developed by another group of project has a lot of possible uses and functions like the repair and maintenance of structures and surfaces related to the construction, such as bridge's legs.

2. Key words

Robot: A mechanical device that sometimes resembles a human and is capable of performing a variety of often complex human tasks on command or by being programmed in advance.

Direct kinematics: Refers to the use of the kinematic equations of a robot to compute the position of the end-effector from specified values for the joint parameters.

Inverse kinematic: The reverse process that computes the joint parameters that achieve a specified position of the end-effector.

Localization: the act of determining the position or determinate property of a body, usually by research or calculation.

Triangulation: a technique for establishing the distance between any two points, or the relative position of two or more points, by using such points as vertices of a triangle or series of triangles, such that each triangle has a side of known or measurable length (base or base line) that permits the size of the angles of the triangle and the length of its other two sides to be established by observations taken either upon or from the two ends of the base line.

Servo: A servomotor (called servo) is a rotary actuator or linear actuator that allows for precise control of angular or linear position, velocity and acceleration. It consists of a suitable motor coupled to a sensor for position feedback. It also requires a relatively sophisticated controller, often a dedicated module designed specifically for use with servomotors.

Link: A rigid part of a manipulator, which connects adjacent joints.

Arm: An interconnected set of links and powered joints comprising a robot manipulator that supports and/or moves a wrist and hand or end-effector through space. The arm itself does not include the end-effector.

3. Introduction

In this document we are going to set out the project of the **design and development of the localization algorithm for a climbing robot**.

The development of a climbing robot is going to be divided in many parts: the mechanical design and production of the robot's components, programing and setting up of the sensors, algorithm of sequence of movements, auto-localization and mapping program… But we are going to focus this project in the software of the robot and more precisely in how the robot is going to be able to know where it is, its position and how it is going to move itself.

So, the main objective of this project is to develop an algorithm and program which, though, the information of the angles of the engines of the robot's arms is able to get the **position of the robot's centre** (where the camera or other principal tools will be installed), and in the same way develop the inverse auxiliary program. This means that we should develop a program which is able to return us the angles needed in the arm's engines to be in the desired position.

With this we will be able to know where the robot depending on the arms' position is and how must be the arms to arrive at the objective position. Then using both programs in a cyclic sequence the robot will have the ability of moving itself following the points that we program.

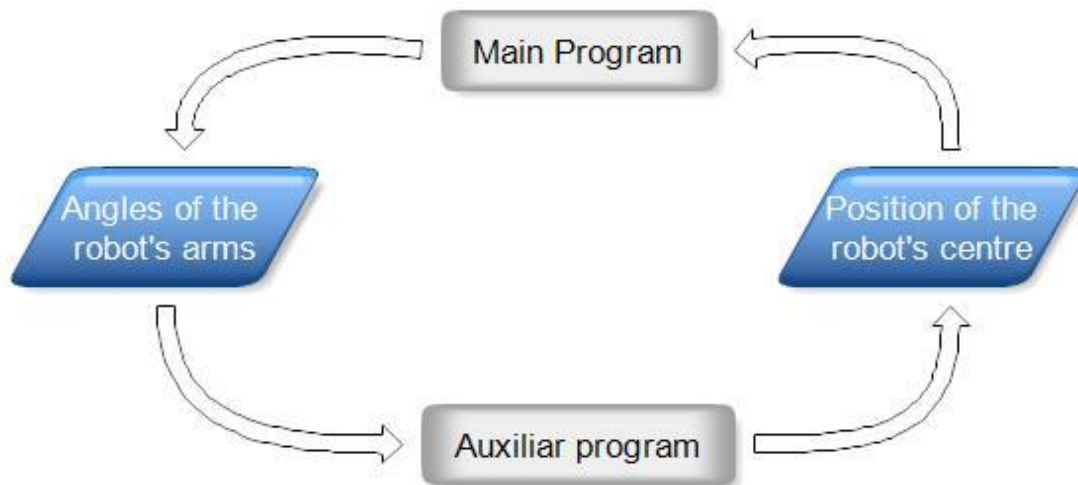

Figure 1: cycle of running process, direct and inverse kinematic.

3.1 Motivation

First of all, it is necessary to talk about the problem that encouraged us to do this project. We are going to focus our attention on the buildings like bridges and more particularly in these solid ones that are built with materials like concrete, not for example suspension bridges or drawbridges. We can have other examples like residential, commercial or industrial buildings but we have selected bridges because it is the type of structure that is supposed to bear more variable weight and fatigue.

In this kind of buildings and this kind of materials the pressure, the forces and over time usually cause cracking and this splitting is one of the most important cause of accidents or deterioration which result in a waste of money, time, workforce or in the worst case, the destruction of the structure.

Figure 2: Cracking in surfaces made of concrete

Because of that we are going to develop a robot capable of checking surfaces and detecting cracks for the following reparation. This robot must be able to climb and to go to areas which the human can't access (this is the main reason for the creation of the robot, the difficulty of checking big buildings or complicated zones with precision).

If we analyse the design of the new bridges we can realise that there are a lot of them built with reinforced concrete, and in these cases it is really usual to find regular holes in the surface, so this could be really useful for us to use them like support or grip to climb instead of making new ones.

Figure 3: holes in bridges

3.2 Structure

So, in this project the chapters are going to be organized by the next sequence of topics:

1. First of all we are going to analyse the **robot's mechanism**, because it is the first step to program the controller of the robot's servomotors.

2. Then we will **describe** how the **algorithm** must be in order to satisfy all the necessities and fit the requirements needed. We will be helped by diagrams, goals presentation etc.

3. Once that we have planned how the project may be developed we are going to do the **software presentation** with the documentation of the algorithm and program's code in a more precise way.

4. And finally to **conclude** we will analyse the project approach selected, advantages and disadvantages and possible improvements to compare with other possibilities, check if it is the best solution, and to help new projects that will use this one like support.

4. Problem presentation

In the previous chapter we have introduce the problem and the motivation of this project so now in this chapter we are going to talk about how we are going to solve them.

So we need to develop the algorithm and programs for the robot's movement, for this purpose we will control de **angles of the arms** with electrical engines (servos) and this geometrical configuration will depend on the physical structure of the robot. Because of that the first step is to analyse the robot's mechanism.

4.1 Description of robot's mechanism

The structure of the robot will be formed by one centre piece and 3 arms with 2 linkers and one final effector per arm emulating the configuration of a human arm with the articulations of **shoulder, elbow and wrist.**

The **central platform** will be equipped with a camera to see the surface (this camera will detect the cracks and imperfections) and other tools which will depend on the function desired for the robot (reparation, mapping, automation of complex process for human…). This piece is the most important and is the one whose position will be necessary for the control program, this is going to be considered the **robot's centre** and we will work with the coordinates of this point to control the camera or the tools implemented. The central piece will be connected with the first link of the 3 arms by one servomotor emulating the shoulder.

The **first link** will be connected between the central piece (shoulder) and the second link (elbow). This project is only a prototype so the length of the link may vary between different versions so in the program this distance shouldn't be fixed and we must have the possibility of changing it.

The **second link** connects the first link (elbow) and the final effector which will hook or fix the robot to the surface (wrist).

The **final effector** is a piece equipped with infrared sensors to detect the holes or the variations in the surface (these sensors are going to be the "eyes" of our robot) and with a mechanism to fix the arm to the wall's surface. This mechanism could be a drill or a hook.

For an easier understanding of the mechanism we have a schematic picture of the pieces and the disposition of them:

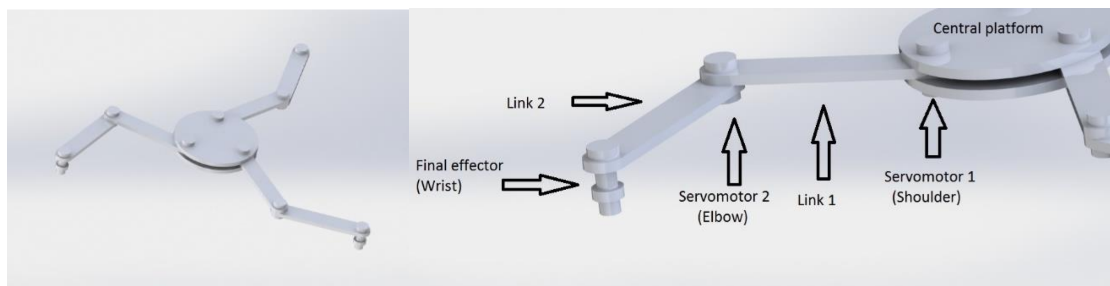

Figure 4: Robot's structure

4.2 Description of algorithm

In this part of the chapter we are going to describe the **algorithm** we want to develop with the **functions** required and with the **objectives** we want to achieve.

It's really important to know what information and equipment we have to start the design of the algorithm. The most essential things we must have are:

1. **A map of the climbing area**. For the location of the robot it is necessary because with this and with other information (next point) the algorithm should be able to determine the centre's position by a triangulation method similar to the operation of a GPS.

2. The **info of the 6 angles** provided by the **sensors** on the electric servomotors. This sensor's measure must have a minimum accuracy for the correct functioning of the program. The maximum error must be ±0.5°, but it will depend on the map (previous point), the distances between holes and the robot's size (Because 0.5° is not going to cause the same error if we are working with distances like 10 cm or like 1 meter).

4.2.1 What we want to do

The main objective of this project is to be able to **determine the position of the robot's centre** only with the information provided by the sensors of the arms and a map of the area, in other words, the principal goal is to know where the robot is using the arms' angles and the map of the surface. But for that we are going to divide the program in different tasks or subprograms.

Here we have a short and summarized description only to understand better the operation of the algorithm. In the next chapter it will be described in detailed.

First Part – Relative distances' calculation.

First of all we will need a program which can calculate the relative distance between the final effectors from the known angles. In a simple way this means that the program must calculate the distance between the "hands" of the robot, and if the robot is affixed to the wall's surface, these distances will be the same distances as the ones between 3 holes of the map.

Second part – Distance's comparison and hole's search.

Now we have the distances between the 3 holes in which our robot is affixed and with this, the next step is to find which of these 3 holes we have and where they are on the map. For that, the program will compare the 3 distances with all the possible distances between holes of the map and will consider a solution as correct when it finds 3 similar distances (similar because of the little error that we can commit) in 3 adjacent holes. These 3 holes must form a triangle which we will use for future calculations.

Third part – Calculate centre's position.

Once we arrive to this part with the holes position and the relative position of the robot, we can find the absolute position of the centre by simple calculations.

Fourth Part – Representation of the robot's configuration

At this point, the program has found all the information required: distances of the robot's configuration, in which wholes the robot is affixed, where is the centre etc.

So consequently, for an easier interpretation of the results, the program will create different charts to show the final information in a graphical way:

1. A chart showing the position of all the holes of the map to verify if there is any error before running the whole program.

2. A chart with the relative position of the robot showing the configuration adopted by the arms.

In the case that everything is right we have the possibility of running the program again, so the algorithm will superimpose the two charts and will show in one graphic the map and the absolute position of the robot like in the real case.

4.2.2 Goals

In this project, we have different goals and the achievement of them will help us to continue improving and focus the development of the process in the right way.

The main goal is obviously to achieve the proper **function of the robot**, it means that the robot must be able to **localize itself** in and know the **position of its centre** in the map.

A further aspect which must be addressed is the fact that we don’t have perfect equipment, sensors or analogue to digital converter and this will introduce an error in our program. However the program must be **sufficiently accurate** in order not to allow this error to change or affect the outcome.

Due to the fact that the program is going to be installed in the microcontroller of the robot, the program has to be as short and simple as possible, in other words, we must choose **the most efficient** possibility.

Another proposed goal is to be capable of checking if the program is working correctly or not. With this aim there are 2 different possibilities: the first one is to check with the robot in a real case, and the other option is to develop the **auxiliary inverse program** able to emulate the configuration of the robot.

And last but not least, for the interaction between the user and the program, it is really interesting the idea of developing and showing the **results in a visual way** to make easier the comprehension and interpretation of the outcomes.

4.2.3 Diagrams

In this part of the chapter we are going to try to explain how the algorithm will work when we run the program but in summarized diagrams only to understand the main structure and the fundamental principles. In the next chapter we will analyse step by step the whole code of the program with comments and remarks which will explain in detail all the functions, equations and commands.

Part 1 - Relative distances’ calculation.

This part is the one which may calculate all the relative distances of the robot based on the robot’s angles provided by the robot. The method is explained in the diagram and each calculation process has its own image to represent the calculations we have made.

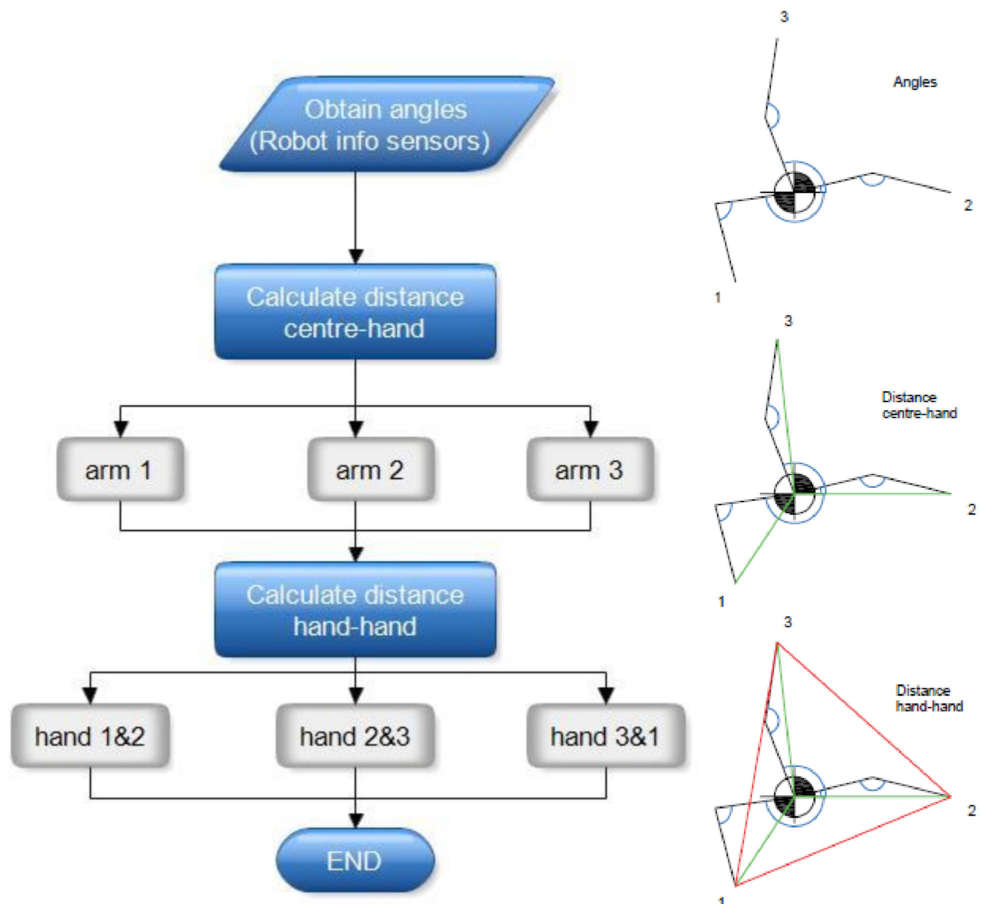

Figure 5: Diagram Part 1

Part 2 - Distance's comparison and hole's search

Now we have the 3 distances between holes obtained in the part 1 and we should find in which part of the map are they are. It is known that the three holes must be adjacent to each other, so this means that it is not necessary to compare all the possible distances, only those which are between contiguous holes. For each hole, 8 distances must be compared like the picture shows:

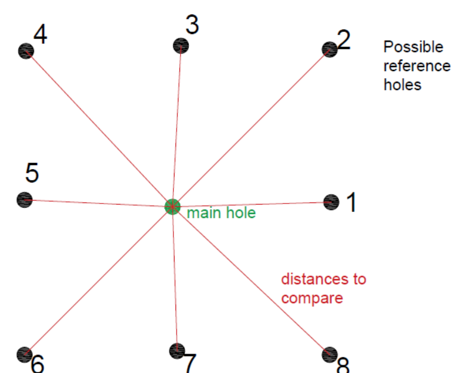

Figure 6: possible holes' combination

But to achieve this objective it's enough to compare 4 holes with one referenced hole and then, change this referenced hole. But we might be careful with the borders because they could be conflictive for the program.

Univerza v Mariboru

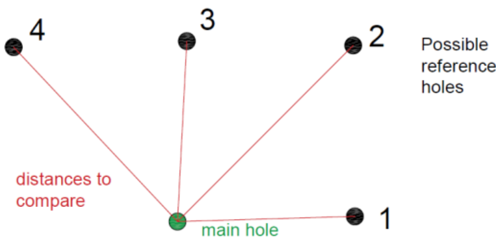

Figure 7: Most efficient way of holes' combination

Finally the algorithm is going to work like the next diagram shows:

Select first hole
Compare distance 1
Change hole
Right hole
Diagonal hole1
Up hole
Diagonal hole2
Equality?
No
Yes
Compare distances 2&3 with adjacents holes
Equality?
Yes
Solution found:
Point 1
Point 2
Point 3
END

Figure 8: Diagram Part 2

Part 3 - Calculate centre's position.

In the third part, the program is going to find the centre's position with the angles of the structure. For that, we will calculate the distances of the configuration again to avoid possible errors and to make independent the third and the first part in order to have the option of running both programs separately.

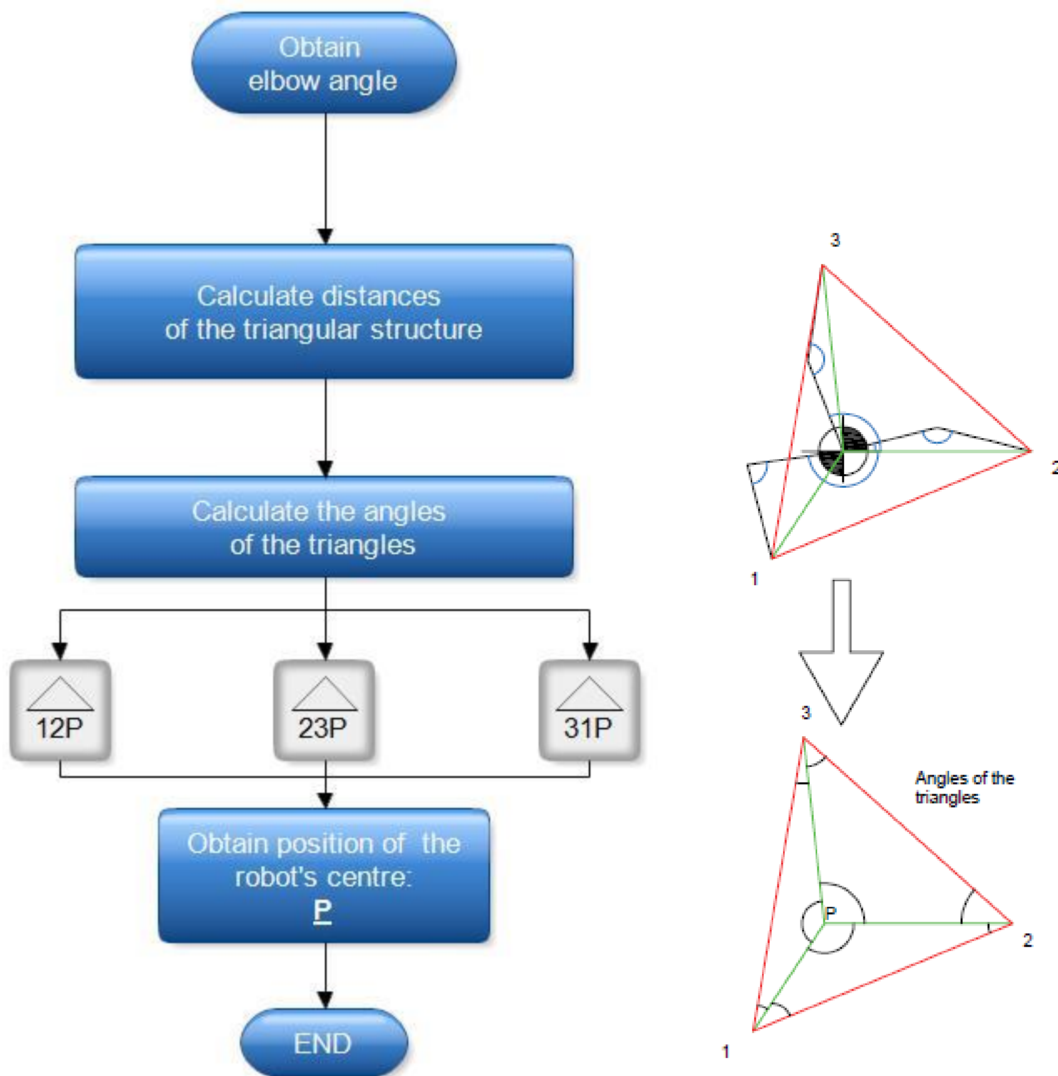

Figure 9: Diagram Part 3

Part 4 - Representation of the robot's configuration

The fourth part is divided into 2 parts:

The first part is focused on making a graphic of the allocation of the holes on the wall, or in other words, printing the map of holes. Although this piece of algorithm belong to the fourth part, it is going to be run at the beginning, inside of the first part, with the aim of checking that everything is well configured before running the rest of the program (it is possible to insert one break point the first times to check the map of holes is OK).

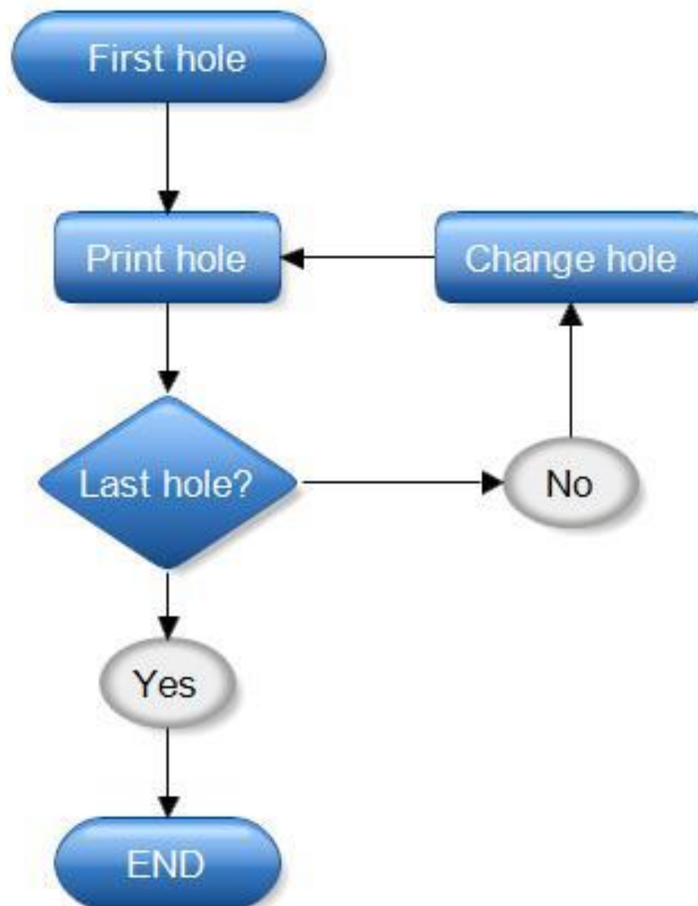

Figure 10: Diagram Part 4.1

The second part has been created with the aim of printing the real configuration of the robot. For that, the only information available is the 4 points of the robot (3 hands and the centre). So, the others points (elbows) will be obtained with equations and trigonometry.

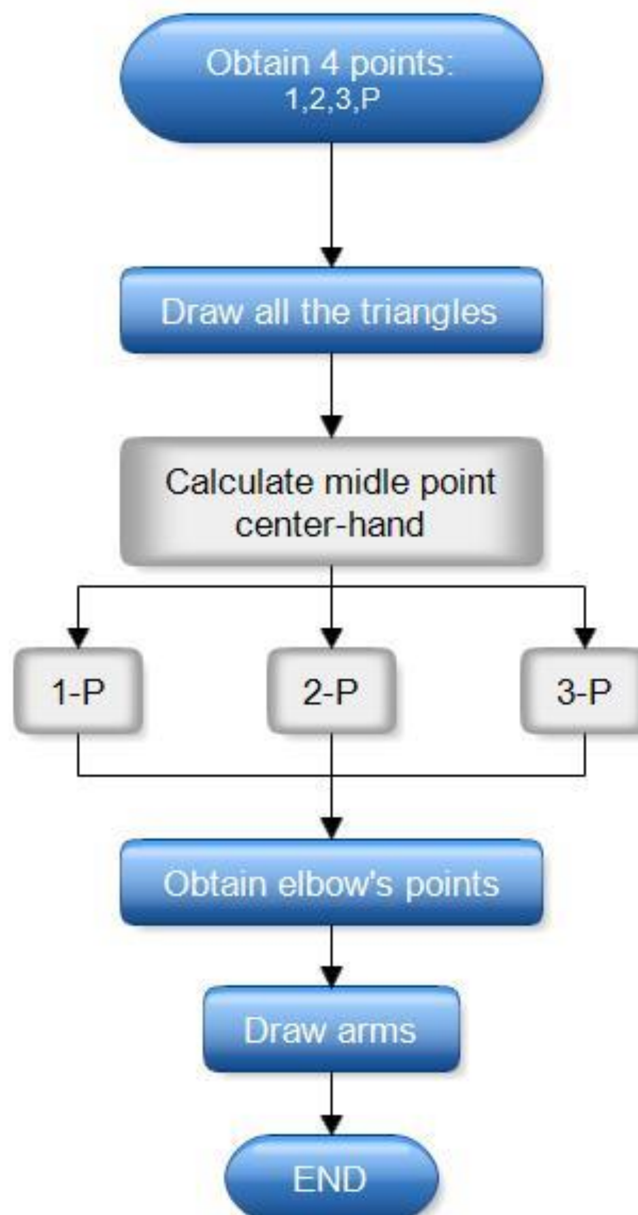

Figure 11: Diagram Part 4.2

Auxiliary program

The auxiliary program is using the points set up by the user and a point clicked in the screen to calculate the centre's position and the angles of all the articulations of the robot.

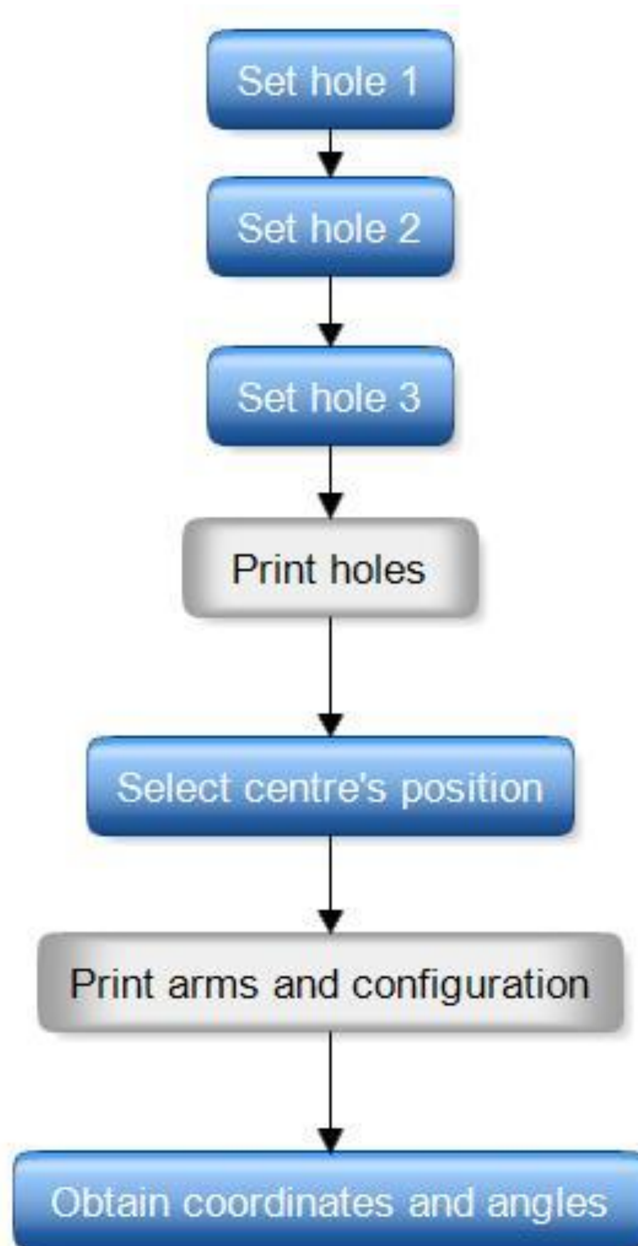

Figure 12: Diagram auxiliary program

5. Software description

This is the main chapter of the project in which the functions and basis of the program are explained in detail and the code is analysed.

5.1 Documentation of algorithm in detail

In this part of the chapter we are going to analyse the code in detail and we are commenting every function to explain how it works.

5.1.1 Relative distances' calculations – Part 1

```
1       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2       %%%%%%%%%%%%%%%%%%%%%% program starts, localization %%%%%%%%%%
3       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4
5       %% info provided by the robot %%%
6
7       % the first angle is the shoulder
8       % the second is the elbow of each arm
9  -    angle1=[(-24)*pi/180,(85)*pi/180];%arm1
10 -    angle2=[(78)*pi/180,(55)*pi/180];%arm2
11 -    angle3=[(-155)*pi/180,(103)*pi/180];%arm3
12 -    l=10;%lenght of the links
13 -    precision=0.1;
14
15      %% Part 1 - relative distances' calculations %%%%
16      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
17      %%%% distance centre-hand%%%%
18
19      %arm1
20 -    rdx1=+l*cos(angle1(1))+l*cos(angle1(1)-angle1(2)+pi);
21 -    rdy1=+l*sin(angle1(1))+l*sin(angle1(1)-angle1(2)+pi);
22
23      %arm2
24 -    rdx2=+l*cos(angle2(1))+l*cos(angle2(1)-angle2(2)+pi);
25 -    rdy2=+l*sin(angle2(1))+l*sin(angle2(1)-angle2(2)+pi);
26
27      %arm3
28 -    rdx3=+l*cos(angle3(1))+l*cos(angle3(1)-angle3(2)+pi);
29 -    rdy3=+l*sin(angle3(1))+l*sin(angle3(1)-angle3(2)+pi);
30
```

Figure 13: program code 1

Between lines 5 and 14 is the part dedicated to the inputs, so here we will declare the angles of the robot provided by the sensors of the arms.

We will set up the inputs in vectors called “*angle 1, 2 and 3”* in which the first term will be the angle of the first link measured from the horizontal referenced like 0°, so if the link is totally vertical the angle will be 90°, totally horizontal pointing to the left 180° and totally vertical pointing down 270° or -90°. The factor *“*pi/180”* is used to calculate the angles in radians because the program will work in radians but always the results will be shown in degrees to be more intuitive, so this factor will appear several times in the program.

The term *“l”* (in this case initialised as 10) is the length of the robot’s links, it is variable because we can use the program for different versions of the robot and this is only a prototype. The program is prepared to work with centimetres but is applicable to any other measures.

The term called *“precision”* will be explained later when the program will use it. In our case is initiated like 0.1.

In lines 16 – 30 the program calculate the relative coordinates of the final points with the centre as reference (0, 0). So we are obtaining the relative distance of the hands “*rdx”* and “*rdy”*.

With this information, in lines 31-34 the program calculates the distances between the hands of the robot in the vector *“h”* (of height).

```
31        % distnaces between hands or holes
32 -      h(1)=sqrt((rdx1-rdx2)^2+(rdy1-rdy2)^2);
33 -      h(2)=sqrt((rdx3-rdx2)^2+(rdy3-rdy2)^2);
34 -      h(3)=sqrt((rdx1-rdx3)^2+(rdy1-rdy3)^2);
```

Figure 14: program code 2

All the equations and the terms previously explained are represented in the next picture:

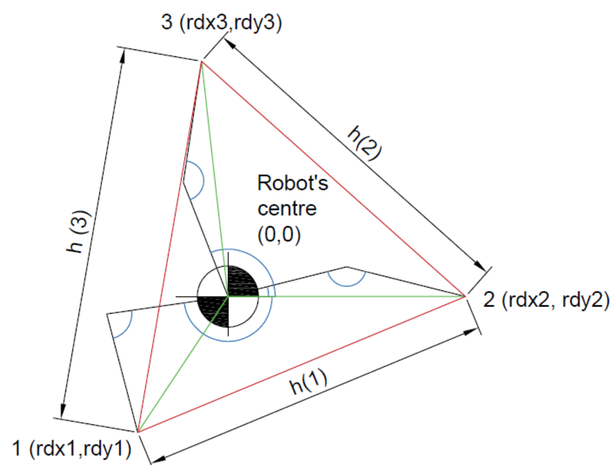

Figure 15: distances’ disposition

Now it's time to declare the map of wholes that we are using in the program. For the collection and storage of this information, a matrix is going to be used. In this matrix called *"whole"* (whole map of holes), we will put the coordinates of all the points of the holes using complex numbers, the real part of the number will be the X coordinate, the imaginary part will be the Y coordinate. If we work with this method the calculations will be simplified because the absolute value of the complex number will be the real distance between 2 points.

```
37
38        %%%%%%%% hole's map%%%%%%%%%
39 -      whole=[1,0,0,0;0,1,0,0;0,0,1,0;0,0,0,1];
40 -      whole(1,1)=0+0i;
41 -      whole(2,1)=20+0.5i;
42 -      whole(3,1)=40+0.00000001i;
43 -      whole(4,1)=59.5+0.3i;
44 -      whole(1,2)=20.3i;
45 -      whole(2,2)=19.4+19.4i;
46 -      whole(3,2)=40.2+20i;
47 -      whole(4,2)=60+20.1i;
48 -      whole(1,3)=0.2+39.8i;
49 -      whole(2,3)=20.5+40.5i;
50 -      whole(3,3)=40+40i;
51 -      whole(4,3)=59.5+39.5i;
52 -      whole(1,4)=60i;
53 -      whole(2,4)=19.8+59.8i;
54 -      whole(3,4)=40+60i;
55 -      whole(4,4)=60.3+60.3i;
56 -      whole
57 -      s=size(whole);
58 -      sx=s(1);
59 -      sy=s(2);
60
```

Figure 16: program code 3

The program will show the matrix after has been completed in line 56.

```
whole =

         0                    0 +20.3000i    0.2000 +39.8000i         0 +60.0000i
  20.0000 + 0.5000i   19.4000 +19.4000i   20.5000 +40.5000i   19.8000 +59.8000i
  40.0000 + 0.0000i   40.2000 +20.0000i   40.0000 +40.0000i   40.0000 +60.0000i
  59.5000 + 0.3000i   60.0000 +20.1000i   59.5000 +39.5000i   60.3000 +60.3000i
```

Figure 17: console – result 1

At the lines 57 – 59 the function size obtain the size of the map, saving the number of coloms in *"sx"* and the number of lines in *"sy"* (size of the matrix or amount of holes).

From now on, if we need to use the position of a hole in the program we will refer to it like whole(nº of colom, nº of line), being the first hole (the one of the botton-left corner) the *"whole(1,1)"* and the last (the one of the oposit corner) the *"whole(sx,sy)"*, in this case *"whole(4,4)"*.

5.1.2 Representation of the map of holes – Part 4.1

Arrived to this point the part 2 is supposed to start but first, the program is running the part 4.1 which will show us a chart of the holes' allocation and this will give us the possibility of checking if we have configured the map correctly to avoid possible errors before the program continue. The first time is advisable to insert a break point after this part to see the chart and verify that is OK before run the whole program.

```
61
62 -     hold on
63 -     for j=1:1:sy
64 -         for i=1:1:sx
65 -             plot(whole(i,j),'p')
66 -         end
67 -     end
```

Figure 18: program code 4

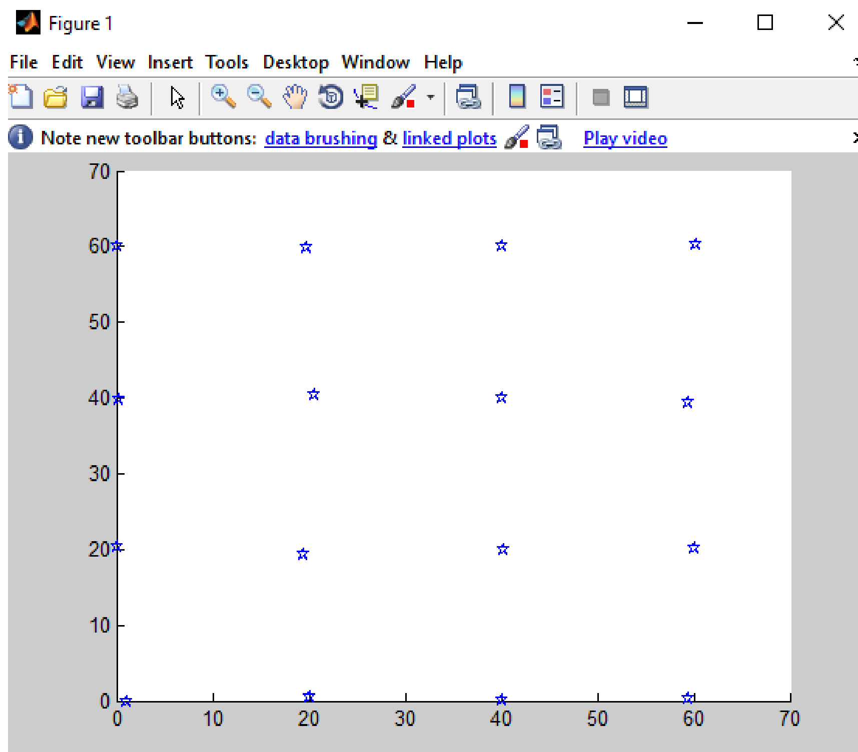

Figure 19: console – results 2

5.1.3 Extra checking function

```
70
71      %%% info of the robot%%%%
72      %%%theoretical way to get the distances
73 -    ht(3)=abs(whole(4,3)-whole(3,2));
74 -    ht(2)=abs(whole(3,3)-whole(4,3));
75 -    ht(1)=abs(whole(3,2)-whole(3,3));
76
```

Figure 20: program code 5

Before starting Part 2, in lines 70 – 76 the program creates a vector called *"ht"* (theoretical height), the terms of this vector are the theoretical distance between the holes that we want to select (in the example are the holes (4,3), (3,2) and (3,3)). This function is really useful for checking if the program are calculating the distances properly.

If, for example, we have the robot situated in this holes the program is going to calculate the distances between holes in the vector *"h"*, the program can make little mistakes and introduce errors because of the accuracy of the sensors, so the perfect or theoretical measurement will be the *"ht"* vector, so then it is possible to compare with our result *"h"* to detect how much error we have.

5.1.4 Distance's comparison and hole's search – Part 2

Now it is time to explain and run the part 2. First of all the program initialise the variables that are going to be used.

```
76
77      %% Part 2 - Distance's comparison and hole's search %%
78      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
79      %%%%%%%%%%% compare distances%%%%%%%%%%%%%%%
80      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
81 -    look=1;
82 -    x=1; %coord. of the point we are going to compare
83 -    y=1;
84 -    xr=1; %coord. of the reference point
85 -    yr=1;
86 -    p1=[0,0]; %coord of the solution
87 -    p2=[0,0];
88 -    p3=[0,0];
89
90
```

Figure 21: program code 6

In this sub algorithm the program is going to calculate and compare the distance between two holes, one hole will be the main point and is going to be affixed, *"x"* and *"y"* are the coordinates of this point (column and line), and *"xr"* and *"yr"* are the coordinates of the reference point which is going to be used to compare and obtain the distances, this point will change, because with one main point, the program is going to use 4 different reference points to compare.

The vectors “p1, p2 and p3” are the vectors in which the program will save the coordinates of the 3 solution points.

```
91
92
93 -  for j=1:1:sy
94 -      for i=1:1:sx
95 -          for k=1:1:4
96
97 -              x=i;
98 -              y=j;
99
```

Figure 22: program code 7

For comparing the distances the program is using *“for”* loops, the first *“for j=1:1:sy”* and the *“for i=1:1:sx”* are going to be used for selecting he main hole and along this part of algorithm, the program is using all the points as main point. Is because of that, the loops ends with the last points of the matrix *“whole”*, which are *“sx”* and *“sy”*.

The loop *“for k=1:1:4”* is going to determine which comparison is going to do the program (watch figure 24).

```
100 -              if k==1
101 -                  if y~=4
102 -                  xr=x;
103 -                  yr=y+1;
104 -                  end
105 -              end
106
107 -              if k==2
108 -                  if (x~=4 & y~=4)
109 -                  xr=x+1;
110 -                  yr=y+1;
111 -                  end
112 -              end
113
114 -              if k==3
115 -                  if x~=4
116 -                  xr=x+1;
117 -                  yr=y;
118 -                  end
119 -              end
120
121 -              if k==4
122 -                  if (x~=1 & y~=4)
123 -                  xr=x-1;
124 -                  yr=y+1;
125 -                  end
126 -              end
127
```

Figure 23: program code 8

When k=1 the reference point which is going to be compared with the main point to obtaining the distance will be the adjacent hole of the right, with k=2 the hole will be the one adjacent in diagonal of the top right corner, with k=3 the one of the top and with k=4 will be the diagonal of the top left corner.

The next image shows how the program is comparing distances and how the changes of the loops' variables are going to affect in the search:

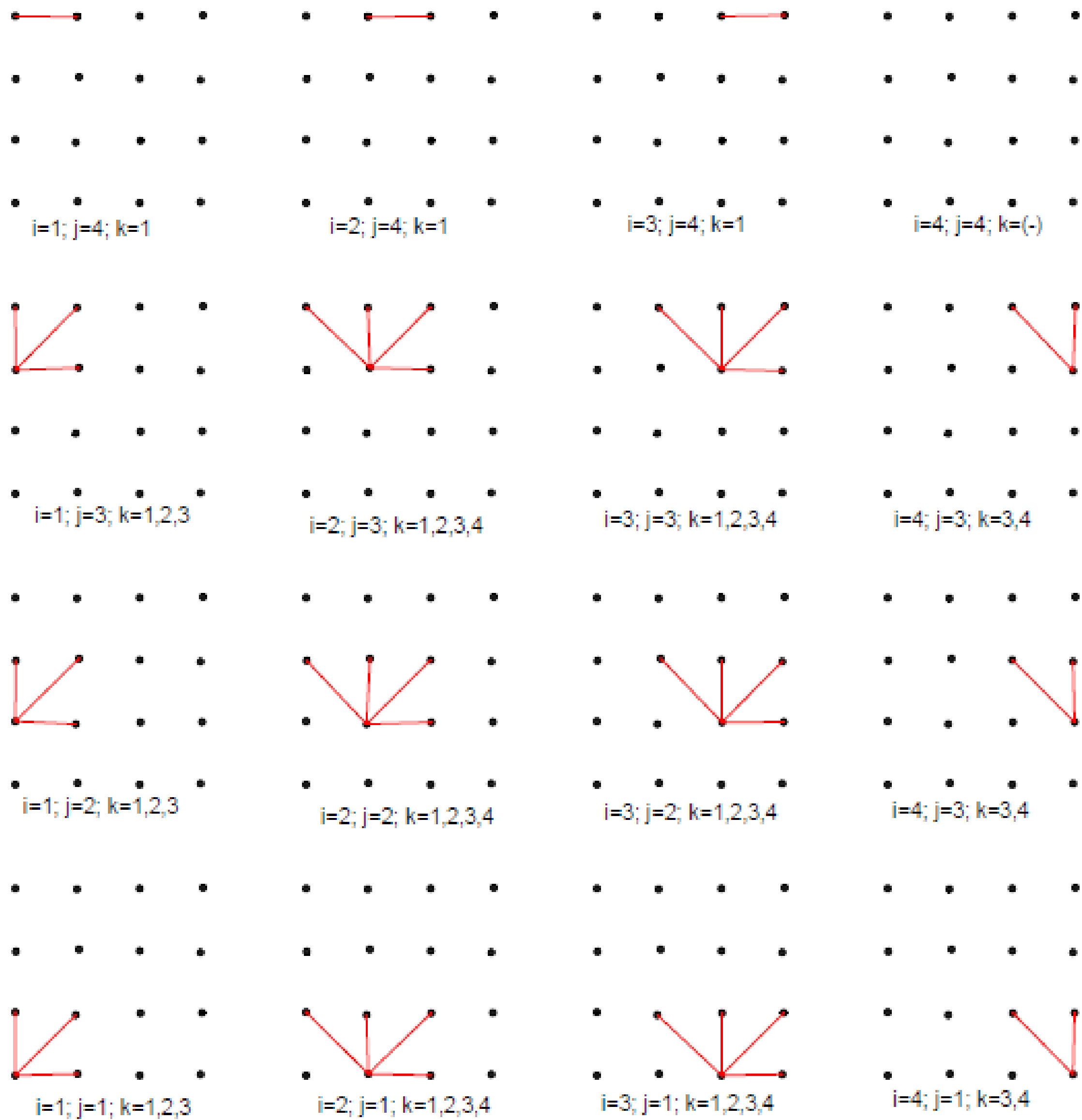

Figure 24: Comparison of holes with the variables of the loops "for"

Now the program is starting to compare distances

```
128
129
130 -        dist1=abs(whole(x, y)-whole(xr, yr));
131
132 -        if abs(dist1-h(1))<precision %%first distance found
133              %we should find the others 2 distances
134 -            p1(1)=x;
135 -            p1(2)=y;
136 -            p2(1)=xr;
137 -            p2(2)=yr;
138
139
```

Figure 25: program code 9

"dist1" is the difference between the distance we have and the distance we are looking for. This comparison is going to consider that a distance is close enough to be OK when the result of *"dist1"* is within the range established by the variable *"precision"*. So changing this variable, we can change the desired accuracy for the program. Sometimes due to errors perhaps the program is not going to find any solution and then we may change the value of *"precision"* for another higher to be not so exigent or if the program finds two possible solution the value may be reduce to discarded one option.

If the program finds the distance searched is going to consider that two holes as a solution so they will be saved in vectors *"p1"* and *"p2"*.

The next part of the program is really complex so let's going to try to explain it in the most simple and schematic way.

LOOP 1: the program will only run if the distance found is vertical, between the main point and the reference point on the top (line 141). We must compare all the possibilities to find the correct point.

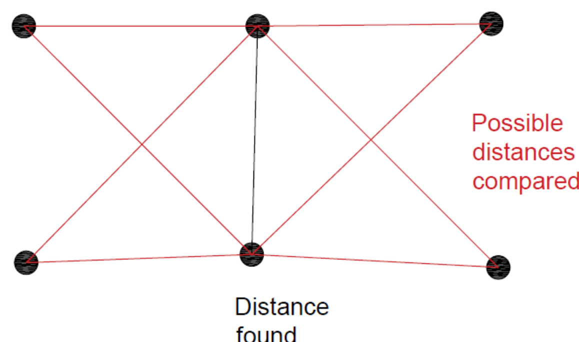

Figure 26: possible distances – vertical found

Line 143: the program only run if the points are not on the right border (*x=4*, if the matrix is different we must put *"sx").*

Line 144-182: the program is comparing the two distances with the holes from the right of the initial distance found.

Line 144: establish *"dist2"* like distance between one found hole *(xr, yr)* and one possible hole *(x+1, y).*

Line 146: establish *"dist3"* like distance between the possible hole *(x+1, y)* and the other found hole *(x, y).*

Line 148: if *dist2=h2* and *dist3=h3,* solution found.

Line 154: if *dist2=h3* and *dist3=h2,* solution found.

This process is going to be repeated constantly with other configurations and possible holes but the principles are going to be the same.

If the distances are found the solution will be saved in *"p3".*

```
140
141 -   if(x==xr)%%vertical
142
143 -     if x~=4
144 -       dist2=abs(whole((x+1), y)-whole(xr, yr));%vertical- right down
145 -       if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
146 -           dist3=abs(whole((x+1), y)-whole(x, y));
147
148 -        if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
149 -           p3(1)=x+1;
150 -           p3(2)=y;
151 -           look=0;
152 -        end
153
154 -        if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
155 -           p3(1)=x+1;
156 -           p3(2)=y;
157 -           look=0;
158 -        end
159
```

Figure 27: program code 10

```
160
161
162 -        end
163
164 -         dist2=abs(whole(x, y)-whole((xr+1), yr));%vertical- right up
165 -         if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
166 -             dist3=abs(whole(xr,yr)-whole((xr+1), yr));
167
168 -             if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
169 -            p3(1)=xr+1;
170 -            p3(2)=yr;
171 -            look=0;
172 -             end
173
174 -            if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
175 -            p3(1)=xr+1;
176 -            p3(2)=yr;
177 -            look=0;
178 -             end
179
180 -         end
181 -      end
182
```

Figure 28: program code 11

Now the program is going to compare the distances with the left side so in line 184 the program won't continue if the holes are on the left border because there aren't more holes and this will give us an error. The process is the same as the previous part of code.

```
184 -      if x~=1
185 -         dist2=abs(whole((x-1), y)-whole(xr, yr));%vertical-left down
186 -         if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
187 -             dist3=abs(whole((x-1), y)-whole(x,y));
188
189 -            if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
190 -            p3(1)=x-1;
191 -            p3(2)=y;
192 -            look=0;
193 -            end
194 -           if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
195 -            p3(1)=x-1;
196 -            p3(2)=y;
197 -            look=0;
198 -           end
199
200 -          dist2=abs(whole(x, y)-whole((xr-1), yr));%vertical- left up
201 -          if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
202 -              dist3=abs(whole(xr, yr)-whole((xr-1), yr));
203
204 -            if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
205 -            p3(1)=xr-1;
206 -            p3(2)=yr;
207 -            look=0;
208 -            end
209 -            if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
210 -            p3(1)=xr-1;
211 -            p3(2)=yr;
212 -            look=0;
213 -            end
214
```

Figure 29: program code 12

END OF LOOP 1

LOOP 2: the program will only run this part if the distance found between the main hole and the reference hole is horizontal (line 223). In this case, the 3rd hole could be on the top or under the found holes. There are 4 possible holes and we are going to analyse the 4 options with the same method as the loop 1.

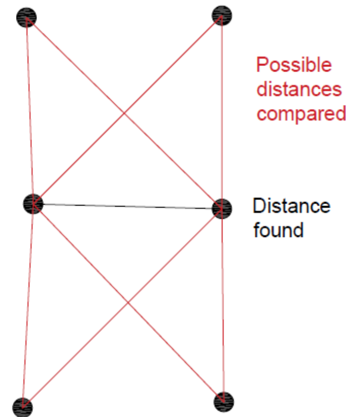

Figure 30: possible distances – horizontal found

```
223 -     if(y==yr)%horizonal
224
225 -      if y~=4
226 -       dist2=abs(whole(x, (y+1))-whole(xr, yr));%horizontal- up left
227 -       if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
228 -           dist3=abs(whole(x, (y+1))-whole(x, y));
229
230 -           if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
231 -           p3(1)=x;
232 -           p3(2)=y+1;
233 -           look=0;
234 -           end
235 -           if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
236 -           p3(1)=x;
237 -           p3(2)=y+1;
238 -           look=0;
239 -           end
240 -       end
```

Figure 31: program code 13

```
241
242 -      dist2=abs(whole(x, y)-whole(xr, (yr+1)));%horizontal- up right
243 -      if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
244 -          dist3=abs(whole(xr, yr)-whole(xr, (yr+1)));
245
246 -        if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
247 -         p3(1)=xr;
248 -         p3(2)=yr+1;
249 -         look=0;
250 -        end
251 -        if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
252 -         p3(1)=xr;
253 -         p3(2)=yr+1;
254 -         look=0;
255 -        end
256 -      end
257 -    end
258
```

Figure 32: program code 14

```
260 -  if y~=1
261 -   dist2=abs(whole(x, (y-1))-whole(xr, yr));%horizontal- down left
262 -   if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
263 -       dist3=abs(whole(x, (y-1))-whole(x, y));
264
265 -       if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
266 -       p3(1)=x;
267 -       p3(2)=y-1;
268 -       look=0;
269 -       end
270 -       if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
271 -       p3(1)=x;
272 -       p3(2)=y-1;
273 -       look=0;
274 -       end
275 -   end
```

Figure 33: program code 15

```
277 -     dist2=abs(whole(x, y)-whole(xr, (yr-1)));%horizontal- down right
278 -     if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
279 -         dist3=abs(whole(xr, yr)-whole(xr, (yr-1)));
280
281 -       if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
282 -       p3(1)=xr;
283 -       p3(2)=yr-1;
284 -       look=0;
285 -       end
286 -       if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
287 -       p3(1)=xr;
288 -       p3(2)=yr-1;
289 -       look=0;
290 -       end
291 -     end
292 -   end
293
294 - end%y==yr
```

Figure 34: program code 16

END OF LOOP 2

LOOP 3: this loop is designed for those cases in which the found distance is diagonal and the 3rd hole can be only on the right or the left of the distance (structure of a square).

In line 299 the program is analysing one possible diagonal and in the line 334 the other.

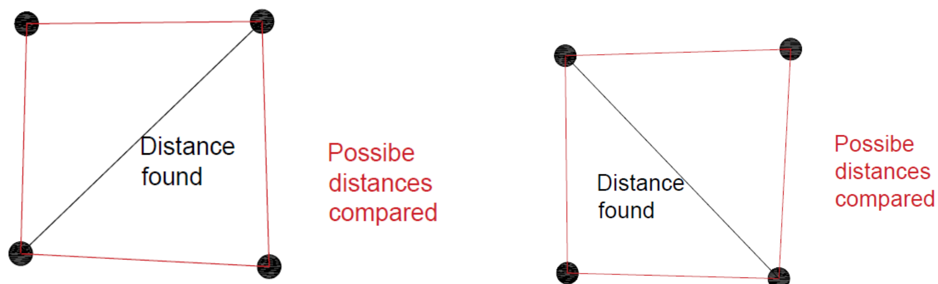

Figure 35: possible distances – diagonal found

Once again the method of comparison is the same.

```
297 -  f((x~=xr)&(y~=yr))%%diagonal
298
299 -      if( (x<xr)&(y<yr) )%diag. up right1
300 -          dist2=abs(whole((x+1), y)-whole(xr, yr));%hole down
301 -          if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
302 -              dist3=abs(whole((x+1), y)-whole(x, y));
303
304 -              if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
305 -              p3(1)=x+1;
306 -              p3(2)=y;
307 -              look=0;
308 -              end
309 -              if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
310 -              p3(1)=x+1;
311 -              p3(2)=y;
312 -              look=0;
313 -              end
314 -          end
315
```

Figure 36: program code 17

```
316 -          dist2=abs(whole(x, y)-whole((xr-1), yr));%hole up
317 -          if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
318 -              dist3=abs(whole(xr, yr)-whole((xr-1), yr));
319
320 -              if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
321 -              p3(1)=xr-1;
322 -              p3(2)=yr;
323 -              look=0;
324 -              end
325 -              if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
326 -              p3(1)=xr-1;
327 -              p3(2)=yr;
328 -              look=0;
329 -              end
330 -          end
331 -      end
332
```

Figure 37: program code 18

```
333
334 -       if(  (x>xr)&(y<yr)  )%diag. up left2
335 -          dist2=abs(whole((x-1), y)-whole(xr, yr));%hole down
336 -          if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
337 -              dist3=abs(whole((x-1), y)-whole(x, y));
338
339 -              if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
340 -              p3(1)=x-1;
341 -              p3(2)=y;
342 -              look=0;
343 -              end
344 -              if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
345 -              p3(1)=x-1;
346 -              p3(2)=y;
347 -              look=0;
348 -              end
349 -          end
350
```

Figure 38: program code 19

```
350
351 -          dist2=abs(whole(x, y)-whole((xr+1), yr));%hole up
352 -          if (abs(dist2-h(2))<precision)|(abs(dist2-h(3))<precision)
353 -              dist3=abs(whole(xr, yr)-whole((xr+1), yr));
354
355 -              if (abs(dist2-h(2))<precision)&(abs(dist3-h(3))<precision)
356 -              p3(1)=xr+1;
357 -              p3(2)=yr;
358 -              look=0;
359 -              end
360 -              if (abs(dist2-h(3))<precision)&(abs(dist3-h(2))<precision)
361 -              p3(1)=xr+1;
362 -              p3(2)=yr;
363 -              look=0;
364 -              end
365 -          end
366 -      end
```

Figure 39: program code 20

END OF LOOP 3

Univerza v Mariboru

```
369
370 -         end% (x<>xr) & (y<>yr)
371
372
373 -     end
374
375 -     if look ==0
376 -         disp('solution found')
377 -         cord1=p1
378 -         cord2=p2
379 -         cord3=p3
380 -         look=1;
381 -     end
382
383 -     end
384 -         end
385 -
```

Figure 40: program code 21

When the program finds the right solution, the 3rd hole is saved in *"p3"* and the variable *"look"* is changed to the value 0.

In line 375 if *look=0* the program is going to display the message *"solution found"* and is going to show and safe the coordinates of the 3 holes in which the robot is affixed.

```
solution found

cord1 =

     2     3


cord2 =

     3     3


cord3 =

     3     2
```

Figure 41: console – result 3

In the line 380 look will be again 1 to continue the program and search for another possible solution. If the *"precision"* value is good enough we are obtaining only one possible solution.

5.1.5 Calculation of a centre's position – Part 3

For avoiding errors and making all the parts of the program independent of each other's the distances of the robot will be calculated again, but in this case the program will only use the elbow angle (line 397-401).

```
390
391      %% Part 3 - Calculate centre's position.%%
392      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
393      %%%%%%%%%%%%%%%%%%%%% start triangulation program %%%%%%%%%%%%%%%%%%%%
394      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
395
396
397      %%%information of the robot%%%
398      %we use only the angle of the elbow
399 -    angle1=angle1(2);
400 -    angle2=angle2(2);
401 -    angle3=angle3(2);
402
403      % change of reference -> now p1=0,0; p2=p2-p1; p3=p3-p1.
404 -    p1=[real(whole(cord1(1),cord1(2))-whole(cord1(1),cord1(2))),imag(whole(cord1(1),cord1(2))-whole(cord1(1),cord1(2)))];
405 -    p2=[real(whole(cord2(1),cord2(2))-whole(cord1(1),cord1(2))),imag(whole(cord2(1),cord2(2))-whole(cord1(1),cord1(2)))];
406 -    p3=[real(whole(cord3(1),cord3(2))-whole(cord1(1),cord1(2))),imag(whole(cord3(1),cord3(2))-whole(cord1(1),cord1(2)))];
407      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
408
409
410
411      %%%variables we will use%%%%%%%%
412 -    long=[0,0,0];
413 -    d=[0,0,0];
414 -    x=[0,0,0];
415 -    t1=[0,0,0];
416 -    t2=[0,0,0];
417 -    t3=[0,0,0];
418 -    p=[0,0];
419      %%%%%%%%%%%%%%%%%%%
420
```

Figure 42: program code 22

For the future calculations, working with small numbers is making easier the equations and it will help us minimising the possible errors. Is because of that, we are changing the reference, using one of the holes (*"p1"*) as origin (0, 0).

```
423      %%%distances %%%%%%%%%%%%%%%%%
424
425      %distances between center and holes
426 -    long(1)=sqrt((1+1*cos(pi-angle1))^2+(1*sin(pi-angle1))^2);%centre-hole1
427 -    long(2)=sqrt((1+1*cos(pi-angle2))^2+(1*sin(pi-angle2))^2);%centre-hole2
428 -    long(3)=sqrt((1+1*cos(pi-angle3))^2+(1*sin(pi-angle3))^2);%centre-hole3
429
430      % distances between holes
431 -    d(1)=sqrt((p1(1)-p2(1))^2+(p1(2)-p2(2))^2);%hole1-hole2
432 -    d(2)=sqrt((p2(1)-p3(1))^2+(p2(2)-p3(2))^2);%hole2-hole3
433 -    d(3)=sqrt((p1(1)-p3(1))^2+(p1(2)-p3(2))^2);%hole1-hole3
434      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 43: program code 23

```
436
437     %%%angles%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
438     %hole1 -> 1
439     %hole2 -> 2
440     %hole3 -> 3
441     %centre -> P
442
443     %angles of the triangle 1-2-P
444 -   t1(1)=(180/pi)*acos((d(1)^2+long(1)^2-long(2)^2)/(2*d(1)*long(1)));
445 -   t1(2)=(180/pi)*acos((d(1)^2-long(1)^2+long(2)^2)/(2*d(1)*long(2)));
446 -   t1(3)=(180/pi)*acos((-d(1)^2+long(1)^2+long(2)^2)/(2*long(2)*long(1)));
447
448     %angles of the triangle 2-3-P
449 -   t2(1)=(180/pi)*acos((d(2)^2+long(2)^2-long(3)^2)/(2*d(2)*long(2)));
450 -   t2(2)=(180/pi)*acos((d(2)^2-long(2)^2+long(3)^2)/(2*d(2)*long(3)));
451 -   t2(3)=(180/pi)*acos((-d(2)^2+long(2)^2+long(3)^2)/(2*long(2)*long(3)));
452
453     %angles of the triangle 3-1-P
454 -   t3(1)=(180/pi)*acos((d(3)^2+long(3)^2-long(1)^2)/(2*d(3)*long(3)));
455 -   t3(2)=(180/pi)*acos((d(3)^2-long(3)^2+long(1)^2)/(2*d(3)*long(1)));
456 -   t3(3)=(180/pi)*acos((-d(3)^2+long(3)^2+long(1)^2)/(2*long(3)*long(1)));
457     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
458
```

Figure 44: program code 24

Between lines 423 and 458 the program is calculating the robot's distances and the angles of the theoretical triangles formed by the holes and the centre's position.

```
459
460
461     %%%center's position
462 -   alpha=atan2((p2(2)-p1(2)),(p2(1)-p1(1)));
463     %relative position (hole1 is the reference; distance hole1-centre)
464 -   p(1)=long(2)*cos(alpha-(pi*t1(2)/180));
465 -   p(2)=long(2)*sin(alpha-(pi*t1(2)/180));
466 -   p
467     %absolute position ((0,0) is the reference; absolute reference)
468 -   robpos(1)=real(whole(cord1(1),cord1(2)))+p(1);
469 -   robpos(2)=imag(whole(cord1(1),cord1(2)))+p(2);
470 -   robpos
471
```

Figure 45: program code 25

In line 462 the angle *"alpha"* is the angle of the distance between the centre and the centre.

"p" is the relative position of the centre (origin set in the hole p1).

"robpos" is the absolute position of the robot's centre.

```
p =

   14.5066  -11.7553


robpos =

   35.0066   28.7447
```

Figure 46: console – result 4

5.1.6 Representation of the robot's configuration – Part4.2

Lines 477-483: change to an absolute reference.

```
473       %% Part 4 - Representation of the robot's configuration
474       %%%%%representation
475 -     figure
476
477       %change again reference to absolute (0,0)
478       %for drawing correct coord.
479 -     p1=[real(whole(cord1(1),cord1(2))),imag(whole(cord1(1),cord1(2)))];
480 -     p2=[real(whole(cord2(1),cord2(2))),imag(whole(cord2(1),cord2(2)))];
481 -     p3=[real(whole(cord3(1),cord3(2))),imag(whole(cord3(1),cord3(2)))];
482 -     p=robpos;
483
484       %draw triangles of the body
485 -     a=[p1(1) p2(1) p3(1) p1(1)];
486 -     b=[p1(2) p2(2) p3(2) p1(2)];
487 -     plot(a, b);
488 -     hold on;
489 -     plot(p(1), p(2), 'p');
490 -     c1=[p2(1) p(1) p1(1)];
491 -     c2=[p2(2) p(2) p1(2)];
492 -     plot(c1, c2);
493 -     c3=[p(1) p3(1)];
494 -     c4=[p(2) p3(2)];
495 -     plot(c3, c4);
496
```

Figure 47: program code 26

The first step to print the configuration of the robot is to print the triangles that we have used in the previous calculus of the other parts. The vectors *"a"* and *"b"* are used to form and print the triangle of the holes and *"c1","c2","c3"and "c4"* are used for the triangles formed with the centre.

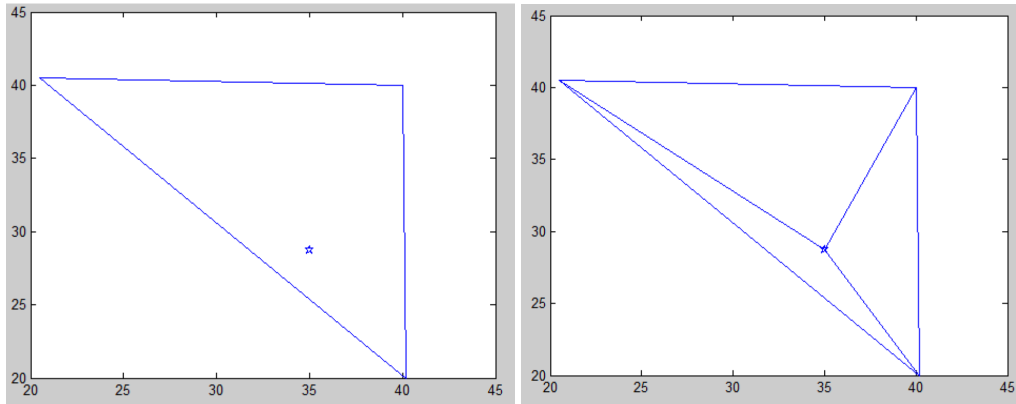

Figure 48: console – result 5

```
499       %%%%leg 1%%%%%
500 -     loop=1;
501 -     mp1=(p+p1)/2;%midle point
502       %plot(mp1(1), mp1(2), 'p');
503 -     epsilon1=atan2((p(2)-p1(2)),(p(1)-p1(1)))-pi/2;%angle of the line
504       %between the midle point and the elbow
505
506 -     for i=1:1:3
507 -         if loop==1
508 -     beta1=asin(long(i)/(2*l));%1/2 of the angle of the elbow
509 -     gama1=pi/2-beta1;%angle of the shoulder with the line to the hole
510 -     h1=l*sin(gama1);
511 -     fp1(1)=mp1(1)-h1*cos(epsilon1);
512 -     fp1(2)=mp1(2)-h1*sin(epsilon1);
513
514 -     check1=fp1-p1;
515 -     sol1=sqrt(check1(1)^2+check1(2)^2);
516 -     check1=fp1-p;
517 -     sol2=sqrt(check1(1)^2+check1(2)^2);
518
519 -             if (abs(sol1-l)<0.1)&(abs(sol2-l)<0.1)
520 -                 c5=[p1(1) fp1(1) p(1)];
521 -                 c6=[p1(2) fp1(2) p(2)];
522 -                 plot(c5, c6,'g');
523 -                 loop=0;
524 -             end
525 -         end
526 -     end
527
528
```

Figure 49: program code 27

The structure showed in lines 499-526 is going to be used 3 times, once per arm, so we will only explain it once and the others 2, the program will repeat the same process but with the variables of the others arms.

To print the arm we have the centre and the final point of the hole but the point of the elbow is needed. With this aim the program first is finding the middle point between the hole and the centre, as we know the two links has the same length so the elbow must be in the perpendicular line that passes through this point, and obtaining the angles of the arm, is easy to get the elbow's point. The loop "*for*" is only because the program doesn't know which angles are in which arms and it must be checked. The variables *"solX"* is the practical length of the link. If the program doesn't commit any error this value theoretically must be the one that we have set up in the variable *"l"* at the beginning (in our case 10).

```
532        %%%%leg 2%%%%%
533 -      loop=1;
534 -      mp2=(p+p2)/2;
535        %plot(mp3(1), mp3(2), 'p');
536 -      epsilon2=atan2((p(2)-p2(2)),(p(1)-p2(1)))-pi/2;
537
538 -    for i=1:1:3
539 -          if loop==1
540 -      beta2=asin(long(i)/(2*l));
541 -      gama2=pi/2-beta2;
542 -      h2=l*sin(gama2);
543 -      fp2(1)=mp2(1)-h2*cos(epsilon2);
544 -      fp2(2)=mp2(2)-h2*sin(epsilon2);
545
546 -      check2=fp2-p2;
547 -      sol3=sqrt(check2(1)^2+check2(2)^2);
548 -      check2=fp2-p;
549 -      sol4=sqrt(check2(1)^2+check2(2)^2);
550
551 -          if (abs(sol3-l)<0.1)&(abs(sol4-l)<0.1)
552 -          c7=[p2(1) fp2(1) p(1)];
553 -          c8=[p2(2) fp2(2) p(2)];
554 -          plot(c7, c8,'g');
555 -          loop=0;
556 -          end
557
558
559 -          end
560 -      end
561
562
```

Figure 50: program code 28

Univerza v Mariboru

```
562
563      %%%%leg 3%%%%%
564 -    loop=1;
565 -    mp3=(p+p3)/2;
566      %plot(mp3(1), mp3(2), 'p');
567 -    epsilon3=atan2((p(2)-p3(2)),(p(1)-p3(1)))-pi/2;
568
569 -    for i=1:1:3
570 -        if loop==1
571 -    beta3=asin(long(i)/(2*l));
572 -    gama3=pi/2-beta3;
573 -    h3=l*sin(gama3);
574 -    fp3(1)=mp3(1)-h3*cos(epsilon3);
575 -    fp3(2)=mp3(2)-h3*sin(epsilon3);
576
577 -    check3=fp3-p3;
578 -    sol5=sqrt(check3(1)^2+check3(2)^2);
579 -    check3=fp3-p;
580 -    sol6=sqrt(check3(1)^2+check3(2)^2);
581
582 -            if (abs(sol5-l)<0.1)&(abs(sol6-l)<0.1)
583 -            c9=[p3(1) fp3(1) p(1)];
584 -            c10=[p3(2) fp3(2) p(2)];
585 -            plot(c9, c10,'g');
586 -            loop=0;
587 -            end
588 -        end
589 -    end
590
591
```

Figure 51: program code 29

RESULT:

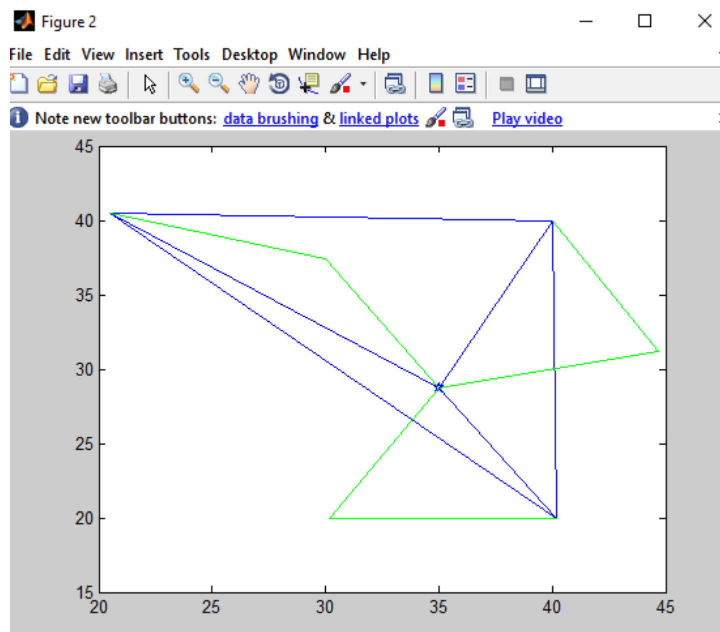

Figure 52: console – result 6

When we run the same program twice, the chart of the part 4.1 and the chart of the part 4.2 will be superimposed. As result of that, we can see the global situation of the robot in the wall.

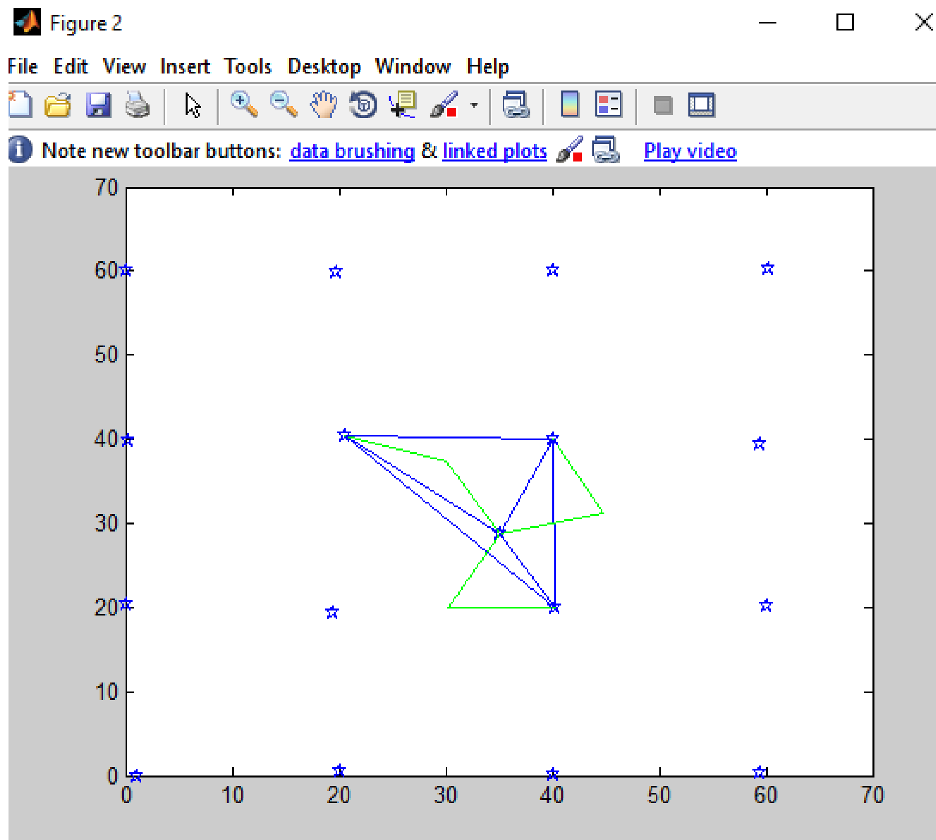

Figure 53: console – result 7

```
593
594 -     totlenght=[sol1,sol2;sol3,sol4;sol5,sol6]
595
```

Figure 54: program code 30

The theoretical length of the links is set up with the variable *"l"*, in our case is 10. But now we have printed the robot and the program can commit errors, mistake or have a little deviation due to the accuracy. So for this reason the program is going to measure the length of the 6 links in the picture with the aim of checking if the representation is accurate enough. The 6 measurements or dimensions are going to be saved in the matrix *"totlength"*. Each line represent one arm, the first term is the shoulder link and the second term is the elbow term, if there are any significant difference we can consider is this error is acceptable (for example in our case less than 0.2). Considering the representation right or accurate enough is up to our criteria.

```
totlenght =

   10.0000   10.0000
   10.0000   10.0000
   10.0050   10.0050
```

Figure 55: console – result 8

In this example we can see that the program has commit an error of 0.005 cm in both links of the third arm, we will consider the representation good enough.

5.1.7 Inverse program

Owing to the fact that the inverse program is an auxiliary program, only is going to be explained how must be used and the basis, not all the code as the other algorithm.

First of all when we run the program an interactive figure appear, in this new window on the top-right corner we can set up the desired position of the holes we are using. If we click in the option *"Potrdi" (confirm)* the holes will be showed in the chart.

In the second step, after having set up the holes, we should click in the button *"Zazeni" (Start)* and now we have the option of selecting a point in the chart which is being the desired centre.

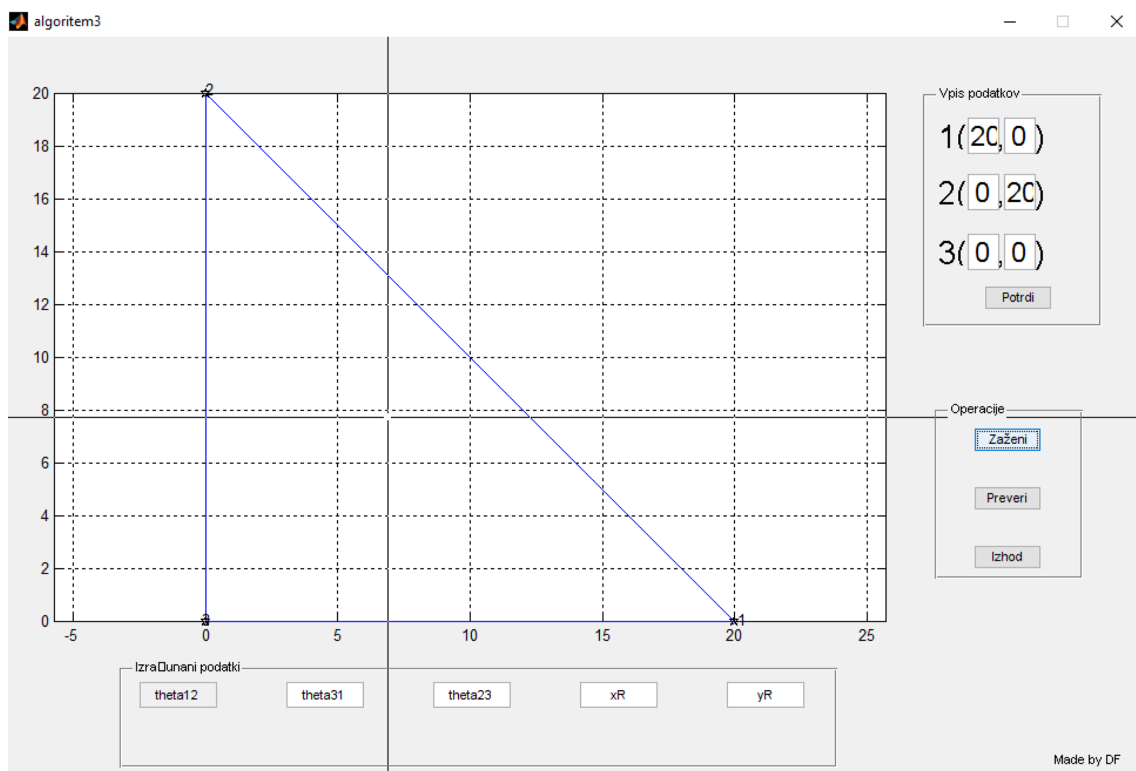

Figure 56: auxiliary program - interactive interface 1

As a result, the program is going to draw the auxiliary lines of the triangles that we use for the future equations. The angles *"theta12","theta23" and "theta31"* are the angles of the centre (1-CENTRE-2, 2-CENTRE-3 and 3-CENTRE-1) and the *"xR" and "yR"* are the coordinates of the centre.

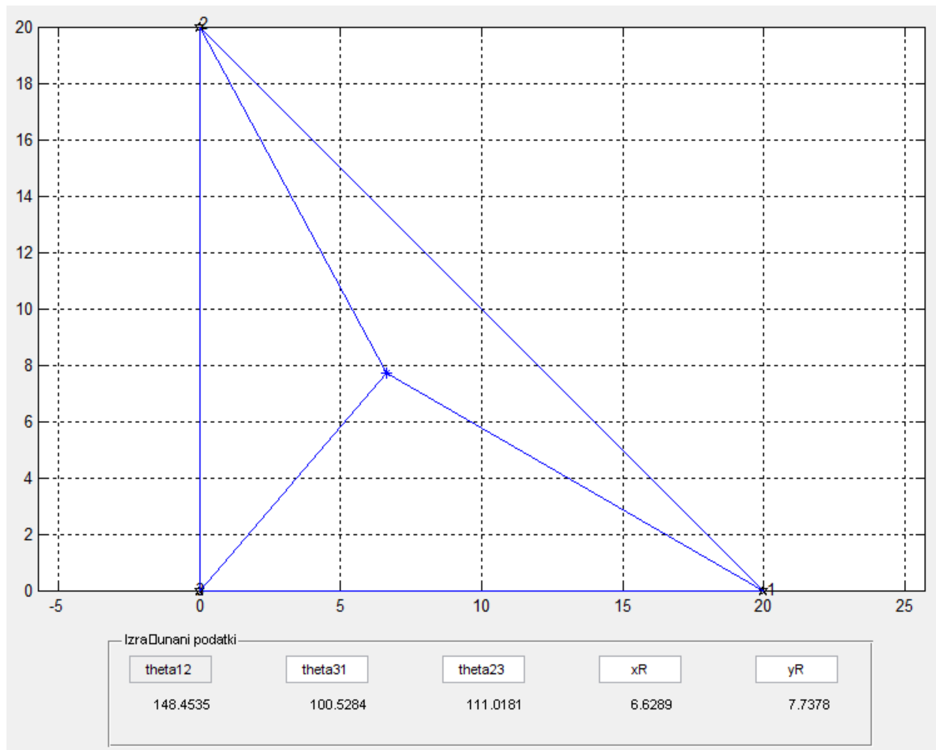

Figure 57: auxiliary program - interactive interface 2

At the same time the program will show a new picture with the configuration of the robot's arms. After using the main program we can compare with the new configuration in order to check if it is right.

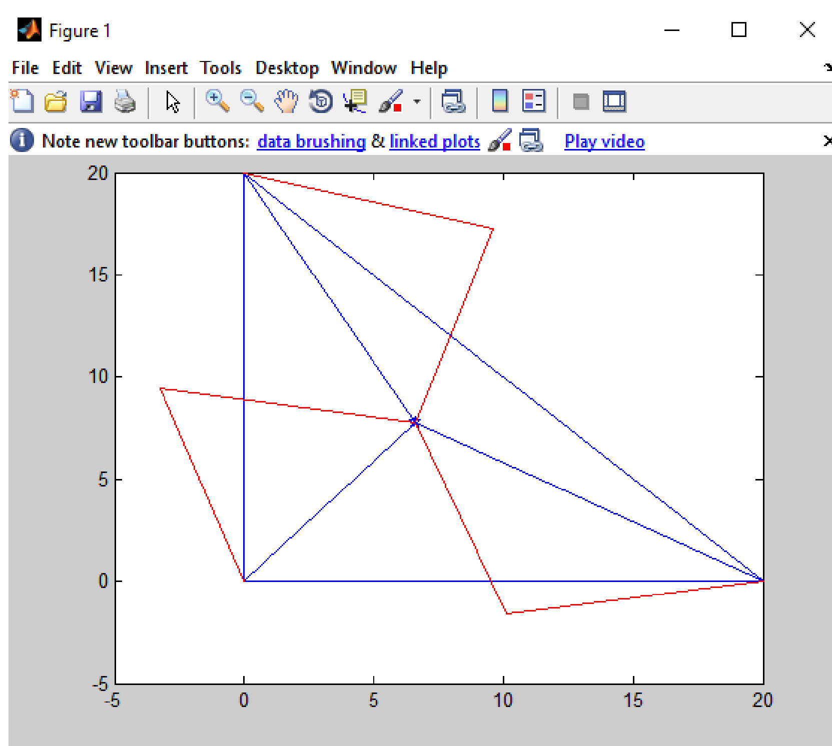

Figure 58: console – result 9

A useful function is to use the Part 1 of the main program after this picture, because the robot's configuration and the map of holes are going to be superimposed, showing the real position of the robot (with this aim we can use break points to make easier the execution, it is not necessary to run the whole program).

The result of the program in the console will be the next

```
l =

    10


cen =

    6.6289    7.7378


p1 =

    20     0


p2 =

     0    20


p3 =

     0     0

angulo1 =

  -69.4851  101.1454


angulo2 =

   72.5795   88.3680


angulo3 =

  170.0409   61.2544
```

Figure 59: console - result 10

The variable *"l"* is the length of the links, and is adjustable in the code of the program (analysed in the next paragraph); *"cen"* is the centre's position;*"p1", "p2" and "p3"* are the 3 points selected as holes; *"angulo1", "angulo2"and "angulo3"* are the angles of the 3 arms, the first term is the shoulder's angle and the second the elbow's angle. We are using this information for checking the next main program and the results of both should be compared with the same result.

The only part of code that we are going to analyse is the following:

```
415      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
416      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
417  -   l=10
418
419  -   cen=[str2double(get(handles.text11,'string')),
420  -   p1=[str2double(get(handles.edit9,'string')),st
421  -   p2=[str2double(get(handles.edit13,'string')),s
422  -   p3=[str2double(get(handles.edit15,'string')),s
423
424  -   long(1)=sqrt((cen(1)-p1(1))^2+(cen(2)-p1(2))^2
425  -   long(2)=sqrt((cen(1)-p2(1))^2+(cen(2)-p2(2))^2
426  -   long(3)=sqrt((cen(1)-p3(1))^2+(cen(2)-p3(2))^2
427
```

Figure 60: program code 31

In line 417 we can set up the length of the links and the next lines (from line 419 on) the program is going to calculate the rest of the results.

As it has been said, we are not going to explain the rest of the code because is an auxiliary program and it will take a lot of time, with the understanding of how can be used is enough for this project.

6. Conclusion

Finally in this project we have developed a program which meets all the requirements and to sum up we are going to analyse how the option selected meets the objectives, compare advantages and disadvantages and think about how it can be improved for future versions.

6.1 Result

As a result of the project we have obtained two programs. The main program is the localization and triangulation of the position for the robot and the other is the auxiliary program to simulate the body and configuration of the robot.

If we continue working, with this beginning, it would be possible configure all the algorithm in the robot's controller. So the last step is to implement this in a real case and see how the final prototype works.

6.2 Advantages and disadvantages

In the following part the advantages and disadvantages are going to be analysed, here we are going to be able to see the highlights, good and bad points and in the next section the decisions that can be taken to improve the design or corrective actions.

6.2.1 Advantages

The main advantages of the program are the next:

- **It works properly**. The principal objective has been met and it is the most important point, to fit the requirements set up at the beginning of this project.
- The **algorithm is divided in different parts**, and all of them are **independent** from each other's. Because of that, it is possible to run only one or more parts by separately, not with the whole program, this facilitates the error's detection and makes it easier to work with the different functions of the algorithm.
- **Variability**. As already mentioned there are variables in both programs (the main program and the auxiliary), that enable us to change information and by adjusting them it is possible to use the same program for different configurations or even different robots (arms angles, surfaces' maps, links' lengths…).
- The **efficiency** of the program is good enough. After the design the next step is to implement the program in a microcontroller of a robot with others functions, so the algorithm may be as simple as possible.
- The **map of holes is easily configured**. At the beginning of the program we can define the grips of the surface that we are going to climb, this surface is easily interchangeable and this lends to the robot versatility.
- **Precision adjustable**. The variable *"precision"* helps us determining the accuracy needed in each surface. That measure will depend on the distances in the map and the length of the robot and is used in order to eliminate wrong solutions or not to be too demanding.

6.2.2 Disadvantages

The main disadvantages are:

- The **success of the program** in the real case **depends on the map and the sensors accuracy**. For this reason the program must be checked in real conditions or emulated with artificial errors, doing that it would be possible to determine the quality of the sensors needed.

- **It is not useful for other prototypes**. It is true that with the versatility of the program's variables, a change in the configuration would be viable, but only with the same structure (3 arms with "spider" structure).

6.3 What could be improved

The final section of this chapter is going to focus on the possibility of strengthening the algorithm's qualities and project's functions.

Develop **equations for movements**. The functions and equations developed in this project are only for a static model of the robot, the program is capable to recognise the position of the robot, the holes in which is griped, and the angles that the arms must have in order to arrive at the desired position. In other words, this means that the algorithm knows where the robot is, which angles have its arms and which angles must have if we want to go to the final position. But the algorithm is not able to predict how the angles must change, it knows the initial and final angle, but not how they must change. The next natural step is to develop the dynamic equations for the robot.

Developing an **interface for interacting** is a good option for improving. This program is the first version for the robot and has an experimental character. For these reasons all the variables are defined in the code. Developing an interface will help the program to be more visual, an example is the way in which the auxiliary program is run, it has charts, texts, options and buttons. Doing something similar with the main program would be of great help for the user.

When the program is not able to find any solution with the provided angles the program continues executing the code but without any point saved in the variable *"p1", "p2" and "p3"* (solution points). Because of that we consider that creating **a message of "error solution doesn't found"** is advisable.

There are a lot of ways or methods that can be used to improve and update the first version of the program but we are going to consider a completely case of success when it would be combined with the other parts of the main project and the final robot works properly. Until this final point the project, the program and the algorithm are subject to **constant change**. If we are able to arrive to this point, the project will be considered completely concluded and finished.