Technical University of Denmark

31388 Advanced Autonomous Robots

Project

Group 20
Yann Boudigou, s191713
Virgile Blanchet-Møhl, s163927
Sélim Kamal, s191729
Sunniva Greta Melkild, s192286

June 1, 2020

Contents

1 Introduction and objectives

After going through all the exercises of the semester, we started to look into the final assignment. It quickly appeared that the assignment would require various knowledge from all the exercises, from the basis of smr-cl programming to using laser plug-ins. We would then need to make sure everything was understood by everyone so we could work on this project and divide the workload in a smart way making people work on the aspects of the project they are most comfortable with. Also, when reading the assignment carefully, we identified two main tasks both composed of different sub-tasks.

The first task would be to recognize the shape and size of an unknown object placed at an unknown position inside a designated area. In this area, an object would be placed that could be either a triangle or a rectangle and of different sizes. We only know for sure is that it would be high enough for the laser to see it. It obviously comes to mind that a good use of the server will be crucial to successfully solve this task. And plug-ins would have to be written to ensure the best detection and analyze.

The second task could be resumed as driving through guidemarks. It is essentially about driving through a known environment, avoiding obstacles and going through optimal routes to navigate between guidemarks, each indicating the number of the next one. We can assume this is a task that would mostly require knowledge about mapping, route planning, localization and smart smr-cl programming. The maze also indicates a certain level of precision would need to be achieved.

We then decided to split the project in two parts before putting them together in the end. This choice was made when we thought the best way to solve the tasks(recognizing the object and driving through the guidemarks) was to first run an object detection algorithm that would make the smr scan the designated area and then a driving algorithm that would make the robot follow the path indicated by the guidemarks. We will then be discussing both parts separately in our report.

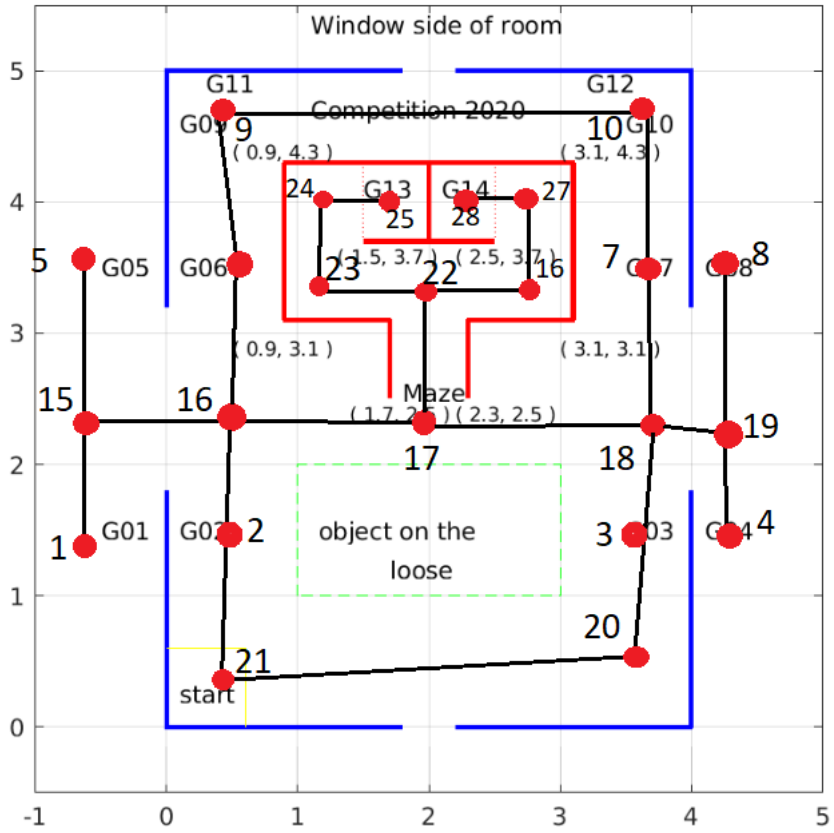
2 Guidemark route

2.1 Concepts used to solve the problem

(Written by Sunniva Melkild - s192286)

2.1.1 Mapping of the maze

In order for the robot to be able to navigate correctly and drive to all the guidemarks we used the graphplanner, which is implemented as a plugin to the laserscanner. The graphplanner was used to create a map which was connected so that the robot could navigate to all the different guidemarks. We started by sketching a map to plan out how the robot would navigate in the maze.



The image shows all the points we created using "addpoint". We made points close to all the guidemarks so that it was easy for the robot to read the guidemark. All the points were given a number as seen in the image. We also added some points for the robot to easily navigate when using the drivew function. For example, if we wanted to go from point 15 to point 2, we would first use the drivew command to get to point 16, and then use the drivew command to get from point 16 to point 2.

In order for the robot to be able to navigate to the guidemarks inside the maze we had to add many points inside the maze to create a route that worked. Then we created connections between the different points using "addcon". All the connections had to be connected in both directions in order for the route to work. For instance, to create point 21 with coordinates(0.25,0.25) and point 2 with coordinates (0.5,1.5), and connect them, we wrote the code:

```
laser "addpoint pno=21 x=0.25 y=0.25"
laser "addpoint pno=2 x=0.5 y=1.5"
laser "addcon pno1=2 pno2=21"
laser "addcon pno1=21 pno2=2"
```

2.1.2 Findroute and drive

After all the points are connected the command "calculatecost" would calculate the cost of the connections. To make the robot compute a route to go from one point to another one in the map, the command "findroute" would find the shortest route which is the route with the lowest cost for the smr to follow to get to the endpoint. For instance, to find the shortest route from the point (0.5,0.5) to (2,1.5) we would use

```
"findroute startx="0.5" starty="0.5" endx="2" endy="1.5""
```

When using findroute, the function returned the number of points N in the route in the variable l4. The x,y and theta coordinate of each point N in the route was returned in $l5, $l6 and $l7 when calling getpoint. For instance the line

```
stringcat "getpoint p="5""
```

would return the x,y and theta coordinates of point number 5 in the route.

The robot would drive to each point using the command "drivew". The drivew command drives along a line defined by a point and an angle. To use drivew you insert the x and y coordinate of the point you want to go to, the angle that the robot is gonna approach the point with and the stopping distance to the line perpendicular to the line defined by the point and the angle. For instance the line

```
drivew 2 1.5 90 :($targetdist <0.1)
```

will drive along the line defined by the point (2,1.5) with a 90 degree angle and stop when the distance is less than 0.1.

For the robot to be able to follow a given route to get to a point in the map using findroute we created a loop containing getpoint and drivew. After the function

findroute had been called, we knew the number of points in the route from reading the variable $l4. We stored this value in the variable n. The loop would use an if-condition where it would run the loop if n is greater than or equal to zero. Findroute would return the route points in an order where the highest number is the first point in the route, and the point 0 is the last point in the route. The loop for driving to get to the end point would then call getpoint p="n" and use the returned x,y and theta values in drivew. The angle has to b converted to degrees before inserting it in drivew. In the last line in the loop we subract 1 from n. The loop would then iterate until point 0 was read and driven to, which is the end position. This gave us the following code for the loop

```
%Number of points in the route
n=$l4

label "start"
stringcat "getpoint p="n""
laser "$string"
wait 1
eval $l5
eval $l6
p1x= $l5
p1y= $l6
p1th=$l7*180/3.1415

drivew p1x p1y p1th :($targetdist <0.1)
n=n-1
if (n>=0) "start"
```

This loop would be called everytime the robot was going to a new guidemark. The coordinates for the guidemark would be given to "findroute" and the loop would iterate until the robot was at the end postition and ready to read a new guidemark.

2.2 Driving between guidemarks

(Written by Selim Kamal - s191729)

Once the "world" was mapped with carefully chosen points, the main concepts and functions understood and the task visualised, we finally had to make a simple but thorough smr-cl loop that will implement the discussed function and solve the given task of driving through the guidmarks.
The first thing we did was to convert the task to a simple suite of basic actions :

```
1) Go to the first given guidemark
2) Then read the number contained in the guidmark
3) Go to the guidemark with this number
4) Repeat until the number 98 is found
5) Then go back to the starting position
```

We then started to think how can each action could actually be done.
- For the first one, we would need the position of the robots and the position of the first guidmark, then find a route between them and then follow it.
- For the second one, we will need to be able to read the value stored in a guidemark.
- The third one comes down to the same things as the first one but with a different target.
- The fourth strongly suggest a loop for action 3.
- And the fifth would be like the third but with an imaginary guidemark at the starting position.

Writing down these observations, we came to the conclusion that action number 3 would be the key part of our main loop with the other actions being reduced to a change of parameters except for reading the guidmarks.
We then came with the following algorithm that would deal with all four possible cases : Going to the first guidemark, going to the assigned guidemark, going back to starting position if number 98 is found and ending if -1 is read(problem when reading a guidemark or the robot is back to starting position and there's no guidemarks to be seen)

```
Define the first target
label "start"

Go to the target

Get the value from the guidemark

If the target is equal to -1
    End the program
Else if the target is equal to 98
```

```
    Set target to starting coordinates
Go to label start
```

We then had our main loop algorithm but a few challenges were left to be solved. How to face the guidemarks and read them ? In fact, once the robot drove to the coordinates of the guidemark, it still needs to face it and read it. We decided to make it face by calculating the difference between the robot's real angle in the world coordinates and the one needed to read the guidemark. And then the guidmark would be read by storing the value currently viewed by the smr and stored in the fiducialid variable.
All this finally made us come up with this pseudo-code:

```
Define the first target

label "start"

//Find a route to the target
Get target's coordinates
Get robot's coordinates
Find a route between from robot's position to target's position

//Follow the route(inner loop)
For each point in the route :
    Get the point's coordinates
    Drive to these coordinates

//Face the guidmark
Get the robot's angle
Get the angle needed to read the guidemark
Calculate the difference between both
Turn the robot of the calculated value


//Read the guidmark and set new target accordingly
Wait for the robot to stabilize
Store the currently viewed value($fiducialid as the new target

//Close the loop by making the robot park or go to next target
If the target is equal to -1
    End the program
Else if the target is equal to 98
    Set target to starting coordinates
Go to label start
```

Once this was done, the only task left was to write it in smr-cl. When this was also done, we made some tests and discovered some errors that led us to make some changes. First we had troubles finding the robot's position in world coordinates.
We tried a first approach using odometry that was giving some results but because of its relativity to the start position and tendency to miscalculations, it was leading to some errors and a non-satisfying precision. We then tried a second approach using this smr-cl code :

```
invtrans $l0 $l1 $l2 $odox $odoy $odoth
x1 = $res0
y1 = $res1
angle = $res2
```

This approach was more successful and the result satisfied us enough so it is the code part we are using in the final code.

A second problem we encountered was running the findroute and getpoint functions with variables instead of numbers. In fact, the smr-cl line

```
laser "findroute startx="x1" starty="y1" endx="x2" endy="y2""
```

doesn't work and we then had to find a solution. It appeared the solution was to first store a string in the buffer converting the variables into numbers and then to output this string to the laser server who could then read it. The code would look like that :

```
stringcat "findroute startx="x1" starty="y1" endx="x2" endy="y2""
laser "$string"
```

The next problem we had to solve was concerning the driving. We indeed used the drivew command with the next point's coordinates

```
drivew x y th :($targetdist <0.1)
```

but it appeared the drone had "too large" movements making it encounter walls :

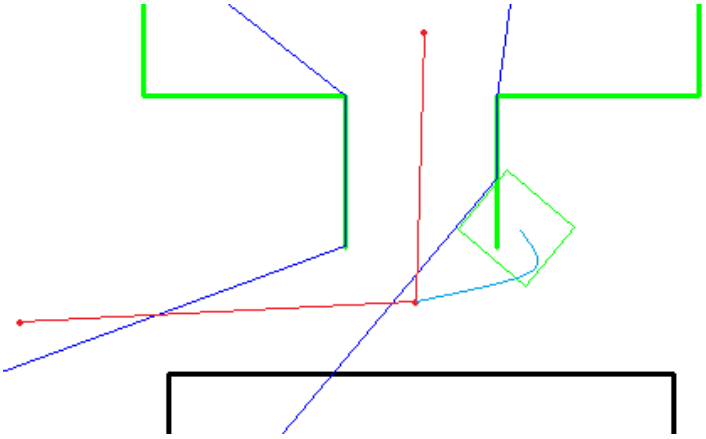


Figure 1: Desired path in red and real path in blue

After analyzing the situation, we understood the problem was actually that the robot was turning and moving at the same time causing these weird movements and lack of precision. This was due to previous drivew commands running at the same time at new ones. The problem was solved using a simple stop command following the drivew command. when the targetdist was found to be inferior to 0.1 the robot would then stop and when it would be time to drive again it would start over :

```
drivew p1x p1y p1th :($targetdist <0.1)
stop
```

This modification made the robot driving way more precise as wished but also made it slower as a side effect. Time not being our priority, we kept this modification.

The last problem we had with the drivew command appeared when we tried to explore the central part of the maze and the drone got stuck for no apparent reasons.

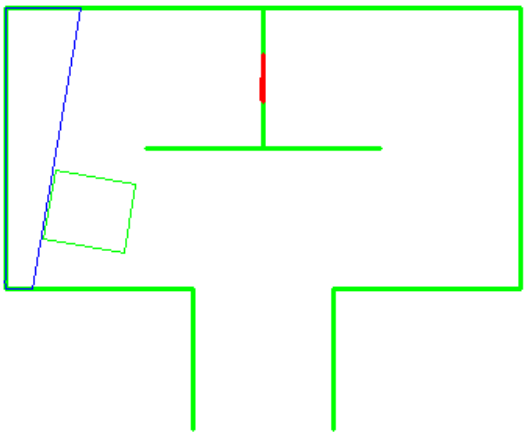


Figure 2: Robot stuck in the maze

After reviewing different possibilities we understood the robot's sensors were prohibiting it to move due to the very close presence of walls(obstacles). We simply solve this problem by adding an a command to ignore obstacles while driving and trusting our choice of points to prevent any collision.
The final code was then :

```
ignoreobstacles
drivew p1x p1y p1th :($targetdist <0.1)
stop
```

3 Object recognition

3.1 Plugin

3.1.1 Function detect

(Written by Virgile Blanchet-Møhl - s163927)

In order to determine which object is being scanned, the data from the laser scans must first be processed before being able to be used for object recognition. The detect function was written to this end and will be described in the following section.
The detect function is called to extract data from the laser scans of the object, said laser scans output the data in polar coordinates and within the reference system of the laser position at the time of the scan, as can be observed from a scan as seen in Figure 3. The data must as such first be converted into Cartesian coordinates and then into world coordinates using the current position and orientation of the laser, or in other words the robot pose.

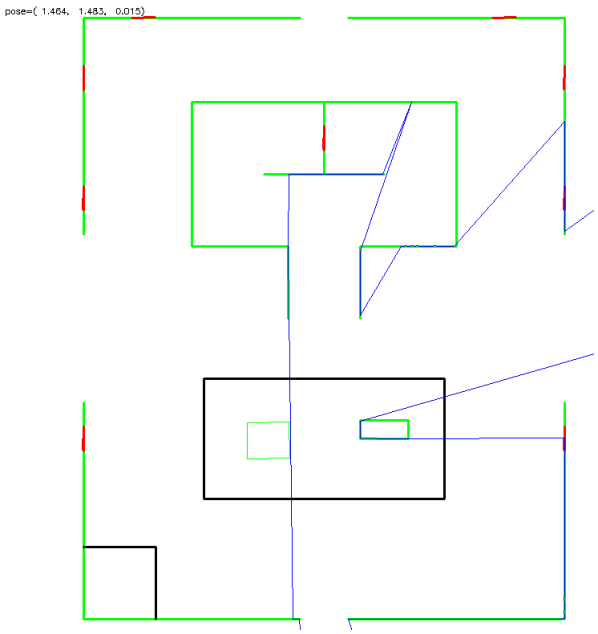


Figure 3: Laser and world reference systems illustrated

The detect functions uses the current pose of the robot in world coordinates in order to change reference systems, these are extracted using the modified aulocalize plugin, which prints the position in a text file.
The following code serves to convert the data from scan data into usable world coordinates.

```
int dataI;
for (i = 0, dataI = 0; i < data->getRangeCnt(); i++) { // range are stored as an integer in current units
  bool rangeValid;
  double r = data->getRangeMeter(i, &rangeValid);
  if (!rangeValid || r < minRange)
    continue;
  double alpha = data->getAngleRad(i);

  double xr = r * cos(alpha);
  double yr = r * sin(alpha);
  double xw = pose_X + LSRPOSE*cos(pose_theta) + (xr-yr*tan(pose_theta))*cos(pose_theta);
  double yw = pose_Y + LSRPOSE*sin(pose_theta) + (yr+xr*tan(pose_theta))*cos(pose_theta);

  if(!(xw<1-EPSILON || xw>3+EPSILON || yw>2+EPSILON || yw<1-EPSILON))
  {
            X[dataI] = xr;
            Y[dataI] = yr;
            dataI++;
  }
}
```

The last section of the above code serves to check the world coordinates of each point before proceeding, only considering any scan points of which the world coordinates are found within the bounds of the marked area within which the loose object can be found. This also serves to exclude any points of the maze and walls the laser scan may happen to observe. A margin of error ϵ is used to ensure points on the edge of the area are not excluded due to observational error.
The extracted points, in laser coordinates, that remain in consideration are then put through a ransac function which outputs an estimate of each observed line using the X and Y values, defined in lines 16 and 17 of the above code respectively as the bounded laser scan coordinates, given to the function as an α and r value in laser coordinates.
The following code takes the laser coordinates line values produced by the ransac function and converts them to world coordinates before confining the *alpha* of each line to an interval of $]-\pi, \pi]$ and taking the absolute value of the radius. Each of the lines α and r are then committed to a .txt file for use in the compute function described in the next section 3.1.2 once all scans have been made.

```
for (itLas = GFLL.begin(); itLas != GFLL.end(); itLas++) {
              LEL_ARLine line = (*itLas).toARLine();
              line.r = line.r + LSRPOSE*cos(line.alpha);
              line.alpha = line.alpha + pose_theta;
              line.r = line.r + pose_X*cos(line.alpha) + pose_Y*sin(line.alpha);

              if(line.r < 0)
              {
                      line.r = -line.r;
                      if(line.alpha > 0)
```

```
                                line.alpha -= M_PI;
                        else
                                line.alpha += M_PI;
                }

                myFlux_lines << line.r << " " << line.alpha << endl;
              }
```

3.1.2 Function compute

(Written by Yann Boudigou - s191713)

Since we collected all the data of the extracted lines in the extracted_lines.txt file, we needed now to compute this in order to identify the object. This is done by calling the function compute, whose operation will be explained in this section.

In order to make to that easier we first created new structures to process data.

```
typedef struct Line Line;
struct Line
{
        double r;
        double alpha;
};

// contains the carthesian equation of a line (ax+by+c=0)
typedef struct Eq_line Eq_line;
struct Eq_line
{
        double a;
        double b;
        double c;
};

typedef struct Point Point;
struct Point
{
        double x;
        double y;
};
```

We created a vector of lines that we filled with the extracted_lines.txt. Due to the large number of scans, sometimes the robot sees the same line twice. That's why we implemented a for loop to remove the redundant lines. When the program sees twice the same line within one epsilon, it computes the average of the two, re-write it on the first one, and delete the second one.

```
// remove redundant lines
for(it1=lines_obj.begin(); it1<lines_obj.end()-1; ++it1)
{
        for(it2=it1+1; it2<lines_obj.end(); ++it2)
        {
                if(abs(it1->r-it2->r)<EPSILON_R && abs(it1->alpha-it2->alpha)<EPSILON_ALPHA)
                {
                        it1->r = (it1->r+it2->r)/2;
                        it1->alpha = (it1->alpha+it2->alpha)/2;
                        lines_obj.erase(it2);
                        --it2;
                }
        }
}
```

Now that we had the lines of the object, we needed to compute :

- the shape of the object
- which object of the list it corresponds too
- the orientation of the object
- the origin of the object

In order to do that, we decided to compute the intersection points of the lines so that we have the vertices of the object. And with the vertices we were able to find all the information we needed on the object. As the polar representation (r, α) was not the easiest one to process, we converted it into a classic Cartesian equation (ax+by+c=0).

```
1  // compute carthesian equation of the lines
2  for(it1=lines_obj.begin(); it1!=lines_obj.end(); ++it1)
3  {
4          myFlux_results << "object line (r, alpha) : " << it1->r << " " << it1->alpha << endl;
5          double vx = (it1->r)*cos(it1->alpha-(M_PI/2));
6          double vy = (it1->r)*sin(it1->alpha-(M_PI/2));
7          double Ax = (it1->r)*cos(it1->alpha);
8          double Ay = (it1->r)*sin(it1->alpha);
9          eq_line.a = vy;
10         eq_line.b = -vx;
11         eq_line.c = -Ax*vy+Ay*vx;
12         eq_lines_obj.push_back(eq_line);
13 }
```

vx and vy are the components of the vector v, a vector collinear to the line. Ax and Ay are the coordinates of a point A, which is part of the line. You might notice that we print the extracted lines of the object in the myFlux_results (at line 4), this is part of the result file, which will be detailed afterwards. Once we get the Cartesian equations of the lines it was easy to compute the intersection points.

```
// compute intersection points
for(it_eq1=eq_lines_obj.begin(); it_eq1<eq_lines_obj.end()-1; ++it_eq1)
{
        for(it_eq2=it_eq1+1; it_eq2<eq_lines_obj.end(); ++it_eq2)
        {
                //going from ax+by+c=0 to y=ax+b
                double a1 = -(it_eq1->a)/(it_eq1->b);
                double b1 = -(it_eq1->c)/(it_eq1->b);
                double a2 = -(it_eq2->a)/(it_eq2->b);
                double b2 = -(it_eq2->c)/(it_eq2->b);

                if(abs(a2-a1)>EPSILON_PARALLEL) //non-parallel lines
                {
                        point.x = (b2-b1)/(a1-a2);
                        point.y = a1*(point.x)+b1;
                        if(!(point.x<1-EPSILON || point.x>3+EPSILON || point.y>2+EPSILON ||
                           point.y<1-EPSILON)) //intersection point in the rectangle of detection
                        {
                                intersec_point.push_back(point);
                                nb_intersec_points++;
                                myFlux_results << "intersection point (x, y) : " << point.x << " " <<
                                point.y << endl;
                        }
                }
        }
}
```

We first check if the two lines we're processing are parallel (if so there's no intersection point of course). But this condition was not enough. Indeed, as we were looking at the difference between the two guiding coefficients a of the line, when these lines were vertical, the two coefficients a were almost infinite so even a slight difference between them would be enough to make the program think these lines are not parallel. That's why we added the condition that the intersection point needs to be in the rectangle of detection to be valid.

Next step was to identify the object. Differentiate the triangles and the rectangles was easy because it just depends on the number of intersection points. To identify which rectangle or which triangle it was, we needed the lengths of the sides. But there's only two lengths that define the object, and we computed 6 distances between points for the square and 3 for the triangle. So we needed to ignore some of them in order to keep only the two ones that were interesting us. We first decided to order them by size, the longest ones in first. We used a priority_queue in order to do that. Then, we knew that mathematically, the biggest distances would correspond to the diagonal of either the square of the rectangle so we could get rid of the first element (the two first for the square). Finally we just needed to ignore one element every two elements for the square, and we ended up with only the two lengths of the sides of interest (saved in the long_side and short_side variables).

```
// compute the distance between the points
priority_queue<double> dist;
double max_distance = 0;
double min_distance = 1;
double distance;
pair<Point, Point> farest_points, closest_points;

for(unsigned int i =0; i<intersec_point.size()-1; ++i)
{
  for(unsigned int j=i+1; j<intersec_point.size(); ++j)
  {
    distance = sqrt(pow((intersec_point[i].x-intersec_point[j].x),2)+
                    pow((intersec_point[i].y-intersec_point[j].y),2));
    dist.push(distance);

    if(distance>max_distance)
    {
      max_distance = distance;
      farest_points.first = intersec_point[i];
      farest_points.second = intersec_point[j];
    }

    if(distance<min_distance)
    {
      min_distance = distance;
      closest_points.first = intersec_point[i];
      closest_points.second = intersec_point[j];
    }
  }
}
```

We get rid of the extra distances with these lines of code :

```
if(nb_intersec_points == 3)
{
        myFlux_results << "It's a triangle" << endl;
        dist.pop();
        long_side = dist.top();
        dist.pop();
        short_side = dist.top();
        myFlux_results << long_side << " " << short_side << endl;
```

```
else if(nb_intersec_points == 4)
{
        myFlux_results << "It's a rectangle" << endl;
        dist.pop();
        dist.pop();
        long_side = dist.top();
        dist.pop();
        dist.pop();
```

```
        short_side = dist.top();
        myFlux_results << long_side << " " << short_side << endl;
```

As you can see, we also computed the two closest and furthest points of the shape. This is because we needed some landmarks to know which intersection point correspond to which vertice of the shape. For the triangle, we found that the origin is the point which is not among the furthest points (as shown in Figure 4), and we computed the orientation by finding the vector between the origin and the "point of interest" and shifting its angle by -$\pi/2$.

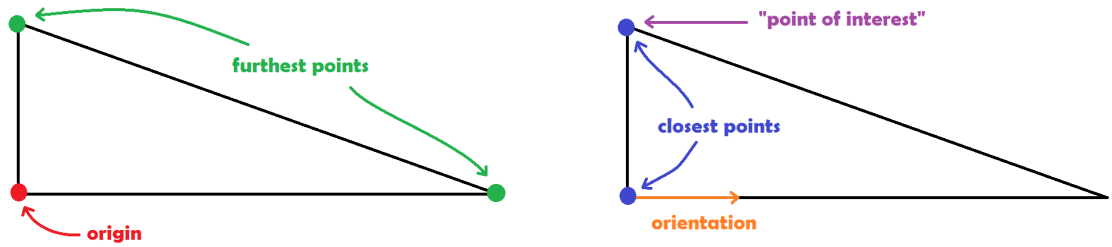


Figure 4: Correspondence between names and points in the triangle

```
for(unsigned int i=0; i<intersec_point.size(); ++i)
{
        // in the triangle the origin point is the one not among the farest points
        if(intersec_point[i].x != farest_points.first.x && intersec_point[i].y
           != farest_points.first.y && intersec_point[i].x != farest_points.second.x &&
           intersec_point[i].y != farest_points.second.y)
        {
                origin.x = intersec_point[i].x;
                origin.y = intersec_point[i].y;
        }

        // we focus on the point which belongs to the shortest and longest side of the triangle
        if(((intersec_point[i].x == farest_points.first.x && intersec_point[i].y ==
             farest_points.first.y) || (intersec_point[i].x == farest_points.second.x &&
             intersec_point[i].y == farest_points.second.y)) && ((intersec_point[i].x ==
             closest_points.first.x && intersec_point[i].y == closest_points.first.y) ||
             (intersec_point[i].x == closest_points.second.x && intersec_point[i].y ==
             closest_points.second.y)))
        {
                point_of_interest.x = intersec_point[i].x;
                point_of_interest.y = intersec_point[i].y;
        }
}
myFlux_results << "Origin (x, y) : " << origin.x << " " << origin.y << endl;

double vx = point_of_interest.x-origin.x;
double vy = point_of_interest.y-origin.y;
```

```
28 orientation = -atan(vx/vy); //the two pi/2 cancel each other
29
30 if(orientation > M_PI)
31  orientation -= M_PI;
32 if(orientation < -M_PI)
33  orientation += M_PI;
34
35 myFlux_results << "Orientation (in rad) = " << orientation << " +2*k*pi" << endl;
```

Same goes for the rectangle, but this time we computed the origin by doing an average of the coordinates of the furthest points. To compute the orientation we calculated the angle of the vector connecting the closest points and shifted it by $\pi/2$. It is important to note that there is two possibilities for the furthest and closest points shown in Figure 5. This doesn't change either the origin or orientation. Furthermore, the orientation of the rectangle is defined with a $\pm k\pi$ precision and not $\pm 2k\pi$ like the triangle because of the symmetry.

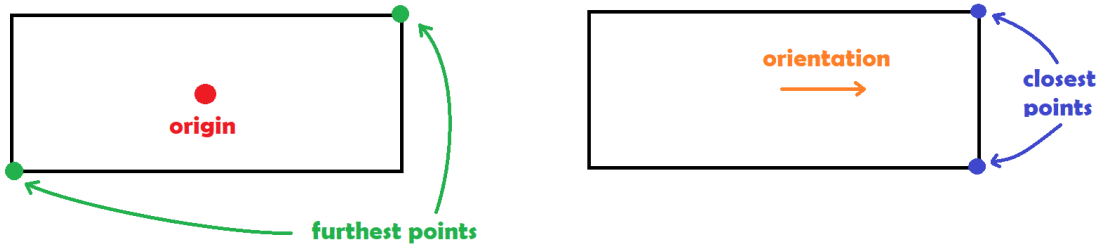


Figure 5: Correspondence between names and points in the rectangle

```
1  origin.x = (farest_points.first.x+farest_points.second.x)/2;
2  origin.y = (farest_points.first.y+farest_points.second.y)/2;
3
4  myFlux_results << "Origin (x, y) : " << origin.x << " " << origin.y << endl;
5
6  double vx = closest_points.first.x-closest_points.second.x;
7  double vy = closest_points.first.y-closest_points.second.y;
8  //cout << vx << " " << vy << endl;
9  orientation = -atan(vx/vy); //the two pi/2 cancel each other
10
11 if(orientation > M_PI)
12  orientation -= M_PI;
13 if(orientation < -M_PI)
14  orientation += M_PI;
15
16 myFlux_results << "Orientation (in rad) = " << orientation << " +k*pi" << endl;
```

To print the results we first used a simple cout, but we quickly realized that the ulmsserver terminal was already overloaded with the call of the function localize every

second. So to display the results in a clear way we decided to write it in a file called "results.txt" as shown below :

```
object line (r, alpha) : 1.51228 1.56697
object line (r, alpha) : 2.69344 -0.00152687
object line (r, alpha) : 1.6415 1.57144
object line (r, alpha) : 2.30364 0.000192948
intersection point (x, y) : 2.69574 1.50198
intersection point (x, y) : 2.30335 1.50348
intersection point (x, y) : 2.69595 1.64324
intersection point (x, y) : 2.30332 1.64299
nb_intersec_points 4
It's a rectangle
0.392629 0.141255
Object 1
Origin (x, y) : 2.49953 1.57249
Orientation (in rad) = 0.000192948 +k*pi
```

In this file we display (in this order) :

- the extracted lines of the object (in polar coordinates)
- the intersection points computed
- the number of intersection points
- the shape (triangle or rectangle)
- the lengths of the sides
- the corresponding object
- the origin of the object
- the orientation of the object

After we finished to code our plugin, we needed to add it in the overall project, and that's done in the SMR-CL implementation.

3.2 Implementation in SMR-CL code

(Written by Virgile Blanchet-Møhl - s163927)

With the SMR starting the course at a known location the object recognition process is set to occur at the start of the run through the course. Given that multiple points of view are required in order to determine the shape of the object the implementation

calls the detect function from a multitude of angles throughout a drive around the bounds within which the loose object is confined, in order to observe each line of the object. To ensure the scans do not miss the object, and to ensure the scans accurately capture the lines of the object, the robot calls the detect function three times across each face of the rectangular area. A balance in the number of scans needs to be struck, as an excess of scans can lead to additional false positives and reduce the overall accuracy of the model, in the event of the scanner seeing nonexistent lines. This issue can be reduced by adding a redundancy check that only forwards lines that appear in multiple scans, this was however not integrated in our project due to time constraints. Due to the rectangular 2 by 1m meters shape of the area, the SMR advances through the area when scanning from either the left or right sides of the are in order for the scan to be more accurate and avoid calling the detect function too far from the object. The implementation of the approach to the object can be found below.

```
before = $ododist
drive :($irdistfrontleft<distobj)|($irdistfrontmiddle<distobj)|
       ($irdistfrontright<distobj)|($drivendist>maxdist)
stop
drivedist = -$ododist+before
if(abs(drivedist)>=maxdist) "jump1"
wait 1.5
laser"detect"
label"jump1"
```

The IR sensor is used to check whether the SMR is approaching the object to avoid interfering with the laser scans, as it can only scan in front of the SMR, the robot must advance along the area three times in order to cover the entire zone. Calling the detect function if the traveled distant is less than the maximum distance to be traveled, which means that the IR scanner has detected the object, if the full distance is traveled however, the detect function call is skipped as that would mean the SMR did not encounter the object. The approach improves the accuracy of each scan and thus the estimate of each line. The SMR travels counterclockwise around the area scanning it as described using the detect function, after all the scans have been concluded the compute function is called in order to identify the object observed by the scans.

4 Conclusion

This project was challenging learning experience and taught us a lot on robot localization, on how to move in a known environment, as well as the basics of object recognition.

This last point has been the one we struggled the most on. The scanning of the robot is not an exact science and the RANSAC algorithm is not infallible : in the beginning we quite often ended up by seeing imaginary lines. We went through the process of improving the way we performed scans as described in the section dedicated to the SMR-CL implementation of the plugin. And even though our code improved a lot throughout the different iterations, we still have troubles to recognize some shapes in some configurations.

However, completing this project was not only challenging because of its difficulty, but also because of the situation caused by the coronavirus. Indeed, we had to find new ways of working together without actually meeting in real life. But in the end, we used online tools and managed to perform the tasks asked. This oddly helped us develop communication and group work skill by working in a more challenging environment and reduced access to each other. It pushed us to be even more efficient in our communications as well as in our workflow so the project would still advance as planned difficulties imposed by the current circumstances.

5 Appendix

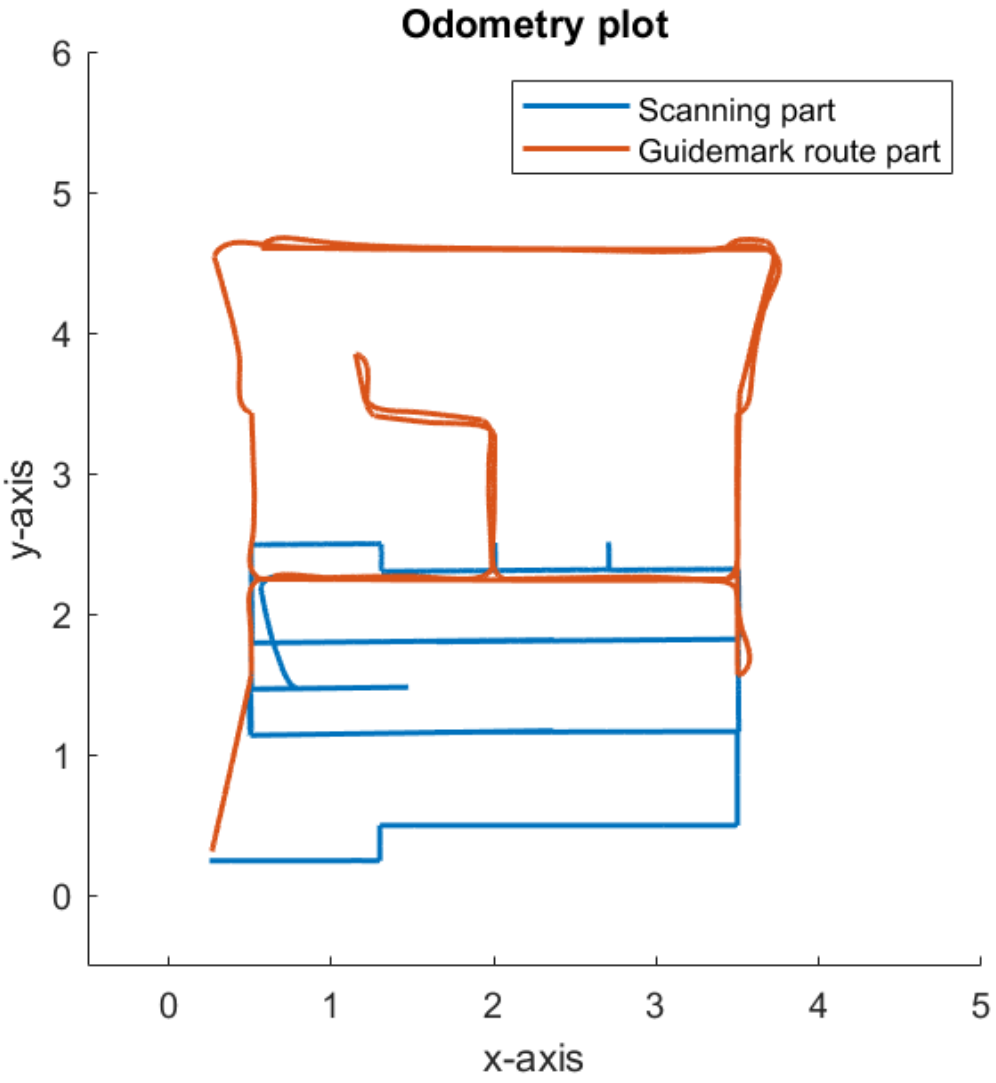


Figure 6: XY plot of the robots route solving the problem